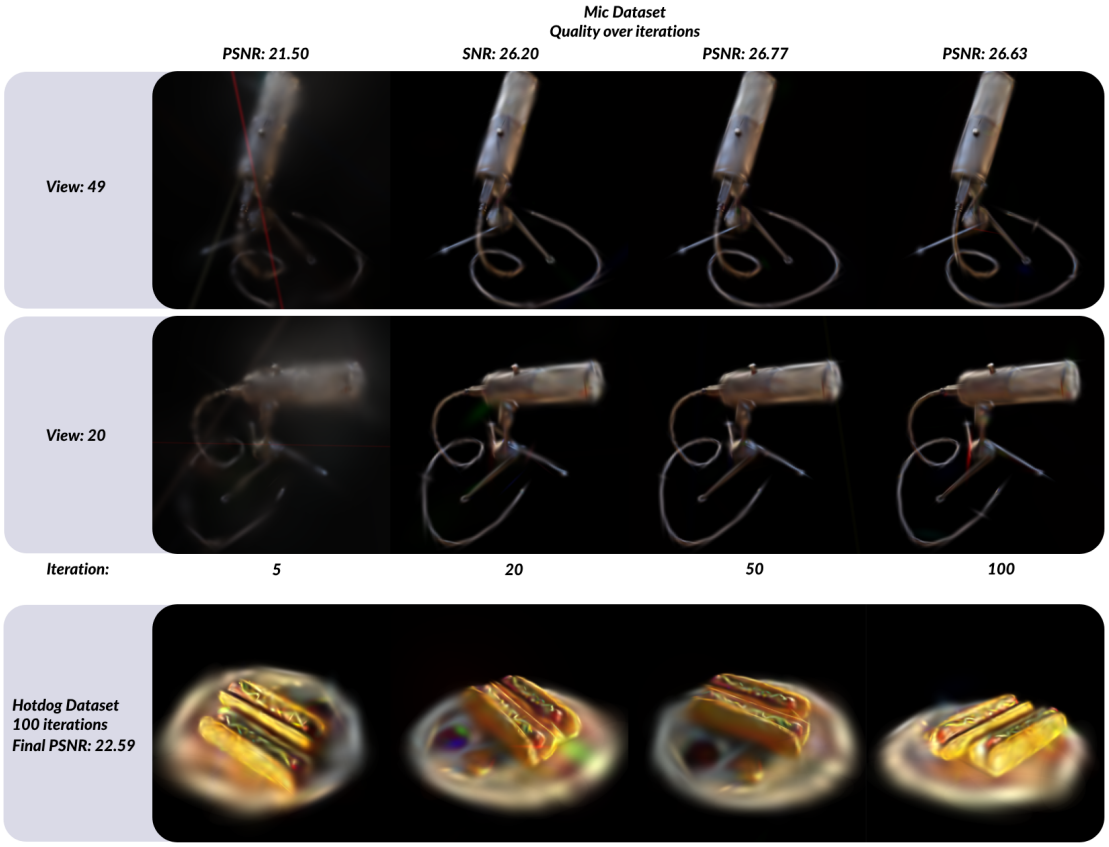
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Gauss-Newton Optimizer for 3D Gaussian Splatting Reconstruction

Master's thesis in Computer science and engineering

Tom Alsrup
Rasmus Almryd

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Gauss-Newton Optimizer for 3D Gaussian Splatting Reconstruction

Tom Alsrup
Rasmus Almryd

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Gauss-Newton Optimizer for 3D Gaussian Splatting Reconstruction

Tom Alsrup, 2025.
Rasmus Almryd, 2025.

Supervisor: Ulf Assarsson, CSE
Examiner: Erik Sintorn, CSE

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration showing the results of training 3DGS for a set number of iterations using our Gauss-Newton optimizer. The scenes are from the Nerf Synthetic Dataset.

Gauss-Newton Optimizer for 3D Gaussian Splatting Reconstruction

Tom Alsrup
Rasmus Almryd
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Recent advancements in neural rendering have enabled highly realistic and efficient representations of 3D scenes from multi-view images. Among these, 3D Gaussian Splatting (3DGS) has emerged as a promising technique that represents scenes using a collection of 3D Gaussian primitives. Since it is a new and active field of research, progress has been made in terms of reducing reconstruction time. We present a new method of optimizing 3DGS by replacing ADAM with a Gauss-Newton (GN) optimizer integrated with the differentiable rasterizer. To save on memory, we introduce custom CUDA kernels that cache radiance and transmittance instead of explicitly storing all gradients. In addition, we introduce a sparsity-aware memory scheme that allows us to store more relevant data while minimizing waste. In each GN iteration, we compute update directions from multiple image subsets using several kernels and aggregate them through a weighted mean. Finally, a line search algorithm is used to determine the optimal update step length based on the sum of squared residuals objective function. Our optimizer converges significantly faster than ADAM per iteration, but comes with a high memory footprint. Due to GN's computational complexity, it requires more execution time to reach a similar quality level as the original ADAM optimizer. Our GN optimizer demonstrates encouraging results during training, and with further improvements and research detailed in this paper, we believe it could become a strong competitor to ADAM.

Keywords: 3D Gaussian splatting, 3DGS, Gauss-newton, ADAM, 3D Reconstruction, Preconditioned Conjugate Gradient method, PCG.

Acknowledgements

Special thanks to our supervisor, Ulf Assarsson, for helping us define the research topic and for being there to support us whenever needed. His feedback and perspective were helpful throughout the project, particularly during troubleshooting.

Contents

List of Figures

List of Tables

1

Introduction

Gaussian splatting is one of several methods of reconstructing scenes from multiple view directions using a limited amount of 2D images. These methods of creating virtual environments of scenes from images are called Novel View Synthesis (NVS). Various 3D scene representations have been developed to support NVS [1][2][3]. Among them, 3D Gaussian Splatting (3DGS) [3] represents the scene as a collection of 3D Gaussians, offering real-time rendering and high-quality image synthesis. Its ability to produce smooth, high-quality representations faster than previous methods [4] makes it valuable in applications ranging from medical diagnosis to virtual reality, computer-aided design, and interactive media [4]. As datasets grow in size and complexity and the demand for real-time interactive visualization increases, the computational cost of 3D Gaussian splatting increases and becomes a significant challenge. Accelerating this technique is important to enable real-time rendering for interactive applications, improve scalability and improve energy efficiency. Speeding up reconstruction time can broaden accessibility, allowing high-quality 3D visualization to be deployed on less powerful hardware, thereby making it more available across various domains.

In 3DGS, the scene is optimized from a set of images through a differentiable rasterization process. Optimizing 3DGS requires a densely sampled set of images and can take an hour or more for high-resolution real-world datasets with many images based on hardware. Reducing the optimization time is important for enabling faster reconstruction. Existing methods accelerate this process by enhancing different aspects of the optimization process. 3DGS utilizes a tile-based differentiable rasterizer, where Gaussians are updated in each iteration using gradient descent by backpropagating the rendering loss. 3DGS also gradually increases the number of Gaussians throughout optimization in a process known as densification. This is achieved by accumulating positional gradients over multiple iterations and using them to guide the splitting and pruning of Gaussians.

Despite the performance improvements from the aforementioned techniques, the optimization process remains computationally intensive, requiring many thousands of gradient descent iterations to converge. To further reduce reconstruction time, our aim is to replace the commonly used ADAM optimizer [5] with a Gauss-Newton (GN) [6] non linear-least squares approach. This is inspired by previous works [7][1] where gradient descent based optimizers were replaced with least squares solvers which resulted in reduced reconstruction time. The Taming 3DGS [8] version of the 3DGS implementation will be used as a baseline and a starting point for our GN implementation.

2

Related Work

2.1 Gauss-Newton optimizer for explicit radiance fields.

Our approach draws inspiration from the work of Rasmuson *et al.* [7], who demonstrated the effectiveness of Gauss-Newton optimization in solving volumetric reconstruction as a non-linear least-squares problem. Their method, PERF, replaces the typical Multilayer Perceptron (MLP) based formulation of NeRF with an explicit voxel grid representation and solves for color and opacity using Gauss-Newton with the Preconditioned Conjugate Gradient (PCG) method. A key insight from their work is that Gauss-Newton converges significantly faster than first-order optimization methods like ADAM, allowing high-quality reconstructions to be achieved faster and with fewer iterations.

We extend this reasoning to 3D Gaussian Splatting, where the optimization problem similarly involves refining a large set of scene parameters based on image observations. We believe that Gauss-Newton could be a promising alternative to first-order methods in 3D Gaussian Splatting as well. By incorporating second-order information, Gauss-Newton has the potential to provide more stable and efficient updates compared to ADAM. Inspired by their results, we aim to explore whether a Gauss-Newton optimizer could provide a speedup in reconstruction time for 3D Gaussian Splatting.

2.2 Taming 3DGS: High-Quality Radiance Fields with Limited Resources

Since the original 3DGS implementation [3] was published there have been improvements made by other researchers. One of these is Taming 3DGS by Mallick *et al.* [8], where several improvements have been made. The authors introduce an improved densification process that selectively adds Gaussian primitives, focusing on those that enhance reconstruction quality with the aim of reducing redundancy and memory consumption. Optimizations to the gradient calculations and alternative parallelization schemes have also been introduced that further accelerate training time. Since our aim is to reduce reconstruction time, we chose to use this version as our baseline to assess whether it can be further improved.

2.3 Gaussian splatting using Levenberg-Marquardt algorithm.

Another least squares approach for optimizing 3D Gaussian representations was demonstrated to be effective by Höllein *et al.* [9], who applied the Levenberg-Marquardt (LM) algorithm to improve the convergence and quality of 3D Gaussian Splatting. Their method reformulated the optimization process as a non-linear least-squares problem, leveraging LM to balance the advantages of both Gauss-Newton and gradient-based approaches. Their results showed improved reconstruction accuracy and faster convergence compared to first-order optimizers, suggesting that second-order methods can be highly effective in this domain.

In addition to replacing first-order gradient descent with LM, they modified the parallelization strategy to better suit the structure of the Hessian approximation, improving computational efficiency on Graphical Processing Units (GPUs). Their approach restructured the optimization to minimize redundant computations and leverage batch processing. These enhancements resulted in improved convergence rates and reconstruction quality.

Given these findings, we believe that Gauss-Newton could offer similar benefits for 3DGS. While LM introduces a damping factor and trust region to improve stability in ill-conditioned scenarios, Gauss-Newton retains its efficiency by leveraging approximate second-order information without requiring full Hessian computation. Inspired by their success, we aim to explore whether Gauss-Newton can provide comparable advantages for optimizing 3D Gaussian parameters, particularly in terms of convergence speed and reconstruction quality.

3

Background

3.1 3D Gaussian Splatting

The 3D Gaussian Splatting method, as described in [3], is a novel and efficient approach to real-time radiance field rendering. At its core, the system represents 3D scenes as a collection of anisotropic 3D Gaussian distributions. These Gaussians serve as the foundational elements for approximating the scene's radiance field, which encodes the density and color of light at every point in space. Unlike traditional methods that rely on voxel grids, point clouds, or meshes, 3D Gaussians offer a more flexible and adaptive representation. Each Gaussian is defined by a total of 59 parameters, which are used to capture its position, shape, orientation, appearance, and transparency in 3D space. These parameters are divided into two main groups: 14 parameters for position, rotation, scaling, and opacity, and 45 parameters for spherical harmonics. The appearance of a Gaussian, including its color and view-dependent lighting effects, is modeled using spherical harmonic coefficients. A spherical harmonic is a mathematical tool here used to represent how light interacts with surfaces from different viewing angles [10]. In the implementation on which this paper is based [3], 3rd-order spherical harmonics are used, requiring 16 coefficients per color channel (RGB). Since there are three color channels, this would initially require 48 parameters. However, the first coefficient represents the direct color and is shared between all channels, reducing the total number of SH parameters to 45. These coefficients enable the Gaussian to accurately capture complex lighting effects, such as reflections and shading, from various viewpoints.

The rendering process, as seen in Figure 3.1, involves projecting these 3D Gaussians onto the 2D image plane using a technique called splatting. During splatting, the Gaussians are rasterized to produce the final image. This process is differentiable, meaning that the gradients of the rendering operation can be computed with respect to the Gaussian parameters. This differentiability enables gradients to be used in optimization. The optimization step allows the Gaussians to be fine-tuned to minimize the difference between the rendered images and the ground truth. The optimization in the original implementation is performed using the stochastic gradient descent method ADAM, which involves minimizing a photometric loss function that measures the difference between the rendered and ground-truth images.

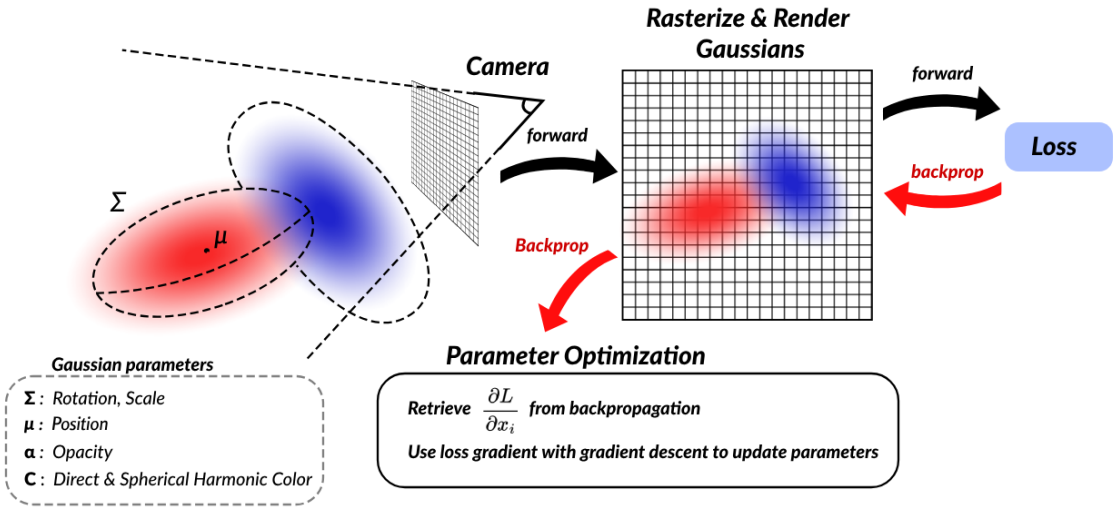


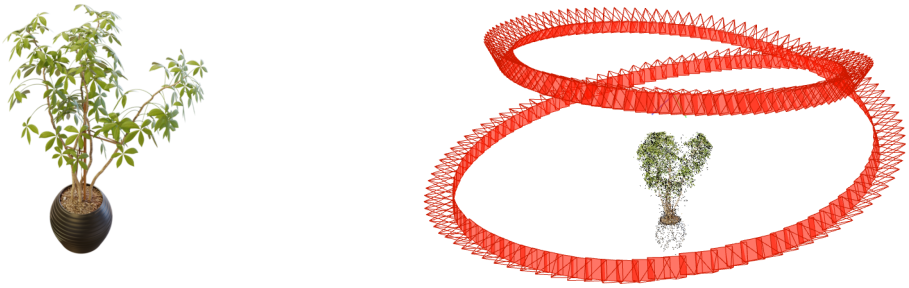
Figure 3.1: Overview of the differentiable rasterizer and parameter optimization for the original Gaussian splatting implementation with ADAM by *et al.* [3]. The $\frac{\partial L}{\partial x}$ is the derivative of the Loss w.r.t all Gaussian parameters x_i.

3D Gaussian Splatting offers several advantages over traditional methods. The flexibility of anisotropic Gaussians allows them to better conform to the local geometry and texture of the scene, improving representation accuracy. It is computationally efficient, enabling high-quality scene representation with fewer Gaussians. Additionally, the method takes advantage of modern GPU hardware for real-time rendering of dynamic 3D scenes.

3.2 Scene Preparation

Throughout the development phase, larger datasets such as those included in [3][9] required a substantial number of Gaussians to represent the environment. To accelerate development and simplify testing, we opted to use scenes from the NeRF Synthetic dataset, as these are less complex and require fewer Gaussians.

These datasets consist of multiple images of an object, with camera parameters calibrated using Structure-from-Motion (SfM) to generate an initial point cloud for Gaussian placement. A simple script was created to reorder the images, which were then processed using COLMAP, a general-purpose SfM pipeline [11][12] to generate the initial point cloud. To produce a well-defined point cloud, we used the pinhole camera model option in COLMAP, which yielded a sufficient number of points, as shown in Figure 3.2.



(a) One of many images of the object. (b) Resulting point cloud.

Figure 3.2: Point cloud generated for the ficus dataset using 200 images from different camera directions seen in (b).

3.3 Optimizer

The ADAM optimizer is a widely used gradient-based optimization algorithm, particularly in machine learning applications, as well as in advanced techniques such as Neural Radiance Fields (NeRF) and 3D Gaussian splatting. ADAM [5] adapts the learning rate for each parameter individually by maintaining estimates of the first moment (mean) and second moment (variance) of the gradients. This allows it to efficiently handle noisy or sparse gradients, making it well-suited for large datasets and complex models. However, while ADAM performs well in many scenarios, it primarily uses first-order information, which can slow convergence in problems with complex, nonlinear structures.

On the other hand, the Gauss-Newton method [6] is a second-order optimization technique commonly used for non-linear least squares problems. It approximates the Newton method by ignoring the second-order derivative terms (Hessian) and instead uses the Jacobian matrix to estimate them, which leads to faster convergence near a minimum. The Gauss-Newton method incorporates more information, potentially making it more efficient in optimizing certain complex problems compared to first-order methods like ADAM. However, it is computationally expensive as it requires the calculation of the Jacobian and solving linear systems at each iteration.

In scenarios where the problem is close to a minimum and the cost of computing the Jacobian is manageable, Gauss-Newton can outperform ADAM due to its ability to compute a more exact update step. By utilizing second-order information and not just gradient direction, Gauss-Newton typically converges more quickly, especially in non-linear models, whereas ADAM may require more iterations to reach an optimal solution. The computational complexity of GN can make it harder to apply, and for large equation systems requires a significant amount of memory on the GPU to store the intermediate values during computations.

3.4 Residuals and Objective Function

Gauss-Newton and similar methods require more information than just gradient direction which is used in gradient descent optimization methods. Gauss-Newton utilizes residuals and their gradients w.r.t model parameters to approach a solution. We thus change the original implementation to maintain a residual for each pixel in each (RGB) color channel. This is equal to $3 * W * H$ total residuals where W and H are the input image width and height in pixels. For comparison, the original implementation uses the average of the RGB values across color channels for each pixel for its total loss. By including and storing more residuals memory consumption will increase as a consequence.

From these residuals, the Gauss-Newton method iteratively tunes the model parameters such that the sum of squared errors is minimized as shown below:

$$S(x) = \sum_{i=1}^{m} r_i(x)^2. \tag{3.1}$$

where r_i are the residuals. In the 3D Gaussian Splatting program the residuals are created by comparing the values in each color channel for each pixel in the ground truth image with the rendered image. This value is then used in the subsequent iterations to gradually converge to an improved image. We employ an L2 loss function for this purpose, which aligns with the structure of Equation 3.1 and is defined as:

$$L_2 = \sum_{i=1}^{n} (r_i^2), r_i = y_g - y_r \tag{3.2}$$

where y_g is the ground truth image values and y_r is the new image rendered from the Gaussian splats.

3.5 Gauss Newton Algorithm

The Gauss-Newton method [6] is an iterative optimization algorithm used to solve nonlinear least squares problems. As previously mentioned, it approximates the Hessian matrix, making it computationally efficient for problems where the residual function is nearly linear. Given a nonlinear function $\mathbf{r}(\mathbf{x})$, the Gauss-Newton update step aims to minimize the sum of squared residuals by solving a linear system at each iteration. The update step avoids explicit matrix inversion by solving a normal equation involving the Jacobian matrix $\mathbf{J}$ of the residuals. The Jacobian contains the first-order partial derivatives of the residuals with respect to the Gaussian parameters. It provides the necessary information to construct the linear system. The algorithm is further explained in **Algorithm 1**.

Algorithm 1 Gauss-Newton Update Step

1: Starting with a value x_0, the update function for the algorithm is:

$$x_{i+1} = x_i - (J_r^T J_r)^{-1} J_r^T r(x_i) \tag{3.3}$$

where x_i represent all color and opacity values.

2: Compute the Jacobian J_r

$$J_r = \begin{bmatrix} \frac{\partial r_1}{\partial x_1} & \cdots & \frac{\partial r_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial r_m}{\partial x_1} & \cdots & \frac{\partial r_m}{\partial x_n} \end{bmatrix}$$

3: Compute update direction:

$$\mathbf{\Delta} = -(\mathbf{J}^\top \mathbf{J})^{-1} \mathbf{J}^\top \mathbf{r}$$

4: Update estimate: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{\Delta}$

5: Check convergence, if not satisfied, repeat.

3.6 Update Step

The update step of the GN algorithm requires an expensive matrix inversion of the large and sparse square matrix $\mathbf{J_r}^T \mathbf{J_r}$. Instead of performing this operation explicitly, the step calculation is rewritten in the following way for a step $\mathbf{\Delta}$:

$$\mathbf{\Delta} = -(\mathbf{J_r}^T \mathbf{J_r})^{-1} \mathbf{J_r}^T \mathbf{r} \tag{3.4}$$

$$\Longleftrightarrow$$

$$\mathbf{J_r}^T \mathbf{J_r} \mathbf{\Delta} = -\mathbf{J_r}^T \mathbf{r} \tag{3.5}$$

With $A = \mathbf{J_r}^T \mathbf{J_r}$, $x = \mathbf{\Delta}$, and $b = -\mathbf{J_r}^T \mathbf{r}$, this amounts to a linear system of equations of the form:

$$Ax = b \tag{3.6}$$

In this instance the matrix $\mathbf{A}$ is very large and sparse. To solve this linear system of equations the Preconditioned Conjugate Gradient algorithm (PCG) [6] is used which can be seen in **Algorithm 2**.

Algorithm 2 The iterative Preconditioned Conjugate Gradient algorithm, with precondition matrix $\mathbf{M}$.

$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$R_{\text{prev}} = \mathbf{r}_0^\top \mathbf{r}_0$
$\mathbf{z}_0 = \mathbf{M}^{-1}\mathbf{r}_0$
$\mathbf{p}_0 = \mathbf{z}_0$
$k = 0$
while true **do**
    $\alpha_k = \frac{\mathbf{r}_k^\top \mathbf{z}_k}{\mathbf{p}_k^\top \mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$
    $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$
    $R = \mathbf{r}_{k+1}^\top \mathbf{r}_{k+1}$
    if $R < EPS$ **or** $R/R_{\text{prev}} > 0.85$ **then**
        return $\mathbf{x}_{k+1}$
    end if
    $R_{\text{prev}} = R$
    $\mathbf{z}_{k+1} = \mathbf{M}^{-1}\mathbf{r}_{k+1}$
    $\beta_k = \frac{\mathbf{r}_{k+1}^\top \mathbf{z}_{k+1}}{\mathbf{r}_k^\top \mathbf{z}_k}$
    $\mathbf{p}_{k+1} = \mathbf{z}_{k+1} + \beta_k \mathbf{p}_k$
    $k = k + 1$
end while

The variables r, z, p and x are temporary variables. As previously stated, the matrix A is very large which makes it unfeasible to create explicitly. This is the case since it will consume memory in the range of Terabytes depending on problem size. To circumvent this, the following identity is employed, as described in [7], to utilize the parallelization capabilities of the GPU:

$$Ap = J_r^T J_r p = \sum_i \nabla r_i (\nabla r_i^T p) \tag{3.7}$$

3.7 Preconditioner

The preconditioner used in the PCG algorithm is the following Jacobi preconditioner:

$$\mathbf{M} = diag(\mathbf{J}_r^\top \mathbf{J}_r) = \left[\sum \left(\frac{\partial \mathbf{r}_i}{\partial x_1} \right)^2, \sum \left(\frac{\partial \mathbf{r}_i}{\partial x_2} \right)^2, \ldots, \sum \left(\frac{\partial \mathbf{r}_i}{\partial x_n} \right)^2 \right] \tag{3.8}$$

The Jacobi preconditioner is used in the PCG algorithm because of its simplicity and efficiency in improving convergence. It approximates the inverse of the system's coefficient matrix by using only the diagonal elements, which makes it computationally inexpensive to apply. The preconditioner scales the system to reduce its condition number, theoretically leading to faster convergence in PCG. The Jacobi preconditioner's low computational cost makes it suitable when dealing with large, sparse systems where more complex preconditioners may be more computationally intensive.

3.8 Line search

The update step, calculated in equation 3.4, is in theory the correct descent direction. However, there is no guarantee that applying the update step will decrease the objective function in each iteration [13]. Therefore, it is often more efficient to take smaller steps in that same direction using a backtracking line search algorithm. This is shown in **Algorithm 3**, which is inspired by Rasmuson *et al* [7]. The Error function, which combines the loss of all batches after applying the update step Δ scaled by α, is used to evaluate the effectiveness of the scaling factor α.

Algorithm 3 Backtracking line search algorithm.

$\alpha = 1.0$
$\gamma = 0.7$
$best\_\alpha$
$\mu_{\text{prev}} = Floating\_point\_max\_val$ // Want a large number.
while $\alpha > threshold$ **do**
    $\mu = \text{Error}(\alpha)$ // From objective function.
    if $\mu < \mu_{\text{prev}}$ **then**
        $best\_\alpha = \alpha$
    end if
    $\alpha = \alpha\gamma$
    $\mu_{\text{prev}} = \mu$
end while

4

Method

The Gauss-Newton optimizer is implemented in CUDA with several distinct kernels and the line search algorithm is built in PyTorch as an addition tied to the original [3] render pipeline. Changes have also been made in the differential rasterizer to support the use of GN and enable several optimizations, including those for the backward and forward passes. Figure 4.1 visualizes our pipeline which starts off with the creation of the SfM point cloud used to initialize Gaussian placement. ADAM is then used to quickly progress the reconstruction for a set number of iterations. In the second stage, GN is employed using multiple views over several batches to converge to the final result. The update step is computed using a weighting scheme across all batches within a GN iteration. The update step is then evaluated via a line search algorithm and the objective function to determine the optimal step length across all parameters. This approach is inspired by Höllein **et al.** [9], who found that using a least squares solver was more efficient for refining the final scene convergence than for the initial stages of reconstruction.

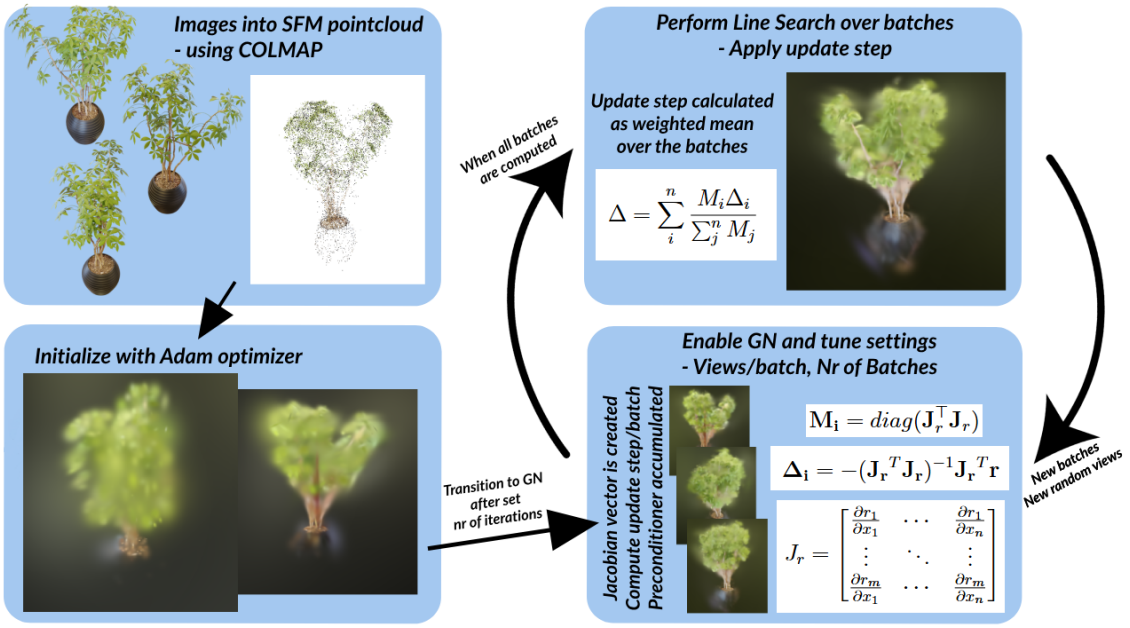


Figure 4.1: **Method Overview** The 3DGS optimization is split in two stages. Initialization for all Gaussians is made using the original ADAM optimizer. In the second stage Gauss Newton is applied to finish the optimization from multiple views.

4.1 Multi-view Optimization

The original Gaussian Splatting implementation [3] used the ADAM optimizer, which optimizes one view of the scene per iteration. In contrast, the Gauss-Newton method benefits from optimizing over multiple views in each iteration, which improves convergence[3][14]. However, this increases the execution time of the GN optimizer, which may offset the performance improvements gained from better convergence. This also increases cache size, as each view must store its own accumulated radiance and transmittance.

4.2 Image Subsampling Scheme

As noted in Section 4.1, optimizing more views per iteration improves the convergence stability of the GN optimizer. However, scaling the number of views indefinitely is infeasible due to VRAM constraints. To address this, we implemented a strategy similar to that proposed in [9], in which each optimization step is divided into smaller view batches. For each batch, the PCG solver is executed, and both the update vector Δ_i and the preconditioner M_i are stored. Here, n is the total number of batches. A weighted mean is then applied across all Δ_i to obtain the final update vector Δ, as shown in Equation 4.1. The preconditioners act as weights for the update vectors to balance their influence on the Gaussian parameters. They reflect how much each parameter in a batch affects color contributions across views.

$$\Delta = \sum_{i}^{n} \frac{M_i \Delta_i}{\sum_j^n M_j} \tag{4.1}$$

During experimentation, dividing views into smaller batches proved significantly more memory-efficient than caching all views simultaneously, reducing memory usage by several orders of magnitude.

4.3 CUDA implementation

Both the original Gaussian Splatting implementation [3] and the optimized version by Mallick *et al.* [8] use either per-pixel or per-splat parallelization when calculating gradient values. However, implementing the Gauss-Newton optimizer requires computing the nonzero elements of the Jacobian matrix. Therefore, we adopt a similar approach to [9], using a per-pixel-per-splat parallelization scheme.

Storing the full Jacobian is infeasible. Each Gaussian has 59 parameters, and scenes may contain several hundred thousand Gaussians. All must be considered when computing the partial derivatives of each residual with respect to each parameter. To address this, we create a cache to store only the accumulated radiance and transmittance values, rather than all gradients. This caching is applied only to pixels affected by Gaussians (see Figure 4.2). Due to the use of multi-view optimization, radiance

and transmittance values for all views are stored together in the same cache. Gradients are calculated on-demand during GN optimization, as recalculating them on the GPU is more efficient and requires less VRAM than storing and retrieving them from memory. During rasterization, contribution values are computed to determine where each cache entry should be stored. These values indicate how many pixels each Gaussian influences. During the backward pass, they are stored in Gaussian order to ensure coalesced memory access [15].

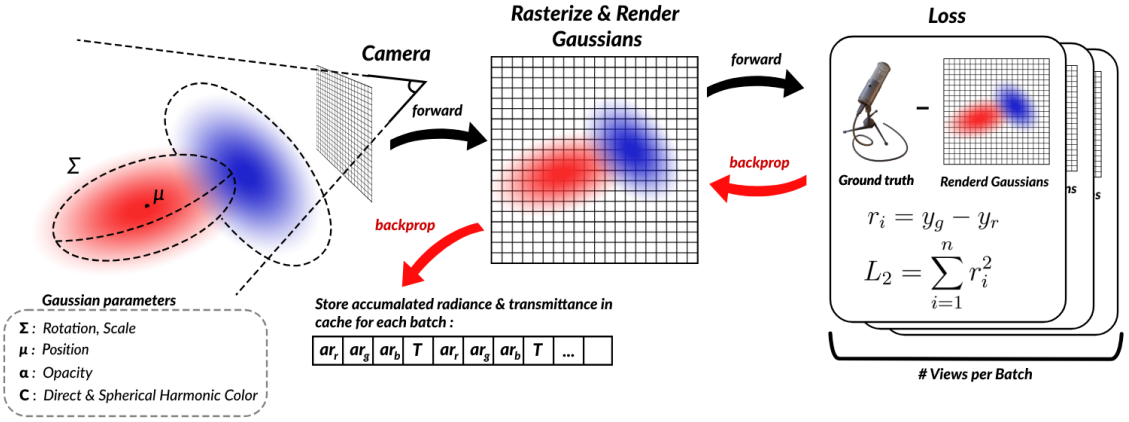


Figure 4.2: Overview of the reworked differentiable rasterizer. Rather than directly computing gradient values, it stores relevant accumulated radiance and transmittance in a cache for later use in the PCG algorithm.

This cache is then used in the PCG algorithm to generate the update step for the Gauss-Newton optimizer, which is implemented using four main CUDA kernels. A common denominator between all kernels is that they use the cache to retrieve radiance and transmittance from the backward-pass to compute the entries in the Jacobian. Below are brief descriptions of the kernels:

Jr: Each thread performs a matrix-vector product of the gradient values from the Jacobian with the corresponding residual value and atomically adds it to the correct parameter position.

Ap: Ap is not a kernel itself, but describes two separate kernels residual_dot_sum and sum_residuals used to split up the calculation in equation 3.7. First, the sum of the dot products $\nabla r_i^T p$ is computed. This result is then used to calculate the scalar-vector product with ∇r_i.

diagJTJ: The **diagJTJ:** kernel computes the diagonal entries required for the Jacobi preconditioner using the per-pixel-per-splat parallel scheme. Each thread squares its computed gradients and atomically adds the result to the corresponding entry in the output vector.

4.4 Evaluation metrics

PSNR (Peak Signal-to-Noise Ratio), SSIM (Structural Similarity Index Measure), and LPIPS (Learned Perceptual Image Patch Similarity) are commonly used metrics for evaluating image quality. PSNR [16] is a mathematical metric that measures the ratio between the maximum possible pixel value and the mean squared error (MSE) between a reference image (in this case, the ground truth) and the rendered result. A higher PSNR value generally indicates greater similarity, although it does not account for perceptual quality. SSIM [16] focuses on the structural similarity between images, evaluating luminance, contrast, and structure across local regions. This aligns more closely with human visual perception and emphasizes structural similarity rather than pixel-wise accuracy. LPIPS goes a step further by leveraging deep neural networks pretrained on image classification tasks [17]. It computes feature activations at various layers and measures their differences to evaluate perceptual similarity. This allows LPIPS to capture perceptual differences, providing a metric that should correlate more effectively with human perception. We use these metrics alongside qualitative visual inspection to assess the final reconstruction quality.

5

Results

In this chapter we will use the Taming-3DGS implementation of Gaussian splatting as a baseline to compare our Gauss-Newton optimizer. We use Taming-3dgs since it by itself provides a speedup over the original version of 3DGS and is the base to our implementation [8].

The results in this section are produced using a Nvidia RTX 3070 GPU with 8GB's of VRAM. Due to the limited VRAM and the fact that storing the cache for gauss newton commands significant memory space we have capped the amount of Gaussians to 500. While this number could potentially be increased, the cache size varies unpredictably between iterations and across datasets. To ensure consistent execution, we fixed the number of Gaussians at 500. This will signficantly impact the reconstruction quality and can therefore not be directly compared to the results in Taming-3DGS [8] and the Levenberg-Marquardt implementation [9]. Additionally, the densification step from [3] has been disabled to maintain a fixed number of 500 Gaussians.

5.1 Reconstruction Quality

In the Figures 5.1, 5.3 and 5.5 the result of optimizing scenes from the nerf syntehtic dataset using our GN implementation and baseline ADAM for 20 iterations is displayed. In order to make the comparison fair between the two methods, no spherical harmonics were considerd for this test since ADAM can have problem with convergence if SH is used to early in the traning process [3]. GN converges more per iteration to a resemblance of the original image than ADAM. However since each GN iteration processes more information and more views over several batches, each iteration takes significantly longer time to execute than ADAM. This result is similar to what was discovered by Höllein *et al.*[9] where using the Levenberg-Marquardt algorithm from the start yielded no improvements to reconstruction time. To determine the rate of improvement per iteration, the PSNR metric is displayed for each dataset. In Figure 5.1 the ficus dataset is seen from four viewpoints. Whilst better overall quality is achieved by GN, it takes significantly longer to process 20 iterations. PSNR progression per iteration for the ficus dataset is displayed in Figure 5.2 where GN improves each iteration whilst ADAM worsens. This trend is only true in the early iterations for ADAM and then changes to steady improvement in PSNR.

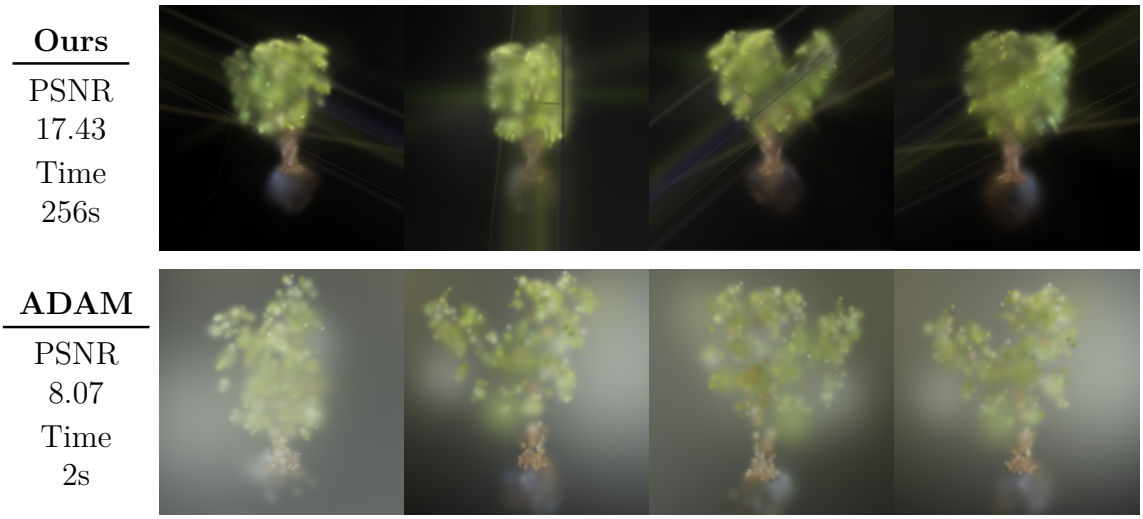


Figure 5.1: Results of reconstruction after 20 iterations for the ficus dataset. To the left reconstruction times and PSNR-scores are shown for both GN and ADAM.

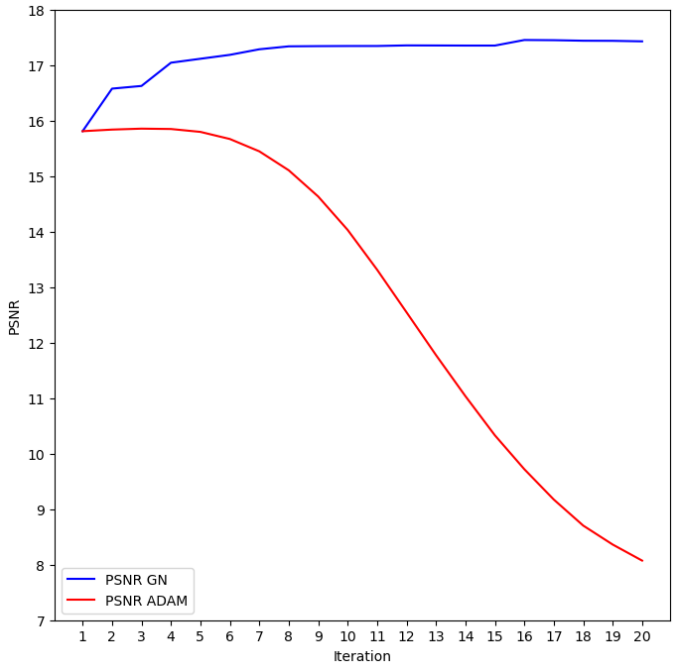


Figure 5.2: PSNR progression per iteration for GN/ADAM on Ficus dataset.

The result for the microphone dataset which can be seen in Figure 5.3 is similar to that of the ficus. After 20 iterations GN has converged more than ADAM but has also taken more total time to execute. The difference is especially visible on the microphones cable and stand where ADAM requires more iterations to represent the structure. Figure 5.4 shows a similar PSNR improvement trend for both optimizers, although at a higher rate per iteration for the GN optimizer.

Figure 5.3: Results of reconstruction after 20 iterations for the microphone dataset. To the left reconstruction times and PSNR-scores are shown for both GN and ADAM.

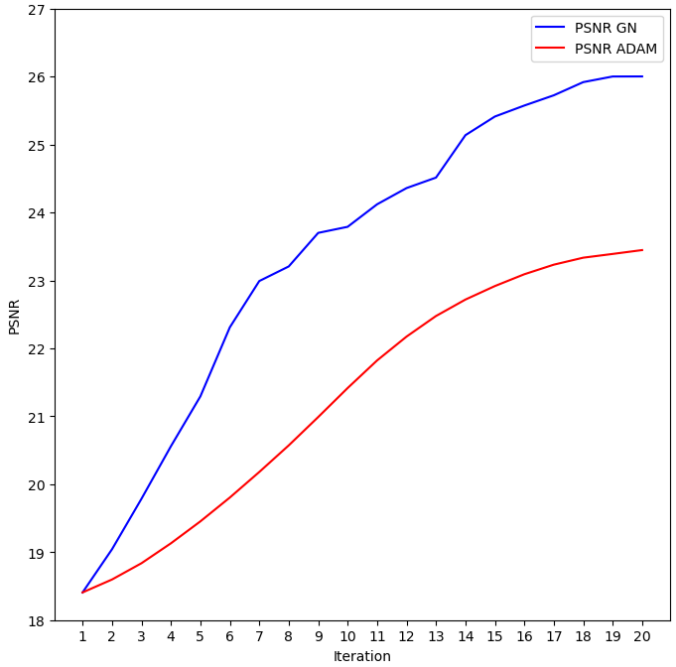


Figure 5.4: PSNR progression per iteration for GN/ADAM on microphone dataset.

The hotdog dataset when trained for 20 iterations shows a similar result to the ficus and can be seen in Figure 5.5. After 20 ADAM iterations it can be see that the majority of Gaussians have not increased much in size from their point cloud initialization. However, GN shows a higher degree of convergence albeit with visible artifacts. The hotdog dataset took more time to compute using GN than the microphone and ficus dataset. PSNR improvement rate per iteration, seen in Figure 5.6 shows improvement for GN but not for ADAM. The improvement rate levels out which was not seen in the other datasets, and for ADAM it improves to later switch and worsen.

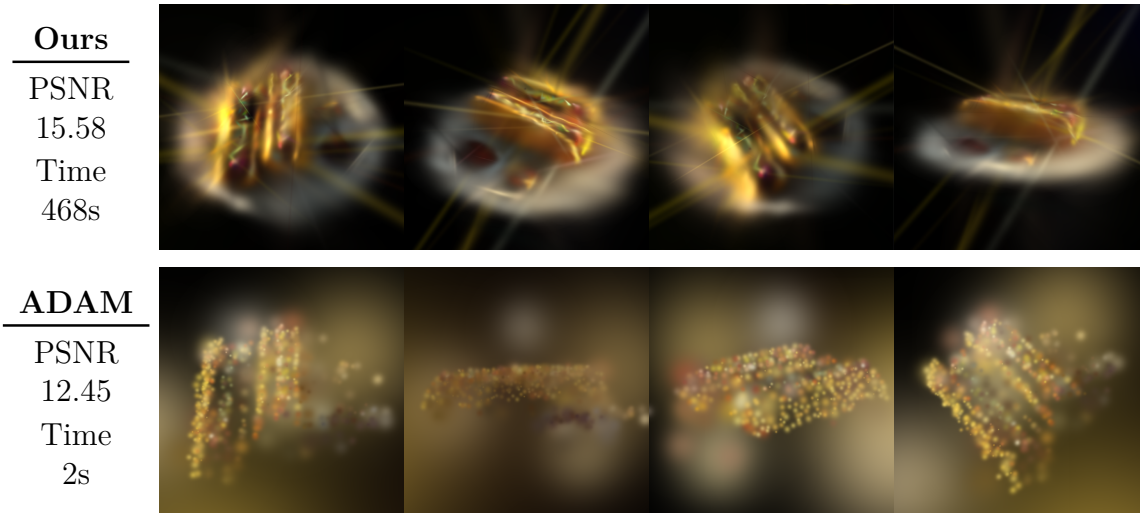


Figure 5.5: Results of reconstruction after 20 iterations for the hotdog dataset. To the left reconstruction times and PSNR-scores are shown for both GN and ADAM.

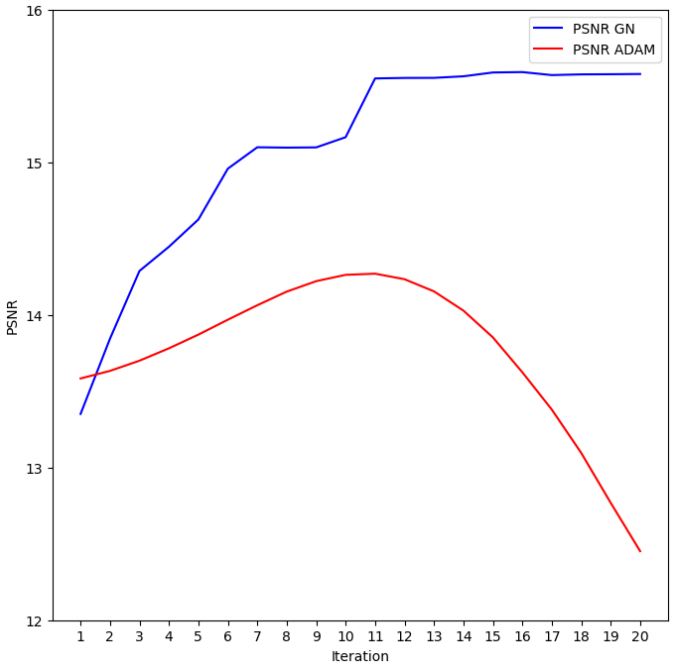


Figure 5.6: PSNR progression per iteration for GN/ADAM on hotdog dataset.

5.2 Performance

As seen in section 5.1, our GN optimzer can achieve higher quality in less iterations than ADAM but at the cost of significantly increased training time. To further explore GN's performance, this chapter compares GN and ADAM for more iterations to observe how spherical harmonics impacts optimization. For this all three spherical harmonics degrees are activated from the start.

The hotdog scene was used to further benchmark the difference between GN and ADAM. In Figure 5.7 the results from GN optimized at 100 iterations is illustrated along with its metrics. A second test with ADAM was then executed until similar quality in terms of PSNR was reached, this took 200 iterations. The visually inspected results are similar, with GN showing less jagged edges and more blended Gaussians around the hotdog bun. This observation is further supported by the slightly higher SSIM score. Figure 5.8 shows the PSNR progression per iteration and per second for ADAM and GN on the hotdog dataset. ADAM progresses rapidly and achieved better quality in less time according to the PSNR metric. GN is slower which is due to the fact that each iteration requires more computations than ADAM. On a per-iteration basis GN converges quicker, which is to be expected.

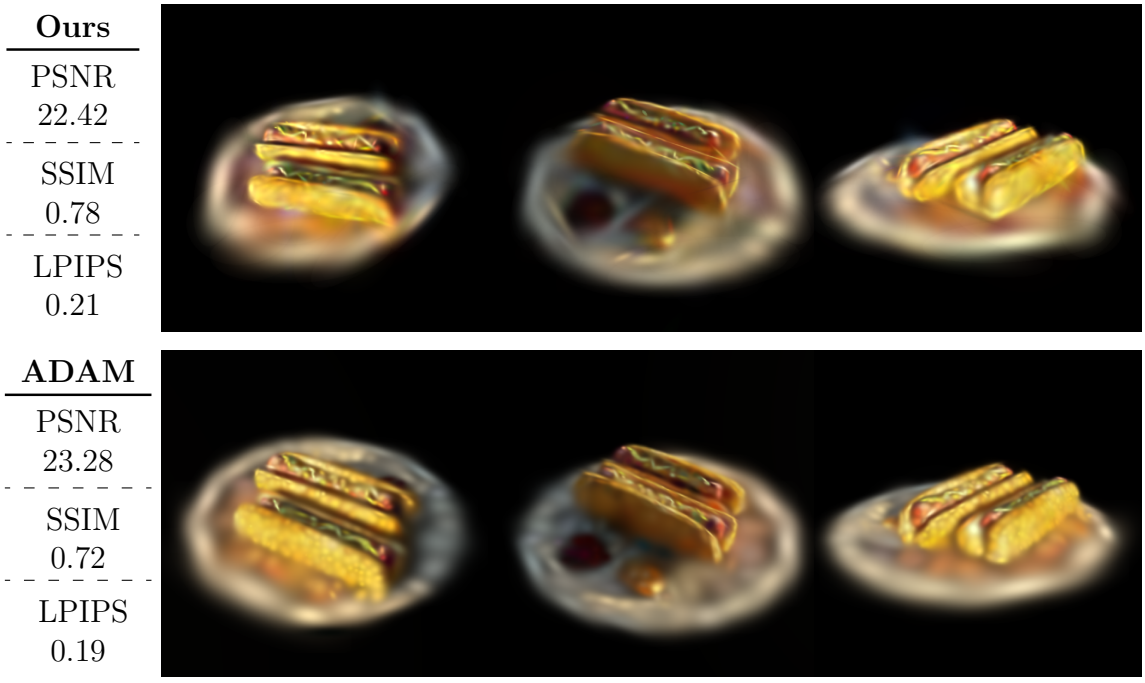
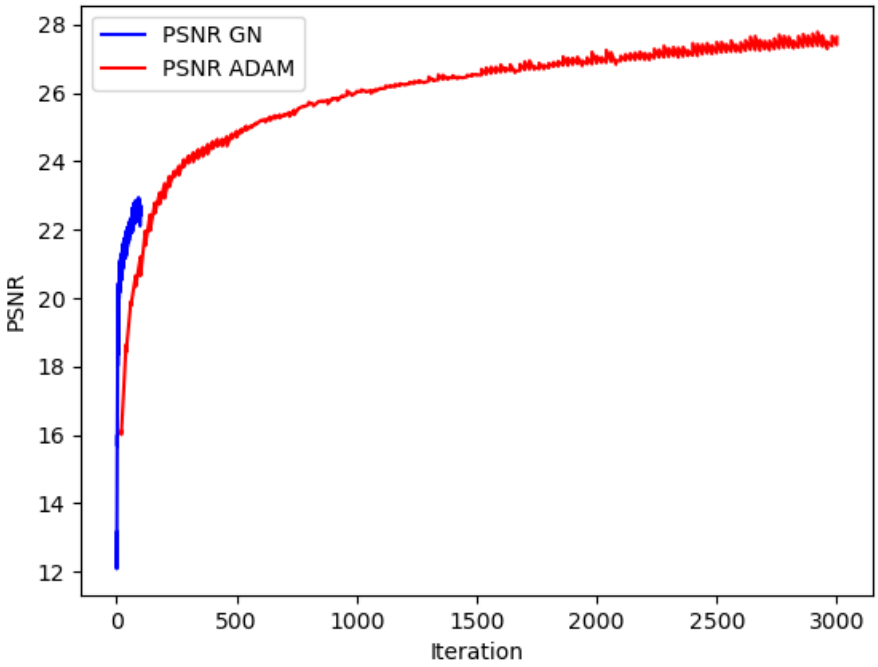
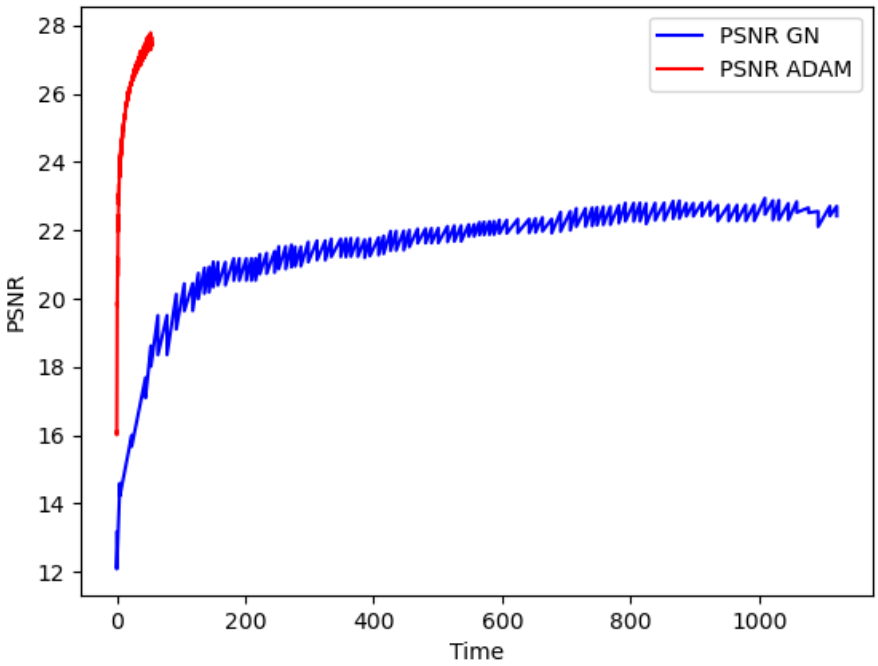


Figure 5.7: Reconstruction result and quality metrics for 100 optmization iterations using GN implementation (Ours) and 200 iterations using of ADAM implementation.



(a) PSNR over iterations.



(b) PSNR over time (seconds).

Figure 5.8: PSNR progression over time and iteration using either Gauss-Newton or ADAM on the hotdog scene from the NeRF-Synthetic dataset.

5.3 ADAM & GN combination

As previously described in the method section, we took inspiration from Höllein *et al.* [9] and combined ADAM and GN into one optimizer. They saw improved performance from first initiating the Gaussinas with the ADAM optimizer for some iterations and then towards the end switching over to the Levenberg-Marquardt optimizer. It would therefore be interesting to see if our GN implementation achieved similar improvements. However, after testing different switching points, as seen in Figure 5.9 and 5.10, barely any benefit was achieved from switching. Rather, the opposite effect can be observed where progression almost stops entirely compared to only using ADAM.

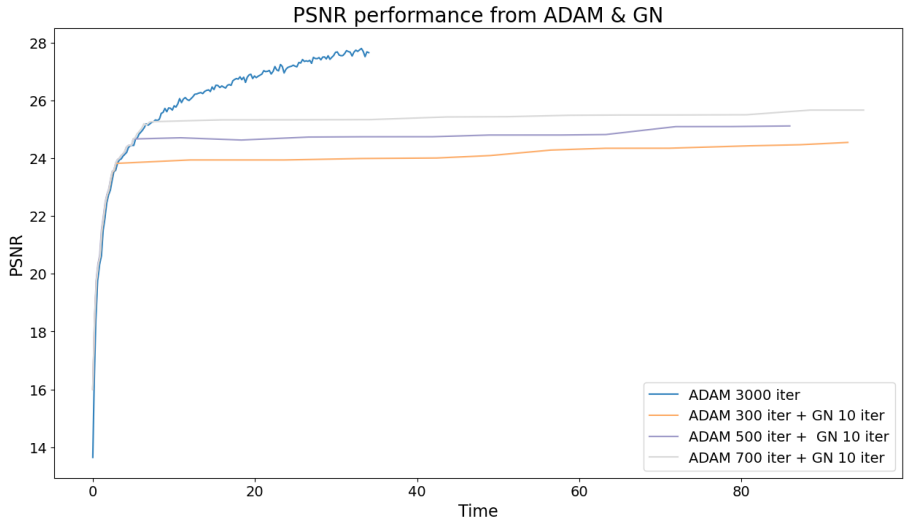


Figure 5.9: PSNR progression from combining ADAM and GN optimizer on hotdog dataset.

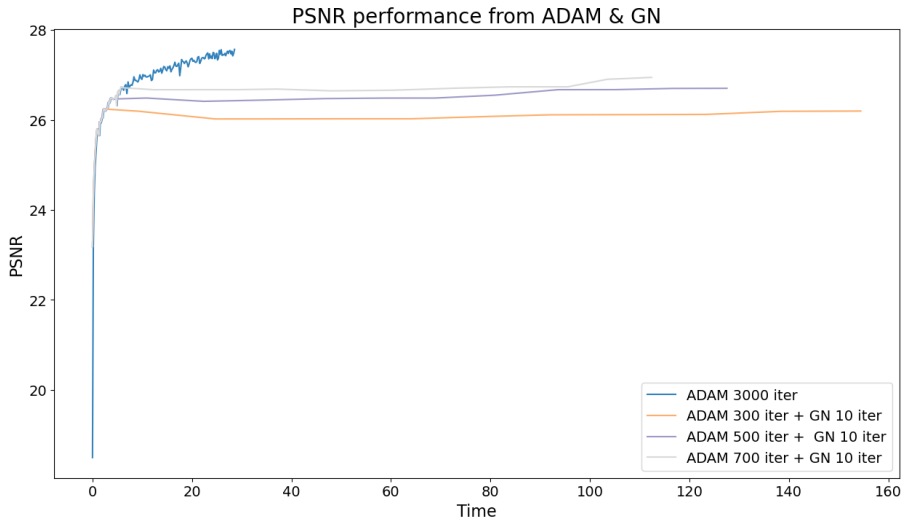


Figure 5.10: PSNR progression from combining ADAM and GN optimizer on mic dataset.

5.4 Multi-view Optimization

As mentioned in Sections 4.1 and 4.2, the Gauss-Newton optimizer achieves more stable convergence when each iteration incorporates multiple views. This section explores how batch size and number of batches will affect reconstruction quality. Table 5.1 and 5.2 shows the results from multiple tests, listing quality and execution time for a number of combinations. The batch size is limited to a maximum of 6 views to fit within the cache while maintaining 500 Gaussians. The best results in terms of quality is achieved when batch size and number of batches is high which means more total views are processed each GN iteration. This, however, comes at the cost of increased computation time. The tests show that with 3 batches and 6 views, or with 2 batches and 6 views yields the most promising results in terms of elapsed time and final quality without spherical harmonics. With spherical harmonics enabled, 3 batches with 6 views and 4 batches with 6 views yield the best results in terms of quality.

Table 5.1: Reconstruction quality of the scene without spherical harmonics as a function of batch size and number of batches over 20 optimization iterations.

# Batches	Batch size	PSNR ↑	SSIM ↑	LPIPS ↓	Time (s) ↓
1	1	1.642	0.180	0.595	13
1	3	19.701	0.487	0.190	78
1	6	22.357	0.377	0.205	132
2	3	23.539	0.448	0.219	149
2	6	24.385	0.692	0.176	258
3	3	21.592	0.359	0.173	181
3	6	25.129	0.811	0.155	306
4	6	25.070	0.680	0.193	495

Table 5.2: Reconstruction quality of the scene with spherical harmonics as a function of batch size and number of batches over 20 optimization iterations.

# Batches	Batch size	PSNR ↑	SSIM ↑	LPIPS ↓	Time (s) ↓
1	1	0.728	0.042	0.556	18
1	3	6.069	0.044	0.594	88
1	6	15.683	0.338	0.320	127
2	3	20.560	0.374	0.231	183
2	6	24.320	0.619	0.195	354
3	3	23.866	0.513	0.177	257
3	6	26.201	0.829	0.145	433
4	6	26.244	0.833	0.143	333
5	6	26.111	0.821	0.156	444

5.5 Runtime Analysis

To understand the performance characteristics of the implementation, an analysis was carried out on the different code segments and kernels. The NVIDIA tool *Nsight Compute* [18] was used to study the execution time of the different segments of the program. An overview of this analysis is shown in Figure 5.11, which highlights that the majority of the execution time is spent in the PCG. This is not surprising, as the PCG involves the most thread launches and requires several kernel reductions per iteration, leading to synchronization overhead and execution stalls between threads.

During the implementation of the Gauss-Newton optimizer, some kernels from the forward and backward passes in the differentiable rasterization pipeline were modified, as discussed in the Methods section. These modifications were inspired in part by efficient practices in raster-based differentiable rendering, such as those demonstrated in DISTWAR [19]. However, these changes had minimal impact on overall performance, especially when measured against the execution time of the PCG and line-search algorithms.

Using the tool, it is evident that the `sum_residuals` kernel accounts for the largest portion of execution time per iteration, followed by `residual_dot_sum`. The line-search implementation also consumes significant computational resources, as it must render all, or a predefined subset, of the views to evaluate the error for each α value in Algorithm 3. However, since line search is performed only once per iteration, it constitutes only a small fraction of the total execution time per iteration.

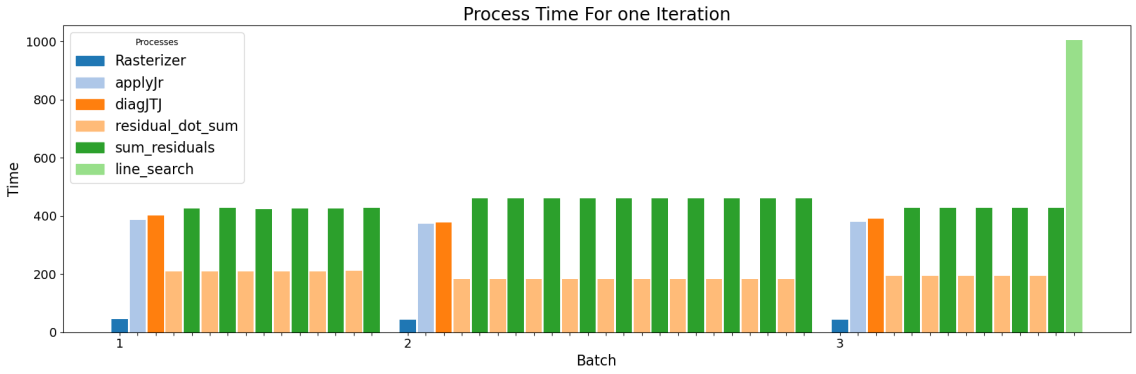


Figure 5.11: Processing time of the main components of the program during a single iteration.

Using the tool *Nsight Systems* [20] from NVIDIA, it becomes clear that the main bottleneck in execution are the kernels `diagJTJ`, `applyJr` and `sum_residuals`, as can be seen in Table 5.3. It is clear from the table that kernel `residual_dot_sum` on average achieves approximately double the memory and compute throughput compared to the other kernels. This is most likely due to the blocking nature of atomic operations in the kernel functions and since `residual_dot_sum` is structured a bit differently, it could be the reason why it performs better.

The profiling tool indicates that all kernels, except for `residual_dot_sum`, exhibit an

L2 Slices Workload Imbalance. This suggests that the number of active cycles varies across L2 cache slices. One potential cause of this imbalance is the unstructured layout of the sparse cache. The cache is organized in Gaussian order and each Gaussian contributes to a varying number of pixels. The required number of cache accesses for L1 and L2 therefore become unpredictable, both on a block and warp level. Furthermore, due to the tiled structure implemented by Mallick *et al.* [8], the same Gaussian may be scattered throughout the sparse cache as a result of duplicate Gaussians being used to enhance parallelization. This scattering could further exacerbate the workload imbalance among L2 cache slices.

Table 5.3: Runtime statistics for the four main kernels of the PCG algorithm.

Kernel	Comp. Thruput [%]	Mem. Thruput [%]	Est. speedup [%]
`applyJr`	14,26	25,18	23.05
`diagJTJ`	13,54	23,90	23.71
`residual_dot_sum`	28,91	44,26	0.00
`residual_sum`	13,92	22,78	24.22

A key advantage of the sparse cache is its memory efficiency, which makes the GN algorithm practical even for large numbers of Gaussians. For example, when optimizing a scene with 500 Gaussians, the spasre cache can reduce memory usage by up to 98.63% compared to the sparse alternative, as shown in Table 5.4. The non-sparse cache in this case corresponds to the size of the cache if no space optimization was made, i.e, all accumulated radiance and transmittance for the jacobian was stored. Notably, the relationship between cache size and the number of Gaussians appear not to be linear, as illustrated in Figure 5.12. This could be due to that as the number of Gaussians increase, the space in a scene they must cover decreases. If the relationship were linear, the resulting memory usage across different Gaussian counts would resemble the evenly spaced lines shown in Figure 5.13.

Table 5.4: Average space savings achieved from using the sparse cache for different scenes over 20 iterations with 500 Gaussians and 6 views.

Scene	Sparse Cache [Bytes]	non-Sparse Cache [Bytes]	Savings [%]
Mic	671 750 369	46 080 000 000	98.54
Chair	715 701 572	46 080 000 000	98.45
Hotdog	676 034 722	46 080 000 000	98.53
Ficus	630 539 406	46 080 000 000	98.63

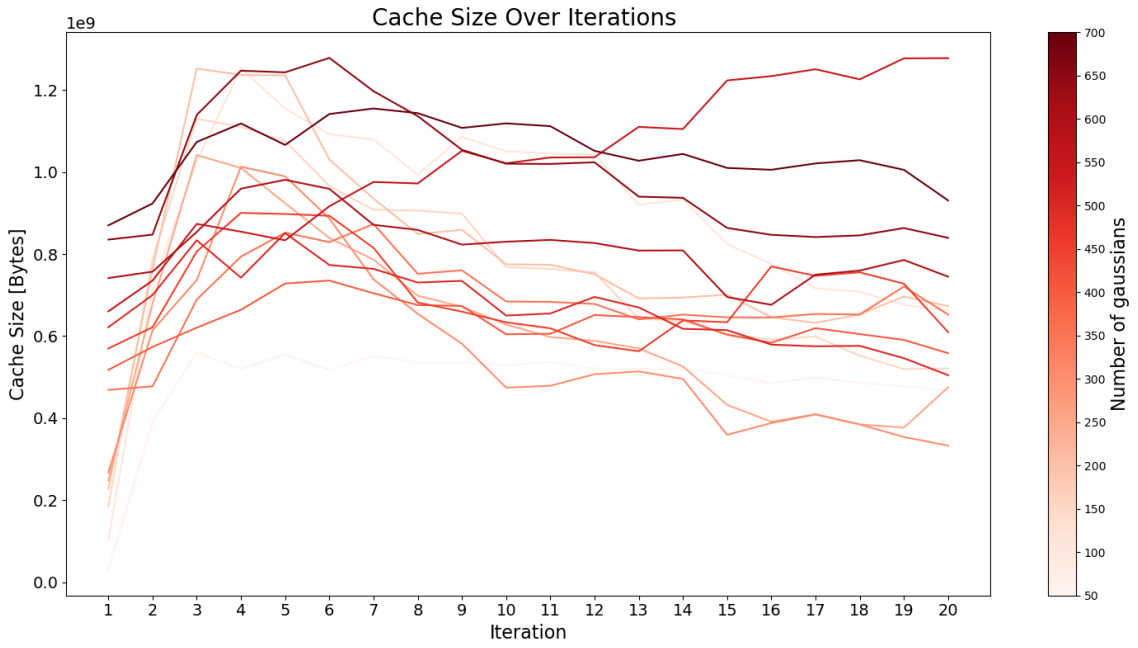


Figure 5.12: Size of sparse cache over 20 iterations for a varying number of Gaussians.

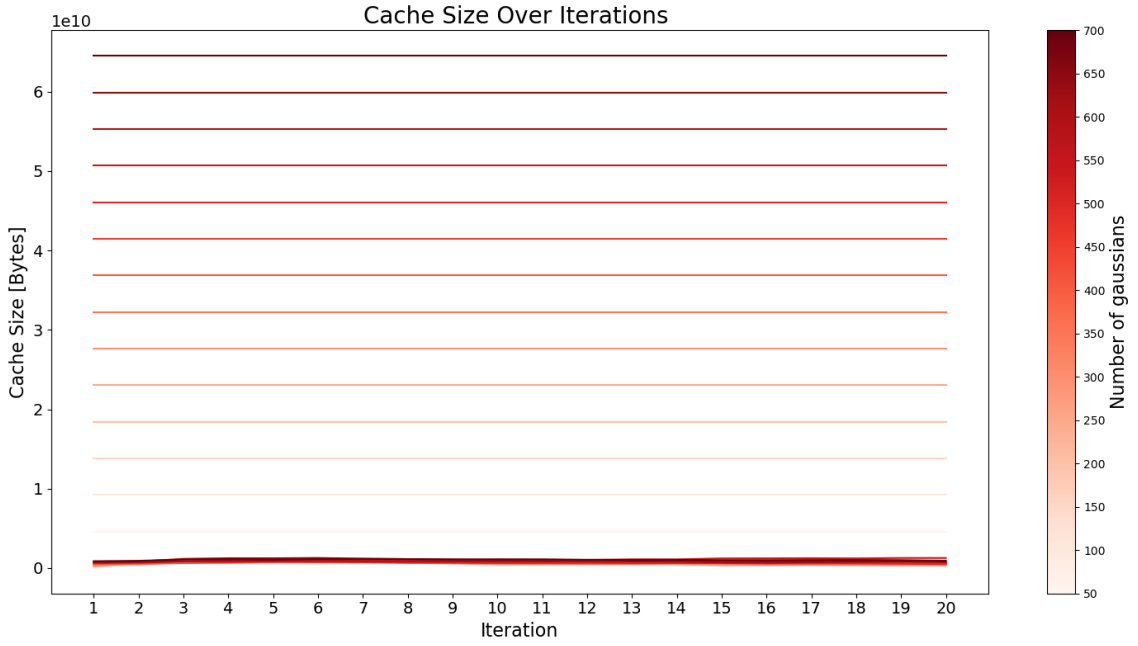


Figure 5.13: Size of non-sparse cache (horizontal lines) compared to the sparse cache (lowest lines) over 20 iterations for a varying number of Gaussians.

6

Discussion

6.1 ADAM and GN Combination

The results presented in Figures 5.9 and 5.10 indicate that switching between the ADAM and Gauss-Newton (GN) optimizers yields no significant performance gains. In fact, the performance progression starts to slow down when GN is activated. Changing the iteration at which the algorithm switches to GN does not appear to improve performance either. Switching earlier also proves ineffective, as GN consistently demonstrates slower convergence relative to ADAM. As a result, while GN may start from a slightly better PSNR value due to the initial progress made by ADAM, it proceeds at a similar, slower pace thereafter. This behavior is particularly illustrated in Figure 5.8(b).

Delaying the switch to GN does not appear beneficial either, since while running separately, ADAM achieves a higher PSNR value overall compared to using GN. GN fails to improve quality beyond what it achieves alone and tends to stagnate, as seen in Figures 5.9 and 5.10. The underlying cause of this lower PSNR has not been definitively identified, but it may be related to the artifacts discussed in Section 6.2. However, it remains untested whether increasing the number of Gaussians and/or number of viwes could impact performance or quality positively, potentially playing to GNs strengths.

6.2 Artifacts

Optimizing a scene from a dataset using GN introduces visible artifacts. In Figures 5.1, 5.3 and 5.5 these artifacts appear as stretched ellipsoids and are in fact Gaussians. The reason behind this is the fact that available resources have forced us to significantly limit the number of views per iteration to not run out of VRAM. When a Gaussian is optimized from a limited number of views, the update vector only takes these into account and thus lacks information from the remaining viewing angles. Our theory is that this makes the algorithm optimize from the information it has, disregarding the result from other unseen views. Since stretching a Gaussian in the view direction does not matter for that iteration the algorithm sees it as an effective way to densify the color. By introducing image subsampling and utilizing more batches per iteration we increase the number of views seen in total for each GN iteration. During testing this has proved to reduce the stretched artifacts as the

total viewing angles seen during a GN iteration increases. The image subsampling scheme however, comes with additional compute costs, each additional batch that is processed must go through the PCG and its kernels which are costly in terms of computation time.

With spherical harmonics enabled, we observe less apparent artifacts during optimization, both in shape and size. The artifacts in early stages of optimization tend to be less elongated and more compact but can display colors that are more distant from the ground truth images. We usually see bright colors of red or green that subside and eventually mostly disappear as optimization progresses. An example of this can be seen in Appendix A, Figure A.1 where green, red and blue areas are visible where there should not be any. These artifacts gradually fade over the optimization process, but the root cause remains unidentified and could be due to implementation errors or too few views. These artifacts are also more viable from viewing angles that are not covered by training images.

6.3 Subsampling

The main reason for implementing an image subsampling scheme is to combat excessive VRAM usage and gain the ability to process more views per iteration. In an ideal situation we would like to process all views of a scene in a single iteration, which would help us make the optimal update step for each view at once. We implement techniques to improve memory efficiency such as reducing the size of the Jacobian matrix. This is done by excluding zero values and instead adding the remaining entries to an array. The array is then accompanied by an indexing array which keeps track of the order of the entries. We decide to only cache the accumulated transmittance and radiance for each pixel-Gaussian pair. These values are then used to re-calculate the necessary gradients each time instead of caching them, which also frees up more VRAM to be used for processing more views and Gaussians. The motivation for introducing additional computations to reduce VRAM usage is that gradient calculations involve simple operations that are highly optimized on the GPU. While our methods to reduce VRAM usage work, it is still not sufficient to process complex scenes with the required amounts of Gaussians on our testing machine with 8 GB of VRAM. When the number of Gaussians increases the need for information storage also increases because each Gaussian is represented using many Gaussian parameters, which in turn affects residuals.

The results from section 5.4 show that increasing the total amount of views per GN iteration leads to better quality in all three metrics. This strengthens our belief that if the pipeline can be optimized to use all views for each GN iteration, convergence is likely to improve. It is also apparent that utilizing more views increases computation time, because more views mean that more Gaussian parameters are introduced. With spherical harmonics activated and using more than two batches, the PSNR, SSIM and LPIPS seem to improve more than when they are not enabled. Increasing the number of batches indefinitely does not appear to improve the quality. This may result from the fact that subsampling is an approximation of how to optimize the scene parameters and not as precise as executing PCG over all views at once. Sub-

sampling takes the average over the batches, which is not equivalent to processing all views simultaneously. Increasing the number of batches may also introduce further inaccuracies. There may be room for improvement in the subsampling scheme, which could be explored further to achieve a better final update step.

6.4 Future Work

Our result show that GN requires a significant amount of VRAM to store all values necessary for computation. This limits us during training and when testing the implementation to generate the results. Our scheme to reduce sparsity improves memory efficiency significantly, and this warrants further investigation to determine whether memory usage can be improved even more. One approach could involve storing the partial derivatives of pixel values with respect to splats, similar to the method used in [9], instead of storing accumulated radiance and transmittance. This could potentially reduce memory usage by up to 17%.

Since each GN iteration takes a long time to compute, a natural next step is to study further optimizations of our kernels. Several kernels utilize atomic operations as part of their functionality and more optimizations to better utilize cache locality could also be explored. There could be novel ways of optimizing the parallelization scheme even more, such as reusing more values and reduce redundant calculations where possible. It is also relevant to evaluate how providing the runtime with more VRAM and compute power, via newer or more capable hardware, affects performance and how many Gaussians can be processed.

During development, we made the decision to perform all testing for ADAM and GN, both separated and combined, using the sum of squares objective function. This decision was made because GN requires this form and we wanted a fair comparison. However, as seen in [9], the original objective function using SSIM and L1 from [3] could be adapted to run with Levenberg-Marquardt. This suggests that it may be compatible with our optimizer when modified, and this warrants further investigation. It is likely to produce better quality since it takes structural information into account, rather than the simple sum of squares objective function.

6.5 Ethical Considerations

While our project and the field of 3D reconstruction aims to advance the field of computer graphics, the increased accessibility could lead to potential misuse. 3D Gaussian Splatting could be used not only for positive, creative applications but also for those with malicious intent.

Increasing realism and accessibility of 3D reconstruction technologies could lead to a rise in the unauthorized use or reproduction of copyrighted works, as individuals may more easily replicate or modify protected content. This could result in more frequent infringements on intellectual property rights, as it becomes easier for people to create and distribute 3D models or digital assets without proper permission or

compensation to the original creators. 3D Gaussian Splatting could potentially be used to replicate a persons likeness without their consent. Currently, the risk is relatively low as producing realistic and convincing results typically requires high-quality images from multiple viewing angles. However, as research in this area progresses and the technology becomes more accessible the risk of misuse increases. Individuals may also be able to distort or edit the original data, potentially spreading misinformation.

6.6 Conclusion

We have presented a Gauss-Newton optimizer for reconstruction of 3D Gaussian Splatting scenes that achieves higher convergence rate per iteration than ADAM. Our results demonstrate that non-linear least squares solvers utilizing second-order information are a competitive alternative to stochastic gradient descent methods within 3DGS. This finding confirms our initial hypothesis that Gauss-Newton can be used as a optimizer for 3DGS in place of ADAM. However, while per-iteration convergence improves, this comes at the cost of significantly higher computational complexity and execution time, meaning that overall speed-up was not achieved in our current implementation.

Although GN have proven capable of driving convergence in 3DGS, we encountered several challenges. The most notable being the appearance of visible artifacts during optimization and the stagnation of the PSNR metric. Artifacts are likely due to implementation errors or a lack of viewing angles per GN iteration.

While our GN-based implementation is not yet a practical replacement for ADAM due to its high memory and compute demands, it opens up several avenues for future work. These include improving memory efficiency, kernel performance, and exploring hybrid optimization strategies. GN show potential to advance the field of 3D scene reconstruction, particularly if further implementation optimizations and hardware support can be leveraged.

Bibliography

[1] K.-A. Aliev, A. Sevastopolsky, M. Kolos, D. Ulyanov, and V. Lempitsky, *Neural point-based graphics*, 2020. arXiv: 1906.08240 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1906.08240.

[2] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, *Nerf: Representing scenes as neural radiance fields for view synthesis*, 2020. arXiv: 2003.08934 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2003.08934.

[3] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering," *ACM Transactions on Graphics*, vol. 42, no. 4, Jul. 2023. [Online]. Available: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/.

[4] G. Chen and W. Wang, *A survey on 3d gaussian splatting*, 2025. arXiv: 2401.03890 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2401.03890.

[5] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1412.6980.

[6] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. Springer New York, NY, USA, 2006.

[7] S. Rasmuson, E. Sintorn, and U. Assarsson, *Perf: Performant, explicit radiance fields*, 2021. arXiv: 2112.05598 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2112.05598.

[8] S. S. Mallick, R. Goel, B. Kerbl, F. V. Carrasco, M. Steinberger, and F. D. L. Torre, *Taming 3dgs: High-quality radiance fields with limited resources*, 2024. arXiv: 2406.15643 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2406.15643.

[9] L. Höllein, A. Boi, M. Zollhöfer, and M. NieSSner, *3dgs-lm: Faster gaussian-splatting optimization with levenberg-marquardt*, 2024. arXiv: 2409.12892 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2409.12892.

[10] R. Green, "Spherical harmonic lighting: The gritty details," in *Game Developers Conference*, Mar. 2003. [Online]. Available: https://3dvar.com/Green2003Spherical.pdf.

[11] J. L. Schönberger, E. Zheng, M. Pollefeys, and J.-M. Frahm, "Pixelwise view selection for unstructured multi-view stereo," in *European Conference on Computer Vision (ECCV)*, 2016.

[12] J. L. Schönberger and J.-M. Frahm, "Structure-from-motion revisited," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[13] W. Sun and Y. Yuan, *Optimization Theory and Methods: Nonlinear Programming* (Springer Optimization and Its Applications), 1st ed. New York, USA: Springer, 2006. [Online]. Available: https://link.springer.com/book/10.1007/b106451.

[14] D. Buffelli, J. McGowan, W. Xu, *et al.*, *Exact, tractable gauss-newton optimization in deep reversible architectures reveal poor generalization*, 2024. arXiv: 2411.07979 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2411.07979.

[15] NVIDIA. "Cuda c++ best practices guide." (2025), [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/. (accessed: on 2025-05-23).

[16] A. Horé and D. Ziou, "Image quality metrics: Psnr vs. ssim," in *2010 20th International Conference on Pattern Recognition*, 2010, pp. 2366–2369. DOI: 10.1109/ICPR.2010.579.

[17] S. Ghazanfari, S. Garg, P. Krishnamurthy, F. Khorrami, and A. Araujo, *R-lpips: An adversarially robust perceptual similarity metric*, 2023. arXiv: 2307.15157 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2307.15157.

[18] NVIDIA. "Nvidia nsight compute." (2025), [Online]. Available: https://developer.nvidia.com/nsight-compute. (accessed: on 2025-05-20).

[19] S. Durvasula, A. Zhao, F. Chen, R. Liang, P. K. Sanjaya, and N. Vijaykumar, *Distwar: Fast differentiable rendering on raster-based rendering pipelines*, 2023. arXiv: 2401.05345 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2401.05345.

[20] NVIDIA. "Nvidia nsight systems." (2025), [Online]. Available: https://developer.nvidia.com/nsight-systems. (accessed: on 2025-05-20).

A

Appendix 1

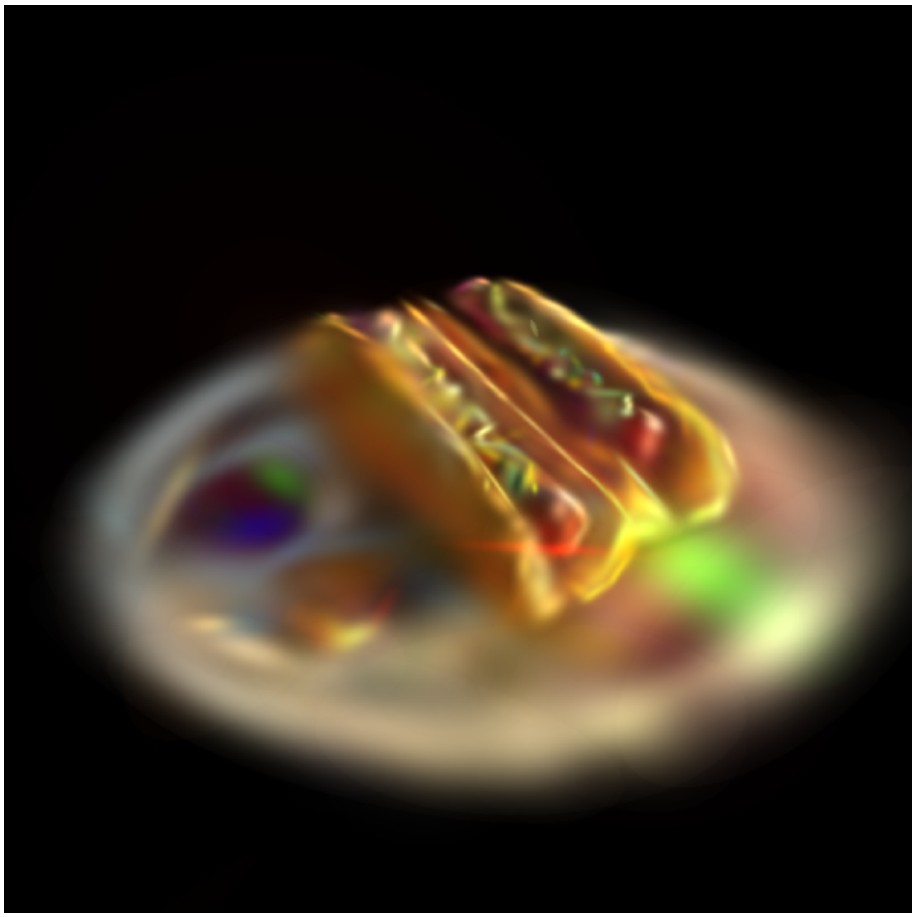


Figure A.1: Artifacts during optimization for GN with spherical harmonics enabled.