Faster-GS: Analyzing and Improving Gaussian Splatting Optimization

Florian Hahlbohm[1] Linus Franke[2] Martin Eisemann[1] Marcus Magnor[1]

[1]Computer Graphics Lab, TU Braunschweig, Germany [2]Inria, Université Côte d'Azur, France
{lastname}@cg.tu-bs.de {firstname.lastname}@inria.fr

https://fhahlbohm.github.io/faster-gaussian-splatting

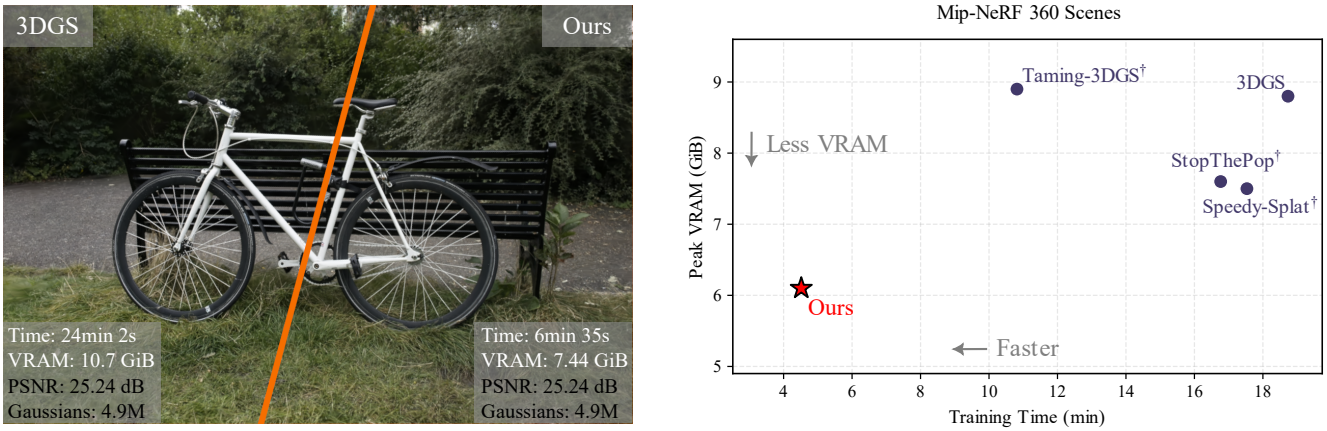

Figure 1. Our method, *Faster-GS*, substantially accelerates training and reduces GPU memory (VRAM) usage compared to the original 3DGS algorithm [37] without altering quality or number of Gaussians (left). Averaged over all Mip-NeRF360 scenes on an RTX 4090 GPU (right), we train 4.1× faster with 30% less VRAM than 3DGS. On the Deep Blending dataset (not depicted), the speedup is more than 5.2×. We also outperform improved implementations from prior works [25, 56, 65]. †Baseline modified to keep quality/#Gaussians unchanged.

Abstract

Recent advances in 3D Gaussian Splatting (3DGS) have focused on accelerating optimization while preserving reconstruction quality. However, many proposed methods entangle implementation-level improvements with fundamental algorithmic modifications or trade performance for fidelity, leading to a fragmented research landscape that complicates fair comparison. In this work, we consolidate and evaluate the most effective and broadly applicable strategies from prior 3DGS research and augment them with several novel optimizations. We further investigate underexplored aspects of the framework, including numerical stability, Gaussian truncation, and gradient approximation. The resulting system, Faster-GS, provides a rigorously optimized algorithm that we evaluate across a comprehensive suite of benchmarks. Our experiments demonstrate that Faster-GS achieves up to 5× faster training while maintaining visual quality, establishing a new cost-effective and resource efficient baseline for 3DGS optimization. Furthermore, we demonstrate that optimizations can be applied to 4D Gaussian reconstruction, leading to efficient non-rigid scene optimization.

1. Introduction

In their seminal work on *3D Gaussian Splatting (3DGS)* [37], Kerbl *et al*. introduce a scene representation and novel view synthesis framework that unifies the strengths of classical point-based rendering [103] with gradient-based optimization techniques from differentiable volumetric rendering [58].

Owing to its remarkable combination of visual fidelity and real-time performance, 3DGS rapidly became the dominant approach for novel view synthesis and inspired a broad range of subsequent research in computer vision and computer graphics [51, 57, 79, 87], as well as interest in domains such as digital film production [62]. The extensive adoption underscores the importance and impact of cost-effective reconstructions, the central theme of this paper.

The sustained momentum of this research area can partially be attributed to the rapid emergence of numerous extensions addressing specific aspects of the original formulation, including anti-aliasing [98], densification [39, 68], compression [1], rendering approximations [24, 65], and improvements in inference speed and scalability [15, 38, 48, 56, 73]. Notably, recent performance-oriented variants achieve high-

quality reconstructions within minutes, even on consumer-grade hardware, further accelerating research and experimentation in this field. However, the rapid pace of progress also poses a practical challenge: the continuous influx of improvements often outpaces the ability to integrate and evaluate them cohesively when developing new methods.

Our work is motivated by two main observations: First, recent research on 3D Gaussian Splatting (3DGS) has led to a fragmented landscape of extensions and optimizations, making it increasingly difficult to assess the upper bound of achievable performance when integrating all available net-positive contributions. This issue is particularly evident for *training performance*, where the goal is to reduce optimization time while maintaining the reconstruction quality of the original 3DGS method [37]. Advances in this domain are often tightly integrated with more fundamental modifications to the algorithm or underlying representation [24, 65]. Other improvements, in contrast, have been developed primarily for inference (i.e., novel view synthesis rendering) [15, 73], although parts of these techniques could, in principle, be adapted for training, which we demonstrate in this work.

Second, while community-driven frameworks such as *gsplat* [96] have made significant progress toward extensible and modular implementations that facilitate fundamental modifications to the algorithm [31, 86], this hinders the integration of performance optimization targeting the original pipeline [37, 103], revealing a gap in the research landscape.

Our work aims to provide a solution to the core problem implied by these observations, *i.e.*, the lack of an updated 3DGS baseline regarding training performance. Thus, in this paper, we both survey the 3DGS follow-up works for performance improvements as well as integrate novel improvements. We evaluate their efficiency and integrate them into a new, optimized 3D Gaussian Splatting framework (*cf*. Fig. 1). Specifically, we observe that most recent approaches aim to **reduce memory accesses** on the GPU through fewer duplicated Gaussians during differentiable rasterization [25, 65, 82], reduced memory accesses during sorting [73], or memory-efficient thread-to-workload setups [15, 56], all with varying degrees of effectiveness. Additionally, several works report performance gains through *kernel fusion*, where multiple computational stages are combined into a single pass [17, 24, 56].

Beyond **integrating**, comparing, and evaluating these established techniques, we reveal and introduce methods to further decrease memory access costs by leveraging **memory coalescence**, which improves cache locality and bandwidth utilization. We apply all optimizations to the training schedule of 3DGS and showcase a speedup of up to 5× in training speeds, resulting in an average reconstruction time of 163 seconds for the 3DGS benchmark [37] with full quality and all Gaussians. We specifically exclude lower-precision and hardware-level optimizations [46, 48], pruning strategies [25], dense Gaussian initialization [43], or feed-forward pipelines [6, 35] as they fundamentally change results, whereas our aim is to retain compatibility with the original widely-used CUDA-based differentiable rasterization pipeline [37].

We further showcase the effectiveness of this work and its potential impact on future 3DGS-based research by extending our implementation to multi-dimensional Gaussians (4D) for dynamic scene reconstruction based on the work by Yang *et al*. [93]. We make the following key contributions:

- A comprehensive survey, discussion, and evaluation of 3DGS training optimization strategies, providing a comparative analysis of previously published improvements.
- Novel performance optimizations that exploit *memory coalescence* and *fuse gradient computations and parameter updates*, significantly accelerating training without compromising reconstruction quality.
- The introduction of *Faster-GS*, an integrated and state-of-the-art 3DGS training pipeline that consolidates all effective techniques for real-time performance.

Our full implementation, including code/scripts for the presented experiments, are available on our project page. Apart from a plug-and-play solution for existing 3DGS methods, it provides a testbed designed to facilitate future research and fair comparison within this rapidly evolving domain.

2. Preliminaries and Related Work

2.1. 3D Gaussian Splatting (3DGS)

To model a scene, 3DGS [37] uses a set of unnormalized 3D Gaussian point primitives, each of which is defined by its 3D mean μ, anisotropic 3D covariance Σ, and a scalar opacity $o \in (0, 1)$. Additionally, each primitive is equipped with a set of spherical harmonics (SH) coefficients to represent view-dependent color changes. In practice, 3DGS uses coefficients up to degree three $\hat{c} \in \mathbb{R}^{3\times 16}$ that can be evaluated for a given viewing direction to obtain an RGB color c.

Rendering. To render a Gaussian, 3DGS first transforms relevant Gaussian parameters to camera space and employs splatting [103] to obtain a 2D Gaussian in screen space.

$$\mu_{\text{2D}} = \begin{pmatrix} \frac{f_x \hat{\mu}_x}{\hat{\mu}_z} + c_x \\ \frac{f_y \hat{\mu}_y}{\hat{\mu}_z} + c_y \end{pmatrix}, \text{ with } \hat{\mu} = W(\mu_x, \mu_y, \mu_z, 1)^\top \quad (1)$$

$$\Sigma_{\text{2D}} = J W_{1:3,1:3} \Sigma W_{1:3,1:3}^\top J^\top, \; J = \begin{pmatrix} \frac{f_x}{\hat{\mu}_z} & 0 & -\frac{f_x \hat{\mu}_x}{\hat{\mu}_z^2} \\ 0 & \frac{f_y}{\hat{\mu}_z} & -\frac{f_y \hat{\mu}_y}{\hat{\mu}_z^2} \end{pmatrix} \quad (2)$$

where $W \in \mathbb{R}^{4\times 4}$ the transformation from world to camera space, $J \in \mathbb{R}^{2\times 3}$ the affine approximation of the perspective projection, and f_x, f_y, c_x, c_y the intrinsic camera parameters, *i.e.*, focal lengths and optical center in screen coordinates. The result is a 2D Gaussian that can, after truncation (Kerbl *et al*. [37] use $\approx 3.33\sigma$), be rasterized and

evaluated at any pixel position $x \in \mathbb{R}^{2\times1}$ efficiently, yielding a transparency value α for blending:

$$\alpha = o \cdot e^{-\frac{1}{2}(x-\mu_{2D})^\top \Sigma_{2D}^{-1}(x-\mu_{2D})}. \quad (3)$$

Pixel colors are computed by alpha-blending all N fragments in each pixel in front-to-back order:

$$C = \sum_{i=1}^{N} \alpha_i T_i c_i + T_N c_{\text{bg}} \text{ with } \quad T_i = \prod_{j=1}^{i-1}(1-\alpha_j), \quad (4)$$

where c_{bg} is an arbitrary background color. As an approximation to exact but prohibitively slow per-pixel sorting [65], 3DGS employs approximate, object-level sorting, where 3D Gaussians are sorted based on $\hat{\mu}_z$ prior to rasterization.

Activation Functions. To ensure stable, gradient-based optimization, 3DGS employs activation functions for all parameters except μ. Opacity is constrained to lie in $(0, 1)$ by applying a Sigmoid activation and the view-dependent color c is clipped to be positive. To ensure Σ is a valid covariance, *i.e.*, positive semi-definite, 3DGS optimizes separate scaling and rotation components from which Σ is computed during rendering using $\Sigma = RSS^\top R^\top$, where R is a 3D rotation matrix and S is a diagonal scaling matrix. R is optimized as a quaternion that is normalized before rendering. Furthermore, so that the scales of the 3D Gaussian remain positive, the exponential function is used as an activation prior to rendering. Mathematically, this means S stores the standard deviations of the 3D Gaussian along the principal axes defined by R, *i.e.*, the three eigenvectors of Σ.

Optimization Schedule and Hyperparameters. Starting from a sparse point cloud or random initialization, Kerbl *et al*. [37] introduce adaptive density control (ADC), a set of carefully tuned heuristics for cloning, splitting, and pruning Gaussians during optimization. Specifically, 3DGS tracks the magnitude of the μ_{2D} gradients during training for all Gaussians and clones or splits Gaussians at regular intervals. Note that this growing set of Gaussians causes memory fragmentation, slowing down optimization in later iterations.

The training schedule and hyperparameters Kerbl *et al*. provided with the initial code release have mostly remained unchanged in follow-up work. Models are trained for 30000 iterations, in each of which an image is rendered from a randomly sampled training viewpoint and compared with the ground truth image, with losses as a combination of L1 and D-SSIM and parameter updates with Adam [41]. For a full breakdown of the schedule, see Sec. A.

An important change that has been integrated into the official 3DGS codebase of Kerbl *et al*. [37] since the initial release is a change in the opacity learning rate, which was halved from 0.05 to 0.025 following Mallick *et al*. [56]. This change affects the number of Gaussians created during optimization, with the new, lower learning rate leading to slightly cleaner reconstructions with fewer Gaussians.

2.2. Improvements and Follow-up Works

Rendering. At the heart of the original 3DGS algorithm and implementation by Kerbl *et al*. [37] is a tile-based, differentiable software rasterizer implemented in C++/CUDA. To reduce overhead, extensions for computing tight bounding boxes and exact splat/tile intersections have been proposed [25, 65, 82]. A major effort has also been on reducing approximations and artifacts in 3DGS rendering by moving to ray-based evaluation schemes [24, 59, 77, 99] or proper volumetric rendering [3–5, 12] to avoid distortion artifacts tied to the original splatting approach [33], improving blending accuracy at the pixel level [24, 30, 40, 65], and suitable anti-aliasing strategies [74, 98]. Other works focus on improving performance specifically during inference, *e.g.*, to speed up rendering when the number of primitives is high by devising optimized pipelines for sorting Gaussians [73], or improving the underlying data access patterns and control flow [15, 21]. Complementary to these approaches, several works target architecture-level optimizations and use hardware acceleration [46, 48, 90], which trade configurability and numerical precision for performance. Recent work also investigates efficient rendering on HMDs through foveated rendering [17, 80].

Optimization. During optimization, subsequent works improve the original algorithm by analyzing and enhancing the underlying densification heuristics [14, 19, 68, 97], distributing Gaussians based on anchor points [52], integrating probabilistic models [39], or fully replacing densification in favor of dense initialization [43]. An equally important aspect of optimization is avoiding excessive growth in the number of Gaussians, *e.g.*, by pruning obsolete Gaussians [18, 61] or by reducing the number created during optimization indirectly by repeatedly reducing the opacity of each Gaussian [65]. Other work addresses challenges commonly associated with real-world data, *e.g.*, camera lens distortions [86], dynamic distractors [70], and textureless regions and lighting variations, which can be resolved through depth regularization [11] and decoupled appearance modules [49]. Recent works also investigate alternatives for updating the Gaussian parameters during training by extending the Adam optimizer [41] to account for visibility [56] or by replacing it with second-order optimization algorithms [34, 45]. Furthermore, adaptive Gaussian scheduling [9] improves optimization speeds and feed-forward methods rely on large, pretrained models to fully avoid per-scene optimization and reduce reliance on dense input images [6, 10, 35, 76, 78, 88].

Representation and Applications. Efficiency and portability, especially to allow for the reconstruction of larger scenes also recently gained interest. Approaches in this area reason about the importance of Gaussians and their attributes to adjust how they are stored [1] and loaded [102], or apply level-

of-detail techniques to enable fast and high-quality rendering of large scenes [38, 44, 67, 84, 91]. Significant effort has also been invested in extending 3DGS to non-rigid, *i.e.*, dynamic scenes [32, 53, 83, 85, 89, 92, 93], efficient editing of trained models [8, 54, 73], and meshing [7, 20, 22, 31, 66, 99, 100]. Beyond the original 3D Gaussian-based representation, there are also multiple works that build on the underlying pipeline proposed by Kerbl *et al*. [37], but use different primitives to *e.g*. improve surface reconstruction [31, 95], the representation of sharp edges [27, 28, 50, 81], or enable exact, *i.e.*, overlap-aware volumetric rendering [55].

GPU Optimization. In this work, we exploit various GPU optimization techniques to accelerate training, which allows us to achieve significant speedups over previously introduced techniques. GPUs follow a SIMT (single-instruction multiple-threads) compute paradigm with individual small kernels, which is computationally fast due to the high amount of parallelism in the system [29]. Kernel efficiency is commonly determined by the highest throughput. Typically, kernels are either memory-bound (spending most of their time waiting for or fetching data) or compute-bound, where the arithmetic instructions take the majority of the time. Based on this, different optimization techniques can be used, such as exploiting GPU shared memory to allow multiple threads to load and access data efficiently and share costs. For details, see the comprehensive survey of Hijma *et al*. [29].

3. Method

We introduce a high-performance 3DGS optimization framework, *Faster-GS*, which follows the same paradigm as the original method [37] with significant speed increases. We first describe the scope and basis for this work (Sec. 3.1), then consolidate and review recent optimization techniques for 3DGS (Sec. 3.2), and integrate further improvements (Sec. 3.3). Lastly, we present easy integration into 4D Gaussian Splatting (Sec. 3.4).

3.1. Scope and Basis

Gaussian Splatting Performance. Rendering splats with large screen space extensions involves scattered memory writes and is a common performance issue compared to, *e.g*., pixel-sized splats [16, 69, 71]. Kerbl *et al*. [37] circumvent this problem by using a tile-based software rasterizer, which splits the image-plane into 16×16 tiles and intersects the bounding box of each splat with them, duplicating Gaussians for later stages into per-tile splat lists while streamlining memory. Processing the splat lists is severely memory-bound, especially due to the high number of floats necessary for each Gaussian (see Sec. 2.1), which need to be loaded. As such, reducing memory and memory accesses is the predominant way to accelerate 3DGS optimization, which we will discuss in the following sections.

Scope of this Work. Our objective is to adhere closely to the original 3DGS optimization and outcomes, facilitating easy integration into existing works as well as prevalent frameworks [37, 75, 96]. We avoid extensive pruning or culling during reconstruction, which can enhance performance; however, it incurs (minimal) quality losses [14, 18, 25, 65]. Additionally, we avoid compression, as it requires a careful trade-off between quality and compute [1]. However, while outside the scope of this work, further integration of pruning or compression should greatly increase speeds further [25].

Basis Implementation. For a clean testbed and open-source version, we developed a refactored 3DGS implementation with several improvements aimed at enhancing numerical stability, efficiency, and modularity. In particular, using front-to-back alpha blending for the backward pass (Eq. (4)) removes the need for division-by-zero checks, and refined handling of degenerate quaternions of Gaussian covariances stabilizes gradients. Furthermore, explicit handling of μ_{2D} gradients and visibility masks reduces VRAM overhead. For further details, see Sec. B.1. This *basis* version increases performance by about 15% compared to Kerbl *et al*. [37].

3.2. A Survey of Recent Improvements

In the following, we compile and group techniques from recent 3DGS literature to address memory bottlenecks, ensuring no negative impact on reconstruction quality. We highlight required contributions to the training pipeline and integrate minor novel optimizations in them.

Splat Bounding. The splatting algorithm of Eqs. (1) and (2) results in a 2D Gaussian on the image plane. 3DGS skips all fragments created during rasterization where α (see Eq. (3)) is below $\tau_\alpha = 1/255$. This corresponds to truncating the Gaussian at roughly 3.33σ and allows bounding the relevant area of the splat with a 2D ellipse. As efficient tile-based rendering requires creating per-tile lists of all contributing splats, the bounding box of this 2D ellipse is of interest. In 3DGS, Kerbl *et al*. use a square-shaped bounding box, which – as previously discussed by Radl *et al*. [65] – underestimates the size of this bounding box for opaque, axis-aligned splats. This is because each splat is bound with an axis-aligned square around the circle with radius 3σ instead of 3.33σ, the value corresponding to τ_α. A natural improvement used in prior works [25, 65, 82] is to bound the splats with an axis-aligned rectangle instead of a square, where the center of the rectangle is at μ_{2D} and its half-extents are given by $\sqrt{\Sigma_{2D_{1,1}}}$ and $\sqrt{\Sigma_{2D_{2,2}}}$ respectively.

The opacity o of the Gaussian can also be factored in by multiplying $-2\ln(\tau_\alpha/o)$ with the radicand before applying the square root. This factor follows from setting Eq. (3) equal to τ_α and solving for the numerator of the exponent. This full, opacity-aware formulation leads to a notable decrease in false positives included in each per-tile splat list.

Tile-based Culling. While the aforementioned tight rectangles are the optimal axis-aligned bounding box, they can still overestimate tile intersections for some splats. To eliminate this remaining overhead, Radl *et al*. [65] and Hanson *et al*. [25] propose algorithms for efficiently computing what tiles each ellipse overlaps with. Hanson *et al*. iterate over the shorter side of the bounding rectangle and determine the first and last tile a splat may overlap with in each row/column, which minimizes the number of computations but may cause warp divergence when the rectangle sizes are very different. In contrast, Radl *et al*. implement a load-balanced approach for checking all tiles inside the bounding rectangle by computing the point where the value of the Gaussian is maximal for each tile. For our implementation, we select the approach by Radl *et al*. as we determined it to be faster due to the simpler control flow and added load balancing, but note that the two approaches are not mutually exclusive.

Sorting. To create the per-tile splat lists used for the tiled rasterization approach, 3DGS writes key/value pairs for each tile/Gaussian pair. The original implementation uses 64-bit keys, where the most significant 32 bits indicate the tile index and the 32 least significant ones contain the bits of $\hat{\mu}_z$, *i.e*., depth information. After writing these key/value pairs to a large buffer, they are sorted using a single radix sort to obtain depth-sorted lists of splats for each tile. Recent work by Schütz *et al*. [73] shows that separating this sorting step into two stages, one to establish depth ordering and a second to obtain per-tile lists, reduces VRAM usage as well as the total time spent on sorting. Note that this change requires using a stable sorting algorithm, *e.g*., radix sort. See Sec. B.2 for details.

Per-Gaussian Backward. In the original 3DGS implementation, by far the most expensive operation in terms of runtime is the backward pass computing the alpha blending gradients. This is because each splat may contribute to an arbitrary number of pixels, which introduces the need for using atomic operations for accumulation. As explored by Durvasula *et al*. [13], this is computationally suboptimal, which they solve through custom atomic functions.

Recent work by Mallick *et al*. [56] avoids this problem by parallelizing over Gaussians instead of pixels in the backward pass, which reduces the number of required atomic operations by a factor equal to the number of pixels in each tile, *i.e*., 256 as usually a tile size of 16×16 pixels is used for 3DGS. For efficiency, they store the alpha blending state at each non-empty pixel after every 32^{nd} splat in the respective tile list during the forward pass. While this approach speeds up training significantly by addressing a major bottleneck, it is the only addition that increases VRAM usage.

We integrate and improve the design of Mallick *et al*. [56] by exploiting shared memory to further reduce memory costs and reduce overall VRAM allocations (see Sec. B.3).

Rasterization Kernel Fusion. While the PyTorch-based frontend of the original 3DGS implementation makes it very flexible and easy to extend, most instructions are set up as non-fused, individual CUDA kernels, which frequently have to load and store buffers. Two methods for mitigating this are employed: First, prior works [24, 56] skip the concatenation of the two SH coefficient buffers (3DGS stores these separately to allow for different learning rates of the view-independent and view-dependent bands) and pass these buffers separately to the rasterization backend. This fuses the concatenation into the rasterization kernel and has positive effects on performance (see Sec. 4). Second, we fuse the activation functions for the scales, rotations, and opacities of the Gaussians into the rasterizer to avoid any PyTorch-related overhead. This is already commonly used to speed up inference rendering [24, 73] and for training we can also fuse the gradient computations required for added benefits.

3.3. Refining Optimized Implementations

Integrating the previously surveyed optimizations results in a strong reduction in memory costs and improved performance. We further investigate optimization techniques, adapting this optimized baseline.

Parameter Updates. As we will show in our experiments, a surprisingly expensive part of the training pipeline is the optimizer steps, which adjust each Gaussian's parameters. We find that the main performance issue with the original 3DGS implementation w.r.t. to the optimizer originates from the use of a non-fused Adam update routine. Recent versions of PyTorch allow users to avoid this by passing `fused=True`. A slightly faster alternative involves using FusedAdam from the NVIDIA apex library as a drop-in replacement.

However, for further increased performance, we develop our own Adam implementation in which we precisely match the behavior of the PyTorch implementation while optimizing away all unnecessary overhead. It fully exploits the fusion of the kernel into CUDA, together with fast math operations and fewer overall instructions through fused-multiply additions, further accelerating the optimization.

Locality-preserving Densification. Through the strong reduction of memory costs, we find that memory layout becomes a throttling factor. During densification, new Gaussians are added at the end of the parameter buffers, which causes spatially close Gaussians to be far apart in memory, which results in uncoalesced memory accesses, as neighboring threads need to access different parts of the memory.

To better align Gaussians, we introduce a simple addition to training when densification is active. We regularly apply z-ordering [72] to the current set of Gaussians to ensure neighboring Gaussians in 3D are also close inside the parameter buffers. This reduces warp divergence and cache misses, resulting in faster training when scenes contain many Gaus-

sians. While z-ordering is computationally efficient (roughly 4 ms per million Gaussians), we find that frequent application has diminishing returns. We empirically determined performing this step every 5000 iterations works well across scenes. We further observe that it performs effectively only when used with the per-Gaussian backward pass, as numerous atomic operations in the original backward pass would otherwise heighten atomic contention.

Fusing Backward and Optimizer. As we will show in our experiments, applying all aforementioned improvements leads to a major speedup over the original 3DGS implementation. When profiling the performance of the resulting framework, we find that the GPU spends between 40% and 60% of the total training time on the optimized Adam update routine. To alleviate this bottleneck, we fuse the parameter updates directly into the backward pass of the rasterization module by first loading moments and computing all parameter updates during gradient computation. This reduces VRAM requirements (especially for large amounts of Gaussians) as no additional buffers for the parameters are necessary. However, to maintain correctness w.r.t. the Adam step, we need to perform parameter updates for parameters receiving a gradient of zero in a given iteration, *e.g.*, due to their Gaussian being outside the viewing frustum. This reduces attainable performance improvements with this fused approach.

A recent idea by Mallick *et al*. [56] is to skip updates for these invisible Gaussians, which fits our fused design exceptionally well. We see this as an optional extension for further acceleration, however, as it can cause inconsistencies and performance regressions compared to the original 3DGS implementation (see our evaluation and Mallick *et al*. [56]).

3.4. Extension to Dynamic Scenes

Managing dynamic objects in a scene is a critical issue in 3D reconstruction. Although these elements can occasionally be considered distractors [70], the dynamic object, such as a human, frequently constitutes the most relevant aspect of a scene. We integrate our performance-optimized Gaussian Splatting framework to support optimization of 4D Gaussians based on the approach by Yang *et al*. [93]. A 4D Gaussian is constructed analogously to a 3D Gaussian (Sec. 2.1), with the addition of two parameters accounting for the mean and scale along the temporal dimension. The 4D rotation is separated into a left-isoclinic and a right-isoclinic rotation, each represented by a quaternion. For rendering a given timestep t, Yang *et al*. compute the conditional 3D Gaussian:

$$\mu_{3D|t} = \mu_{1:3} + \Sigma_{1:3,4}\Sigma_{4,4}^{-1}(t - \mu_4), \tag{5}$$

$$\Sigma_{3D|t} = \Sigma_{1:3,1:3} - \Sigma_{1:3,4}\Sigma_{4,4}^{-1}\Sigma_{4,1:3}. \tag{6}$$

In combination with the value of the marginal distribution, *i.e.*, the 1D Gaussian $p(t) = \mathcal{N}(t; \mu_4, \Sigma_{4,4})$ evaluated at t, multiplied by the result of Eq. (3), Yang *et al*. develop a differentiable approach for 4D Gaussian rendering.

We integrate this approach into our optimized 3DGS framework by adapting the data model accordingly and extending the rasterizer kernels to compute the conditional/marginal Gaussians and the relevant gradients in the forward and backward pass respectively. We also extend our training schedule to match that of Yang *et al*., who render and propagate the loss for multiple images in each training iteration. We note that with this, the previous optimizations are directly transferable to 4D Gaussian optimization.

4. Evaluation

We evaluate our developed framework in a comprehensive suite of experiments with three main goals. Confirming that the quality has not regressed relative to pertinent baselines, examining the speed enhancements related to each addition both individually and collectively, and analyzing the extension of our non-rigid reconstruction method.

4.1. Setup

As baselines, we compare with the official 3DGS implementation by Kerbl *et al*. [37], the 3DGS implementation of Radl *et al*. with tight, opacity-aware bounding boxes and load-balanced tile-based culling [65], a variant of Speedy-Splat [25] only using the SnugBox and AccuTile features, and the 3DGS-accel branch of the official 3DGS codebase. The latter is effectively identical to the 3DGS implementation from Taming-3DGS [56], which uses opacity-aware tile-based culling without load balancing, per-Gaussian backwards, and separate SH buffers within the rasterizer. For all methods, we use the fused SSIM implementation by Goel *et al*. as proposed in Taming-3DGS [56] during loss computation. We unify hyperparameters across all methods, which is necessary following a recent change in the official 3DGS codebase. Training images are uploaded to VRAM before optimization, which is excluded from the reported training times but included in peak VRAM. Unless otherwise noted, all experiments were conducted on the same hardware using a single RTX 4090 GPU. We use the standard benchmark for 3DGS methods, *i.e.*, 13 scenes from the Mip-NeRF360 [2], Tanks and Temples [42], and Deep Blending [26] datasets with a 7:1 train/test split. Image quality metrics (PSNR, SSIM, and LPIPS [101]) are computed under identical conditions, *i.e.*, with the same script, where we ensure a correct LPIPS computation by normalizing images to $[-1, 1]$. We also use pre-downscaled images for training and testing when these are provided with the dataset [2].

4.2. Results

In Tab. 1, we show averaged results for baselines and our implementation. As expected, all methods achieve the same image quality and optimize to roughly the same number of Gaussians. Note that image quality results can vary significantly between runs, even when using the same fixed random

Table 1. Quantitative comparisons on the Mip-NeRF360, Tanks and Temples, and Deep Blending datasets. For baselines marked with † we enable only those contributions that do not affect quality. The three best results are highlighted in **green** in descending order of saturation.

Method	Mip-NeRF360 [2]						Tanks and Temples [42]						Deep Blending [26]					
	PSNR↑	SSIM↑	LPIPS↓	Train↓	VRAM↓	#Gs↓	PSNR↑	SSIM↑	LPIPS↓	Train↓	VRAM↓	#Gs↓	PSNR↑	SSIM↑	LPIPS↓	Train↓	VRAM↓	#Gs↓
3DGS [37]	27.53	0.815	0.256	18m44s	8.8GiB	2.74M	23.77	0.852	0.204	11m26s	4.7GiB	1.57M	29.81	0.907	0.305	19m43s	8.1GiB	2.47M
Speedy-Splat† [25]	27.53	0.816	0.255	17m32s	7.5GiB	2.72M	23.77	0.852	0.205	10m34s	4.1GiB	1.57M	29.79	0.906	0.304	18m40s	7.1GiB	2.55M
StopThePop† [65]	27.54	0.816	0.255	16m46s	7.6GiB	2.73M	23.76	0.852	0.205	9m51s	4.1GiB	1.57M	29.83	0.907	0.304	17m47s	7.1GiB	2.55M
Taming-3DGS† [56]	27.53	0.815	0.256	10m49s	8.9GiB	2.73M	23.78	0.852	0.203	7m04s	4.9GiB	1.57M	29.81	0.906	0.305	9m01s	8.4GiB	2.47M
Basis Impl. [Ours]	27.57	0.816	0.255	15m57s	6.3GiB	2.67M	23.79	0.853	0.204	9m39s	3.4GiB	1.52M	29.74	0.907	0.304	17m15s	6.0GiB	2.52M
Ours	27.56	0.816	0.254	4m31s	6.1GiB	2.73M	23.75	0.853	0.204	3m04s	3.4GiB	1.55M	29.78	0.906	0.304	3m46s	6.0GiB	2.61M

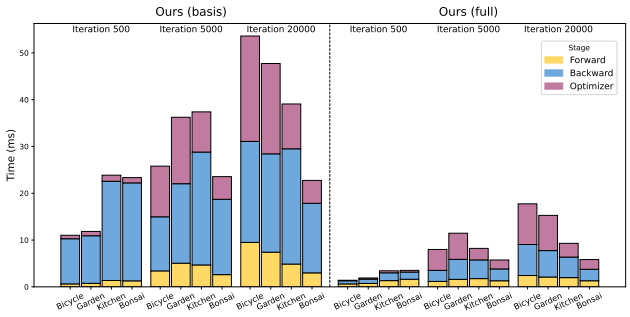

Figure 2. Runtime comparison for the basis and full versions of our optimized 3DGS framework. We measure the time it takes to compute the forward/backward pass and the optimizer step respectively during iteration 500, 5000, and 20000 when training four scenes from the Mip-NeRF360 dataset [2].

seed due to the interaction between floating point math and random ordering during gradient accumulation. For some scenes, *e.g.*, Bonsai from Mip-NeRF360 [2], this can lead to a major difference of up to 0.5 dB PSNR in all implementations. Thus presented image quality metrics are averaged across five runs. The main difference between methods is in the training time and VRAM consumption, where our implementation outperforms all baselines significantly, by up to 5.2× compared to 3DGS [37] and 2.4× compared to Taming-3DGS [56] on the Deep Blending scenes. Notably, our basis implementation also achieves strong results.

Individual Components. Starting from our basis implementation, we show the isolated performance impact of all changes in Tab. 2. We report results for the indoor and outdoor scenes of the Mip-NeRF360 [2] separately due to different optimization behavior. Specifically, the number of Gaussians created during densification are a lot higher for outdoor scenes (3.8M vs. 1.3M Gaussians on average), while indoor scenes use images with roughly 1.5× more pixels.

Particularly impressive speedups are obtained by using any of the three fused Adam techniques, with our implementation consistently outperforming those from PyTorch and apex. The per-Gaussian backward pass for alpha blending also leads to a major speedup – especially when the number of primitives is small – but at the same time is the only change that increases VRAM usage.

We also find that the load-balanced approach for creating Gaussian/tile instances has an increasingly negative impact on training speed as the number of primitives increases. Note that full tile-based culling is also affected. When analyzing this, we found the underlying issue to be warp divergence, where more than half of the threads in each warp are inactive because their Gaussian is invisible from the current viewpoint. While load balancing is meant to address precisely this issue, we find that the associated uncoalesced memory accesses are a bigger bottleneck.

We can resolve this issue by sorting Gaussians in z-order at regular intervals during training. While this works great for scenes with many Gaussians, it significantly slows down training when the number of Gaussians is small due to a massive increase in atomic contention in the alpha blending backward pass. Specifically, threads in different tiles are more likely to write to the same cache line, which reduces performance as operations are serialized and cache lines are invalidated after every write (a.k.a. false sharing). Fortunately, this is much less of an issue with the per-Gaussian backward pass enabled in our full implementation, where omitting our repeated z-ordering during densification slows down training across all scene subsets.

In Fig. 2, we investigate the time spent per algorithm step and find that computing parameter updates is a significant fraction of the runtime. We evaluate approaches for decreasing this bottleneck in Tab. 3. Fusing the optimizer with the backward pass of the rasterizer (see Sec. 3.3) provides a small speedup and VRAM reduction over our full version but reduces its extensibility. After integration, parameter updates remain the main bottleneck of training. Skipping all parameter updates for invisible Gaussians [56] or omitting view-dependent SH coefficients speeds up training and reduces VRAM usage, but it also slightly degrades quality.

GPU Comparison. We investigate the advantages of our improved 3DGS framework across different GPUs from the three most recent consumer-grade generations of Nvidia GPUs. As shown in Tab. 4, newer GPUs exhibit greater speedup, suggesting anticipated performance improvements in upcoming hardware generations. On an RTX 5090 GPU, training takes 163 seconds on average, a 5× improvement over the original implementation.

Table 2. Ablations on the Mip-NeRF360, Tanks and Temples, and Deep Blending datasets using an RTX 4090 GPU. Relative improvements were computed before rounding and indicate the speedup/reduction in training time and peak VRAM usage respectively.

Method	Mip-NeRF360 [2] - Outdoor		Mip-NeRF360 [2] - Indoor		Tanks and Temples [42]		Deep Blending [26]	
	Training↓	VRAM↓	Training↓	VRAM↓	Training↓	VRAM↓	Training↓	VRAM↓
Basis	17m07s	6.39GiB	14m29s	6.23GiB	9m39s	3.43GiB	17m15s	6.04GiB
+ fused activations	16m19s (1.05×)	6.27GiB (0.98×)	14m00s (1.03×)	6.19GiB (0.99×)	9m19s (1.04×)	3.37GiB (0.98×)	16m25s (1.05×)	5.94GiB (0.98×)
+ separate SH buffers	15m39s (1.09×)	5.72GiB (0.89×)	14m10s (1.02×)	5.99GiB (0.96×)	9m13s (1.05×)	3.13GiB (0.91×)	16m31s (1.04×)	5.57GiB (0.92×)
+ rectangular AABBs	16m52s (1.02×)	6.39GiB (1.00×)	13m58s (1.04×)	6.16GiB (0.99×)	9m17s (1.04×)	3.38GiB (0.99×)	16m44s (1.03×)	5.97GiB (0.99×)
+ rect. AABBs w/ opacity	16m46s (1.02×)	6.38GiB (1.00×)	13m47s (1.05×)	6.12GiB (0.98×)	9m13s (1.05×)	3.36GiB (0.98×)	16m44s (1.03×)	5.92GiB (0.98×)
+ load-balanced instancing	17m10s (1.00×)	6.40GiB (1.00×)	14m22s (1.01×)	6.23GiB (1.00×)	9m35s (1.01×)	3.42GiB (1.00×)	17m07s (1.01×)	6.04GiB (1.00×)
+ full tile-based culling	16m53s (1.01×)	6.39GiB (1.00×)	13m40s (1.06×)	6.11GiB (0.98×)	9m11s (1.05×)	3.35GiB (0.98×)	16m43s (1.03×)	5.89GiB (0.97×)
+ separate sorting	16m56s (1.01×)	6.33GiB (0.99×)	14m05s (1.03×)	6.12GiB (0.98×)	9m34s (1.01×)	3.36GiB (0.98×)	16m58s (1.02×)	5.90GiB (0.98×)
+ per-Gaussian backward	14m14s (1.20×)	7.69GiB (1.20×)	7m52s (1.84×)	7.84GiB (1.26×)	6m56s (1.39×)	4.49GiB (1.31×)	10m27s (1.65×)	8.36GiB (1.38×)
+ fused Adam (PyTorch)	14m03s (1.22×)	6.42GiB (1.00×)	13m40s (1.06×)	6.23GiB (1.00×)	8m40s (1.11×)	3.42GiB (1.00×)	15m22s (1.12×)	6.05GiB (1.00×)
+ fused Adam (Apex)	13m12s (1.30×)	6.41GiB (1.00×)	12m59s (1.12×)	6.22GiB (1.00×)	8m02s (1.20×)	3.42GiB (1.00×)	14m38s (1.18×)	6.04GiB (1.00×)
+ fused Adam (Ours)	12m50s (1.33×)	6.40GiB (1.00×)	12m55s (1.12×)	6.23GiB (1.00×)	7m55s (1.22×)	3.42GiB (1.00×)	14m32s (1.19×)	6.04GiB (1.00×)
Full w/o z-ordering	5m52s (2.92×)	5.99GiB (0.94×)	3m17s (4.41×)	6.25GiB (1.00×)	3m11s (3.02×)	3.36GiB (0.98×)	3m50s (4.50×)	5.95GiB (0.98×)
Full	5m31s (3.10×)	5.99GiB (0.94×)	3m14s (4.47×)	6.29GiB (1.01×)	3m04s (3.14×)	3.39GiB (0.99×)	3m46s (4.58×)	5.96GiB (0.99×)

Table 3. Integrating the optimizer step into the backward pass results in a minor speed enhancement. Avoiding updates for non-visible Gaussians or excluding view-dependent spherical harmonics (SH) affects quality. Results are averaged over the five outdoor scenes from the Mip-NeRF360 dataset [2].

	PSNR↑	Train↓	VRAM↓	#Gs↓
Full	24.72	5m31s	6.0GiB	3.87M
+ fused updates	24.73	5m04s	5.6GiB	3.89M
+ fused updates (skip invisible)	24.59	3m03s	5.1GiB	3.34M
+ fused updates (SH degree=0)	24.38	2m24s	3.2GiB	3.85M

Table 4. Training time across all 13 scenes with different GPUs.

	RTX 3090	RTX 4090	RTX 5090
3DGS [37]	23m46s	17m46s	13m05s
Ours (Full)	6m03s (3.9×)	4m10s (4.3×)	2m43s (4.8×)

Table 5. Comparison with the reference implementation [93] for our extension to 4D Gaussians on the synthetic D-NeRF dataset [64].

	PSNR↑	SSIM↑	LPIPS↑	Train↓	VRAM↓	#Gs↓
Yang *et al*. [93]	31.52	0.960	0.051	18m09s	1.9GiB	0.83M
Ours	31.79	0.960	0.051	6m22s (2.8×)	1.2GiB	0.79M

4.3. Dynamic Scenes

To evaluate our extension to 4D reconstruction (see Sec. 3.4), we compare our implementation against the reference implementation by Yang *et al*. [93] on the eight synthetic scenes from the D-NeRF [64] dataset. All scenes are trained and evaluated at the native dataset resolution of 800×800 pixels using the provided train/test splits. Note that we use one consistent set of hyperparameters across all scenes: We initialize with 100K random points, use the standard view-dependent color parametrization from 3DGS (SH up to degree three), and train for 30000 iterations with a batch size of four. The results in Tab. 5 show that our speedup for standard 3DGS translates to the dynamic scene reconstruction setting as our improved implementation trains up to 3× faster while using less VRAM and maintaining quality. See Sec. C.5 for additional quantitative comparisons on real-world data.

5. Discussion, Limitations, and Future Work

Our optimized framework significantly accelerates Gaussian Splatting and we find that the remaining bottlenecks are tied to the computation of parameter updates. This motivates the integration of second-order optimization algorithms [34, 45] or more compact view-dependent appearance representations [94], which we leave as future work. We also highlight that further optimizations, *e.g*., fusing the forward and backward passes [63] or mixed precision training [60], are possible but will come with a tradeoff between simplicity, robustness, and optimal performance. While not within the scope of our work, we present results for integrating state-of-the-art anti-aliasing and densification techniques [39, 98] as well as an inference-optimized rasterizer implementation based on our testbed to facilitate this process (see Secs. C.1 to C.3). Furthermore, our evaluation sets aside valuable training improvements w.r.t. to artifacts [65], controllability [56], and informed pruning techniques [25] that could be added in the future.

6. Conclusion

In this paper, we surveyed recent 3DGS follow-up works for performance improvements and systematically evaluated their effectiveness. We further integrate memory-efficient adaptations to arrive at a new, optimized Gaussian Splatting framework, *Faster-GS*, that trains 3D and 4D Gaussian scenes up to 5× faster than prior work. Furthermore, we reduce VRAM requirements by up to 30%, making our approach especially cost-effective and feasible to use on lower-end hardware. Ultimately, our framework enables full 3DGS reconstruction in less than two minutes. With its code release, we hope to significantly accelerate future Gaussian Splatting-based view synthesis research.

Acknowledgments

This work was partially funded by the DFG projects "Real-Action VR" (ID 523421583) and "Increasing Realism of Omnidirectional Videos in Virtual Reality" (ID 491805996). Linus Franke was supported by the ERC Advanced Grant NERPHYS (ID 101141721).

References

[1] Milena T. Bagdasarian, Paul Knoll, Yi-Hsin Li, Florian Barthel, Anna Hilsmann, Peter Eisert, and Wieland Morgenstern. 3DGS.zip: A survey on 3D Gaussian splatting compression methods. *CGF*, 44(2), 2025. 1, 3, 4

[2] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-NeRF 360: Unbounded anti-aliased neural radiance fields. In *CVPR*, pages 5460–5469, 2022. 6, 7, 8, 15, 16, 17

[3] Hugo Blanc, Jean-Emmanuel Deschaud, and Alexis Paljic. RayGauss: Volumetric Gaussian-based ray casting for photorealistic novel view synthesis. In *WACV*, 2025. 3

[4] Hugo Blanc, Jean-Emmanuel Deschaud, and Alexis Paljic. RayGaussX: Accelerating Gaussian-based ray marching for real-time and high-quality novel view synthesis. In *ICCV*, 2025.

[5] Adam Celarek, Georgios Kopanas, George Drettakis, Michael Wimmer, and Bernhard Kerbl. Does 3d gaussian splatting need accurate volumetric rendering? *CGF*, 44(2), 2025. 3

[6] David Charatan, Sizhe Lester Li, Andrea Tagliasacchi, and Vincent Sitzmann. PixelSplat: 3D Gaussian splats from image pairs for scalable generalizable 3D reconstruction. In *CVPR*, pages 19457–19467, 2024. 2, 3

[7] Danpeng Chen, Hai Li, Weicai Ye, Yifan Wang, Weijian Xie, Shangjin Zhai, Nan Wang, Haomin Liu, Hujun Bao, and Guofeng Zhang. PGSR: Planar-based Gaussian splatting for efficient and high-fidelity surface reconstruction. *IEEE TVCG*, 31(9):6100–6111, 2025. 4

[8] Yiwen Chen, Zilong Chen, Chi Zhang, Feng Wang, Xiaofeng Yang, Yikai Wang, Zhongang Cai, Lei Yang, Huaping Liu, and Guosheng Lin. GaussianEditor: Swift and controllable 3D editing with Gaussian splatting. In *CVPR*, pages 21476–21485, 2024. 4

[9] Youyu Chen, Junjun Jiang, Kui Jiang, Xiao Tang, Zhihao Li, Xianming Liu, and Yinyu Nie. Dashgaussian: Optimizing 3d gaussian splatting in 200 seconds. In *CVPR*, 2025. 3

[10] Yuedong Chen, Haofei Xu, Chuanxia Zheng, Bohan Zhuang, Marc Pollefeys, Andreas Geiger, Tat-Jen Cham, and Jianfei Cai. MVSplat: Efficient 3D Gaussian splatting from sparse multi-view images. In *ECCV*, pages 370–386, 2025. 3

[11] Jaeyoung Chung, Jeongtaek Oh, and Kyoung Mu Lee. Depth-regularized optimization for 3D Gaussian splatting in few-shot images. In *CVPRW*, 2024. 3

[12] Jorge Condor, Sebastien Speierer, Lukas Bode, Aljaz Bozic, Simon Green, Piotr Didyk, and Adrian Jarabo. Don't splat your Gaussians: Volumetric ray-traced primitives for modeling and rendering scattering and emissive media. *ACM TOG*, 44(1), 2025. 3

[13] Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, and Nandita Vijaykumar. Distwar: Fast differentiable rendering on raster-based rendering pipelines, 2023. 5

[14] Guangchi Fang and Bing Wang. Mini-Splatting: Representing scenes with a constrained number of Gaussians. In *ECCV*, pages 165–181, 2024. 3, 4

[15] Guofeng Feng, Siyan Chen, Rong Fu, Zimu Liao, Yi Wang, Tao Liu, Boni Hu, Lining Xu, Zhilin Pei, Hengjie Li, Xiuhong Li, Ninghui Sun, Xingcheng Zhang, and Bo Dai. FlashGS: Efficient 3D Gaussian splatting for large-scale and high-resolution rendering. In *CVPR*, pages 26652–26662, 2025. 1, 2, 3

[16] Linus Franke, Darius Rückert, Laura Fink, and Marc Stamminger. TRIPS: Trilinear point splatting for real-time radiance field rendering. *CGF*, 43(2), 2024. 4

[17] Linus Franke, Laura Fink, and Marc Stamminger. VR-Splatting: Foveated radiance field rendering via 3D Gaussian splatting and neural points. *PACMCGIT*, 8(1), 2025. 2, 3

[18] Sharath Girish, Kamal Gupta, and Abhinav Shrivastava. EAGLES: Efficient accelerated 3D Gaussians with lightweight encodings. In *ECCV*, pages 54–71, 2024. 3, 4

[19] Glenn Grubert, Florian Barthel, Anna Hilsmann, and Peter Eisert. Improving adaptive density control for 3d gaussian splatting. In *VISIGRAPP*, 2025. 3

[20] Antoine Guédon and Vincent Lepetit. SuGaR: Surface-aligned gaussian splatting for efficient 3d mesh reconstruction and high-quality mesh rendering. In *CVPR*, pages 5354–5363, 2024. 4

[21] Hao Gui, Lin Hu, Rui Chen, Mingxiao Huang, Yuxin Yin, Jin Yang, Yong Wu, Chen Liu, Zhongxu Sun, Xueyang Zhang, et al. Balanced 3DGS: Gaussian-wise parallelism rendering with fine-grained tiling, 2024. 3

[22] Antoine Guédon, Diego Gomez, Nissim Maruani, Bingchen Gong, George Drettakis, and Maks Ovsjanikov. MILo: Mesh-in-the-loop Gaussian splatting for detailed and efficient surface reconstruction. *ACM TOG*, 2025. 4

[23] Florian Hahlbohm, Linus Franke, Leon Overkämping, Paula Wespe, Susana Castillo, Martin Eisemann, and Marcus Magnor. A bag of tricks for efficient implicit neural point clouds. In *VMV*, 2025. 18

[24] Florian Hahlbohm, Fabian Friederichs, Tim Weyrich, Linus Franke, Moritz Kappel, Susana Castillo, Marc Stamminger, Martin Eisemann, and Marcus Magnor. Efficient perspective-correct 3D Gaussian splatting using hybrid transparency. *CGF*, 44(2), 2025. 1, 2, 3, 5, 15

[25] Alex Hanson, Allen Tu, Geng Lin, Vasu Singla, Matthias Zwicker, and Tom Goldstein. Speedy-Splat: Fast 3D Gaussian splatting with sparse pixels and sparse primitives. In *CVPR*, pages 21537–21546, 2025. 1, 2, 3, 4, 5, 6, 7, 8

[26] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM TOG*, 37(6), 2018. 6, 7, 8, 15

[27] Jan Held, Renaud Vandeghen, Adrien Deliege, Abdullah Hamdi, Anthony Cioppa, Silvio Giancola, Andrea Vedaldi, Bernard Ghanem, Andrea Tagliasacchi, and Marc

Van Droogenbroeck. Triangle splatting for real-time radiance field rendering, 2025. 4

[28] Jan Held, Renaud Vandeghen, Abdullah Hamdi, Adrien Deliege, Anthony Cioppa, Silvio Giancola, Andrea Vedaldi, Bernard Ghanem, and Marc Van Droogenbroeck. 3D convex splatting: Radiance field rendering with 3D smooth convexes. In *CVPR*, pages 21360–21369, 2025. 4

[29] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. Optimization techniques for gpu programming. *ACM Comput. Surv.*, 55(11), 2023. 4

[30] Qiqi Hou, Randall Rauwendaal, Zifeng Li, Hoang Le, Farzad Farhadzadeh, Fatih Porikli, Alexei Bourd, and Amir Said. Sort-free Gaussian splatting via weighted sum rendering. In *ICLR*, 2025. 3

[31] Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. 2D Gaussian splatting for geometrically accurate radiance fields. In *SIGGRAPH*, 2024. 2, 4

[32] Junkai Huang, Saswat Subhajyoti Mallick, Alejandro Amat, Marc Ruiz Olle, Albert Mosella-Montoro, Bernhard Kerbl, Francisco Vicente Carrasco, and Fernando De la Torre. Echoes of the coliseum: Towards 3d live streaming of sports events. *ACM TOG*, 44(4), 2025. 4

[33] Letian Huang, Jiayang Bai, Jie Guo, Yuanqi Li, and Yanwen Guo. On the error analysis of 3D Gaussian splatting and an optimal projection strategy. In *ECCV*, pages 247–263, 2024. 3

[34] Lukas Höllein, Aljaž Božič, Michael Zollhöfer, and Matthias Nießner. 3DGS-LM: Faster Gaussian-splatting optimization with Levenberg-Marquardt. In *ICCV*, 2025. 3, 8

[35] Lihan Jiang, Yucheng Mao, Linning Xu, Tao Lu, Kerui Ren, Yichen Jin, Xudong Xu, Mulin Yu, Jiangmiao Pang, Feng Zhao, Dahua Lin, and Bo Dai. AnySplat: Feed-forward 3D Gaussian splatting from unconstrained views. *ACM TOG*, 44(6), 2025. 2, 3

[36] Moritz Kappel, Florian Hahlbohm, and Timon Scholz. NeRFICG, 2026. 13

[37] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkuehler, and George Drettakis. 3D Gaussian splatting for real-time radiance field rendering. *ACM TOG*, 42(4), 2023. 1, 2, 3, 4, 6, 7, 8, 12, 13, 15, 17

[38] Bernhard Kerbl, Andreas Meuleman, Georgios Kopanas, Michael Wimmer, Alexandre Lanvin, and George Drettakis. A hierarchical 3D Gaussian representation for real-time rendering of very large datasets. *ACM TOG*, 43(4), 2024. 1, 4

[39] Shakiba Kheradmand, Daniel Rebain, Gopal Sharma, Weiwei Sun, Yang-Che Tseng, Hossam Isack, Abhishek Kar, Andrea Tagliasacchi, and Kwang Moo Yi. 3D Gaussian splatting as Markov chain Monte Carlo. In *NeurIPS*, 2024. 1, 3, 8, 17

[40] Shakiba Kheradmand, Delio Vicini, George Kopanas, Dmitry Lagun, Kwang Moo Yi, Mark Matthews, and Andrea Tagliasacchi. StochasticSplats: Stochastic rasterization for sorting-free 3D Gaussian splatting. In *ICCV*, 2025. 3

[41] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015. 3, 12

[42] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and Temples: Benchmarking large-scale scene reconstruction. *ACM TOG*, 36(4), 2017. 6, 7, 8, 15

[43] Dmytro Kotovenko, Olga Grebenkova, and Björn Ommer. EDGS: Eliminating densification for efficient convergence of 3DGS, 2025. 2, 3

[44] Jonas Kulhanek, Marie-Julie Rakotosaona, Fabian Manhardt, Christina Tsalicoglou, Michael Niemeyer, Torsten Sattler, Songyou Peng, and Federico Tombari. LODGE: Level-of-detail large-scale Gaussian splatting with efficient rendering. In *NeurIPS*, 2025. 4

[45] Lei Lan, Tianjia Shao, Zixuan Lu, Yu Zhang, Chenfanfu Jiang, and Yin Yang. 3DGS2: Near second-order converging 3D Gaussian splatting. In *SIGGRAPH*, 2025. 3, 8

[46] Sixu Li, Ben Keller, Yingyan (Celine) Lin, and Brucek Khailany. GauRast: Enhancing gpu triangle rasterizers to accelerate 3D Gaussian splatting. In *DAC*, 2025. 2, 3

[47] Tianye Li, Mira Slavcheva, Michael Zollhoefer, Simon Green, Christoph Lassner, Changil Kim, Tanner Schmidt, Steven Lovegrove, Michael Goesele, Richard Newcombe, and Zhaoyang Lv. Neural 3D video synthesis from multi-view video. In *CVPR*, pages 5511–5521, 2022. 18

[48] Zimu Liao, Jifeng Ding, Siwei Cui, Ruixuan Gong, Boni Hu, Yi Wang, Hengjie Li, Hui Wang, Xingcheng Zhang, and Rong Fu. TC-GS: A faster Gaussian splatting module utilizing tensor cores. In *SIGGRAPH Asia*, 2025. 1, 2, 3

[49] Jiaqi Lin, Zhihao Li, Xiao Tang, Jianzhuang Liu, Shiyong Liu, Jiayue Liu, Yangdi Lu, Xiaofei Wu, Songcen Xu, Youliang Yan, and Wenming Yang. VastGaussian: Vast 3D Gaussians for large scene reconstruction. In *CVPR*, 2024. 3

[50] Rong Liu, Dylan Sun, Meida Chen, Yue Wang, and Andrew Feng. Deformable beta splatting. In *SIGGRAPH*, 2025. 4

[51] Xian Liu, Xiaohang Zhan, Jiaxiang Tang, Ying Shan, Gang Zeng, Dahua Lin, Xihui Liu, and Ziwei Liu. Humangaussian: Text-driven 3d human generation with gaussian splatting. In *CVPR*, pages 6646–6657, 2024. 1

[52] Tao Lu, Mulin Yu, Linning Xu, Yuanbo Xiangli, Limin Wang, Dahua Lin, and Bo Dai. Scaffold-GS: Structured 3D Gaussians for view-adaptive rendering. In *CVPR*, 2024. 3

[53] Jonathon Luiten, Georgios Kopanas, Bastian Leibe, and Deva Ramanan. Dynamic 3D Gaussians: Tracking by persistent dynamic view synthesis. In *3DV*, pages 800–809, 2024. 4

[54] Zhaofeng Luo, Zhitong Cui, Shijian Luo, Mengyu Chu, and Minchen Li. VR-Doh: Hands-on 3D modeling in virtual reality. *ACM TOG*, 44(4), 2025. 4

[55] Alexander Mai, Peter Hedman, George Kopanas, Dor Verbin, David Futschik, Qiangeng Xu, Falko Kuester, Jonathan T. Barron, and Yinda Zhang. EVER: Exact volumetric ellipsoid rendering for real-time view synthesis. *arXiv:2410.01804*, 2024. 4

[56] Saswat Subhajyoti Mallick, Rahul Goel, Bernhard Kerbl, Markus Steinberger, Francisco Vicente Carrasco, and Fernando De La Torre. Taming 3DGS: High-quality radiance fields with limited resources. In *SIGGRAPH Asia*, 2024. 1, 2, 3, 5, 6, 7, 8, 13, 14, 16, 17

[57] Hidenobu Matsuki, Riku Murai, Paul HJ Kelly, and Andrew J Davison. Gaussian splatting slam. In *CVPR*, pages 18039–18048, 2024. 1

[58] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, pages 405–421, 2020. 1

[59] Nicolas Moenne-Loccoz, Ashkan Mirzaei, Or Perel, Riccardo de Lutio, Janick Martinez Esturo, Gavriel State, Sanja Fidler, Nicholas Sharp, and Zan Gojcic. 3D Gaussian Ray Tracing: Fast tracing of particle scenes. *ACM TOG*, 43(6), 2024. 3

[60] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM TOG*, 41(4), 2022. 8

[61] Michael Niemeyer, Fabian Manhardt, Marie-Julie Rakotosaona, Michael Oechsle, Daniel Duckworth, Rama Gosula, Keisuke Tateno, John Bates, Dominik Kaeser, and Federico Tombari. RadSplat: Radiance field-informed Gaussian splatting for robust real-time rendering with 900+ fps. In *3DV*, pages 134–144, 2025. 3

[62] Julien Philip, Li Ma, Pascal Clausen, Wenqi Xian, Ahmet Levent Taşel, Mingming He, Xueming Yu, David M. George, Ning Yu, Oliver Pilarski, and Paul Debevec. Detail enhanced gaussian splatting for large-scale volumetric capture. In *SIGGRAPH Asia*, 2025. 1

[63] Yohan Poirier-Ginter, Jeffrey Hu, Jean-François Lalonde, and George Drettakis. Editable physically-based reflections in raytraced gaussian radiance fields. In *SIGGRAPH Asia*, 2025. 8

[64] Albert Pumarola, Enric Corona, Gerard Pons-Moll, and Francesc Moreno-Noguer. D-NeRF: Neural radiance fields for dynamic scenes. In *CVPR*, pages 10313–10322, 2021. 8

[65] Lukas Radl, Michael Steiner, Mathias Parger, Alexander Weinrauch, Bernhard Kerbl, and Markus Steinberger. StopThePop: Sorted Gaussian splatting for view-consistent real-time rendering. *ACM TOG*, 43(4), 2024. 1, 2, 3, 4, 5, 6, 7, 8, 16

[66] Lukas Radl, Felix Windisch, Thomas Deixelberger, Jozef Hladky, Michael Steiner, Dieter Schmalstieg, and Markus Steinberger. SOF: Sorted opacity fields for fast unbounded surface reconstruction, 2025. 4

[67] Kerui Ren, Lihan Jiang, Tao Lu, Mulin Yu, Linning Xu, Zhangkai Ni, and Bo Dai. Octree-GS: Towards consistent real-time rendering with LOD-structured 3D Gaussians. *IEEE TPAMI*, 2025. 4

[68] Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. Revising densification in Gaussian splatting. In *ECCV*, pages 347–362, 2024. 1, 3

[69] Darius Rückert, Linus Franke, and Marc Stamminger. ADOP: Approximate differentiable one-pixel point rendering. *ACM TOG*, 41(4), 2022. 4

[70] Sara Sabour, Lily Goli, George Kopanas, Mark Matthews, Dmitry Lagun, Leonidas Guibas, Alec Jacobson, David Fleet, and Andrea Tagliasacchi. SpotLessSplats: Ignoring distractors in 3D Gaussian splatting. *ACM TOG*, 44(2), 2025. 3, 6

[71] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. Software rasterization of 2 billion points in real time. *PACMCGIT*, 5(3), 2022. 4

[72] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. Rendering point clouds with compute shaders and vertex order optimization. *CGF*, 40(4):115–126, 2021. 5

[73] Markus Schütz, Christoph Peters, Florian Hahlbohm, Elmar Eisemann, Marcus Magnor, and Michael Wimmer. Splatshop: Efficiently editing large Gaussian splat models. *CGF*, 2025. 1, 2, 3, 4, 5, 14

[74] Michael Steiner, Thomas Köhler, Lukas Radl, Felix Windisch, Dieter Schmalstieg, and Markus Steinberger. AAA-Gaussians: Anti-aliased and artifact-free 3D Gaussian rendering. In *ICCV*, pages 27650–27659, 2025. 3, 15, 17

[75] LichtFeld Studio. A high-performance c++ and cuda implementation of 3d gaussian splatting, 2025. 4

[76] Stanislaw Szymanowicz, Christian Rupprecht, and Andrea Vedaldi. Splatter Image: Ultra-fast single-view 3D reconstruction. In *CVPR*, pages 10208–10217, 2024. 3

[77] Chinmay Talegaonkar, Yash Belhe, Ravi Ramamoorthi, and Nicholas Antipa. Volumetrically consistent 3D Gaussian rasterization. In *CVPR*, 2025. 3

[78] Jiaxiang Tang, Zhaoxi Chen, Xiaokang Chen, Tengfei Wang, Gang Zeng, and Ziwei Liu. LGM: Large multi-view Gaussian model for high-resolution 3D content creation. In *ECCV*, pages 1–18, 2024. 3

[79] Fabio Tosi, Youmin Zhang, Ziren Gong, Erik Sandström, Stefano Mattoccia, Martin R Oswald, and Matteo Poggi. How nerfs and 3d gaussian splatting are reshaping slam: a survey. *arXiv preprint arXiv:2402.13255*, 4:1, 2024. 1

[80] Xuechang Tu, Lukas Radl, Michael Steiner, Markus Steinberger, Bernhard Kerbl, and Fernando de la Torre. VRSplat: Fast and robust Gaussian splatting for virtual reality. *PACMCGIT*, 8(1), 2025. 3

[81] Nicolas von Lützow and Matthias Nießner. LinPrim: Linear primitives for differentiable volumetric rendering. *arXiv:2501.16312*, 2025. 4

[82] Xinzhe Wang, Ran Yi, and Lizhuang Ma. AdR-Gaussian: Accelerating Gaussian splatting with adaptive radius. In *SIGGRAPH Asia*, 2024. 2, 3, 4

[83] Yifan Wang, Peishan Yang, Zhen Xu, Jiaming Sun, Zhanhua Zhang, Yong Chen, Hujun Bao, Sida Peng, and Xiaowei Zhou. FreeTimeGS: Free Gaussian primitives at anytime anywhere for dynamic scene reconstruction. In *CVPR*, pages 21750–21760, 2025. 4

[84] Felix Windisch, Thomas Köhler, Lukas Radl, Michael Steiner, Dieter Schmalstieg, and Markus Steinberger. A LoD of Gaussians: Unified training and rendering for ultra-large scale reconstruction with external memory, 2025. 4

[85] Guanjun Wu, Taoran Yi, Jiemin Fang, Lingxi Xie, Xiaopeng Zhang, Wei Wei, Wenyu Liu, Qi Tian, and Xinggang Wang. 4D Gaussian splatting for real-time dynamic scene rendering. In *CVPR*, pages 20310–20320, 2024. 4

[86] Qi Wu, Janick Martinez Esturo, Ashkan Mirzaei, Nicolas Moenne-Loccoz, and Zan Gojcic. 3DGUT: Enabling distorted cameras and secondary rays in Gaussian splatting. In *CVPR*, pages 26036–26046, 2025. 2, 3

[87] Jianfeng Xiang, Zelong Lv, Sicheng Xu, Yu Deng, Ruicheng Wang, Bowen Zhang, Dong Chen, Xin Tong, and Jiaolong Yang. Structured 3d latents for scalable and versatile 3d generation. In *CVPR*, pages 21469–21480, 2025. 1

[88] Haofei Xu, Songyou Peng, Fangjinhua Wang, Hermann Blum, Daniel Barath, Andreas Geiger, and Marc Pollefeys. DepthSplat: Connecting Gaussian splatting and depth. In *CVPR*, pages 16453–16463, 2025. 3

[89] Zhen Xu, Yinghao Xu, Zhiyuan Yu, Sida Peng, Jiaming Sun, Hujun Bao, and Xiaowei Zhou. Representing long volumetric video with temporal Gaussian hierarchy. *ACM TOG*, 43(6), 2024. 4

[90] Mengtian Yang, Yipeng Wang, Chieh-Pu Lo, Xiuhao Zhang, Sirish Oruganti, and Jaydeep P. Kulkarni. GSAcc: Accelerate 3D Gaussian splatting via depth speculation and gaussian-centric rasterization. In *DAC*, 2025. 3

[91] Xijie Yang, Linning Xu, Lihan Jiang, Dahua Lin, and Bo Dai. Virtualized 3D Gaussians: Flexible cluster-based level-of-detail system for real-time rendering of composed scenes. In *SIGGRAPH*. Association for Computing Machinery, 2025. 4

[92] Ziyi Yang, Xinyu Gao, Wen Zhou, Shaohui Jiao, Yuqing Zhang, and Xiaogang Jin. Deformable 3D Gaussians for high-fidelity monocular dynamic scene reconstruction. In *CVPR*, 2024. 4

[93] Zeyu Yang, Hongye Yang, Zijie Pan, and Li Zhang. Real-time photorealistic dynamic scene representation and rendering with 4D Gaussian splatting. In *ICLR*, 2024. 2, 4, 6, 8, 18

[94] Lior Yariv, Peter Hedman, Christian Reiser, Dor Verbin, Pratul P. Srinivasan, Richard Szeliski, Jonathan T. Barron, and Ben Mildenhall. BakedSDF: Meshing neural SDFs for real-time view synthesis. In *SIGGRAPH*, 2023. 8

[95] Keyang Ye, Tianjia Shao, and Kun Zhou. When Gaussian meets surfel: Ultra-fast high-fidelity radiance field rendering. *ACM TOG*, 44(4), 2025. 4

[96] Vickie Ye, Ruilong Li, Justin Kerr, Matias Turkulainen, Brent Yi, Zhuoyang Pan, Otto Seiskari, Jianbo Ye, Jeffrey Hu, Matthew Tancik, and Angjoo Kanazawa. gsplat: An open-source library for gaussian splatting. *Journal of Machine Learning Research*, 26(34):1–17, 2025. 2, 4

[97] Zongxin Ye, Wenyu Li, Sidun Liu, Peng Qiao, and Yong Dou. AbsGS: Recovering fine details for 3D Gaussian splatting, 2024. 3

[98] Zehao Yu, Anpei Chen, Binbin Huang, Torsten Sattler, and Andreas Geiger. Mip-Splatting: Alias-free 3D Gaussian splatting. In *CVPR*, pages 19447–19456, 2024. 1, 3, 8, 15, 16, 17

[99] Zehao Yu, Torsten Sattler, and Andreas Geiger. Gaussian Opacity Fields: Efficient adaptive surface reconstruction in unbounded scenes. *ACM TOG*, 43(6), 2024. 3, 4

[100] Baowen Zhang, Chuan Fang, Rakesh Shrestha, Yixun Liang, Xiaoxiao Long, and Ping Tan. RaDe-GS: Rasterizing depth in Gaussian splatting, 2024. 4

[101] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, pages 586–595, 2018. 6

[102] Brent Zoomers, Maarten Wijnants, Ivan Molenaers, Joni Vanherck, Jeroen Put, Lode Jorissen, and Nick Michiels. PRoGS: Progressive rendering of Gaussian splats. In *WACV*, 2025. 3

[103] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. EWA volume splatting. In *Proc. of the Conference on Visualization*, pages 29–36, 2001. 1, 2

A. 3DGS Training Details

To complement our brief summary of the 3DGS training schedule (Sec. 2.1), we supplement the remaining details and all relevant hyperparameter values in the following. Note that the specified hyperparameters precisely reflect the optimization schedule of the original implementation by Kerbl *et al*. [37]. Recall that to reconstruct a scene from a set of training images, 3DGS first initializes Gaussians at random positions or based on an input point cloud. As this initial point cloud is often very sparse, Kerbl *et al*. [37] introduce adaptive density control (ADC), a set of carefully tuned heuristics for cloning, splitting, and pruning Gaussians during optimization. Specifically, 3DGS tracks the magnitude of the μ_{2D} gradients during training for all Gaussians. At regular intervals, Gaussians for with the average magnitude of this gradient exceeds a predefined threshold are selected for densification. Of the selected Gaussians, large ones are each split into two new, smaller Gaussians with new positions being sampled from the respective parent Gaussian distributions, while those that are small are simply duplicated in place. Additionally, Gaussians that either have a very low opacity or are too large w.r.t. scene size are removed. Densification starts after a warm-up period of 600 iterations and is repeated every 100 iterations thereafter. Gaussians are split/cloned when their average positional gradient across all iterations where they were visible (*i.e*., inside the viewing frustum) since the previous densification step exceeds $2e{-}4$, and pruned when their opacity is below 0.05. After the final densification step in iteration 14900, Gaussians are no longer added or removed from the model.

To further encourage pruning of floaters and incorrectly placed Gaussians, 3DGS resets the opacity of all Gaussians to a small value multiple times during the optimization. The opacity reset is performed every 3000 iterations while densification is active, *i.e*., four times in total and clips opacity values to 0.01 from above.

For the loss function, 3DGS uses a weighted combination of L1 and D-SSIM (*i.e*. $1-$ SSIM) terms, with weights being 0.8 and 0.2 respectively. Parameter updates are then performed using the ADAM optimizer [41] with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e-15$. Learning rates are set to 0.005, 0.001, and 0.025 for scale, rotation, and opacity respectively. We again highlight an important change that has been integrated into the official 3DGS codebase of Kerbl *et al*. [37] since the initial release is a change in the opacity

learning rate, which was halved from 0.05 to 0.025 following Mallick *et al*. [56]. As stated in the main paper, this change affects how many Gaussians are created during optimization with the new, lower learning rate leading to slightly cleaner reconstructions with fewer Gaussians. For the Gaussian means, the learning rate is exponentially decayed from $1.6\text{e}{-4}$ to $1.6\text{e}{-6}$ during optimization and scaled by an additional constant factor that depends on the size of the scene to be reconstructed. For the three view-independent components of the SH coefficients a learning rate of $2.5\text{e}{-3}$ is used, while remaining components use $1.25\text{e}{-4}$. To prevent 3DGS from fitting diffuse color information with view-dependent SH coefficients, only the 0^{th} degree is used at the start of training and the next higher degree being enabled every 1000 iterations.

B. Implementation Details

In this section, we provide a detailed description of our refactored 3DGS implementation that is the *basis* of our testbed as well details of our implementation for the improvements proposed in prior works (*cf*. Sec. 3.2).

B.1. Testbed Basis

As our goal is to investigate the effectiveness of recently proposed optimizations for the original implementation by Kerbl *et al*. [37], we first start off by creating a clean, refactored 3DGS implementation as the basis for our testbed. To avoid developing routines for data loading/management, logging, interactive viewing, and other functionalities commonly required for novel-view synthesis methods, we build our implementation on top on the NeRFICG framework [36]. A key advantage of it is that our *Faster-GS* implementation retains modularity, simplifying future integration into other codebases. Similar to Kerbl *et al*. [37], we use a PyTorch implementation with custom C++/CUDA extensions for frequently executed operations to ensure optimal performance.

The most important of these extensions is the differentiable software rasterizer for 3D Gaussians. Our implementation is based on the original 3DGS, but features a complete rewrite of all CUDA kernels that includes various suitable simplifications and multiple small optimizations. One of these simplifications involves removing the reliance on the OpenGL projection matrix where we instead use the intrinsic camera parameters directly (see Eqs. (1) and (2)). This not only simplifies the math but also allows for properly rendering images with a non-centered optical center. An arguably more significant change is that we compute the gradients of the alpha blending (see Eq. (4)) in front-to-back order. The original implementation does this in back-to-front order, which arguably makes the kernel code much more difficult to read. More importantly, however, back-to-front order requires additional workarounds to ensure numerically stable gradient computation. To this end, Kerbl *et al*. first limit the maximum opacity a fragment can have when computing Eq. (3) by computing $\hat{\alpha} = \min(0.99, \alpha)$, which ensures that repeated division by $(1 - \hat{\alpha}_i)$ when computing the previous transmittance in the backward pass is stable. For similar reasons, they use a somewhat unconventional approach for stopping the computation of Eq. (4) early once the transmittance falls below a threshold $\tau = 1\text{e}{-4}$. Instead of first blending the fragment and checking whether to stop afterward, they first check whether the transmittance would be below τ if the fragment was to be blended and skip it if that check returns true. In combination these fixes ensure numerical stability but arguably also create counterintuitive behavior in certain edge cases. With our implementation using front-to-back order to compute the alpha blending gradients, we can get stable gradients without these workarounds. In a similar spirit, we also employ a more direct approach for handling degenerate 3D Gaussians created by the parameter updates during training. Specifically, a Gaussian in 3DGS is non-degenerate, if and only if its quaternion q as well as all three of its scales are non-zero. To obtain numerically stable gradients in single precision, these values must also not be too small. Therefore, we do not render a Gaussian when $\|q\| < 1\text{e}{-4}$ or $|\Sigma_{2\text{D}}| < 1\text{e}{-6}$. Note that as the former of these conditions is not view-dependent, we add it as an additional pruning condition during densification.

We also apply multiple small optimizations to the PyTorch-based frontend of our 3DGS framework. First, we replace the implicit approach Kerbl *et al*. use to make the rasterizer return the $\mu_{2\text{D}}$ gradients and visibility masks required as the metrics for densification by passing a persistent buffer storing these values to the rasterizer and marking it as non-differentiable. This is slightly faster, requires less VRAM, and arguably much cleaner in general. We also investigate VRAM fragmentation issues originating to frequent changes in buffer sizes during densification. We find that, even when the maximum number of Gaussians is known before optimization, it will never be possible fully avoid these issues due to the number of tile instances being view- and optimization-dependent. Fortunately, recent PyTorch versions support expandable memory segments, which enables lower and more predictable VRAM consumption during training.

Next, we revisit the implementation of the cloning, splitting, and pruning routines used for densification. We find that the original implementation copies the Gaussians' parameters much more often than needed, which we avoid to eliminate unnecessary overhead. Somewhat surprisingly, we also found that the official 3DGS implementation by Kerbl *et al*. actually only performs 29855 optimizer steps with non-zero gradients because the densification routines zero out all gradients. We fix this in our implementation by performing the optimizer step right after the backward pass. While this means ours implementation performs 145 additional opti-

mizer steps, it only makes a small difference in terms of runtime as the number of calls to the forward and backward passes of the rasterizer are not affected by this.

Lastly, we find that after a recent update the official 3DGS codebase clips the colors of the rendered images to the $[0, 1]$ range during training. While the rendered colors can, due to the activation function applied to c, never take on negative values, values be larger than one are possible. However, as clipping sets the gradient to zero, pixels with a final color marginally above one would not be used for optimization. As this may cause unexpected behavior when the input images contain many white pixels due to, *e.g.*, white walls or sky, we stick to the original approach of only clipping during inference.

B.2. Separate Sorting

As detailed in Sec. 3.2, we follow Schütz *et al*. [73] and separate the sorting step for obtaining depth-sorted per-tile splat lists into two sorting routines, which has two key advantages. First, we no longer need a 64-bit data type to store the keys for sorting leading to reduced VRAM usage. The reason for this is that the first sort can use 32-bit keys for depth sorting while the second sort can use 16-bit (or 32-bit keys for very high image resolutions) for the tile sorting. This is possible because the tiled rendering approach with 16×16 pixel tiles makes it so for images up to a resolution of $2^{16} \cdot 256$, *i.e*. roughly 16 megapixels, tile indices fit into a 16-bit unsigned integer. Note that this limit is slightly lower when image dimensions are not a multiple of 16. The second advantage is in the complexity of the sorting, which for radix sort is $\mathcal{O}(kn)$ with k being the number of bits in the sorting key. In the original 3DGS implementation, sorting is performed on a large buffer with 64-bit keys where each Gaussian can contribute an arbitrary amount of entries depending on the number of tiles it contributes to. Importantly, the depth value in the lower 32 bits of the key is the same for all entries from the same Gaussian, while the tile index in the higher 32 bits will be different. Assuming each of our n Gaussians is visible in an average of 8 tiles, the complexity of the combined radix sort is $\mathcal{O}(8 \cdot (16 + 32)n) = \mathcal{O}(384n)$ Note that we assume a key size of 48 bits as the radix sort implementation used by Kerbl *et al*. supports accounting for the fact that not all of the higher 32 bits are needed to store the tile indices (usually 16 bits suffice) and can therefore be disregarded during sorting. When separating the sorting, the complexities of the two steps become $\mathcal{O}(32n)$ and $\mathcal{O}(8 \cdot 16n)$ respectively, *i.e*. the combined complexity is $\mathcal{O}(32n) + \mathcal{O}(128n)$. This clearly shows the advantage of using the separate sorting approach with increasingly higher benefits as the number of Gaussians increases.

Similar to Schütz *et al*. [73], we identify the indices of all visible Gaussians as well as their depth during pre-processing and write them to compacted buffers used for depth sorting through the use of an atomic counter. An important change we make to optimize training performance is that we do not apply this compaction to the intermediate Gaussian values needed for rasterization and blending. This allows our implementation to avoid an additional indirection during gradient computation in the backward pass.

B.3. Per-Gaussian Backward

A major part of the speedup in our optimized implementation comes from the per-Gaussian backward pass (Sec. 3.2) proposed by Mallick *et al*. [56]. It requires three major changes to the implementation. First, additional buffers are allocated to store the intermediate color and transmittance after every 32$^{\text{nd}}$ Gaussian as well as the final number of contributing Gaussians at each pixel. Note that the size of these buffers can easily be determined from the known lengths of the per-tile lists. Second, the forward pass for blending needs to fill these buffers with the corresponding values, which introduces negligible overhead. Third, the backward pass for blending now operates on so-called buckets of 32 Gaussians each where each bucket is associated with a single tile. For each bucket, 32 threads (*i.e*., one warp) are launched with each thread accumulating the gradient of a single Gaussian across the tile associated with the respective bucket. Starting from the buckets initial color and transmittance at each pixel as written in the forward pass, the threads replay the blending process using warp-level primitives to efficiently compute the necessary gradients. Finally, each thread writes the accumulated gradients to global memory using atomics as Gaussians can be inside multiple buckets across tiles. We refer the reader to the the original paper for further details [56].

In addition to integrating our changes with respect to early stopping and numerical stability (see Sec. 3.1), we also extend the idea of Mallick *et al*. for improved performance. In their implementation, the first thread in each warp repeatedly loads the alpha blending state for the next pixel from global memory. As threads must remain synchronized, the entire warp stalls on these global loads. We optimize this by having all 32 threads in the warp collaboratively load a batch of alpha blending states into shared memory (one per thread) before they are needed. From that point on, the first thread in each warp reads the next state directly from shared instead of global memory. Because shared memory access has much lower latency than global memory, this significantly reduces warp stalls. Profiling shows that this change improves the kernel runtime by up to 2×.

C. Further Experiments and Results

In the following, we present results of multiple experiments that are complementary to the evaluation in the main paper.

Table 6. Inference frame rates for all 13 scenes from the Mip-NeRF360, Tanks and Temples, and Deep Blending datasets [2, 26, 42]. The depicted values are the average frames per second when rendering the test set of the respective scene 100 times at the native resolution. For benchmarking 3DGS, we follow Hahlbohm *et al*. [24] and bake all activation functions before rendering to avoid any PyTorch-related overhead. For Ours, we add an inference-optimized version of the forward pass to our testbed where we enable all improvements (*cf*. Sec. 3) that accelerate inference.

	Bicycle	Flowers	Garden	Stump	Treehill	Bonsai	Counter	Kitchen	Room	Train	Truck	DrJohnson	Playroom	Average
3DGS [37]	161.8	387.8	223.7	329.7	306.2	405.4	261.0	219.6	253.2	317.5	340.8	221.3	348.0	290.4
Ours	547.1	901.3	628.5	821.0	824.7	1239.7	1018.3	745.4	1122.4	833.0	863.1	919.2	1280.8	903.4
↪ speedup	3.4×	2.3×	2.8×	2.5×	2.7×	3.1×	3.9×	3.4×	4.4×	2.6×	2.5×	4.2×	3.7×	3.1×

C.1. Inference Rendering Performance

Apart from rapid optimization, another highly relevant aspect of efficient 3D Gaussian Splatting is fast inference rendering.

Of course, many of the improvements that we implement in our testbed (Sec. 3) positively influence frame rate. In Tab. 6, we show that an inference-optimized version of the forward pass from our testbed leads to more than 3× faster rendering during inference.

C.2. Efficient Anti-Aliasing

In Mip-Splatting [98], Yu *et al*. propose two extensions for optimization and rendering that, in combination, effectively prevent aliasing artifacts that occur in the original 3DGS approach when changing the sampling rate after training, *e.g*., by adjusting the focal length or camera distance.

The first extension is a 3D smoothing filter that prevents Gaussians from becoming smaller than the maximal sampling frequency induced by the images used for training. For each primitive, they define the maximal sampling rate of the k^{th} Gaussian as

$$\hat{\nu}_k = \max\left(\left\{\mathbb{1}_n(\mu_k) \cdot \frac{f_n}{d_n}\right\}_{n=1}^{N}\right), \tag{7}$$

where $\mathbb{1}_n(\cdot)$ is an indicator function stating whether the input point is visible in the n^{th} training image, N is the number of training images, f_n is the focal length of the n^{th} training view in pixel units, and d_n is the z-depth of μ_k for the respective view. As $\hat{\nu}_k$ changes whenever μ_k is updated during optimization, Yu *et al*. recompute this value for all Gaussians every 100 iterations during training.

For rendering, the 3D smoothing filter is applied through a Gaussian low-pass filter that influences the three scales $\mathbf{s}_k$ and the opacity o_k of each Gaussian:

$$\hat{\mathbf{s}}_k = \sqrt{\mathbf{s}_k^2 + \frac{\kappa_{3D}}{\hat{\nu}_k^2}} \quad \text{and} \quad \hat{o}_k = \sqrt{\frac{|\text{diag}(\mathbf{s}_k^2)|}{|\text{diag}(\mathbf{s}_k^2 + \frac{\kappa_{3D}}{\hat{\nu}_k^2})|}} o_k, \tag{8}$$

where κ_{3D} is a hyperparameter controlling the variance of the Gaussian filter (0.2 by default).

While disabled by default, we implement two version of this 3D smoothing filter in *Faster-GS*. The first version closely matches the original implementation by Yu *et al*. For the second version, we closely adhere to the underlying theory but use a more direct and efficient approach for enforcing the implied size constraints during optimization. Specifically, we in-place clip the scales of each Gaussian from below based on the 3D smoothing filter after each optimizer step:

$$\mathbf{s}_k = \max(\mathbf{s}_k, \frac{\sqrt{\kappa_{3D}}}{\hat{\nu}_k}). \tag{9}$$

We find that this retains all advantages while being significantly cheaper to compute as this update is independent of gradient computations. Furthermore, we find that accounting for the change in volume of each Gaussian by modifying its opacity is not needed in practice (*cf*. Steiner *et al*. [74]), allowing for further simplifications. To accelerate the repeated computation of $\hat{\nu}_k$ during training, we use a fused CUDA implementation by Hahlbohm *et al*. [24].

The second extension of Yu *et al*. is a 2D Mip filter that mitigates artifacts when rendering Gaussians from further away or with larger focal length than during training. It is an extension to the 2D Gaussian filter that Kerbl *et al*. use in the original 3DGS [37] to prevent aliasing caused by projected 2D Gaussians falling between the pixels due to being too small. Kerbl *et al*. use a 2D Gaussian filter with variance κ_{3D} (0.3 by default):

$$\hat{\Sigma}_{2D} = \Sigma_{2D} + \kappa_{3D}\mathbf{I}. \tag{10}$$

While this approach of dilating every Gaussian works very well during optimization, it does cause aliasing issues during inference. As the virtual camera moves further from a Gaussian it becomes smaller and smaller until the point where $\hat{\Sigma}_{2D}$ in Eq. (10) is dominated by the dilation kernel, which leads to increasingly blurry renderings. To avoid this, Yu *et al*. multiply a view-dependent compensation factor onto the opacity of each Gaussian:

$$\hat{o} = \sqrt{\frac{|\Sigma_{2D}|}{|\hat{\Sigma}_{2D}|}} o. \tag{11}$$

Note that because the full approach of Yu *et al*. combines the 3D smoothing filter with this 2D Mip filter, they use a smaller variance for the 2D Gaussian filter (0.1 by default).

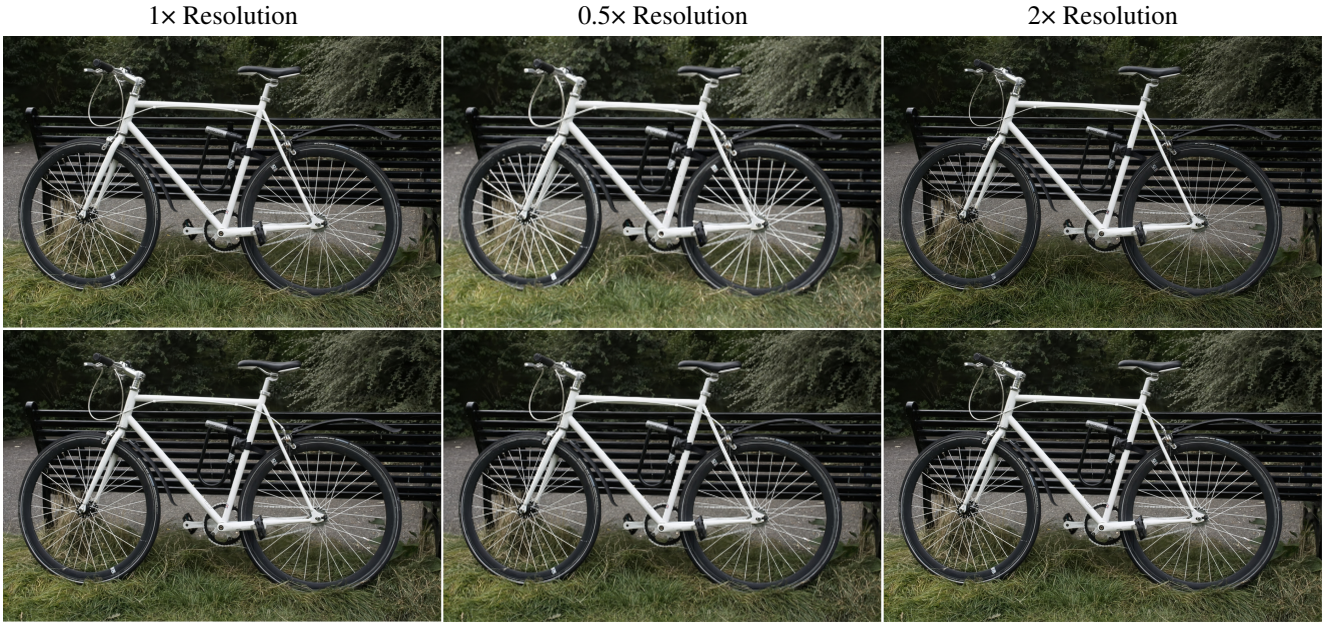

Figure 3. We show renderings of two of our models. The first one (top) was trained without anti-aliasing techniques, the second one (bottom) has them enabled. It is clearly visible that rendering at a different resolution compared to the one used during training (1×) leads to aliasing artifacts (top). The implemented anti-aliasing techniques significantly reduce these artifacts leading to higher visual fidelity (bottom).

Table 7. Quantitative comparisons of approaches for anti-aliasing based on Mip-Splatting on the nine scenes from the Mip-NeRF360 dataset [2]. All approaches optimize to the same quality, but the anti-aliased version of *Faster-GS* trains much faster and requires less VRAM compared to the original implementation [98]. We highlight the significant speedup from our simplified 3D smoothing filter as well as the a more consistent number of Gaussians due to our revised backward pass for the 2D Mip filter.

	PSNR$^{\uparrow}$	Train$^{\downarrow}$	VRAM$^{\downarrow}$	#Gs$^{\downarrow}$
Mip-Splatting [98]	27.53	19m56s	8.0GiB	2.82M
Ours	27.56	4m31s	6.1GiB	2.73M
+ original 3D filter	27.53	5m58s	6.3GiB	2.71M
+ our 3D filter	27.55	4m32s	6.1GiB	2.72M
+ 2D Mip filter	27.54	4m30s	6.2GiB	2.78M
+ full anti-aliasing	27.54	4m31s	6.1GiB	2.70M

Table 8. Single-scale training and multi-scale evaluation on the nine scenes from the Mip-NeRF360 dataset [2]. For both approaches, training is done at the default scene resolution, *i.e.*, with 4×/2× downsampling for outdoor/indoor scenes respectively. We then evaluate the model at the training resolution (1×) as well as at half (0.5×) and double (2×) that resolution for each scene. The results confirm the effectiveness of the anti-aliased version of *Faster-GS*.

	PSNR$^{\uparrow}$		
	1× Res.	0.5× Res.	2× Res.
Ours	27.56	25.11	25.73
+ full anti-aliasing	27.54	28.39	26.87

To complement the 3D smoothing filter, we also integrate the 2D Mip filter into *Faster-GS*. For optimal performance, we make sure to use the smaller opacity values resulting from Eq. (11) to when computing opacity-aware bounding boxes (see Sec. 3.2). We also find that the derivatives of Eq. (11) w.r.t. Σ_{2D} can be numerically unstable causing gradients to explode. When investigating this, we found that the original implementation [98] frequently clips extreme values, while re-implementations [56, 65] compute gradients in a way that does not match the analytical derivative and it is unclear whether this is done on purpose. We find that a much more effective and practical approach to this issue is to simply detach the compensation factor Eq. (11) from the gradient computation for Σ_{2D}. Note that we still provide a reasonably stable implementation for the full analytical derivative that can optionally be enabled.

In Tab. 7, we show results for a single-scale training and same-scale evaluation experiment on the nine scenes from the Mip-NeRF360 dataset [2]. Our simplifications for the 3D smoothing filter effectively eliminate any training overhead compared to the original implementation [98]. We also find that our changes to the backward pass of the 2D Mip filter result in more stable optimization behavior as indicated by the average number of Gaussians that is more similar to the baseline. Specifically, in the original implementation as well as in *Faster-GS* with only the 2D Mip filter enabled, we find that the optimization sometimes creates additional tiny and elongated, *i.e.*, degenerate Gaussians, which slightly re-

duce overall quality of the reconstruction. Most importantly, however, we find that our optimized implementation of the two extensions from Yu *et al*. [98] enables fully anti-aliased training and rendering inside our framework. Note that we adjusted the official Mip-Splatting implementation to use the fused SSIM implementation from Taming-3DGS [56] and the updated opacity learning rate for fair comparison (*cf*. Sec. 4.1). We further validate the effectiveness of the implemented anti-aliasing approach in Tab. 8 and Fig. 3.

C.3. Fast MCMC Densification

In their work 3DGS-MCMC, Kheradmand *et al*. [39] propose an alternative approach for densification that treats the set of 3D Gaussians as Markov Chain Monte Carlo (MCMC) samples. Based on Stochastic Gradient Langevin Dynamics (SGLD) updates, they add noise to the Gaussian means after each training iteration. The splitting, cloning, and pruning steps used in adaptive density control [37] are replaced by a re-localization scheme that tries to preserves sample probability. Gaussian are selected for re-localization when they can no longer meaningfully contribute to renderings, *i.e*., when they are small or have low opacity. To encourage optimal distribution of a preset number of Gaussians, Kheradmand *et al*. also add regularization terms to the loss function.

This approach to 3DGS densification has three main advantages: it reduces reliance on the initial point cloud, allows for specifying the number of Gaussians prior to optimization, and it leads to improved rendering quality. These advantages motivate us to integrate an optimized version into our *Faster-GS* framework. Specifically, we fuse the noise injection step into a single CUDA kernel, as we found that it is a main bottleneck w.r.t. training time. In Tab. 9, we compare our optimized version with the original implementation [39] on the nine scenes from the Mip-NeRF360 dataset [2]. Note that we adjusted the original implementation of 3DGS-MCMC [39] to use the fused SSIM implementation from Taming-3DGS [56] for fair comparison.

For the sake of reproducibility, we provide the target number of Gaussians for the nine scenes in alphabetical order: 6131954, 1244819, 1222956, 3636448, 5834784, 1852335, 1593376, 4961797, 3783761 (values taken from Steiner *et al*. [74]).

Table 9. Quantitative comparison of our and the original implementation [39] for MCMC densification on the nine scenes from the Mip-NeRF360 dataset [2]. While both versions achieve similar quality, our optimized implementation is significantly faster and uses less VRAM during training.

	PSNR$^{\uparrow}$	Train$^{\downarrow}$	VRAM$^{\downarrow}$
3DGS-MCMC [39]	27.83	27m22s	8.9GiB
Ours	28.00	6m17s	7.2GiB

C.4. About Gaussian Truncation and Opacity

In this subsection, we will investigate the effects of truncating Gaussians at different standard deviations as well as an alternative interpretation of the Gaussian opacities in 3DGS.

By definition, Gaussians have infinite support, *i.e*., have a non-zero value at any query point. For rendering a Gaussian, however, the near-zero values obtained when querying it far away from its mean can safely be skipped. Therefore, the original 3DGS approach [37] truncates the projected 2D Gaussians at roughly 3.33σ. Importantly, Kerbl *et al*. do not solely truncate based on the standard deviation of the 2D Gaussian as defined by Σ_{2D} (see Eq. (2)) but also factor in opacity. In their implementation, they achieve this by skipping all fragments during blending where the computed transparency value α (see Eq. (3)) is below a threshold $\tau_\alpha = 1/255$. However, because computing α involves the opacity, a Gaussian with an opacity below τ_α can never be visible or receive gradients in the original implementation. Therefore, the clipping value of $1/100$ used by the opacity reset enforces an upper bound for τ_α. A larger threshold would result in all fragments being discarded after the first reset. This limit results in truncation at $\sqrt{-2\ln(1/100)} \approx 3.03$ standard deviations for Gaussians that have an opacity close to one. In other words, the approach for implicit Gaussian truncation based on fragment α used in the original 3DGS implementation prevents consistent, opacity-independent truncation at fewer than 3.03 standard deviations during training.

We propose a modification that can avoid this issue. Our idea is to check whether the response of a Gaussian at a given pixel is below τ_α before multiplying by the opacity. This then allows for truncation at fewer standard deviations and also addresses an issue w.r.t. how 3DGS computes opacity gradients in the backward pass that was recently brought

Table 10. Quantitative comparisons of truncating Gaussians at different standard deviations on the nine scenes from the Mip-NeRF360 dataset [2]. We find that densification creates fewer Gaussians when truncating at $1/2\sigma$ leading to reduced quality. While our modification for opacity-independent truncation slows down training, we think that the increased flexibility provides an interesting avenue for future work. We highlight that the most aggressive truncation (1σ) makes it so the minimum opacity of a contributing fragment is ≈ 0.61, which means Gaussians are close to opaque. All configurations use our full anti-aliasing (see Tab. 7).

	PSNR$^{\uparrow}$	Train$^{\downarrow}$	VRAM$^{\downarrow}$	#Gs$^{\downarrow}$
3.33σ (default)	27.54	4m31s	6.1GiB	2.70M
4σ	27.54	5m05s	6.4GiB	2.78M
4σ w/ modification	27.65	5m27s	6.6GiB	2.78M
3.33σ w/ modification	27.57	5m07s	6.4GiB	2.79M
3σ w/ modification	27.62	4m55s	6.3GiB	2.78M
2σ w/ modification	27.23	3m41s	5.5GiB	2.22M
1σ w/ modification	25.86	2m26s	4.5GiB	1.39M

up by Hahlbohm *et al*. [23]: The analytical derivatives of Eqs. (3) and (4) provide a non-zero gradient for the opacity of a Gaussian even when its value is zero. In the original implementation, Kerbl *et al*. disregard these gradients as they interpret the opacity as part of the Gaussian response. While this is a reasonable approach that clearly works well in practice, we still think it is worth investigating. We also think that more aggressive truncation that uses, *e.g*., 2σ could be an interesting avenue for future work that renders splats as opaque 2D ellipses to avoid the need for depth-ordered alpha blending.

We show results for different truncation configurations in Tab. 10. Note that all version use our full anti-aliasing as we find that it integrates particularly well with the opacity-independent approach for truncation.

C.5. Additional 4D Reconstruction Results

We also tested our extension to 4D Gaussians on three real scenes from the multi-view dataset by Li *et al*. [47]. We preprocess scenes similar to Yang *et al*. to obtain an initial point cloud and use a consistent hyperparameter configuration for their baseline [93] and our implementation: Scenes are trained and evaluated at 1352×1014 using the provided train/test splits, models use the standard view-dependent color parametrization from 3DGS (SH up to degree three), and training is done for 30000 iterations with a batch size of four. Results are shown in Tab. 11.

Table 11. Comparison with the reference implementation [93] for our extension to 4D Gaussians on three scenes (*Coffee Martini*, *Cook Spinach*, and *Flame Steak*) from the neural 3D video dataset [47].

	PSNR$^{\uparrow}$	Train$^{\downarrow}$	VRAM$^{\downarrow}$
Yang *et al*. [93]	30.13	96m43s	8.7GiB
Ours	30.54	18m57s	3.7GiB