3DGS-LM: Faster Gaussian-Splatting Optimization with Levenberg-Marquardt

Lukas Höllein[1] Aljaž Božič[2] Michael Zollhöfer[2] Matthias Nießner[1]
[1]Technical University of Munich [2]Meta
https://lukashoel.github.io/3DGS-LM/

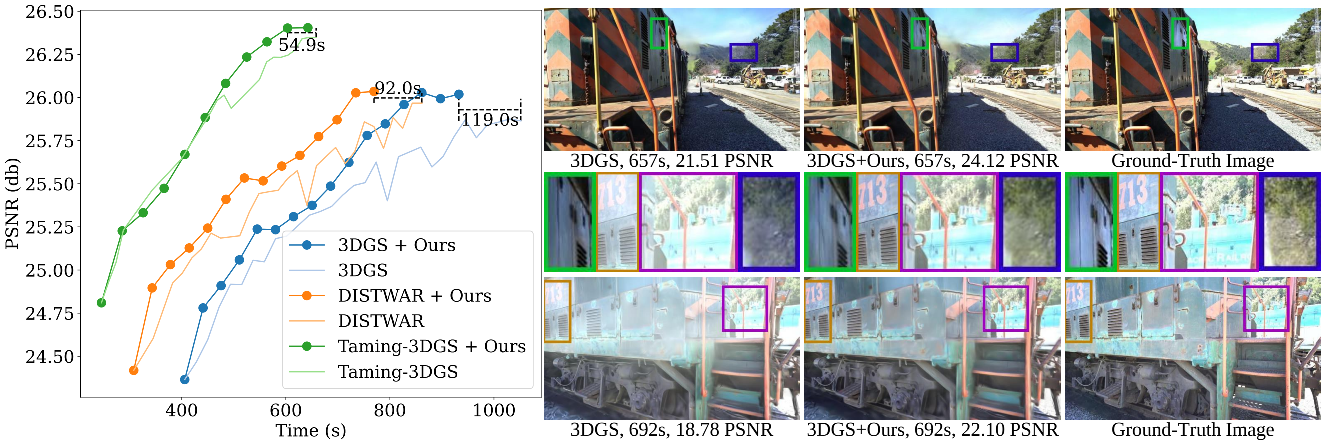

Figure 1. Our method accelerates 3D Gaussian Splatting (3DGS) [20] reconstruction by replacing the ADAM optimizer with a tailored Levenberg-Marquardt. Left: starting from the same initialization, our method converges 30% faster on the Tanks&Temples "Train" scene. Right: after the same amount of time, our method produces higher quality renderings (e.g., better brightness and contrast).

Abstract

We present 3DGS-LM, a new method that accelerates the reconstruction of 3D Gaussian Splatting (3DGS) by replacing its ADAM optimizer with a tailored Levenberg-Marquardt (LM). Existing methods reduce the optimization time by decreasing the number of Gaussians or by improving the implementation of the differentiable rasterizer. However, they still rely on the ADAM optimizer to fit Gaussian parameters of a scene in thousands of iterations, which can take up to an hour. To this end, we change the optimizer to LM that runs in conjunction with the 3DGS differentiable rasterizer. For efficient GPU parallelization, we propose a caching data structure for intermediate gradients that allows us to efficiently calculate Jacobian-vector products in custom CUDA kernels. In every LM iteration, we calculate update directions from multiple image subsets using these kernels and combine them in a weighted mean. Overall, our method is 30% faster than the original 3DGS while obtaining the same reconstruction quality. Our optimization is also agnostic to other methods that accelerate 3DGS, thus enabling even faster speedups compared to vanilla 3DGS.

1. Introduction

Novel View Synthesis (NVS) is the task of rendering a scene from new viewpoints, given a set of images as input. NVS can be employed in Virtual Reality applications to achieve photo-realistic immersion and to freely explore captured scenes. To facilitate this, different 3D scene representations have been developed [2, 3, 20, 28, 30, 36]. Among those, 3DGS [20] (3D Gaussian-Splatting) is a point-based representation that parameterizes the scene as a set of 3D Gaussians. It offers real-time rendering and high-quality image synthesis, while being optimized from a set of posed images through a differentiable rasterizer.

3DGS is optimized from a set of posed input images that densely capture the scene. The optimization can take up to an hour to converge on high-resolution real-world scene datasets with a lot of images [42]. It is desirable to reduce the optimization runtime which enables faster usage of the reconstruction for downstream applications. Existing methods reduce this runtime by improving the optimization along different axes. First, 3DGS renders images with a tile-based, differentiable rasterizer that is implemented in CUDA. In every iteration, the Gaussians get updated with gradient descent by backpropagating a rendering loss through the rasterizer. By improving the speed of the forward- and backward-pass, recent methods accelerate the optimization [11, 14, 27, 41]. Second, in 3DGS the number of Gaussians is gradually grown during optimization, which is known as densification. To facilitate this, 3DGS accumulates positional gradients over multiple iterations and uses these statistics to split and prune Gaussians. Recently, GS-MCMC [22], Taming-3DGS [27], Mini-Splatting [13], and Revising-3DGS [5] propose novel densification schemes that reduce the number of required Gaussians to represent the scene. This makes the optimization more stable and also faster, since fewer Gaussians must be optimized.

Despite these improvements, the optimization still takes significant resources, requiring thousands of gradient descent iterations to converge. To this end, we aim to reduce the runtime by improving the underlying optimization during 3DGS reconstruction. More specifically, we propose to replace the widely used ADAM [23] optimizer with a tailored Levenberg-Marquardt (LM) [29]. This allows us to accelerate 3DGS reconstruction (Fig. 1 left) by over 30% on average. Concretely, we propose a highly-efficient GPU parallelization scheme for the preconditioned conjugate gradient (PCG) algorithm within the inner LM loop in order to obtain the respective update directions. To this end, we extend the differentiable 3DGS rasterizer with custom CUDA kernels that compute Jacobian-vector products. Our proposed caching data structure for intermediate gradients (Fig. 3) then allows us to perform these calculations fast and efficiently in a data-parallel fashion. In order to scale caching to high-resolution image datasets, we calculate update directions from multiple image subsets and combine them in a weighted mean. Overall, this allows us to improve reconstruction time by 30% compared to state-of-the-art 3DGS baselines while achieving the same reconstruction quality (Fig. 1 right).

To summarize, our contributions are:

- we propose a tailored 3DGS optimization based on Levenberg-Marquardt that improves reconstruction time by 30% and which is agnostic to other 3DGS acceleration methods.
- we propose a highly-efficient GPU parallelization scheme for the PCG algorithm for 3DGS in custom CUDA kernels with a caching data structure to facilitate efficient Jacobian-vector products.

2. Related Work

2.1. Novel-View-Synthesis

Novel-View-Synthesis is widely explored in recent years [2, 3, 18, 20, 28, 30, 36]. Neural Radiance Fields (NeRF) [28] are particularly successful, because they achieve highly photo-realistic image synthesis results by optimizing MLP weights through volume rendering. To improve training time from days to minutes, NeRF was combined with explicit representations like voxel grids [15, 35], hash grids [30], or points [40].

3D Gaussian Splatting (3DGS) [20] extends this idea by representing the scene as a set of 3D Gaussians, that are rasterized into 2D splats and then α-blended into pixel colors. The approach gained popularity, due to the real-time rendering capabilities of high quality images. Since its inception, 3DGS was improved along several axes (next to improving optimization runtime). Recent methods improve the quality of rendered images [17, 26, 43] and the efficiency during rendering [31, 34]. Others obtain better surface reconstructions [16, 19], reduce the memory requirements of the Gaussians [32], and enable training and rendering of large-scale scenes [21, 46]. We similarly adopt 3DGS as our scene representation, but focus on improving the per-scene optimization time.

2.2. Speed-Up Gaussian Splatting Optimization

Obtaining a 3DGS scene reconstruction can be accelerated in several ways. One line of work reduces the number of Gaussians by changing the densification heuristics [5, 13, 22, 26, 27]. Recent methods focusing on sparse-view reconstruction train a neural network as data-driven prior, that directly outputs Gaussians in a single forward pass [6–8, 12, 25]. In contrast, we focus on the dense-view and per-scene optimization setting, i.e., we do not require a data prior. Other works improve the optimization runtime by improving the implementation of the underlying differentiable rasterizer [11, 14, 27, 41]. We demonstrate that our method is compatible with these approaches, i.e., our optimizer can be plugged into these methods to even further accelerate the optimization.

2.3. Gauss-Newton Optimization For 3D Reconstruction Tasks

NeRF and 3DGS are typically optimized with stochastic gradient descent (SGD) optimizers like ADAM [23] for thousands of iterations. In contrast, many works in RGB-D fusion employ the Gauss-Newton (or Levenberg-Marquardt) algorithms to optimize objectives for 3D reconstruction tasks [9, 10, 37, 38, 47, 48]. By doing so, these methods can quickly converge in an order of magnitude fewer iterations than SGD. Motivated by this, we aim to accelerate 3DGS optimization by adopting the Levenberg-Marquardt algorithm as our optimizer. Rasmuson *et al.* [33] implemented the Gauss-Newton algorithm for reconstructing a NeRF based on voxel grids. Their technical approach is related to ours, but we implement it for the 3DGS representation in a different way. Concretely, we subsample images in every iteration and introduce a caching data structure. This allows us to achieve state-of-the-art rendering quality, while significantly accelerating the optimization in comparison to existing methods.

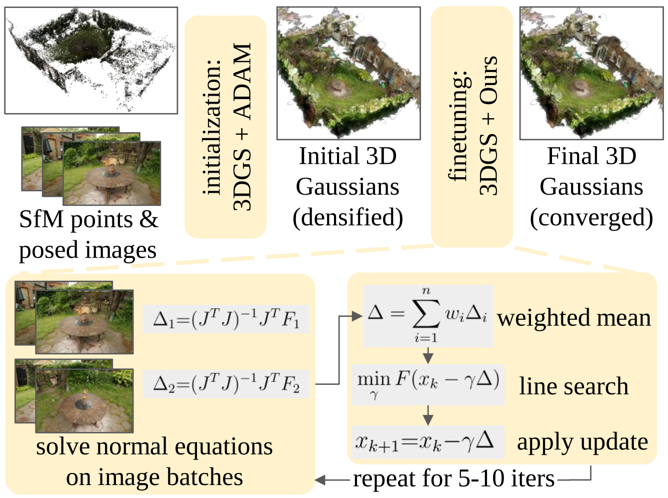

Figure 2. **Method Overview.** We accelerate 3DGS optimization by framing it in two stages. First, we use the original ADAM optimizer and densification scheme to arrive at an initialization for all Gaussians. Second, we employ the Levenberg-Marquardt algorithm to finish optimization.

3. Method

Our pipeline is visualized in Fig. 2. First, we obtain an initialization of the Gaussians from a set of posed images and their SfM point cloud as input by running the standard 3DGS optimization (Sec. 3.1). In this stage the Gaussians are densified, but remain unconverged. Afterwards, we finish the optimization with our novel optimizer. Concretely, we optimize the sum of squares objective with the Levenberg-Marquardt (LM) [29] algorithm (Sec. 3.2), which we implement in efficient CUDA kernels (Sec. 3.3). This two-stage approach accelerates the optimization compared to only using first-order optimizers.

3.1. Review of Gaussian-Splatting

3D Gaussian Splatting (3DGS) [20] models a scene as a set of 3D Gaussians, each of which is parameterized by a position, rotation, scaling, and opacity. The view-dependent color is modeled by Spherical Harmonics coefficients of order 3. To render an image of the scene from a specific viewpoint, all visible Gaussians are first projected into 2D splats with a tile-based differentiable rasterizer. Afterwards, the splats are α-blended per-pixel to obtain a final pixel color. To fit the Gaussians to image observations, a rendering loss $\mathcal{L}$ is optimized with the ADAM [23] optimizer w.r.t. all Gaussian parameters $\mathbf{x}$:

$$\mathcal{L}(\mathbf{x})=\frac{1}{N}\sum_{i=1}^{N}(\lambda_1\mathcal{L}_1(\hat{c}_i,c_i)+\lambda_2(1-\mathcal{L}_{SSIM}(\hat{c}_i,c_i))) \quad (1)$$

where λ_1=0.2, λ_2=0.8, $\hat{c}_i$ is the rendered color, and c_i the ground-truth color for one color channel of one pixel. Typically, ADAM uses a batch size of 1, meaning a random image per iteration is sampled to perform an update step. The number of Gaussians are initialized from the SfM points and is gradually grown during the first half of the optimization, which is known as densification [20].

3.2. Levenberg-Marquardt Optimization for 3DGS

We employ the LM algorithm for optimization of the Gaussians by reformulating the rendering loss as a sum of squares energy function:

$$E(\mathbf{x})=\sum_{i=1}^{N}\left(\sqrt{\lambda_1\mathcal{L}_1(\hat{c}_i,c_i)}^2+\sqrt{\lambda_2(1-\mathcal{L}_{SSIM}(\hat{c}_i,c_i))}^2\right) \quad (2)$$

where we have in total N=$6HWP$ residuals for P images with HxW pixels each and 3 color channels. We take the square root of the $\mathcal{L}_1$ and $\mathcal{L}_{SSIM}$ losses, to convert Eq. (1) into the required form for the LM algorithm. In other words, we use the identical objective, but a different optimizer. In contrast to ADAM, the LM algorithm requires a large batch size (ideally all images) for every update step to achieve stable convergence [29]. In practice, we select large enough subsets of all images to ensure reliable update steps (see Sec. 3.3.2 for more details).

3DGS uses the structural similarity index measure (SSIM) as loss function during optimization. In SSIM, the local neighborhood of every pixel gets convolved with Gaussian kernels to obtain the final per-pixel score [39]. As a consequence, gradients from the loss flow from every pixel to all Gaussians in the local neighborhood of that pixel. In contrast, the $\mathcal{L}_1$ loss only provides gradients from every pixel to the Gaussians along the corresponding ray. For simplicity, we approximate the gradient flow in $\mathcal{L}_{SSIM}$ by backpropagating the per-pixel scores only to the center pixels (ignoring the contribution to other pixels in the local neighborhood). We implement it following the derivation of Zhao *et al.* [45]. This approximation allows us to keep rays independent from each other when calculating the Jacobian, which is a desirable property for our CUDA kernel implementation (see Sec. 3.3 for more details).

Obtaining Update Directions In every iteration of our optimization we obtain the update direction $\Delta \in \mathbb{R}^M$ for all M Gaussian parameters by solving the normal equations:

$$(\mathbf{J}^T\mathbf{J} + \lambda_{reg} diag(\mathbf{J}^T\mathbf{J}))\Delta = \mathbf{J}^T\mathbf{F}(\mathbf{x}) \quad (3)$$

where $\mathbf{F}(\mathbf{x}) \in \mathbb{R}^N$ is the residual vector corresponding to Eq. (2) and $\mathbf{J} \in \mathbb{R}^{NxM}$ the corresponding Jacobian matrix.

In a typical dense capture setup, we optimize over millions of Gaussians and have hundreds of high-resolution images. Even though $\mathbf{J}$ is a sparse matrix (each row only contains non-zero values for the Gaussians that contribute to the color of that pixel), it is therefore not possible to materialize $\mathbf{J}$ in memory. Instead, we employ the preconditioned conjugate gradient (PCG) algorithm, to solve Eq. (3) in a *matrix-free* fashion. We implement PCG in custom CUDA kernels, see Sec. 3.3 for more details.

Apply Parameter Update After we obtained the solution Δ, we run a line search to find the best scaling factor γ for updating the Gaussian parameters:

$$\min_{\gamma} E(\mathbf{x}_k - \gamma\Delta) \tag{4}$$

In practice, we run the line search on a 30% subset of all images, which is enough to get a reasonable estimate for γ, but requires fewer rendering passes. Afterwards, we update the Gaussian parameters as: $\mathbf{x}_{k+1}{=}\mathbf{x}_k - \gamma\Delta$. Similar to the implementation of LM in CERES [1], we adjust the regularization strength λ_{reg} after every iteration based on the quality of the update step. Concretely, we calculate

$$\rho = \frac{||F(\mathbf{x}+\Delta)||^2 - ||F(\mathbf{x})||^2}{||\mathbf{J}\Delta + F(\mathbf{x})||^2 - ||F(\mathbf{x})||^2} \tag{5}$$

and only keep the update if $\rho > 1e{-}5$, in which case we reduce the regularization strength λ_{reg} by half. Otherwise we revert the update and double λ_{reg}.

3.3. Memory Efficient PCG Implementation

The PCG algorithm obtains the solution to the least squares problem of Eq. (3) in multiple iterations. In practice, we run the algorithm for $n_{iters}{=}8$ iterations and implement it with custom CUDA kernels. We summarize it in Algorithm 1.

Most of the work in every PCG iteration is consumed by calculating the matrix-vector product $\mathbf{g}_i{=}\mathbf{J}^T\mathbf{J}\mathbf{p}_i$. We implement this product in two stages by first calculating $\mathbf{u}_i{=}\mathbf{J}\mathbf{p}_i$ and then $\mathbf{g}_i{=}\mathbf{J}^T\mathbf{u}_i$. Calculating the non-zero values of $\mathbf{J}$ requires backpropagating from every residual through the α-blending and splat projection steps back to all Gaussian parameters. The original tile-based rasterizer of 3DGS [20] implements the calculation of $\mathbf{J}^T\mathbf{x}$ with a *per-pixel* parallelization. That is, every thread handles one ray, stepping backwards along all splats that this ray hit. We found that this parallelization is too slow for an efficient PCG implementation. The reason is that we have to repeat this calculation multiple times: per PCG iteration we do it once for $\mathbf{u}_i$ and once for $\mathbf{g}_i$. As a consequence, we re-calculate the same intermediate α-blending states, since the loop over splats is evaluated multiple times.

Our key idea is to change the parallelization from *per-pixel* to *per-pixel-per-splat*. We summarize this pattern in Fig. 3. We describe the backward pass from a residual r to a Gaussian parameter x_i as:

$$\frac{\partial r}{\partial x_i} = \frac{\partial r}{\partial p}\frac{\partial p}{\partial s}\frac{\partial s}{\partial x_i} \tag{6}$$

Algorithm 1: We run the PCG algorithm with custom CUDA kernels (blue) in every LM iteration.

```
    Input  : Gaussians and cameras $\mathcal{G}$, Residuals $\mathbf{F}$
    Output: Update Direction $\boldsymbol{\Delta}$
 1  $\mathbf{b}, \mathcal{C}$ = buildCache($\mathcal{G}, \mathbf{F}$)        // $\mathbf{b}{=}-\mathbf{J}^T\mathbf{F}$
 2  $\mathcal{C}$ = sortCacheByGaussians($\mathcal{C}$)
 3  $\mathbf{M}^{-1} = 1/$diagJTJ($\mathcal{G}, \mathcal{C}$)
 4  $\mathbf{x_0} = \mathbf{M}^{-1}\mathbf{b}$
 5  $\mathbf{u}_0$ = applyJ(sortX($\mathbf{x}_0$), $\mathcal{G}, \mathcal{C}$)     // $\mathbf{u}_0{=}\mathbf{J}\mathbf{x}_0$
 6  $\mathbf{g}_0$ = applyJT($\mathbf{u}_0, \mathcal{G}, \mathcal{C}$)          // $\mathbf{g}_0{=}\mathbf{J}^T\mathbf{u}_0$
 7  $\mathbf{r}_0 = \mathbf{b} - \mathbf{g}_0$
 8  $\mathbf{z}_0 = \mathbf{M}^{-1}\mathbf{r}_0$
 9  $\mathbf{p}_0 = \mathbf{z}_0$
10  for $i = 0$ to $n_{iters}$ do
11      $\mathbf{u}_i$ = applyJ(sortX($\mathbf{p}_i$), $\mathcal{G}, \mathcal{C}$)   // $\mathbf{u}_i{=}\mathbf{J}\mathbf{p}_i$
12      $\mathbf{g}_i$ = applyJT($\mathbf{u}_i, \mathcal{G}, \mathcal{C}$)        // $\mathbf{g}_i{=}\mathbf{J}^T\mathbf{u}_i$
13      $\alpha_i = \frac{\mathbf{r}_i^T\mathbf{z}_i}{\mathbf{p}_i^T\mathbf{g}_i}$
14      $\mathbf{x}_{i+1}{=}\mathbf{x}_i{+}\alpha_i\mathbf{p}_i$
15      $\mathbf{r}_{i+1}{=}\mathbf{r}_i{-}\alpha_i g_i$
16      $\mathbf{z}_{i+1}{=}\mathbf{M}^{-1}\mathbf{r}_{i+1}$
17      $\beta_i = \frac{\mathbf{r}_{i+1}^T\mathbf{z}_{i+1}}{\mathbf{r}_i^T\mathbf{z}_i}$
18      $\mathbf{p}_{i+1} = \mathbf{z}_{i+1} + \beta_i\mathbf{p}_i$
19  end for
20  return $\mathbf{x}_{i+1}$
```

where $\frac{\partial r}{\partial p}$ is the gradient from the residual to the pixel, $\frac{\partial p}{\partial s}$ from the pixel to the projected splat, and $\frac{\partial s}{\partial x_i}$ from the splat to the Gaussian parameter. Calculating $\frac{\partial p}{\partial s}$, requires access to the accumulated transmittance and color in the α-blending process for that pixel until splat s. Instead of looping over all splats along a ray and recalculating $\frac{\partial p}{\partial s}$ every time, we cache this gradient once (Fig. 3 left). Every time we need to calculate $\mathbf{u}_i$ or $\mathbf{g}_i$ in the PCG algorithm, we then read the corresponding gradients from the cache (Fig. 3 right). This allows us to directly parallelize over all splats in all pixels, which drastically accelerates the runtime since we no longer have to loop over rays.

Storing these gradients in a cache consumes additional GPU memory and is largely controlled by how many images (rays) we process in each PCG iteration and how many splats contribute to the final color along each ray. We propose an efficient subsampling scheme in Sec. 3.3.2 to limit the memory to the available budget.

3.3.1 CUDA Kernel Design

We describe the parallelization pattern and output of every CUDA kernel that we use to implement Algorithm 1. We refer to the supplemental material for more details.

`buildCache` We use the *per-pixel* parallelization to cal-

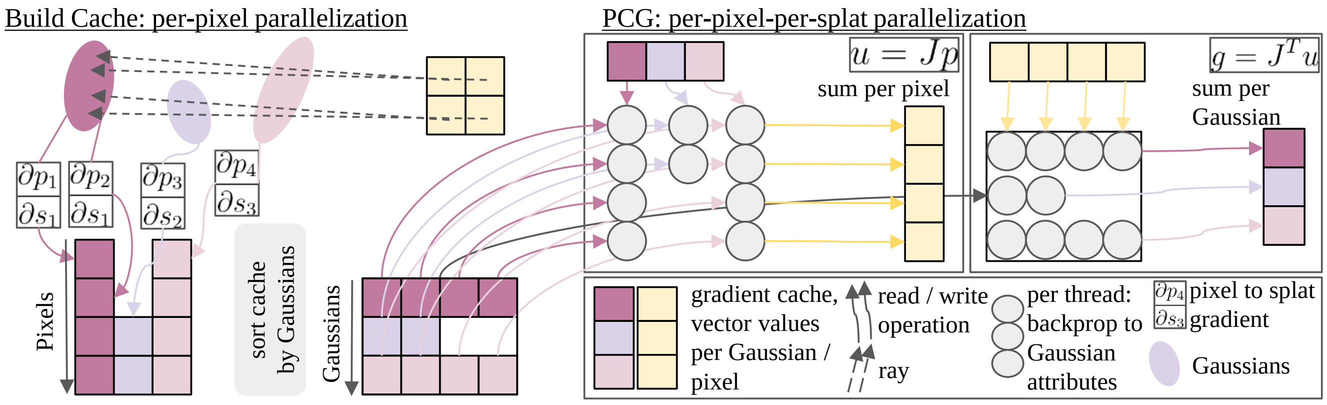
Build Cache: per-pixel parallelization
PCG: per-pixel-per-splat parallelization
∂p_1 ∂s_1
∂p_2 ∂s_1
∂p_3 ∂s_2
∂p_4 ∂s_3
Pixels
sort cache by Gaussians
Gaussians
$u = Jp$
sum per pixel
$g = J^T u$
sum per Gaussian
gradient cache, vector values per Gaussian / pixel
read / write operation
ray
per thread: backprop to Gaussian attributes
pixel to splat gradient
Gaussians

Figure 3. **Parallelization Strategy And Caching Scheme.** We implement the PCG algorithm with efficient CUDA kernels, that use a gradient cache to calculate Jacobian-vector products. Before PCG starts, we create the gradient cache following the *per-pixel* parallelization of 3DGS [20]. Afterwards, we sort the cache by Gaussians to ensure coalesced read accesses. The cache decouples splats along rays, which allows us to parallelize *per-pixel-per-splat* during PCG.

culate all gradients $\frac{\partial p}{\partial s}$. For coalesced write accesses, we store the cache values sorted by pixels (Fig. 3 left). Additionally, we calculate $\mathbf{b} = -\mathbf{J}^T\mathbf{F}$ following the implementation of the differentiable rasterizer in 3DGS [20], by splitting this calculation into three smaller kernels, where we write out the cache completely in the first kernel.

`sortCacheByGaussians` All remaining kernels require the cache to be sorted over Gaussians to ensure coalesced read accesses. We sort the cache by Gaussians and calculate a prefix sum over the number of entries per Gaussian. Subsequent kernels use this to index into the cache.

`diagJTJ` We use the Jacobi preconditioner $\mathbf{M}^{-1}{=}1/\text{diag}(\mathbf{J}^T\mathbf{J})$ when running PCG (Algorithm 1). This kernel computes the required entries by utilizing the *per-pixel-per-splat* parallelization and the cache. Every thread directly squares its calculated gradients and atomically adds them to the output vector.

`applyJ` We compute $\mathbf{u}_i{=}\mathbf{J}\mathbf{p}_i$ by utilizing the *per-pixel-per-splat* parallelization and the cache. Additionally, we resort the input vector $\mathbf{p}_i$ for a coalesced memory access using the `sortX` kernel. Every thread sums up the calculated gradients and atomically adds them to the output vector.

`applyJT` We compute $\mathbf{g}_i{=}\mathbf{J}^T\mathbf{u}_i$ by utilizing the *per-pixel-per-splat* parallelization and the cache. Similar to `buildCache`, we split the computation in three kernels.

3.3.2 Image Subsampling Scheme

Our caching data structure consumes additional GPU memory. For high resolution images in a dense reconstruction setup, the number of rays and thus the cache size can grow too large. To this end, we split the images into batches and solve the normal equations independently, following Eq. (3). This allows us to store the cache only for one batch at a time. Concretely, for n batches, we obtain n update vectors and combine them in a weighted mean:

$$\Delta = \sum_{i=1}^{n} \frac{\mathbf{M}_i \Delta_i}{\sum_{k=1}^{n} \mathbf{M}_k} \tag{7}$$

where we use the inverse of the PCG preconditioner $M_i{=}\text{diag}(\mathbf{J}_i^T\mathbf{J}_i)$ as the weights. We refer to the supplement for a derivation of the weights. These weights balance the importance of update vectors across batches based on how much each Gaussian parameter contributed to the final colors in the respective images. This subsampling scheme allows us to control the cache size relative to the number of images in a batch. In practice, we can choose batch sizes between 25 and 70 images and use up to $n{=}4$ batches per LM iteration. We either select the images at random or, if the scene was captured along a smooth trajectory, in a strided fashion to maximize the scene coverage in every batch.

3.4. 3DGS Optimization in Two Stages

We utilize our LM implementation in the second stage of 3DGS optimization (see Fig. 2). Before that, we use the ADAM optimizer to obtain an initialization of the Gaussian parameters. It is also possible to use the LM optimizer from the beginning, however this does not bring any additional speed-up (see Fig. 4). In the beginning of optimization, gradient descent makes rapid progress by optimizing the Gaussians from a single image per iteration. In contrast, we sample many images in every LM iteration, which makes every iteration more time-consuming. This additional compute overhead is especially helpful to converge to optimal Gaussian parameters quicker (see Fig. 1 left).

Splitting the method in two stages also allows us to complete the densification of the Gaussians before employing the LM optimizer, which simplifies the implementation.

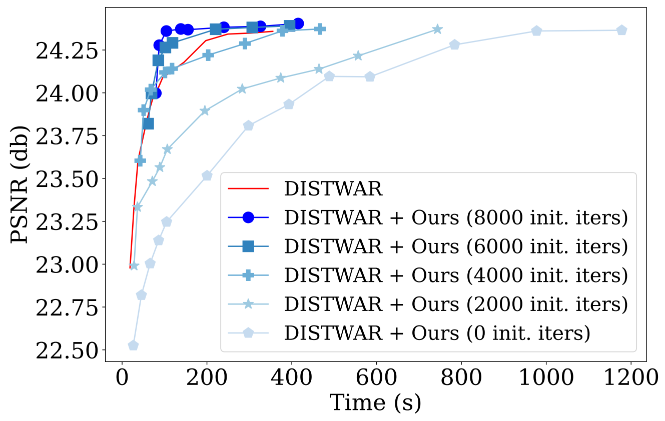

Figure 4. **Comparison of initialization iterations.** In our first stage, we initialize the Gaussians with gradient descent for K iterations, before finetuning with our LM optimizer. After 6000 or 8000 iterations, our method converges faster than the baseline. With less iterations, pure LM is slower, which highlights the importance of our two stage approach. Results reported on the "garden" scene from MipNeRF360 [28] without densification.

4. Results

Baselines We compare our LM optimizer against ADAM in multiple reference implementations of 3DGS. This shows, that our method is compatible with other runtime improvements. In other words, we can swap out the optimizer and retain everything else. Concretely, we compare against the original 3DGS implementation [20], its reimplementation "gsplat" [41], and DISTWAR [11]. Additionally, we compare against Taming-3DGS [27] by utilizing their released source code, that contains the rasterizer improvements and uses the 3DGS [20] densification heuristics (referred to as "Taming-3DGS†" in the following). We run all baselines for 30K iterations with their default hyperparameters.

Datasets and Metrics We benchmark our runtime improvements on three established datasets: Tanks&Temples [24], Deep Blending [18], and MipNeRF360 [4]. These datasets contain in total 13 scenes that cover bounded indoor and unbounded outdoor environments. We fit all scenes for every method on the same NVIDIA A100 GPU using the train/test split as proposed in the original 3DGS [20] publication. To measure the quality of the reconstruction, we report peak signal-to-noise ratio (PSNR), structural similarity (SSIM), and perceptual similarity (LPIPS) [44] averaged over all test images. Additionally, we report the optimization runtime and the maximum amount of consumed GPU memory.

Implementation Details For our main results, we run the first stage for 20K iterations with the default hyperparameters of the respective baseline. The densification is completed after 15K iterations. Afterwards, we only have to run 5 LM iterations with 8 PCG iterations each to converge on all scenes. This showcases the efficiency of our optimizer. Since the image resolutions are different for every dataset, we select the batch-size and number of batches such that the consumed memory for caching is similar. We select 25 images in 4 batches for MipNeRF360 [4], 25 images in 3 batches for Deep Blending [18], and 70 images in 3 batches for Tanks&Temples [24]. We constrain the value range of λ_{reg} for stable updates. We define it in $[1e{-}4, 1e4]$ for Deep Blending [18] and Tanks&Temples [24] and in the interval $[1e{-}4, 1e{-}2]$ for MipNeRF360 [4].

4.1. Comparison to Baselines

We report our main quantitative results in Tab. 1. Our LM optimizer can be added to all baseline implementations and accelerates the optimization runtime by 30% on average. The reconstructions show similar quality across all metrics and datasets, highlighting that our method arrives at similar local minima, just faster. We also provide a per-scene breakdown of these results in the supplemental material. On average our method consumes 53 GB of GPU memory on all datasets. In contrast, the baselines do not use an extra cache and only require between 6-11 GB of memory. This showcases the runtime-memory tradeoff of our approach.

We visualize sample images from the test set in Fig. 5 for both indoor and outdoor scenarios. After the same amount of optimization runtime, our method is already converged whereas the baselines still need to run longer. As a result, the baselines still contain suboptimal Gaussians, which results in visible artifacts in rendered images. In comparison, our rendered images more closely resemble the ground truth with more accurate brightness / contrast and texture details.

4.2. Ablations

Is the L1/SSIM objective important?
We utilize the same loss functions in our LM optimizer as in the original 3DGS implementation, namely the $\mathcal{L}_1$ and $\mathcal{L}_{SSIM}$ losses (see Eq. (1)). Since LM energy terms are defined as a sum of squares, we adopt the square root formulation of these loss functions to arrive at an identical objective (see Eq. (2)). We compare this choice against fitting the Gaussians with only an $\mathcal{L}_2$ loss, that does not require taking a square root. Concretely, we compare the achieved quality and runtime of LM against ADAM for both the $\mathcal{L}_2$ loss and the $\mathcal{L}_1$ and $\mathcal{L}_{SSIM}$ losses. As can be seen in Tab. 2, we achieve faster convergence and similar quality in both cases. However, the achieved quality is inferior for both LM and ADAM when only using the $\mathcal{L}_2$ loss. This highlights the importance of the $\mathcal{L}_1$ and $\mathcal{L}_{SSIM}$ loss functions and why we adopt them in our method as well.

How many images per batch are necessary? The key hyperparameters in our model are the number of images in a batch and how many batches to choose for every LM iteration (Sec. 3.3.2). This directly controls how much GPU memory our optimizer consumes and how much time we

Method	MipNeRF-360 [4]				Tanks&Temples [24]				Deep Blending [18]			
	SSIM↑	PSNR↑	LPIPS↓	Time (s)	SSIM↑	PSNR↑	LPIPS↓	Time (s)	SSIM↑	PSNR↑	LPIPS↓	Time (s)
3DGS [20]	0.813	27.40	0.218	1271	0.844	23.68	0.178	736	0.900	29.51	0.247	1222
+ Ours	0.813	27.39	0.221	**972**	0.845	23.73	0.182	**663**	0.903	29.72	0.247	**951**
DISTWAR [11]	0.813	27.42	0.217	966	0.844	23.67	0.178	601	0.899	29.47	0.247	841
+ Ours	0.814	27.42	0.221	**764**	0.844	23.67	0.183	**537**	0.902	29.60	0.248	**672**
gsplat [41]	0.814	27.42	0.217	1064	0.846	23.50	0.179	646	0.904	29.52	0.247	919
+ Ours	0.814	27.42	0.221	**818**	0.844	23.68	0.183	**414**	0.902	29.58	0.249	**716**
Taming-3DGS† [27]	0.810	27.44	0.224	684	0.847	23.79	0.176	463	0.902	29.71	0.242	590
+ Ours	0.814	27.41	0.221	**589**	0.844	23.72	0.183	**393**	0.902	29.72	0.249	**540**

Table 1. **Quantitative comparison of our method and baselines.** By adding our method to baselines, we accelerate the optimization time by 30% on average while achieving the same quality. We can combine our method with others, that improve runtime along different axes. This demonstrates that our method offers an orthogonal improvement, i.e., the LM optimizer can be plugged into many existing methods.

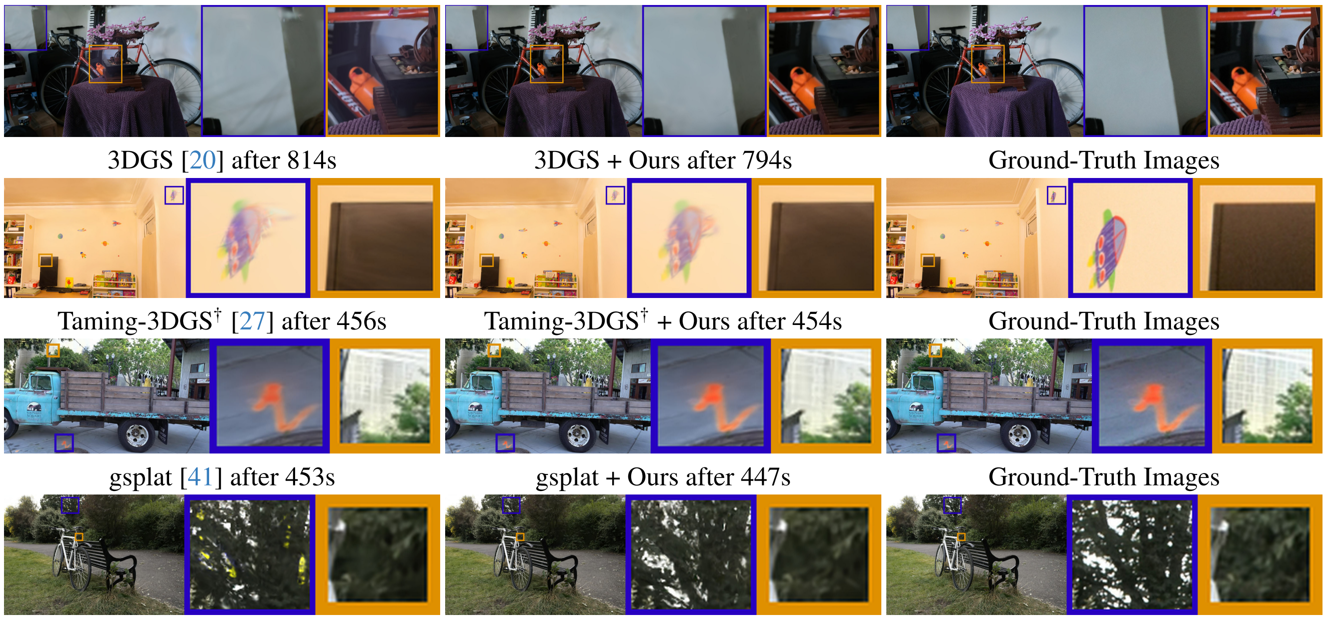

Figure 5. **Qualitative comparison of our method and baselines.** We compare rendered test images after similar optimization time. All baselines converge faster when using our LM optimizer, which shows in images with fewer artifacts and more accurate brightness / contrast.

need per iteration. We compare different number of images in Tab. 3 (top) on the NeRF-Synthetic [28] dataset. We choose this dataset because it is possible to construct the cache for all 100 training images in 33 GB of GPU memory and optimize the scene with one batch of all images. We compare this against our proposed subsampling scheme for different number of images in a single batch. Decreasing the number of images in a batch results in only slightly worse quality, but also yields faster convergence and reduces GPU memory consumption linearly down to 15GB for 40 images. This demonstrates that subsampling images does not negatively impact the convergence of the LM optimizer in our task. This motivates our usage of subsampling on the real-world datasets in our main results.

Are we better than full-batch ADAM? One reason why LM requires drastically fewer iterations to converge than ADAM is the larger batch size (number of training images per iteration). Concretely, we require only 5 additional LM iterations after the initialization, whereas ADAM runs for another 10K iterations to converge. We can similarly increase the batch size in ADAM to reduce the number of iterations. However, as can be seen in Tab. 3 (bottom),

Method	SSIM↑	PSNR↑	LPIPS↓	Time (s)
3DGS [20] (L1, SSIM)	0.862	27.23	**0.108**	1573
3DGS + Ours (L1, SSIM)	**0.863**	**27.29**	0.110	**1175**
3DGS [20] (L2)	0.854	27.31	0.117	1528
3DGS + Ours (L2)	**0.857**	**27.48**	**0.114**	**1131**

Table 2. **Ablation of objective.** We compare using the L1/SSIM losses against the L2 loss. For both, 3DGS [20] optimized with ADAM and combined with ours, we achieve better results with the L1/SSIM objective. In both cases, our method accelerates the convergence. We report results on the garden scene from the Mip-NeRF360 [4] dataset starting from the same initialization.

Method	*#I*	*#B*	SSIM↑	PSNR↑	LPIPS↓	Time (s)
3DGS + Ours	5	100	**0.969**	**33.77**	**0.030**	242
3DGS + Ours	5	80	**0.969**	33.73	0.031	233
3DGS + Ours	5	60	0.968	33.69	0.031	223
3DGS + Ours	5	40	0.967	33.51	0.032	212
3DGS [20]	5	100	0.967	33.48	0.033	164
3DGS [20]	50	100	0.968	33.57	0.032	218
3DGS [20]	100	100	**0.969**	33.65	0.031	270

Table 3. **Ablation of batch-size.** We vary the number of iterations (*#I*) and number of images in a batch (*#B*). Reducing *#B* for our method reduces runtime and consumed memory, while only slightly impacting quality. This demonstrates that image subsampling Sec. 3.3.2 is compatible with LM in our task. Increasing *#I* in 3DGS [20] with ADAM is slower and achieves worse quality than Ours using LM. We report average results on the NeRF-Synthetic [28] dataset starting from the same initialization.

the achieved quality is worse for the same batch size and number of iterations. When running for more iterations, ADAM eventually converges to similar quality, but needs more time. This highlights the efficiency of our optimizer: since we solve the normal equations in Eq. (2), one LM iteration makes a higher quality update step than ADAM which only uses the gradient direction.

4.3. Runtime Analysis

We analyze the runtime of our LM optimizer across multiple iterations in Fig. 6. The runtime is dominated by solving Eq. (3) with PCG and building the cache (Sec. 3.3). Sorting the cache, rendering all selected images, and the line search (Eq. (4)) are comparably faster. When running PCG, we execute the `applyJ` and `applyJT` kernels up to 8 times using the *per-pixel-per-splat* parallelization pattern (Algorithm 1). In contrast, we execute the `buildCache` kernel once and parallelize *per-pixel*, which is only marginally faster. This demonstrates the speed advantage obtained by using our proposed caching structure. We also provide a detailed profiling analysis of the kernels in the supplement.

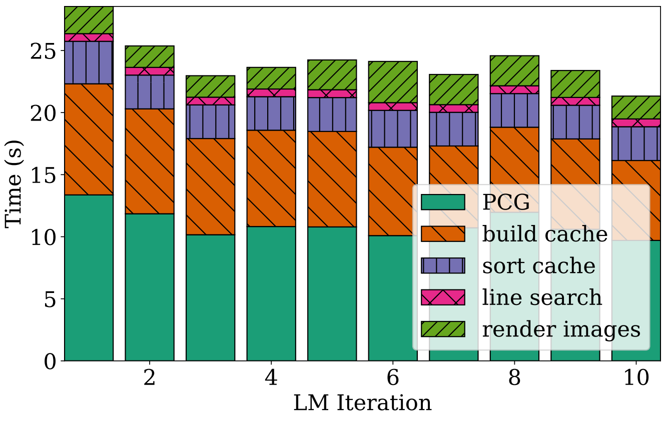

Figure 6. **Runtime Analysis.** One iteration of our LM optimizer is dominated by solving PCG and building the cache. Measured on the "garden" scene from Mip-NeRF360 [4] after densification.

4.4. Limitations

By replacing ADAM with our LM scheme, we accelerate the convergence speed by 30% on average for all datasets and baselines. However, some drawbacks remain.

First, our approach has a higher GPU memory footprint than baselines, due to our gradient cache (Sec. 3.3). Depending on the number of images and their resolution, this can make it hard to run our method on smaller GPUs. Following Mallick *et al.* [27], one could reduce the cache size by storing the gradients $\frac{\partial p}{\partial s}$ only for every 32nd splat along a ray and re-doing the α-blending in these local windows.

Second, our two-stage approach relies on ADAM to first complete the densification. The original 3DGS [20] densifies Gaussians up to 140 times, which is not easily transferable to the granularity of only 5 LM iterations. Instead, one could explore recent improvements in densification and integrate them into our method [5, 22].

5. Conclusion

We have presented 3DGS-LM, a method that accelerates the reconstruction of 3D Gaussian-Splatting [20] by replacing the ADAM optimizer with a tailored Levenberg-Marquardt (LM) (Sec. 3.2). We show that with our data parallelization scheme we can efficiently solve the normal equations with PCG in custom CUDA kernels (Sec. 3.3). Employed in a two-stage approach (Sec. 3.4), this leads to a 30% runtime acceleration compared to baselines. We further demonstrate that our approach is agnostic to other methods [11, 27, 41], which further improves the optimization runtime. Overall, we believe that the ability of faster 3DGS reconstructions with our method will open up further research avenues and make existing 3DGS more practical across a wide range of real-world applications.

6. Acknowledgements

This project was funded by a Meta sponsored research agreement. In addition, the project was supported by the ERC Starting Grant Scan2CAD (804724) as well as the German Research Foundation (DFG) Research Unit "Learning and Simulation in Visual Computing". We thank Justin Johnson for the helpful discussions in an earlier project with a similar direction and Peter Kocsis for the helpul discussions about image subsampling. We also thank Angela Dai for the video voice-over.

References

[1] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team. Ceres Solver, 2023. 4

[2] Kara-Ali Aliev, Artem Sevastopolsky, Maria Kolos, Dmitry Ulyanov, and Victor Lempitsky. Neural point-based graphics. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXII 16*, pages 696–712. Springer, 2020. 1, 2

[3] Jonathan T Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P Srinivasan. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 5855–5864, 2021. 1, 2

[4] Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5470–5479, 2022. 6, 7, 8, 13, 14, 15, 16, 17, 18

[5] Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. Revising densification in gaussian splatting. *arXiv preprint arXiv:2404.06109*, 2024. 2, 8

[6] David Charatan, Sizhe Lester Li, Andrea Tagliasacchi, and Vincent Sitzmann. pixelsplat: 3d gaussian splats from image pairs for scalable generalizable 3d reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 19457–19467, 2024. 2

[7] Anpei Chen, Haofei Xu, Stefano Esposito, Siyu Tang, and Andreas Geiger. Lara: Efficient large-baseline radiance fields. In *European Conference on Computer Vision (ECCV)*, 2024.

[8] 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. *arXiv preprint arXiv:2403.14627*, 2024. 2

[9] Angela Dai, Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Christian Theobalt. Bundlefusion: Real-time globally consistent 3d reconstruction using on-the-fly surface reintegration. *ACM Transactions on Graphics (ToG)*, 36(4): 1, 2017. 2

[10] Zachary DeVito, Michael Mara, Michael Zollhöfer, Gilbert Bernstein, Jonathan Ragan-Kelley, Christian Theobalt, Pat Hanrahan, Matthew Fisher, and Matthias Niessner. Opt: A domain specific language for non-linear least squares optimization in graphics and imaging. *ACM Transactions on Graphics (TOG)*, 36(5):1–27, 2017. 2

[11] Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, and Nandita Vijaykumar. Distwar: Fast differentiable rendering on raster-based rendering pipelines. *arXiv preprint arXiv:2401.05345*, 2023. 2, 6, 7, 8, 13, 16

[12] Zhiwen Fan, Wenyan Cong, Kairun Wen, Kevin Wang, Jian Zhang, Xinghao Ding, Danfei Xu, Boris Ivanovic, Marco Pavone, Georgios Pavlakos, et al. Instantsplat: Unbounded sparse-view pose-free gaussian splatting in 40 seconds. *arXiv preprint arXiv:2403.20309*, 2024. 2

[13] Guangchi Fang and Bing Wang. Mini-splatting: Representing scenes with a constrained number of gaussians. *arXiv preprint arXiv:2403.14166*, 2024. 2

[14] Guofeng Feng, Siyan Chen, Rong Fu, Zimu Liao, Yi Wang, Tao Liu, Zhilin Pei, Hengjie Li, Xingcheng Zhang, and Bo Dai. Flashgs: Efficient 3d gaussian splatting for large-scale and high-resolution rendering. *arXiv preprint arXiv:2408.07967*, 2024. 2

[15] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5501–5510, 2022. 2

[16] Antoine Guédon and Vincent Lepetit. Sugar: Surface-aligned gaussian splatting for efficient 3d mesh reconstruction and high-quality mesh rendering. *CVPR*, 2024. 2

[17] Abdullah Hamdi, Luke Melas-Kyriazi, Jinjie Mai, Guocheng Qian, Ruoshi Liu, Carl Vondrick, Bernard Ghanem, and Andrea Vedaldi. Ges: Generalized exponential splatting for efficient radiance field rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 19812–19822, 2024. 2

[18] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (ToG)*, 37(6):1–15, 2018. 2, 6, 7, 15, 16, 17, 18

[19] Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. 2d gaussian splatting for geometrically accurate radiance fields. In *SIGGRAPH 2024 Conference Papers*. Association for Computing Machinery, 2024. 2

[20] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42 (4), 2023. 1, 2, 3, 4, 5, 6, 7, 8, 12, 13, 15

[21] 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 Transactions on Graphics*, 43(4), 2024. 2

[22] Shakiba Kheradmand, Daniel Rebain, Gopal Sharma, Weiwei Sun, Jeff Tseng, Hossam Isack, Abhishek Kar, Andrea Tagliasacchi, and Kwang Moo Yi. 3d gaussian splatting as markov chain monte carlo. *arXiv preprint arXiv:2404.09591*, 2024. 2, 8

[23] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 2, 3

[24] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics (ToG)*, 36 (4):1–13, 2017. 6, 7, 15, 16, 17, 18

[25] Tianqi Liu, Guangcong Wang, Shoukang Hu, Liao Shen, Xinyi Ye, Yuhang Zang, Zhiguo Cao, Wei Li, and Ziwei Liu. Mvsgaussian: Fast generalizable gaussian splatting reconstruction from multi-view stereo. *arXiv preprint arXiv:2405.12218*, 2024. 2

[26] 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 *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 20654–20664, 2024. 2

[27] Saswat Subhajyoti Mallick, Rahul Goel, Bernhard Kerbl, Francisco Vicente Carrasco, Markus Steinberger, and Fernando De La Torre. Taming 3dgs: High-quality radiance fields with limited resources. *arXiv preprint arXiv:2406.15643*, 2024. 2, 6, 7, 8, 13, 18

[28] 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. *Communications of the ACM*, 65(1):99–106, 2021. 1, 2, 6, 7, 8

[29] Jorge J Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis: proceedings of the biennial Conference held at Dundee, June 28–July 1, 1977*, pages 105–116. Springer, 2006. 2, 3

[30] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM transactions on graphics (TOG)*, 41(4):1–15, 2022. 1, 2

[31] 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. *arXiv.org*, 2024. 2

[32] Panagiotis Papantonakis, Georgios Kopanas, Bernhard Kerbl, Alexandre Lanvin, and George Drettakis. Reducing the memory footprint of 3d gaussian splatting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 7(1):1–17, 2024. 2

[33] Sverker Rasmuson, Erik Sintorn, and Ulf Assarsson. Perf: performant, explicit radiance fields. *Frontiers in Computer Science*, 4:871808, 2022. 2

[34] 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. *arXiv preprint arXiv:2403.17898*, 2024. 2

[35] Cheng Sun, Min Sun, and Hwann-Tzong Chen. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5459–5469, 2022. 2

[36] A. Tewari, J. Thies, B. Mildenhall, P. Srinivasan, E. Tretschk, W. Yifan, C. Lassner, V. Sitzmann, R. Martin-Brualla, S. Lombardi, T. Simon, C. Theobalt, M. Nießner, J. T. Barron, G. Wetzstein, M. Zollhöfer, and V. Golyanik. Advances in Neural Rendering. *Computer Graphics Forum (EG STAR 2022)*, 2022. 1, 2

[37] Justus Thies, Michael Zollhöfer, Matthias Nießner, Levi Valgaerts, Marc Stamminger, and Christian Theobalt. Real-time expression transfer for facial reenactment. *ACM Trans. Graph.*, 34(6):183–1, 2015. 2

[38] Justus Thies, Michael Zollhofer, Marc Stamminger, Christian Theobalt, and Matthias Niessner. Face2face: Real-time face capture and reenactment of rgb videos. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 2

[39] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004. 3

[40] Qiangeng Xu, Zexiang Xu, Julien Philip, Sai Bi, Zhixin Shu, Kalyan Sunkavalli, and Ulrich Neumann. Point-nerf: Point-based neural radiance fields. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5438–5448, 2022. 2

[41] 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. *arXiv preprint arXiv:2409.06765*, 2024. 2, 6, 7, 8, 13, 17

[42] Chandan Yeshwanth, Yueh-Cheng Liu, Matthias Nießner, and Angela Dai. Scannet++: A high-fidelity dataset of 3d indoor scenes. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12–22, 2023. 1

[43] Zehao Yu, Anpei Chen, Binbin Huang, Torsten Sattler, and Andreas Geiger. Mip-splatting: Alias-free 3d gaussian splatting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 19447–19456, 2024. 2

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

[45] Hang Zhao, Orazio Gallo, Iuri Frosio, and Jan Kautz. Loss functions for image restoration with neural networks. *IEEE Transactions on computational imaging*, 3(1):47–57, 2016. 3

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

[47] Michael Zollhöfer, Matthias Nießner, Shahram Izadi, Christoph Rehmann, Christopher Zach, Matthew Fisher, Chenglei Wu, Andrew Fitzgibbon, Charles Loop, Christian Theobalt, et al. Real-time non-rigid reconstruction using an rgb-d camera. *ACM Transactions on Graphics (ToG)*, 33(4): 1–12, 2014. 2

[48] Michael Zollhöfer, Angela Dai, Matthias Innmann, Chenglei Wu, Marc Stamminger, Christian Theobalt, and Matthias Nießner. Shading-based refinement on volumetric signed

distance functions. *ACM Transactions on Graphics (ToG)*, 34(4):1–14, 2015. 2

3DGS-LM: Faster Gaussian-Splatting Optimization with Levenberg-Marquardt

Supplementary Material

A. More Details about CUDA Kernel Design

We introduce the necessary CUDA kernels to calculate the PCG algorithm in Sec. 3.3.1. In this section, we provide additional implementation details.

The necessary computations for the `buildCache` and `applyJT` steps in the PCG algorithm (see Algorithm 1) are split across three kernels. This follows the original design of the 3DGS differentiable rasterizer [20]. In both cases, we need to calculate the Jacobian-vector product of the form $\mathbf{g} = \mathbf{J}^T\mathbf{u}$ where $\mathbf{J} \in \mathbb{R}^{NxM}$ is the Jacobian matrix of N residuals and M Gaussian parameters and $\mathbf{u} \in \mathbb{R}^N$ is an input vector. The k-th element in the output vector is calculated as

$$\mathbf{g}_k = \sum_{i=0}^{N} \frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k}\mathbf{u}_i = \frac{\partial \mathbf{y}_k}{\partial \mathbf{x}_k}\sum_{i=0}^{N} \frac{\partial \mathbf{r}_i}{\partial \mathbf{y}_k}\mathbf{u}_i \tag{8}$$

where $\mathbf{r}_i$ is the i-th residual and $\mathbf{x}_k$ is the k-th Gaussian parameter. Following the chain-rule, it is possible to split up the gradient $\frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k} = \frac{\partial \mathbf{r}_i}{\partial \mathbf{y}_k}\frac{\partial \mathbf{y}_k}{\partial \mathbf{x}_k}$. We can use this to split the computation across three smaller kernels, where only the first needs to calculate the sum over all residuals: $\sum_{i=0}^{N} \frac{\partial \mathbf{r}_i}{\partial \mathbf{y}_k}\mathbf{u}_i$. The other kernels then only calculate the remaining steps by parallelizing over Gaussians. Since the sum needs to be implemented atomically, i.e., multiple threads write to the same output position, this is the main bottleneck for the kernel implementation. By splitting the computation in three smaller parts, we only need to do the costly summation in the first kernel and only to intermediate attributes. Concretely $\mathbf{y}_k$ are the 2D mean, color, and opacity attributes of the k-th projected Gaussian. Since both the `buildCache` and `applyJT` kernels use the gradient cache and our proposed *per-pixel-per-splat* parallelization pattern, we can implement the sum by first doing a warp reduce and then only issuing one `atomicAdd` statement per warp.

In contrast, the `applyJ` and `diagJTJ` computations cannot be split up into smaller kernels. Concretely, the `applyJ` kernel calculates $\mathbf{u} = \mathbf{J}\mathbf{p}$ with $\mathbf{p} \in \mathbb{R}^M$. The k-th element in the output vector is calculated as

$$\mathbf{u}_k = \sum_{i=0}^{M} \frac{\partial \mathbf{r}_k}{\partial \mathbf{x}_i}\mathbf{p}_i = \sum_{i=0}^{N} \frac{\partial \mathbf{r}_k}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_i}\mathbf{p}_i \tag{9}$$

Similarly, the `diagJTJ` kernel calculates $\mathbf{M} = \text{diag}(\mathbf{J}^T\mathbf{J}) \in \mathbb{R}^M$. The k-th element in the output vector is calculated as

$$\mathbf{g}_k = \sum_{i=0}^{N}(\frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k})^2 = \sum_{i=0}^{N}(\frac{\partial \mathbf{r}_i}{\partial \mathbf{y}_k}\frac{\partial \mathbf{y}_k}{\partial \mathbf{x}_k})^2 \tag{10}$$

In both cases it is not possible to move part of the gradients outside of the sum. As a consequence, both the `applyJ` and `diagJTJ` are implemented as one kernel, where each thread directly calculates the complete backward path to all Gaussian attributes. This slightly increases the number of required registers and the runtime compared to the `applyJT` kernel (see Tab. 4).

The `applyJ` kernel requires the input vector $\mathbf{p}$ to be sorted per Gaussian to make reading from it coalesced. That is: $\mathbf{p} = [x_1^a, ..., x_1^z, ..., x_M^a, ..., x_M^z]^T$, where x_k^a is the value corresponding to the a-th parameter of the k-th Gaussian. In total, each Gaussian consists of 59 parameters: 11 for position, rotation, scaling, and opacity and 48 for all Spherical Harmonics coefficients of degree 3. In contrast, all other kernels require the input vector to be sorted per attribute to make reading from it coalesced. That is: $\mathbf{q} = [x_1^a, ..., x_M^a, ..., x_1^z, ..., x_M^z]^T$. We use the structure of $\mathbf{q}$ for all other vector-vector calculations in Algorithm 1 as well. Whenever we call the `applyJ` kernel, we thus first call the `sortX` kernel that restructures $\mathbf{q}$ to the layout of $\mathbf{p}$.

B. Derivation of Weights for Subsampling

We sample batches of fewer images to decrease the size of the gradient cache (see Sec. 3.3.2). To combine the update vectors from multiple batches, we calculate the weighted mean, as detailed in Eq. (7). This weighted mean approximates the true solution without any image subsampling. That is, to obtain the update vector, we have to solve the normal equations, as detailed in Eq. (3). When subsampling images, we split the number of total residuals M into smaller chunks. Let's consider the case of two chunks, labeled as 1, 2. The normal equations (without subsampling) can then be written as:

$$\begin{bmatrix}\mathbf{J}_1^T & \mathbf{J}_2^T\end{bmatrix}\begin{bmatrix}\mathbf{J}_1 \\ \mathbf{J}_2\end{bmatrix}\Delta = \begin{bmatrix}\mathbf{J}_1^T & \mathbf{J}_2^T\end{bmatrix}\begin{bmatrix}\mathbf{F}_1(\mathbf{x}) \\ \mathbf{F}_2(\mathbf{x})\end{bmatrix} \tag{11}$$

where we drop the additional LM regularization term for clarity and divide the Jacobian and residual vector into separate matrices according to the chunks. The solution to the normal equations is obtained by:

$$\Delta = (\mathbf{J}_1^T\mathbf{J}_1 + \mathbf{J}_2^T\mathbf{J}_2)^{-1}(\mathbf{J}_1^T\mathbf{F}_1(\mathbf{x}) + \mathbf{J}_2^T\mathbf{F}_2(\mathbf{x})) \tag{12}$$

In contrast, when we subsample images, we obtain two separate solutions as:

$$\Delta_1 = (\mathbf{J}_1^T\mathbf{J}_1)^{-1}\mathbf{J}_1^T\mathbf{F}_1(\mathbf{x}) \tag{13}$$

$$\Delta_2 = (\mathbf{J}_2^T\mathbf{J}_2)^{-1}\mathbf{J}_2^T\mathbf{F}_2(\mathbf{x}) \tag{14}$$

We can rewrite Eq. (12) as a weighted mean of Δ_1, Δ_2:

$$\Delta = K^{-1}(\mathbf{J}_1^T\mathbf{J}_1)(\mathbf{J}_1^T\mathbf{J}_1)^{-1}(\mathbf{J}_1^T\mathbf{F}_1(\mathbf{x})) \tag{15}$$
$$+K^{-1}(\mathbf{J}_2^T\mathbf{J}_2)(\mathbf{J}_2^T\mathbf{J}_2)^{-1}(\mathbf{J}_2^T\mathbf{F}_2(\mathbf{x})) \tag{16}$$
$$= w_1\Delta_1 + w_2\Delta_2 \tag{17}$$

where $K = (\mathbf{J}_1^T\mathbf{J}_1 + \mathbf{J}_2^T\mathbf{J}_2)$, $w_1 = K^{-1}(\mathbf{J}_1^T\mathbf{J}_1)$, $w_2 = K^{-1}(\mathbf{J}_2^T\mathbf{J}_2)$. Calculating these weights requires to materialize and invert K, which is too costly to fit in memory. To this end, we approximate the true weights w_1 and w_2 with $\tilde{w}_1 = \text{diag}(w_1)$ and $\tilde{w}_2 = \text{diag}(w_2)$. This directly leads to the weighted mean that we employ in Eq. (7).

C. Detailed Analysis of Runtime

We provide additional analysis of the CUDA kernels by running the `Nsight Compute`[1] profiler. We provide results in Tab. 4 measured on a RTX3090 GPU for building/resorting the gradient cache and running one PCG iteration on the MipNerf360 [4] "garden" scene with a batch size of one image. We add the suffixes `_p1`, `_p2`, `_p3` to signal the three kernels that we use to implement the respective operation (see Appendix A).

D. Results per Scene

We provide a per-scene breakdown of our main quantitative results against all baselines on all datasets. The comparisons against 3DGS [20] are in Tab. 5. The comparisons against DISTWAR [11] are in Tab. 6. The comparisons against gsplat [41] are in Tab. 7. The comparisons against Taming-3DGS [27] are in Tab. 8.

[1]https://developer.nvidia.com/nsight-compute

Kernel	Duration (ms) ↓	Compute Throughput (%)↑	Memory Throughput (%)↑	Register Count ↓
`buildCache_p1`	31.32	78.56	78.56	64
`buildCache_p2`	0.53	17.43	87.94	58
`buildCache_p3`	4.12	4.54	73.45	74
`sortCacheByGaussians`	5.04	61.17	61.17	18
`diagJTJ`	41.60	71.13	71.13	90
`sortX`	4.45	15.15	60.30	36
`applyJ`	10.98	86.32	86.32	80
`applyJT_p1`	3.93	75.79	75.79	34
`applyJT_p2`	0.37	18.83	89.69	40
`applyJT_p3`	3.20	4.75	78.48	48

Table 4. **Profiler Analysis of CUDA kernels.** We provide results measured on a RTX3090 GPU for building/resorting the gradient cache and running one PCG iteration on the MipNerf360 [4] "garden" scene with a batch size of one image.

Method	Scene	MipNeRF-360 [4]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
3DGS [20]	treehill	0.631	22.44	0.330	1130
+ Ours	treehill	0.633	22.57	0.334	**836**
3DGS [20]	counter	0.905	28.96	0.202	1178
+ Ours	counter	0.904	28.89	0.206	**927**
3DGS [20]	stump	0.769	26.56	0.217	1234
+ Ours	stump	0.774	26.67	0.218	**895**
3DGS [20]	bonsai	0.939	31.99	0.206	1034
+ Ours	bonsai	0.938	31.84	0.208	**794**
3DGS [20]	bicycle	0.764	25.20	0.212	1563
+ Ours	bicycle	0.765	25.30	0.218	**1141**
3DGS [20]	kitchen	0.925	31.37	0.128	1389
+ Ours	kitchen	0.924	31.21	0.128	**1156**
3DGS [20]	flowers	0.602	21.49	0.340	1132
+ Ours	flowers	0.600	21.52	0.344	**819**
3DGS [20]	room	0.917	31.36	0.221	1210
+ Ours	room	0.916	31.10	0.224	**1004**
3DGS [20]	garden	0.862	27.23	0.109	1573
+ Ours	garden	0.863	27.30	0.110	**1175**

Method	Scene	Deep Blending [18]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
3DGS [20]	playroom	0.901	29.90	0.247	1085
+ Ours	playroom	0.905	30.24	0.246	**861**
3DGS [20]	drjohnson	0.898	29.12	0.246	1359
+ Ours	drjohnson	0.901	29.23	0.248	**1040**

Method	Scene	Tanks & Temples [24]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
3DGS [20]	train	0.811	21.95	0.209	636
+ Ours	train	0.811	22.07	0.214	**579**
3DGS [20]	truck	0.877	25.40	0.148	837
+ Ours	truck	0.876	25.36	0.151	**747**

Table 5. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics.

Method	Scene	MipNeRF-360 [4]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
DISTWAR [11]	treehill	0.633	22.47	0.327	898
+ Ours	treehill	0.635	22.54	0.332	**669**
DISTWAR [11]	counter	0.905	29.00	0.203	790
+ Ours	counter	0.904	28.91	0.205	**687**
DISTWAR [11]	stump	0.771	26.60	0.216	1017
+ Ours	stump	0.773	26.70	0.217	**760**
DISTWAR [11]	bonsai	0.939	32.13	0.206	677
+ Ours	bonsai	0.938	31.92	0.208	**578**
DISTWAR [11]	bicycle	0.763	25.19	0.212	1333
+ Ours	bicycle	0.764	25.26	0.218	**971**
DISTWAR [11]	kitchen	0.925	31.31	0.127	957
+ Ours	kitchen	0.924	31.14	0.128	**838**
DISTWAR [11]	flowers	0.602	21.45	0.340	884
+ Ours	flowers	0.596	21.48	0.348	**671**
DISTWAR [11]	room	0.916	31.41	0.221	803
+ Ours	room	0.916	31.40	0.224	**680**
DISTWAR [11]	garden	0.862	27.23	0.109	1338
+ Ours	garden	0.861	27.32	0.112	**1023**

Method	Scene	Deep Blending [18]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
DISTWAR [11]	playroom	0.900	29.81	0.247	729
+ Ours	playroom	0.905	30.24	0.246	**586**
DISTWAR [11]	drjohnson	0.898	29.13	0.247	953
+ Ours	drjohnson	0.901	29.13	0.249	**758**

Method	Scene	Tanks & Temples [24]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
DISTWAR [11]	train	0.812	22.05	0.209	504
+ Ours	train	0.810	22.10	0.216	**440**
DISTWAR [11]	truck	0.877	25.29	0.148	698
+ Ours	truck	0.877	25.28	0.150	**635**

Table 6. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics.

Method	Scene	MipNeRF-360 [4]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
gsplat [41]	treehill	0.634	22.44	0.324	973
+ Ours	treehill	0.635	22.54	0.332	**701**
gsplat [41]	counter	0.908	28.99	0.201	903
+ Ours	counter	0.904	28.91	0.205	**762**
gsplat [41]	stump	0.769	26.53	0.218	1097
+ Ours	stump	0.774	26.70	0.217	**793**
gsplat [41]	bonsai	0.937	31.95	0.208	783
+ Ours	bonsai	0.938	31.92	0.208	**646**
gsplat [41]	bicycle	0.765	25.21	0.206	1398
+ Ours	bicycle	0.765	25.26	0.218	**988**
gsplat [41]	kitchen	0.926	31.17	0.128	1086
+ Ours	kitchen	0.924	31.14	0.128	**921**
gsplat [41]	flowers	0.600	21.53	0.338	965
+ Ours	flowers	0.601	21.48	0.348	**709**
gsplat [41]	room	0.920	31.48	0.219	913
+ Ours	room	0.916	31.39	0.224	**753**
gsplat [41]	garden	0.869	27.48	0.105	1462
+ Ours	garden	0.861	27.32	0.112	**1085**

Method	Scene	Deep Blending [18]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
gsplat [41]	playroom	0.907	29.89	0.248	799
+ Ours	playroom	0.904	30.90	0.247	**626**
gsplat [41]	drjohnson	0.901	29.16	0.244	1040
+ Ours	drjohnson	0.901	29.07	0.251	**805**

Method	Scene	Tanks & Temples [24]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
gsplat [41]	train	0.811	21.64	0.209	558
+ Ours	train	0.809	22.09	0.216	**381**
gsplat [41]	truck	0.880	25.35	0.149	735
+ Ours	truck	0.877	25.28	0.150	**447**

Table 7. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics.

Method	Scene	MipNeRF-360 [4]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
Taming-3DGS† [27]	treehill	0.630	22.61	0.340	638
+ Ours	treehill	0.635	22.53	0.332	**510**
Taming-3DGS† [27]	counter	0.904	28.96	0.205	583
+ Ours	counter	0.904	28.91	0.205	**553**
Taming-3DGS† [27]	stump	0.764	26.51	0.225	728
+ Ours	stump	0.773	26.70	0.217	**579**
Taming-3DGS† [27]	bonsai	0.938	32.16	0.208	508
+ Ours	bonsai	0.938	31.92	0.208	**468**
Taming-3DGS† [27]	bicycle	0.756	25.18	0.224	882
+ Ours	bicycle	0.764	25.26	0.218	**702**
Taming-3DGS† [27]	kitchen	0.924	31.13	0.129	742
+ Ours	kitchen	0.924	31.14	0.128	**696**
Taming-3DGS† [27]	flowers	0.594	21.42	0.351	620
+ Ours	flowers	0.598	21.48	0.348	**507**
Taming-3DGS† [27]	room	0.916	31.64	0.224	561
+ Ours	room	0.916	31.39	0.224	**529**
Taming-3DGS† [27]	garden	0.861	27.36	0.113	895
+ Ours	garden	0.861	27.32	0.112	**748**

Method	Scene	Deep Blending [18]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
Taming-3DGS† [27]	playroom	0.903	29.99	0.245	502
+ Ours	playroom	0.904	30.23	0.247	**453**
Taming-3DGS† [27]	drjohnson	0.902	29.43	0.240	678
+ Ours	drjohnson	0.901	29.21	0.249	**627**

Method	Scene	Tanks & Temples [24]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
Taming-3DGS† [27]	train	0.815	22.18	0.205	411
+ Ours	train	0.809	22.06	0.217	**349**
Taming-3DGS† [27]	truck	0.879	25.40	0.146	514
+ Ours	truck	0.879	25.36	0.150	**436**

Table 8. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics.