Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Loss-Guided Density Control and Efficient Backward Optimisation for Compact Gaussian Splatting

Xiaofeng Che
Chengdu University of Technology
Chen Wang
cwang@cdut.edu.cn
Chengdu University of Technology
Haoran Zhou
Shenzhen Sensing Data Technology Company Limited
Guangle Yao
Chengdu University of Technology
Xulong Liu
JINGTIAN & GONGCHENG
Shaoze Ye
Shenzhen Sensing Data Technology Company Limited

Research Article

Keywords: Novel View Synthesis, 3D Gaussian splatting, Efficient Training, Loss-Guided Density Control, Efficient CUDA Backward Optimisation

Posted Date: October 19th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10572680/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Loss-Guided Density Control and Efficient Backward Optimisation for Compact Gaussian Splatting

Xiaofeng Che[1], Chen Wang[2*], Haoran Zhou[3], Guangle Yao[4], Xulong Liu[5], Shaoze Ye[6]

[1,2,4]College of Computer Science and Cybersecurity, Chengdu University of Technology, Chengdu, 610051, China.
[3,6]Shenzhen Sensing Data Technology Company Limited, Shenzhen, 518000, China.
[5]JINGTIAN & GONGCHENG, Beijing, 100020, China.

*Corresponding author(s). E-mail(s): cwang@cdut.edu.cn;
Contributing authors: binarystar671@foxmail.com; hr@sensing-sz.com;
yaoguangle19@cdut.edu.cn; liu.xulong@jingtian.com;
372767021@qq.com;

Abstract

Novel view synthesis based on explicit Gaussian primitives has achieved an effective balance between rendering quality and real-time visualisation, but its training process can remain computationally expensive because adaptive density control may introduce many redundant primitives. This paper proposes an efficient training framework for compact Gaussian Splatting through loss-guided densification, opacity-gradient pruning, and backward-pass optimisation. First, a loss accumulation guided density-control strategy is introduced. Instead of relying only on image-plane gradient magnitude, the method reuses the loss map generated during standard backpropagation to identify persistently under-reconstructed image regions. A binary high-loss mask is accumulated across multiple iterations, and each Gaussian primitive receives a densification score according to its coverage of these regions. Only primitives that consistently contribute to high-error areas are allowed to split or clone, thereby reducing unnecessary growth in well-reconstructed regions. Second, an opacity-gradient-based pruning criterion is designed to remove unstable or low-contribution primitives after densification. By combining opacity,

opacity gradients, and training loss over multiple views, the proposed pruning strategy aims to preserve useful primitives while suppressing redundant ones. Third, the CUDA backward pass is optimised through warp-level parallelism and vectorised memory access to reduce redundant global memory transactions. Experiments on Mip-NeRF 360, Deep Blending, and Tanks & Temples indicate that the proposed framework substantially reduces training time and the final number of Gaussian primitives while maintaining competitive PSNR, SSIM, and LPIPS performance compared with representative efficient Gaussian-splatting baselines. The results suggest that loss-aware density control can provide a practical route towards faster and more compact Gaussian Splatting. Our project page is available at https://github.com/Allen-key-20/Loss-Guided-Density-Control-and-Efficient-Backward-Optimisation.git

Keywords: Novel View Synthesis, 3D Gaussian splatting, Efficient Training, Loss-Guided Density Control, Efficient CUDA Backward Optimisation

1 Introduction

Novel view synthesis (NVS) is a fundamental task in computer vision and computer graphics, which is critical for downstream applications such as augmented reality (AR), virtual reality (VR), and digital twins. Early neural scene representations, notably Neural Radiance Fields (NeRF) [1], demonstrated impressive visual fidelity but were constrained by prohibitive computational demands. Recently, 3D Gaussian Splatting [2] (3DGS) has revolutionised this field by achieving state-of-the-art rendering quality and real-time performance through explicit Gaussian primitives. Owing to this efficiency, 3DGS has gained significant traction across various visual computing domains, including dynamic scene modelling [3], autonomous driving [4], and SLAM [5].

Despite these breakthroughs, the widespread adoption of 3DGS remains hindered by training overhead [6]. In particular, Adaptive Density Control continuously introduces new Gaussians as training progresses. If densification and pruning are not well designed, a large number of redundant Gaussians can be produced, which increases computational cost and reduces training efficiency [7]. Therefore, controlling the number of Gaussians while preserving reconstruction quality is important for reducing training time.

A limitation of existing density control strategies is that their criteria are not directly aligned with the optimisation objective. Conventional methods usually rely on image-plane gradients to guide densification. However, these signals mainly capture local variations and do not reflect the global loss distribution well. Some approaches [8, 9] try to estimate loss through additional sampling, which gives a more direct indication of high error regions. However, this requires repeated forward rendering and introduces extra computational cost.

To address these issues, we propose a Loss Accumulation Guided (LAG) densification strategy. Unlike existing methods that require external estimation, our method

directly leverages the loss maps generated during the standard training pipeline. During the backward pass, we compute a single-channel binary mask to indicate which pixels already have small reconstruction errors. For each Gaussian, we count how many of its covered pixels have errors above a threshold and average this count over K iterations to obtain a Densification Score (DS). A Gaussian with a low DS mainly covers well-reconstructed pixels; splitting or cloning it is unnecessary because the optimisation objective for those pixels has already been met. Conversely, Gaussians with a high DS are allowed to densify. This principle suppresses unnecessary Gaussian growth in regions where the reconstruction is already adequate, without requiring additional rendering passes.

To further refine the scene representation, we introduce a pruning strategy after the densification stage. This strategy constructs a pruning metric by computing, on a per-pixel basis, the ratio of the opacity gradient to the opacity value. The metric is aggregated across the Gaussian's coverage and weighted by the current iteration's training loss. By accumulating these scores over K iterations to obtain a pruning score (PS), we remove Gaussians that exceed a threshold. This mechanism is integrated with opacity-based pruning to discard Gaussians with low rendering contribution.

In addition, we modify the CUDA implementation of the 3DGS backward pass kernel. By redesigning thread organisation for warp-level data reuse and adjusting memory layouts for vectorised access, we reduce redundant global memory transactions and improve cache utilisation. These low-level optimisations reduce the overhead of executing the backward pass.

In summary, our contributions are as follows:

- A loss accumulation guided densification strategy that inhibits unnecessary Gaussian generation in well-reconstructed regions.
- A pruning strategy that identifies and removes Gaussians that are unstable or have low rendering contribution.
- CUDA optimisations for the backward pass, including warp-level parallelism and vectorised access schemes, to reduce memory access overhead.

2 Related Work

2.1 Gaussian Densification Control

In 3DGS, adaptive density control dynamically adjusts the distribution of Gaussians according to the current reconstruction state, and it strongly affects both model size and reconstruction quality. The original 3DGS method performs densification based on image-plane gradients, where Gaussians with large gradients are split or cloned. However, since this strategy relies primarily on local gradient magnitude, it may lead to excessive densification in high-frequency textures, introducing redundant Gaussians and reducing training efficiency [10].

To address this issue, existing works improve densification from different perspectives. Some works reduce the need for densification by enhancing the representation ability of each Gaussian. For example, EAGLES [11] improves representation capacity by introducing lightweight feature encoding, while ADR-Gaussian [12] adaptively

adjusts the radius to better match Gaussian coverage with scene structure. Other works [13–15] modify the Gaussian basis function to further enhance representation capacity.

Another line of work directly constrains or regulates the densification process. For example, Taming-3DGS [16] introduces a Gaussian budget to control model size, while DashGaussian [17] uses a resolution scheduling strategy to progressively refine the scene during training. AbsGS [18] improves sensitivity to high frequency regions by using absolute gradients. In addition, methods such as [19–23] restrict Gaussian generation by introducing colour or other constraints.

Furthermore, some works attempt to directly use reconstruction errors to guide densification. For example, FastGS [24] constructs loss maps through multi-view forward rendering and refines Gaussians based on pixel-wise L1 loss. Compared with gradient-based signals, these methods can more directly locate under-reconstructed regions. However, they usually require multiple additional forward rendering passes to estimate error scores, which introduces computational overhead.

In summary, obtaining stable error guidance without additional overhead remains an open problem.

2.2 Gaussian Pruning

In addition to controlling the generation of Gaussians, another line of research focuses on reducing model complexity through pruning, thereby improving both training and rendering efficiency. Existing studies have shown that a large number of redundant Gaussians are often present in 3DGS models. For example, RadSplat [25] and LightGaussian [26] both highlight the significant redundancy in the original representation.

Early methods typically perform pruning based on heuristic saliency metrics. Works such as [27–30] design scoring functions based on opacity, scale, or contribution, and remove Gaussians with low importance.

To improve pruning accuracy, some works introduce evaluation mechanisms based on optimization sensitivity. For example, PUP 3DGS [31] performs an approximate analysis of parameter influence to identify Gaussians that contribute less to the overall loss across multiple views.

In addition, some methods approach compression from the perspective of structural constraints. For example, Mini-Splatting [32] achieves compact representations by limiting the number of Gaussians and incorporating resampling strategies, while DropGaussian [33] introduces structural regularisation through random dropping. MaskGaussian [34] further controls the participation of Gaussians in rendering in a probabilistic manner, balancing representation capability and compression ratio.

In summary, identifying low-contribution Gaussians with low computational overhead remains an area of interest for improving overall efficiency.

2.3 Rendering and System-Level Optimization

In addition to density control and pruning, computational redundancy in the rendering pipeline is another factor affecting the efficiency of 3DGS. The original rasterization

method adopts a relatively conservative strategy when estimating Gaussian projection ranges in image-plane. As a result, many redundant Gaussian–tile correspondences are introduced, leading to unnecessary computations during the rasterization stage [12].

To address this issue, one line of work reduces redundant computation by more accurately estimating the influence region of Gaussians on the image-plane. For example, Speedy-Splat [35] proposes the SnugBox and AccuTile methods to tighten Gaussian projection regions. FastGS [24] introduces a Compact Box strategy to reduce the number of Gaussian–tile pairs. StopThePop [36] and FlashGS [37] further improve rasterization efficiency by optimising tile intersection computation. These methods mainly improve forward rendering efficiency by reducing the number of pixels involved in the computation.

Another line of work focuses on optimisation strategies. For instance, 3DGS-LM [38] adopts a customised optimiser to accelerate convergence, while 3DGS2 [39] introduces a near second-order update strategy to improve optimisation efficiency. At the GPU execution level, DISTWAR [40] reduces thread contention by minimising atomic operations during backpropagation. In addition, methods such as [41–43] improve rendering stability and reduce computational complexity by introducing structural constraints or geometric priors.

Although several works have explored GPU execution optimisation during training, memory access patterns and thread organisation during the backward pass are still relatively unexplored.

3 Proposed Method

3.1 Preliminaries

3DGS represents a scene using a set of explicit anisotropic 3D Gaussians. The scene is composed of N Gaussian primitives, each defined as:

$$G(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \tag{1}$$

where $\mathbf{x}$ is the 3D coordinate, $\boldsymbol{\mu}$ is the centre and $\boldsymbol{\Sigma}$ is the covariance matrix of the Gaussian.

During forward rendering, these Gaussians are transformed into the camera coordinate system using the camera transformation. Their covariance matrices are also transformed into camera space through the Jacobian matrix. By marginalising along the viewing direction, a 2D covariance $\boldsymbol{\Sigma}_{2D}$ is obtained, which forms an elliptical Gaussian distribution on the image-plane:

$$G_{2D}(P) = \exp\left(-\frac{1}{2}(P-\boldsymbol{\mu}_{2D})^T\boldsymbol{\Sigma}_{2D}^{-1}(P-\boldsymbol{\mu}_{2D})\right) \tag{2}$$

where P is the pixel coordinate and $\boldsymbol{\mu_{2D}}$ is the coordinate of the centre on the image-plane.

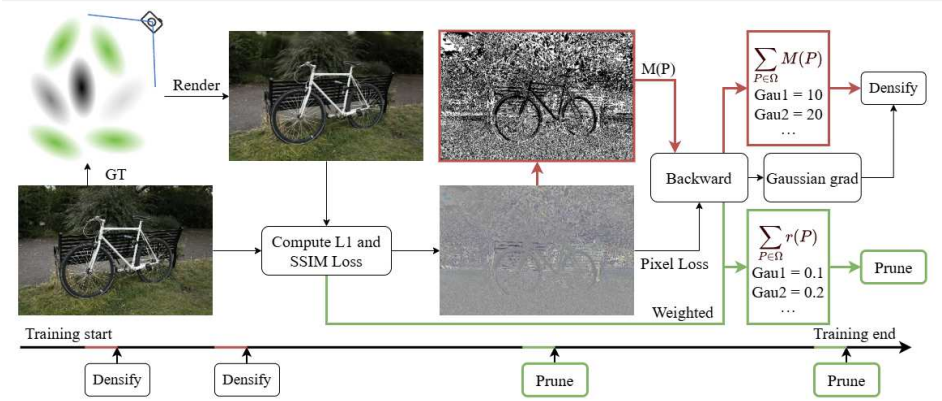


Figure 1: Overview. The timeline shows the training process of 30,000 iterations. The upper part shows the workflow of a single iteration. Red indicates the LAG strategy, and green indicates the pruning strategy.

Each Gaussian combines opacity σ with colour attributes c to represent pixel colour. After sorting the Gaussians by depth, the colour C of pixel P is accumulated using α-blending over n Gaussians:

$$C(P) = \sum_{i \in n} c_i \alpha_i(P) \prod_{j=1}^{i-1} (1 - \alpha_j(P)), \quad \alpha(P) = \sigma G_{2D}(P) \tag{3}$$

During the backward pass, each Gaussian influences multiple pixels within its projected region on the image-plane. Its gradient is computed by accumulating gradients from these pixels. As a result, each Gaussian receives optimisation signals from multiple local pixels.

3.2 Overview

As shown in Figure 1, our method is composed of three parts. At the algorithm level, we implement two strategies to maintain a compact Gaussian set: LAG densification inhibits unnecessary Gaussian generation in well-reconstructed regions, while opacity-gradient-based pruning identifies and removes unstable or low-contribution Gaussians. At the implementation level, we introduce CUDA optimisations that reduce redundant memory accesses and improve warp-level parallelism during the backward pass. Together, these components reduce training time while maintaining rendering quality.

3.3 Loss Accumulation Guided Densification

In the original 3DGS [2], the densification process is triggered by the average magnitude of 2D positional gradients. However, these gradients measure the sensitivity of the rendering loss to positional perturbations, which may not consistently correlate with the reconstruction quality [44]. Consequently, in regions containing high-frequency

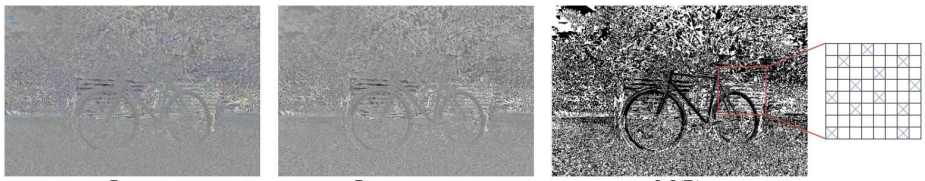


Figure 2: Pipeline for obtaining $M(P)$. After computing the three-channel average of the loss, normalisation and thresholding produce the binary mask $M(P)$.

textures or complex appearance variations, the gradients can become large [45]. This leads to the generation of many unnecessary Gaussians, increasing the computational load of the rasteriser.

The training efficiency of 3DGS is related to the number of Gaussians, as more Gaussians increase the per-iteration computation. Therefore, we propose the LAG densification strategy that uses an accumulated training loss constraint to suppress unnecessary Gaussian generation.

Our strategy establishes a connection between the optimisation objective and the density control mechanism. During the backward pass, we reuse the joint loss to identify pixels where the reconstruction error is already small. As shown in Figure 2, we average the three channels to obtain a single-channel loss map, then apply a thresholding operation. Pixels P with loss values below the threshold are considered well-reconstructed, while those above the threshold are marked as under-reconstructed. We define a binary mask $M(P)$:

$$M(P) = \begin{cases} 1 & L_m(P) > \tau \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

where $L_m(P)$ is the mean loss at pixel P. When a Gaussian predominantly covers pixels with $M(P) = 0$ (*i.e.*, low-loss, well-reconstructed pixels), generating additional Gaussians in that region is unnecessary because the optimisation objective for those pixels has already been met. Conversely, pixels with $M(P) = 1$ indicate under-reconstructed areas where densification may still be beneficial.

The loss distribution from a single rendering can be affected by viewpoint changes and scene noise. Therefore, using one iteration alone may not reliably reflect which regions are well-reconstructed. To address this, we accumulate $M(P)$ over K iterations. For each Gaussian, we project it onto the image-plane to obtain the covered pixel area Ω and compute the DS averaged over K iterations:

$$DS = \frac{1}{K} \sum_{t=1}^{K} \sum_{P \in \Omega_t} M_t(P) \tag{5}$$

where Ω_t is the set of pixels covered by the Gaussian in the t-th iteration, and $M_t(P)$ is $M(P)$ for that iteration. A Gaussian with a low DS consistently covers pixels where

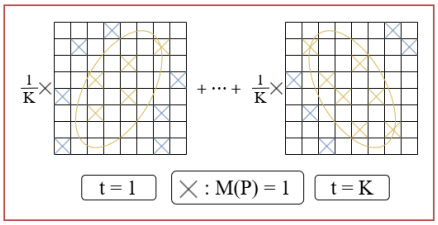


Figure 3: Calculation of DS. The grid represents the image-plane, where each cell corresponds to a pixel. The yellow ellipse indicates the projection of a Gaussian on the image-plane.

$M(P) = 0$, corresponding to low reconstruction error. In this case, densification is suppressed because the optimisation goal for those pixels has been achieved. An intuitive interpretation is: if a Gaussian mostly covers well-reconstructed pixels, splitting or cloning it will not help reduce the loss further, so the operation should be avoided.

As shown in Figure 3, by averaging the high loss pixel count over K iterations, DS provides a stable and reliable measure of whether a Gaussian is consistently located in a region where the current reconstruction is already sufficient.

During densification, we use the densification score as an additional constraint alongside the original 3DGS gradient criterion. A Gaussian is allowed to split or clone only when both constraints are satisfied. This dual-constraint mechanism inhibits Gaussian growth in well-reconstructed areas. By maintaining a smaller set of Gaussians, the per-iteration computation is reduced, which shortens training time.

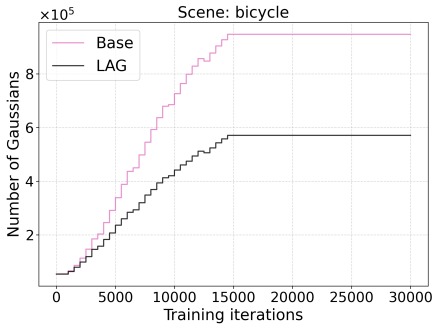
We compare the Base method with our proposed LAG strategy on the bicycle scene from the Mip-NeRF 360 [46] dataset. Figure 4 illustrates the number of Gaussians and average PSNR progression during training. Specifically, the left subplot illustrates how the number of Gaussians changes over training iterations. The right subplot shows the average PSNR computed per epoch.

Figure 5 presents qualitative comparisons between the two methods. The left image corresponds to the Base method, and the right image shows the result produced by our LAG strategy.

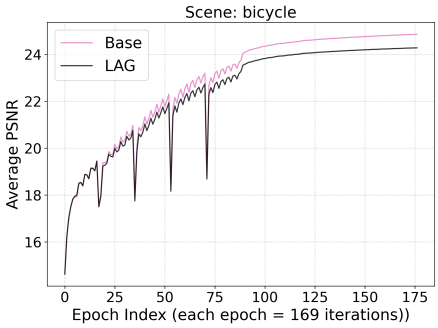
Quantitative results are summarised in Table 1. The LAG strategy reduces the peak number of Gaussians N_{GS} and shortens the training time. The reduction in Gaussians is accompanied by a small decrease in rendering metrics.

Table 1: Quantitative comparison between Base and LAG on the bicycle scene. Training time is reported in seconds.

Method	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}
Base	224.1	0.751	25.153	0.254	947138
LAG	190.0	0.734	24.939	0.279	570835



(a) Number of Gaussian comparison



(b) Average PSNR comparison

Figure 4: Comparison of Gaussian count and average PSNR during training on the bicycle scene. As the bicycle scene contains 169 training images, we define one epoch as a set of 169 iterations and calculate the average PSNR for each epoch by averaging the PSNR values across all iterations within that set.



(a) Base



(b) LAG

Figure 5: Rendered images on the bicycle scene.

3.4 Pruning Based on Opacity and Its Gradients

While the LAG densification strategy suppresses unnecessary Gaussian generation, a pruning mechanism is needed to remove existing Gaussians that have become redundant during training. We consider two types of redundant Gaussians, each addressed by a different criterion. The first type is *low-contribution* Gaussians: primitives whose opacity has become negligible and therefore contribute little to the rendered image. These are directly removed by an opacity threshold. The second type is *unstable* Gaussians: primitives that still exhibit large opacity gradients while having relatively low opacity, indicating that their parameters have not yet converged despite their limited contribution. These are identified through a gradient-based scoring mechanism.

To capture unstable Gaussians, we design a pruning metric based on the ratio of the opacity gradient to the opacity value at pixel level. During the backward pass, the

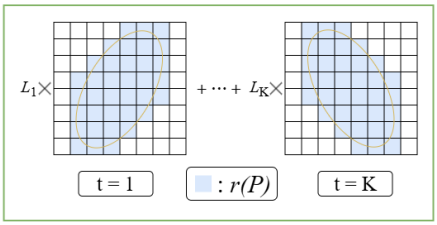


Figure 6: The calculation process of PS. The blue area represents pixels covered by Gaussian.

loss function produces gradients with respect to pixel-wise opacity. Because Gaussians have spatially non-uniform responses on the image-plane, different pixels contribute differently to the same Gaussian. We define a pixel-wise score as:

$$r(P) = \frac{1}{\alpha(P)} \left| \frac{\partial L(P)}{\partial \alpha(P)} \right| \tag{6}$$

where $L(P)$ is the loss at pixel P and $\alpha(P)$ is the opacity of the Gaussian at that pixel. This formulation specifically targets unstable Gaussians: a primitive with low opacity but a large gradient produces a high score, indicating that it remains sensitive to optimisation. In contrast, a well-converged Gaussian, which has a stable α and small gradient, receives a low score.

For a single view, we accumulate the pixel-wise scores within the region covered by the Gaussian to obtain a view-level score:

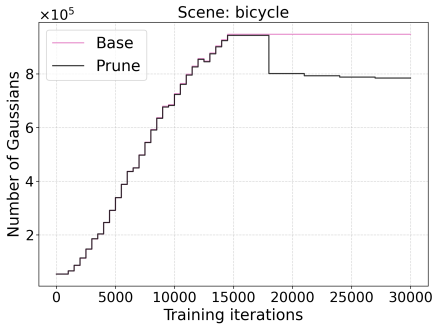
$$R = \sum_{P \in \Omega} r(P) \tag{7}$$

As shown in Figure 6, over K consecutive views, we weight each view-level score by the corresponding training loss and accumulate:

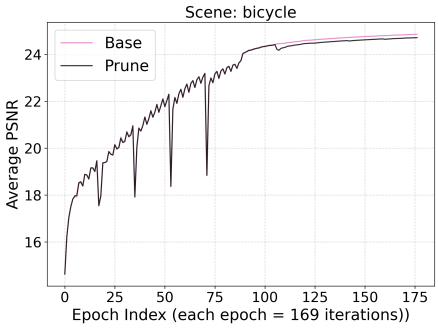
$$PS = \frac{1}{K} \sum_{t=1}^{K} L_t \cdot R_t \tag{8}$$

where L_t is the loss at the t-th iteration and R_t is the view-level score. Weighting by the training loss prevents Gaussians located in already well-reconstructed regions from being penalised for moderate instability.

In practice, pruning is performed periodically after the densification stage is completed. During pruning, we directly remove Gaussians with PS exceeding the threshold. Additionally, we directly discard Gaussians with opacity values below a predefined minimum. Through this strategy, the model can continuously remove redundant Gaussians during the training process, thereby reducing the workload of the rasteriser and the time taken for the backward pass.



(a) Number of Gaussian comparison



(b) Average PSNR comparison Prune

Figure 7: Comparison of Gaussian count and average PSNR during training on the bicycle scene.



(a) Base



(b) Prune

Figure 8: Rendered images on the bicycle scene.

We also compare the Base method with our proposed pruning strategy on the bicycle scene from the Mip-NeRF 360 [46] dataset. The comparison results are shown in Figure 7. Our pruning strategy can remove some Gaussians with almost no impact on rendering quality. Figure 8 presents qualitative comparisons and Table 2 presents quantitative results between the two methods. Our pruning strategy can effectively remove redundant Gaussians during training, with little impact on rendering quality.

Table 2: Quantitative comparison between Base and Prune on the bicycle scene. Prune refers to our pruning strategy.

Method	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}
Base	224.1	0.751	25.153	0.254	947138
Prune	211.7	0.751	25.142	0.257	784225

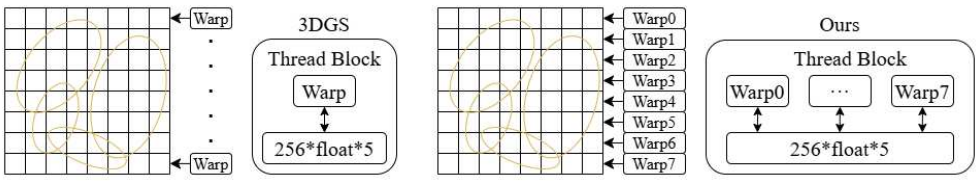


Figure 9: Comparison of thread organisation. In the original 3DGS, each thread block contains 32 threads, and each thread loads data from multiple pixels. In our method, each block uses 8 warps (256 threads), with each thread corresponding to a single pixel. Tile-level data are stored in shared memory for reuse across warps.

3.5 CUDA Parallel Optimisation for the Backward Pass

In addition to the density control strategies above, we modify the CUDA implementation of the 3DGS backward pass to reduce memory access overhead. These changes do not alter the rendering results or the optimisation behaviour; they focus on thread organisation and data access patterns.

Background. In CUDA, threads are grouped into *warps* (typically 32 threads on NVIDIA GPUs) that execute instructions in lockstep. Multiple warps form a *thread block*, whose threads can communicate through *shared memory* — a small on-chip memory with lower latency than global device memory. *Vectorised memory instructions* (*e.g.*, loading four floats with a single `float4` instruction) and *memory coalescing* (arranging memory accesses so that consecutive threads access consecutive addresses) improve bandwidth utilisation.

Motivation. In the original 3DGS, rendering is performed on fixed-size image tiles (typically 16×16 pixels). During the backward pass, multiple warps processing different Gaussians may repeatedly access pixel data within the same tile. Because the original implementation uses 32 threads per blocks, pixel data are fetched from global memory multiple times by different warps, creating redundant transactions.

Warp-level data reuse. We assign one thread block of 8 warps to each tile, creating a one-to-one mapping between threads and pixels inside the tile. As shown in Figure 9, each 16×16 tile contains 256 pixels. Tile-level data (pixel colours $\in \mathbb{R}^3$, transmittance $\in \mathbb{R}$, and the list of contributing Gaussians $\in \mathbb{R}$) are loaded once from global memory into shared memory and reused by all warps in the block. This reduces redundant global accesses because shared memory is an order of magnitude faster than global memory for intra-block reuse.

Vectorised memory access. As shown in Figure 10, we pack intermediate variables into `float4` format during the forward pass. For each pixel, the colour and transmittance are stored as a single `float4`. During the backward pass, one 128-bit transaction loads all four values, whereas four separate 32-bit loads would be needed otherwise. This reduces instruction count and improves memory bandwidth utilisation.

Memory coalescing via layout reorganisation. Both forward and backward passes involve indexed reads after depth sorting, which creates irregular access patterns. To address this, we reorganise the data layout during the forward pass by storing

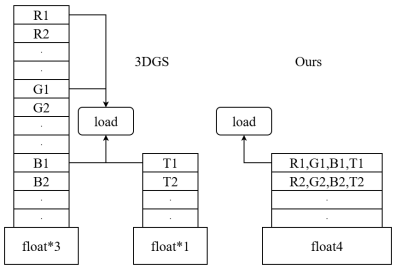


Figure 10: Modification of the data storage format. Intermediate variables are written in `float4` format during the forward pass. `float4` packs four 32-bit floats into a single 128-bit operand, enabling one-instruction loads. The "float*3" represents the three floating-point attributes of each Gaussian.

accessed data in contiguous memory according to the access order. This changes backward pass memory access from irregular to sequential, enabling coalesced reads that reduce access latency.

Portability. These optimisations make certain assumptions. (i) *Warp size:* The design assumes a warp size of 32, standard on NVIDIA GPUs. (ii) *Memory alignment:* `float4` access requires 16-byte alignment, which we ensure during buffer allocation. (iii) *GPU architecture:* The speedup from shared memory reuse depends on the available shared memory per block (48 KB or more on modern NVIDIA architectures). Older GPUs with less shared memory may require adjustments, yet our method can still run on the RTX 3060.

These optimisations are independent of the proposed density control strategy and can be integrated into existing 3DGS pipelines.

4 Experiments

4.1 Experimental setup

Datasets. We evaluate on the same datasets as the original 3DGS paper: Mip-NeRF 360 [46], Deep Blending [47], and Tanks & Temples [48].

Metrics. We evaluate the performance using PSNR, SSIM, LPIPS, and training time. To ensure a fair comparison of training time, only the iterative training time is measured, excluding data loading and saving to avoid the influence of different storage devices.

Implementation Details. We follow the original 3DGS strategy by using gradients for cloning, while adopting absolute gradients from AbsGS [18] for splitting. In addition, we incorporate the Compact Box strategy from FastGS [24]. In our default setting, densification is performed every 500 iterations until 15,000 iterations, after which pruning is conducted every 3,000 iterations. The DS threshold is 30 by default, K is 10, and the PS threshold is 0.09. All experiments are conducted on an NVIDIA RTX 3090 GPU, and all compared methods are based on their official implementations. For fairness, all methods use the fused-ssim to calculate SSIM loss during training.

Table 3: Quantitative results. N_{GS} denotes the final number of Gaussians in millions. Training time is reported in seconds.

	Mip-NeRF 360 [46]					Deep Blending [47]					Tanks & Temples [48]				
	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}
3DGS [2]	1002.4	0.816	27.522	0.215	2.74M	829.5	0.907	29.826	0.238	2.48M	655.0	0.853	23.752	0.169	1.57M
Taming-3DGS [16]	913.9	0.819	27.847	0.214	2.33M	639.1	0.908	29.866	0.242	1.57M	465.7	0.853	24.252	0.181	1.02M
Speedy-Splat [35]	790.9	0.786	26.931	0.287	0.32M	728.9	0.903	29.642	0.268	0.25M	463.1	0.820	23.410	0.240	0.18M
DashGaussian [17]	571.7	0.819	27.713	0.215	2.25M	386.8	0.906	29.766	0.245	1.95M	413.6	0.852	24.005	0.175	1.35M
FastGS [24]	210.8	0.798	27.510	0.261	0.38M	159.4	0.905	30.025	0.267	0.22M	154.4	0.842	24.137	0.209	0.24M
Ours	193.6	0.806	27.617	0.247	0.41M	145.2	0.906	30.102	0.263	0.24M	129.8	0.845	24.154	0.202	0.26M

Table 4: Comparisons on memory. Mem allocated and Mem reserved are obtained using PyTorch's `torch.cuda.max_memory_allocated()` and `torch.cuda.max_memory_reserved()` after training, reported in MB.

	Mip-NeRF 360 [46]			Deep Blending [47]			Tanks & Temples [48]		
	Mem allocated↓	Mem reserved↓	Storage	Mem allocated↓	Mem reserved↓	Storage	Mem allocated↓	Mem reserved↓	Storage
3DGS [2]	9111	15223	644.9	8632	13767	586.0	4972	8223	370.7
Taming [16]	7549	14630	552.0	6261	9570	371.1	3572	6858	240.9
Speedy [35]	8580	9838	75.0	7007	8024	59.2	2162	4029	42.9
DashGS [17]	8650	14757	531.5	7968	12910	461.2	4824	8878	318.5
FastGS [24]	4385	6392	89.2	3885	5236	51.2	2337	3627	57.3
Ours	4450	6784	98.0	3952	5755	57.3	2373	3884	62.1

4.2 Comparison with Existing Methods

We compare the proposed method with several Gaussian Splatting methods, including 3DGS, Taming-3DGS [16], Speedy-Splat [35], DashGaussian [17], and FastGS [24]. As shown in Table 3, the proposed method reduces training time across all three datasets while maintaining reconstruction quality.

Compared with FastGS [24], the proposed method achieves shorter training time even with a comparable number of Gaussians. This suggests that the CUDA backward pass optimisations contribute to the reduction in training time.

The proposed method maintains comparable PSNR and SSIM values. Figure 11 shows a qualitative comparison across methods. In particular, the counter scene [46] in the fourth column includes specular highlights and reflective surfaces, allowing a visually more informative comparison.

4.3 More Comparison Results

In Table 4, we supplement the comparison of peak GPU memory usage and file storage size between our method and other methods. In PyTorch, `torch.cuda.max_memory_allocated()` and `torch.cuda.max_memory_reserved()` represent the peak of actually allocated GPU memory and the peak of GPU memory requested from the CUDA driver, respectively. Storage indicates the storage size of the ply file generated after the training is completed, reported in MB. Mem allocated and Mem reserved are related to the peak number of Gaussians. Storage is proportional to the final number of Gaussians N_{GS}.

In Table 5, we supplement the comparison of our method and other methods in terms of rendering frame rate (FPS), and the time for forward and backward propagation. Since each method generates a different number of Gaussians, in order to



Figure 11: Qualitative comparison across methods. Each row corresponds to one method, and each column corresponds to a scene.

make a fair comparison, the PLY files generated by the original 3DGS [2] are used for comparison. The comparison procedure is as follows. After loading the PLY file, we perform 100 rounds of test-set rendering. In each iteration, the render function produces an image and the back function performs backpropagation. The total running time of the render function is recorded as the forward time, and the total running time of the back function is recorded as the backward time. FPS is calculated based on the forward time and the total number of iterations.

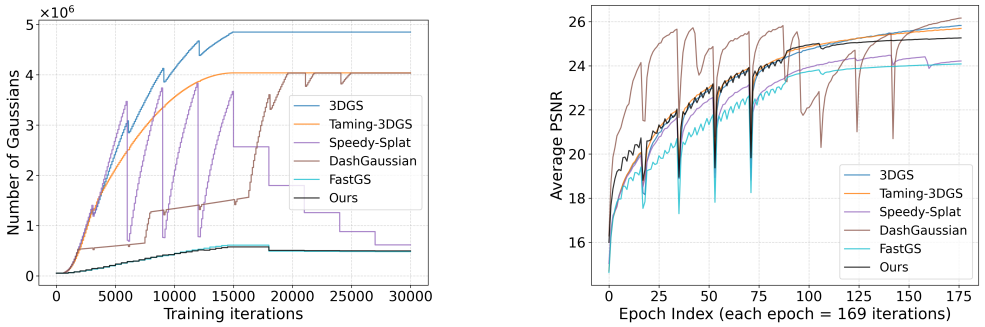
Under the same PLY file, the proposed method requires less time, which indicates that the CUDA optimisations reduce execution time for the backward pass.

Figure 12 shows the number of Gaussians and average PSNR over iterations for different methods on the bicycle scene of Mip-NeRF 360 [46]. Note that DashGS [17] uses a scheduler to adjust rendering resolution during training, so the PSNR curves in Figure 12b are not computed at full resolution. In Figure 12a, the proposed method increases the number of Gaussians more slowly during training, while the PSNR shown in Figure 12b remains comparable to 3DGS.

Table 6 compares the proposed method with other density-control and acceleration methods. The proposed method reuses the loss from the training process to guide

Table 5: Comparison of forward and backward propagation times for different datasets, reported in seconds. FPS is calculated based on the forward propagation time. The PLY file comes from bicycle scene of Mip-NeRF 360 [46]

	Mip-NeRF 360 [46]			Deep Blending [47]			Tanks & Temples [48]		
	FPS↑	Forward time↓	Backward time ↓	FPS↑	Forward time↓	Backward time ↓	FPS↑	Forward time↓	Backward time ↓
3DGS [2]	167	16.9	42.6	187	17.3	37.9	215	16.3	37.4
Taming [16]	173	16.3	36.7	193	16.7	33.8	219	16.0	28.6
Speedy [35]	201	14.6	74.9	221	14.9	90.0	241	14.6	53.6
DashGS [17]	167	16.9	42.5	185	17.4	37.9	213	16.4	37.4
FastGS [24]	307	9.4	33.6	350	9.2	32.0	313	11.2	27.8
Ours	322	9.0	30.2	377	8.6	29.3	339	10.3	24.6



(a) Number of Gaussians

(b) Average PSNR

Figure 12: Comparison of the number of Gaussians and average PSNR for different methods.

Table 6: Comparison of the proposed method with other density-control methods. In the CUDA-level optimisation column, fw indicates forward propagation has been optimised, and bw indicates backward propagation has been optimised.

	signal used for densification	extra rendering	pruning	CUDA-level optimisation
DashGS [17]	grad + densify rate	No	No	No
Abs-GS [18]	grad + abs-grad	No	No	No
Taming [16]	grad + scoring function	Yes	No	Yes(bw)
FastGS [24]	grad + L1	Yes	Yes	Yes(fw)
Ours	grad + L1 + SSIM	No	Yes	Yes(fw&bw)

densification without requiring additional rendering passes. The second aspect is the CUDA backward pass optimisations.

4.4 Ablation study

The ablation results in Table 7 illustrate the contribution of each component and their interactions. We discuss four observations.

Table 7: Ablation study on three datasets. Base denotes the baseline method combining the Compact Box strategy [24] and Abs-GS [18] splitting. Prune refers to our pruning strategy; cu-opt refers to CUDA optimisations. Full represents our complete method. For all configurations, the densification interval is 500.

	Mip-NeRF 360 [46]					Deep Blending [47]					Tanks & Temples [48]				
Ablation	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}
Base	245.7	0.813	27.722	0.232	0.77M	201.8	0.909	30.094	0.249	0.73M	175.8	0.854	24.157	0.184	0.49M
Base+cu-opt	226.7	0.813	27.740	0.232	0.77M	186.3	0.910	30.091	0.250	0.73M	155.4	0.854	24.150	0.185	0.49M
LAG	215.4	0.806	27.655	0.245	0.47M	160.5	0.906	30.099	0.262	0.30M	151.7	0.846	24.006	0.198	0.32M
LAG+cu-opt	200.1	0.806	27.624	0.245	0.47M	150.0	0.907	30.104	0.261	0.30M	135.5	0.846	24.000	0.198	0.31M
Prune	233.7	0.813	27.716	0.234	0.63M	187.6	0.909	30.112	0.252	0.53M	164.6	0.852	24.243	0.188	0.39M
Prune+cu-opt	216.9	0.813	27.725	0.234	0.63M	176.4	0.909	30.091	0.251	0.53M	148.2	0.852	24.257	0.188	0.39M
LAG+Prune	203.4	0.806	27.627	0.247	0.41M	150.0	0.905	30.103	0.263	0.24M	141.9	0.846	24.155	0.201	0.26M
full	193.6	0.806	27.617	0.247	0.41M	145.2	0.906	30.102	0.263	0.24M	129.8	0.845	24.154	0.202	0.26M

Effect of LAG. Comparing Base with LAG shows that the LAG strategy reduces the N_{GS}. For example, on Mip-NeRF 360 [46], N_{GS} decreases from 0.77M to 0.47M. This reduction leads to shorter training time because fewer Gaussians are processed per iteration. The rendering metrics decrease slightly, which reflects the trade-off between compactness and reconstruction accuracy.

Effect of pruning. Comparing Base with Prune shows that the pruning strategy removes additional Gaussians after densification, further reducing N_{GS} and training time while having a negligible effect on rendering metrics. The pruning mechanism targets redundant Gaussians: low-contribution primitives are removed by the opacity threshold, and unstable ones by the PS.

Relationship between LAG and pruning. When both are used together (LAG+Prune), the reduction in N_{GS} is larger than with either component alone. On Mip-NeRF 360 [46], LAG+Prune achieves 0.41M Gaussians, compared to 0.47M (LAG) and 0.63M (Prune), indicating that the two strategies are complementary.

Effect of CUDA optimisations. Comparing configurations with and without CUDA optimisations (*e.g.*, Base vs. Base+cu-opt and LAG vs. LAG+cu-opt) shows a consistent reduction in training time across all datasets whilst rendering metrics remain nearly unchanged. The speedup from CUDA optimisations is independent of the number of Gaussians: even when the Gaussian count has already been reduced by LAG or pruning (LAG+cu-opt vs. LAG), the CUDA optimisations still provide additional time savings. For instance, on Mip-NeRF 360 [46], LAG requires 215.4s and LAG+cu-opt requires 200.1s, a reduction of approximately 7%. This is because the CUDA optimisations target memory access patterns in the backward pass kernel, which are present regardless of how many Gaussians are being processed. Therefore, the CUDA optimisations remain beneficial even in already-compact configurations.

Summary. The three components target different aspects of training efficiency: LAG controls Gaussian generation, Prune removes unhelpful primitives post-generation, and CUDA optimisations reduce per-iteration execution overhead. Together, they achieve shorter training time while maintaining reconstruction quality.

4.5 Ablation of component effectiveness

Table 8 shows the effect of each component applied individually on top of a 3DGS* [2] baseline. The Compact Box component [24] reduces training time with little change in

Table 8: 3DGS* [2] denotes a baseline with adjusted colour learning rate and densification interval. "Only LAG", "Only Prune", and "Only cu-opt" indicate that each component is enabled individually on the 3DGS* [2] baseline. The densification interval is 500 for all configurations, and the colour learning rate follows FastGS [24].

	Mip-NeRF 360 [46]					Deep Blending [47]					Tanks & Temples [48]				
Component	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}
3DGS*	340.5	0.802	27.559	0.250	0.99M	256.5	0.907	30.018	0.252	0.74M	220.9	0.850	24.301	0.189	0.65M
Compact Box [24]	262.8	0.803	27.543	0.247	1.07M	204.9	0.908	30.019	0.251	0.78M	188.2	0.850	24.226	0.189	0.67M
Abs Split [18]	340.9	0.814	27.818	0.233	0.74M	262.0	0.908	30.008	0.251	0.71M	215.6	0.854	24.236	0.185	0.48M
Only LAG	286.5	0.791	27.407	0.269	0.44M	208.2	0.905	30.044	0.266	0.33M	180.4	0.841	24.116	0.207	0.29M
Only Prune	317.0	0.800	27.457	0.253	0.78M	234.1	0.908	29.926	0.254	0.54M	205.2	0.847	23.994	0.195	0.48M
Only cu-opt	308.9	0.802	27.587	0.250	1.00M	235.4	0.909	29.928	0.253	0.74M	199.2	0.850	24.353	0.190	0.65M

Table 9: DSTH represents the threshold of *DS*. Sub 20 means subtracting 20 from the default threshold.

	Mip-NeRF 360 [46]					Deep Blending [47]					Tanks & Temples [48]				
DSTH	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}
sub 20	230.8	0.812	27.732	0.236	0.58M	169.0	0.906	30.035	0.258	0.37M	166.6	0.851	24.112	0.189	0.41M
sub 10	223.4	0.809	27.690	0.240	0.52M	165.0	0.906	30.069	0.260	0.33M	158.5	0.848	24.017	0.194	0.35M
add 0	215.4	0.806	27.655	0.245	0.47M	160.5	0.906	30.099	0.262	0.30M	151.7	0.846	24.006	0.198	0.32M
add 10	211.3	0.803	27.592	0.250	0.44M	159.9	0.905	30.031	0.264	0.28M	148.1	0.844	23.945	0.202	0.29M
add 20	207.1	0.800	27.508	0.255	0.41M	157.3	0.904	29.867	0.266	0.26M	145.4	0.841	23.757	0.206	0.26M

the number of Gaussians, as it eliminates unnecessary Gaussian-tile matches during forward propagation. The Abs Split component [18] improves rendering quality while using fewer Gaussians. The LAG strategy reduces the number of Gaussians but also leads to a small decrease in rendering metrics. The pruning strategy reduces the final number of Gaussians. The CUDA optimisations reduce training time without notably affecting rendering quality.

4.6 Ablation of hyperparameters

We tested the impact of different DS thresholds on training results, and the results are shown in Table 9. The default threshold for DS is 30, set to 60 for the playroom scene in the DeepBlending dataset, and 30 for other scenes. Since the thresholds vary across different scenes, all different thresholds are represented by the difference compared to the default threshold. As DSTH increases, the number of Gaussians N_{GS} gradually decreases. This is because a larger threshold imposes stricter constraints on the densification of Gaussians. The decrease in rendering quality and training time is influenced by N_{GS}.

We tested the impact of different pruning frequencies on the pruning strategy, and the results are shown in Table 10. In most cases, pruning frequency has little effect on pruning. As shown in Figure 7a, a larger number of Gaussians was removed only during the first pruning, while subsequent prunings removed only a small number of Gaussians. It is worth noting that when the interval equals 1000, this means the first pruning occurs at 16,000 iterations, and there is a drop in rendering quality on the Tanks & Temples [48] dataset, indicating that our pruning strategy removed Gaussians that were still useful. When the frequency equals 5000, the number of Gaussians is the highest, indicating that after 5000 iterations, some Gaussians have stabilised or become highly contributory, so our pruning strategy removes fewer Gaussians.

Table 10: Effect of different pruning frequencies on pruning. "Freq 1000" indicates pruning is performed every 1,000 iterations.

	Mip-NeRF 360 [46]					Deep Blending [47]					Tanks & Temples [48]				
Freq	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}
1000	232.9	0.813	27.733	0.235	0.62M	183.9	0.908	29.879	0.252	0.50M	162.0	0.841	23.299	0.202	0.38M
2000	233.9	0.813	27.727	0.235	0.62M	186.6	0.909	30.088	0.253	0.51M	164.1	0.850	24.098	0.191	0.38M
3000	233.7	0.813	27.716	0.234	0.63M	187.6	0.909	30.112	0.252	0.53M	164.6	0.852	24.243	0.188	0.39M
4000	233.1	0.813	27.716	0.234	0.62M	185.4	0.909	30.079	0.251	0.51M	163.4	0.851	24.176	0.189	0.39M
5000	238.9	0.813	27.691	0.234	0.64M	193.2	0.910	30.115	0.251	0.55M	169.4	0.852	24.141	0.187	0.41M

Table 11: The impact of different PS thresholds on the pruning strategy. PSTH represents the threshold of PS.

	Mip-NeRF 360 [46]					Deep Blending [47]					Tanks & Temples [48]				
PSTH	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}
0.07	235.6	0.813	27.721	0.234	0.63M	189.4	0.909	30.133	0.251	0.53M	165.5	0.851	24.078	0.189	0.39M
0.08	235.7	0.813	27.707	0.234	0.63M	189.7	0.909	29.992	0.251	0.53M	165.8	0.852	24.144	0.189	0.39M
0.09	233.7	0.813	27.716	0.234	0.63M	187.6	0.909	30.112	0.252	0.53M	164.6	0.852	24.243	0.188	0.39M
0.10	236.0	0.813	27.732	0.234	0.63M	189.8	0.909	30.130	0.251	0.53M	165.7	0.851	24.119	0.189	0.39M
0.11	236.1	0.813	27.718	0.234	0.63M	190.0	0.908	30.128	0.251	0.53M	165.3	0.852	24.175	0.188	0.39M

Table 12: The effect of different K on the full method

	Mip-NeRF 360 [46]					Deep Blending [47]					Tanks & Temples [48]				
K	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}	Time↓	SSIM↑	PSNR↑	LPIPS↓	N_{GS}
5	192.6	0.805	27.572	0.249	0.39M	141.5	0.904	30.026	0.266	0.22M	127.5	0.845	24.152	0.203	0.26M
10	193.6	0.806	27.617	0.247	0.41M	145.2	0.906	30.102	0.263	0.24M	129.8	0.845	24.154	0.202	0.26M
15	197.1	0.806	27.636	0.247	0.42M	152.6	0.905	29.946	0.264	0.25M	131.1	0.846	24.208	0.201	0.26M
20	204.2	0.805	27.602	0.247	0.42M	152.8	0.905	29.975	0.263	0.25M	131.0	0.845	24.182	0.202	0.26M
25	198.5	0.805	27.604	0.247	0.42M	152.0	0.906	29.949	0.263	0.25M	131.6	0.845	24.098	0.202	0.26M

We tested the impact of different PS thresholds on the pruning strategy, and the results are shown in Table 11. In most cases, pruning is not sensitive to the threshold, because the PS values of unstable Gaussians typically differ substantially from those of stable, well-converged Gaussians.

We tested different K on the full method, and the results are shown in Table 12. Different values of K yield different patterns across datasets. This is related to the specific scenario, because the training sets of different scenarios are usually of different sizes. With the same K renderings, the coverage of Gaussians varies, which affects both densification and pruning.

5 Conclusion

This paper proposes an optimisation framework to reduce the training time of 3DGS. At the algorithm level, we introduce a Loss Accumulation Guided densification strategy and an opacity-gradient-based pruning strategy. The densification strategy uses accumulated training loss to suppress unnecessary Gaussian generation in well-reconstructed regions. The pruning strategy removes redundant Gaussians through two criteria: an opacity threshold for low-contribution primitives and a gradient-based score for unstable ones. These two strategies work together to maintain a compact

Gaussian set. At the implementation level, we introduce CUDA optimisations, including warp-level parallelism and vectorised data access, to reduce memory access overhead during the backward pass.

Experimental results across multiple datasets show that the proposed method reduces total training time while maintaining rendering quality.

6 Limitations

Limitations of the LAG Strategy. First, the LAG strategy uses a uniform threshold τ for all pixels to obtain $M(P)$. Pixels in a black background and pixels in texture-rich foreground areas are treated equally when their loss values are identical. Second, $M(P)$ is recorded over K consecutive iterations with randomly sampled training images. This stochastic selection may occasionally include non-critical views, which can introduce noise into the densification score.

Limitations of the Pruning Strategy. First, PS accumulated over K iterations cannot cover all Gaussians in the scene. Pruning is restricted to the subset of Gaussians that project onto the image-plane during these views, leaving unseen Gaussians unpruned. Second, pixel-level pruning scores may penalise Gaussians that are still adapting in difficult regions. As shown in Table 10, pruning that occurs too early can reduce rendering quality.

Limitations of the CUDA Optimisation. Our CUDA optimisations modify the execution patterns of the original backpropagation kernel without altering the mathematical formulation of 3DGS. Consequently, the improvement in training speed is limited to the backward pass overhead.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Funding No funding was received for conducting this study.

Author Contributions Methodology: [Xiaofeng Che]; Formal analysis and investigation: [Xiaofeng Che, Chen Wang]; Writing - original draft preparation: [Xiaofeng Che]; Writing - review and editing: [Chen Wang, Guangle Yao]; Resources: [Haoran Zhou, Shaoze Ye]; Supervision: [Xulong Liu].

References

[1] Mildenhall, B., Srinivasan, P.P., Tancik, M., Barron, J.T., Ramamoorthi, R., Ng, R.: Nerf: Representing scenes as neural radiance fields for view synthesis. Communications of the ACM **65**(1), 99–106 (2021)

[2] Kerbl, B., Kopanas, G., Leimkühler, T., Drettakis, G., *et al.*: 3d gaussian splatting for real-time radiance field rendering. ACM Trans. Graph. **42**(4), 139–1 (2023)

[3] Liu, M., Liu, J., Zhang, Y., Li, J., Yang, M.Y., Nex, F., Cheng, H.: 4dstr: Advancing generative 4d gaussians with spatial-temporal rectification for high-quality and consistent 4d generation. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 40, pp. 7224–7232 (2026)

[4] Wei, X., Ye, Z., Gu, Y., Zhu, Z., Guo, Y., Shen, Y., Zhao, S., Lu, M., Sun, H., Wang, B., et al.: Parkgaussian: Surround-view 3d gaussian splatting for autonomous parking. arXiv preprint arXiv:2601.01386 (2026)

[5] Yin, H., Yang, Z., Zhang, L., Rigoll, G., Betz, J.: Roger-slam: A robust gaussian splatting slam system for noisy and low-light environment resilience. IEEE Transactions on Instrumentation and Measurement (2026)

[6] Hahlbohm, F., Franke, L., Eisemann, M., Magnor, M.: Faster-gs: Analyzing and improving gaussian splatting optimization. arXiv preprint arXiv:2602.09999 (2026)

[7] Zhou, Z., Xiong, Y.-J., Zhang, J.-C., Xia, C.-M., Qiu, X., Zhan, H.: Gradient-direction-aware density control for 3d gaussian splatting. arXiv preprint arXiv:2508.09239 (2025)

[8] Deng, X., Diao, C., Li, M., Yu, R., Xu, D.: Improving densification in 3d gaussian splatting for high-fidelity rendering. arXiv preprint arXiv:2508.12313 (2025)

[9] Zhou, H., Ni, Z.: Perceptual-gs: Scene-adaptive perceptual densification for gaussian splatting. arXiv preprint arXiv:2506.12400 (2025)

[10] Zeng, Z., Wang, Y., Ju, L., Guan, T.: Frequency-aware density control via reparameterization for high-quality rendering of 3d gaussian splatting. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 39, pp. 9833–9841 (2025)

[11] Girish, S., Gupta, K., Shrivastava, A.: Eagles: Efficient accelerated 3d gaussians with lightweight encodings. In: European Conference on Computer Vision, pp. 54–71 (2024). Springer

[12] Wang, X., Yi, R., Ma, L.: Adr-gaussian: Accelerating gaussian splatting with adaptive radius. In: SIGGRAPH Asia 2024 Conference Papers, pp. 1–10 (2024)

[13] Chen, L., Hu, Y., Zhang, Y., Ge, Y., Zhang, H., Cai, X.: Frequency-importance gaussian splatting for real-time lightweight radiance field rendering. Multimedia Tools and Applications **83**(35), 83377–83401 (2024)

[14] Liu, R., Sun, D., Chen, M., Wang, Y., Feng, A.: Deformable beta splatting. In: Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers, pp. 1–11 (2025)

[15] Li, H., Liu, J., Sznaier, M., Camps, O.: 3d-hgs: 3d half-gaussian splatting. In:

Proceedings of the Computer Vision and Pattern Recognition Conference, pp. 10996–11005 (2025)

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

[17] Chen, Y., Jiang, J., Jiang, K., Tang, X., Li, Z., Liu, X., Nie, Y.: Dashgaussian: Optimizing 3d gaussian splatting in 200 seconds. In: Proceedings of the Computer Vision and Pattern Recognition Conference, pp. 11146–11155 (2025)

[18] Ye, Z., Li, W., Liu, S., Qiao, P., Dou, Y.: Absgs: Recovering fine details in 3d gaussian splatting. In: Proceedings of the 32nd ACM International Conference on Multimedia, pp. 1053–1061 (2024)

[19] Deng, X., Diao, C., Li, M., Yu, R., Xu, D.: Efficient density control for 3d gaussian splatting. arXiv preprint arXiv:2411.10133 (2024)

[20] Kim, S., Lee, K., Lee, Y.: Color-cued efficient densification method for 3d gaussian splatting. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 775–783 (2024)

[21] Wang, X., Shan, L.: Gdgs: 3d gaussian splatting via geometry-guided initialization and dynamic density control. arXiv preprint arXiv:2507.00363 (2025)

[22] Zhang, Z., Hu, W., Lao, Y., He, T., Zhao, H.: Pixel-gs: Density control with pixel-aware gradient for 3d gaussian splatting. In: European Conference on Computer Vision, pp. 326–342 (2024). Springer

[23] Dai, X., Yao, L.: Dtgs: Defocus-tolerant view synthesis using gaussian splatting. Computer Animation and Virtual Worlds **36**(3), 70045 (2025)

[24] Ren, S., Wen, T., Fang, Y., Lu, B.: Fastgs: Training 3d gaussian splatting in 100 seconds. arXiv preprint arXiv:2511.04283 (2025)

[25] Niemeyer, M., Manhardt, F., Rakotosaona, M.-J., Oechsle, M., Duckworth, D., Gosula, R., Tateno, K., Bates, J., Kaeser, D., Tombari, F.: Radsplat: Radiance field-informed gaussian splatting for robust real-time rendering with 900+ fps. In: 2025 International Conference on 3D Vision (3DV), pp. 134–144 (2025). IEEE

[26] Fan, Z., Wang, K., Wen, K., Zhu, Z., Xu, D., Wang, Z.: Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps. Advances in neural information processing systems **37**, 140138–140158 (2024)

[27] Ali, M.S., Bae, S.-H., Tartaglione, E.: Elmgs: Enhancing memory and computation scalability through compression for 3d gaussian splatting. In: 2025 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV),

pp. 2591–2600 (2025). IEEE

[28] Ali, M.S., Qamar, M., Bae, S.-H., Tartaglione, E.: Trimming the fat: Efficient compression of 3d gaussian splats through pruning. arXiv preprint arXiv:2406.18214 (2024)

[29] Wu, R., Jiao, L., Liu, G., Zhu, L., Fei, X., Mu, Y., Fan, C.: Lfgs: A lightweight framework for efficient 3d gaussian splatting with minimal memory footprint. Computers & Graphics, 104309 (2025)

[30] Papantonakis, P., Kopanas, G., Kerbl, B., Lanvin, A., Drettakis, G.: Reducing the memory footprint of 3d gaussian splatting. Proceedings of the ACM on Computer Graphics and Interactive Techniques **7**(1), 1–17 (2024)

[31] Hanson, A., Tu, A., Singla, V., Jayawardhana, M., Zwicker, M., Goldstein, T.: Pup 3d-gs: Principled uncertainty pruning for 3d gaussian splatting. In: Proceedings of the Computer Vision and Pattern Recognition Conference, pp. 5949–5958 (2025)

[32] Fang, G., Wang, B.: Mini-splatting: Representing scenes with a constrained number of gaussians. In: European Conference on Computer Vision, pp. 165–181 (2024). Springer

[33] Park, H., Ryu, G., Kim, W.: Dropgaussian: Structural regularization for sparse-view gaussian splatting. In: Proceedings of the Computer Vision and Pattern Recognition Conference, pp. 21600–21609 (2025)

[34] Liu, Y., Zhong, Z., Zhan, Y., Xu, S., Sun, X.: Maskgaussian: Adaptive 3d gaussian representation from probabilistic masks. In: Proceedings of the Computer Vision and Pattern Recognition Conference, pp. 681–690 (2025)

[35] Hanson, A., Tu, A., Lin, G., Singla, V., Zwicker, M., Goldstein, T.: Speedy-splat: Fast 3d gaussian splatting with sparse pixels and sparse primitives. In: Proceedings of the Computer Vision and Pattern Recognition Conference, pp. 21537–21546 (2025)

[36] Radl, L., Steiner, M., Parger, M., Weinrauch, A., Kerbl, B., Steinberger, M.: Stopthepop: Sorted gaussian splatting for view-consistent real-time rendering. ACM Transactions on Graphics (TOG) **43**(4), 1–17 (2024)

[37] Feng, G., Chen, S., Fu, R., Liao, Z., Wang, Y., Liu, T., Hu, B., Xu, L., Pei, Z., Li, H., *et al.*: Flashgs: Efficient 3d gaussian splatting for large-scale and high-resolution rendering. In: Proceedings of the Computer Vision and Pattern Recognition Conference, pp. 26652–26662 (2025)

[38] Höllein, L., Božič, A., Zollhöfer, M., Nießner, M.: 3dgs-lm: Faster gaussian-splatting optimization with levenberg-marquardt. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 26740–26750

(2025)

[39] Lan, L., Shao, T., Lu, Z., Zhang, Y., Jiang, C., Yang, Y.: 3dgs2: Near second-order converging 3d gaussian splatting. In: Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers, pp. 1–10 (2025)

[40] Durvasula, S., Zhao, A., Chen, F., Liang, R., Sanjaya, P.K., Vijaykumar, N.: Distwar: Fast differentiable rendering on raster-based rendering pipelines. arXiv preprint arXiv:2401.05345 (2023)

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

[42] Ververas, E., Potamias, R.A., Song, J., Deng, J., Zafeiriou, S.: Sags: Structure-aware 3d gaussian splatting. In: European Conference on Computer Vision, pp. 221–238 (2024). Springer

[43] Wei, M., Wu, Q., Zheng, J., Rezatofighi, H., Cai, J.: Normal-gs: 3d gaussian splatting with normal-involved rendering. arXiv preprint arXiv:2410.20593 (2024)

[44] Zhang, J., Zhan, F., Xu, M., Lu, S., Xing, E.: Fregs: 3d gaussian splatting with progressive frequency regularization. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 21424–21433 (2024)

[45] Xu, R., Fang, L., Liu, C., Zhang, Y., Wei, Q.: Global-aware voxelized adaptive density control for efficient and compact 3d gaussian splatting. Neurocomputing, 133400 (2026)

[46] Barron, J.T., Mildenhall, B., Verbin, D., Srinivasan, P.P., Hedman, P.: Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 5470–5479 (2022)

[47] Hedman, P., Philip, J., Price, T., Frahm, J.-M., Drettakis, G., Brostow, G.: Deep blending for free-viewpoint image-based rendering. ACM Transactions on Graphics (ToG) **37**(6), 1–15 (2018)

[48] Knapitsch, A., Park, J., Zhou, Q.-Y., Koltun, V.: Tanks and temples: Benchmarking large-scale scene reconstruction. ACM Transactions on Graphics (ToG) **36**(4), 1–13 (2017)



Xiaofeng Che received the B.E. degree in process equipment and control engineering from Xinjiang University in 2022. He is currently pursuing the M.S. degree with the Chengdu University of Technology. His research interests include computer vision and 3D reconstruction.



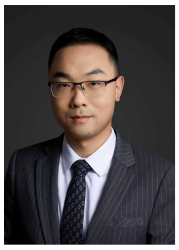
Chen Wang received the Ph.D. degree in signal and information processing from the University of Electronic Science and Technology of China. He is currently a Teacher with the Chengdu University of Technology. His research interests include computer vision and 3D reconstruction.



Haoran Zhou received the B.S. degree from the University of Seattle, USA. He is the founder of Shenzhen Sensing Data Technology Company Limited, specializing in the research and development of 3D reconstruction, 3DAIGC, pattern recognition, tree modeling and other fields. The individual has applied for and obtained more than fifteen invention patents.



Guangle Yao received the Ph.D. degree in Engineering from the University of Electronic Science and Technology of China. He is currently an Associate Professor with the Chengdu University of Technology. His research interests include computer vision and 3D scene understanding.



Xulong Liu received both the B.S. and M.S. degrees in electrical engineering, information technology and computer engineering from RWTH Aachen University, Germany. He is currently associated with Jingtian & Gongcheng Law Firm, Beijing, specializing in network & data law and AI law. His research interests include data governance and architecture design for intelligent terminals and Web 3.0.



Shaoze Ye received the M.S. degree from Chengdu University of Technology. His research encompasses the integrated application of artificial intelligence, 3D vision technology, and business scenarios. As the first inventor, he have been granted 12 invention patents and 8 utility model patents.