Multi-GPU Parallelization for 3D Gaussian Splatting

Nadula Kadawedduwa
nadulakadawedduwa@gmail.com

Quan Nguyen
nguyqu03@umd.edu

Tan Dat Dao
dtdat@umd.edu

Abstract

3D tomography reconstruction from multiple viewpoints has been a long-standing research area in the computer vision field, and 3D Gaussian Splatting (3DGS) is the state-of-the-art in recent years. It trains a set of 3D Gaussians in space to fit a set of views by back-propagating loss calculated against the input images. The original implementation only utilizes a single GPU, thereby limiting the number of input views and resolution of images to what the memory can hold. We implemented a parallelized version of their original algorithm which resolves these issues by adding support for efficient usage on clusters with access to multiple GPUs. We use two main schema to distribute work and data across said GPUs: Batching and data parallelism across gaussian.

The slides and source code for our project can be accessed via the following links:

- Presentation and source code on Google Drive: `https://drive.google.com/drive/folders/1mbn9__-iktzhJoSr-ARNy-KJEzgprrnp?usp=sharing`.
- Project environment setup guide on GitHub: QuanHNguyen232/cmsc714-group-prj-Spring25.

1 Introduction

3D tomography reconstruction from multiple viewpoints has been a long-standing research area in the computer vision field, and 3D Gaussian Splatting (3DGS) is the state-of-the-art in recent years. The most cited paper which brought gaussian splatting to the forefront of tomography [1], and the implementation which we will be referencing going forwards, released in 2023. Since then, the technology has been iterated and improved upon to facilitate more efficient and specialized use cases (i.e. few-view reconstruction or time-varying scenes). However, there is still more work to be done in the direction of scalability.

1.1 Implementation from SIGGRAPH 2023

As stated previously, the original implementation we iterated on only used a single GPU for its training process. Here we outline the process of the original implementation to make clear what we changed. The original 3D Gaussian Splatting (3DGS) pipeline consists of the following key stages:

1. **Initialization:** A sparse set of 3D Gaussian primitives is initialized in world space. Each Gaussian is parameterized by its position, covariance (scale), opacity, and color.
2. **Rendering:** For each training image (i.e., a viewpoint), the Gaussians are projected into screen space and composited into a 2D image using a differentiable splatting rasterizer. This process is accelerated using CUDA and taking advantage of PyTorch's built-in GPU operations along with custom CUDA extensions.
3. **Loss Evaluation:** The rendered image is compared with the corresponding ground-truth image using a pixel-wise loss function, typically the mean squared error (MSE) or the structural similarity index (SSIM). This loss serves as the supervisory signal for training.

Preprint. Under review.

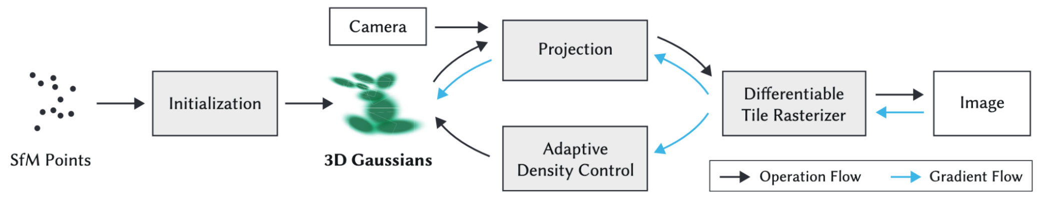

Figure 1: Diagram of gaussian splatting pipeline from *3d gaussian splatting for real-time radiance field rendering* [1]

4. **Backpropagation:** The parameters of each Gaussian (e.g., position, opacity, scale) are updated using gradient descent through PyTorch's autograd engine.

The entire pipeline operates on a single GPU, which simplifies development and debugging but limits scalability. High-resolution scenes or large datasets are constrained by available memory and compute on a single device.

Our work aims to overcome these limitations by developing a multi-GPU parallel training pipeline. This parallelization improves data capacity and throughput and enables reconstruction of larger and higher resolution scenes by distributing the computational workload across multiple GPUs.

An output using this implementation on custom data can be found in the interim report.

2 Implementation

2.1 Batching

The first schema we implemented was batching. The original implementation only rasterized, calculated loss, and back-propagated for a single view per iteration. By rasterizing N views per iteration ($N =$ number of GPUs) and averaging the losses, then back-propagating, we can converge more efficiently. That is to say, we can achieve a higher quality compared to non-batching in the same number of iterations.

While each GPU in our parallelized version follows largely the same operations as the single GPU implementation, they differ in some key places.

1. All images and cameras are loaded onto the rank 0 GPU and scattered to the other processes such that each GPU has a mutually exclusive subset. This reduces the *per-GPU* memory usage, but does not change *overall* memory usage.
2. Each GPU now deals with a different view. Rasterization and loss calculation is done on a per-GPU basis (for that GPU's view), and then all the different losses are combined using an all reduce AVG. Now that every GPU has the same loss, all GPUs will have the same gaussians at the end of the iteration.
   (a) While it is possible to have just one GPU do the back-propagation after gathering and averaging loss to it, the gaussian model would need to by synced at the end of the iteration anyways. The model's size makes this unfeasible and it's more efficient runtime-wise to make each GPU maintain its own copy of the gaussians.

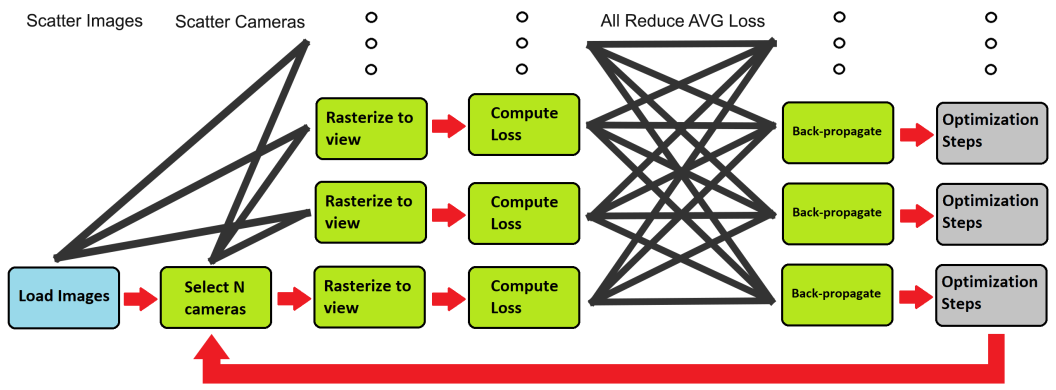

Figure 2: Data flow of the distributed program with batching

2.2 Data-parallelism

Even though batching on its own already drastically increases the size of the dataset we can train on, the granularity of our parallelism is still limited by the resolution of images. Being able to parallelize the work for a single image would allow for a more fine-grained approach and better performance if load-balancing is incorporated.

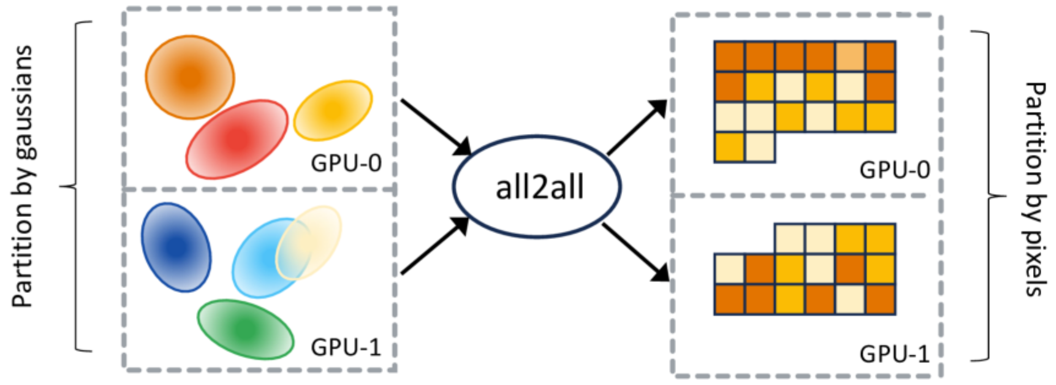

Figure 3: Diagram of gaussian and pixel partitioning from *On scaling up 3d gaussian splatting training* [2]

By splitting up an image into N regions by pixels and partitioning the gaussians based on which subset of pixels them impact, it is possible to split up the processing of a single image among multiple GPUs. This approach relies on the locality of gaussians in that a gaussian contained in one subset of pixels won't have an impact on the pixels of another subset.

Memory usage is further streamlined since only the relevant pixels and gaussians are sent to a GPU. For very large images (4K and above), this fine-grained adaptation could allow for usage of memory space on a GPU which would have otherwise stayed empty because it couldn't fit a full image.

[NOTE]: Our parallel implementation was greatly inspired by [2] and they incorporated dynamic load-balancing into their pipeline. We were not able to get this second scheme or load-balancing working before the submission, but we still saw respectable improvements with just batching.

2.3 Training Configuration

We used the `bicycle` dataset from the official 3D Gaussian Splatting (3DGS) repository for all our experiments. To test the scalability and performance of our parallel implementation, we ran experiments on different image resolutions, ranging from $1/8$ resolution (approximately 720p) to full 4K resolution.

Dataset:

- **Name:** bicycle (COLMAP format)
- **Views:** 194

Training Parameters:

- **Iterations:** Between 3,000 to 7,000
- **Number of GPUs:** 2-4 A100 GPUs (80GB) on Zaratan
- **Batch size:** N (number of GPUs) [1 view per GPU per iteration]
- **Loss Function:** Mean Squared Error (MSE)

Distributed Setup:

- PyTorch's `torch.distributed` with NCCL backend
- Each process runs on a dedicated GPU with synchronized gradient updates
- Experiments launched via `torchrun -nproc_per_node=X train.py ...`

[NOTE]: To reduce memory overhead and improve reproducibility, we fixed the same random seed across all runs and disabled view-dependent optimizations. Results were validated using the original 3DGS rendering and metrics pipeline (`render.py` and `metrics.py`).

3 Results

3.1 Quantitative

Table 1 presents the quantitative results of our multi-GPU implementation compared to the single-GPU baseline. We evaluated our approach using the bicycle dataset at 1/8 resolution. We trained for 6000 iterations for 1, 2, and 4 GPUs, recording the PSNR every 1000 iterations and the total time to train.

Iteration	**PSNR** $\uparrow$	**Time** (seconds)
1 GPU	26.45 dB	171.42
2 GPUs	25.52 dB	166.12
4 GPUs	23.78 dB	221.75

Table 1: Quantitative results training on GPUs.

Figure 4 illustrates the quantitative performance for the first 500 iterations. All configurations show a steady increase in PSNR as training progresses.

We noticed that the single-GPU configuration slightly outperforms others at the 500-iteration mark (20.32 dB), both the 2-GPU (20.31 dB) and 4-GPU (20.18 dB) configurations follow closely. In addition, the 4-GPU configuration shows more fluctuation in early training (e.g., at 100–300 iterations), which may be due to synchronization overhead or less stable gradient aggregation when scaling further.

Overall, the graph confirms that our parallel training implementation retains competitive convergence behavior within the first 500 iterations.

However, as we continue training, according to figure 5, our results show some important findings:

- 1-GPU Performance: The single GPU configuration achieved the highest final PSNR of 26.45 dB after 6000 iterations, which shows steady convergence throughout training.
- 2-GPU Performance: This setup reached 25.52 dB while training 3% faster than single GPU (166.12s vs 171.42s). Although the final PSNR is 0.93 dB lower, the minimal time improvement suggests that the overhead of multi-GPU coordination nearly offsets the parallelization benefits at this scale.

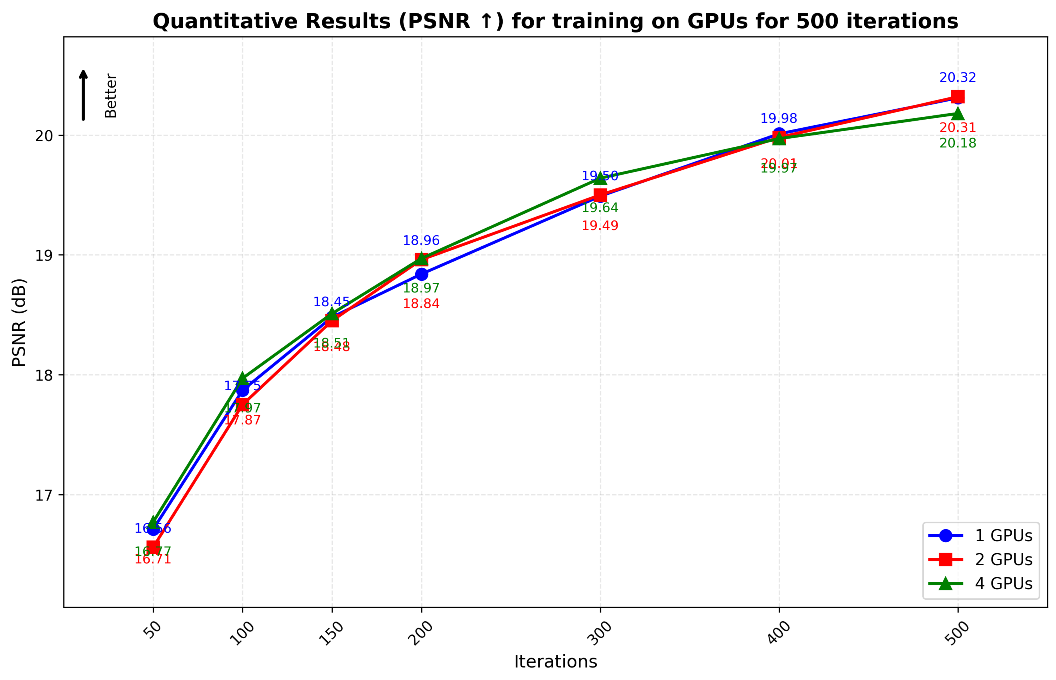

Figure 4: Comparing PSNR in training multiple GPUs for the first 500 iterations

- 4-GPU Performance: The 4-GPU configuration performed not as good as the other configurations, which only achieves 23.78 dB while taking 29% longer to train (221.75s). This degradation indicates that our current implementation faces some scaling challenges beyond 2 GPUs.

Here are the convergence patterns (in PSNR) we recorded at every 1000 iterations:

- For a single GPU, the curve is a smooth increase in PSNR over iterations.
- For 2 GPUs, however, the curve initially shows competitive performance in comparison to a single GPU, but it then converges more slowly in later iterations.
- For 4 GPUs, it has inconsistent improvement, which indicates synchronization or gradient averaging issues.

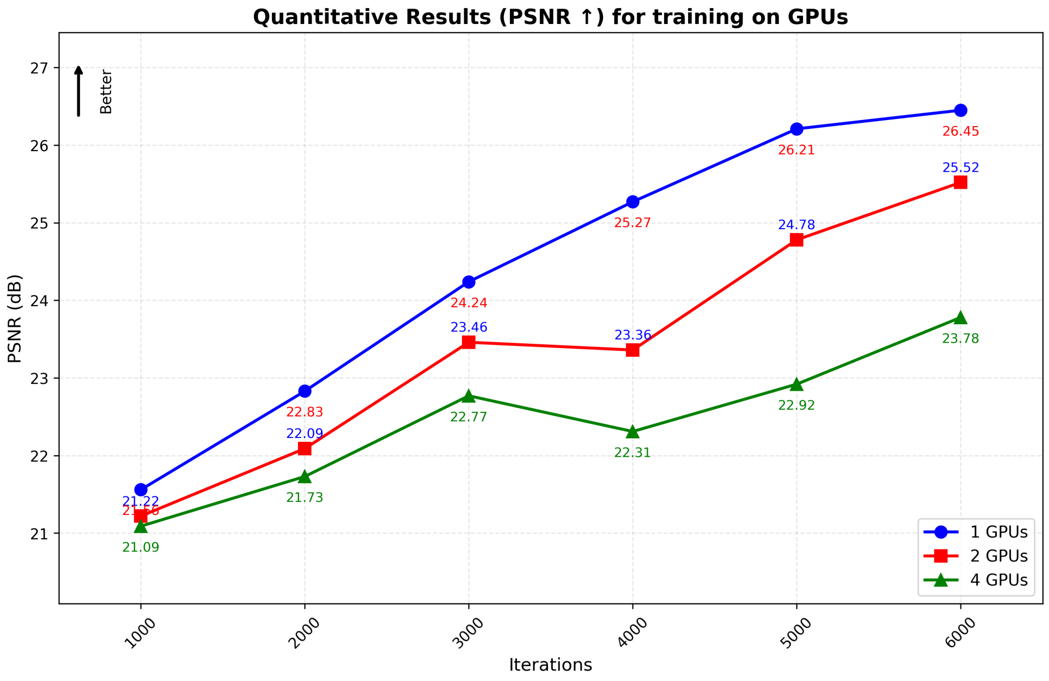

Figure 5: Comparing PSNR in training multiple GPUs

3.2 Qualitative

Since we found that batching was most effective for lower iterations, we tested how that difference in PSNR translated to the human perception of the 3D scene. We ran the original and batched model (using 2 GPU) for 200 iterations and visualized the 3D scene using the 3DGS viewer whose binaries are included in the original SIGGRAPH implementation [1].

Overall, we can see finer details in the foliage and grass in the batched version 3.2. By moving closer to the bike in 3D space, we can see that the spokes on the tire and treads are more visible in the batched output as compared to the original. While the metrics for the two scenes does not differ much PSNR-wise (< 0.01), the details present in the batched version are easily visible as a glance.

3.3 Reasoning

The reason batching was so effective for lower iteration counts, but faltered at higher ones is likely because of how greatly the views varied. While averaging loss for early iterations allows us to get a closer approximation of distance from the true target distribution (And therefore take larger steps using the $lr * \sqrt{N}$ term [2]), it can be detrimental for later iterations when we are trying to learn finer details that vary greatly from view to view.

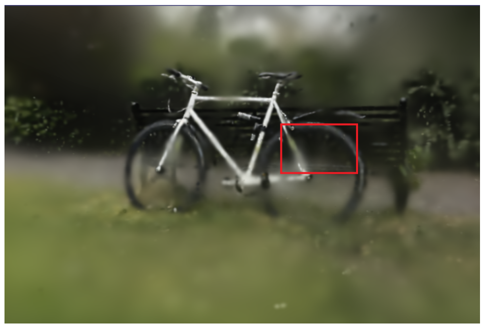

Figure 6: Capture from Cam 0 of 200 iteration on 1 GPU output

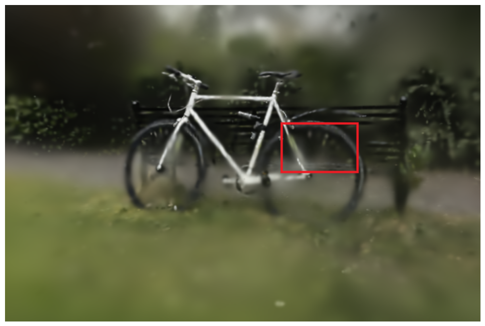

Figure 7: Capture from Cam 0 of 200 iteration on 2 GPU output

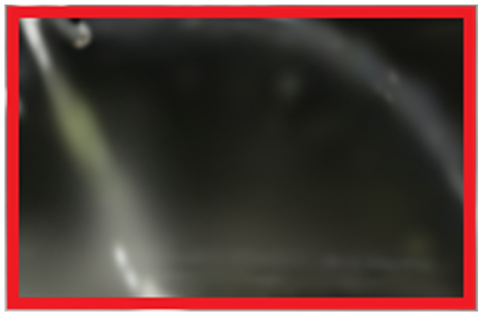

Figure 8: Zoom of 1 GPU output shows no detail in wheel

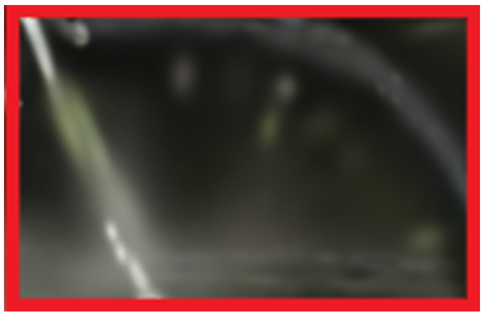

Figure 9: Zoom of 2 GPU output shows spokes and more details on wheel

4 Conclusion

While we were not able to include the pixel partitioning and load-balancing features we set out to do, we did implement an efficient batching solution that improved convergence at lower iterations and laid the groundwork for further work to improve scalability. At those low iterations, our method showed quantitative and substantial qualitative results in terms of quality and very promising results in terms of memory usage.

Since each GPU now only requires $1/N$ the amount of memory compared to the single GPU in the original implementation which had to fit the full dataset, we can use larger and higher quality data for a better reconstruction.

By developing a better learning rate scheduler, it should be possible to surpass the PSNR of the original implementation at greater iterations as well.

5 Challenges

Adapting the existing code-base into an efficient, parallelized pipeline came with a number of difficulties and compromises largely resulting from the control limitations that using CUDA through python comes with.

1. One of the main issues with the our current implementation is how images are loaded onto each GPU. Since all of the data is stored on the rank 0 GPU before scattering, this strictly caps the size of our input dataset. We attempted to load each GPU with its own images directly at initialization, but this resulted in memory access errors during optimization phase.
   (a) Since pruning and densification during the optimization phase requires access to all images (For occlusion and culling), giving each GPU its stand-alone dataset meant there were some images it had no access to. When scattering all images from rank 0,

this issue is handled by PyTorch's Distributed Data Parallel (DDP) which keeps track of GPU memory locations for those steps.

2. The reason why we had each GPU maintain its own copy of the gaussians was because the communication cost to sync them was too high. While the gaussian models are certainly large, they typically shouldn't cause such delays since the NCCL communication is so well optimized. The delays in communication were a result of PyTorch's handling of mixed models.
   (a) While PyTorch has efficient methods of communication based on the NCCL backend for both python objects (send_object_list(), recv_object_list, etc.) and data on the GPU (The standard send, recv, broadcast, etc.), it does not have any for mixed objects (i.e. The gaussian model which has data on both the RAM (Small values like number of gaussians or file locations) and the GPU (Large model with gaussian positions and rotations). This means all data needs to be serialized from the CPU RAM to the GPU before sending which was just too much overhead.
   (b) Since we weren't aiming to optimize compute usage here (Mainly aiming for memory and runtime), we felt that maintaining copies was a fine fix.

6 Future Work

6.1 Benchmark Context

To contextualize our work, we refer to the original evaluation methodology presented in the 3D Gaussian Splatting paper. Their results were benchmarked across 13 real-world and 8 synthetic scenes using datasets such as:

- **Mip-NeRF360**: Indoor and outdoor unbounded real-world scenes.
- **Tanks and Temples**: A standard dataset for 3D scene reconstruction evaluation.
- **Deep Blending**: A challenging real-world dataset for view synthesis.
- **Blender Synthetic**: A set of photorealistic NeRF synthetic scenes.

Their evaluation pipeline computes common metrics: PSNR, SSIM, and LPIPS across all methods under identical conditions using the provided `full_eval.py` script.

6.2 Prior Results and Observations

The authors report that 3DGS achieves state-of-the-art rendering quality while also enabling real-time performance. Compared to prior baselines like Instant NGP and Mip-NeRF360, their approach reached:

- Comparable or higher **PSNR** (e.g., 27.21 dB on Mip-NeRF360 scenes at 30K iterations).
- Superior **SSIM** and **LPIPS**, indicating better perceptual quality.
- Training times are reduced to minutes, and rendering at over 30 FPS, in contrast to slower volumetric NeRF variants.

For instance, on Tanks and Temples scenes, 3DGS reported a PSNR of 23.14 dB and an SSIM of 0.841, outperforming even the high-quality Mip-NeRF360 method in perceptual fidelity.

6.3 Our Contribution in Context

Our work builds upon this by addressing scalability, specifically multi-GPU training, which the original implementation lacked. While 3DGS already performs efficiently, our distributed version enables:

- **Faster convergence at lower iteration counts** through multi-view batching.
- **Memory distribution** across GPUs to support larger or higher-resolution scenes.

Unlike some academic methods that rely on external libraries, our design maintains simplicity, which relies only on PyTorch DDP and native CUDA-enabled operations. This lowers the barrier for scaling Gaussian Splatting to HPC environments.

6.4 Limitations

While our modifications improve efficiency, there are a few limitations that remain:

- **Full model replication overhead**: In our current implementation, each GPU maintains a full copy of the Gaussian model. While this simplifies synchronization, it significantly increases memory usage, especially at higher resolutions or with more GPUs. This approach occasionally leads to out-of-memory (OOM) errors. We frequently encountered during experiments on high-resolution scenes or large-scale training setups.
- **Static load balancing**: Our parallelization assumes uniform work distribution; heterogeneous hardware or irregular workloads could cause performance bottlenecks.

6.5 Future Directions

Our current work successfully introduces multi-GPU support for 3DGS through a straightforward and yet effective batching mechanism. However, several approaches remain open for future exploration and optimization:

- While our implementation leverages view-level batching, a more detailed pixel-based parallelism could allow multiple GPUs to process different regions of a single image. This would enable even finer load balancing and better GPU utilization for extremely high-resolution scenes.
- We currently assume a static and uniform workload per GPU, which may not scale well with heterogeneous clusters or datasets with varying computational complexity per view. Integrating a dynamic scheduler that adapts load distribution at runtime could significantly improve performance in diverse hardware environments.
- Our experiments were limited to a subset of test scenes at various resolutions due to computational constraints. As a next step, training and evaluating on the full suite of datasets used in the original 3DGS paper (e.g., Mip-NeRF360, Tanks and Temples, Deep Blending, Synthetic Blender) would allow for a direct, quantitative comparison and help identify any trade-offs introduced by the parallelization method.
- Future work could explore a hybrid approach that enables more frequent synchronization of Gaussian parameters across GPUs while minimizing communication overhead.
- Further reducing the memory footprint, through techniques like mixed-precision training (FP16), could allow for larger batch sizes or more complex models.

Ultimately, our distributed pipeline demonstrates that with modest architectural changes, 3DGS can be made substantially more scalable. This opens the door for further application in high-performance computing environments where multi-GPU resources are readily available, and training time and memory usage are critical constraints.

References

[1] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering, 2023.

[2] Hexu Zhao, Haoyang Weng, Daohan Lu, Ang Li, Jinyang Li, Aurojit Panda, and Saining Xie. On scaling up 3d gaussian splatting training, 2024.