ON SCALING UP 3D GAUSSIAN SPLATTING TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

3D Gaussian Splatting (3DGS) is increasingly popular for 3D reconstruction due to its superior visual quality and rendering speed. However, 3DGS training currently occurs on a single GPU, limiting its ability to handle high-resolution and large-scale 3D reconstruction tasks due to memory constraints. We introduce Grendel, a distributed system designed to partition 3DGS parameters and parallelize computation across multiple GPUs. As each Gaussian affects a small, dynamic subset of rendered pixels, Grendel employs sparse all-to-all communication to transfer the necessary Gaussians to pixel partitions and performs dynamic load balancing. Unlike existing 3DGS systems that train using one camera view image at a time, Grendel supports batched training with multiple views. We explore various optimization hyperparameter scaling strategies and find that a simple *sqrt(batch_size)* scaling rule is highly effective. Evaluations using large-scale, high-resolution scenes show that Grendel enhances rendering quality by scaling up 3DGS parameters across multiple GPUs. On the 4K "Rubble" dataset, we achieve a test PSNR of 27.28 by distributing 40.4 million Gaussians across 16 GPU, compared to a PSNR of 26.28 using 11.2 million Gaussians on a single GPU.

Figure 1: Two large-scale, high-resolution scene reconstructions using Grendel, our distributed 3D Gaussian rendering system. Both images are rendered using 16 GPUs. The left and right images are represented using 40 million and 24 million gaussians respectively. Grendel achieves state of the art quality (PSNR) for both scenes.

1 INTRODUCTION

3D Gaussian Splatting (Kerbl et al., 2023) (3DGS) has emerged as a popular technique for 3D novel view synthesis, primarily due to its faster training and rendering compared to previous approaches such as NeRF (Mildenhall et al., 2020). However, most existing 3DGS pipelines are constrained to using a single GPU for training, creating memory and computation bottlenecks when applied to high-resolution or larger-scale scenes. For example, the standard Rubble dataset (Turki et al., 2022) contains 1657 images, each with a 4K resolution. A single A100 40GB GPU can hold up to 11.2

million Gaussians – well below the quality saturation point for 3DGS. As we demonstrate in Section 5.2.1, increasing the number of Gaussians continues to improve reconstruction quality. Therefore, in order to *scale up* 3DGS training in terms of parameter count and speed, we develop the Grendel distributed training system, consisting of distributing 3DGS training across multiple GPUs and an empirical rule to automatically adapt training hyperparameters based on the batch size.

Current approaches that work around the single-device 3DGS training pipeline limitations when handling large-scale scenes typically involve compression schemes or selective rendering (Kerbl et al., 2024; Ren et al., 2024; Lu et al., 2024). However, these methods are still confined to a single GPU and are fundamentally limited by similar memory and computational bottlenecks when scaling up further.

As large-scale distributed machine learning has become industry standard and a cornerstone for many state-of-the-art AI models such as Large Language Models (Shoeybi et al., 2020; Rajbhandari et al., 2020), it is reasonable to expect that distributing workloads across multiple GPUs leads to similar performance and scale improvements, where it allows 3DGS to handle larger, higher-resolution scenes and produce higher-quality outputs.

However, 3DGS stands out as a gradient-based but *not* a neural network-based training framework. It features a unique computation pipeline with dynamic and imbalanced workload patterns within a training iteration. Consequently, existing SysML frameworks and toolkits, such as those in (Shoeybi et al., 2020; Rajbhandari et al., 2020; Li et al., 2020; Zhao et al., 2023), do not apply to this domain. The frameworks that target neural networks assume a consistent and balanced workload consisting of a series of regular tensor operations (*e.g.* GEMM) that are distinct from the dynamic, non-uniform computation patterns exhibited by 3DGS training and rendering. Therefore, designing a scalable distributed system for 3DGS is uniquely challenging.

In this paper, we present several key observations on scaling up Gaussian Splatting training that inform the design of our distributed learning pipeline for 3DGS. For instance, we note that each stage of the 3DGS training pipeline in an iteration can be effectively parallelized, but the axes of parallelization differ across stages. In the *mixed parallelism* of the 3DGS training pipeline, some computations operate on individual output pixels (allowing for pixel-wise parallelism), while others operate on individual 3D Gaussians (allowing for Gaussian-wise parallelism). This enables parallelism but necessitates communication between pipeline stages. To minimize communication, we also observe that 3DGS exhibits *spatial locality*, where only a small number of Gaussians affect each image patch. Finally, the computational intensity of rendering an output pixel changes as training progresses, so we identify the *dynamic and unbalanced workloads*. As a result, any static workload partitioning strategy will be suboptimal.

In this paper, we describe Grendel, a distributed 3DGS training framework designed under these observations. Grendel uses Gaussian-wise distribution–that is, it distributes Gaussians across GPUs–for steps in a training iteration that exhibit Gaussian-wise parallelism, and pixel-wise distribution for other steps. It minimizes the communication overhead when switching between Gaussian-wise and pixel-wise distribution by assigning contiguous image areas to GPUs during pixel-wise distribution and exploiting spatial locality to minimize the number of Gaussians transferred among GPUs. Finally, Grendel employs a dynamic load balancer that uses previous training iterations to distribute pixel-wise computations to minimize workload imbalance.

Grendel additionally scales up training by batching up multiple images. This differs from conventional 3DGS training that exclusively uses a batch size of 1, which would lead to reduced GPU utilization in our distributed framework. To maintain data efficiency and reconstruction quality with larger batches, one needs to re-tune optimizer hyperparameters. To this end, we introduce an automatic hyperparameter scaling rule for batched 3DGS training based on a heuristical *independent gradients hypothesis*. We empirically validate the effectiveness of our proposed approach — Grendel supports distributed training with large batch sizes (we test up to 32) while maintaining reconstruction quality and data efficiency compared to batch size $= 1$.

In summary, our work makes the following contributions:

- We describe the design and implementation of Grendel, a scalable, memory-efficient, adaptive distributed training system for 3DGS. Grendel allows batched 3DGS training to be scaled up and run on up to 32 GPUs.
- We explore the large-batch training dynamics of 3DGS to identify a simple *sqrt(batch_size)* learning rate scaling strategy, which enables efficient, hyperparameter-tuning-free training for batch sizes beyond one.

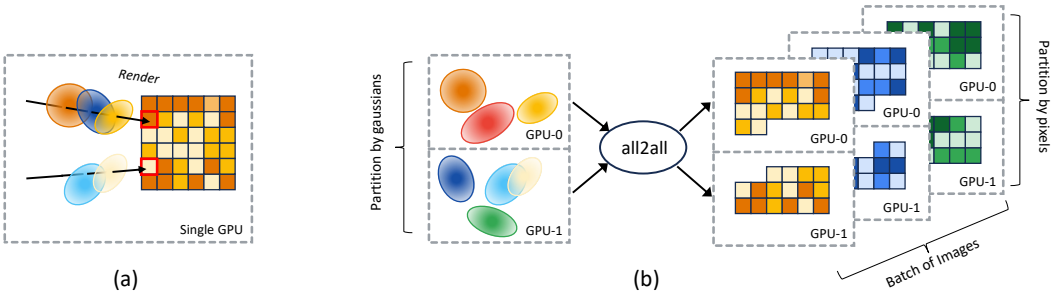


Figure 2: (a) Traditional 3DGS training pipeline using a single GPU vs. (b) Our Grendel system that distributes 3D Gaussians across multiple GPUs to alleviate the GPU memory bottleneck. We also partition the computation in the pixel and batch dimensions to for further speedup. Every square represents a 16×16 block of pixels.

- We show that Grendel enables high-resolution large scale scene rendering: we use 16 GPUs and render 4K images for large-scale Rubble scene from MegaNERF (Turki et al., 2022). For this scene, Grendel uses 40.4 million Gaussians to achieve a PSNR of 27.28, outperforming the current state-of-the-art. The memory required exceeds a single GPU's capacity, making it difficult to render this scene at this quality without Grendel's techniques.

2 Gaussian Splatting: Background, Opportunities and Challenges

3D Gaussian Splatting (Kerbl et al., 2023) (3DGS) is a rendering method that represents 3D scenes using a (potentially large) set of anistropic 3D Gaussians. Each 3D Gaussian is represented by four learnable parameters: (a) its 3D position $x_i \in \mathbb{R}^3$; (b) its shape described by a 3D covariance matrix computed using the Guassian's scaling vector $s_i \in \mathbb{R}^3$ and rotation vector $q_i \in \mathbb{R}^4$; (c) its opacity $\alpha_i \in \mathbb{R}$; and (d) its spherical harmonics $sh_i \in \mathbb{R}^{48}$. The color contribution of each Gaussian is detemined by these parameters and by the viewing-direction.

2.1 Background on 3D Gaussian Training

To train 3DGS, the user provides an initial point cloud (may be random or estimated) for a scene and a set of posed images from different angles. The training process initializes Gaussians using the point cloud. Each training step selects a random camera view and uses the current Gaussian parameters to render the view. It then computes loss by comparing the rendered image to the ground truth, and uses back-propagation to update the Gaussian parameters. The training process also uses an adaptive densification mechanism to add Gaussians to under-reconstructed areas, by cloning or splitting existing ones based on their position variance and scale threshold, with more details in A.1.

Concretely, the training pipeline consists of four steps: Gaussian transformation, image rendering, loss calculation, and backpropagation. Standard approaches to backpropagation are used in this setting, and we detail the remaining three steps below:

1. **Gaussian transformation:** Given a camera view v and the associated screen space, each Gaussian i is transformed and projected to determine its position $x_{v,i} \in \mathbb{R}^2$ on screen, its distance $depth_{v,i} \in \mathbb{R}$ from the screen, and its coverage (or footprint radius) $radius_{v,i} \in \mathbb{R}$. Additionally, the color of each Gaussian $c_{v,i}$ is determined according to the viewing direction using its learnable spherical harmonics coefficients $sh_i \in \mathbb{R}^{48}$.
2. **Rendering:** After Gaussian transformation, the image is rendered by computing each pixel's color. To do so, for a given pixel p, 3DGS first finds all Gaussians that intersect with p. We say that a Gaussian i intersects with p if p lies within $radius_{v,i}$ of the Gaussian i's projected center $x_{v,i}$. Then 3DGS iterates over intersecting Gaussians in increasing depth (i.e. in increasing $depth_{v,i}$) and uses alpha-composition to combine their contributions until a threshold opacity has been reached.
3. **Loss calculation:** Finally, the 3DGS computes the L1 and SSIM loss by comparing the rendered image to the ground truth image. The L1 loss measures the absolute difference between pixel colors, while the SSIM loss measures the similarity between pixel windows. Both metrics are computed per-pixel for both forward and backward implementations.

2.2 Opportunities and Challenges in distributing 3DGS

In designing Grendel for scaling up 3D Gaussian Splatting training, we exploit the following opportunities in the above-described training process and address several challenges:

Opportunity: mixed parallelism. Each of the steps described above is inherently parallel but requires different kinds of work partitioning. In particular, the Gaussian transformation step operates on individual Gaussians and thus should be partitioned by Gaussians. On the other hand, the rendering and loss calculation steps operate on individual pixels(or pixels windows for SSIM loss) and thus should be partitioned by pixel.

Opportunity: spatial locality. Most Gaussians intersect a small contiguous area of the rendered image due to their typically small radius. As illustrated in Figure 3, 90% of the 3D Gaussians in three scenes (Rubble, Bicycle, and Train) have a radius $< 2\%$ of image width. Consequently, a pixel is affected by a small subset of the scene's 3D Gaussians, with significant overlap among neighboring pixels' Gaussians.

Challenge: dynamic and unbalanced workloads. Different image areas intersect varying quantities of Gaussians, as shown in Fig 4. For instance, an image region containing the sky likely corresponds to fewer Gaussians than a region with a person. Additionally, the density, position, shape, and opacity of Gaussians change throughout training. Therefore, the number of Gaussians and their mapping to pixels evolve over time, leading to computational workload imbalances across different image regions and over the training period. Fixed partitioning schemes thus suffer from load imbalance. Refer to the Appendix §A.4 for further details.

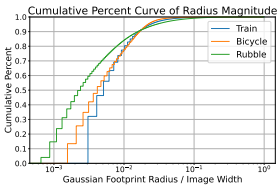


Figure 3: Cumulative percents of $Radius_i$ relative to image width

Challenge: absence of batching. Current 3DGS systems process images one at a time, which suffices for single GPU training. However, as shown in §5, this approach is inefficient in a distributed setting with multiple GPUs. Effective training with larger batch sizes necessitates an understanding of the unique optimization dynamics of 3DGS, which may differ from those of conventional neural networks.

3 System Design

Here, we describe how Grendel exploits the mixed parallelism and spatial locality of 3DGS (§3.1) to address the challenge of dynamic and unbalanced workloads (§3.2).

3.1 Mixed parallelism training

Figure 2(b) provides an overview of Grendel's design. Grendel distributes work according to 3DGS' *mixed parallelism*: it uses Gaussian-wise distribution—where each GPU operates on a disjoint subset of Gaussians–for the Gaussian transformation step, and pixel-wise distribution–where each GPU operates on a disjoint subset of pixels— for the image rendering and loss computation step. The *spatial locality* characteristic allows Grendel to benefit from sparse all-to-all communication when transitioning between these stages.

Gaussian-wise Distribution. Grendel partitions the Gaussians, including their parameters and optimizer states, and distributes them uniformly across GPUs. Then, each GPU independently computes the Gaussian transformation for the set of 3D Gaussians assigned to it. We found that the amount of computation required does not significantly vary across Guassians, and thus evenly distributing Gaussians across GPUs allows us to fit the maximal number of Gaussians while speeding up computation linearly for this step.

Pixel-wise Distribution. We distribute contiguous image areas across GPUs for the image rendering and loss computation steps. Distributing contiguous areas allows us to exploit spatial locality and reduce the number of Gaussians transferred among GPUs. In our implementation, we partition each image in a batch by dividing it into 16×16-pixel blocks, serializing the blocks, and then distributing consecutive subsequences of blocks to different GPUs using an adaptive strategy (§3.2). For batching, each GPU can be assigned blocks from different images in a batch, as shown in Figure 2(b).

Transferring Gaussians with sparse all-to-all communication. To render an image pixel, a GPU needs access to Gaussians that intersect the pixel, which cannot be pre-determined as they are view-dependent and change during training. Therefore, Grendel includes a communication step after the Gaussian transformation. As 3DGS exhibits spatial locality, each pixel partition only requires a small subset of all 3D Gaussians. We leverage this to reduce communication: each GPU first decides the set of intersecting Gaussians for rendering a pixel partition (Figure 5) before using a sparse all-to-all communication to retrieve Gaussians intersecting with any pixels in the partition. A reversed all-to-all communication is done during the backward pass.

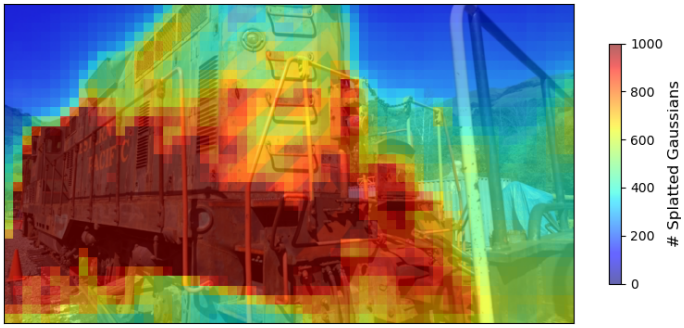


Figure 4: We present a heatmap of the per-tile imbalance in the number of rendered Gaussians on the Train Dataset (Knapitsch et al., 2017). Redder tiles indicate more Gaussian splats. Distant, low-detail areas like the sky need fewer Gaussians than detailed foreground regions like the train, highlighting an imbalance in rendering intensity.

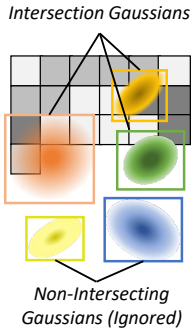


Figure 5: Each GPU only considers Gaussians whose footprints intersect with its assigned pixel render area.

Although Grendel's design bears some resemblance to FSDP (Zhao et al., 2023) used for distributed neural network training, there are important differences. Firstly, unlike weight sharding in FSDP, Gaussian-wise distribution in Grendel is not merely for storage but for also for computation (the Gaussian transformation). Secondly, unlike FSDP which transfers weight shards using the dense all-gather communication, Grendel transfers only relevant Guassians using sparse all-to-all communication.

3.2 Iterative Workload Rebalancing

Pixel-wise Distribution Rebalancing. As discussed in §2.2, the computational load of rendering a pixel varies across space (different pixels) and time (different training iterations). Thus, unlike in distributed neural network training, a uniform or fixed distribution cannot guarantee balanced workloads, so an adaptive pixel distribution strategy is needed.

We record the rendering time of each pixel of each training image during every epoch after the first few. Since the scene generally changes smoothly between consecutive epochs during training, the rendering time of each pixel also changes slowly. Therefore, the rendering times from previous epochs form a good estimate of a pixel's rendering time in the current epoch. Based on this estimate, we can adaptively assign pixels to different GPUs such that the workloads are approximately balanced.

Specifically, Grendel measures the running time (including image rendering, loss computation, and the corresponding backward computation) of each block of pixels assigned to a GPU, computes the average per-pixel computation time for the GPU, and uses this average to approximate the computation time for any pixel p assigned to the GPU. For example, if a GPU is assigned pixels p_0 through p_n, and takes time t for all of these pixels, then Grendel assumes that pixel p_i where $i \in [0, n]$ requires $\frac{t}{n}$ time for computation. In subsequent iterations, the image is re-split so that the sum of the computation time for pixels assigned to all GPUs are equal. In our implementation, we use 16×16 pixel blocks as the split granularity. We show the pseudocode (Algorithm 1) for calculating the Division Points to split an image into load-balanced subsequences of blocks.

Algorithm 1 Calculation of Division Points

Require: ET_j, B (number of pixel blocks), G (number of GPUs)
Ensure: DP (division points)

```
CT ← TORCH.CUMSUM(ET)                          ▷ Cumulative sum of ET
ET_gpu ← CT[B − 1]/G                           ▷ ET is Estimated Running Time per GPU
TH ← TORCH.ARANGE(0, G) · ET_gpu               ▷ Thresholds for Division Points
DP ← TORCH.SEARCHSORTED(CT, TH)                ▷ Division Points
return DP
```

Gaussian-wise Distribution Rebalancing. When training starts, we distribute 3D Gaussians uniformly among the GPUs. As training progresses, new Gaussians are added by cloning and splitting existing ones(§2.1). Newly added Gaussians make the distribution imbalanced as different Gaussians densify at different rates that depend on the scene's local details. Therefore, we redistribute the 3D Gaussians after every few densification steps to restore uniformity.

4 SCALING HYPERPARAMETERS FOR BATCHED TRAINING

To efficiently scale to multiple GPUs, Grendel increases the batch size beyond one, enabling partitioning of both images and pixels within each image, as shown in Figure 2(b).

However, increasing the batch size without adjusting hyperparameters, particularly the learning rate, can result in unstable and inefficient training (Goyal et al., 2017; Qiao et al., 2021), and hyperparameter tuning is often tedious. Though some methods simplify learning-rate tuning for deep neural networks, they either build on SGD (Goyal et al., 2017) (we use Adam) or they leverage the layer-wise structure of neural networks (Ginsburg et al., 2018; You et al., 2020) (3DGS is not neural network). We propose an automatic hyperparameter scaling rule for the Adam optimizer to enable hyperparameter-tuning-free batched 3DGS training. Our result is driven by the *Independent Gradients Hypothesis* unique to 3DGS. Interestingly, our proposed square-root scaling rule coincides with recent works that study Adam learning rate scaling for neural network training, despite differing assumptions (Granziol et al., 2022; Li et al., 2024c).

We propose to scale Adam's **learning rate** and **momentum** based on batch size as follows:

$$\lambda' = \lambda \times \sqrt{\text{batch_size}} \tag{1}$$

$$\beta_1', \beta_2' = \beta_1^{\text{batch_size}}, \beta_2^{\text{batch_size}} \tag{2}$$

where λ is the original learning rate, and β_1, β_2 are the original first and second moments in Adam. $\lambda', \beta_1', \beta_2'$ are the adjusted hyperparameters to work with a greater batch size. We refer to these as the square-root learning rate scaling and the exponential momentum scaling rules.

Independent Gradients Hypothesis. To derive these scaling rules, we first consider 3D GS training in a simplified setting, assuming that gradients calculated from each camera view are *independent* of those induced from other views. Consequently, if we are given a batch of b camera views, taking b sequential gradient descent steps for each view in the batch is equivalent to taking one bigger step where the gradients are summed together. If we were using the vanilla gradient descent algorithm and averaging the gradients in a batch, setting the learning rate to scale linearly with the batch size achieves this equivalence. However, 3D GS uses Adam, an adaptive learning rate optimizer that (1) divides the gradients by the square root of the per-parameter second moment estimate, and (2) uses momentum to combine current gradients and past gradients in an exponential-moving-average fashion, making a bigger update different from simply summing up smaller batch-size-one updates. Under the *independent gradients hypothesis*, we derive the following corrections to Adam hyperparameters to approximate batch-size-one training with a larger batch:

Let us denote g_k as the gradient of some parameter evaluated at view k, and $g = \frac{\sum_{j \in V} g_j}{|V|}$ as the full-batch gradient (mean of gradients across views), where V is the set of all views. Let us further assume $\mathbb{E}[g_k] = 0$ for all k. By the independence assumption: $\text{Cov}(g_k, g_j) = \mathbb{E}[(g_k - 0)(g_j - 0)] = 0 \quad \text{when} \quad k \neq j \quad \text{and} \quad \mathbb{E}[(g_k)^2] \quad \text{when} \quad k = j$.

Then, parameter update from a batch-size-1 Adam step (without momentum) on view k is:

$$\Delta^{\{k\}} = \frac{g_k}{\sqrt{\mathbb{E}\left[\mathbb{E}_{j \in V}\left[g_j^2\right]\right]}} = \frac{g_k}{\sqrt{\mathbb{E}\left[|V| g^2\right]}} = \frac{g_k}{\sqrt{|V|}\sqrt{\mathbb{E}\left[g^2\right]}}.$$

However, the parameter update from one Adam step (without momentum) on a batch of views $B \subseteq V$ of size b is:

$$\Delta^{\{B\}} = \frac{\sum_{k \in B} g_k / b}{\sqrt{\mathbb{E}\left[\mathbb{E}_{B' \subseteq V}\left[\left(\sum_{j \in B'} g_j / b\right)^2\right]\right]}} = \frac{\sum_{k \in B} g_k / b}{\sqrt{\mathbb{E}\left[\frac{|V|}{b} g^2\right]}} = \frac{\sum_{k \in B} g_k / b}{\sqrt{\frac{|V|}{b}}\sqrt{\mathbb{E}\left[g^2\right]}} = \frac{1}{\sqrt{b}} \frac{\sum_{k \in B} g_k}{\sqrt{|V|}\sqrt{\mathbb{E}\left[g^2\right]}}.$$

Thus, setting the learning rate $\lambda' = \lambda \times \sqrt{b}$ allows the batch update $\Delta^{\{B\}}$ to match with the total individual updates $\sum_{k \in B} \Delta^{\{k\}}$. Alongside the square-root learning rate scaling (Eq 1), we also propose an exponential momentum scaling to accommodate larger batches (Eq 2). Initially used by Busbridge et al. (2023), this rule scales the momentum parameters with $\beta' = \beta^{\text{batch_size}}$, which exponentially decreases the influence of past gradients when the batch size increases.

We wish to stress that in the real world, even though some cameras share similar poses, a set of random cameras generally observe different parts of a scene, hence the gradients in a batch are mostly sparse and can be thought of as roughly independent. We empirically study the independent gradient hypothesis and evaluate our proposed scaling rules.

4.1 EMPIRICAL EVIDENCE OF *Independent Gradients*

To see if the *Independent Gradients Hypothesis* holds in practice, we analyze the average per-parameter variance of the gradients in real-world settings. We plot the sparsity and variance of the gradients of the diffuse color parameters starting at pre-trained checkpoints on the "Rubble" dataset (Turki et al., 2022) against the batch size in Figure 6. We find that the inverse of the variance increases roughly linearly, then transitions into a plateau. We find this behavior in all three checkpoint iterations, representing early, middle, and late training stages. The initial linear increase of the precision suggests that gradients are roughly uncorrelated at batch sizes used in this work (up to 32) and supports the *independent gradients hypothesis*. While a single image may have sparse gradients; in a large batch, gradients overlap and become less sparse. They also grow more correlated, as camera with similar poses are expected to offer similar gradients.

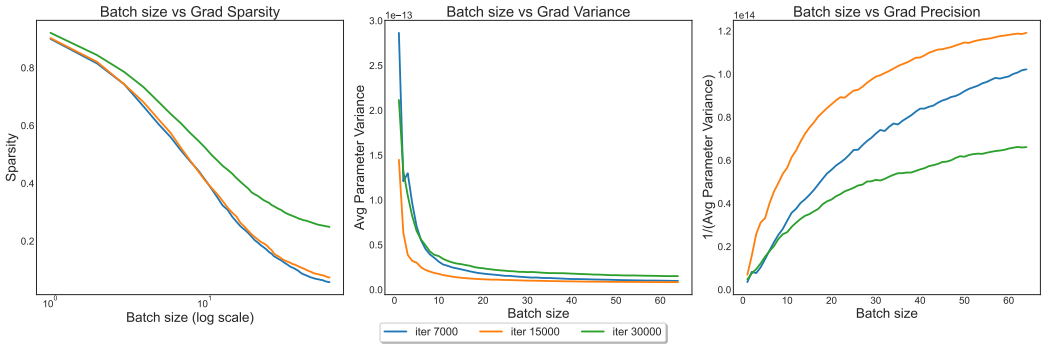


Figure 6: Gradients are roughly uncorrelated in practice. On the "Rubble" dataset (Turki et al., 2022), the inverse of the average parameter variance increases linearly, then rises to a plateau, suggesting that the gradients are roughly uncorrelated initially but become less so as the batch size becomes large. Averaged over 32 random trials.

4.2 EMPIRICAL TESTING OF PROPOSED SCALING RULES

To empirically validate the proposed learning rate and momentum scaling rules, we train the "Rubble" scene up to iteration 15,000 using a batch size of 1. Then, we reset the Adam optimizer states and continue training with different batch sizes. We compare how well different learning rate and momentum scaling rules maintain a similar training trajectory when switching to larger batch sizes in Figure 13(Appendix B.3). Without loss of generality, we focus on the diffuse color parameters for this analysis. Figure 13a compares three different learning rate scaling rules $\in$ [constant, sqrt, linear] where only our proposed "sqrt" holds a high update cosine similarity and a similar update magnitude across different training batch sizes. Similarly, 13b shows our proposed exponential momentum scaling rule keeps update cosine similarity higher than the alternative which leaves the momentum coefficients unchanged.

5 EVALUATION

Our evaluation aims to demonstrate Grendel's scalability, showing both that it can render high-resolution images from large scenes, and that its performance scales with additional hardware resources. And we include results from abalation study to demonstrate the importance of Grendel's dynamic load balancing. The ablation study for the learning rate scaling strategies have already been discussed in 4.2, along with our analysis.

5.1 SETTING AND DATASETS

Experimental Setup. We conducted our evaluation in the Perlmutter GPU cluster NERSC. Each node we used was equipped with 4 A100 GPUs with 40GB of GPU memory, and interconnected with each other using 25GB/s NVLink per direction. Servers were connected to each other using a 200Gbps Slingshot network.

Datasets. We evaluate Grendel using the datasets and corresponding resolution settings shown in Table 1. Of these, Rubble and MatrixCity Block_All represent the large scale datasets that are out of reach for most existing 3DGS systems, while other datasets are commonly used in 3DGS papers. These datasets vary in area size and resolution to comprehensively test our system.

Dataset	# Scenes	Resolutions	# Images	Test Set Setting
Tanks & Temple (Knapitsch et al., 2017)	2	$\sim$ 1K	251 to 301	1/8 of all images
DeepBlending (Hedman et al., 2018)	2	$\sim$ 1K	225 to 263	1/8 of all images
Mip-NeRF 360 (Barron et al., 2022)	9	1080P	100 to 330	1/8 of all images
Rubble (Turki et al., 2022)	1	4591 $\times$ 3436	1657	official test set
MatrixCity Block_All (Li et al., 2023)	1	1080P	5620	official test set

Table 1: Scenes used in our evaluation: We cover scenes of varying sizes and resolutions.

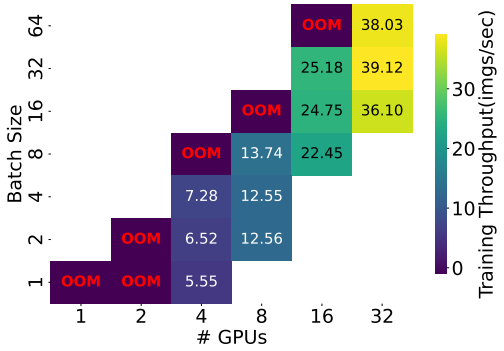


Figure 7: To avoid OOM, 4 GPUs are needed to train the large 4K "Rubble" scene. We further improve throughput by distributing across even more GPUs and increasing the batch size.

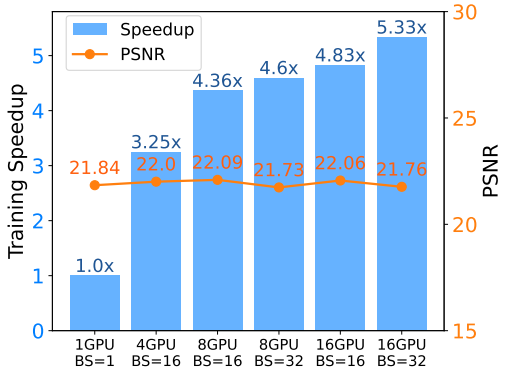


Figure 8: Even for the small 'Train' scene, our distributed training with larger batch sizes achieves speedup without compromising test PSNR. All configs train for 30K total images.

Evaluation Metrics. We report image quality using SSIM, PSNR and LPIPS values, and throughput in training images per second. We take both forward and backward time into consideration of throughput. And note that throughput in images per second may differ from throughput in iterations per second, as one iteration includes the batch size number of images.

5.2 Performance and Memory Scaling

We start by evaluating Grendel's scaling, and how additional GPUs impact computation performance and memory.

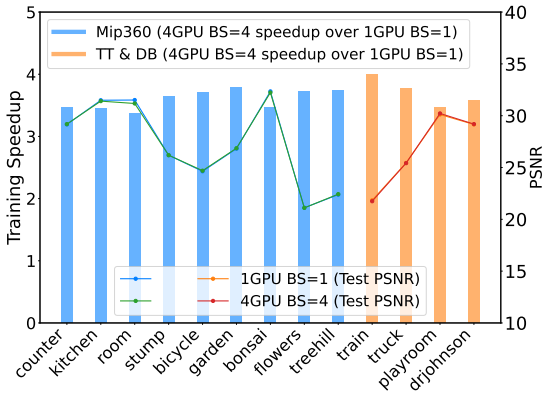


Figure 9: Training Speedup and PSNR on Mip-NeRF360 and Tanks&Temples+Deep Blending.

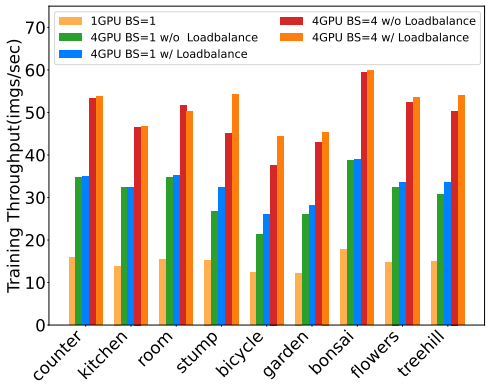


Figure 10: Speedup from Iterative Load Balancing and increased batch sizes on Mip-NeRF360.

Computation. We evaluated how additional GPUs impact Grendel's performance using both large-scale (Rubble) and small-scale (Train and Mip-Nerf360) datasets.

We used the Rubble scene to evaluate the training throughput. For this experiment we used 35 million Gaussians which have been trained to convergence. Because of the time required to render 4K images

for this scene, we measured throughput for training over another 10,000 images times, and in Figure 7 we report throughput (in images per second) as we vary the number of GPUs (x-axis) and batch size (y-axis). We observe that we cannot render this scene with a single GPU (regardless of batch size) because of its memory requirements. Furthermore, both increasing the number of GPUs and increasing batch size yield performance improvements: performance increases from 5.55 images per second (4 GPUs, batch size 1) to 38.03 images per-second (32 GPUs, batch size 64).

Next, we use the 980×545 resolution Train scene to evaluate both throughput and image quality during scaling. Due to its small, low-resolution nature, it can be trained from scratch for each experiment. Our results in Figure 8 show that additional GPUs improve throughput while maintaining image quality when trained with the same total number of images. Notably, our 16-GPU setup with a batch size of 32 completes training on 30K images in just 2 minutes and 42.97 seconds, representing the state-of-the-art training speed to the best of our knowledge.

As shown in Figure 9, we also achieve a 3x to 4x speed up using 4 GPU and a batch size of 4, without PSNR degradation across 13 scenes from the Mip-Nerf360 dataset (first half) and the Tanks & Temple and Deep Blending datasets (second half). We use default hyperparameters from the 3DGS repository (Kerbl et al., 2023). We train on the same number of images: 50k for Mip-NeRF 360 and 30k for the slightly smaller TT & DB datasets, to ensure convergence and a fair comparison.

Memory Scaling. Scaling the number of GPUs increases memory, allowing more Gaussians to represent a scene. We tested this by adding Gaussians through densification until we ran out of memory. Figure 14 (In Appendix C.2) illustrates the number of Gaussians that Grendel can accommodate (in millions) with batch sizes of 1, 4, and 16, as the number of GPUs increases. The results demonstrate linear scaling. In §5.2.1 we show the utility of using additional Gaussians.

We provide additional details about experiments from this section in Appendix C.2.

5.2.1 Gaussian Quantity vs. Reconstruction Quality

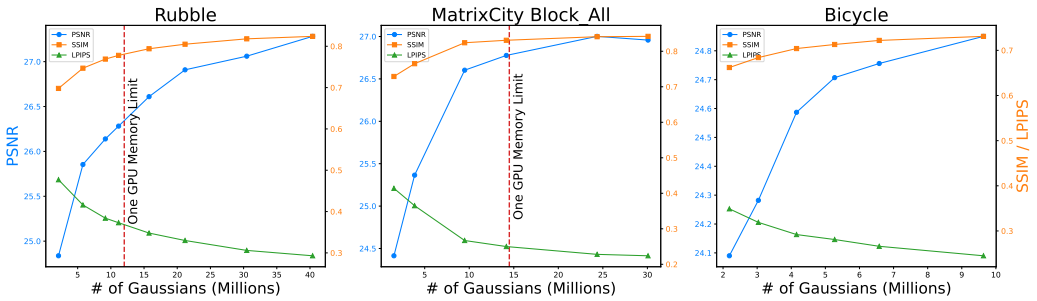


Figure 11: Scalability Statistics: Gaussian Quantity vs. Reconstruction Quality
Using more Gaussians results in better test metrics for reconstruction. The red line indicates the number of Gaussians a single GPU can handle, which is insufficient for achieving high-quality results.

Scaling to multiple GPUs allows Grendel to use a larger number of Gaussians to represent scenes. A larger number of Gaussians can capture fine grained scene details, and should thus be able to better reconstruct large-scale, high-resolution scenes. We evaluated this effect using three scenes: Rubble, MatrixCity Block_All, and Bicycle, and varied the number of Gaussians used by changing densification settings: we lowered the gradient norm threshold to initiate densification and reduced the threshold for splitting Gaussians instead of cloning until the densification mechanism produced the target number of Gaussians without manual interference. We rendered Rubble and Matrix City Block_All using 16 GPUs and a batch size of 16, while we used 4 GPUs and a batch size of 4 for Bicycle. The difference in number of GPUs and batch sizes is due to differences in scene sizes: bicycle is much smaller than the other two datasets.

In Figure 11, we show that image quality metrics (PSNR, SSIM and LPIPS) improve as we add more Gaussians. The red line in the Rubble and Matrix City Block_All graphs shows the number of Gaussians that can fit on a single GPU (Bicycle, being smaller, can be rendered on a single GPU). Figure 12 shows the rendered images as we scale Gaussians quantity, and demonstrates the quality improvements are human visible. These results demonstrate benefits of using more Gaussians, and demonstrate the necessity of multi-GPU 3DGS training systems like Grendel.

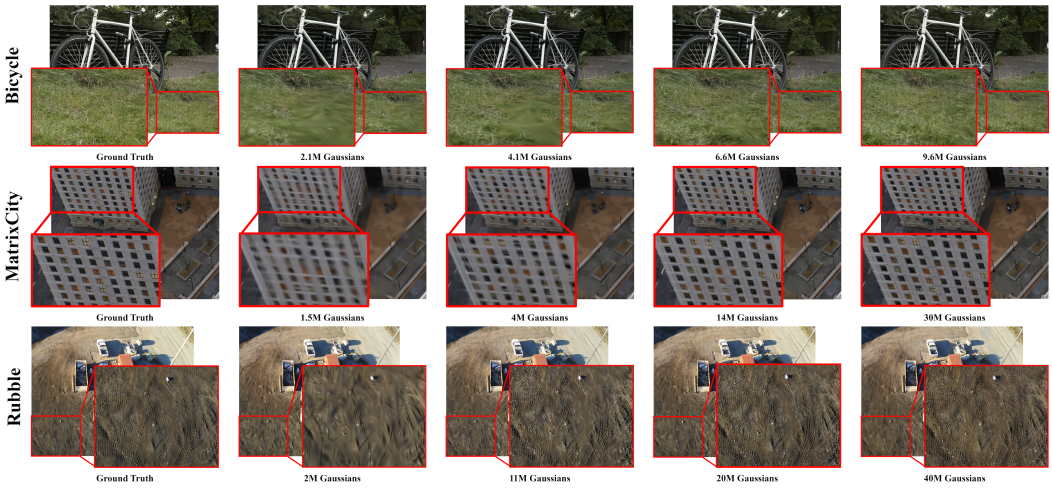


Figure 12: Visualization: Gaussian Quantity vs. Reconstruction Quality

5.3 Ablation Study

Figure 10 illustrates that our load balancing techniques and increased batch size significantly improve training throughput on 1080p Mip-NeRF360 dataset, compared to the one GPU baseline and our straightforward distributed system with a conventional batch size of one and no load balancing. Similar results are observed with the 4K Rubble Dataset, as shown in Figure 15 (In Appendix C.2). Although good speed can be achieved without load balancing, load balancing allows us to consistently achieve even higher throughput across various types and scales of scenes.

6 Related Works

Large-scale scene reconstruction. Prior works have proposed the divide-and-conquer approach to scale 3DGS to work with large scenes. VastGaussian (Lin et al., 2024), CityGaussisan (Liu et al., 2024), and Hierarchical Gaussian (Kerbl et al., 2024) divide large scenes into small regions and train each region separately, and then merge the resulting sub-models. Hierarchical Gaussian (Kerbl et al., 2024), Octree-GS (Ren et al., 2024) and CityGaussisan (Liu et al., 2024) describe level-of-detail based approaches to adaptively reduce the number of Gaussians considered for distant objects. Grendel's system-level parallelization is complementary to these algorithmic innovations. For example, the initial coarse training step in both CityGaussisan (Liu et al., 2024), and Hierarchical Gaussian (Kerbl et al., 2024) can utilize Grendel for multi-GPU acceleration. Similar methods (Turki et al., 2022; Yuanbo et al., 2022; Li et al., 2024b) have been employed to scale NeRF, but they are not directly applicable to 3DGS due to its distinct computation pattern, as discussed in Section 1.

Distributed Training for 3DGS. DOGS (Chen & Lee, 2024) modifies training with ADMM distributed optimization, averaging Gaussians shared across partition boundaries every 100 iterations. This method enables asynchronous training, which can potentially impact the convergence rate. In contrast, Grendel preserves the original 3DGS algorithm, maintaining the same convergence characteristics as the single-GPU version. RetinaGS (Li et al., 2024a) also retains the original 3DGS algorithm but employs a distinct parallelization strategy. In RetinaGS, each GPU renders the entire image based on its local partition of Gaussians, with the rendered outputs subsequently merged. However, this approach results in redundant computations, as many GPUs render pixels beyond the opacity saturation depth unnecessarily. Related works on distributed training for neural networks are further discussed in Appendix D.

Large Batch Size Training. Large batch training has bee widely adopted to improve the ML training performance and efficiency, but it has also been recognized by prior work (Keskar et al., 2017) that increasing batch size can adversely impact model performance. This has led to the development of empirical rules, including linear scaling and learning rate warmum (Goyal et al., 2017), and layer-wised adaptive rate scaling (You et al., 2020; Ginsburg et al., 2018). Theoretical work (Li et al., 2021; Malladi et al., 2022; Busbridge et al., 2023) have also predicted and validated scaling laws for increasing batch sizes. Our work is inspired by these, but focuses on batch size scaling for 3DGS.

REPRODUCIBILITY STATEMENT

We provide hyperparameter details in Appendix C.2. We will also release code for training, rendering, and evaluation. We will release pre-trained model checkpoints for Rubble and MatrixCity datasets to reproduce the results reported in the paper.

REFERENCES

Jonathan Barron, Ben Mildenhall, Dor Verbin, Pratul Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *CVPR*, 2022. doi: 10.1109/CVPR52688.2022.00539.

Dan Busbridge, Jason Ramapuram, Pierre Ablin, Tatiana Likhomanenko, Eeshan Gunesh Dhekane, Xavier Suau, and Russell Webb. How to scale your EMA. In *NeurIPS*, 2023. URL https://openreview.net/forum?id=DkeeXVdQyu.

Yu Chen and Gim Hee Lee. Dogs: Distributed-oriented gaussian splatting for large-scale 3d reconstruction via gaussian consensus, 2024. URL https://arxiv.org/abs/2405.13943.

Boris Ginsburg, Igor Gitman, and Yang You. Large batch training of convolutional networks with layer-wise adaptive rate scaling, 2018. URL https://openreview.net/forum?id=rJ4uaX2aW.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Diego Granziol, Stefan Zohren, and Stephen Roberts. Learning rates as a function of batch size: A random matrix theory approach to neural network training. *Journal of Machine Learning Research*, 23(173):1–65, 2022.

Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 2018.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS*, 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), July 2023. URL https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/.

Bernhard Kerbl, Andreas Meuleman, Georgios Kopanas, Michael Wimmer, Alexandre Lanvin, and George Drettakis. A hierarchical 3d gaussian representation for real-time rendering of very large datasets. *ACM Transactions on Graphics*, 2024. URL https://repo-sam.inria.fr/fungraph/hierarchical-3d-gaussians/.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR*, 2017. URL https://openreview.net/forum?id=H1oyRlYgg.

Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics*, 2017.

Bingling Li, Shengyi Chen, Luchao Wang, Kaimin Liao, Sijie Yan, and Yuanjun Xiong. Retinags: Scalable training for dense scene rendering with billion-scale 3d gaussians, 2024a. URL https://arxiv.org/abs/2406.11836.

Ruilong Li, Sanja Fidler, Angjoo Kanazawa, and Francis Williams. Nerf-xl: Scaling nerfs with multiple gpus, 2024b.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. In *VLDB*, 2020.

Shuaipeng Li, Penghao Zhao, Hailin Zhang, Xingwu Sun, Hao Wu, Dian Jiao, Weiyan Wang, Chengjun Liu, Zheng Fang, Jinbao Xue, et al. Surge phenomenon in optimal learning rate and batch size scaling. *arXiv preprint arXiv:2405.14578*, 2024c.

Yixuan Li, Lihan Jiang, Linning Xu, Yuanbo Xiangli, Zhenzhi Wang, Dahua Lin, and Bo Dai. Matrixcity: A large-scale city dataset for city-scale neural rendering and beyond. In *ICCV*, 2023.

Zhiyuan Li, Sadhika Malladi, and Sanjeev Arora. On the validity of modeling SGD with stochastic differential equations (SDEs). In *NeurIPS*, 2021. URL `https://openreview.net/forum?id=goEdyJ_nVQI`.

Jiaqi Lin, Zhihao Li, Xiao Tang, Jianzhuang Liu, Shiyong Liu, Jiayue Liu, Yangdi Lu, Xiaofei Wu, Songcen Xu, Youliang Yan, and Wenming Yang. Vastgaussian: Vast 3d gaussians for large scene reconstruction. In *CVPR*, 2024.

Yang Liu, He Guan, Chuanchen Luo, Lue Fan, Junran Peng, and Zhaoxiang Zhang. Citygaussian: Real-time high-quality large-scale scene rendering with gaussians. In *CVPR*, 2024.

Tao Lu, Mulin Yu, Linning Xu, Yuanbo Xiangli, Limin Wang, Dahua Lin, and Bo Dai. Scaffold-gs: Structured 3d gaussians for view-adaptive rendering. In *CVPR*, 2024.

Sadhika Malladi, Kaifeng Lyu, Abhishek Panigrahi, and Sanjeev Arora. On the SDEs and scaling rules for adaptive gradient algorithms. In *NeurIPS*, 2022. URL `https://openreview.net/forum?id=F2mhzjHkQP`.

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *SOSP*, 2019.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *SOSP*, 2021.

NERSC. Perlmutter architecture. URL `https://docs.nersc.gov/systems/perlmutter/architecture/`. Accessed: 2024-05-22.

Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *OSDI*, 2021.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC*, 2020.

Kerui Ren, Lihan Jiang, Tao Lu, Mulin Yu, Linning Xu, Zhangkai Ni, and Bo Dai. Octree-gs: Towards consistent real-time rendering with lod-structured 3d gaussians. *arXiv preprint arXiv:2403.17898*, 2024.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. In *SC*, 2020.

Haithem Turki, Deva Ramanan, and Mahadev Satyanarayanan. Mega-nerf: Scalable construction of large-scale nerfs for virtual fly-throughs. In *CVPR*, 2022.

Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *EuroSys*, 2019.

Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. Gspmd: General and scalable parallelization for ml computation graphs. In *arXiv:2105.04663*, 2021.

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. In *ICLR*, 2020. URL `https://openreview.net/forum?id=Syx4wnEtvH`.

Xiangli Yuanbo, Xu Linning, Pan Xingang, Zhao Nanxuan, Rao Anyi, Theobalt Christian, Dai Bo, and Lin Dahua. Bungeenerf: Progressive neural radiance field for extreme multi-scale scene rendering. In *ECCV*, 2022.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023.

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *OSDI*, 2022.

A Additional Preliminaries & Observations Details

This appendix provides additional information about 3DGS, beyond what was covered in §2.

A.1 Densification Process

Densification is the process by which 3DGS adds more Gaussians to improve details in a particular region. A Gaussian that shows significant position variance across training steps, might either be clones or split. The decision on whether to clone or split depends on whether their scale exceeds a threshold. Hyperparameters determine the start and stop iteration for densification, its frequency, the gradient threshold for initiating densification, and the scale threshold that determines whether to split or clone. To create more Gaussians, we need to increase the stop iteration and frequency, and decrease the gradient threshold for densification. If we aim to capture more details using smaller Gaussians, we should lower the scale threshold to split more Gaussians. The training process also includes pruning strategies such as eliminating Gaussians with low opacity and using opacity reset techniques to remove redundant Gaussians.

A.2 Z-buffer

The indices of intersecting gaussians for each pixel are stored in a Z-buffer, used in both forward and backward. This Z-buffer is the switch between View-dependent Gaussian Transformation and Pixel Render. Since a single gaussian can project onto multiple pixels within its footprint, the total size of all pixels' Z-buffers exceeds both the count of 3DGS and pixels. The Z-buffer itself, along with auxiliary buffers needed for sorting it, etc, consumes significant activation memory. This can also lead to out-of-memory (OOM) errors if the resolution, scene size, or batch size is increased.

A.3 Mixed Parallelism

In the main text, some steps of 3DGS are not mentioned, but these steps can also be parallelized. The Gaussian transformation backward and gradient updates by the optimizer are also Gaussian-wise computations and will be distributed the same way as the Gaussian transformation forward. Similarly, the Render Backward and Loss Backward computations are pixel-wise and will be distributed just like the Render Forward.

Regarding the memory aspect, each Gaussian has independent transformed states, gradients, optimizer states, and parameters. Therefore, we save these states together on the corresponding GPU that contains their parameters. And activation states like significant Z-buffers, auxiliary buffers for sorting and other functions, loss intermediate activations are managed pixel-wise along with the image distribution.

Regarding densification mechanism, since we clone, split or prune Gaussians independently based on their variance, we perform this process locally on the GPU that stores them.

A.4 Dynamic Unbalanced Workloads

Physical scenes are naturally sparse on a global scale. Different areas have different densities of 3D gaussians (i.e sky and a tree). Thus, the intensity of rendering not only varies from pixel to pixel within an image but also differs between various images, leading to workloads unbalance. Figure 4 shows the differences in render intensity across the image.

Besides, during the training, gaussians parameters are continuously changing. More precisely, the change of 3D position parameters and co-variance parameters affect each gaussian's coverage of pixels on the screen. The change of opacity parameters affect the number of gaussians that contribute to each pixel. Both of them lead to render intensity change. The densification process targets areas under construction. During training, simpler scene elements are completed first, allowing more complex parts to be progressively densified. This means Gaussians from different regions densify at varying rates. The dynamic nature of the workloads is more pronounced at the beginning of training, as it initially focuses on constructing the global structure before filling in local details.

The different computational steps have distinct characteristics in terms of workload dynamicity. Even though, the rendering computation is dynamic and unbalanced; computation intensity for loss calculation remains consistent across pixels, and the view-dependent transformation maintains a uniform computational intensity across gaussians. Actually, render forward and backward have different patterns of unbalance and dynamicity. The computational complexity for the forward process scales with the number of 3DGS intersecting the ray. In contrast, the complexity of the backward process depends on Gaussians that contributed to color and loss before reaching opacity saturation, typically those on the first surface. Then, running time for render forward and backward, loss forward and backward have different dominating influence factors, and every step takes a significant amount of time.

B Additional Design Details

B.1 Scheduling Granularity: Pixel Block Size

In our design, we organize these pixels from all the images in a batch into a single row. Then, we divide this row into parts, and each GPU takes care of one part. However, if there are a lot of pixels, the strategy scheduler computation overhead will be very large. So we group the pixels into blocks of 16 by 16 pixels, put these blocks in a row and allocate these blocks instead. The size of block is essentially the scheduling granularity, which is a trade-off between scheduler overhead and uneven workloads due to additional blocks. After scheduling, we will have a 2D boolean array, compute_locally[i][j], indicating whether the pixel block at i-th row and j-th column should be computed by the local GPU. We will then render only the pixels within the blocks where compute_locally is true.

B.2 Gaussian Distribution Rebalance

An important observation is that distributing pixels to balance runtime doesn't necessarily balance the number of Gaussians each GPU touches in rendering; So, to minimize total communication volume, GPUs may need to store varying quantity of Gaussians based on the formula above. Specifically, only the forward runtime correlates directly with the number of touched 3DGS; however, the time it takes for pixel-wise loss calculations and rendering backward depends on the quantity of pixels and the count of gaussians that are indeed contributed to the rendered pixel color, respectively. In our experiments, random redistribution leads to fastest training here, even if its overall communication volume is not the minimum solution. Because in our experiment setting, we use NCCL all2all as the underlying communication primitive, which prefers the uniform send and receive volume among different GPU. If we change to use communication primitive that only cares about the total communication volume, then we may need to change to other redistribution strategy.

B.3 Scaling Rule Hyperparameter Ablation

The effectiveness of our automatic hyperparameter scaling rules is demonstrated in the ablation study, as shown in Figure 13.

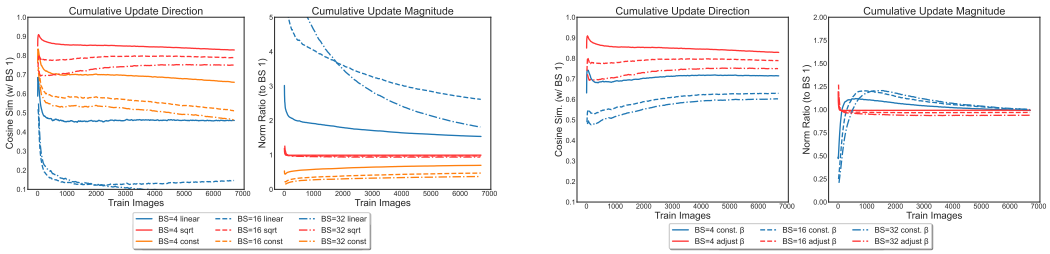
C Additional Experiments Setting and Statistics

C.1 Statistics for Mip-NeRF 360, Tank&Temples and DeepBlending datasets

We provide full statistics of training results on Mip-NeRF 360, Tank&Temples and DeepBlending datasets in Table 2.

C.2 Scalability

Table 3, 4 and 5 show the increased reconstruction quality with more gaussians. While many hyperparameters influence the number of Gaussians created by densification, we focused on adjusting three key parameters: (1) the stop iteration for densification, (2) the threshold for initiating densification, and (3) the threshold for deciding whether to split or clone a Gaussian. Initially, we gradually increased the densification stop iteration to 5,000 iterations. However, due to the pruning mechanism,



(a) Learning rate scaling rules vs. BS invariance.

(b) Momentum scaling rules vs. BS invariance.

Figure 13: We plot the training trajectories of the diffuse color parameters on "Rubble", when training with batch size $\in [4, 16, 32]$ using different learning rate and momentum scaling strategies. Cumulative weight updates using the square-root learning rate scaling rule (a, red curves) and exponential momentum scaling rule (b, red curves) maintain high cosine similarity to batch-size 1 updates and have norms that are roughly invariant to the batch size. All trajectories in (a) employ our proposed exponential momentum scaling but differing learning rate scaling; while all trajectories in (b) employ our proposed square-root learning rate scaling but differing momentum scaling.

Dataset	Scene	1 GPU (bsz=1)		4 GPU (bsz=4)	
		PSNR	Throughput	PSNR	Throughput
Mip-NeRF360	counter	29.16	16.25	29.19	56.24
	kitchen	31.49	14.24	31.40	49.16
	room	31.51	15.82	31.18	53.36
	stump	26.19	14.95	26.19	54.53
	bicycle	24.63	12.01	24.69	44.44
	garden	26.82	12.10	26.86	45.83
	bonsai	32.34	17.87	32.23	61.88
	flowers	21.11	14.47	21.10	53.94
	treehill	22.38	14.78	22.43	55.31
Tank&Temples	train	21.84	34.72	21.75	101.69
	truck	25.44	27.55	25.42	95.85
DeepBlending	playroom	30.11	21.98	30.22	75.38
	drjohnson	29.15	17.74	29.19	62.11

Table 2: Performance Comparison Between Non-Distribution and 4 GPU Distribution

this adjustment alone proved insufficient. Consequently, we also lowered the two thresholds to generate more Gaussians. For a fair comparison, all other densification parameters—such as the interval, start iteration, and opacity reset interval—were kept constant. For the Rubble scene, each experiment run for the same 125 epochs, exposing models to 200,000 images, ensuring consistency. Although training larger models for longer durations and lowering the positional learning rate improved results in my observations, we maintained consistent training steps and learning rates across all experiments to ensure fairness.

Table 6, 7 show the Throughput Scalability by Increasing batch size and leveraging more GPUs, for Rubble and Train scene, respectively. Essentially, more GPUs and larger batch size give higher throughput. More GPUs provide more computational power while larger batch size can utilze these GPUs better.

Table 8 demonstrates that additional GPUs increase available memory for more Gaussians, evaluated on the Rubble scene with various batch sizes reflecting different levels of activation memory usage. We can achieve linear scaling as illustrated in Figure 14. Essentially, more GPUs provide additional memory to store Gaussians, while a larger batch size increases activation memory usage, leaving less memory available for Gaussians.

Experiment	n3dgs	Results			Densification Settings	
		PSNR	SSIM	LPIPS	Stop Iter	Thresholds
EXP 1	2114045	24.84	0.70	0.48	5000	(0.0002, 0.01)
EXP 2	5793396	25.85	0.75	0.42	15000	(0.0002, 0.01)
EXP 3	9173931	26.14	0.77	0.38	50000	(0.0002, 0.01)
EXP 4	11168630	26.28	0.78	0.37	50000	(0.00018, 0.008)
EXP 5	15754744	26.61	0.79	0.35	50000	(0.00015, 0.005)
EXP 6	21177774	26.91	0.80	0.33	50000	(0.00013, 0.003)
EXP 7	30474202	27.06	0.82	0.31	50000	(0.0001, 0.002)
EXP 8	40397406	**27.28**	**0.82**	**0.29**	50000	(0.00008, 0.0016)

Table 3: Scalablity on Rubble: Gaussian Quantity, Results and Hyperparameter Settings

Experiment	n3dgs	Results			Densification Settings	
		PSNR	SSIM	LPIPS	# Start Points	# Densify Iter
EXP 1	1545568	24.41	0.73	0.41	1545568	0
EXP 2	3867136	25.36	0.77	0.36	3867136	0
EXP 3	9485755	26.6	0.82	0.27	7743616	5000
EXP 4	14165332	26.78	0.83	0.25	15540941	5000
EXP 5	24355726	**27.0**	**0.84**	0.23	15540941	30000
EXP 6	30074630	26.96	**0.84**	**0.22**	15540941	40000

Table 4: MatrixCity Block_All Statistics: Gaussian Quantity, Results and Hyperparameter Settings

Experiment	n3dgs	Results			Densification Settings	
		PSNR	SSIM	LPIPS	Stop Iter	Thresholds
EXP 1	2185112	24.09	0.66	0.35	5000	(0.0002, 0.01)
EXP 2	3035508	24.28	0.68	0.32	7000	(0.0002, 0.01)
EXP 3	4154806	24.59	0.70	0.29	10000	(0.0002, 0.01)
EXP 4	5272686	24.71	0.71	0.28	15000	(0.0002, 0.01)
EXP 5	6579244	24.76	0.72	0.27	15000	(0.00018, 0.008)
EXP 6	9636072	**24.85**	**0.73**	**0.25**	15000	(0.00015, 0.005)

Table 5: Bicycle Statistics: Gaussian Quantity, Results and Hyperparameter settings

D Additional Related Works

Distributed training for neural networks. Existing work have exploited various types of parallelism to train neural networks across GPUs. These include data parallelism (Li et al., 2020), tensor parallelism (Shoeybi et al., 2020; Narayanan et al., 2021), pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019) and FSDP (Zhao et al., 2023; Rajbhandari et al., 2020). Several systems also support multiple types parallelism and/or aim to automatically partition the workload to optimize speed (Zheng et al., 2022; Xu et al., 2021; Wang et al., 2019). However, as we discussed earlier (§1), neural network and 3DGS have very different computation patterns. The former performs repeated layer-wise computation dominated by dense matrix multiply operations while the latter's 3 stages training process is irregular and sparse. As a result, although Grendel's distribution strategy may resemble those seen in existing work (e.g., FSDP Zhao et al. (2023)), the details are quite different.

GPU Count	bsz=1	bsz=2	bsz=4	bsz=8	bsz=16	bsz=32	bsz=64
1 GPU	OOM						
2 GPU	OOM						
4 GPU	5.55	6.52	7.28	OOM			
8 GPU		12.56	12.55	13.74	OOM		
16 GPU				22.45	24.75	25.18	OOM
32 GPU					36.10	39.12	38.03

Table 6: Scalability on Rubble: Speed up from More GPU and Larger Batch Size

Experiment	# GPU	Batch Size	Throughput	PSNR
EXP 1	1	1	34.72	21.84
EXP 2	4	16	112.78	22.01
EXP 3	8	16	151.52	22.09
EXP 4	8	32	159.57	21.73
EXP 5	16	16	167.60	22.06
EXP 6	16	32	185.19	21.76

Table 7: Scalability on Train: Speed up from More GPU and Larger Batch Size

GPU Count	bsz=1	bsz=4	bsz=16
1 GPU	12.71 M	7.10 M	OOM
2 GPU	31.40 M	21.80 M	3.91 M
4 GPU	63.44 M	43.48 M	19.55 M
8 GPU	116.85 M	82.31 M	36.44 M
16 GPU	230.41 M	169.37 M	74.98 M
32 GPU	354.46 M	313.10 M	150.21 M

Table 8: Scalability on Rubble: More Available memory with more GPU

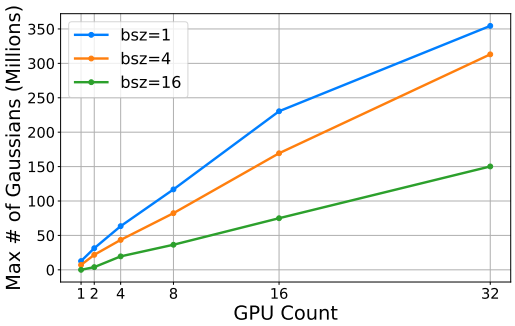


Figure 14: More GPUs provide additional memory to support more Gaussians before encountering OOM.

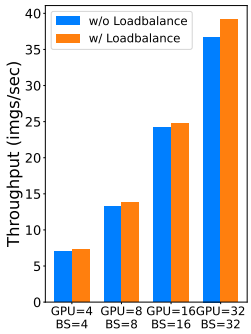


Figure 15: Load balancing and larger batch accelerate training on 4K Rubble Scene