# Faster-GS: Analyzing and Improving Gaussian Splatting Optimization

Supplementary Material

## Contents



## 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*. [6]. 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*. [6] 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 $\mu_{2D}$ 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 $2\text{e}{-4}$, 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 [8] with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1\text{e}{-15}$. 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*. [6] 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*. [11]. 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 $0^{\text{th}}$ 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*. [6], 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 [5]. A key advantage of it is that our *Faster-GS* implementation retains modularity, simplifying future integration into other codebases. Similar to Kerbl *et al*. [6], 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, *i.e*., once the transmittance falls below a threshold $\tau = 1\mathrm{e}{-4}$. Instead of blending a fragment and then checking whether to stop afterward, they check whether blending it would reduce the transmittance below $\tau$ and terminate beforehand if so. 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 $\|q\| < 1\mathrm{e}{-4}$ or $|\Sigma_{2\mathrm{D}}| < 1\mathrm{e}{-6}$. 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 $\mu_{2\mathrm{D}}$ 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 our implementation performs 145 additional optimizer 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*. [13] 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

Table 6. Inference frame rates for all 13 scenes from the Mip-NeRF360, Tanks and Temples, and Deep Blending datasets [1, 4, 9]. 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*. [3] 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 [6] | 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× |

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*. [13], 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*. [11]. 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 [11].

In addition to integrating our changes with respect to early stopping and numerical stability (see Sec. B.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.

### 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 [16], 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 $k^{\text{th}}$ 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 $n^{\text{th}}$ training image, $N$ is the number of training images, $f_n$ is the focal length of the $n^{\text{th}}$ training view in pixel units, and $d_n$ is the $z$-depth of $\mu_k$ for the respective view. As $\hat{\nu}_k$ changes whenever $\mu_k$ 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 $\kappa_{3D}$ 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*. [14]), 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*. [3].

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 [6] 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 $\kappa_{3D}$ (0.3 by default):

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

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).

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. $\Sigma_{2D}$ can be numerically unstable causing gradients to explode. When investigating this, we found that the original implementation [16] frequently clips extreme values, while re-implementations [11, 12] 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 $\Sigma_{2D}$. 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 [1]. Our simplifications for the 3D smoothing filter effectively eliminate any training overhead compared to the original implementation [16]. We also find that our changes to the backward pass of the 2D Mip filter

Table 7. Quantitative comparisons of approaches for anti-aliasing based on Mip-Splatting on the nine scenes from the Mip-NeRF360 dataset [1]. 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 [16]. 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 [16] | 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 |

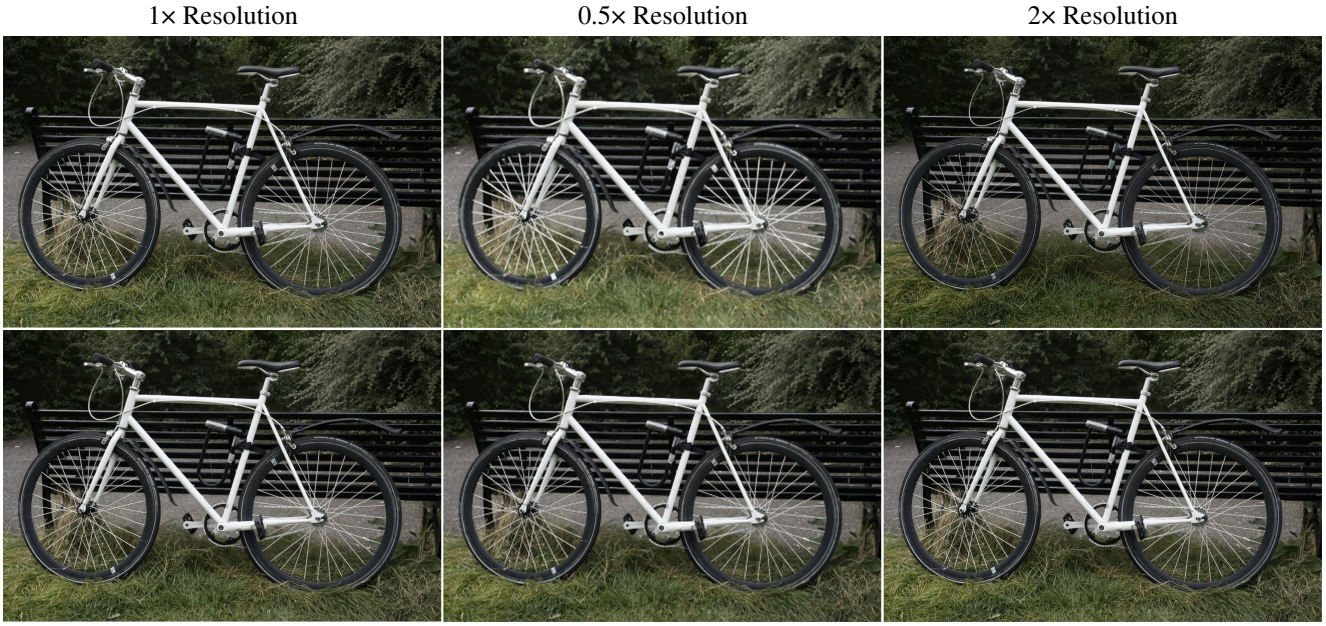


Figure 3. We show renderings of two of our models. The first one (top) was trained without anti-aliasing techniques, the second one (bottom) has them enabled. It is clearly visible that rendering at a different resolution compared to the one used during training (1×) leads to aliasing artifacts (top).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

Table 8. Single-scale training and multi-scale evaluation on the nine scenes from the Mip-NeRF360 dataset [1]. 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↑ | | |
|---|---|---|---|
| | 1× Res. | 0.5× Res. | 2× Res. |
| Ours | 27.56 | 25.11 | 25.73 |
| + full anti-aliasing | 27.54 | 28.39 | 26.87 |

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 reduce overall quality of the reconstruction. Most importantly, however, we find that our optimized implementation of the two extensions from Yu *et al*. [16] 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 [11] 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*. [7] 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 [6] 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 [7] on the nine scenes from the Mip-NeRF360 dataset [1]. Note that we adjusted the original implementation of 3DGS-MCMC [7] to use the fused SSIM implementation from Taming-3DGS [11] for fair comparison.

For the sake of reproducibility, we provide the target

Table 9. Quantitative comparison of our and the original implementation [7] for MCMC densification on the nine scenes from the Mip-NeRF360 dataset [1]. While both versions achieve similar quality, our optimized implementation is significantly faster and uses less VRAM during training.

| | PSNR↑ | Train↓ | VRAM↓ |
|---|---|---|---|
| 3DGS-MCMC [7] | 27.83 | 27m22s | 8.9GiB |
| Ours | 28.00 | 6m17s | 7.2GiB |

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*. [14]).

## 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 [6] truncates the projected 2D Gaussians at roughly $3.33\sigma$. Importantly, Kerbl *et al*. do not solely truncate based on the standard deviation of the 2D Gaussian as defined by $\Sigma_{2D}$ (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 $\alpha$ (see Eq. (3)) is below a threshold $\tau_\alpha = 1/255$. However, because computing $\alpha$ involves the opacity, a Gaussian with an opacity below $\tau_\alpha$ 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 $\tau_\alpha$. 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 $\alpha$ 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 $\tau_\alpha$ 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 up by Hahlbohm *et al*. [2]: 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.

Table 10. Quantitative comparisons of truncating Gaussians at different standard deviations on the nine scenes from the Mip-NeRF360 dataset [1]. 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 ($1\sigma$) makes it so the minimum opacity of a contributing fragment is $\approx 0.61$, which means Gaussians are almost opaque. All configurations use our full anti-aliasing (see Tab. 7).

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

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*., $2\sigma$ 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*. [10]. We preprocess scenes similar to Yang *et al*. to obtain an initial point cloud and use a consistent hyperparameter configuration for their baseline [15] 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 [15] for our extension to 4D Gaussians on three scenes (*Coffee Martini*, *Cook Spinach*, and *Flame Steak*) from the neural 3D video dataset [10].

| | PSNR↑ | Train↓ | VRAM↓ |
|---|---|---|---|
| Yang *et al*. [15] | 30.13 | 96m43s | 8.7GiB |
| Ours | 30.54 | 18m57s | 3.7GiB |

## References


[1] 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. 3, 4, 5, 6

[2] 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 *Vis. Model. Vis.*, 2025. 6

[3] 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. *Comput. Graph. Forum*, 44(2):e70014, 2025. 3, 4

[4] 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): 257:1–257:15, 2018. 3

[5] Moritz Kappel, Florian Hahlbohm, and Timon Scholz. NeRFICG, 2026. 1

[6] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkuehler, and George Drettakis. 3D Gaussian splatting for real-time radiance field rendering. *ACM TOG*, 42(4):139:1–139:14, 2023. 1, 3, 4, 5, 6

[7] 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. 5, 6

[8] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015. 1

[9] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and Temples: Benchmarking large-scale scene reconstruction. *ACM TOG*, 36(4):78:1–78:13, 2017. 3

[10] 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. 6

[11] 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*, pages 2:1–2:11, 2024. 1, 3, 4, 5

[12] 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):64:1–64:17, 2024. 4

[13] Markus Schütz, Christoph Peters, Florian Hahlbohm, Elmar Eisemann, Marcus Magnor, and Michael Wimmer. Splatshop: Efficiently editing large Gaussian splat models. *Comput. Graph. Forum*, 2025. 2, 3

[14] 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. 4, 6

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

[16] Zehao Yu, Anpei Chen, Binbin Huang, Torsten Sattler, and Andreas Geiger. Mip-Splatting: Alias-free 3D Gaussian splatting. In *CVPR*, pages 19447–19456, 2024. 3, 4, 5