

GS-Scale: Unlocking Large-Scale 3D Gaussian Splatting Training via Host Offloading

Donghyun Lee¹ Dawoon Jeong¹ Jae W. Lee¹ Hongil Yoon^{1,2}

¹Seoul National University ²Google

{eudh1206, daun20211, jaewlee}@snu.ac.kr, hongilyoon@google.com

Abstract

The advent of 3D Gaussian Splatting has revolutionized graphics rendering by delivering high visual quality and fast rendering speeds. However, training large-scale scenes at high quality remains challenging due to the substantial memory demands required to store parameters, gradients, and optimizer states, which can quickly overwhelm GPU memory. To address these limitations, we propose GS-Scale, a fast and memory-efficient training system for 3D Gaussian Splatting. GS-Scale stores all Gaussians in host memory, transferring only a subset to the GPU on demand for each forward and backward pass. While this dramatically reduces GPU memory usage, it requires frustum culling and optimizer updates to be executed on the CPU, introducing slowdowns due to CPU's limited compute and memory bandwidth. To mitigate this, GS-Scale employs three system-level optimizations: (1) *selective offloading* of geometric parameters for fast frustum culling, (2) *parameter forwarding* to pipeline CPU optimizer updates with GPU computation, and (3) *deferred optimizer update* to minimize unnecessary memory accesses for Gaussians with zero gradients. Our extensive evaluations on large-scale datasets demonstrate that GS-Scale significantly lowers GPU memory demands by 3.3-5.6 \times , while achieving training speeds comparable to GPU without host offloading. This enables large-scale 3D Gaussian Splatting training on consumer-grade GPUs; for instance, GS-Scale can scale the number of Gaussians from 4 million to 18 million on an RTX 4070 Mobile GPU, leading to 23-35% LPIPS (learned perceptual image patch similarity) improvement.

1 Introduction

Differentiable rendering methods [22, 37, 38, 51] have significantly improved the quality and efficiency of novel view synthesis. Among these innovations, 3D Gaussian Splatting (3DGS) [22] has emerged as a state-of-the-art technique, offering high visual quality and fast rendering by representing a 3D scene with millions of trainable 3D Gaussian primitives.

However, the increasing demand for reconstructing larger and more visually detailed 3D scenes has led to a significant surge in the number of Gaussians required during training [26, 31, 33, 34, 44, 53], pushing the limit of GPU memory. For example, in Rubble [46] scene, reaching the highest visual quality requires about 40 million Gaussians resulting in 53 GB of GPU memory, far exceeding the capacity of any

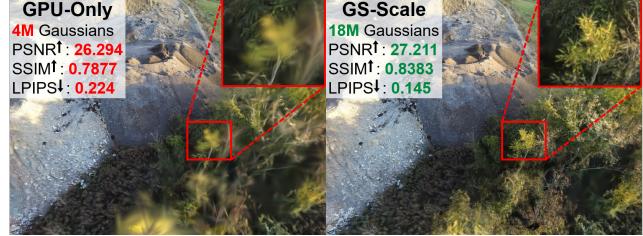


Figure 1. Comparison of the maximum rendering quality achievable in 3DGS training using a GPU-only system and GS-Scale. Training is conducted on RTX 4070 Mobile GPU with Rubble scene. Higher is better for PSNR and SSIM, lower is better for LPIPS.

single consumer-grade GPU. These high memory demands present a major obstacle to scaling the number of Gaussians in 3DGS training, leading to reduced scene expressiveness and, consequently, degraded rendering quality.

Recent works [26, 31, 53] have addressed these challenges through distributed training across multiple GPUs. Such multi-GPU setups entail high hardware costs and considerable maintenance complexity, making them impractical for most users. This limitation is particularly critical in personal or small-scale professional settings, where 3DGS is often applied to reconstruct scenes from user-provided images, such as VR hobbyists modeling personal spaces [2, 3, 6], interior designers designing 3D virtual rooms [1, 42], and real estate professionals supporting 3D virtual tours [5]. Thus, enabling high-quality 3DGS training on a single consumer-grade GPU is essential for accessible deployment.

We present **GS-Scale**, a fast, memory-efficient, and scalable 3D Gaussian Splatting training system built upon host (CPU) offloading. Our key observation is that in each training iteration, only a small subset of Gaussian parameters participates in forward and backward passes. Leveraging this property, GS-Scale stores all Gaussian parameters and optimizer states in host memory, transferring only the necessary subset to the GPU on demand. While this approach dramatically reduces GPU memory usage, it forces computationally intensive frustum culling and memory intensive optimizer updates onto the CPU, leading to significant slowdowns due to limited compute power and memory bandwidth of CPU. To address these challenges, GS-Scale incorporates three system-level optimizations:

- *Selective Offloading*: Only geometric attributes of parameters are kept on GPU for fast frustum culling, while the rest are offloaded to host memory.
- *Parameter Forwarding*: By pre-updating only necessary parameters, this optimization breaks the dependency between CPU optimizer updates and GPU forward & backward passes, enabling pipelining.
- *Deferred Optimizer Update*: By deferring updates for Gaussians with zero gradients, the amount of memory accesses is substantially reduced while achieving identical training results.

Through extensive evaluations across various datasets and platforms, we demonstrate that GS-Scale can train much larger scenes on consumer-grade GPUs while maintaining training speeds comparable to GPU without host offloading. For example, GS-Scale can scale the number of Gaussians from 4 million to 18 million on an RTX 4070 Mobile GPU, yielding a 35.3% improvement in LPIPS for the Rubble scene (Figure 1).

Our contributions are summarized as follows.

- We empirically observe a sparse workload characteristic: during each training iteration, only a small subset of Gaussian parameters is involved in the forward and backward passes.
- We analyze GPU memory bottlenecks in 3DGS training and identify host-offloading opportunities based on this sparse workload characteristics.
- We propose GS-Scale, a fast, memory efficient, and scalable training system for 3DGS. To the best of our knowledge, GS-Scale is the first host offloading based training system for 3DGS.
- We implement GS-Scale on top of gsplat [49] library and comprehensively evaluate the performance on various datasets and GPU platforms. GS-Scale demonstrates substantial GPU memory savings and comparable training speed with GPU without host offloading, unlocking large-scale 3DGS training.

2 Background

2.1 Novel View Synthesis

Novel view synthesis generates photorealistic 3D scene images from previously unseen viewpoints using a set of 2D images captured from multiple viewpoints. It has broad applications in diverse fields such as virtual reality (VR) [9, 15, 21], augmented reality (AR) [8, 52], and digital twins [17]. Traditional explicit 3D reconstruction methods using meshes, voxels, or point clouds often struggle with complex visual phenomena and suffer quality degradation from incomplete or inaccurate reconstructions. Differentiable rendering methods have recently emerged, offering substantial improvements in reconstruction fidelity and rendering quality. The following sections describe them.

2.2 Neural Radiance Fields (NeRF)

Neural Radiance Fields (NeRF) [37] has revolutionized the field of novel view synthesis by adopting an implicit neural representation of 3D scenes as continuous volumetric functions, overcoming limitations of explicit 3D methods. NeRF represents each 3D point with 5D coordinates (x, y, z, θ, ϕ) , where (x, y, z) denotes the 3D position and (θ, ϕ) represents the viewing direction. Multi-Layer Perceptrons (MLPs) map these coordinates to a volume density σ and a view-dependent color c , simultaneously modeling both the geometric structure and appearance of the scene. Rendering involves casting rays from the camera into the scene, taking multiple sample points along each ray, predicting the color c_i and density σ for each point with the MLP, and accumulating them to compute the final pixel color via a classical volume rendering. Training optimizes MLP-based 3D scene representations by minimizing the difference between the rendered images and the corresponding ground-truth images.

NeRF and its variants [7, 16, 38, 46] has achieved a significant breakthrough in novel view synthesis, demonstrating superior quality. However, its dependence on MLP leads to high computational cost in both training and rendering, limiting its deployment in latency-sensitive applications.

2.3 3D Gaussian Splatting

3D Gaussian Splatting [22] is state-of-the-art differentiable rendering method representing 3D scenes with trainable 3D Gaussian primitives. Each 3D Gaussian has 59 parameters: center position $mean \in \mathbb{R}^3$, $scale \in \mathbb{R}^3$ controlling its spatial extent, rotation represented by $quaternion \in \mathbb{R}^4$, $opacity$ that determines transparency, and *spherical harmonics* (SH) coefficients which encodes view-dependent color. The SH models how a point's color changes with viewing direction; A common degree of $L = 3$ yields 16 coefficients per color channel and 48 parameters in total for RGB.

3DGS renders images by projecting 3D Gaussians onto a 2D plane and accumulating colors in depth order. The training process minimizes the difference between rendered images and ground-truth images via backpropagation (refer to Section 2.4). Unlike NeRF, 3DGS uses an explicit representation, eliminating the need for MLP computation during rendering and training, leading to faster speeds and higher visual quality. However, this explicit representation significantly increases the number of required parameters, leading to a much higher memory footprint for both rendering and training. The number of Gaussians directly determines the parameter count. More Gaussians are necessary for higher rendering quality, increasing memory pressure.

2.4 Training Pipeline of 3D Gaussian Splatting

The 3D Gaussian Splatting pipeline, illustrated in Figure 2, begins with 3D Gaussians that are initialized based on a 3D point cloud obtained from Structure-from-Motion (SfM) [43].

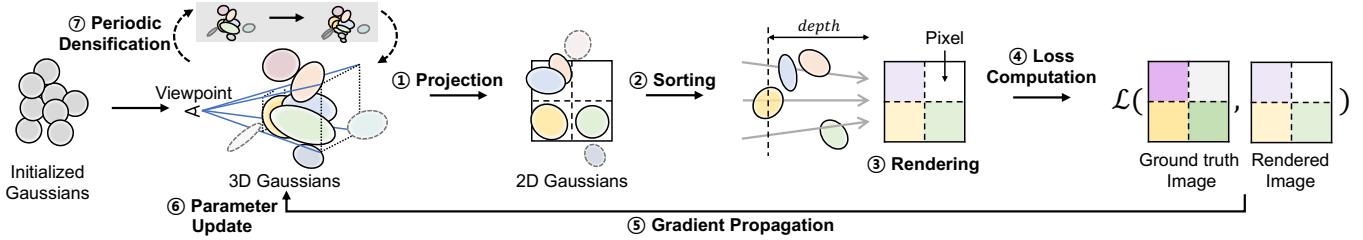


Figure 2. Overview of 3D Gaussian Splatting Training Pipeline.

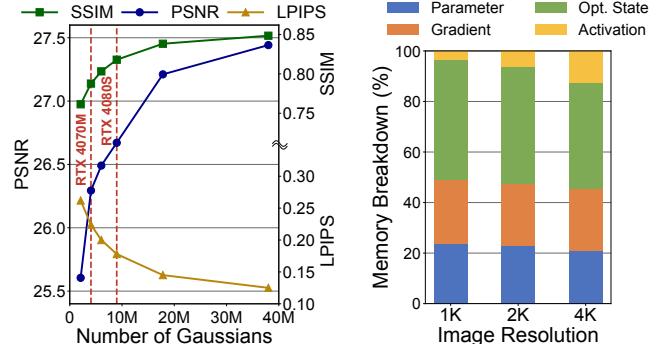
Training iteratively performs following seven key steps. ① 3D Gaussians (ellipsoids) are projected onto the image plane, producing 2D Gaussians (ellipses). Gaussians outside the near and far planes of the viewing frustum are excluded from projection (*frustum culling*). Projection of 3D Gaussians consists of two steps. First, the geometric parameters of each 3D Gaussian (i.e., 3D *mean*, *scale*, *quaternion*) are transformed into its 2D counterparts (2D *mean* and covariance matrix). Second, the RGB color of each 2D Gaussian is computed from the spherical harmonics coefficients and the current view direction. After the projection, *frustum culling* is performed again, excluding 2D Gaussians outside the image boundaries from subsequent processing. ② The resulting 2D Gaussians are sorted by depth to ensure correct occlusion ordering. ③ The color of depth sorted 2D Gaussians are blended to produce the final rendered image by using the same classical volume rendering equation as NeRF. ④ The difference between the rendered image and the corresponding ground-truth image is computed to produce the loss value. ⑤ The gradients of this loss are backpropagated. ⑥ The backpropagated gradients are used to update the 3D Gaussian parameters. ⑦ Periodically (e.g., every 100 iterations), 3DGS performs densification, adaptively controlling Gaussian density to improve scene representation quality. Gaussians with large accumulated gradients are split or cloned to capture fine details, while insignificant, low-opacity Gaussians are pruned. This step stops after a predefined iteration threshold.

3 Motivation

3.1 Scaling Challenges in 3D Gaussian Splatting

3D Gaussian Splatting (3DGS) demands significantly more memory than NeRF-based methods due to its explicit scene representation.

This memory pressure intensifies during training, consuming over four times the memory of the Gaussian parameters due to the need to store gradients, two optimizer states per parameter (momentum and variance in case of Adam optimizer), and additional activation memory. As demonstrated in Figure 3a, increasing the number of Gaussians improves rendering quality, but the GPU memory limit restricts 3DGS scalability. A single RTX 4080 Super GPU can train about 9 million Gaussians, limiting the PSNR to 26.67 on Rubble



(a) Effect of the Number of Gaussians on Rendering Quality

(b) Breakdown of GPU Memory Usage

Figure 3. Scaling Challenges and Memory Bottleneck Analysis on 3D Gaussian Splatting Training.

scene. This limitation is critical given that 3D Gaussian Splatting is frequently used to train user-captured personalized 3D scenes, which often relies on consumer-grade GPUs with limited memory capacity.

3.2 Memory Bottleneck Analysis in Training

Figure 3b shows a detailed breakdown of GPU memory usage with varying image resolutions measured on Building [46] scene. We observe that Gaussian parameters, gradients, and optimizer states account for around 90% of the total memory usage, while activations, used during forward and backward propagation, only comprise around 10%. This trend becomes even more pronounced when lower image resolutions are used because activation size scales with the number of rendered pixels. Considering that 1K to 4K resolutions are commonly used in 3DGS [33, 34, 44, 46, 53], reducing GPU memory usage requires targeting the Gaussian-related components rather than activations.

3.3 Opportunities of Host Offloading

A unique characteristic of 3D Gaussian Splatting training pipeline is that only Gaussians within the viewing frustum are used for rendering (forward propagation), loss computation, and backward propagation. Our profiling results in Figure 4 show that each training iteration utilizes only 8.28% of total Gaussians on average in large-scale scenes. Most

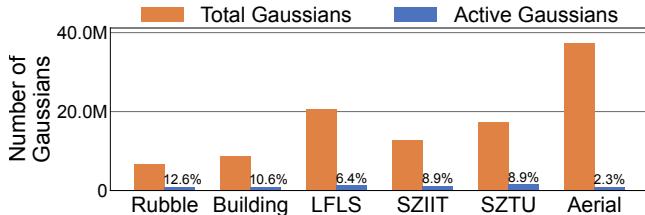


Figure 4. Average Number of Gaussians Used while Training Compared to Total Gaussians.

training stages operate on a small subset of Gaussians, except for frustum culling, which needs access to all Gaussians, and optimizer updates, which update all parameters and optimizer states. This insight suggests offloading all Gaussian parameters and optimizer states to host (CPU) memory, transferring only necessary data to GPU memory on demand to significantly save GPU memory.

3.4 Challenges in Host Offloading

While conceptually simple, offloading Gaussian parameters and optimizer states to host memory introduces several significant challenges as described below.

Challenge 1: Frustum culling is slow on the CPU. Identifying Gaussians within the viewing frustum requires processing the entire set of Gaussians. Accurate frustum culling requires projecting each 3D Gaussian onto the 2D image plane to determine whether it lies within the image boundaries. Performing this compute-intensive operation on the CPU, with its significantly lower FLOPS compared to a GPU, introduces substantial overhead.

Challenge 2: Slow optimizer updates on CPU due to low CPU memory bandwidth and inefficient nature of Adam optimizer. Adam [24], which is the most widely used optimizer in 3DGS [22, 49, 53], updates all parameters and optimizer states including those with zero gradients since its momentum terms remain nonzero even when the gradients are zero (refer to Equation 1). Thus, all parameters and optimizer states must be updated by CPU, regardless of whether the corresponding Gaussians were involved in forward/backward propagation since GPU memory cannot hold them all. Given that optimizer updates are memory-bound and CPU memory bandwidth is typically much lower than GPU memory, this leads to considerable training slowdown.

Challenge 3: Peak memory usage is bound by the most demanding training image. Although only the Gaussians within the viewing frustum of each training image are fetched on demand, the peak memory usage is determined by the image that requires the largest number of Gaussians. Even if most training images activate a small subset of Gaussians, a single image with an exceptionally large coverage (i.e., image seen from a far viewpoint) can dominate the peak memory requirement, limiting the effectiveness of host offloading.

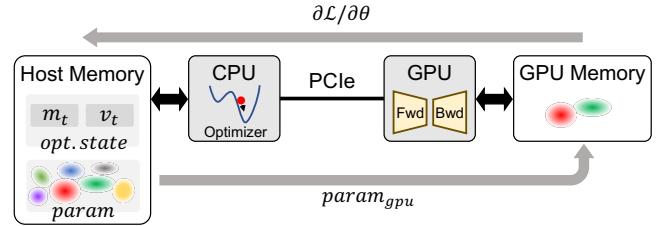


Figure 5. Baseline GS-Scale with Host Offloading.

4 GS-Scale Design

This section introduces GS-Scale, our novel system designed to overcome the three challenges in the host offloading. GS-Scale leverages strategic host offloading combined with several system-level optimizations to enable efficient and scalable 3D Gaussian Splatting training on commodity GPUs.

4.1 Baseline GS-Scale with Host Offloading

To the best of our knowledge, no prior work has explored host (CPU) offloading to reduce GPU memory usage in 3D Gaussian Splatting training. Thus, we first implement a baseline training system that offloads Gaussians to host memory without specific optimizations proposed in Section 4.2. Figure 5 illustrates the system. All Gaussian parameters and optimizer states reside in host memory. Only the necessary Gaussians are transferred to GPU memory via a PCIe interconnect for forward and backward passes, with gradients then sent back to the CPU for optimizer updates.

Training Process: Figure 6 illustrates the training iteration of the baseline GS-Scale, and Figure 9b shows the corresponding execution timeline. The memory state at each timestamp (i.e., T_0 , T_1 , T_2) in Figure 6 is a snapshot of the system at the corresponding point in time shown in Figure 9b. ① Training begins with CPU-based frustum culling identifying Gaussians within the training image's viewing frustum (Gaussian #1 and #3). This step relies solely on spatial relationships, thus only geometric parameters, i.e., mean, scale, quaternion, are used (refer to the hatched area). ② The IDs of the selected Gaussians are stored in `valid_ids`, and the corresponding parameters (i.e., W_1 , W_3) are transferred to GPU memory via PCIe. ($t = T_0$). ③ Forward and backward passes are performed on the GPU ($t = T_1$). ④ The gradients (i.e., G_1 , G_3) are transferred back to the CPU. ⑤ Adam optimizer updates all Gaussian parameters and states on the CPU ($t = T_2$). Note that Adam optimizer also updates Gaussian parameters that do not receive gradients (i.e., W_2 , W_4) because their corresponding optimizer states (i.e., O_2 , O_4) can remain nonzero. As shown in Figure 6, updated weights and optimizer states are highlighted for clarity.

Performance Challenges: Despite reducing GPU memory, this baseline system incurs significant training time overhead, making around 4× slower than GPU-only training.

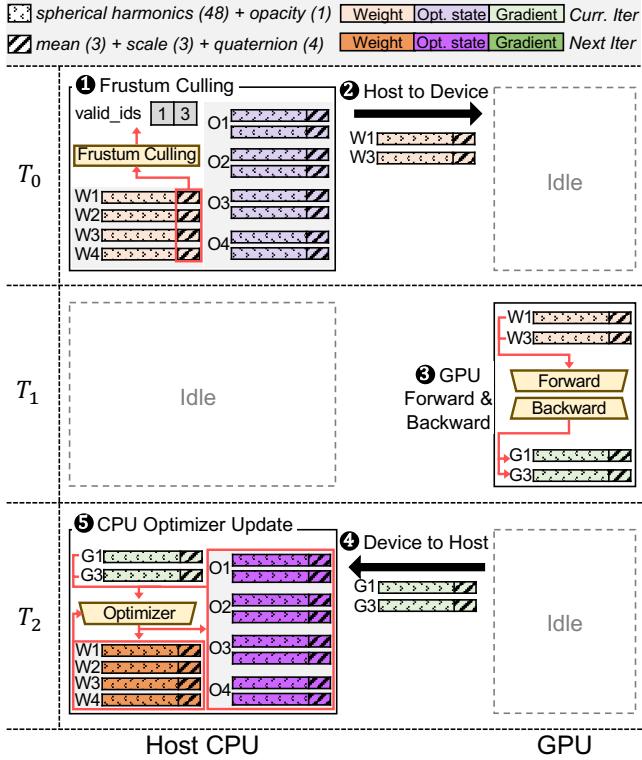


Figure 6. Training Iteration in Baseline GS-Scale.

Figure 7 presents its training time breakdown, measured on a laptop with an RTX 4070 Mobile GPU. As discussed in Section 3.4, the primary bottlenecks are frustum culling and optimizer updates, both executed on the host CPU.

- *Slow Frustum Culling on CPU (1)*: The CPU has significantly lower compute capability ($52\times$ less peak FLOPS on ASUS TUF Gaming F17 laptop) compared to the GPU, making the compute-intensive frustum culling a major bottleneck when performed on CPU.
- *Slow Optimizer Updates on CPU (5)*: The CPU’s memory bandwidth is $3\times$ slower than the GPU’s, which turns memory-intensive optimizer updates into a major bottleneck when executed on the CPU.
- *GPU Idle Time Due to Dependency (3, 5)*: A dependency exists between GPU-based forward/backward propagation and CPU-based optimizer updates, causing the GPU to remain idle for a significant amount of time during CPU execution.

4.2 GS-Scale Optimizations

To address these bottlenecks, we will explore various system-level optimization opportunities in the subsequent sections. These optimizations leverage the unique characteristics of the 3D Gaussian Splatting training pipeline (Section 3.3).

4.2.1 Selective Offloading. To mitigate the CPU-based frustum culling bottleneck, we propose *selective offloading*,

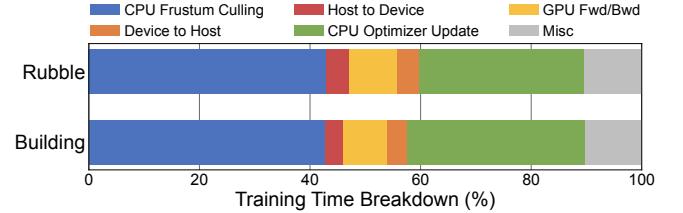


Figure 7. Training Time Breakdown of Baseline GS-Scale Measured on RTX 4070 Mobile GPU.

moving this operation to the GPU. Since only the position and the size of Gaussians (i.e., mean, scale, quaternion) are needed to determine visibility within the viewing frustum, only the geometric attributes of all parameters are kept on the GPU for fast frustum culling. The geometric attributes comprise only 10 out of 59 Gaussian parameters, resulting in a modest 17% GPU memory overhead. This is a worthwhile trade-off for significantly faster GPU-based frustum culling and reduced training time. Also, the non-geometric attributes (83%) are offloaded to host memory, still achieving considerable memory saving.

4.2.2 Breaking Data Dependency via Parameter Forwarding. Optimizer updates become the primary bottleneck after selective offloading. To mitigate this overhead, we introduce a pipelined training scheme, enabling concurrent execution of forward/backward passes on GPU and CPU optimizer updates. Typically, such pipelining is not feasible due to the data dependency since updated parameters are needed for the next training iteration’s forward pass.

We identify a unique opportunity offered by 3D Gaussian Splatting training workloads. Each training iteration requires only a small subset of parameters corresponding to Gaussians within the viewing frustum of the next training image. We exploit this with *parameter forwarding*, which performs early updates of only the parameters needed for the next iteration on the CPU and forwards the updated parameters to the GPU. Remaining parameters are updated asynchronously on the CPU in a lazy manner, enabling pipelining with forward/backward passes on GPU.

Figure 8 demonstrates the detailed working example of GS-Scale with both *selective offloading* and *parameter forwarding*. The memory state snapshot of Figure 8 (T'_0 , T'_1 , T'_2 , and T'_3) corresponds to timestamps on the execution timeline of Figure 9c. We assume that the forward and backward propagation for visible Gaussians #1 and #3 are complete as a part of the $(N-1)^{th}$ training iteration. Their gradients (G1 and G3) have been generated and stored in host memory at T'_0 . The frustum culling operation for the N^{th} training iteration is also complete, identifying visible Gaussians #1 and #2 and assigning 1 and 2 to *valid_ids*. These *valid_ids* are used for the forward and backward propagation of the N^{th} training iteration.

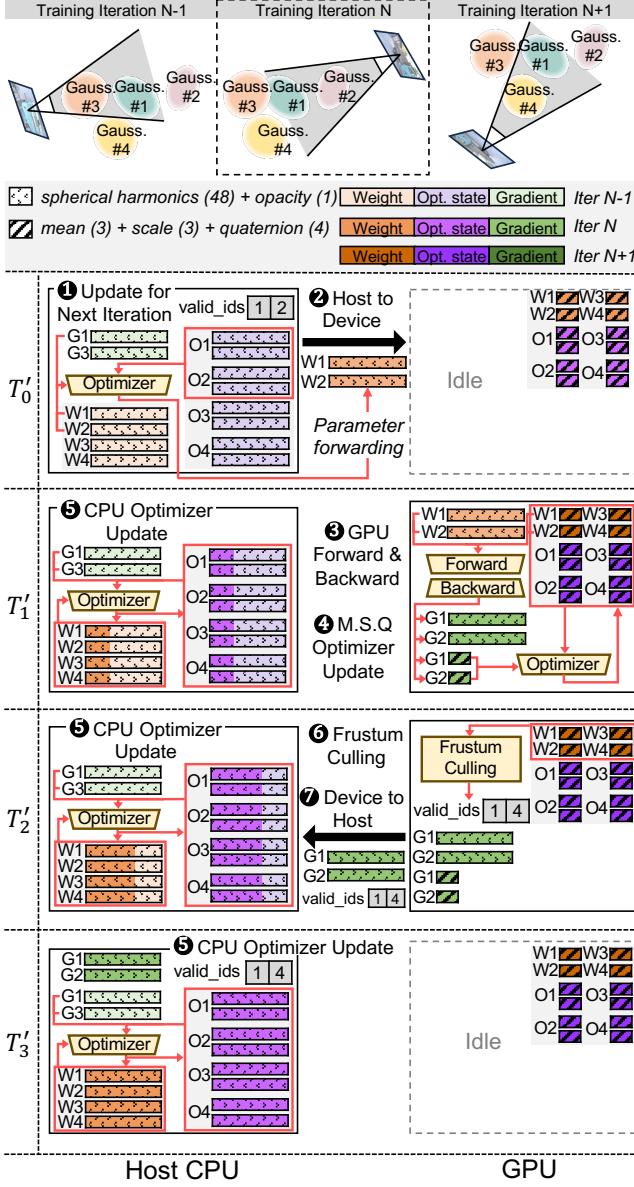


Figure 8. N^{th} Training Iteration in GS-Scale with Selective Offloading and Parameter Forwarding.

① *Parameter forwarding* updates only the parameters required for the current iteration (W1 and W2) by using the corresponding gradient (G1) from the previous iteration; G2 is zero at this point. ② The updated parameters are transferred to the GPU via PCIe; CPU-side copies of the parameters and optimizer states remain unchanged ($t = T'_0$). To mitigate the transfer overhead, parameters are partitioned into 32MB chunks, enabling pipelined execution between CPU-side optimizer updates and host-to-device transfers. The execution timeline in Figure 9c illustrates this scheme. ③ Once transferred, the GPU executes forward/backward passes using these parameters and geometric parameters (mean, scale, quaternion of Gaussian #1 and #2) already on the GPU via

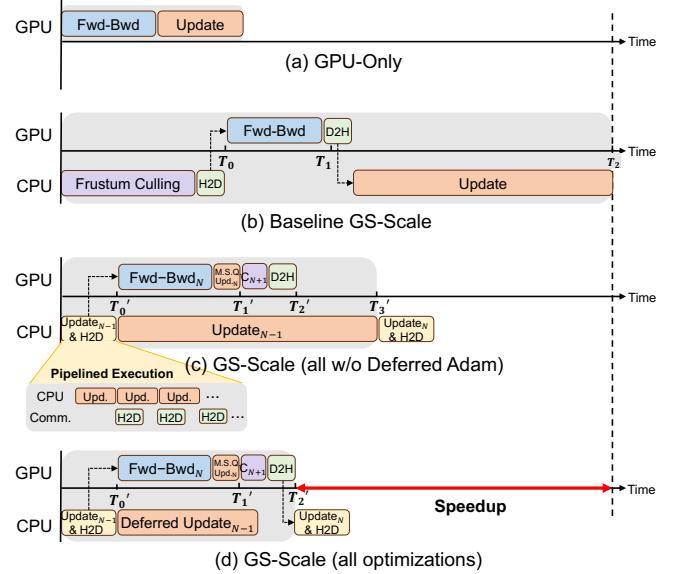


Figure 9. Execution Timeline of GPU-Only, Baseline GS-Scale, and GS-Scale with optimizations. H2D/D2H denote Host to Device transfers, M.S.Q. is mean/scale/quaternion.

selective offloading. ④ Since geometric parameters and the corresponding optimizer states always reside in the GPU, they are immediately updated after the forward/backward pass ($t = T'_1$). ⑤ Concurrently ($t = T'_1$), remaining parameters and their optimizer states, including non forwarded ones to the GPU, are lazily updated on the CPU, minimizing GPU idle time. Note that this optimizer process is a part of the $(N-1)^{th}$ training iteration. ⑥ Once the GPU-side geometric parameters are updated, frustum culling is performed using the updated parameters and the next training image, identifying the visible Gaussians for the $(N+1)^{th}$ iteration. ⑦ Finally, gradients of non-geometric parameters are transferred back to CPU and held until the CPU completes its optimizer updates ($t = T'_3$).

4.3 Deferred Optimizer Update

While parameter forwarding enables pipelining between CPU and GPU, a slow CPU-based optimizer can still dominate overall execution, as shown in Figure 9c. This is mostly due to low CPU memory bandwidth and Adam optimizer’s inefficiency (updating all states and parameters, even those with zero gradients). To further accelerate the CPU-based Adam optimizer without algorithmic changes, we propose *deferred optimizer update*. Although we use Adam as an example, *deferred optimizer update* can be extended to most momentum-based optimizers, such as SGD (stochastic gradient descent) with momentum and AdamW [35].

4.3.1 Optimization Opportunities. We can defer updates for parameters and optimizer states with zero gradients because their values can be precisely reconstructed by tracking

deferred iterations. This is due to momentum based optimizer's deterministic behavior when gradients are zero, as shown in Adam's example (Equation 1). If gradient g_t is zero, momentum and variance m_t and v_t are simply scaled by β_1 and β_2 , respectively.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1)g_t, \quad \hat{m}_t = \frac{m_t}{1 - \beta_1^t} \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2)g_t^2, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \\ w_{t+1} &= w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \end{aligned} \quad (1)$$

This property enables us to restore current optimizer states (m_t and v_t) from deferred optimizer states (m_{t-d-1} and v_{t-d-1} , $t > d$) and the defer count d . If the gradient remains zero for d consecutive iterations and becomes non-zero at iteration t , momentum and variance can be reconstructed by simply multiplying scaling factors as follows:

$$\begin{aligned} m_t &= \underbrace{\beta_1^{d+1} m_{t-d-1} + (1 - \beta_1)g_t}_{m_scale} \\ v_t &= \underbrace{\beta_2^{d+1} v_{t-d-1} + (1 - \beta_2)g_t^2}_{v_scale} \end{aligned} \quad (2)$$

Parameter w_t can also be restored from deferred parameter w_{t-d} , momentum m_{t-d-1} , and variance v_{t-d-1} by repeatedly applying the weight update d times:

$$\begin{aligned} w_t &= w_{t-d} - \sum_{l=0}^{d-1} \frac{\eta}{\sqrt{\hat{v}_{t-d+l}} + \epsilon} \hat{m}_{t-d+l} \\ &= w_{t-d} - \sum_{l=0}^{d-1} \frac{\eta}{\sqrt{\frac{\beta_2^{l+1} v_{t-d-1}}{1 - \beta_2^{t-d+l}} + \epsilon}} \cdot \frac{\beta_1^{l+1} m_{t-d-1}}{1 - \beta_1^{t-d+l}} \\ &\approx w_{t-d} - \frac{m_{t-d-1}}{\sqrt{v_{t-d-1}} + \epsilon} \sum_{l=0}^{d-1} \underbrace{\frac{\eta}{\sqrt{\frac{\beta_2^{l+1}}{1 - \beta_2^{t-d+l}}}} \cdot \frac{\beta_1^{l+1}}{1 - \beta_1^{t-d+l}}}_{w_scale} \end{aligned} \quad (3)$$

Assuming that ϵ is small, we can factor out m_{t-d-1} and v_{t-d-1} , making the remaining expression (i.e., w_scale) a precomputable constant, which simplifies weight restoration. Note that ϵ is typically a very small constant introduced to prevent divide-by-zero errors and this approximation has negligible effect on training, which we substantiate in Section 5.5. Finally, the restored w_t , m_t , and v_t are used to produce final weight w_{t+1} with the original Adam rule.

4.3.2 Implementation. We propose the *deferred optimizer update*, which defers updates for Gaussians not involved in forward and backward propagation. Instead of immediate updates, a 4-bit counter increments for deferred updates, allowing up to 15 deferrals. Parameters and optimizer states are restored only when their corresponding gradient becomes non-zero or the counter reaches its maximum. Even

```

1 // N: Number of total Gaussians
2 // D: Dimension of each parameter
3 float param[N][D], grad[N][D], mom[N][D], var[N][D];
4 char counter[N]; int MAX = 15; // MAX: Max counter value
5 float lr, b1, b2, eps; // Hyperparameters
6
7 // n: Number of Gaussians with nonzero gradient
8 def deferred_update(vector<int> valid_ids[n], int step):
9  /* Gaussian IDs that need to be restored */
10 vector<int> update_ids;
11 update_ids = union(valid_ids, where(counter == MAX));
12
13 /* Precompute scaling factor */
14 float param_lut[MAX], mom_lut[MAX], var_lut[MAX];
15 float scale = b1 / sqrt(b2);
16 param_lut[0] = 0;
17 for i = 1 to MAX:
18  param_lut[i] = scale * param_lut[i-1] +
19    (lr * b1) / (sqrt(b2 / (1 - pow(b2, step - i))) *
20    (1 - pow(b1, step - i)));
21 for i = 0 to MAX:
22  mom_lut[i] = pow(b1, i+1);
23  var_lut[i] = pow(b2, i+1);
24
25 /* Perform optimizer update */
26 float bias_correction = sqrt(1 - pow(b2, step));
27 float step_size = lr / (1 - pow(b1, step));
28 for id in update_ids:
29  float delay = counter[id];
30  float w_scale = param_lut[delay];
31  float m_scale = mom_lut[delay];
32  float v_scale = var_lut[delay];
33  for k = 0 to D:
34   float w = param[id][k]; float g = grad[id][k];
35   float m = mom[id][k]; float v = var[id][k];
36   float m_new = m_scale * m + (1 - b1) * g;
37   float v_new = v_scale * v + (1 - b2) * g * g;
38   mom[id][k] = m_new;
39   var[id][k] = v_new;
40   w = (w_scale * m) / (sqrt(v) + eps);
41   float denom = sqrt(v) / bias_correction + eps;
42   param[id][k] = w - step_size * m_new / denom;
43
44 /* Update counter for deferred Gaussians */
45 for id = 0 to N:
46  counter[id] += 1;
47 for id in update_ids:
48  counter[id] = 0;
49 }
```

Figure 10. Pseudocode of Deferred Optimizer Update.

with conservative estimates, this results in only 6.7% (1/15) unnecessary updates due to counter saturation.

Figure 10 details the pseudocode. A set of Gaussians to be updated (`update_ids`) is determined as the union of those with nonzero gradients (`valid_ids`) and those whose counter has reached MAX (Line 11). Three scaling factors for parameter and optimizer state restoration are precomputed and stored in lookup tables for each deferred step d (Line: 14–23), following the equations in Section 4.3.1. For each Gaussian in `update_ids`, its defer count is read (Line 29), scaling factors are retrieved (Line: 30–32), parameters and states are reconstructed (Line: 34–40), and the standard Adam update is applied (Line: 41–42). Counters for updated Gaussians are then reset, while deferred ones increment by 1 (Line: 45–48).

Deferred optimizer update significantly reduces memory accesses, proportional to the ratio of used to total Gaussians, while incurring minimal overhead. Each counter lookup/update requires a single 8-bit memory access (`char` datatype)

Table 1. Specifications of GPU Platforms

GPU	GPU Memory		PCIe BW	Host Memory		R_{bw}
	Size	BW		Size	BW	
Laptop						
RTX 4070M	8 GB	256 GB/s	16 GB/s	32 GB	83.2 GB/s	3.1
Desktop						
RTX 4080S	16 GB	736 GB/s	32 GB/s	64 GB	89.6 GB/s	8.2
Server						
H100 80GB	80 GB	2.04 TB/s	64 GB/s	1 TB	614.4 GB/s	3.3

per Gaussian. However, a full optimizer update requires $7D * 32$ -bit accesses per Gaussian ($4D$ reads and $3D$ writes for parameters, gradients, and optimizer states, where D is parameter dimension, i.e., 59; refer to Line 33-42). Parameter and optimizer state restoration adds some computation but incurs no additional memory accesses (Line 40), thus having little impact on overall execution time, as optimizer updates are primarily memory-bound.

4.3.3 Integration to GS-Scale. Deferred optimizer update integrates into the GS-Scale pipeline with minor adjustments to *parameter forwarding*. Since forwarded parameters must be accurate, weight restoration is performed before forwarding the parameters. Neither CPU-stored original parameters nor counters are modified during this parameter forwarding process (1 in Figure 8), while they are updated in the actual CPU optimizer update process (5 in Figure 8).

4.4 Balance-Aware Image Splitting Training

Even with GS-Scale’s significant GPU memory savings, peak memory usage is bound by the the maximum number of Gaussians from the most demanding training image. To address such cases, we propose *balance-aware image splitting training*. When the ratio of active to total Gaussians exceeds a predefined threshold `mem_limit`, an image is spatially partitioned into two sub-regions, each processed separately. Each sub-region undergoes independent frustum culling, followed by separate forward and backward passes to compute individual losses and gradients. The gradients are transferred to CPU immediately after they are computed and are later aggregated on the CPU, mitigating GPU memory pressure. Optimizer update is applied once for both regions using the aggregated gradients on the CPU, ensuring mathematical equivalence to the original training pipeline. Splitting a demanding image into two can halve memory usage during forward/backward passes, preserving memory savings. While more splits are possible, two sufficed in our benchmarks.

Finding an optimal split point is critical to balance Gaussian counts, as naive equal-area splitting often leads to imbalance due to varying Gaussian density. Our *balance-aware image splitting strategy*, applied once before training using the initial 3D Gaussians, addresses this. We efficiently balance counts by starting at the image midpoint, performing frustum culling on both sides, and then iteratively adjusting

Table 2. Evaluated Benchmark Scenes

Dataset	Scene	Resolution	Type
Mill-19 [46]	Rubble Building	1152×864	Real World & Outdoor
GauU-Scene [48]	LFLS SZIIT SZTU	1600×1064	Real World & Outdoor
MatrixCity [28]	Aerial	1600×900	Synthetic

the split toward the less-populated side via a 5-step binary search. This process adds only 0.08% overhead to total training time. Despite slight changes resulting from densification, our benchmarks show an average split point ratio of 0.551:0.449, maintaining balance throughout training.

5 Evaluation

5.1 Methodology

We build GS-Scale on gsplat v1.5.0 [49], a popular PyTorch [40] based 3D Gaussian Splatting framework, which achieves state-of-the-art performance in terms of both training speed and memory usage. We implement pipelined CPU–GPU execution using Python’s threading module and *deferred optimizer update* as a custom C++ PyTorch extension with OpenMP parallelization. The source code of GS-Scale is available at <https://github.com/SNU-ARC/GS-Scale.git>. Experiments are conducted primarily on laptop and desktop platforms, with additional evaluation on a server. We use ASUS TUF Gaming F17 laptop [4] with Intel Core i7-13620H CPU and RTX 4070 Mobile GPU, desktop with Intel Core i9-13900K CPU and RTX 4080 Super GPU, and server with 2×Intel Xeon Gold 6530 CPU and H100 PCIe 80GB GPU. Table 1 shows detailed specifications. R_{bw} [39] denotes the ratio of GPU memory bandwidth to that of CPU. All platforms use CUDA 12.4 and PyTorch 2.2.0.

We evaluate GS-Scale on large-scale datasets (Table 2). We use 4× downsampled images for Mill-19 dataset and 1.6k resolution downsampled images for the other datasets following previous works [33, 34, 44, 46]. Following the original 3DGS recipe, we use a batch size of 1, as larger batches offer minimal throughput gains due to limited parameter sharing. We use three standard rendering quality metrics: Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index Measure (SSIM), and Learned Perceptual Image Patch Similarity (LPIPS), where higher PSNR/SSIM and lower LPIPS indicate better quality. GPU peak memory usage is measured via PyTorch CUDA Memory Management APIs¹.

We adjust densification settings (i.e., stop iteration, densification threshold, and split/clone decision threshold) to scale up or scale down Gaussian counts for each scene, following

¹GPU peak memory is measured based on allocated memory. Since PyTorch maintains reserved memory pools larger than the allocated memory to reduce allocation/deallocation overhead, OOM errors may occur before allocated memory reaches the GPU memory capacity.

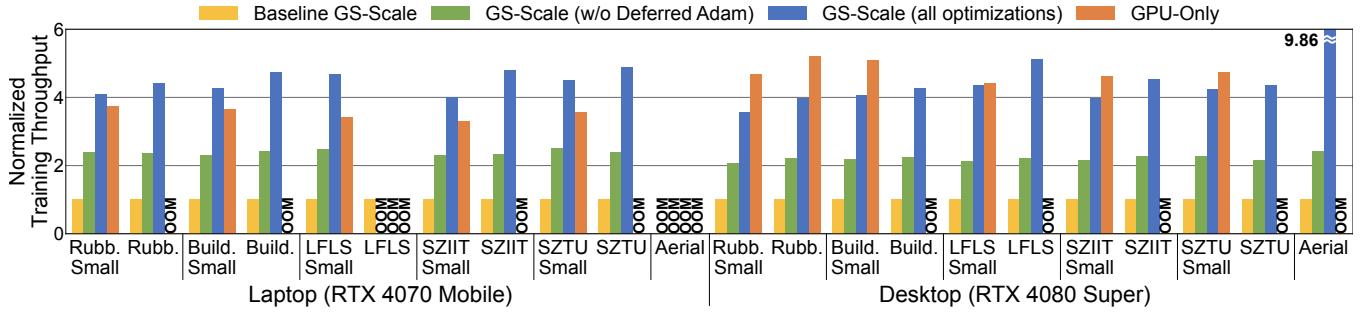


Figure 11. Training Throughput Normalized to Baseline GS-Scale.

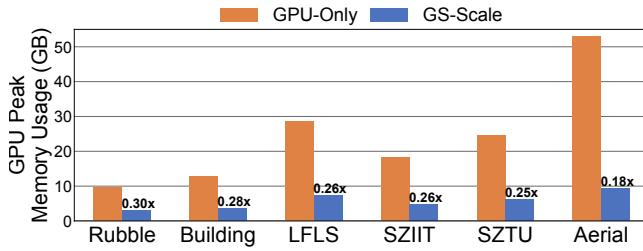


Figure 12. Peak GPU Memory Usage Savings with GS-Scale.

the Grendel’s methodology [53]. A `mem_limit` of 0.3 is used for all experiments, splitting images when active Gaussians exceed 30% of the total.

5.2 Memory Savings

We evaluate GS-Scale’s memory savings against the GPU-Only system. Figure 12 demonstrates that GS-Scale achieves a substantial 3.98× geomean reduction in peak memory usage across all datasets. The memory savings correlate with the ratio of used to total Gaussians, explaining the greatest improvement in the Aerial scene. Further reduction is limited in the Aerial scene despite its low used Gaussian ratio because 17% of parameters and optimizer states remain resident on the GPU due to *selective offloading*.

5.3 Training Throughput and Memory Efficiency

Figure 11 evaluates training throughput across four systems: (1) baseline GS-Scale, (2) GS-Scale with all optimizations except deferred optimizer update, (3) GS-Scale with all optimizations, and (4) GPU-Only system without host offloading. Six scenes are evaluated across laptop and desktop platforms and training speed is measured in epoch time. We make smaller versions of each scene by adjusting densification settings to enable throughput comparisons. However, we could not create a smaller version that fits into GPU memory for the Aerial scene, as its Gaussian count is already too large at initialization. Since our downsizing strategy [53] relies on limiting densification, scenes that trigger OOM errors before densification cannot be further downsized.

GPU-only system frequently encounters OOM errors due to limited memory, but GS-Scale’s significant memory savings enable much larger-scale 3DGS training. For instance, the Aerial scene alone demands over 50GB of GPU memory without host offloading (Section 5.2), causing OOM on both GPU-only systems. However, GS-Scale reduces peak GPU memory usage by 5.5×, allowing the Aerial scene to be trained on an RTX 4080 Super desktop. Furthermore, GS-Scale achieves comparable training throughput to GPU-Only systems, reaching geomean of 1.22× (laptop) and 0.84× (desktop) of GPU-Only performance (excluding OOM cases). A *takeaway is that GS-Scale enables much larger scene training and consistently maintains high training throughput, even as GPU-Only systems frequently encounter OOM errors.*

5.4 Impact of Proposed Optimizations

Figure 11 shows how GS-Scale’s optimizations improve the training throughput over the baseline GS-Scale. We see a geomean improvement of 4.47× on laptop and 4.57× on desktop (excluding OOM cases), demonstrating GS-Scale’s effectiveness. The performance of GS-Scale depends on two key factors: (1) the GPU to CPU memory bandwidth ratio (R_{bw}) and (2) the average ratio of used to total Gaussians (Figure 4). PCIe bandwidth also has some effect, but its impact is limited as it accounts for only a small portion of the overall training time. On platforms with lower R_{bw} (like laptops), GS-Scale can perfectly pipeline CPU optimizer updates with GPU execution, even surpassing GPU-Only speeds where operations are executed sequentially. This is because lower GPU memory bandwidth slows down the memory bound backward pass (i.e., gradient accumulation) on GPU, providing enough time for CPU updates to be pipelined. Furthermore, a lower ratio of used to total Gaussians amplifies the benefits of the deferred optimizer update, as memory access reduction scales with this ratio, explaining the notable speedups in Aerial and LFLS scenes.

5.5 Training Quality Impact of GS-Scale

The only approximation in GS-Scale is ignoring the ϵ term in the *deferred optimizer update* for factoring out momentum and variance terms. To analyze its impact, we compare the

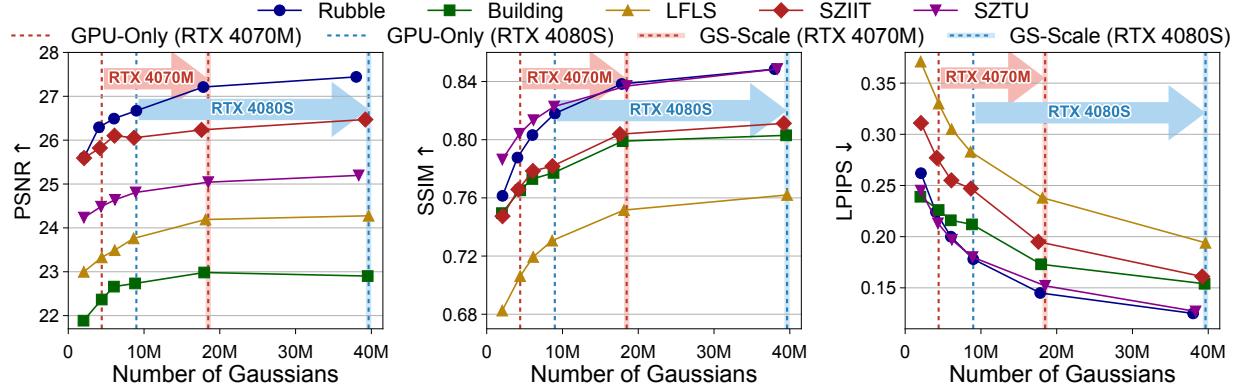


Figure 13. Evaluation of GS-Scale’s Rendering Quality and Scalability Across Gaussian Scales.

Table 3. Impact of GS-Scale on Training Quality.

Scene	Method	PSNR \uparrow	SSIM \uparrow	LPIPS \downarrow
Rubble	Original	26.63	0.808	0.194
	GS-Scale	26.62	0.808	0.194
Building	Original	22.74	0.777	0.211
	GS-Scale	22.78	0.777	0.211
LFLS	Original	24.04	0.752	0.234
	GS-Scale	24.08	0.752	0.233
SZIIT	Original	26.28	0.797	0.213
	GS-Scale	26.29	0.797	0.213
SZTU	Original	24.90	0.835	0.155
	GS-Scale	24.95	0.836	0.155
Aerial	Original	27.69	0.873	0.127
	GS-Scale	27.66	0.873	0.128

rendering quality of models trained with the original method and with GS-Scale. Table 3 shows this approximation has negligible impact on rendering quality, confirming that GS-Scale maintains the rendering quality of the trained models as in the original training pipeline.

5.6 Improved Scalability and Rendering Quality

Leveraging its memory savings discussed in Section 5.2, GS-Scale enables training with substantially more Gaussians under the same GPU memory budget, leading to higher rendering quality. We assess this by examining rendering quality changes with increasing Gaussian counts. Figure 13 demonstrates that more Gaussians consistently yield higher PSNR and SSIM and lower LPIPS, indicating better rendering and reconstruction quality. The figure also shows GS-Scale extends the maximum Gaussians scaling across different platforms and systems. On a laptop with RTX 4070 Mobile GPU, GS-Scale scales the number of Gaussians from 4 million to 18 million, achieving geomean 2.6% PSNR and 5.1% SSIM increases, and a 28.7% LPIPS decrease. On a desktop with RTX 4080 Super GPU, it scales the number of Gaussians from 9 million to 40 million, resulting in geomean 1.6% PSNR and 3.6% SSIM increases, and 30.5% LPIPS decrease. These

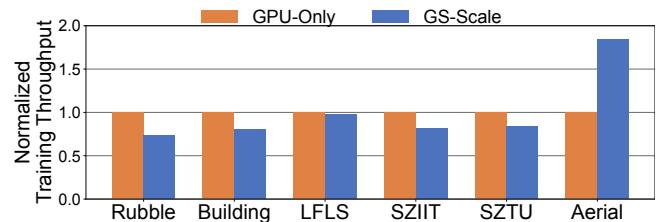


Figure 14. Training Throughput on Server Platform.

results substantiate that the scalability of GS-Scale directly translates into higher rendering quality across platforms.

5.7 Evaluation on Server Platform

Although GS-Scale is primarily designed for laptop and desktop platforms, we also evaluate it on server platform with H100 GPU to demonstrate its broader applicability. The results on server shown in Figure 14 follows a similar trend with laptop and desktop platforms, while substantial speedup is achieved on Aerial scene thanks to the large speedup gain from *deferred optimizer update* as discussed in Section 5.3. We also observe that the overall training throughput normalized to GPU-only on the server is relatively lower than that of laptop despite having a similar $R_{b,w}$ value. This is because the server consists of two NUMA nodes, making it relatively harder for *deferred optimizer update* with random memory accesses to exploit the peak CPU memory bandwidth compared to laptop with single node.

5.8 Sensitivity Study

Sensitivity to mem_limit. Figure 15a and b demonstrate how GPU memory usage and training throughput changes with varying `mem_limit` thresholds. We can save more GPU memory by decreasing this threshold at the cost of slower training throughput. Slowdown results from the additional frustum culling and gradient accumulation required by image splitting. In our experiments, we use `mem_limit` of 0.3 to prioritize training throughput over memory savings.

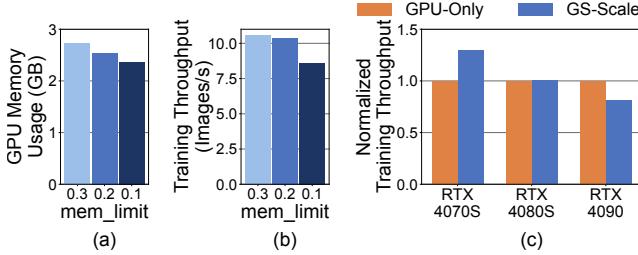


Figure 15. (a), (b) Sensitivity to `mem_limit` on Rubble scene. (c) Sensitivity to GPU on LFLS scene. Desktop is used.

Sensitivity to GPU. Figure 15c shows GS-Scale’s training throughput on additional desktop GPUs (RTX 4070 Super, RTX 4090). Higher R_{bw} values on RTX 4090 ($R_{bw} = 11.3$ VS. $R_{bw} = 5.6$ on RTX 4070 Super) with greater GPU memory bandwidth (1.01 TB/s VS. 504.2 GB/s) explains its lower normalized throughput of GS-Scale compared to GPU-Only.

Sensitivity to Image Resolution. Figure 16 shows that GPU memory savings slightly decrease as training image resolution increases since growing activation memory (Figure 3b) reduces the relative portion of offloadable parameters, optimizer states, and gradients. Conversely, relative training throughput increases. This is because higher resolutions slow down the GPU-based forward/backward pass, providing more slack for pipelining CPU-based optimizer updates.

6 Related Work

Acceleration on 3D Gaussian Splatting Rendering. Several works have been proposed to accelerate 3DGS rendering through both software optimizations [12, 13, 18, 20, 30, 41, 45, 54] and specialized accelerators [14, 25, 27, 32, 47, 50]. For software-only solutions, GS-Cache [45] reduces redundant computations via caching data across frames, combined with an efficient scheduler and optimized GPU kernels. FlashGS [13] eliminates unnecessary computations via a precise intersection test and improves GPU utilization by overlapping memory access with computation. For hardware accelerators, GSCore [25] introduces the first dedicated accelerator for 3DGS, eliminating sorting and rasterization for unnecessary Gaussians through algorithm-hardware co-design. Lumina [14] also mitigates sorting and rasterization bottlenecks by sharing sorting results across frames and caching previous rendering results via hardware support. MetaSapiens [32] adopts efficiency-aware pruning and foveated rendering, co-designed with a specialized accelerator to enable real-time rendering.

Acceleration on 3D Gaussian Splatting Training. 3DGS training suffers from large amount of atomic operations during gradient accumulation and various works [10, 11, 19, 29, 36] have been proposed to address this bottleneck. DISTWAR [11] accelerates atomic operations by enabling

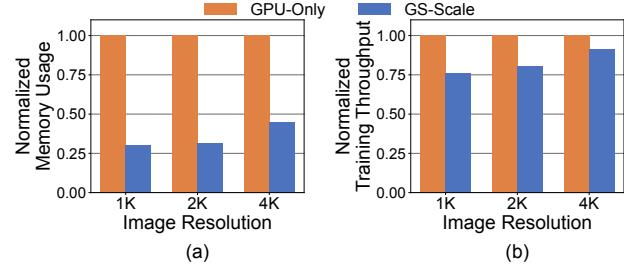


Figure 16. Impact of Image Resolution on Memory Usage and Throughput on Desktop for Rubble Scene.

warp-level reduction and leveraging L2 atomic units (ROP units). ARC [10] further addresses atomic bottlenecks by introducing specialized hardware unit for atomic operations. GSArch [19] reduces both off-chip and on-chip memory accesses in 3DGS training through gradient pruning and on-chip memory access rearrangement.

Scaling 3D Gaussian Splatting. Recent works have tackled the challenges in large scale 3DGS training via both algorithm level and system level solutions. Most algorithm-level approaches [23, 26, 31, 33, 34, 44] follow a divide-and-conquer strategy: partitioning the 3D scene into smaller chunks, training them independently, and later merging the results. While this avoids out-of-memory errors, it fundamentally alters the original 3DGS training recipe. Grendel [53] is the first framework to enable large-scale 3DGS training without modifying the original 3DGS algorithm. By addressing GPU load imbalance and inter-GPU communication overhead in a distributed setting, Grendel supports efficient training with tens of millions of Gaussians. Importantly, Grendel shows that simply supporting the original 3DGS pipeline alone leads to substantially faster training and superior rendering quality compared to divide-and-conquer methods, highlighting the importance of system-level solutions in 3DGS training.

7 Conclusion

3D Gaussian Splatting offers high visual quality and fast rendering speed, but its training demands significant GPU memory. GS-Scale resolves this by offloading Gaussians to host memory, transferring only necessary subsets to the GPU on demand, greatly reducing GPU memory usage. GS-Scale also optimizes CPU-based frustum culling and optimizer updates through selective offloading, parameter forwarding, and a deferred optimizer update. Experiments show GS-Scale saves GPU memory demands by 3.3×-5.6×, maintaining training throughput comparable to GPU-only systems. This enables GS-Scale to facilitate much larger-scale 3DGS training on commodity GPUs, achieving geometric mean 28.7% and 30.5% LPIPS improvement on an RTX 4070 Mobile GPU and RTX 4080 Super GPU respectively.

References

[1] 2022. Design better buildings, together. <https://www.arkio.is>.

[2] 2022. Metaverse 3D models. <https://sketchfab.com/tags/metaverse>.

[3] 2023. 3D Gaussian Splatting: Create and view splats for free. <https://poly.cam/tools/gaussian-splatting>.

[4] 2023. ASUS TUF Gaming F17. <https://www.asus.com/laptops/for-gaming/tuf-gaming/asus-tuf-gaming-f17-2023>.

[5] 2025. Virtual Tours: Explore properties and spaces from anywhere in the world. <https://www.realhorizons.in/tours>.

[6] 2025. Your invitation to explore the world in 3D. <https://scaniverse.com>.

[7] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. 2022. TensoRF: Tensorial Radiance Fields. In *European Conference on Computer Vision (ECCV)*.

[8] Jianchuan Chen, Jingchuan Hu, Gaige Wang, Zhonghua Jiang, Tian-song Zhou, Zhiwen Chen, and Chengfei Lv. 2025. TaoAvatar: Real-Time Lifelike Full-Body Talking Avatars for Augmented Reality via 3D Gaussian Splatting. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[9] Nianchen Deng, Zhenyi He, Jiannan Ye, Budmonde Duinkharjav, Praneeth Chakravarthula, Xubo Yang, and Qi Sun. 2022. FoV-NeRF: Foveated Neural Radiance Fields for Virtual Reality. In *IEEE Transactions on Visualization and Computer Graphics*.

[10] Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, Yushi Guan, Christina Giannoula, and Nandita Vijaykumar. 2025. ARC: Warp-level Adaptive Atomic Reduction in GPUs to Accelerate Differentiable Rendering. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[11] Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, and Nandita Vijaykumar. 2024. DISTWAR: Fast Differentiable Rendering on Raster-based Rendering Pipelines. *arXiv preprint arXiv:2401.05345* (2024).

[12] Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, and Zhangyang Wang. 2024. LightGaussian: Unbounded 3D Gaussian Compression with 15 \times Reduction and 200+ FPS. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[13] Guofeng Feng, Siyan Chen, Rong Fu, Zimu Liao, Yi Wang, Tao Liu, Boni Hu, Liming Xu, Zhilin Pei, Hengjie Li, Xiuhong Li, Ninghui Sun, Xingcheng Zhang, and Bo Dai. 2025. FlashGS: Efficient 3D Gaussian Splatting for Large-scale and High-resolution Rendering. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[14] 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 *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[15] Linus Franke, Laura Fink, and Marc Stamminger. 2025. VR-Splatting: Foveated Radiance Field Rendering via 3D Gaussian Splatting and Neural Points. *Proc. ACM Comput. Graph. Interact. Tech. (PACMCGIT)* (2025).

[16] Stephan J. Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien Valentin. 2021. FastNeRF: High-Fidelity Neural Rendering at 200FPS. In *IEEE/CVF International Conference on Computer Vision (ICCV)*.

[17] Junfu Guo, Yu Xin, Gaoyi Liu, Kai Xu, Ligang Liu, and Ruizhen Hu. 2025. ArticulatedGS: Self-supervised Digital Twin Modeling of Articulated Objects using 3D Gaussian Splatting. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[18] Alex Hanson, Allen Tu, Vasu Singla, Mayuka Jayawardhana, Matthias Zwicker, and Tom Goldstein. 2025. PUP 3D-GS: Principled Uncertainty Pruning for 3D Gaussian Splatting. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[19] Houshu He, Gang Li, Fangxin Liu, Li Jiang, Xiaoyao Liang, and Zhuoran Song. 2025. GSArch: Breaking Memory Barriers in 3D Gaussian Splatting Training via Architectural Support. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[20] Qiqi Hou, Randall Rauwendaal, Zifeng Li, Hoang Le, Farzad Farhadzadeh, Fatih Porikli, Alexei Bourd, and Amir Said. 2025. Sort-free Gaussian Splatting via Weighted Sum Rendering. In *International Conference on Learning Representations (ICLR)*.

[21] Ying Jiang, Chang Yu, Tianyi Xie, Xuan Li, Yutao Feng, Huamin Wang, Minchen Li, Henry Lau, Feng Gao, Yin Yang, and Chenfanfu Jiang. 2024. VR-GS: A Physical Dynamics-Aware Interactive Gaussian Splatting System in Virtual Reality. In *ACM Transactions on Graphics (SIGGRAPH)*.

[22] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. In *ACM Transactions on Graphics (SIGGRAPH)*.

[23] Bernhard Kerbl, Andréas Meuleman, Georgios Kopanas, Michael Wimmer, Alexandre Lanvin, and George Drettakis. 2024. A Hierarchical 3D Gaussian Representation for Real-Time Rendering of Very Large Datasets. In *ACM Transactions on Graphics (SIGGRAPH)*.

[24] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*.

[25] Junseo Lee, Seokwon Lee, Jungi Lee, Junyoung Park, and Jaewoong Sim. 2024. GSCore: Efficient Radiance Field Rendering via Architectural Support for 3D Gaussian Splatting. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[26] Bingling Li, Shengyi Chen, Luchao Wang, Kaimin Liao, Sijie Yan, and Yuanjun Xiong. 2024. RetinaGS: Scalable Training for Dense Scene Rendering with Billion-Scale 3D Gaussians. In *arXiv preprint arXiv:2406.11836*.

[27] Chaojian Li, Sixu Li, Linrui Jiang, Jingqun Zhang, and Yingyan Celine Lin. 2025. Uni-Render: A Unified Accelerator for Real-Time Rendering Across Diverse Neural Renderers. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[28] Yixuan Li, Lihan Jiang, Linning Xu, Yuanbo Xiangli, Zhenzhi Wang, Dahua Lin, and Bo Dai. 2023. MatrixCity: A Large-scale City Dataset for City-scale Neural Rendering and Beyond. In *IEEE/CVF International Conference on Computer Vision (ICCV)*.

[29] Kaimin Liao, Hua Wang, Zhi Chen, Luchao Wang, and Yaohua Tang. 2025. LiteGS: A High-performance Framework to Train 3DGS in Subminutes via System and Algorithm Codesign. *arXiv preprint arXiv:2503.01199* (2025).

[30] Zhimeng Liao, Xinyang Li, Shaohui Liu, Jiakai Zhang, Xian Liu, Yikai Wang, Ying Feng, Xiaoxiao Long, Shuguang Cui, and Wenping Wang. 2024. EAGLES: Efficient Accelerated 3D Gaussians with Lightweight Encodings. In *European Conference on Computer Vision (ECCV)*.

[31] Jiaqi Lin, Zhihao Li, Xiao Tang, Jianzhuang Liu, Shiyong Liu, Jiayue Liu, Yangdi Lu, Xiaofei Wu, Songcen Xu, Youliang Yan, and Wenming Yang. 2024. VastGaussian: Vast 3D Gaussians for Large Scene Reconstruction. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[32] Weikai Lin, Yu Feng, and Yuhao Zhu. 2025. MetaSapiens: Real-Time Neural Rendering with Efficiency-Aware Pruning and Accelerated Foveated Rendering. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[33] Yang Liu, He Guan, Chuanchen Luo, Lue Fan, Naiyan Wang, Junran Peng, and Zhaoxiang Zhang. 2024. CityGaussian: Real-time High-quality Large-Scale Scene Rendering with Gaussians. In *European Conference on Computer Vision (ECCV)*.

[34] Yang Liu, Chuanchen Luo, Zhongkai Mao, Junran Peng, and Zhaoxiang Zhang. 2025. CityGaussianV2: Efficient and Geometrically Accurate Reconstruction for Large-Scale Scenes. In *International Conference on Learning Representations (ICLR)*.

[35] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations (ICLR)*.

[36] Saswat Subhajyoti Mallick, Rahul Goel, Bernhard Kerbl, Francisco Vicente Carrasco, Markus Steinberger, and Fernando De La Torre. 2024. Taming 3DCS: High-Quality Radiance Fields with Limited Resources. In *SIGGRAPH Asia 2024 Conference Papers*.

[37] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In *European Conference on Computer Vision (ECCV)*.

[38] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. In *ACM Transactions on Graphics (SIGGRAPH)*.

[39] Yeonhong Park, Jake Hyun, Hojoon Kim, and Jae W. Lee. 2025. DecDEC: A Systems Approach to Advancing Low-Bit LLM Quantization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[41] Kerui Ren, Lihan Jiang, Tao Lu, Mulin Yu, Liming Xu, Zhangkai Ni, and Bo Dai. 2025. Octree-GS: Towards Consistent Real-time Rendering with LOD-Structured 3D Gaussians. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (2025).

[42] Manuel-Andreas Schneider, Lukas Höllerin, and Matthias Nießner. 2025. WorldExplorer: Towards Generating Fully Navigable 3D Scenes. *arXiv preprint arXiv:2506.01799* (2025).

[43] Noah Snavely, Steven M. Seitz, and Richard Szeliski. 2006. Photo tourism: exploring photo collections in 3D. In *ACM Transactions on Graphics (SIGGRAPH)*.

[44] Mai Su, Zhongtao Wang, Huishan Au, Yilong Li, Xizhe Cao, Chengwei Pan, Yisong Chen, and Guoping Wang. 2025. HUG: Hierarchical Urban Gaussian Splatting with Block-Based Reconstruction for Large-Scale Aerial Scenes. In *IEEE/CVF International Conference on Computer Vision (ICCV)*.

[45] Miao Tao, Yuanzhen Zhou, Haoran Xu, Zeyu He, Zhenyu Yang, Yuchang Zhang, Zhongling Su, Lining Xu, Zhenxiang Ma, Rong Fu, Hengjie Li, Xingcheng Zhang, and Jidong Zhai. 2025. GS-Cache: A GS-Cache Inference Framework for Large-Scale Gaussian Splatting Models. *arXiv preprint arXiv:2502.14938* (2025).

[46] Haithem Turki, Deva Ramanan, and Mahadev Satyanarayanan. 2022. Mega-nerf: Scalable construction of large-scale nerfs for virtual fly-throughs. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[47] Linye Wei, Jiajun Tang, Fan Fei, Boxin Shi, Runsheng Wang, and Meng Li. 2025. No Redundancy, No Stall: Lightweight Streaming 3D Gaussian Splatting for Real-time Rendering. In *International Conference on Computer-Aided Design (ICCAD)*.

[48] Butian Xiong, Zhuo Li, and Zhen Li. 2024. GauU-Scene: A Scene Reconstruction Benchmark on Large Scale 3D Reconstruction Dataset Using Gaussian Splatting. *arXiv preprint arXiv:2401.14032* (2024).

[49] Vickie Ye, Ruilong Li, Justin Kerr, Matias Turkulainen, Brent Yi, Zhuoyang Pan, Otto Seiskari, Jianbo Ye, Jeffrey Hu, Matthew Tancik, and Angjoo Kanazawa. 2024. gsplat: An Open-Source Library for Gaussian Splatting. *arXiv preprint arXiv:2409.06765* (2024).

[50] Zhifan Ye, Yonggan Fu, Jingqun Zhang, Leshu Li, Yongan Zhang, Sixu Li, Cheng Wan, Chenxi Wan, Chaojian Li, Sreemanth Prathipati, and Yingyan Celine Lin. 2025. Gaussian Blending Unit: An Edge GPU Plug-in for Real-Time Gaussian-Based Rendering in AR/VR. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[51] Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. 2022. Plenoxels: Radiance Fields without Neural Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[52] Hongjia Zhai, Xiyu Zhang, Boming Zhao, Hai Li, Yijia He, Zhaopeng Cui, Hujun Bao, and Guofeng Zhang. 2025. SplatLoc: 3D Gaussian Splatting-based Visual Localization for Augmented Reality. *IEEE Transactions on Visualization and Computer Graphics (TVCG)* (2025).

[53] Hexu Zhao, Haoyang Weng, Daohan Lu, Ang Li, Jinyang Li, Aurojit Panda, and Saining Xie. 2025. On Scaling Up 3D Gaussian Splatting Training. In *International Conference on Learning Representations (ICLR)*.

[54] Brent Zoomers, Maarten Wijnants, Ivan Molenaers, Joni Vanherck, Jeroen Put, Lode Jorissen, and Nick Michiels. 2025. PRoGS: Progressive Rendering of Gaussian Splats. In *IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*.