# 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/

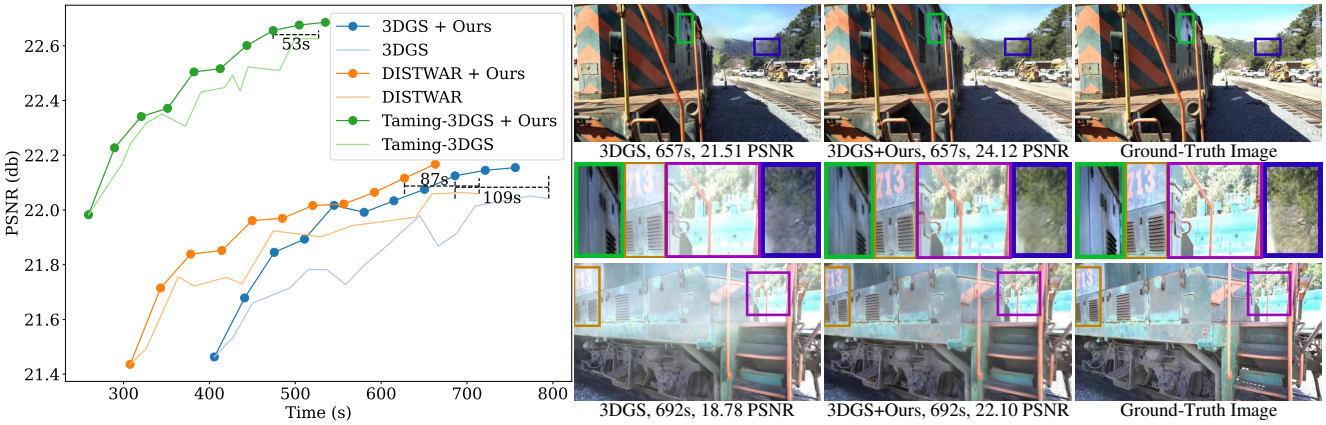


Figure 1. Our method accelerates 3D Gaussian Splatting (3DGS) [23] reconstruction by replacing the ADAM optimizer with a tailored Levenberg-Marquardt. Left: starting from the same initialization, our method converges faster on the Tanks&Temples `TRAIN` scene. Right: after the same amount of time, our 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 20% 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, 23, 33, 35, 42]. Among those, 3DGS [23] (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 [49]. 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, methods accelerate the rendering speed of the tile-based, differentiable rasterizer or the backward-pass that is specifically tailored for optimization with gradient descent [12, 15, 32, 48]. For example, Durvasula *et al*. [12] employ warp reductions for a

more efficient sum of rendering gradients, while Mallick *et al*. [32] utilizes a splat-parallelization for backpropagation. Second, in 3DGS the number of Gaussians is gradually grown during optimization, which is known as densification. Recently, GS-MCMC [25], Taming-3DGS [32], Mini-Splatting [14], 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 and rendered in every iteration.

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 [26] optimizer with a tailored Levenberg-Marquardt (LM) [34]. LM is known to drastically reduce the number of iterations by approximating second-order updates through solving the normal equations (Tab. 4). This allows us to accelerate 3DGS reconstruction (Fig. 1 left) by over 20% 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 20% 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 20% 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, 19, 23, 33, 35, 42]. NeRF [33] achieves highly photo-realistic image synthesis results through differentiable volumetric rendering. It was combined with explicit representations to accelerate optimization runtime [7, 16, 35, 41, 47]. 3D Gaussian Splatting (3DGS) [23] extends this idea by representing the scene as a set of 3D Gaussians, that are rasterized into 2D splats and then $\alpha$-blended into pixel colors. The approach gained popularity, due to the ability to render high quality images in real-time. Since its inception, 3DGS was improved along several axes. Recent methods improve the image quality by increasing or regularizing the capacity of primitives [18, 20, 22, 31, 50]. Others increase rendering efficiency [36, 40], obtain better surface reconstructions [17, 21], reduce the memory requirements [37], and enable large-scale reconstruction [24, 53]. We similarly adopt 3DGS as our scene representation and focus on improving the per-scene optimization runtime.

### 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, 14, 25, 30–32]. Other methods focus on sparse-view reconstruction and train a neural network as data prior, that outputs Gaussians in a single forward pass [6, 8, 9, 13, 29, 46, 54]. In contrast, we focus on the dense-view and per-scene optimization setting, i.e., we are not limited to sparse-view reconstruction. Most related are methods that improve the implementation of the underlying differentiable rasterizer. In [12, 48] the gradient descent backward pass is accelerated through warp-reductions, while [32] improves its parallelization pattern and [15] accelerates the rendering. In contrast, we completely replace the gradient descent optimization with LM through a novel and tailored GPU parallelization scheme. We demonstrate that we are compatible with those existing methods, i.e., we further reduce runtime by plugging our optimizer into their scene initializations.

### 2.3. Optimizers For 3D Reconstruction Tasks

NeRF and 3DGS are typically optimized with stochastic gradient descent (SGD) optimizers like ADAM [26] 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 [10, 11, 43, 44, 55, 56]. 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*. [39] implemented the Gauss-Newton algorithm for reconstructing low-resolution NeRFs based on dense voxel grids. In contrast, we exploit the explicit Gaussian primitives of 3DGS to perform highly-efficient Jacobian-vector products in a data-parallel fashion. This allows us to achieve state-of-the-art rendering quality, while significantly accelerating the optimization in comparison to ADAM-based methods.

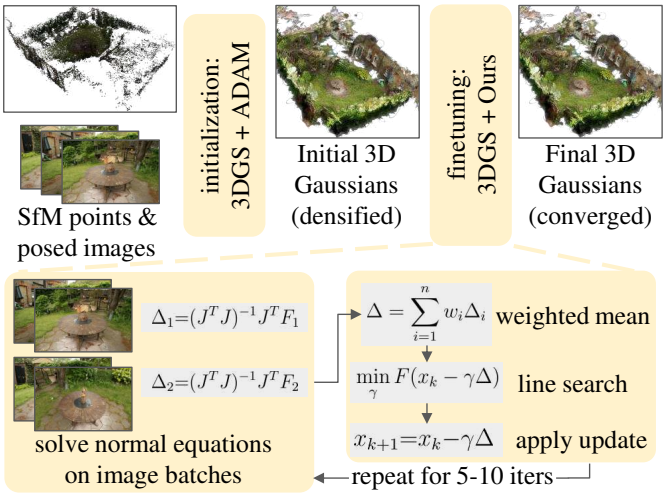


Figure 2. **Method Overview.** We accelerate 3DGS optimization by framing it in two stages. First, we use the original ADAM optimizer and densification scheme to arrive at an initialization for all Gaussians. Second, we employ the Levenberg-Marquardt algorithm to finish optimization.

## 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) [34] 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) [23] 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 given viewpoint, all Gaussians are first projected into 2D Gaussian splats with a tile-based differentiable rasterizer. Afterwards, they are $\alpha$-blended along a ray to obtain the pixel color $c$:

$$c = \sum_{i\in\mathcal{N}} c_i\alpha_i T_i, \quad \text{with } T_i = \prod_{j=1}^{i-1}(1-\alpha_j) \tag{1}$$

where $c_i$ is the color of the $i$-th splat along the ray, $\alpha_i$ is given by evaluating the 2D Gaussian multiplied with its opacity, and $T_i$ is the transmittance. To fit all Gaussian parameters $\mathbf{x}\in\mathbb{R}^M$ to posed image observations, a rendering loss is minimized with the ADAM [26] optimizer:

$$\mathcal{L}(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N}(\lambda_1|c_i-C_i| + \lambda_2(1-\text{SSIM}(c_i,C_i))) \tag{2}$$

where $\lambda_1$=0.8, $\lambda_2$=0.2, and $C_i$ the ground-truth for one pixel. Typically, 3DGS uses a batch size of 1 by sampling a random image per update step. The Gaussians are initialized from the SfM points and their number is gradually grown during the first half of the optimization, which is known as densification [23].

### 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:

$$E(\mathbf{x}) = \sum_{i=1}^{N}\sqrt{\lambda_1|c_i-C_i|}^2 + \sqrt{\lambda_2(1-\text{SSIM}(c_i,C_i))}^2 \tag{3}$$

where we have two *separate* residuals $r_i^{\text{abs}}=\sqrt{\lambda_1|c_i-C_i|}$ and $r_i^{\text{SSIM}}=\sqrt{\lambda_2(1-\text{SSIM}(c_i,C_i))}$ per color channel of each pixel. We take the square root of each loss term, to convert Eq. (2) 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 [34]. In practice, we select large enough subsets of all images to ensure reliable update steps (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:

$$(\mathbf{J}^T\mathbf{J} + \lambda_{\text{reg}}\text{diag}(\mathbf{J}^T\mathbf{J}))\Delta = -\mathbf{J}^T\mathbf{F}(\mathbf{x}) \tag{4}$$

where $\mathbf{F}(\mathbf{x})=[r_1^{\text{abs}},...,r_N^{\text{abs}},r_1^{\text{SSIM}},...,r_N^{\text{SSIM}}]\in\mathbb{R}^{2N}$ is the residual vector corresponding to Eq. (3) and $\mathbf{J}\in\mathbb{R}^{2N\times M}$ the corresponding Jacobian matrix.

In a typical dense capture setup, we optimize over millions of Gaussians and have hundreds of high-resolution images [4, 19, 27]. 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. (4) 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 $\Delta$, we run a line search to find the best scaling factor $\gamma\in\mathbb{R}$ for updating the Gaussian parameters:

$$\min_\gamma E(\mathbf{x}_k+\gamma\Delta) \tag{5}$$

In practice, we run the line search on a 30% subset of all images, which is enough to get a reasonable estimate for $\gamma$, 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 $\lambda_{\text{reg}}{\in}\mathbb{R}$ after every iteration based on the quality of the update step. Concretely, we calculate

$$\rho = \frac{||\mathbf{F}(\mathbf{x})||^2 - ||\mathbf{F}(\mathbf{x}+\gamma\Delta)||^2}{||\mathbf{F}(\mathbf{x})||^2 - ||\mathbf{J}\gamma\Delta + \mathbf{F}(\mathbf{x})||^2} \tag{6}$$

and only keep the update if $\rho{>}1\text{e}{-}5$, in which case we reduce the regularization strength as $\lambda_{\text{reg}}{*}{=}1{-}(2\rho{-}1)^3$. Otherwise, we revert the update and double $\lambda_{\text{reg}}$.

### 3.3. Efficient Parallelization Scheme For PCG

The PCG algorithm obtains the solution to the least squares problem of Eq. (4) in multiple iterations. We run the algorithm for up to $\text{n}_{\text{iters}}{=}8$ iterations and implement it with custom CUDA kernels. We summarize it in Algorithm 1.

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

**Input :** Gaussians and cameras $\mathcal{G}$, $\mathbf{F}$, $\lambda_{\text{reg}}$
**Output:** Update direction $\Delta$

$\mathbf{b}, \mathcal{C} = \texttt{buildCache}(\mathcal{G}, \mathbf{F})$ // $\mathbf{b}{=}-\mathbf{J}^T\mathbf{F}$
$\mathcal{C} = \texttt{sortCacheByGaussians}(\mathcal{C})$
$\mathbf{M}^{-1} = 1/\texttt{diagJTJ}(\mathcal{G}, \mathcal{C})$
$\mathbf{x_0} = \mathbf{M}^{-1}\mathbf{b}$
$\mathbf{u}_0 = \texttt{applyJ}(\texttt{sortX}(\mathbf{x}_0), \mathcal{G}, \mathcal{C})$ // $\mathbf{u}_0{=}\mathbf{J}\mathbf{x}_0$
$\mathbf{g}_0 = \texttt{applyJT}(\mathbf{u}_0, \mathcal{G}, \mathcal{C})$ // $\mathbf{g}_0{=}\mathbf{J}^T\mathbf{u}_0$
$\mathbf{r}_0 = \mathbf{b} - (\mathbf{g}_0{+}\lambda_{\text{reg}}\mathbf{M}\mathbf{x}_0)$
$\mathbf{z}_0 = \mathbf{M}^{-1}\mathbf{r}_0$
$\mathbf{p}_0 = \mathbf{z}_0$
**for** $i = 0$ **to** $n_{iters}$ **do**
  $\mathbf{u}_i = \texttt{applyJ}(\texttt{sortX}(\mathbf{p}_i), \mathcal{G}, \mathcal{C})$ // $\mathbf{u}_i{=}\mathbf{J}\mathbf{p}_i$
  $\mathbf{g}_i = \texttt{applyJT}(\mathbf{u}_i, \mathcal{G}, \mathcal{C})$ // $\mathbf{g}_i{=}\mathbf{J}^T\mathbf{u}_i$
  $\mathbf{g}_i \mathrel{+}= \lambda_{\text{reg}}\mathbf{M}\mathbf{p}_i$
  $\alpha_i = \frac{\mathbf{r}_i^T\mathbf{z}_i}{\mathbf{p}_i^T\mathbf{g}_i}$
  $\mathbf{x}_{i+1}{=}\mathbf{x}_i{+}\alpha_i\mathbf{p}_i$
  $\mathbf{r}_{i+1}{=}\mathbf{r}_i{-}\alpha_i\mathbf{g}_i$
  $\mathbf{z}_{i+1}{=}\mathbf{M}^{-1}\mathbf{r}_{i+1}$
  $\beta_i = \frac{\mathbf{r}_{i+1}^T\mathbf{z}_{i+1}}{\mathbf{r}_i^T\mathbf{z}_i}$
  $\mathbf{p}_{i+1} = \mathbf{z}_{i+1} + \beta_i\mathbf{p}_i$
  **if** $||\mathbf{r}_{i+1}||^2 < 0.01||\mathbf{b}||^2$ **then**
    break
  **end if**
**end for**
**return** $\mathbf{x}_{i+1}$

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 compute it 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 the residuals through the $\alpha$-blending (Eq. (1)) and splat projection steps to the Gaussian parameters. The tile-based rasterizer of 3DGS [23] performs this calculation using 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 the repetition of the ray marching: per PCG iteration we do it once for $\mathbf{u}_i$ and once for $\mathbf{g}_i$. As a consequence, the same intermediate $\alpha$-blending states (i.e., $T_s$, $\frac{\partial c}{\partial \alpha_s}$, $\frac{\partial c}{\partial c_s}$ for every splat $s$ along the ray) are re-calculated multiple (up to 18) times.

**Cache-driven parallelization** We propose to change the parallelization to *per-pixel-per-splat* (summarized in Fig. 3). That is, one thread handles all residuals of one ray for one splat. Each entry of $\mathbf{J}$ is the gradient from a residual $r$ (either of the L1 or SSIM terms) to a Gaussian parameter $x_i$. Conceptually, this can be computed in three stages:

$$\frac{\partial r}{\partial x_i} = \frac{\partial r}{\partial c}\frac{\partial c}{\partial s}\frac{\partial s}{\partial x_i} \tag{7}$$

where $\frac{\partial r}{\partial c}$ denotes the gradient from the residual to the rendered color, $\frac{\partial c}{\partial s}$ from the color to the projected splat, and $\frac{\partial s}{\partial x_i}$ from the splat to the Gaussian parameter. The first and last factors of Eq. (7) can be computed *independently* for each residual and splat respectively, which allows for an efficient parallelization. Similarly, we can calculate $\frac{\partial c}{\partial s}$ independently, if we have access to $T_s$ and $\frac{\partial c}{\partial \alpha_s}$. Instead of looping over all splats along a ray multiple times, we cache these quantities once (Fig. 3 left). When calculating $\mathbf{u}_i$ or $\mathbf{g}_i$, we then read these values from the cache (Fig. 3 right). This allows us to parallelize over all splats in all pixels, which drastically accelerates the runtime. The cache size is 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 that limits the cache size to the available budget.

3DGS uses the structural similarity index measure (SSIM) as loss term during optimization (Eq. (2)). In SSIM, the local neighborhood of every pixel gets convolved with Gaussian kernels to obtain the final per-pixel score [45]. We calculate $\frac{\partial r}{\partial c}$ for the SSIM residuals by backpropagating the per-pixel scores to the center pixels (ignoring the contribution to other pixels in the local neighborhood). This allows us to keep rays independent of each other thereby allowing for an efficient parallelization. We implement it following the derivation of Zhao *et al*. [52].

**Mapping of PCG to CUDA kernels** We cache all gradients $\frac{\partial c}{\partial s}$ using the `buildCache` operation. Following the implementation of the differentiable rasterizer in 3DGS [23], it uses the *per-pixel* parallelization and calculates the gradient update $\mathbf{b}{=}-\mathbf{J}^T\mathbf{F}$. For coalesced read

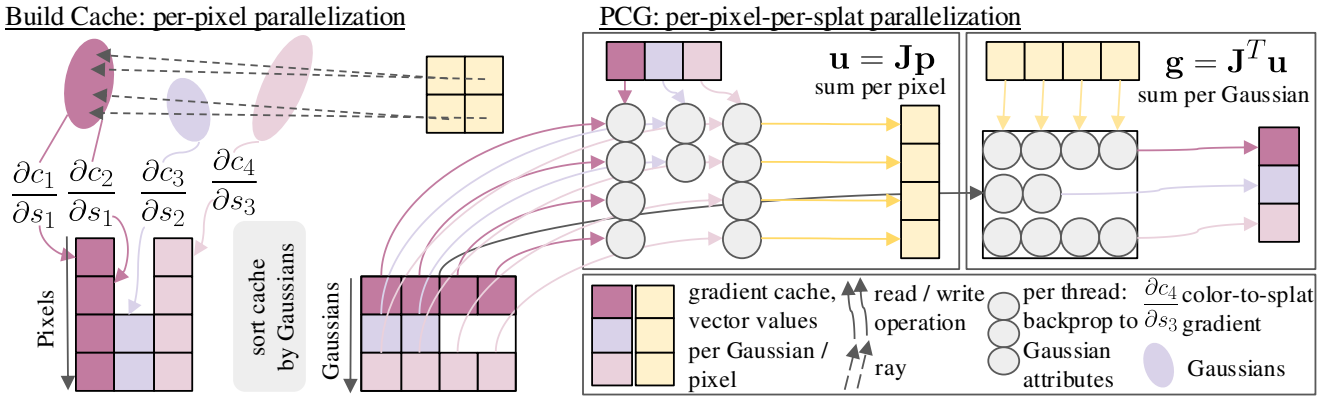


Figure 3. **Parallelization Strategy And Caching Scheme.** We implement the PCG algorithm with efficient CUDA kernels, that use a gradient cache to calculate Jacobian-vector products. Left: before PCG starts, we create the gradient cache following the *per-pixel* parallelization of 3DGS [23]. Afterwards, we sort the cache by Gaussians to ensure coalesced read accesses. Right: the cache decouples splats along rays, which allows us to parallelize *per-pixel-per-splat* when computing $\mathbf{u} = \mathbf{J}\mathbf{p}$ and $\mathbf{g} = \mathbf{J}^T\mathbf{u}$ during PCG.

and write accesses, we first store the cache sorted by pixels (Fig. 3 left). Afterwards, we re-sort it by Gaussians using the `sortCacheByGaussians` kernel. We use the Jacobi preconditioner $\mathbf{M}^{-1}=1/\text{diag}(\mathbf{J}^T\mathbf{J})$ and calculate it once using the *per-pixel-per-splat* parallelization in the `diagJTJ` kernel. The inner PCG loop involves two kernels that are accelerated by our novel parallelization scheme. First, `applyJ` computes $\mathbf{u}=\mathbf{J}\mathbf{p}$, which we implement as a per-pixel sum aggregation. Afterwards, `applyJT` computes $\mathbf{g}=\mathbf{J}^T\mathbf{u}$. This per-Gaussian sum can be efficiently aggregated using warp reductions. We compute the remaining vector-vector terms of Algorithm 1 directly in PyTorch [38]. We refer to the supplementary material for more details.

**Image Subsampling Scheme** Our cache 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. (4). This allows us to store the cache only for one batch at a time. Concretely, for $n_b$ batches, we obtain $n_b$ update vectors and combine them in a weighted mean:

$$\Delta = \sum_{i=1}^{n_b} \frac{\mathbf{M}_i \Delta_i}{\sum_{k=1}^{n} \mathbf{M}_k} \tag{8}$$

where we use the inverse of the PCG preconditioner $\mathbf{M}_i=\text{diag}(\mathbf{J}_i^T\mathbf{J}_i)$ as the weights. We refer to the supplementary material 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 rendered 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 choose batch sizes of 25-70 images and up to $n_b=4$ 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 scene coverage in all batches.

### 3.4. 3DGS Optimization In Two Stages

Our pipeline utilizes the LM optimizer in the second stage of 3DGS optimization (see Fig. 2). Before that, we run the ADAM optimizer to obtain an initialization of the Gaussian parameters. We compare this against running our LM optimizer directly on the Gaussian initialization obtained from the SfM point cloud (following [23]). Fig. 4 shows, that our LM converges faster for better initialized Gaussians and eventually beats pure ADAM. In contrast, running it directly on the SfM initialization is slower. This demonstrates that quasi second-order solvers like ours are well-known to be more sensitive to initialization. In other words, gradient descent makes rapid progress in the beginning, but needs more time to converge to final Gaussian parameters. The additional compute overhead of our LM optimization is especially helpful to converge more quickly. This motivates us to split the method in two stages. It also allows us to complete the densification of the Gaussians before employing the LM optimizer, which simplifies the implementation.

## 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 [23], its reimplementation "gsplat" [48], and DISTWAR [12]. Additionally, we compare against Taming-3DGS [32] by utilizing their "budgeted" approach as the fastest baseline in terms of runtime. We run all baselines for 30K iterations with their default hyperparameters.

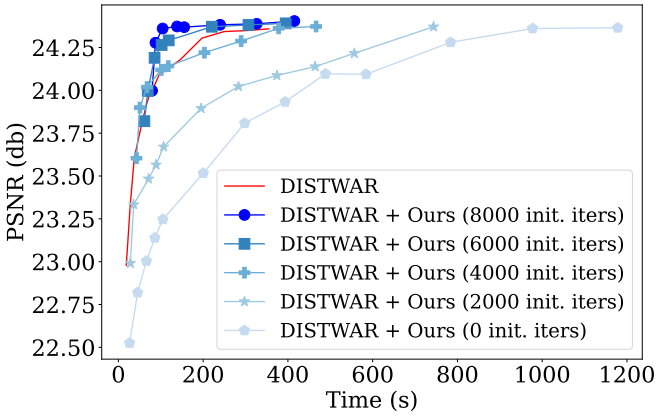


Figure 4. **Comparison of initialization iterations.** In our first stage, we initialize the Gaussians with gradient descent for K iterations, before finetuning with our LM optimizer. After K=6000 or K=8000 iterations, our method converges faster than the baseline. With less iterations, pure LM is slower, which highlights the importance of our two stage approach. Results reported on the GARDEN scene from MipNeRF360 [33] without densification.

**Datasets / Metrics** We benchmark our runtime improvements on three established datasets: Tanks&Temples [27], Deep Blending [19], 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 [23] publication. To measure the quality of the reconstruction, we report peak signal-to-noise ratio (PSNR), structural similarity (SSIM), and perceptual similarity (LPIPS) [51] 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 [19], and 70 images in 3 batches for Tanks&Temples [27]. We constrain the value range of $\lambda_{\text{reg}}$ for stable updates. We define it in $[1e{-}4, 1e4]$ for Deep Blending [19] and Tanks&Temples [27] 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 20% 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 supplementary 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 objective in our LM optimizer as in the original 3DGS implementation, namely the L1 and SSIM loss terms (Eq. (2)). Since LM energy terms are defined as a sum of squares, we adopt the square root formulation of these loss terms to arrive at an identical objective (Eq. (3)). We compare this choice against fitting the Gaussians with only an L2 loss, that does not require taking a square root. Concretely, we compare the achieved quality and runtime of LM against ADAM for both the L2 loss and the L1 and 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 L2 loss. This highlights the importance of the L1 and SSIM loss terms and why we adopt them in our method as well. We show in the supplementary material, that computing these loss terms instead of the simpler L2 residuals does not negatively impact the efficiency of our CUDA kernels.

**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). This controls the runtime of one iteration and how much GPU memory our optimizer consumes. We compare different numbers of images in Tab. 3 on the NeRF-Synthetic [33] dataset in a single batch per LM iteration, i.e., $n_b{=}1$. Using the full dataset (100 images) produces the best results. 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.

**Are we better than multi-view ADAM?** Our method converges with fewer iterations than baselines. Concretely, we require only 5-10 additional LM iterations after the initial-

| Method | MipNeRF-360 [4] | | | | Tanks&Temples [27] | | | | Deep Blending [19] | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) |
| 3DGS [23] | 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 [12] | 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 [48] | 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 [32] | 0.793 | 27.14 | 0.260 | 566 | 0.833 | 23.76 | 0.209 | 366 | 0.900 | 29.84 | 0.274 | 447 |
| + Ours | 0.791 | 27.13 | 0.260 | **453** | 0.832 | 23.72 | 0.209 | **310** | 0.901 | 29.91 | 0.275 | **347** |

Table 1. **Quantitative comparison of our method and baselines.** By adding our method to baselines, we accelerate the optimization time by 20% 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.

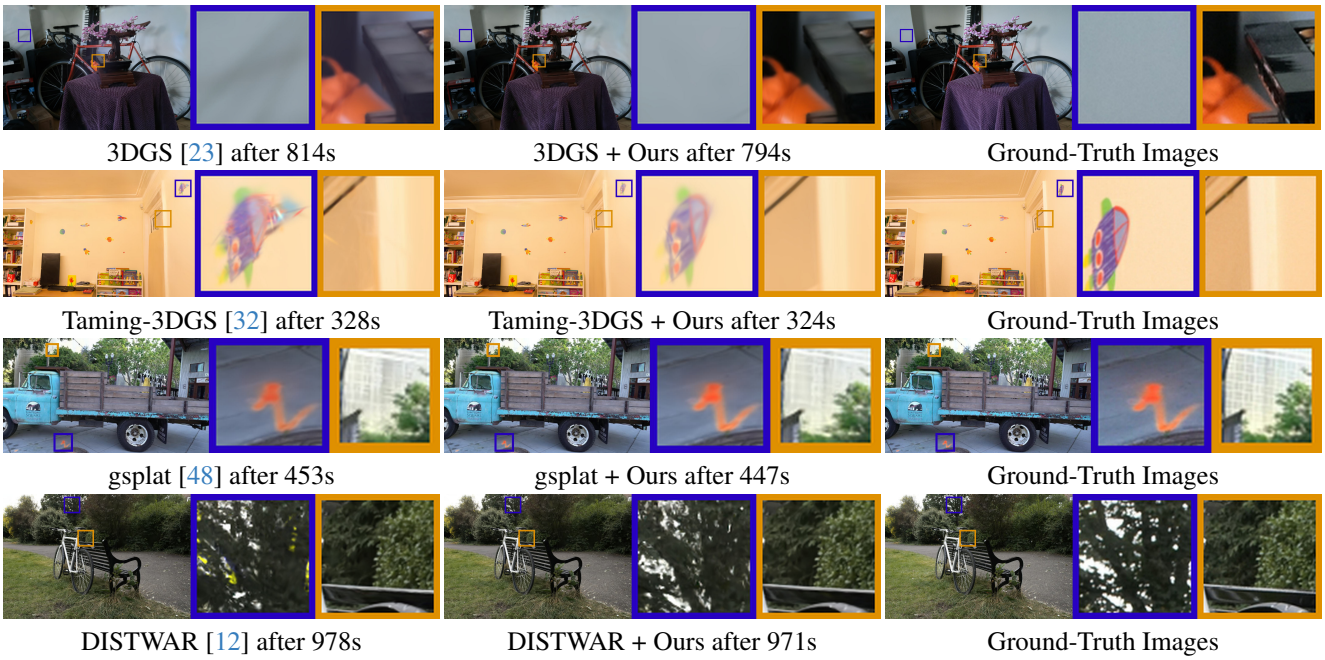


Figure 5. **Qualitative comparison of our method and baselines.** We compare rendered test images after similar optimization time. All baselines converge faster when using our LM optimizer, which shows in images with fewer artifacts and more accurate brightness / contrast.

ization, whereas ADAM runs for another 10K iterations. We increase the batch-size (number of images) for the baselines, such that the same number of multi-view constraints are observed for the respective update steps. However, as can be seen in Tab. 4, the achieved quality is worse for ADAM after the same 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. (3), 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. (4) with PCG and building the cache (Sec. 3.3). Sorting the cache, rendering the selected images, and the line search (Eq. (5)) are comparatively faster. During PCG, we run the `applyJ` and `applyJT` kernels up to 9 times, parallelizing *per-pixel-per-splat*. In contrast, we run the `buildCache` kernel once, parallelizing *per-pixel*, which is only marginally faster. This shows the advantage of our

| Method | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) |
|---|---|---|---|---|
| 3DGS [23] (L1/SSIM) | 0.862 | 27.23 | **0.108** | 1573 |
| 3DGS + Ours (L1/SSIM) | **0.863** | **27.29** | 0.110 | **1175** |
| 3DGS [23] (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 [23] optimized with ADAM and combined with ours, we achieve better results with the L1/SSIM objective. In both cases, our method accelerates the convergence. Results on the GARDEN scene from MipNeRF360 [4].

| Batch Size | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) | Mem (Gb) |
|---|---|---|---|---|---|
| 100 | **0.969** | **33.77** | **0.030** | 242 | 32.5 |
| 80 | **0.969** | 33.73 | 0.031 | 233 | 29.8 |
| 60 | 0.968 | 33.69 | 0.031 | 223 | 22.6 |
| 40 | 0.967 | 33.51 | 0.032 | 212 | 15.4 |

Table 3. **Ablation of batch-size.** Selecting fewer images per LM iteration reduces runtime and consumed GPU memory, while only slightly impacting quality. This demonstrates that image subsampling (Sec. 3.3) is compatible with LM in our task. Results obtained after initialization with 3DGS [23] and with $n_b$=1.

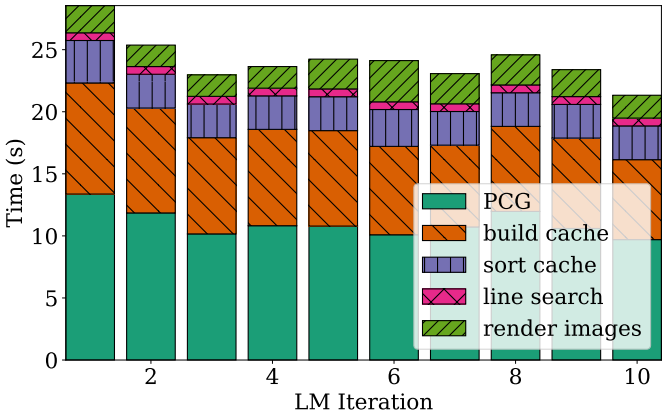


Figure 6. **Runtime Analysis.** One iteration of our LM optimizer is dominated by solving PCG and building the cache. Measured on the GARDEN scene from Mip-NeRF360 [4] after densification.

proposed parallelization scheme: the same Jacobian-vector product runs much faster. We also provide a detailed profiling analysis of our kernels in the supplementary material.

### 4.4. Limitations

By replacing ADAM with our LM scheme, we accelerate the 3DGS convergence speed by 20% on average for all datasets and baselines. However, some drawbacks remain. First, our approach requires more GPU memory than baselines, due to our gradient cache (Sec. 3.3). Depending on the number and resolution of images, this might require additional CPU offloading of cache parts to run our method on smaller GPUs. Following Mallick *et al.* [32], one can further reduce the cache size by storing the gradients $\frac{\partial c}{\partial s}$ only for every 32nd splat along a ray and re-doing the $\alpha$-blending in these local windows. Second, our two-stage approach relies on ADAM for the densification. 3DGS [23] densifies Gaussians up to 140 times, which is not easily transferable to the granularity of only 5-10 LM iterations. Instead, one could explore and integrate recent alternatives [5, 25, 30].

| Method | Iterations | Batch-Size | Time (s) | PSNR↑ |
|---|---|---|---|---|
| 3DGS [23] | 10,000 | 1 | 1222 | 29.51 |
| 3DGS [23] | 50 | 75 | 962 | 29.54 |
| 3DGS [23] | 130 | 75 | 1193 | 29.68 |
| + Ours | 5 | 75 | **951** | 29.72 |
| DISTWAR [12] | 10,000 | 1 | 841 | 29.47 |
| DISTWAR [12] | 50 | 75 | 681 | 29.49 |
| DISTWAR [12] | 130 | 75 | 814 | 29.58 |
| + Ours | 5 | 75 | **672** | 29.60 |
| gsplat [48] | 10,000 | 1 | 919 | 29.52 |
| gsplat [48] | 50 | 75 | 724 | 29.53 |
| gsplat [48] | 130 | 75 | 892 | 29.56 |
| + Ours | 5 | 75 | **716** | 29.58 |
| Taming-3DGS [32] | 10,000 | 1 | 447 | 29.84 |
| Taming-3DGS [32] | 50 | 75 | 328 | 29.86 |
| Taming-3DGS [32] | 130 | 75 | 391 | 29.91 |
| + Ours | 5 | 75 | **347** | 29.91 |

Table 4. **Analysis of multi-view constraints.** We obtain higher quality update steps from our LM optimization and need fewer iterations to converge. Using equally many images in a batch, baselines using ADAM still require more iterations and runtime to reach similar quality. Results averaged on DeepBlending [19].

## 5. Conclusion

We have presented 3DGS-LM, a method that accelerates the reconstruction of 3D Gaussian-Splatting [23] 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 20% runtime acceleration compared to baselines. We further demonstrate that our approach is agnostic to other methods [12, 32, 48], which further improves the optimization runtime; i.e., we can easily combine our proposed optimizer with faster 3DGS methods. Overall, we believe that the ability of faster 3DGS reconstructions with our method will open up further research avenues like [28] and make 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. 3, 6, 7, 8, 14, 15, 16

[5] Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. Revising densification in gaussian splatting. *European Conference on Computer Vision*, 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, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. Tensorf: Tensorial radiance fields. In *European conference on computer vision*, pages 333–350. Springer, 2022. 2

[8] Anpei Chen, Haofei Xu, Stefano Esposito, Siyu Tang, and Andreas Geiger. Lara: Efficient large-baseline radiance fields. In *European conference on computer vision*, 2024. 2

[9] 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. *European conference on computer vision*, 2024. 2

[10] 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

[11] 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

[12] 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. 1, 2, 5, 7, 8, 14, 15

[13] 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

[14] Guangchi Fang and Bing Wang. Mini-splatting: Representing scenes with a constrained number of gaussians. *European conference on computer vision*, 2024. 2

[15] Guofeng Feng, Siyan Chen, Rong Fu, Zimu Liao, Yi Wang, Tao Liu, Boni Hu, Linning Xu, Zhilin Pei, Hengjie Li, et al. Flashgs: Efficient 3d gaussian splatting for large-scale and high-resolution rendering. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 26652–26662, 2025. 1, 2

[16] 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

[17] Antoine Guédon and Vincent Lepetit. Sugar: Surface-aligned gaussian splatting for efficient 3d mesh reconstruction and high-quality mesh rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5354–5363, 2024. 2

[18] 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

[19] 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, 3, 6, 7, 8, 15, 16

[20] Jan Held, Renaud Vandeghen, Abdullah Hamdi, Adrien Deliege, Anthony Cioppa, Silvio Giancola, Andrea Vedaldi, Bernard Ghanem, and Marc Van Droogenbroeck. 3D convex splatting: Radiance field rendering with 3D smooth convexes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025. 2

[21] 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

[22] Yi-Hua Huang, Ming-Xian Lin, Yang-Tian Sun, Ziyi Yang, Xiaoyang Lyu, Yan-Pei Cao, and Xiaojuan Qi. Deformable

radial kernel splatting. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 21513–21523, 2025. 2

[23] 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, 14, 15

[24] 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

[25] 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

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

[27] 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. 3, 6, 7, 15, 16

[28] Lei Lan, Tianjia Shao, Zixuan Lu, Yu Zhang, Chenfanfu Jiang, and Yin Yang. 3dgs2: Near second-order converging 3d gaussian splatting. *arXiv preprint arXiv:2501.13975*, 2025. 8

[29] 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. *European conference on computer vision*, 2024. 2

[30] Tao Lu, Ankit Dhiman, R Srinath, Emre Arslan, Angela Xing, Yuanbo Xiangli, R Venkatesh Babu, and Srinath Sridhar. Turbo-gs: Accelerating 3d gaussian fitting for high-quality radiance fields. *arXiv preprint arXiv:2412.13547*, 2024. 2, 8

[31] 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

[32] Saswat Subhajyoti Mallick, Rahul Goel, Bernhard Kerbl, Markus Steinberger, Francisco Vicente Carrasco, and Fernando De La Torre. Taming 3dgs: High-quality radiance fields with limited resources. In *SIGGRAPH Asia 2024 Conference Papers*, New York, NY, USA, 2024. Association for Computing Machinery. 1, 2, 5, 7, 8, 14, 16

[33] 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

[34] 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

[35] 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

[36] 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. *International Conference on 3D Vision 2025*, 2025. 2

[37] 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

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. 5

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

[40] 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

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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. 4

[46] Haofei Xu, Songyou Peng, Fangjinhua Wang, Hermann Blum, Daniel Barath, Andreas Geiger, and Marc Pollefeys. Depthsplat: Connecting gaussian splatting and depth. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 16453–16463, 2025. 2

[47] 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

[48] 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. 1, 2, 5, 7, 8, 14, 16

[49] 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

[50] 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

[51] 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

[52] 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. 4

[53] Hexu Zhao, Haoyang Weng, Daohan Lu, Ang Li, Jinyang Li, Aurojit Panda, and Saining Xie. On scaling up 3d gaussian splatting training. In *European Conference on Computer Vision*, pages 14–36. Springer, 2025. 2

[54] Chen Ziwen, Hao Tan, Kai Zhang, Sai Bi, Fujun Luan, Yicong Hong, Li Fuxin, and Zexiang Xu. Long-lrm: Long-sequence large reconstruction model for wide-coverage gaussian splats. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2025. 2

[55] 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

[56] 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. In this section, we provide additional implementation details.

### A.1. Parallelization Pattern

We implement the *per-pixel-per-splat* parallelization pattern in our CUDA kernels by reading subsequent entries from the gradient cache in subsequent threads. This makes reading cache values perfectly coalesced and therefore minimizes the overhead caused by the operation. The gradient cache is sorted over Gaussians, which means that subsequent entries refer to different rays (pixels) that saw this projected Gaussian (splat). In general, one thread handles all residuals corresponding to the respective pixel, i.e., many computations are shared across color channels and are therefore combined.

### A.2. Design Of `buildCache` And `applyJT`

The necessary computations for the `buildCache` and `applyJT` steps in PCG (see Algorithm 1) are split across three kernels. This follows the original design of the 3DGS differentiable rasterizer [23]. In both cases, we 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:

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

where $\mathbf{r}_i$ is the $i$-th residual and $\mathbf{x}_k$ is the $k$-th Gaussian parameter. In other words, $\mathbf{g}_k$ is a sum over all residuals for 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: $\sum_{i=0}^{N}\frac{\partial \mathbf{r}_i}{\partial \mathbf{y}_k}\mathbf{u}_i$. This sum is the main bottleneck for the kernel implementation, since it needs to be implemented atomically (i.e., multiple threads write to the same output position). The other kernels then only calculate the remaining steps by parallelizing over Gaussians. We slightly abuse notation and denote with $\mathbf{y}_k$ the 2D mean, color, and opacity attributes of the $k$-th projected Gaussian. That is, the first kernel sums up the partial derivatives to each of these attributes in separate vectors. In the following, we add the suffixes `_p1, _p2, _p3` to denote the three kernels for the respective operation (where `_p1` refers to the kernel that calculates the sum over residuals).

The `buildCache_p1` utilizes the original *per-pixel* parallelization, whereas the `applyJT_p1` kernel use the gradient cache and our proposed *per-pixel-per-splat* parallelization pattern. The gradient cache is sorted over Gaussians, i.e., subsequent entries correspond to different rays of the same splat. This allows us to efficiently implement the sum by first performing a segmented warp reduce and then only issuing one `atomicAdd` statement per warp.

### A.3. Design Of `applyJ` And `diagJTJ`

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:

$$\mathbf{u}_k = \sum_{i=0}^{M} \frac{\partial \mathbf{r}_k}{\partial \mathbf{x}_i}\mathbf{p}_i = \sum_{i=0}^{N} \frac{\partial \mathbf{r}_k}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_i}\mathbf{p}_i \tag{10}$$

In other words, $\mathbf{u}_k$ is a sum over all Gaussian attributes for the $k$-th residual. 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:

$$\mathbf{M}_k = \sum_{i=0}^{N} \left(\frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k}\right)^2 = \sum_{i=0}^{N} \left(\frac{\partial \mathbf{r}_i}{\partial \mathbf{y}_k}\frac{\partial \mathbf{y}_k}{\partial \mathbf{x}_k}\right)^2 \tag{11}$$

In both cases it is not possible to move part of the gradients outside of the sum. As a consequence, both `applyJ` and `diagJTJ` are implemented as one kernel, where each thread directly calculates the final partial derivatives to all Gaussian attributes. This slightly increases the number of required registers and the runtime compared to the `applyJT` kernel (see Tab. 5). The `diagJTJ` kernel makes use of the same segmented warp reduce as `applyJT_p1` for efficiently summing up the squared partial derivatives. The `applyJ` kernel first sums up over all Gaussian attributes within each thread separately. Then, we only issue one `atomicAdd` statement for each residual per thread.

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 output vector to be sorted per attribute to make writing to 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 $\mathbf{p}$.

### A.4. Precomputation Of Residual-To-Pixel Weights

We adopt the square root formulation of the residuals in our energy formulation (see Eq. (3)). We efficiently precompute the contribution of the square root to the partial derivatives of Eq. (9), Eq. (10), and Eq. (11). In the following, we divide the partial derivatives from the $i$-th residual to the $k$-th Gaussian attribute into two stages:

$$\frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k} = \frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i} \frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k} \tag{12}$$

where $\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i}$ goes from the residual to the rendered pixel color and $\frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k}$ from the pixel color to the Gaussian attribute. Since we adopt the L1 and SSIM loss terms and take their square root, the terms $\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i}$ need to be calculated accordingly. In contrast, when using the L2 loss they take a trivial form of $\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i}$=1. In the following, we show that we can simplify the calculation of $\frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k}$ in the kernels by precomputing $\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i}$.

The $k$-th element of $\mathbf{g}$ is calculated as:

$$\mathbf{g}_k = (\mathbf{J}^T \mathbf{J} \mathbf{p})_k = \sum_{i=0}^{N} \frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k} \sum_{j=0}^{M} \frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_j} \mathbf{p}_j \tag{13}$$

By substituting Eq. (12) into Eq. (13), we factor out $\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i}$:

$$\mathbf{g}_k = \sum_{i=0}^{N} (\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i})^2 \frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k} \sum_{j=0}^{M} \frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_j} \mathbf{p}_j \tag{14}$$

The terms $\frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k}$ are identical for a residual of the same pixel and color channel that corresponds to either the L1 or SSIM loss terms, respectively. To avoid computing $\frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k}$ twice (and therefore doubling the grid size of all kernels), we instead sum up the contribution of both loss terms:

$$(\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i})^2 = (\frac{\partial \mathbf{r}_i^1}{\partial \mathbf{c}_i})^2 + (\frac{\partial \mathbf{r}_i^2}{\partial \mathbf{c}_i})^2 \tag{15}$$

where $\mathbf{r}_i^1$ corresponds to the $i$-th L1 residual and $\mathbf{r}_i^2$ to the $i$-th SSIM residual. Additionally, we implement the multiplication with $(\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i})^2$ in Eq. (14) as elementwise vector product (denoted by $\odot$) of $\hat{\mathbf{u}}$=$[\sum_{j=0}^{M} \frac{\partial \mathbf{c}_0}{\partial \mathbf{x}_j} \mathbf{p}_j \dots \sum_{j=0}^{M} \frac{\partial \mathbf{c}_N}{\partial \mathbf{x}_j} \mathbf{p}_j]$ and $\nabla \mathbf{r}$=$[(\frac{\partial \mathbf{r}_0}{\partial \mathbf{c}_0})^2 \dots (\frac{\partial \mathbf{r}_N}{\partial \mathbf{c}_N})^2]$:

$$\mathbf{g}_k = \sum_{i=0}^{N} \frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k} (\hat{\mathbf{u}} \odot \nabla \mathbf{r})_i \tag{16}$$

This avoids additional uncoalesced global memory reads to $\nabla \mathbf{r}$ in the CUDA kernels. Instead, we calculate $\hat{\mathbf{u}} \odot \nabla \mathbf{r}$ in a separate operation after the `applyJ` kernel and before `applyJT`. This also simplifies the kernels, since they now only need to compute $\frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k}$ instead of $\frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k}$. Therefore, the only runtime overhead of using the L1 and SSIM residual terms over the L2 residuals is the computation of $\nabla \mathbf{r}$. However, this can be efficiently computed using backpropagation (autograd) and is therefore not a bottleneck.

## B. Derivation Of Image Subsampling Weights

We subsample batches of images to decrease the size of the gradient cache (see Sec. 3.3). To combine the update vectors from multiple batches, we calculate their weighted mean, as detailed in Eq. (8). This weighted mean approximates the "true" solution to the normal equations (Eq. (4)), that does not rely on any image subsampling (and instead uses all available training images). When subsampling images, we split the number of total residuals into smaller chunks. In the following, we consider the case of two chunks (labeled as $_1$ and $_2$), but the same applies to any number of chunks. We re-write the normal equations (without subsampling) using the chunk notation as:

$$\begin{bmatrix} \mathbf{J}_1^T & \mathbf{J}_2^T \end{bmatrix} \begin{bmatrix} \mathbf{J}_1 \\ \mathbf{J}_2 \end{bmatrix} \Delta = \begin{bmatrix} \mathbf{J}_1^T & \mathbf{J}_2^T \end{bmatrix} \begin{bmatrix} \mathbf{F}_1(\mathbf{x}) \\ \mathbf{F}_2(\mathbf{x}) \end{bmatrix} \tag{17}$$

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

$$\Delta = (\mathbf{J}_1^T \mathbf{J}_1 + \mathbf{J}_2^T \mathbf{J}_2)^{-1} (\mathbf{J}_1^T \mathbf{F}_1(\mathbf{x}) + \mathbf{J}_2^T \mathbf{F}_2(\mathbf{x})) \tag{18}$$

In contrast, when we subsample images, we solve the normal equations separately and obtain two solutions:

$$\Delta_1 = (\mathbf{J}_1^T \mathbf{J}_1)^{-1} \mathbf{J}_1^T \mathbf{F}_1(\mathbf{x}) \tag{19}$$

$$\Delta_2 = (\mathbf{J}_2^T \mathbf{J}_2)^{-1} \mathbf{J}_2^T \mathbf{F}_2(\mathbf{x}) \tag{20}$$

We can rewrite Eq. (18) as a weighted mean of $\Delta_1$, $\Delta_2$:

$$\Delta = K^{-1} (\mathbf{J}_1^T \mathbf{J}_1)(\mathbf{J}_1^T \mathbf{J}_1)^{-1} (\mathbf{J}_1^T \mathbf{F}_1(\mathbf{x})) \tag{21}$$

$$+K^{-1} (\mathbf{J}_2^T \mathbf{J}_2)(\mathbf{J}_2^T \mathbf{J}_2)^{-1} (\mathbf{J}_2^T \mathbf{F}_2(\mathbf{x})) \tag{22}$$

$$= w_1 \Delta_1 + w_2 \Delta_2 \tag{23}$$

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. (8).

| Kernel | Runtime (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 5. **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.

## C. Detailed Runtime Analysis

We provide additional analysis of the CUDA kernels by running the `NVIDIA Nsight Compute` profiler. We provide results in Tab. 5 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).

Comparing the runtime of the `buildCache` and `applyJT` kernels reveals the advantage of our proposed *per-pixel-per-splat* parallelization pattern. Both compute the identical Jacobian-vector product, but the `buildCache` kernel relies on the *per-pixel* parallelization pattern of the original 3DGS rasterizer [23]. However, we compute the result 4.8x faster using the gradient cache in the `applyJT` kernel. We also note that the compute and memory throughput as well as the register count of both kernels are roughly similar. This signals that our kernel implementation is equally efficient, i.e., there are no inherent drawbacks using our proposed GPU parallelization scheme.

## 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 [23] are in Tab. 6. The comparisons against DISTWAR [12] are in Tab. 7. The comparisons against gsplat [48] are in Tab. 8. The comparisons against Taming-3DGS [32] are in Tab. 9. Our method shows consistent acceleration of the optimization runtime on all scenes, while achieving the same quality.

| Method | Scene | MipNeRF-360 [4] | | | |
|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) |
| 3DGS [23] | treehill | 0.631 | 22.44 | 0.330 | 1130 |
| + Ours | treehill | 0.633 | 22.57 | 0.334 | **836** |
| 3DGS [23] | counter | 0.905 | 28.96 | 0.202 | 1178 |
| + Ours | counter | 0.904 | 28.89 | 0.206 | **927** |
| 3DGS [23] | stump | 0.769 | 26.56 | 0.217 | 1234 |
| + Ours | stump | 0.774 | 26.67 | 0.218 | **895** |
| 3DGS [23] | bonsai | 0.939 | 31.99 | 0.206 | 1034 |
| + Ours | bonsai | 0.938 | 31.84 | 0.208 | **794** |
| 3DGS [23] | bicycle | 0.764 | 25.20 | 0.212 | 1563 |
| + Ours | bicycle | 0.765 | 25.30 | 0.218 | **1141** |
| 3DGS [23] | kitchen | 0.925 | 31.37 | 0.128 | 1389 |
| + Ours | kitchen | 0.924 | 31.21 | 0.128 | **1156** |
| 3DGS [23] | flowers | 0.602 | 21.49 | 0.340 | 1132 |
| + Ours | flowers | 0.600 | 21.52 | 0.344 | **819** |
| 3DGS [23] | room | 0.917 | 31.36 | 0.221 | 1210 |
| + Ours | room | 0.916 | 31.10 | 0.224 | **1004** |
| 3DGS [23] | garden | 0.862 | 27.23 | 0.109 | 1573 |
| + Ours | garden | 0.863 | 27.30 | 0.110 | **1175** |

| Method | Scene | Deep Blending [19] | | | |
|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) |
| 3DGS [23] | playroom | 0.901 | 29.90 | 0.247 | 1085 |
| + Ours | playroom | 0.905 | 30.24 | 0.246 | **861** |
| 3DGS [23] | drjohnson | 0.898 | 29.12 | 0.246 | 1359 |
| + Ours | drjohnson | 0.901 | 29.23 | 0.248 | **1040** |

| Method | Scene | Tanks & Temples [27] | | | |
|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) |
| 3DGS [23] | train | 0.811 | 21.95 | 0.209 | 636 |
| + Ours | train | 0.811 | 22.07 | 0.214 | **579** |
| 3DGS [23] | truck | 0.877 | 25.40 | 0.148 | 837 |
| + Ours | truck | 0.876 | 25.36 | 0.151 | **747** |

Table 6. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics against the 3DGS [23] baseline.

| Method | Scene | MipNeRF-360 [4] | | | |
|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) |
| DISTWAR [12] | treehill | 0.633 | 22.47 | 0.327 | 898 |
| + Ours | treehill | 0.635 | 22.54 | 0.332 | **669** |
| DISTWAR [12] | counter | 0.905 | 29.00 | 0.203 | 790 |
| + Ours | counter | 0.904 | 28.91 | 0.205 | **687** |
| DISTWAR [12] | stump | 0.771 | 26.60 | 0.216 | 1017 |
| + Ours | stump | 0.773 | 26.70 | 0.217 | **760** |
| DISTWAR [12] | bonsai | 0.939 | 32.13 | 0.206 | 677 |
| + Ours | bonsai | 0.938 | 31.92 | 0.208 | **578** |
| DISTWAR [12] | bicycle | 0.763 | 25.19 | 0.212 | 1333 |
| + Ours | bicycle | 0.764 | 25.26 | 0.218 | **971** |
| DISTWAR [12] | kitchen | 0.925 | 31.31 | 0.127 | 957 |
| + Ours | kitchen | 0.924 | 31.14 | 0.128 | **838** |
| DISTWAR [12] | flowers | 0.602 | 21.45 | 0.340 | 884 |
| + Ours | flowers | 0.596 | 21.48 | 0.348 | **671** |
| DISTWAR [12] | room | 0.916 | 31.41 | 0.221 | 803 |
| + Ours | room | 0.916 | 31.40 | 0.224 | **680** |
| DISTWAR [12] | garden | 0.862 | 27.23 | 0.109 | 1338 |
| + Ours | garden | 0.861 | 27.32 | 0.112 | **1023** |

| Method | Scene | Deep Blending [19] | | | |
|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) |
| DISTWAR [12] | playroom | 0.900 | 29.81 | 0.247 | 729 |
| + Ours | playroom | 0.905 | 30.24 | 0.246 | **586** |
| DISTWAR [12] | drjohnson | 0.898 | 29.13 | 0.247 | 953 |
| + Ours | drjohnson | 0.901 | 29.13 | 0.249 | **758** |

| Method | Scene | Tanks & Temples [27] | | | |
|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) |
| DISTWAR [12] | train | 0.812 | 22.05 | 0.209 | 504 |
| + Ours | train | 0.810 | 22.10 | 0.216 | **440** |
| DISTWAR [12] | truck | 0.877 | 25.29 | 0.148 | 698 |
| + Ours | truck | 0.877 | 25.28 | 0.150 | **635** |

Table 7. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics against the DISTWAR [12] baseline.

| Method | Scene | MipNeRF-360 [4] | | | |
|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) |
| gsplat [48] | treehill | 0.634 | 22.44 | 0.324 | 973 |
| + Ours | treehill | 0.635 | 22.54 | 0.332 | **701** |
| gsplat [48] | counter | 0.908 | 28.99 | 0.201 | 903 |
| + Ours | counter | 0.904 | 28.91 | 0.205 | **762** |
| gsplat [48] | stump | 0.769 | 26.53 | 0.218 | 1097 |
| + Ours | stump | 0.774 | 26.70 | 0.217 | **793** |
| gsplat [48] | bonsai | 0.937 | 31.95 | 0.208 | 783 |
| + Ours | bonsai | 0.938 | 31.92 | 0.208 | **646** |
| gsplat [48] | bicycle | 0.765 | 25.21 | 0.206 | 1398 |
| + Ours | bicycle | 0.765 | 25.26 | 0.218 | **988** |
| gsplat [48] | kitchen | 0.926 | 31.17 | 0.128 | 1086 |
| + Ours | kitchen | 0.924 | 31.14 | 0.128 | **921** |
| gsplat [48] | flowers | 0.600 | 21.53 | 0.338 | 965 |
| + Ours | flowers | 0.601 | 21.48 | 0.348 | **709** |
| gsplat [48] | room | 0.920 | 31.48 | 0.219 | 913 |
| + Ours | room | 0.916 | 31.39 | 0.224 | **753** |
| gsplat [48] | garden | 0.869 | 27.48 | 0.105 | 1462 |
| + Ours | garden | 0.861 | 27.32 | 0.112 | **1085** |

| Method | Scene | Deep Blending [19] | | | |
|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) |
| gsplat [48] | playroom | 0.907 | 29.89 | 0.248 | 799 |
| + Ours | playroom | 0.904 | 30.90 | 0.247 | **626** |
| gsplat [48] | drjohnson | 0.901 | 29.16 | 0.244 | 1040 |
| + Ours | drjohnson | 0.901 | 29.07 | 0.251 | **805** |

| Method | Scene | Tanks & Temples [27] | | | |
|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) |
| gsplat [48] | train | 0.811 | 21.64 | 0.209 | 558 |
| + Ours | train | 0.809 | 22.09 | 0.216 | **381** |
| gsplat [48] | truck | 0.880 | 25.35 | 0.149 | 735 |
| + Ours | truck | 0.877 | 25.28 | 0.150 | **447** |

Table 8. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics against the gsplat [48] baseline.

| Method | Scene | MipNeRF-360 [4] | | | | |
|---|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) | #G (M) |
| Taming [32] | treehill | 0.625 | 23.00 | 0.385 | 479 | 0.78 |
| + Ours | treehill | 0.623 | 23.03 | 0.332 | **381** | 0.78 |
| Taming [32] | counter | 0.896 | 28.59 | 0.223 | 646 | 0.31 |
| + Ours | counter | 0.894 | 28.51 | 0.205 | **537** | 0.31 |
| Taming [32] | stump | 0.734 | 25.96 | 0.292 | 405 | 0.48 |
| + Ours | stump | 0.733 | 25.98 | 0.217 | **315** | 0.48 |
| Taming [32] | bonsai | 0.934 | 31.73 | 0.221 | 634 | 0.41 |
| + Ours | bonsai | 0.932 | 31.64 | 0.208 | **504** | 0.41 |
| Taming [32] | bicycle | 0.716 | 24.78 | 0.295 | 485 | 0.81 |
| + Ours | bicycle | 0.709 | 24.75 | 0.218 | **376** | 0.81 |
| Taming [32] | kitchen | 0.918 | 30.85 | 0.141 | 722 | 0.48 |
| + Ours | kitchen | 0.918 | 30.84 | 0.128 | **597** | 0.48 |
| Taming [32] | flowers | 0.554 | 21.00 | 0.407 | 465 | 0.57 |
| + Ours | flowers | 0.549 | 20.99 | 0.348 | **365** | 0.57 |
| Taming [32] | room | 0.906 | 31.12 | 0.251 | 621 | 0.22 |
| + Ours | room | 0.906 | 31.21 | 0.224 | **521** | 0.22 |
| Taming [32] | garden | 0.852 | 27.24 | 0.128 | 638 | 1.90 |
| + Ours | garden | 0.852 | 27.25 | 0.112 | **483** | 1.90 |

| Method | Scene | Deep Blending [19] | | | | |
|---|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) | #G (M) |
| Taming [32] | playroom | 0.900 | 30.29 | 0.278 | 419 | 0.18 |
| + Ours | playroom | 0.902 | 30.41 | 0.280 | **324** | 0.18 |
| Taming [32] | drjohnson | 0.899 | 29.38 | 0.269 | 475 | 0.40 |
| + Ours | drjohnson | 0.899 | 29.40 | 0.271 | **370** | 0.40 |

| Method | Scene | Tanks & Temples [27] | | | | |
|---|---|---|---|---|---|---|
| | | SSIM↑ | PSNR↑ | LPIPS↓ | Time (s) | #G (M) |
| Taming [32] | train | 0.815 | 22.18 | 0.205 | 411 | 0.36 |
| + Ours | train | 0.802 | 22.28 | 0.241 | **328** | 0.36 |
| Taming [32] | truck | 0.879 | 25.40 | 0.146 | 514 | 0.27 |
| + Ours | truck | 0.862 | 25.16 | 0.178 | **292** | 0.27 |

Table 9. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics against the Taming-3DGS [32] baseline. Additionally, we include the number of Gaussians in millions (#G (M)) that we obtained using the default hyperparameters for "budgeting".