CLM: Removing the GPU Memory Barrier for 3D Gaussian Splatting

Hexu Zhao*
New York University
New York, NY, USA
hz3496@nyu.edu

Xiwen Min*
New York University
New York, NY, USA
xm2336@nyu.edu

Xiaoteng Liu
New York University
New York, NY, USA
xiaoteng.liu@nyu.edu

Moonjun Gong
New York University
New York, NY, USA
gw2396@nyu.edu

Yiming Li
New York University
New York, NY, USA
yl7516@nyu.edu

Ang Li
Pacific Northwest National Laboratory
Richland, WA, USA
University of Washington
Seattle, WA, USA
ang.li@pnnl.gov

Saining Xie
New York University
New York, NY, USA
sx352@nyu.edu

Jinyang Li
New York University
New York, NY, USA
jinyang@cs.nyu.edu

Aurojit Panda
New York University
New York, NY, USA
apanda@cs.nyu.edu

Abstract

3D Gaussian Splatting (3DGS) is an increasingly popular novel view synthesis approach due to its fast rendering time, and high-quality output. However, scaling 3DGS to large (or intricate) scenes is challenging due to its substantial memory requirement, which exceeds the memory capacity of most GPUs. In this paper, we describe CLM, a system that allows 3DGS to render large scenes using a single consumer-grade GPU, e.g., RTX4090. It does so by offloading Gaussians to CPU memory, and loading them into GPU memory only when necessary. To improve performance and reduce communication overheads, CLM uses a novel offloading strategy based on insights into 3DGS's memory access patterns. This strategy enables efficient pipelining, which overlaps GPU-to-CPU communication, GPU computation and CPU computation. Furthermore, CLM exploits these access patterns to reduce communication volume. Our evaluation shows that the resulting implementation can render a large scene that requires 102 million Gaussians on a single RTX4090 and achieve state-of-the-art reconstruction quality. The code is open-sourced at: https://github.com/nyu-systems/CLM-GS

*Hexu and Xiwen contributed equally.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790140

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Computing methodologies** → *Graphics systems and interfaces.*

Keywords: GPU memory offloading; Heterogeneous systems for ML; 3D Gaussian Splatting

ACM Reference Format:
Hexu Zhao, Xiwen Min, Xiaoteng Liu, Moonjun Gong, Yiming Li, Ang Li, Saining Xie, Jinyang Li, and Aurojit Panda. 2026. CLM: Removing the GPU Memory Barrier for 3D Gaussian Splatting. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790140

1 Introduction

Recently, there has been significant interest [3, 5, 8, 23, 40, 51] in using 3D Gaussian Splatting (3DGS) for novel view synthesis (Figure 1). Given a set of *posed images* (i.e., images with position and orientation) for a 3D scene, 3DGS iteratively trains a *scene representation* that consists of a large number of anisotropic 3D Gaussians that capture the scene's appearance and geometry. Users can then use the trained scene representation to render images from a previously unseen view. Compared to other novel view synthesis approaches, 3DGS has faster rendering time while achieving comparable image quality, thus leading to its surging popularity.

The quality of images rendered using 3DGS depends on the fidelity of the trained scene representation. Scenes that capture a large area or contain intricate details require a larger number of Gaussians. As a result, 3DGS's memory footprint grows as scene size, scene intricacy, or output image resolution increases. State-of-the-art 3DGS implementations run

on GPUs, where memory is not plentiful. Therefore, memory capacity has been a barrier when scaling 3DGS and applying it to large intricate scenes with high image resolution. As we explain in §7, prior work on scaling 3DGS either adds significant cost because they use multiple GPUs [9, 30, 51], or compromises rendering quality because they reduce the scene representation's fidelity [14, 15, 24, 32, 35, 40, 50].

In this paper, we describe CLM, a system that scales 3DGS without requiring multiple GPUs or hurting rendered image quality. CLM's design is based on the insight that the computation of 3DGS is inherently sparse; i.e. each training iteration only accesses a small subset of the scene's Gaussians. Thus, it is sufficient to load only this subset into GPU memory, while leaving the remaining Gaussians offloaded to the more plentiful CPU memory. Despite this straightforward insight, as the GPU-CPU communication incurs significant overhead, it is challenging to realize the idea of memory offloading with good performance.

We develop a novel 3DGS-specific offloading strategy for CLM. Our offloading strategy minimizes performance overheads and scales to large scenes by leveraging four observations (§3) about the 3DGS training pipeline:

(i) The set of Gaussians accessed by each view (aka a training image) can be computed ahead-of-time, thereby allowing the loading of Gaussians for one iteration to be overlapped with the computation for the previous iteration to reduce communication overhead (§4.1, §4.2).
(ii) There is substantial overlap between the Gaussians accessed by different views, which allows us to cache the overlapping Gaussians to reduce the communication volume during each training iteration (§4.2.1).
(iii) The training process exhibits spatial locality, i.e., views in the same region tend to access the same Gaussians, allowing us to schedule training iterations carefully to maximize overlapped accesses across successive iterations in order to minimize overall communication volume (§4.2.3).
(iv) We can further use spatial locality to overlap gradient computation and a substantial portion of the Gaussian parameter update (§4.2.2).

By exploiting the inherent sparsity in scenes and the four observations above, CLM can scale to very large scenes: our evaluation (§6) shows that we can train a large scene with 100 million Gaussians on an a consumer-grade GPU (RTX4090) while achieving output quality on par with (or better than) the state-of-the-art systems. Furthermore, we show that CLM's 3DGS-specific offloading solution incurs modest performance overheads compared to a baseline without offloading when rendering small scenes that can fit in the baseline system's GPU memory.

The rest of this paper is organized as follows: in §2 we provide background on novel-view synthesis and 3DGS; in §3 we detail observations about 3DGS's memory access patterns; in §4 we describe CLM's design and in §5 provide details about our implementation; we evaluate CLM in §6; and present related work in §7.

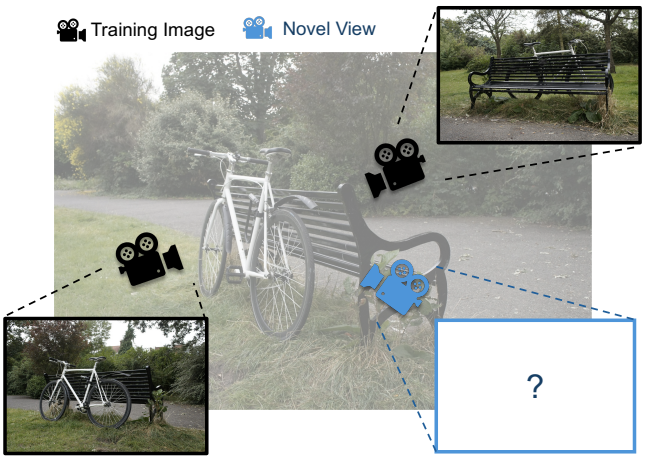


Figure 1. *The novel view synthesis problem: given a set of training images (with known pose) from a scene, render the image from a novel view with an unrecorded camera position and orientation.*

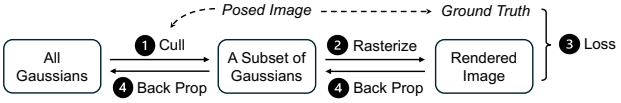


Figure 2. *3D Gaussian Splatting Illustration.*

2 Background and Motivation

This section gives an overview of the 3DGS algorithm and its application, discusses its memory bottleneck, and explains the challenges associated with naive offloading strategies.

2.1 Novel View Synthesis and 3D Gaussian Splatting

Novel View Synthesis (NVS) is the task of rendering an image of a 3D scene from a previously unseen viewpoint. To do so, NVS algorithms take as input a set of posed images (Figure 1), and use this input to construct a *scene representation*, i.e., a 3D model that captures the scene's appearance and geometry. The scene representation is then used to render the desired image. Modern ML-based NVS approaches all aim to learn how to reconstruct the scene from input data but differ in how the scene is represented: 3DGS uses 3D Gaussians [23] while others have used a mesh [34] or a neural network [36].

	3D Position	Covariance (Scale+Rotation)	Spherical Harmonics (Color)	Opacity	Total
#Param	3	3+4	48	1	59

Table 1. *A 3D Gaussian has 59 learnable parameters representing 4 types of attributes.*

3DGS represents the scene as a (potentially very large) collection of anisotropic 3D Gaussians, each of which consists of several dozen parameters representing four types of attributes including position, anisotropic covariance, spherical harmonic coefficients and opacity, as seen in Table 1. These

Scene	Resolution	# Gaussians	Memory Demand
Bicycle [6]	4K	9 M	10 GB
Rubble [45]	4K	40 M	50 GB
Alameda [7]	2K	45 M	60 GB
Ithaca [12]	1K	70 M	80 GB
Matrixcity BigCity [31]	1080P	100 M	110 GB

Table 2. *This table details the necessary number of Gaussians and minimum memory requirements during 3DGS training for 3 scenes with varying resolution and Gaussians quantity. The Rubble, Alameda, Ithaca and BigCity datasets are much larger than the Bicycle dataset and demand more memory than a single 24GB RTX 4090 can supply.*

learnable parameters of a 3D Gaussian dictate its effects on a rendered scene.

3DGS' differentiable rendering allows it to use gradient-based optimization based on minibatch SGD. In particular, Gaussians are initialized either randomly or using a user-provided point cloud generated by COLMAP [43]. Then, training proceeds iteratively. At each training step, one (or a batch of) camera view is selected from among the views represented in the training data set. The selected view is rendered (Figure 2) by ❶ first selecting the set of Gaussians in the camera's frustum (referred to as frustum culling) and then ❷ rasterizing them. Afterwards, ❸ loss is computed by comparing the rendered image with the groundtruth training image, and ❹ backpropagation is run to update the Gaussian attributes. Periodically, adaptive densification [23–25] is performed to increase the number of Gaussians in areas with high reconstruction errors and to prune unnecessary Gaussians.

2.2 Challenges of training 3DGS on a consumer GPU

The memory barrier to scaling 3DGS. State-of-the-art 3DGS implementations [23, 49, 51] run on GPUs for performance. However, representing a complex scene using 3D Gaussians requires a significant amount of memory, more than what is available on most GPUs. To estimate 3DGS' memory consumption, we observe that the model states (i.e., the set of Gaussians representing the scene) consume a majority of the memory used during rendering. As shown in Table 1, each Gaussian has 59 parameters, each of which results in four 4-byte floating point numbers stored during training: the parameter itself, its gradient, and two additional versions as the optimizer state [23, 28]. Thus, for a scene with N Gaussians, the model state alone requires $N \times 59 \times 4 \times 4$ bytes, which can easily exceed the memory capacity of consumer-grade GPUs. For example, RTX 4090 with 24GB can only store the memory state of up to 26 million Gaussians, even if we ignore the memory consumption of the activation state and various other temporary buffers. Table 2 lists the number of Gaussians required to achieve good rendering quality for well-known NVS datasets. Except for the smallest scene (Bicycle), all other larger and more complex scenes such as Rubble, Alameda, Ithaca, and MatrixCity BigCity cannot be trained on a single consumer-grade GPU such as RTX 4090.

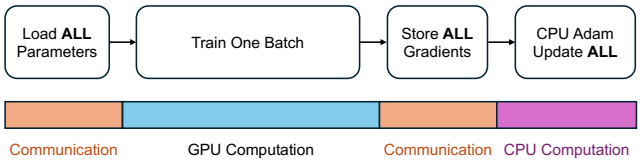


Figure 3. *Runtime decomposition of one batch in naive offloading. It leads to overheads in communication and CPU Adam computation.*

Challenges of offloading. Existing work addresses the memory barrier of 3DGS through multi-GPU training or Gaussian pruning. Both approaches have shortcomings: using multiple GPU training significantly raises training cost, while Gaussian pruning not only degrades quality (see §7 for more details) but also fails to handle very large scenes, where even a pruned model may exceed GPU memory capacity. In contrast, our work develops an orthogonal approach by offloading Gaussians to CPU memory.

A simple approach to address this problem would be to use a technology such as Unified Virtual Memory that uses CPU memory to augment GPU memory, and swaps data in from main memory when required by the GPU. While this is indeed simple it has significant overheads [10], and would thus not suffice for our use case.

Work on deep-learning, e.g. Zero-Offload [41], has shown that it is possible to train a large model without impacting quality by offloading the gradients, optimizer states and optimizer computation to CPU. Thus, one can ask whether a similar offloading approach would work for 3DGS. Figure 3 shows how such a Zero-offload inspired approach could work for 3DGS: in each training step, first, all Gaussians are transferred from CPU memory to GPU memory; next, the forward and backward computation are carried out on the GPU; and finally, gradients are sent back to the CPU where the Adam optimizer [27] is run to update Gaussian parameters.

However, naively applying Zero-offload leads to two problems: First, as Figure 3 shows, naive offloading incurs significant performance overhead due to CPU-GPU communication and the additional time required to run Adam on the CPU. However, naive offloading lacks the means to effectively hide such overhead by overlapping GPU computation with communication and CPU computation. Second, as naive offloading loads all Gaussians to the GPU, its GPU memory requirement remains proportional to the number of Gaussian such that large scenes cannot fit on a single GPU.

3 Our approach: sparsity-guided offloading

Our approach addresses the challenges we discussed above by storing some Gaussians parameters in pinned main memory, and loading them to GPU memory on demand. We reduce the overheads from offloading by taking advantage of several unique characteristics of 3DGS.

3DGS computation is very sparse. 3DGS's computation is sparse, in that only a fraction of the scene's Gaussians are used when rendering a view (during training or inference).

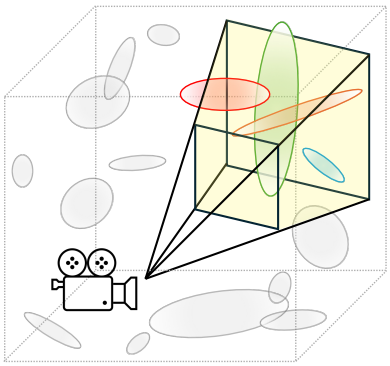


Figure 4. *Frustum Culling: Gaussians outside of camera frustum will not be accessed when rendering the camera's view. Further, the Gaussians accessed when rendering a view are in the same region, i.e., the process exhibits spatial locality. Our approach uses these observations to improve performance and reduce GPU memory requirements. This results in a sparse memory access pattern to gaussians parameters. This also shows that 3DGS rendering has the property of spatial locality.*

This is because each view is associated with a camera pose and only those Gaussians within the camera's frustum can contribute to the rendered image, as illustrated by Figure 4. In fact, 3DGS's rendering workflow explicitly computes the set of Gaussians within the frustum before processing them for a given view before processing them (shown in Figure 2 ❶).

We have found that a single view accesses a very small fraction (less than 1%) of a large scene's Gaussians. We quantify this by calculating the sparsity ρ^i for view i in a scene as $\rho^i = \frac{|\mathcal{S}_i|}{N}$, where $\mathcal{S}_i$ is the set of Gaussians in view i's and N to be the total number of Gaussians. Figure 5 shows the CDF of ρ_i for the datasets in Table 2. As can be seen, larger scenes exhibit higher sparsity (aka smaller ρ). This is expected because, while the number of Gaussians scale as a function of scene size, the volume enclosed by the camera frustum is independent of scene size. For the largest scene (Matrixcity BigCity), we found that the average view only accessed 0.39% of Gaussians, and the maximum number of Gaussian's accessed by a view is 1.06%.

We leverage sparsity by using 3DGS' frustum culling logic to identify the subset of Gaussians needed to process each view (and thus the microbatch) in advance, and only transfer those needed to the GPU.

Sparsity patterns across views exhibit spatial locality. Different views (for the same scene) have different but overlapping sparsity patterns. In other words, for views i and j, $\mathcal{S}_i \neq \mathcal{S}_j$ and the number of Gaussians in the intersection, $|\mathcal{S}_i \cap \mathcal{S}_j|$, is dependent on how much spatial locality exists between these views based on their camera positions and angles.

We exploit spatial locality to optimize data transfer between CPU and GPU in two ways: 1) we compute each microbatch's sparsity pattern in advance and schedule microbatches carefully to increase overlapped access (§4.2.3), and 2) we cache Gaussians used by successive microbatches on the GPU (§4.2.1), thus reducing communication overheads.

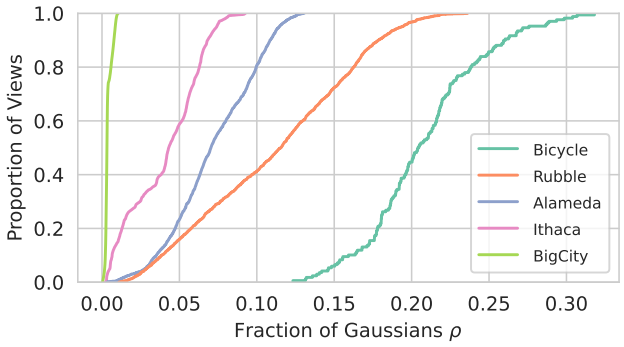


Figure 5. *Empirical cumulative distribution functions (CDF) for the sparsity in Bicycle, Rubble, Alameda, Ithaca, and BigCity.*

Sparsity patterns can be computed using partial Gaussian information. In existing 3DGS implementations, all Gaussian parameters are stored in a single tensor which is used to perform frustum culling on the GPU. Doing so requires all Gaussians to be loaded into GPU memory in order to determine a view's sparsity pattern, which contradicts our earlier design choice to only load those necessary Gaussians. To address this problem, we develop an approach (§4.1) that stores some of the attributes (position, rotation and scale) of all Gaussians on the GPU. As we explain in the next section, this approach is practical because these attributes take relatively little memory.

4 System Design

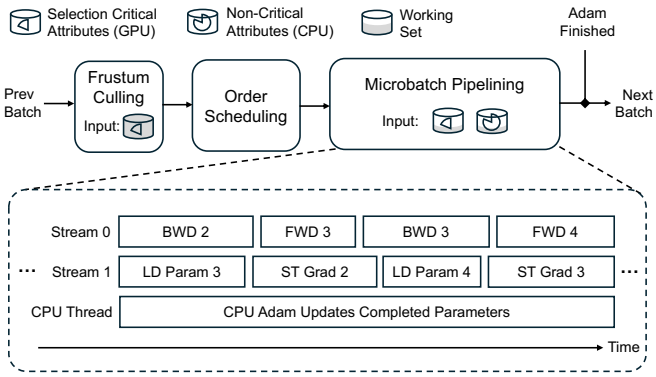


Figure 6. *The workflow of a training step in CLM. For a batch of images, CLM first performs frustum culling for each image, then schedules their rendering order, and finally uses micro-batch pipelining to overlap communication (on GPU stream 1) and Adam optimizer update (on the CPU) with GPU rendering (on GPU stream 0). "FWD" and "BWD" refer to the forward and backward passes of the i-th microbatch, "LD" and "ST" refer to loading Gaussian parameters from CPU and storing the gradients to CPU. The numbers (1,2,3) next to these operations indicate the microbatch ID. The area of shading on attributes denotes the proportion that is active as working set.*

We now describe CLM, a 3DGS system that allows large or highly detailed scenes to be trained/rendered on a single consumer-grade GPU. Our design extends effective GPU memory capacity by offloading Gaussian parameters and

optimizer computation to the CPU, while leveraging the observations from the previous section (§3) to reduce overhead from GPU-CPU communication and CPU computation (§2.2). Our current implementation of CLM is built on CUDA, but the design itself is agnostic to the rendering backend and can be ported to the Vulkan platform [26].

CLM trains a scene representation as shown in Figure 6: First, it selects a batch of training images and views, and then uses frustum culling (§4.1) to compute the set of Gaussians $\mathcal{S}_i$ required by each view i in the batch. We refer to these as in-frustum Gaussians. Next, it divides a batch into several microbatches to enable microbatch pipelining. More importantly, it uses frustum culling's output to determine the order in which microbatches are processed to maximize spatial locality (§4.2.3). Finally, each microbatch is processed in a pipelined fashion to overlap both communication and CPU computation with GPU computation. More concretely, when processing a microbatch, CLM loads into GPU memory those in-frustum Gaussians, using Gaussian Caching (§4.2.1) to avoid redundantly loading any Gaussian that is used by two consecutive microbatches. CLM then executes the forward and backward training pass on the GPU and transfers gradients back to the CPU, where a concurrent CPU thread executes the Adam optimizer and updates the Gaussian parameters. As Figure 6 shows, the CPU-GPU communication for loading in-frustum Gaussians for microbatch i overlaps with the backward GPU computation for microbatch $i-1$; and the GPU-CPU communication for transferring gradients of microbatch i overlaps with the forward GPU computation for microbatch $i+1$. For those Gaussians that are last updated by a microbatch, CLM performs their corresponding Adam updates on CPU, which overlaps with the forward/backward GPU computation done by subsequent microbatches (§4.2.2).

4.1 Attribute-Wise Offload

As we discussed previously in §3, the frustum culling step is run on the GPU and requires access to some attributes (e.g., position) of all Gaussians in the scene. As our goal is to scale to scenes whose Gaussians (by which we mean all attributes) do not fit in GPU memory, we cannot load all Gaussians into GPU memory before running the frustum culling step.

We address this by observing that frustum culling accesses a small subset of each Gaussian's attributes: For each Gaussian, the frustum culling algorithm checks the intersection between the view frustum and the Gaussian. When computing intersection, the algorithm only needs information on the Gaussian's position, scale, and rotation. This is because intersection does not depend on the Gaussian's color (determined by the spherical harmonics) or opacity. In particular, during frustum culling, 3DGS determines whether a Gaussian is in-frustum by computing the intersection between the view frustum and the Gaussian's ellipsoid (which is derived from its scale and rotation) and checking whether it is within 3 standard deviations (3σ) of the Gaussian's distribution. Culling within 3σ is standard practice in existing 3DGS implementations [23, 49]. We refer to the attributes required for frustum culling as *selection-critical attributes*, and the rest of the attributes as *non-critical attributes*.

We observe that the selection-critical attributes constitute less than 20% (10 out of 59 floats) of a Gaussian's memory footprint (Table 1). Thus, CLM keeps the selection-critical attributes for all Gaussians in GPU memory, i.e., they are never swapped out to CPU memory, and no additional CPU-GPU communication is necessary for frustum culling. Non-critical attributes are stored in CPU memory, and loaded into GPU memory only when required.

4.2 Microbatch Pipelining

3DGS training uses minibatch gradient descent. In contrast to existing 3DGS systems that process a whole batch at a time, our design divides each batch into several minibatches to use pipelining and gradient accumulation [22, 38]. In our setting, a batch consists of multiple images and a microbatch consists of a single image. We start the forward pass for microbatch $i+1$ as soon as the backward pass for microbatch i completes.

CLM's use of microbatch pipelining reduces GPU memory requirements: each forward and backward pass processes a single image at a time, reducing the amount of activation memory required. More importantly, as shown in Figure 6, microbatch pipelining allows CLM to overlap communication for one microbatch with the computation for another, thereby hiding communication overhead. CLM uses double-buffering to ensure that communication and computation can be safely overlapped. While the use of double-buffering increases memory requirements, the additional memory requirements are independent of scene and batch size.

We further improve basic microbatch pipelining by incorporating three domain specific optimizations, which are described next: (a) Precise Gaussian Caching (§4.2.1); (b) Overlapped CPU Adam (§4.2.2); and (c) Pipeline Order Optimization (§4.2.3).

4.2.1 Precise Gaussian Caching. Our first optimization builds on the observation that some of the Gaussians accessed by microbatch i and $i+1$ are the same because of spatial locality. Since the frustum culling step has already computed in-frustum Gaussians for each microbatch, CLM uses this information to reduce the number of Gaussians loaded into GPU memory from CPU memory: when loading Gaussians for microbatch $i+1$, it copies the intersecting Gaussians (aka those in $\mathcal{S}_i \cap \mathcal{S}_{i+1}$) from microbatch i's Gaussian parameter tensors (which are already in GPU memory), and only loads those Gaussians not in the intersection from CPU memory. Note that copying Gaussians in this way does not require additional memory beyond what is already allocated for double buffering.

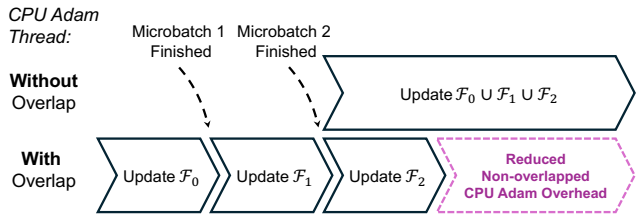


Figure 7. *Illustration of overlapping CPU Adam with batch size 2. The upper half of this image illustrates the overhead caused by CPU Adam, while the lower half demonstrates the benefits of overlapping CPU Adam. Within the diagram, $\mathcal{F}_i$ represents the set of Gaussians that have just finalized before microbatch i and are ready to update their parameters using Adam. For example, $\mathcal{F}_0$ includes Gaussians that are not affected by any images in the batch. $\mathcal{F}_{|Batch|}$ consists of the Gaussians touched in the last microbatch, which cannot be overlapped.*

We also use the same approach to avoid redundant copies when transferring gradients from GPU memory to CPU memory: after microbatch i has finished, we only transfer the gradients that are not going to be updated by microbatch $i+1$, i.e., gradients for Gaussians in the set $\mathcal{S}_i \setminus \mathcal{S}_i \cap \mathcal{S}_{i+1}$. We copy the rest of the gradients for Gaussians $\mathcal{S}_i \cap \mathcal{S}_{i+1}$ into microbatch $i+1$'s gradient buffer to be accumulated.

4.2.2 Overlapped CPU Adam.

At the end of each batch, 3DGS training uses the Adam optimzer [27] to combine the computed gradients and update Gaussian parameters. We observe that many Gaussian updates can be *finalized* early before the last microbatch. In other words, in a batch of size B, the last microbatch (i) that accesses (and thus updates) some Gaussian g might be $i < B$. In this case, it is safe to use Adam to update Gaussian g before the whole batch completes. Doing such early update is desirable because CLM performs Adam update on the CPU, which can be overlapped with the forward and backward GPU computation of subsequent microbatches. CLM implements this optimization (Figure 7): When scheduling a batch, for each Gaussian g, it computes the microbatch L_g at which g is finalized by computing $L_g = \max\{i \mid g \in \mathcal{S}_i\}$ ($L_g = 0$ if g is not accessed by a batch). At the end of every microbatch j, CLM uses CPU Adam to update all Gaussians g whose $L_g = j$. Only those Gaussians that are finalized in the last microbatch ($L_g = B$) do not have their CPU Adam computation overlapped.

4.2.3 Pipeline Order Optimization.

The order in which microbatches are processed within a batch does not affect correctness. This is because gradients calculated for individual microbatches are accumulated over the full batch before applying optimizer update, and thus the same final update is computed for a batch regardless of its microbatch ordering. However, the order of microbatches determines the effectiveness of Gaussian caching and overlapped CPU Adam. If the microbatch schedule leads to a large overlap between Gaussians accessed by consecutive microbatches (i.e., when $\mathcal{S}_{i+1} \cap \mathcal{S}_i$ is large for all i), Gaussian caching can eliminate more communication. Similarly, if the schedule results in a larger number of Gaussians finalized in earlier microbatches, more of the CPU Adam computation can be overlapped. Therefore, CLM tries to find a microbatch schedule that maximizes the effectiveness of both optimizations.

Scheduling of microbatch computation is aided by the observation that 3DGS exhibits spatial locality (§3). As such, any schedule that maximizes Gaussian overlap between consecutive microbatches must process all views in the same region in close temporal proximity. Further, Gaussians are finalized once all views in a region have been rendered. Thus, schedules that maximize overlap also tend to finalize a large number of Gaussians early. Given this observation, we compute a good schedule by formulating the scheduling problem as an instance of the Traveling Salesman Problem (TSP) [16]: we treat each microbatch as a node in the graph, and the distance between two microbatches i and j is given by the symmetric difference between the Gaussians accessed by each ($|\mathcal{S}_i \oplus \mathcal{S}_j|$ which gives the number of Gaussians that do not overlap). TSP computes the shortest Hamiltonian path through this graph, which by construction is the schedule that maximizes overlap. Our implementation uses stochastic local search with a greedy heuristic [11] to quickly generate an optimal solution; see Appendix A.1 for details about our formulation and the greedy search algorithm.

5 Implementation

We implemented CLM by extending Grendel [51], an existing 3DGS training framework. Our extensions added the offloading approach described in the previous section and incorporated the rasterization kernels of gsplat [49] into Grendel. We discuss important details about the implementation below.

5.1 Pre-rendering Frustum Culling

In current implementations of 3DGS[23][49], the frustum culling step is fused to the rendering kernels. These rendering kernels process all Gaussians as input, but only utilize those that intersect with the Frustum. The intersected Gaussians are computed implicitly by cuda threads and registers, without being explicitly stored in GPU memory. In large scenes with low ρ, the majority of input Gaussians are not in-frustum and hence result in substantial wasted GPU computation. Storing intermediate activations for non-in-frustum Gaussians also wastes GPU memory. The backward pass also performs unnecessary computation because it calculates gradients for the entire Gaussian input tensor, even though only in-frustum Gaussians have non-zero gradients.

In our implementation, we perform frustum culling to obtain in-frustum Gaussians indices $\mathcal{S}_i$ and store them in GPU memory before rendering. This allows us to explicitly eliminate unnecessary Gaussians from the input to the rasterization kernels, thus decreasing the input size by ρ_i, reducing N to $|\mathcal{S}_i|$. Doing so reduces both GPU memory and computation usage.

Pre-rendering frustum culling is a simple engineering method that can also be applied to traditional GPU-only training without offloading. The evaluations in Section 6.3 demonstrate that this engineering technique reduces memory usage and significantly enhances throughput when training a highly sparse scene. We will also show that CLM remains highly effective even after eliminating the influence of this engineering trick.

5.2 Selective Loading Kernel

After computing indices for the in-frustum Gaussians, we use a custom kernel to load parameters from CPU memory. As in-frustum Gaussians are spread over CPU memory because of the sparse access pattern, naively copying them individually using `cudaMemcpy` (or `cudaMemcpyAsync`) underutilizes CPU-GPU communication bandwidth. Instead, our implementation stores offloaded Gaussian attributes in pinned CPU memory that can be accessed directly from CUDA code running on the GPU without and we develop a *selective loading kernel* which loads (over PCIe) the in-frustum Gaussian parameters from CPU memory to GPU registers and then stores the register values into GPU memory. Since all of the communication is initiated from the GPU, this kernel avoids CPU-GPU coordination. In addition, the same kernel is also used to copy cached Gaussians from GPU memory for processing.

To further improve communication efficiency, we concatenate and pad attribute tensors when storing them in CPU memory so that all attributes of a Gaussian are stored in contiguous memory and are cache-line aligned. The selective loading kernel splits Gaussians attributes when loading them into GPU memory and concatenates those in-frustum Gaussian attributes together. Implementing splitting and concatenation logic in the same kernel reduces computational overheads.

We also develop a similar kernel to efficiently transfer gradients from the GPU to CPU memory.

5.3 Separate Communication Stream

For pipelined execution of microbatches, we employ two CUDA streams: one for computation, and the other for communication. As illustrated in Figure 6, the parameters loading and gradients storing are interleaved in the communication stream, analogously to the 1F1B pipelining method for training neural networks described in [18]. We add CUDA events to correctly synchronize operations across streams, making sure that in-frustum gaussians parameters are loaded before executing the microbatch and gradients are offloaded after finishing the backward pass. We increase the communication stream priority over the computation stream to prevent delays in executing the communication kernel by GPU. These delays, as we observed, impede subsequent microbatch processing and ultimately slow down the overall training process.

The kernel responsible for gradient offloading needs a minor adjustment to support gradient accumulation in pipelining. In the gradient offloading kernel, rather than directly storing gradients' values to CPU memory, it first fetches the previously accumulated gradients, adds them within cuda registers, and then stores the sum back in CPU memory.

Our pipeline approach involves prefetching parameters for the upcoming microbatch while postponing gradient offloading from the prior one. We use double buffer to achieve this, which can raise memory usage. However, with our single communication stream and 1F1B setup, along with precise timing in managing creation and deallocation, we prevent the coexistence of double buffers for previous microbatch gradients and upcoming microbatch parameters. This effectively controls the additional memory usage from double buffer.

5.4 Thread for CPU Adam update

Our CPU Adam implementation extends the Zero-offload [41] implementation to allow updating a subset of Gaussians (which have completed gradients at each call). We execute CPU Adam on a dedicated thread CPU thread. To allow concurrent execution, we release the Python GIL lock within the CPU adam thread, allowing the primary Python thread to continue processing. A signal buffer in CPU pinned memory is used to synchronize the GPU communication stream and the CPU Adam thread. Specifically, the GPU communication stream sets the gradient completion signal via DMA after the gradient transfer kernel finishes, while the CPU Adam thread waits on the signal buffer before performing the Adam update after the microbatch.

6 Evaluation

In this section, we evaluate CLM and the following are the highlights of our results:

- CLM enables 3DGS training of models up to 6.1x larger through CPU offloading, compared to GPU-only training baselines.
- CLM enhances reconstruction quality by training larger model, achieving PSNR 25.15 for BigCity[31] using 102 million Gaussians. In contrast, the GPU-only training reaches a PSNR of 23.93, using only 15 million Gaussians to avoid running out of memory.
- CLM has modest offloading overhead. It achieves 86%–97% of the throughput of an enhanced GPU-only baseline on RTX 2080 Ti and 55%–90% on RTX 4090. Compared to naive offloading, CLM is 1.38 to 1.92 faster.

6.1 Setting and Datasets

Testbeds. We run our evaluation on two testbeds: One is a machine with an AMD Ryzen Threadripper PRO 5955WX 16-core CPU, 128 GB RAM, and a 24 GB NVIDIA RTX 4090 GPU connected over PCIe 4.0. The other is a machine with Intel Xeon E5-2660 v3 20-Core CPU, 256 GB RAM, and a 11 GB NVIDIA RTX 2080 Ti GPU connected over PCIe 3.0.

These two settings allow us to evaluate CLM under different computation and communication speed. In particular, the RTX 2080 Ti has about 7x fewer FLOPs (cuda core) than the

RTX 4090, and PCIe 3.0 has 2x less bandwidth than PCIe 4.0. Therefore, in our experiments, the 2080 Ti GPU testbed is more likely to be compute bound.

Datasets. Our evaluation uses the datasets presented in Table 3. These datasets cover a variety of scene sizes (which roughly correlate with the number of images), image resolution, and scene types that collectively represent diverse workloads. Specifically, the scene size affects both the Gaussian model size and the degree of sparsity ρ in rendering; resolution affects both GPU rendering speed and activation memory usage; additionally, the scene topology determines the sparsity patterns. Together, they provide a comprehensive evaluation of our system. We detail the preparation of the datasets in the Appendix A.2.

Scene	# Images	Resolution	Scene Type	BS
Bicycle [6]	200	4K	Yard	4
Rubble [45]	1600	4K	Aerial	8
Alameda [7]	1700	2K	Indoor	8
Ithaca365 [12]	8200	1K	Street	16
MatrixCity BigCity [31]	60000	1080P	Aerial	64

Table 3. *Scenes used in our evaluation: Our selection includes scenes of different sizes, resolutions, and types, representing a diverse range of workload characteristics. We set the training batch size (BS) in our experiments to correspond with their scene sizes.*

Baseline. We choose Grendel-GS [51] as our baseline. Although Grendel-GS is a multi-GPU training system, we run in its single-GPU mode to take advantage of its efficient training framework. We use GSplat's CUDA kernels[1] [49] as the rendering backbone for its memory efficiency by setting the corresponding flag in the Grendel-GS training framework. We refer to this GPU-only implementation as "baseline".

Enhanced Baseline. Additionally, we build an enhanced version of the baseline by adopting CLM's pre-rendering frustum culling feature (§5.1) to avoid unnecessary kernel computation. This version is closer to CLM's kernel implementation, and thus provides a better baseline for evaluating CLM's offloading overhead. We refer to this GPU-only variant as "enhanced baseline".

Naive Offloading. We implement the naive offloading strategy (see §2.2 and Figure 3) on Grendel [51]. The implementation uses pinned memory for GPU-CPU communication as in CLM. It also utilizes the same CPU Adam as CLM (§5.4) and adopts CLM's pre-rendering frustum culling technique for efficient kernel computation. Additionally, it trains each batch one image at a time with gradient accumulation to reduce activation memory usage. By comparing with this naive offloading, we can quantify how CLM's various offloading techniques can improve performance.

6.2 Memory Efficiency

CLM is able to push the model size trainable on a single GPU by up to 6.1 times compared to the enhanced baseline[2]. Larger models enhance the quality of reconstruction, achieving state-of-the-art PSNR for the BigCity scene with 102 million Gaussians.

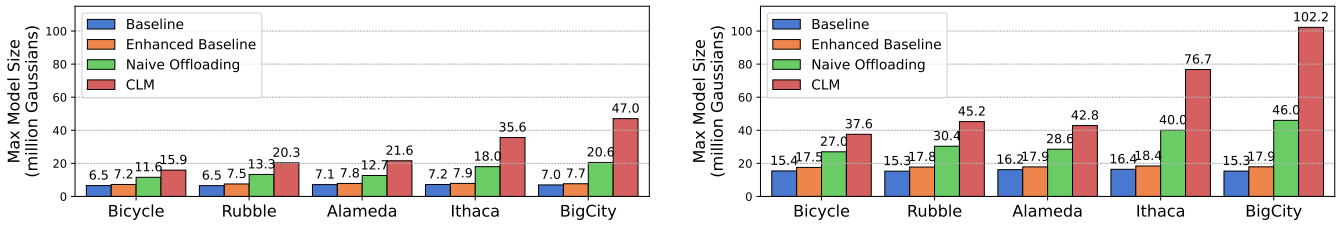
Larger model training made possible. Figure 8 shows the maximum model size that could be trained before encountering an OOM error on each testbed. We can see that CLM allows larger model sizes to be trained across all scenes. Specifically, the GPU-only baseline can support a maximum of 7.2 M and 16.4 M Gaussians on RTX 2080 Ti and RTX 4090, respectively, before running out of memory. The enhanced baseline avoids storing the activations of Gaussians unused in rendering via pre-rendering frustum culling (§5.1), thereby postponing the OOM point to 7.9 M and 18.4 M Gaussians, respectively. Using CPU memory, naive offloading can support up to 20.6 M and 46 M Gaussians before exhausting GPU memory. In contrast, CLM supports up to 47 M and 102.2 M Gaussians—up to 6.1x larger than the enhanced GPU-only baseline, and 2.3x larger than naive offloading.

CLM requires less memory than naive offloading because it does not load all Gaussian parameters to GPU memory before each rendering step. The difference in the maximum supported model sizes by CLM on 2080 Ti vs 4090 (e.g., 47 M vs. 102.2 M Gaussians for BigCity) roughly reflect their GPU memory capacities (11 GB vs. 24 GB). We also observe that the maximum trainable model size is dependent on the scene. In particular, scenes that have higher resolution (see Table 2) or worse sparsity (i.e., high ρ, as shown in Figure 5)—such as Bicycle, Rubble—require more activation memory, leaving less memory available for Gaussian parameters compared to scenes like Ithaca and BigCity, which have lower resolution and lower ρ.

Larger models improve reconstruction quality. This experiment assesses the importance of scalability model sizes using the BigCity scene[31], a city-scale benchmark covering 25.3 km^2 with extensive details. We train models consisting of 6.4 M, 12.8 M, 15.3 M, 25.6 M, 51.1 M, and 102.2 M Gaussians, doubling the size incrementally. Among these, the 15.3 M model is the largest one that can be trained by the GPU-only baseline on the 24 GB RTX 4090 testbed before running out of memory. We train each of these models for 500,000 steps using CLM and evaluate the reconstruction quality using peak signal-to-noise ratio (PSNR), where a higher PSNR indicates better reconstruction quality (aka the rendered image is closer to the ground truth). Figure 9 shows that CLM improves reconstruction quality by allowing larger models to be trained using additional CPU memory. With a model size of 102.2 M Gaussians, CLM achieves a PSNR of 25.15. In contrast, the

[1] These CUDA kernels consume over 95% time in 3DGS training.

[2] We enable the PyTorch's `expandable_segments` feature in all experiments to minimize the impact of GPU memory fragmentation.



(a) RTX 2080 Ti *(b) RTX 4090*

Figure 8. *Maximum trainable model size without OOM across two GPU testbeds and five scenes. CLM consistently enables significantly larger models. The BigCity scene shows the most notable improvement—6.1x larger than the enhanced baseline and 2.3x over naive offloading on RTX 2080 Ti; and 5.7x and 2.2x larger respectively on RTX 4090.*

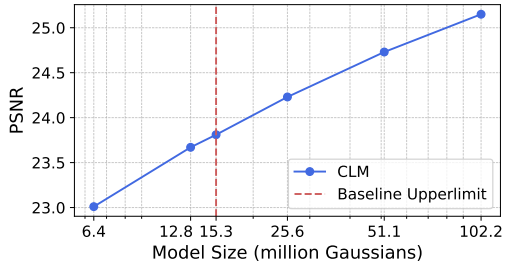
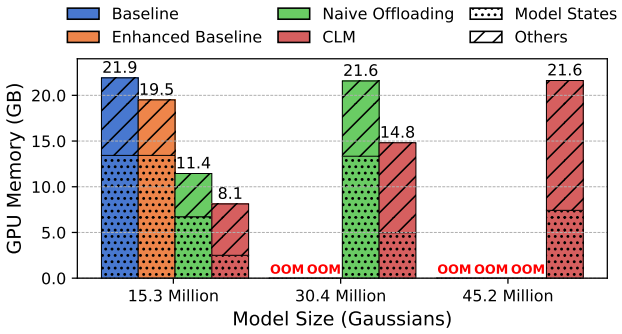


Figure 9. *Scalability on BigCity. X axis (log scale) is the model size presented in million Gaussians. Y axis (linear) is the PSNR evaluated on rendered images. Each data point is trained using CLM on a single 24GB RTX 4090 GPU for 500000 steps. This shows CLM enables scaling to 102 million Gaussians and achieve a state-of-the-art PSNR of 25.15.*

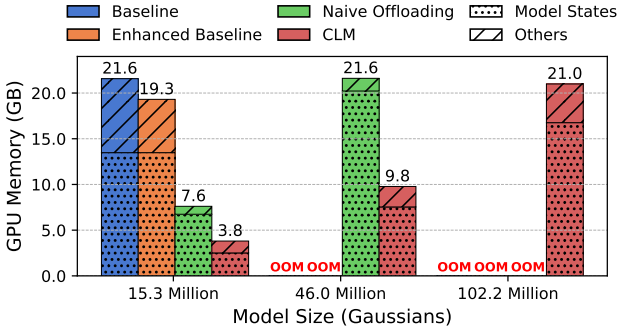
GPU-only baseline is limited to 15.3 M Gaussians and yields a lower PSNR of 23.93.

Breakdown of GPU memory consumption. We examine different systems' GPU memory consumption statistics when training the Rubble and BigCity scenes. We use model sizes 15.3 M, 30.4 M, and 45.2 M Gaussians for Rubble; and 15.3 M, 46.0 M, and 102.2 M for BigCity, corresponding to the maximum supported model sizes of the baseline, naive offloading and CLM, respectively (see Figure 8).

For the Rubble scene (see Figure 10a), during the training of the 15.3 M model where all four systems operate without running out of memory, the baseline consumes the most GPU memory, whereas CLM requires the least. The enhanced baseline and naive offloading fall somewhere in the middle. The two baselines consume the same amount of model state memory, with a small difference in "others" because the pre-rendering frustum culling (§5.1) in the enhanced baseline removes redundant activation states. For the 30.4 M and 45.2 M models, the baselines encounter GPU OOM. In contrast, both naive offloading and CLM support the training of the 30.4 M model, illustrating the advantages of GPU memory saving through offloading. Despite keeping some selection-critical attributes of all Gaussians on the GPU, CLM uses less GPU memory than naive offloading, which offloads all attributes to CPU memory. This is because naive offloading transfers all attributes of all Gaussians to the GPU before each rendering, which consumes larger GPU memory. While CLM



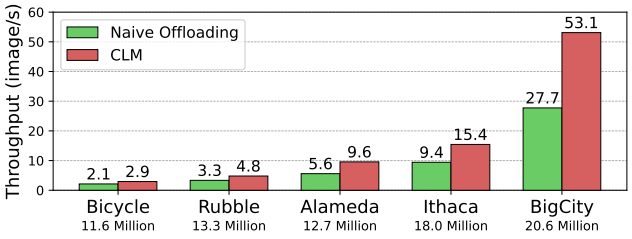
(a) Rubble



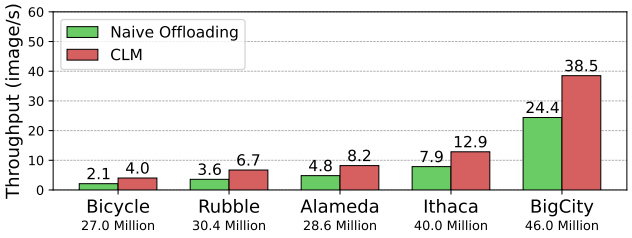
(b) BigCity

Figure 10. *Memory usage breakdowns for Rubble using varying model sizes on the 4090 testbed. Each bar is composed of two parts: model states (bottom) and others (upper). CLM consumes the least GPU memory, thus avoiding OOM.*

significantly cuts down memory for Gaussian model states, it slightly elevates the "others" memory usage. This is due to the double buffer design in our pipelining (§5.3) which enables parameters prefetching and delayed gradient offloading. All of the above observations can be drawn similarly for the BigCity scene in Figure 10b. The primary difference is that BigCity has a smaller ρ and lower resolution compared to Rubble, leading to model states memory dominating activation memory. Consequently, offloading model state in BigCity results in a more substantial overall memory reduction.

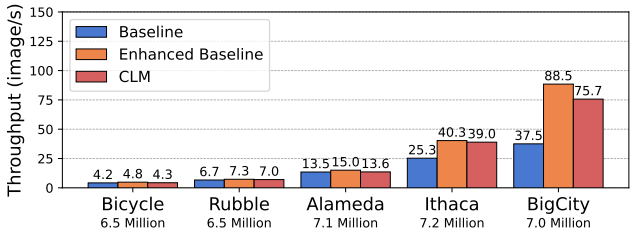


(a) RTX 2080 Ti

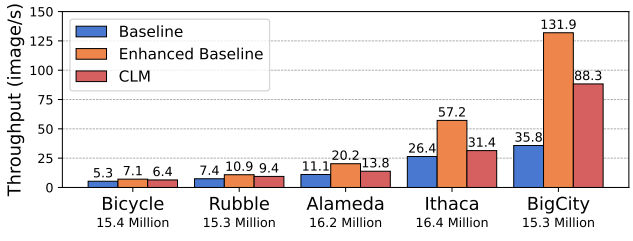


(b) RTX 4090

Figure 11. *The training throughput of CLM vs. naive offloading. For each scene-testbed pair, we use the largest model size supported by naive offloading from Figure 8 to avoid OOM. CLM achieves up to 1.92x (BigCity) and 1.90x (Bicycle) speedup on RTX 2080 Ti and RTX 4090, respectively.*



(a) RTX 2080 Ti



(b) RTX 4090

Figure 12. *The training throughput of CLM training vs. GPU-only baselines. For each scene-testbed pair, we use the largest model size supported by the baselines as assessed in Figure 8 to prevent OOM. Comparing CLM to enhanced baseline eliminates the effect of pre-rendering frustum culling. On RTX 2080 Ti, CLM reaches 86% (BigCity) to 97% (Ithaca) of the enhanced baseline's throughput; on RTX 4090, it achieves 55% (Ithaca) to 90% (Bicycle).*

6.3 Performance

We compare CLM's training performance to that of naive offloading to evaluate the effectiveness of CLM's design choices. We also quantify CLM's offloading overhead by comparing its performance to GPU-only baselines. We measure performance by training throughput, calculated as the number of images processed per second during training.

CLM vs. naive offloading. We evaluate CLM's performance compared to naive offloading on both testbeds. Our evaluation covers all scenes in Table 3. For each scene, we use the largest model size that could be trained using naive offloading on the given testbed (Figure 8). As can be seen in Figure 11, CLM achieves significant speedup over naive offloading. In particular, for the largest scene BigCity, CLM is 1.92x faster than naive offloading when run on the 2080 Ti, and 1.58x faster on the 4090. As can been observed, CLM's speedups differ between the two testbeds. As RTX 2080 Ti has roughly 7 times fewer FLOPS than the 4090, the GPU computation takes longer on the 2080 Ti, allowing CLM to overlap and hide more of the offloading overhead than on the RTX 4090. This effect is particularly observable in large scenes such as BigCity with high offloading overheads, thus CLM has larger improvements when run on a slow GPU than on a faster GPU.

CLM vs. GPU-only training. We evaluate the performance of CLM compared to GPU-only baselines. To avoid OOM, for each scene in Table 3, we use the maximum model size that can be trained using the baseline (Figure 8).

As shown in Figure 12, CLM achieves similar or much better throughput than the naive baseline on both testbeds. The unexpected improvement compared to the baseline is due to CLM's use of pre-rendering frustum culling as explained in §5.1. This technique enables more efficient computation in scenes with low ρ (e.g., BigCity) by culling more points, resulting in notable performance improvements.

We can evaluate CLM's offloading overheads more fairly by comparing to the enhanced baseline, which also uses (and benefits from) pre-rendering frustum culling. As can be seen in Figure 12, CLM achieves 86% (BigCity) to 97% (Ithaca) of the enhanced baseline's throughput on the RTX 2080 Ti, and 55% (Ithaca) to 90% (Bicycle) on the RTX 4090. The slowdown occurs for CLM because the communication and CPU Adam computation cannot be fully overlapped with GPU computation. As expected, the offloading overheads depend on both testbed and scene characteristics. Among the two testbeds, we observe larger overheads on the RTX 4090 than the RTX 2080 Ti because the longer GPU computation time on the 2080 allows CLM to more effectively overlap communication and CPU Adam computation. In terms of scenes, we find that scenes that require more GPU computation (e.g., because they have higher resolution) or where less communication is required (e.g., because of lower ρ) have smaller overheads, because communication and CPU Adam computation can be more effectively overlapped. For example, the Rubble and Bicycle scenes which are at 4K resolution have a slowdown of less than 20% compared to the enhanced baseline, while Ithaca which has the lowest resolution among the scenes we evaluated (Table 2) has a slowdown of 45%.

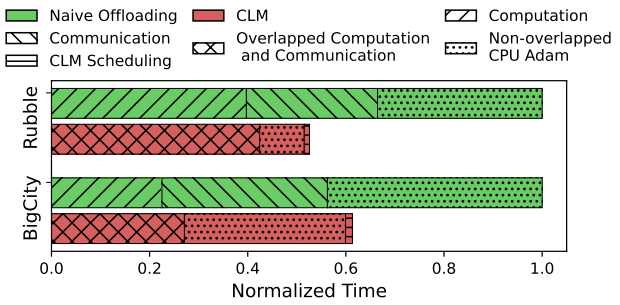


Figure 13. *Runtime Decomposition for Rubble and BigCity using CLM and naive offloading on the 4090 testbed. We normalize using the total time of naive offloading in each scene.*

Running time breakdown. To identify the source of CLM's speedup over naive offloading, we profile the runtime breakdowns for the Rubble and BigCity scenes on RTX 4090 testbed, as shown in Figure 13. We have the following observations. (1) We observe significant communication and CPU Adam overheads in naive offloading, as discussed in §2.2. In both scenes, two overheads together account for more than 50% of the training time. (2) In this figure, CLM's pipeline running time takes into account both communication and computation, as their runtime cannot be well separated. And we observe that CLM's pipeline runtime is notably shorter than the combined computation and communication duration in naive offloading. The overall acceleration results from overlapping communication with computation, along with reduced communication volume by transferring only in-frustum Gaussians. We also observe that CLM's pipeline time, which includes both computation and communication, is only marginally longer than the naive offloading's computation-only time. This indicates that CLM's communication overhead on top of computation is very small. (3) The effect of overlapping CPU Adam differs depending on the scenes. Figure 13 illustrates that CLM reduces CPU Adam latency more in Rubble than in BigCity through overlapping. This disparity arises because, first, CPU Adam requires more time in BigCity due to the increased number of Gaussians. Second, the lower resolution of BigCity means that its computation time is shorter than that of Rubble (see Figure 11), thus making it harder for CPU Adam to overlap. The lesser overlap of CPU Adam in BigCity explains why CLM does not achieve the greatest speedup compared to naive offloading in BigCity, despite its lowest ρ among all scenes. (4) Lastly, Figure 13 shows that the scheduling overheads in CLM involved in determining in-frustum Gaussian indices and computing the microbatch order based on TSP are marginal.

Communication volume reduction. To better understand the communication overhead of CLM, we collect the average communication volume per training batch, as shown in Figure 14. The experiments for each scene use the maximum model size of naive offloading from Figure 8b. We compare CLM against naive offloading, and conduct ablation studies. Specifically, we evaluate a CLM variant without Gaussian Caching (No Cache), as well as three variants with Gaussian Caching enabled, each of which uses a different ordering strategy: "Random Order", "Camera Order" and "GS Count Order", as described in Table 4. In contrast, CLM uses TSP order (see Section 4.2.3). These ablations help clarify how both caching and rendering order affect communication volume during training. We report the size of parameters transferred from CPU to GPU memories in GB.

Strategy	Description
Random Order	Shuffle views uniformly at random.
Camera Order	Sort views by their camera-center coordinate along the scene's principal axis.
GS Count Order	Sort descending by the number of Gaussians visible in each view. Prioritizing views with more Gaussians allows CPU Adam to update more Gaussians earlier, reducing its overhead.
TSP Order (CLM)	Use TSP to find an order which maximizes Gaussian overlap between successive views.

Table 4. *Ordering strategies evaluated in our ablation study. The "Random Order" and "Camera Order" are straightforward; while both "GS Count Order" and "TSP Order" rely on view-Gaussian visibility information and thus require additional processing.*

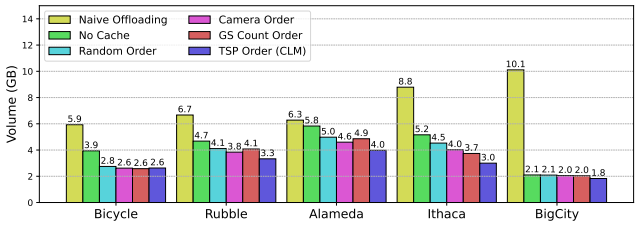


Figure 14. *Average communication volume as measured by bytes transferred from CPU to GPU per training batch. "Naive Offloading" denotes the communication volume for naive offloading without any optimization. "No Cache" denotes the volume for CLM without Gaussian Caching and Order Optimization. The remaining four correspond to the ordering strategies detailed in Table 4.*

Figure 14 shows that CLM consistently decreases communication by 37% (Alameda) to 82% (BigCity) over naive offloading. In BigCity, the technique of selectively loading in-frustum Gaussians by itself significantly decreases communication (79%), whereas Gaussian Caching offers small benefit, i.e. CLM has 12% additional reduction over "No Cache". This is because BigCity has a very low ρ (§5), resulting in fewer Gaussians shared between two images for caching. Conversely, in scenes where each image contains a greater proportion of all Gaussians, like Bicycle, the use of Gaussian Caching yields more significant benefits, i.e. CLM has 33% additional reduction over "No Cache". We also observe that the TSP order consistently results in the lowest communication volume by maximizing cache reuse across microbatches. The greatest reduction is seen on the Ithaca scene — 34% lower than "Random Order", 25% lower than "GS Count Order", and 19% lower than "Camera Order".

Effectiveness of different ordering-strategies. In Table 5a, we compare the average training throughput (measured in

Method	Bicycle	Rubble	Alameda	Ithaca	BigCity
Random Order	3.95	6.23	7.52	12.36	40.89
Camera Order	3.94	6.27	7.58	12.67	**40.94**
GS Count Order	**4.06**	**6.65**	8.01	12.50	40.80
TSP Order	4.03	6.64	**8.24**	**12.77**	40.74

(a) *Training throughput (img/s)*

Method	Bicycle	Rubble	Alameda	Ithaca	BigCity
Random Order	147.457	137.27	178.59	**208.34**	21.85
Camera Order	150.831	144.87	171.32	305.71	**21.05**
GS Count Order	**119.383**	**81.27**	**141.91**	268.62	21.93
TSP Order	147.457	127.11	186.75	406.79	23.36

(b) *CPU Adam trailing time (ms)*

Table 5. *Average training throughput and CPU Adam trailing time under different ordering strategies (see Table 4). The trailing time is calculated as the time spent by CPU Adam after the last gradient has been transferred from the GPU to the CPU. The more sophisticated strategies, "TSP Order" and "GS Count Order", deliver the highest end-to-end throughput. "GS Count Order" incurs the least CPU Adam trailing time; while Figure 14 shows that "TSP Order" achieves the greatest reduction in communication volume.*

processed images per second) across four ordering strategies in Table 4: "Random Order", "Camera Order", "GS Count Order" and "TSP order". The experiments for each scene use the maximum model size supported by naive offloading from Figure 8b on the 4090 testbed. We also report the corresponding communication volumes in Figure 14 and CPU Adam "trailing time" in Table 5b. We calculate "trailing time" as the time from when the last gradients are transferred to CPU memory to when CPU Adam finishes for the batch. Overall, smart reordering consistently improves training throughput over the default "Random Order", with the most significant gain observed on the Alameda scene—achieving a 10% speedup. In contrast, BigCity shows minimal variation across orders in terms of communication volume, CPU Adam trailing time, and thus overall throughput. Among the strategies, "TSP Order" and "GS Count Order" achieve the highest throughput. "TSP Order" consistently minimizes communication volume, while "GS Count Order" reduces CPU Adam trailing time by rendering images that use more Gaussians earlier. An non-intuitive observation is that, for the Ithaca scene, the naive "Random Order" exhibits lower CPU Adam trailing time than "GS Count Order". This is because its communication is significantly slower than the other strategies (see Figure 14). The slower transfer allows more opportunity to overlap with CPU-side computation, thereby reducing its trailing time.

6.4 Hardware Utilization

GPU utilization. We compare the GPU utilization of CLM against naive offloading by profiling using Nsight Systems [39] on the RTX 4090. We collect the `SMs Active` metric at 10 kHz from Nsight Systems in the `GPU_METRICS` table. The values range from 0 to 100 and reflects the percentage of SMs with active warps in flight. A value of 0 indicates that all SMs are idle. Figure 15 presents the Cumulative Distribution Function (CDF) of the GPU idle rate, computed as 100 - `SMs Active`. The x-axis denotes the idle rate, while the y-axis represents the percentage of time. The area under the curve corresponds to the expected value of `SMs Active` during training, reflecting average GPU utilization. For each scene, both CLM and naive offloading are profiled for the same duration—spanning more than 100 batches. We use the maximum model size for naive offloading from Figure 8b. We observe that CLM consistently achieves better GPU utilization, as indicated by higher curves. Additionally, scenes with higher resolution (e.g., Bicycle and Rubble) exhibit better utilization compared to lower-resolution scenes (e.g., Ithaca and BigCity), confirming the intuition that higher-resolution rendering is more computational intensive.

Testbed	Bicycle	Rubble	Alameda	Ithaca	BigCity
RTX 2080 Ti	6.0	8.2	8.4	13.4	17.5
RTX 4090	14.1	17.2	16.1	28.4	37.8

Table 6. *Pinned memory usage (in GB) of CLM for each scene using the maximum model size reported in Figure 8.*

Pinned memory usage. Pinned memory is a limited resource, so we report CLM 's usage on the two testbeds in Table 6, using the corresponding maximum model sizes shown in Figure 8. Even for the largest BigCity model, pinned memory usage peaks at 17.5 GB on the RTX 2080 Ti testbed and 37.8 GB on the RTX 4090—under 10% of the RTX 2080 Ti testbed's 256 GB RAM and 30% of the RTX 4090 testbed's 128 GB RAM, respectively. This efficiency stems from pinning only parameter and gradient tensors (which require GPU DMA) in CLM, while optimizer and auxiliary states remain unpinned. We observe no system performance degradation from this usage.

We further report CPU cores, PCIe bandwidth and GPU memory bandwidth utilization in Appendix A.4.

7 Related Works

Current Approaches to Scaling 3DGS. Several approaches have been suggested to decrease the GPU memory usage of 3DGS. First, systems like [9, 30, 51] use the aggregate memory of multiple GPUs to distribute 3DGS training. However, the need for multiple GPUs and high-performance interconnects adds significant costs, putting these approaches out of reach for most users. Second, some approaches prune Gaussians [14, 15, 17, 40, 50] that do not contribute significantly to the rendered image. Although pruning methods are effective at reducing memory overheads, they can potentially hurt fidelity [17, 50] when Gaussians are pruned aggressively. Our approach is orthogonal, and does not affect fidelity. Furthermore, for very large scenes, even a pruned model might not fit in a single-GPU's memory, making it necessary to combine pruning with approaches such as ours. Furthermore, most of these approaches prune Gaussians after training, and thus cannot be used to scale 3DGS training. Third, divide-and-conquer

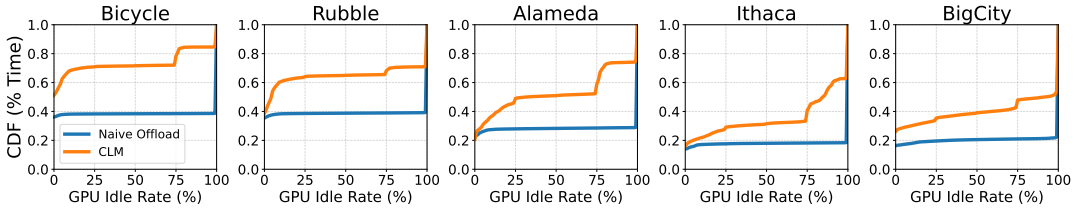


Figure 15. *CDF of GPU Idle Rate (100 - `SMs Active`) Across Scenes for CLM vs. Naive Offloading. Higher curves indicate more time spent at lower idle rates, reflecting better GPU utilization. CLM consistently achieves higher GPU utilization across all scenes.*

methods [24, 32, 35] partition scenes into smaller regions, process each partition in isolation, and finally reconstruct views by combining results from multiple partitions. But Gaussians at partition borders are likely to be used by multiple partitions, and in practice this leads to rendering inconsistencies at partition boundaries and lower reconstruction quality. Furthermore, these techniques are often more complex to use because of additional hyperparameters to tune.

CPU offloading in other ML training system. As the size of ML models increases, the limitations of GPU memory have led to a greater use of CPU offloading techniques for ML training. For example, [21, 41, 42] focus on DNN training in general, while [4, 37] specialize in recommendation systems, [2, 46] in GNNs, and [48] in LLMs, all of which optimize for workloads distinct from 3DGS. Their focused workloads involve tensor operations, such as GEMM; whereas 3DGS is unique differentiable rendering pipeline. There are no studies that have successfully applied these offloading techniques to 3DGS training. In contrast, we demonstrate the feasibility of offloading in 3DGS and introduce tailored designs for improved efficiency. UGache [44] optimizes the sparse access to CPU-based embedding tables, which is common in the offloaded training for the GNN and the recommendation system. Our Gaussian attributes in CPU memory akins to embedding table and is also accessed sparsely. However, UGache assumes that the embedding table is read-only, which is not applicable in 3DGS training. Moreover, UGache overlooks aspects such as spatial locality that are present in 3DGS, thereby missing some opportunities for optimization.

Other 3DGS Training System. Recent studies have developed other hardware and software systems to improve 3DGS Training. GauSPU [47], GScore [29] and MetaSapiens [33], ACR [13] design hardware accelerators specifically for the rendering pipeline in 3DGS training. However, these accelerators are mainly optimized for speed or energy efficiency. Unlike these, CLM emphasizes optimizing GPU memory efficiency. Additionally, our offloading methods may complement these accelerators by allowing a novel view synthesis task to utilize them for speed improvement while using our techniques for extra memory capacity.

8 Discussion and Future Work

Finally, we discuss how CLM can be generalized to other rendering methods and backends, and future directions that use spatial data structures to further aid with scaling.

Support for other rendering backends and methods. CLM is backend-agnostic because it determines where to store data (offloading), how to transfer it (overlapping), and when to render each image (pipelining and ordering), without depending on the specific rendering procedure. This decoupling enables seamless integration with APIs such as Vulkan and alternative rendering approaches like ray tracing, without requiring changes to CLM 's scheduling or offloading logic. Furthermore, the core pipeline naturally extends to a broader class of point-based differentiable rendering techniques, such as 2D Gaussian Splatting [20] and 3D Convex Splatting [19], due to their similar reliance on sparse data access patterns induced by frustum culling. However, CLM cannot generalize to non-point-based novel view synthesis methods, like NeRF[36].

Integration of spatial data structures. As scenes grow larger and more complex, the number of Gaussians increases significantly. Although naive frustum culling—iterating over every Gaussian—is not a yet bottleneck in our current evaluation, it could eventually become one as its time complexity scales linearly with the number of Gaussians. Future work could explore integrating spatial acceleration structures, such as bounding volume hierarchies (BVHs), to organize Gaussians more efficiently and thereby improving frustum culling performance by skip non-intersected regions.

Portability to other GPUs. Our CLM implementation relies on two CUDA features of NVIDIA GPUs: pinned memory for direct memory access (DMA) and multi-streaming to overlap data transfer with computation. Both pinned-memory DMA and multi-streaming are standard features in modern GPUs (e.g., in AMD ROCm[1]), and thus do not fundamentally limit portability.

9 Conclusion

Our goal in designing CLM was to allow 3DGS to be used with larger scenes, without needing to compromise on rendering quality or pay for multi-GPU training. We were able to meet our goals because of the inherent sparsity of 3DGS's computation and its memory access pattern, that allowed us to overlap

GPU-CPU communication, GPU computation and CPU computation. Our approach does not depend on details of how views are rasterized or what kernels are used, and therefore it can be applied to other novel view synthesis or ML methods that exhibit similar computation and memory access patterns.

Acknowledgments

We thank the ASPLOS reviewers for their comments. This research was funded in part by a gift from Impossible, Inc.

References

[1] [n. d.]. *ROCm: An Open-Source Platform for GPU Computing.* https://github.com/ROCm/ROCm

[2] Xin Ai, Qiange Wang, Chunyu Cao, Yanfeng Zhang, Chaoyi Chen, Hao Yuan, Yu Gu, and Ge Yu. 2024. NeutronOrch: Rethinking Sample-Based GNN Training under CPU-GPU Heterogeneous Environments. *Proc. VLDB Endow.* 17, 8 (April 2024), 1995–2008. doi:10.14778/3659437.3659453

[3] Muhammad Salman Ali, Chaoning Zhang, Marco Cagnazzo, Giuseppe Valenzise, Enzo Tartaglione, and Sung-Ho Bae. 2025. Compression in 3D Gaussian Splatting: A Survey of Methods, Trends, and Future Directions. arXiv:2502.19457 [cs.GR] https://arxiv.org/abs/2502.19457

[4] Keshav Balasubramanian, Abdulla Alshabanah, Joshua D Choe, and Murali Annavaram. 2021. cDLRM: Look Ahead Caching for Scalable Training of Recommendation Models. In *Proceedings of the 15th ACM Conference on Recommender Systems* (Amsterdam, Netherlands) *(RecSys '21)*. Association for Computing Machinery, New York, NY, USA, 263–272. doi:10.1145/3460231.3474246

[5] Yanqi Bao, Tianyu Ding, Jing Huo, Yaoli Liu, Yuxin Li, Wenbin Li, Yang Gao, and Jiebo Luo. 2025. 3D Gaussian Splatting: Survey, Technologies, Challenges, and Opportunities. *IEEE Transactions on Circuits and Systems for Video Technology* (2025), 1–1. doi:10.1109/TCSVT.2025.3538684

[6] Jonathan Barron, Ben Mildenhall, Dor Verbin, Pratul Srinivasan, and Peter Hedman. 2022. Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields. In *CVPR*. doi:10.1109/CVPR52688.2022.00539

[7] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. 2023. Zip-NeRF: Anti-Aliased Grid-Based Neural Radiance Fields. arXiv:2304.06706 [cs.CV] https://arxiv.org/abs/2304.06706

[8] Guikun Chen and Wenguan Wang. 2024. A Survey on 3D Gaussian Splatting. arXiv:2401.03890 [cs.CV]

[9] Yu Chen and Gim Hee Lee. 2024. DOGS: Distributed-Oriented Gaussian Splatting for Large-Scale 3D Reconstruction Via Gaussian Consensus. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=HAocQ9dSAX

[10] Steven Chien, Ivy Peng, and Stefano Markidis. 2019. Performance Evaluation of Advanced Features in CUDA Unified Memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 50–57. doi:10.1109/MCHPC49590.2019.00014

[11] Georges A Croes. 1958. A method for solving traveling-salesman problems. *Operations research* 6, 6 (1958), 791–812.

[12] Carlos A. Diaz-Ruiz, Youya Xia, Yurong You, Jose Nino, Junan Chen, Josephine Monica, Xiangyu Chen, Katie Luo, Yan Wang, Marc Emond, Wei-Lun Chao, Bharath Hariharan, Kilian Q. Weinberger, and Mark Campbell. 2022. Ithaca365: Dataset and Driving Perception under Repeated and Challenging Weather Conditions. arXiv:2208.01166 [cs.CV] https://arxiv.org/abs/2208.01166

[13] 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 *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 64–83. doi:10.1145/3669940.3707238

[14] Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, and Zhangyang Wang. 2024. LightGaussian: Unbounded 3D Gaussian Compression with 15x Reduction and 200+ FPS. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=6AelDnrTN2

[15] Guangchi Fang and Bing Wang. 2024. Mini-Splatting: Representing Scenes with a Constrained Number of Gaussians. In *European Conference on Computer Vision*. https://api.semanticscholar.org/CorpusID:268554047

[16] Merrill M Flood. 1956. The traveling-salesman problem. *Operations research* 4, 1 (1956), 61–75.

[17] Alex Hanson, Allen Tu, Geng Lin, Vasu Singla, Matthias Zwicker, and Tom Goldstein. 2024. Speedy-Splat: Fast 3D Gaussian Splatting with Sparse Pixels and Sparse Primitives *(CVPR 2025)*.

[18] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *CoRR* abs/1806.03377 (2018). arXiv:1806.03377 http://arxiv.org/abs/1806.03377

[19] Jan Held, Renaud Vandeghen, Abdullah Hamdi, Adrien Deliege, Anthony Cioppa, Silvio Giancola, Andrea Vedaldi, Bernard Ghanem, and Marc Van Droogenbroeck. 2024. 3D Convex Splatting: Radiance Field Rendering with 3D Smooth Convexes. arXiv:2411.14974 [cs.CV] https://arxiv.org/abs/2411.14974

[20] Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. 2024. 2D Gaussian Splatting for Geometrically Accurate Radiance Fields. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers '24 (SIGGRAPH '24)*. ACM, 1–11. doi:10.1145/3641519.3657428

[21] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1341–1355. doi:10.1145/3373376.3378530

[22] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*. https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[23] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Transactions on Graphics* 42, 4 (July 2023). https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

[24] Bernhard Kerbl, Andreas Meuleman, Georgios Kopanas, Michael Wimmer, Alexandre Lanvin, and George Drettakis. 2024. A Hierarchical 3D Gaussian Representation for Real-Time Rendering of Very Large Datasets. *ACM Transactions on Graphics* (2024). https://repo-sam.inria.fr/fungraph/hierarchical-3d-gaussians/

[25] Shakiba Kheradmand, Daniel Rebain, Gopal Sharma, Weiwei Sun, Jeff Tseng, Hossam Isack, Abhishek Kar, Andrea Tagliasacchi, and Kwang Moo Yi. 2025. 3D Gaussian Splatting as Markov Chain Monte Carlo. arXiv:2404.09591 [cs.CV] https://arxiv.org/abs/2404.09591

[26] Khronos Group. [n. d.]. *Vulkan.* https://www.vulkan.org/

[27] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference for Learning Representations (ICLR)*.

[28] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG] https://arxiv.org/abs/1412.6980

[29] Junseo Lee, Seokwon Lee, Jungi Lee, Junyong Park, and Jaewoong Sim. 2024. GSCore: Efficient Radiance Field Rendering via Architectural

Support for 3D Gaussian Splatting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 497–511. doi:10.1145/3620666.3651385

[30] 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. arXiv:2406.11836 [cs.CV] https://arxiv.org/abs/2406.11836

[31] 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 *ICCV*.

[32] 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 *CVPR*.

[33] Weikai Lin, Yu Feng, and Yuhao Zhu. 2025. MetaSapiens: Real-Time Neural Rendering with Efficiency-Aware Pruning and Accelerated Foveated Rendering. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25)*. ACM, 669–682. doi:10.1145/3669940.3707227

[34] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. 2019. Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning. arXiv:1904.01786 [cs.CV]

[35] Yang Liu, He Guan, Chuanchen Luo, Lue Fan, Junran Peng, and Zhaoxiang Zhang. 2024. CityGaussian: Real-time High-quality Large-Scale Scene Rendering with Gaussians. In *CVPR*.

[36] 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 *ECCV*.

[37] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2022. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 993–1011. doi:10.1145/3470496.3533727

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

[39] NVIDIA Corporation. 2025. *NVIDIA Nsight Systems*. https://developer.nvidia.com/nsight-systems A system-wide performance analysis tool for CPU/GPU profiling.

[40] Panagiotis Papantonakis, Georgios Kopanas, Bernhard Kerbl, Alexandre Lanvin, and George Drettakis. 2024. Reducing the Memory Footprint of 3D Gaussian Splatting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 7, 1 (May 2024), 1–17. doi:10.1145/3651282

[41] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. arXiv:2101.06840 [cs.DC] https://arxiv.org/abs/2101.06840

[42] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 18, 13 pages.

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

[44] Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. 2023. UGACHE: A Unified GPU Cache for Embedding-based Deep Learning. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 627–641. doi:10.1145/3600006.3613169

[45] Haithem Turki, Deva Ramanan, and Mahadev Satyanarayanan. 2022. Mega-NERF: Scalable Construction of Large-Scale NeRFs for Virtual Fly-Throughs. In *CVPR*.

[46] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2020. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. arXiv:1909.01315 [cs.LG] https://arxiv.org/abs/1909.01315

[47] Lizhou Wu, Haozhe Zhu, Siqi He, Jiapei Zheng, Chixiao Chen, and Xiaoyang Zeng. 2024. GauSPU: 3D Gaussian Splatting Processor for Real-Time SLAM Systems. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1562–1573. doi:10.1109/MICRO61859.2024.00114

[48] Jinghan Yao, Sam Ade Jacobs, Masahiro Tanaka, Olatunji Ruwase, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. 2024. Training Ultra Long Context Language Model with Fully Pipelined Distributed Transformer. arXiv:2408.16978 [cs.DC] https://arxiv.org/abs/2408.16978

[49] Vickie Ye and Angjoo Kanazawa. 2023. `gsplat` Library. arXiv:2312.02121 [cs.MS]

[50] Zhaoliang Zhang, Tianchen Song, Yongjae Lee, Li Yang, Cheng Peng, Rama Chellappa, and Deliang Fan. 2024. LP-3DGS: Learning to Prune 3D Gaussian Splatting. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=kzJ9P7VPnS

[51] Hexu Zhao, Haoyang Weng, Daohan Lu, Ang Li, Jinyang Li, Aurojit Panda, and Saining Xie. 2025. On Scaling Up 3D Gaussian Splatting Training. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=pQqeQpMkE7

A Appendix

A.1 Search for TSP solution

We formulate the training image reordering task into a TSP problem, in which each training order corresponds to a tour in the TSP instance. We implement a Stochastic Local Search with well-established greedy heuristics (2-opt and 3-opt [11] in our case). Our algorithm starts with an initial feasible tour, and then iteratively improves the tour by applying local greedy swapping. Our initialized tour is as follows: starts from a randomly chosen city and repeatedly selects the nearest unvisited city as the next destination. In the initialized tour, the nodes are connected by edges to form a chain. During every iterative improvement, we select 2 or 3 edges and remove these edges and reconnect the segments in a new way that reduces the tour length. We perform swaps until no further improvement is found or an adjustable time limit is reached. In our experiments, we use 1 ms as the time limit which is empirically sufficient for us to find an optimal solution (compared to DP-based method). The impressive results could be attributed to the relatively small batch size (the number of nodes in TSP) and ours is a variant of the TSP known as the metric TSP, which is typically easier to address empirically. The metric TSP mandates that the distance function is symmetric and fulfills the triangle inequality, which the symmetric distance complies with.

A.2 Dataset Preparation

3DGS training requires a camera pose for each image. The Bicycle[6], Rubble[45], Alameda[7], and Matrixcity [31] datasets initially support novel view synthesis and come with camera poses. However, the Ithaca dataset, initially designed for autonomous driving, does not include camera poses, so we use colmap [43] to generate camera poses and the point cloud (used for initializing Gaussians) ourselves.

The matrixcity dataset [31] comprises sub-scenes of varying sizes and offers two perspectives: aerial and street. For our evaluation, matrixcity BigCity refers to the largest one among the aerial views scenes.

We conducted experiments on all scenes at their native resolutions, without image downsampling.

A.3 Fragmentation decreases the available memory for accommodating Gaussians

Given most 3DGS training pipeline (and ours) are built on pytorch, another factor that limits available memory for training is memory fragmentation due to the Pytorch Cache Allocator. The PyTorch Cache Allocator manages GPU memory by maintaining a pool of allocated memory blocks to reduce the overhead of frequent memory allocations and deallocations. Although this approach improves speed in many scenarios, it can lead to fragmentation over time, especially in workloads with varying allocation sizes. 3DGS exactly exhibits varying activation states during training, and their model states are frequently densified and pruned, leading to substantial fragmentation challenges. This fragmentation further reduces available GPU memory, hindering the accommodation of additional Gaussian parameters.

A.4 Additional Hardware Utilization

Scene	Metric	Naive offloading (%)	CLM (%)
Bicycle	CPU Util	18.68	**68.98**
	DRAM Read	9.61	**16.72**
	DRAM Write	7.64	**12.55**
	PCIe RX	12.62	**20.85**
	PCIe TX	14.18	**14.20**
Rubble	CPU Util	21.47	**64.83**
	DRAM Read	10.29	**15.86**
	DRAM Write	7.79	**11.52**
	PCIe RX	12.38	**20.94**
	PCIe TX	13.59	**14.09**
Alameda	CPU Util	22.64	**76.80**
	DRAM Read	7.27	**10.61**
	DRAM Write	6.03	**8.34**
	PCIe RX	14.47	**30.88**
	PCIe TX	16.07	**20.53**
Ithaca	CPU Util	24.97	**82.44**
	DRAM Read	8.01	**12.37**
	DRAM Write	5.17	**7.37**
	PCIe RX	16.78	**17.61**
	PCIe TX	**19.11**	12.57
BigCity	CPU Util	25.24	**61.95**
	DRAM Read	8.84	**16.14**
	DRAM Write	2.89	**5.17**
	PCIe RX	**15.37**	10.13
	PCIe TX	**16.97**	7.13

Table 7. *Hardware Utilization of CLM and Naive Offloading Across Five Scenes on RTX 4090.* `CPU Util` *refers to CPU cores utilization.* `DRAM Read/Write` *indicate GPU memory bandwidth utilization.* `PCIe RX` *represents PCIe CPU-to-GPU direction utilization, and* `PCIe TX` *represents GPU-to-CPU direction utilization. All values are percentages of utilization, ranging from 0 to 100. In each row, the bold figure is the one with the higher utilization.*

We additionally report the utilization of CPU cores, GPU DRAM bandwidth and PCIe bandwidth for both CLM and naive offloading across all scenes on RTX 4090, as shown in Table 7. To obtain CPU utilization, we extract thread scheduling events from Nsight Systems' `SCHED_EVENTS` table, which logs timestamps for entering and leaving each CPU core. We calculate the percentage of time each core has a thread in flight, then average across all cores to obtain overall CPU utilization. Additionally, we collect other metrics at a sampling rate of 10 kHz from Nsight Systems' `GPU_METRICS` table: `DRAM Read Bandwidth`, `DRAM Write Bandwidth`, `PCIe RX`, and `PCIe TX`. These metrics reflect the read and write bandwidth utilizations for both GPU Memory and PCIe, respectively. All utilization values are percentages, ranging from 0 to 100.

For CPU core utilization, CLM consistently achieves higher usage than naive offloading. This is because CLM overlaps CPU-side Adam optimization—the primary CPU workload—with GPU computation and communication. In contrast, naive offloading leaves most CPU cores idle while the GPU is computing or transferring Gaussians between CPU and GPU memory.

For DRAM bandwidth, CLM consistently exhibits higher utilization than naive offloading. This is because both approaches perform the same amount of memory access (as the rendering operations are the same), but CLM consistently runs faster, resulting in higher bandwidth utilization over time.

For PCIe utilization, CLM generally shows higher values than naive offloading, except in Ithaca's `PCIe TX` and BigCity's `PCIe RX` and `PCIe TX`. In these 3 cases, naive offloading transfers significantly more data than CLM in each batch (see Table 14), leading to higher utilization. Notably, CLM achieves higher PCIe utilization in most other cases despite transferring less data. We also observe that `PCIe RX` (CPU-to-GPU) utilization in CLM is consistently higher than `PCIe TX` (GPU-to-CPU), due to gradient accumulation in CLM: old gradients are loaded from CPU pinned memory to each CUDA kernel via DMA, summed with new gradients, and written back. This results in bidirectional PCIe usage during gradient offloading, whereas parameter loading is unidirectional from CPU to GPU. Lastly, overall PCIe utilization is low, because the CPU/GPU may be busy and not sending data, and the transfers may be too sparse to fully saturate the bandwidth even when PCIe is active.