ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CVL Computer Vision Lab

3D Gaussian Splatting Representation Compression

Application of Learned Image/Video Compression Methods on 3D Gaussian Splatting

Semester Thesis

Eren Cetin
Department of Electrical Engineering and Information Technology

Advisors: Ertunç Erdil and Yannick Strümpler
Supervisor: Prof. Dr. Ender Konukoğlu

July 17, 2024

Abstract

3D Gaussian splatting has recently gained immense popularity due to its high parallelizability and efficiency, allowing 3D scenes to be rendered faster than neural radiance field-based methods while maintaining comparable quality. However, representing a scene with 3D Gaussian splatting requires a large number of Gaussian primitives, from hundreds of thousands to several millions, resulting in high storage complexity. To address this issue, we investigate the use of learned entropy models from the image compression literature and residual coding for Gaussian attribute compression. We also explore enhancements to the 3D Gaussian splatting algorithm using a Markov Chain Monte Carlo framework and investigate methods to reduce the number of Gaussian primitives through learned primitive masking and importance-based pruning. Our experiments show that optimizing Gaussian primitives with the Markov Chain Monte Carlo framework significantly improves the visual quality of novel view synthesis. Additionally, learned primitive masking and importance-based pruning can reduce the number of Gaussian primitives by up to half without notable quality loss. We demonstrate that learned entropy modeling, combined with a hyperprior network, can integrate seamlessly into optimized Gaussian primitives, reducing their size by up to 10 times without degrading visual quality. As the integration does not require any modification in Gaussian primitives, it is an easy method to adopt. Further investigation of hierarchy generation and residual coding reveals that hierarchy structure with octree representation and weighted averaging does not allow for higher compression efficiency, indicating a more complex Gaussian attribute prediction scheme might be required to increase storage efficiency. These findings highlight the potential for further storage improvements in 3D Gaussian splatting while maintaining high visual quality, paving the way for scalable rendering techniques. The details of the implementation can be accessed in https://github.com/erenovic/GSCompression.

Acknowledgements

I would like to thank my supervisors, Dr. Ertunç Erdil and Yannick Strümpler, for their constant support and helpful advice throughout my research. Their knowledge and encouragement have been very important to this semester thesis.
I also want to recognize the important work of Dr. Bernhard Kerbl, Dr. Andreas Meuleman, Shakiba Kheradmand, and many others, whose research papers provided a strong foundation for this semester thesis.
Finally, I am deeply thankful to my family for their ongoing support, patience, and understanding. Their encouragement has been a source of strength for me.

Contents

1 Introduction **1**
1.1 Focus of this Work . . . 1
1.2 Thesis Organization . . . 2

2 Related Work **3**
2.1 3D Gaussian Splatting . . . 3
2.1.1 Improvements Over 3D Gaussian Splatting Framework . . . 3
2.1.2 Hierarchical Scene Structuring and Gaussian Ordering . . . 4
2.1.3 Gaussian Primitive Pruning . . . 5
2.1.4 Gaussian Splatting Compression . . . 5
2.2 Point Cloud Compression . . . 6
2.3 Learned Entropy Modeling . . . 7

3 Materials and Methods **9**
3.1 Background: 3D Gaussian Splatting . . . 9
3.1.1 Gaussian Primitive Structure . . . 9
3.1.2 Differentiable Rasterization . . . 10
3.1.3 Storage Complexity of Gaussian Primitives . . . 11
3.1.4 Optimization of Gaussian Primitives . . . 11
3.2 Improving 3D-GS through Markov Chain Monte Carlo (MCMC) . . . 13
3.3 Gaussian Primitive Pruning . . . 14
3.4 Learned Entropy Models for 3D Gaussian Primitive Compression . . . 14
3.4.1 Fully-Factorized Entropy Modeling . . . 16
3.4.2 Hierarchical Entropy Modeling . . . 17
3.4.3 Multi-rate Entropy Modeling and Rate Adaptation . . . 19
3.4.4 Disjoint Position Compression . . . 19
3.5 Hierarchy Generation . . . 19
3.5.1 3D Gaussian Octree Building . . . 20
3.5.2 3D Gaussian Primitive Aggregation . . . 21
3.5.3 Residual Coding Using Lower Hierarchy Levels As Predictions . . . 22
3.6 Resources and Framework of Implementation . . . 23

4 Experiments and Results **25**
4.1 Metrics and Datasets . . . 25
4.2 3D Gaussian Splatting Optimization Improvement . . . 25
4.3 3D Gaussian Splatting Primitive Pruning . . . 26
4.4 Gaussian Primitive Compression using Learned Entropy Models . . . 29

4.4.1 Fully-Factorized and Mean-Scale Hyperprior Entropy Models 29
4.4.2 Entropy Penalization Start Iteration and Training Duration 30
4.4.3 Freezing Geometry Parameters 31
4.4.4 Position Compression using DEFLATE and G-PCC 33
4.4.5 Multi-rate Gaussian Primitive Compression 35
4.4.6 Learned Quantization and Attribute Importance 36
4.5 Hierarchy Generation for 3D Gaussian Primitives 38
4.5.1 Effect of Depth Selection 38
4.5.2 Residual Coding for Gaussian Primitives 40

5 Discussion **43**
5.1 3D Gaussian Splatting Optimization Improvement 43
5.2 3D Gaussian Splatting Primitive Pruning 43
5.3 Gaussian Primitive Compression using Learned Entropy Models 44
5.3.1 Position Compression 44
5.3.2 Multi-rate Compression 44
5.3.3 Limitations and Relation to Previous Work 45
5.4 Residual Coding for Gaussian Primitive Compression 45

6 Conclusion **47**

A Experiment Results of Pruning Methods **49**

B Experiment Results of Single Lagrangian Training and Compression **51**
B.0.1 Training Entropy Model for $\mathbf{5K}$ After Optimizing Gaussian Primitives for $\mathbf{25K}$ Iterations 51
B.0.2 Training Entropy Model for $\mathbf{10K}$ After Optimizing Gaussian Primitives for $\mathbf{30K}$ Iterations 55
B.0.3 Experiment Results of Freezing Gaussian Primitive Geometry Attributes During Entropy Model Training 59

C Hierarchy Generation Examples on Different Datasets **65**

List of Figures

3.1 **The parameter distribution of Gaussian primitives.** The spherical harmonics with maximum degree of 3 require 48 floating point values while position, scaling, rotation, and opacity require 3, 3, 4, 1 floating point values, respectively. This situation depicts the importance of high compression for spherical harmonics. 11

3.2 **Gaussian splatting densification and pruning algorithm in a decision tree format**. Gaussian splatting algorithm requires adding or removing Gaussian primitives periodically (once in every 100 iterations) using the described algorithm. The hyperparameters for optimization are used as is from the original 3D-GS descriptions [15]. 12

3.3 **An overview of lossy compression pipeline in high level.** A compression pipeline is composed of an analysis transform to map signals to a low-entropy representation, quantization for applicability, entropy modeling for efficient coding, and synthesis transform to map signal back into domain of interest. 15

3.4 **An overview of fully-factorized entropy model.** The fully-factorized entropy model is utilized for a part of the analysis for learned Gaussian primitive compression. The green modules depict the learned components such as MLP layers or quantization parameters $\mathbf{s}$. Note that the quantization scale $\mathbf{s}$ is shared for analysis and synthesis transforms. The orange modules are the fixed operations such as quantization and entropy coding through learned entropy model. 16

3.5 **An overview of hierarchical entropy model with Gaussian conditional probability model.** A hyper-network is utilized to model Gaussian primitive parameters using a conditional Gaussian distribution. The green modules depict the learned components such as MLP layers or quantization step sizes like $\mathbf{s}$. Note that the quantization scale $\mathbf{s}$ is shared for analysis and synthesis transforms. The orange modules are the fixed operations such as quantization and entropy coding through learned entropy model. The blue box depicts the entropy parameters estimated by the “Hyper Network”. 18

3.6 **A high-level visualization of octree structure [7].** An octree is created by dividing the 3D space into 8 pieces for each bounding box that is non-empty. As a result, a tree structure is acquired where close Gaussians are children of the same node. 20

3.7 **An overview of the residual coding framework for compressing Gaussian primitives.** The green modules depict the learned components such as MLP layers or quantization step sizes like $\mathbf{s}$. Note that the quantization scale $\mathbf{s}$ is shared for analysis and synthesis transforms within each branch. The orange modules are the fixed operations such as quantization and entropy coding. For the entropy model components, mean-scale hyperprior can be utilized instead of fully-factorized entropy model. 23

4.1 **Qualitative comparison of 3D-GS [15] against 3D-MCMC [17].** The qualitative results on *train*, *bicycle*, *flowers*, and *treehill* scenes. As per seen from crops on rasterized images, 3D-MCMC yields superior visual quality over 3D-GS. 27
4.2 **Gaussian primitive position histogram comparison of learned masking and RadSplat pruning on *train* scene from Tanks & Temples [19] dataset.** We draw histograms of Gaussian primitive positions along each dimension. Specifically, we take the $0.2 \times \sigma$ range for each dimension around the mean position to reduce the sample size for histogram. We observe that both RadSplat pruning [27] and learned masking [20] achieve similar distributions for Gaussian primitives. 28
4.3 **Rate-Distortion Curve for fully-factorized and Meanscale-Hyperprior entropy models.** For both fully-factorized and mean-scale hyperprior entropy models, we first optimize the Gaussian primitives for $30K$ iterations and then train the respective entropy model for $10K$ iterations. During the training, we continue optimizing all Gaussian attributes. 30
4.4 **Rate-distortion comparison of training mean-scale hyperprior entropy model with and without optimizing the Gaussian geometry attributes.** We train the entropy model for $5K$ iterations after optimizing Gaussian primiitves for $25K$ iterations. As a result, we deduce that Gaussian geometry attributes are more prone to noise and should not be optimized. . . . 32
4.5 **Comparison of mean-scale hyperprior network against other works in the literature.** Our method achieves competitive results compared to LightGaussian [10], Compact3DGS [20] and Compressed3D [26] while achieving inferior results compared to HAC [6] and Context-GS [35]. 33
4.6 **The size distribution of compressed Gaussian splat representations with DEFLATE algorithm.** Using DEFLATE algorithm, we evaluate the size distribution of Gaussian primitives, decomposed into *entropy model parameters*, *Gaussian primitive positions*, and *Gaussian primitive attributes*. Our evaluation captures only Tanks & Temples since results are only scaled with respect to number of Gaussians for other datasets. 34
4.7 **Rate-distortion curves for multi-rate entropy model against training the entropy model separately for each rate-distortion parameter.** Rate-distortion performance of entropy model slightly falls when multi-rate training is performed. 35
4.8 **Effect of learned scaling on Gaussian primitive attributes and quantization on Tanks & Temples dataset [19].** Each bar represents the mean value of one attribute of Gaussian primitives. The standard deviation is visualized with black lines per attribute. The spherical harmonics are aggregated for R, G, B color channels to have 16 attributes instead of $16 \times 3 = 48$. 37
4.9 **Histogram for the increase in number of Gaussian primitives with increasing depth level in octree structure for *bicycle* scene from Mip-NeRF 360.** Coding of predicted Gaussians (intermediate nodes) enforces a trade-off between representativeness of intermediate nodes and number of intermediate nodes. For given scene, we utilize depth 20 as predictions for actual Gaussians, encoding $1/3$ of primitives as predictions. 38
4.10 **Effect of depth variation on visual quality on *playroom* scene from DeepBlending [12].** Using different octree levels results in changing granularity on images. Specifically, close points appear more blurry while far away points are less affected from averaging of Gaussian primitives due to the resolution. 39
4.11 **Distribution of Gaussian primitive position, aggregated position, and residual position attributes.** The distribution of position per dimension is acquired using kernel density estimation. Note that the distribution gets narrower for residual position elements, indicating a lower entropy. 40

4.12 **Distribution of Gaussian primitive, aggregated, and residual covariance matrices.** The distribution of covariance is calculated using kernel density estimation. Note that the distribution of residual covariance matrix elements gets wider, indicating a larger entropy and higher bitrates for compression. 41
4.13 **Rate-distortion curve comparison for residual compression with compressing covariance matrix against compressing scaling and rotation decomposition.** Two approaches for compressing covariance information yields curves at significantly different bitrates despite using same hyperparameters. 41
4.14 **Size distribution for residual compression.** Most of the bitrate requirement is first due to the residuals and then the aggregated Gaussians. This result is in contrast with the expectations based on the video compression literature as residuals are expected to require small amount of memory. 42

C.1 **Effect of depth variation on visual quality on *train* scene from Tanks & Temples [19] without compression.** Using different levels of octree results in changing granularity of scenes. Specifically, close points occur more blurry while far away points are less affected from averaging of Gaussian primitives. 66
C.2 **Effect of depth variation on visual quality on *truck* scene from Tanks & Temples [19].** Using different levels of Octree results in changing granularity on images. Specifically, close points occur more blurry while far away points are less affected from averaging of Gaussian primitives. 67
C.3 **Effect of depth variation on visual quality on *room* scene from Mip-NeRF 360 [2].** Using different levels of Octree results in changing granularity on images. Specifically, close points occur more blurry while far away points are less affected from averaging of Gaussian primitives. 68

List of Tables

4.1 **Comparison of 3D-GS [15] against 3D-MCMC [17].** We evaluate both 3D-GS and 3D-MCMC on Tanks & Temples [19], DeepBlending [12], and Mip-NeRF 360 [2] scenes in order to devise a base Gaussian optimization method. As comparison reveals superiority of 3D-MCMC with negligible overhead during training, we continue with 3D-MCMC in order to compress 3D Gaussian primitives. For evaluation of both methods, the public repositories of 3D-GS and 3D-MCMC are utilized without change in functionality to reproduce the results. For Scaffold-GS, the experiment results are taken as reported [21]. Note that our evaluation involves *treehill* and *flowers* in contrast to reported results in 3D-GS [15] and 3D-MCMC [17]. 26

4.2 **Comparison of learned masking against RadSplat pruning method on 3D-MCMC.** We evaluate two different pruning methods, namely learned masking proposed in [20] and importance-aware pruning method proposed in [27]. Pruning is an important step of Gaussian splatting compression since the redundancy in number of Gaussians should be first reduced through lowering number of Gaussians. The comparison reveals slightly better results with pruning method proposed with RadSplat in given two scenes. 28

4.3 **The quantitative compression results obtained from the proposed entropy models when trained for 10K iterations after 30K iterations of optimization for Gaussian primitives.** We compare the results with fully-factorized and mean-scale hyperprior entropy models to deduce which appears superior. As a result, mean-scale hyperprior has slightly better compression ratios with very similar performance. 29

4.4 **The quantitative compression results obtained from the proposed entropy models when trained for 5K iterations against $10K$ iterations with varying Gaussian primitive optimization durations.** As training for **10K** iterations is computationally demanding compared to training for an overall $30K$ iterations, we test having a total training time of $30K$ iterations where last $5K$ iterations are utilized for entropy model training. As a result, the improvement with training for an additional $10K$ iterations remains negligible. 31

4.5 **The quantitative compression results obtained from mean-scale hyperprior entropy model when optimization for geometry attributes continues against the case when they are fixed.** Mean-scale hyperprior entropy model is evaluated when it is trained for $5K$ iterations after optimizing the Gaussian primitives for $25K$ iterations. We compare the cases where geometry attributes are further optimized against the case when they are no longer optimized after $25K$ iterations. 31

4.6 **The quantitative compression results obtained from the proposed entropy models and other works in the literature.** 3D-GS [15] and Scaffold-GS [21] are non-compressed Gaussian splatting results included for reference. The results of other works are obtained from Context-GS [35]. Compared with other works in the literature, our approach with fully-factorized and mean-scale hyperprior entropy models achieves competitive results. 32

4.7 **Comparison of DEFLATE and TMC13 position compression algorithms on Tanks & Temples [19] dataset.** We compare DEFLATE [9] and TMC13 [22] compression algorithms for position compression. For our method, we found that DEFLATE algorithm is preferable due to undesired permutation performed by TMC13 during encoding. Since TMC13 permutes Gaussian primitive orders, we need to match Gaussian primitive orders in an ad-hoc manner, causing an increase in encode time depicted in red. 34
4.8 **Comparison of residual coding methods and single bottleneck compression.** We compare residual coding either with compressing covariance or scaling and rotation components. Single bottleneck model is the previously reported mean-scale hyperprior network without optimizing the geometry attributes during entropy model training. 42

A.1 **Comparison of learned masking against RadSplat pruning method on 3D-MCMC on Tanks & Temples [19] scenes.** 49
A.2 **Comparison of learned masking against RadSplat pruning method on 3D-MCMC on DeepBlending [12] scenes.** The Gaussian primitives for each method are optimized for $30,000$ iterations. 49
A.3 **Comparison of learned masking against RadSplat pruning method on 3D-MCMC on MipNeRF 360 [2] scenes.** 50

B.1 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.005$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. 51
B.2 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.005$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. 52
B.3 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.005$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. 52
B.4 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. 52
B.5 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. 53
B.6 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. 53
B.7 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.0001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. 53
B.8 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.0001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. 54
B.9 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.0001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. 54

B.10 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.005$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations. 55

B.11 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.005$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations. 55

B.12 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.005$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations. 56

B.13 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations. 56

B.14 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations. 56

B.15 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations. 57

B.16 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.0001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations. 57

B.17 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.0001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations. 57

B.18 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.0001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations. 58

B.19 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.005$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives. 59

B.20 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.005$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives. 59

B.21 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.005$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives. 60

B.22 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives. 60

B.23 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives. 61
B.24 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives. 61
B.25 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.0001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives. 62
B.26 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.0001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives. 62
B.27 **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives. 63

Chapter 1

Introduction

In recent years, novel view synthesis (NVS) has gained significant momentum, starting with the high success achieved by Neural Radiance Field (NeRF) based methods. Seminal work NeRF [24] has been influential in the domain by suggesting use of neural networks for mapping from the 3D positions to view synthesizing attributes like opacity and color. The high representative capability of neural networks allows for fitting to a scene and capturing high amount of scene details by overfitting network parameters for the scene. However, a setback with NeRF-based NVS methods is the computational cost due to inefficiency born from ray-tracing method, requiring point sampling at unimportant 3D positions [37]. As a result, they suffer from slow training and inference-time rendering.
In contrast, 3D Gaussian Splatting (3D-GS) [15] has emerged as a revolutionary approach due to its high efficiency, short training time and real-time rendering capability. The algorithm is based on "splatting" to project 3D Gaussian primitives onto a 2D image plane for efficient rasterization. This process is highly suitable for parallelization as 3D Gaussians are defined in a limited spatial extent and they are independent from each other [15]. Although the Gaussian splatting method has been around since 1991 [36], the proposal of a differentiable adaptation in 3D-GS [15] has allowed for an efficient optimization algorithm with gradient descent. In contrast to NeRF-based methods, 3D-GS does not require neural networks and is purely based on optimization of 3D Gaussians for further rasterization. As a result, it has an easily scalable and highly representative nature using millions of Gaussian primitives.
Although 3D-GS has been utilized in many domains from scene editing to SLAM applications in a short amount of time, it brings an important obstacle in terms of storage. This obstacle is not existent with NeRF-based NVS methods due to high representative capability of neural networks. As the 3D scenes are represented using millions 3D Gaussian primitives that are composed of geometry and appearance attributes, the storage requirement of a 3D scene grows linear in number of Gaussian primitives. A naive storage of Gaussian primitives with 32-bit representation per attribute results in scene representations requiring hundreds of MB to a few GB storage capacity [11]. This storage drawback is a limiting factor for applications such as autonomous driving, and AR/VR applications due to low storage capacity of edge devices [20].

1.1 Focus of this Work

This work focuses on reducing the storage requirements of 3D-GS to enable more compact scene representations without compromising real-time rendering capabilities. This result can be primarily achieved by 3 main strategies: (1) Reducing the number of 3D Gaussian primitives, (2) Reducing the precision required to represent a scene, (3) Achieving a better entropy modeling to reduce bitrate requirement to store and transmit Gaussians.
The primary focus of this work mainly lies in (1) pruning Gaussian primitives and (3) better performing

entropy modeling, while (2) is utilized only briefly for compressing position information of 3D Gaussian primitives. Using the extensive literature on learned image compression, this thesis focuses on alleviating the high storage complexity of 3D-GS. In addition, methods for controlling (pruning) the number of Gaussian primitives are compared and utilized for a further reduction in the storage requirements. As potential future direction and important step to reduce the entropy by making use of scene structure, a hierarchical scene structure for residual coding idea from video compression is further explored.

1.2 Thesis Organization

For the rest of this semester thesis, the following organization is adopted to provide a concise and fluent analysis:

- Section 2 provides a brief overview of related works that are essential for improving the visual quality of novel view synthesis and reducing the memory requirements of the Gaussian splatting representation. It introduces advancements in the seminal 3D-GS algorithm, Gaussian hierarchical structuring and Gaussian primitive pruning, as well as recent compression methods for Gaussian splatting representations. Finally, it presents related works in the literature of learned entropy modeling and point cloud compression. These discussions provide the necessary context for the reader.

- Section 3 presents the relevant theoretical background for the methods used for the experiments. It includes prior knowledge for the experiments performed and detailed theoretical information, organized in a manner similar to the related works for ease of understanding.

- Section 4, 5, and 6, cover the experimental results, a discussion of the results, and conclusion are provided with visualizations and literature comparisons. The experiments performed for this semester thesis are separated into mutually exclusive subsections to provide the logical flow for the decisions we take throughout the thesis.

Chapter 2

Related Work

2.1 3D Gaussian Splatting

With the introduction of seminal 3D-GS algorithm [15], the pace of the research field has been tremendous. This chapter reviews significant advancements in the field of 3D Gaussian Splatting (3D-GS), focusing on improvements to the seminal 3D-GS algorithm, hierarchical structuring, primitive pruning, and compression methods that have been suggested so far for 3D-GS representation. In addition, we provide a brief introduction of relevant works in point cloud compression and learned entropy modeling for image compression.

2.1.1 Improvements Over 3D Gaussian Splatting Framework

Although the seminal work on 3D-GS [15] appears to be highly successful for novel view synthesis, multiple works in a short time have provided improvements over the seminal work. Specifically, one of the early improvements is Scaffold-GS [21]. Scaffold-GS employs a hierarchical structure for 3D Gaussian primitives anchored on a sparse grid of 3D positions $\mathbf{x}_v$, and calls them "anchor points". Each anchor point is assigned a local context feature $\mathbf{f}_v$, a scaling factor l_v, and a set of learnable offsets, $\mathbf{O}_v$, which are used to generate neural Gaussians that are rasterized later on. The "anchor points" are not rasterized but only used to create neural Gaussians. By combining the learned offsets per anchor point and the scaling factor, the neural Gaussian positions μ_i, are estimated using the anchor points and learnable offsets,

$$\{\mu_0, \ldots, \mu_{k-1}\} = \mathbf{x}_v + \{\mathbf{O}_0, \ldots \mathbf{O}_{k-1}\} \cdot l_v \tag{2.1}$$

Similarly, together with the local context feature, camera direction $\vec{\mathbf{d}}_{vc}$ and camera distance, δ_{vc}, separate multilayer perceptrons (MLPs) are utilized to estimate the neural Gaussian opacities, view-dependent colors, quaternions, and scales. By employing the MLPs for each neural attribute and using anchor primitives, Scaffold-GS achieves a representation that consumes $5\times$ less memory for storage [21].
As a further revision to 3D-GS algorithm, Kheradmand et al. [17] propose visioning 3D Gaussians as random samples drawn from a probability distribution (3D-MCMC) that represents the physical scene. By integrating the framework of Markov Chain Monte Carlo (MCMC), specifically using Stochastic Gradient Langevin Dynamics, the requirement for complex optimization process is eliminated. 3D-MCMC introduces a noise term to the 3D Gaussian updates for exploration while allowing for a probabilistic approach to optimizing the placement and properties of the Gaussians, removing the need for heuristic methods. As a result, it replaces heuristic-based pruning and densification while achieving superior visual quality and robustness compared to seminal 3D-GS [15]. Further details of 3D-MCMC will be described in Section 3.2 in depth.

2.1.2 Hierarchical Scene Structuring and Gaussian Ordering

As an additional improvement over Scaffold-GS [21], Octree-GS [30] further evolves the hierarchical structure by introducing Level-of-Detail (LoD) techniques. To create an LoD structure, they utilize an Octree [23] representation for anchor points that were proposed in Scaffold-GS [21]. In other words, they create K levels of octree by quantizing voxel centers $\mathbf{V} \in \mathbb{R}^{M \times 3}$,

$$\mathbf{V} = \left\{ \left\lceil \frac{\mathbf{P}}{\epsilon} \right\rfloor \cdot \epsilon, \ldots, \left\lceil \frac{\mathbf{P}}{\epsilon/(2^{K-1})} \right\rfloor \cdot \epsilon/(2^{K-1}) \right\} \tag{2.2}$$

where $\mathbf{P}$ are the anchor point locations. During novel view synthesis, they make use of the Octree bounding box size to adaptively select the anchor points that will be used for rasterization in order to reduce the computational complexity and alleviate rendering inefficiency in favor of more important anchor points. Later on, the anchor points are used to estimate the neural Gaussian parameters similar to Scaffold-GS to perform rasterization.

Likewise to extension of Scaffold-GS with an LoD structure in Octree-GS [30], the seminal 3D-GS [15] has been also extended with an LoD structure for the purpose of novel view synthesis in extremely large scenes [16]. Although the work introduces many novel ideas for the purpose of efficient novel view synthesis and representing very large scenes, hierarchy generation is an influential idea which can be further extended for more efficient scene representations. The authors implemented a hierarchical LoD structure that allows parallel processing of smaller chunks of a large dataset. In order to generate a tree-based hierarchy with interior and leaf nodes, they define leaf nodes as the actual Gaussian primitives and intermediate nodes as 3D Gaussians that are built from merging the children attributes. The merging process for the intermediate node positions and covariances is defined as

$$\mu^{(l+1)} = \sum_i w_i \mu_i^{(l)} \tag{2.3}$$

$$\Sigma^{(l+1)} = \sum_i w_i \left(\Sigma_i^{(l)} + (\mu_i^{(l)} - \mu^{(l+1)})(\mu_i^{(l)} - \mu^{(l+1)})^\top \right) \tag{2.4}$$

where $\mu^{(l+1)}$ is the position of intermediate node, $\mu_i^{(l)}$ is the i-th children of the specified intermediate node, and w_i are the normalized weights based on the contribution of each child. The dependency is also similar for the covariance, $\Sigma^{(l+1)}$. The contribution of each child, w_i is calculated based on the child's opacity and surface area,

$$w_i' = o_i \sqrt{|\Sigma_i'|}, \qquad w_i = \frac{w_i'}{\sum_j^N w_j'} \tag{2.5}$$

where o_i is the opacity of the i-th child and $\sqrt{|\Sigma_i'|}$ is estimated using the surface of a Gaussian from 3-σ standard deviation level of a 3D Gaussian ellipsoid. With a similar weighting strategy, opacity and spherical harmonics of children are also merged to create the attributes of parent intermediate nodes,

$$o^{(l+1)} = \sum_i w_i o_i^{(l)}, \qquad \mathrm{SH}^{(l+1)} = \sum_i w_i \mathrm{SH}_i^{(l)} \tag{2.6}$$

where o_i is the opacity and SH_i is the spherical harmonic coefficients of the i-th child.

As a result of merging and tree hierarchy generation, different levels of the tree can be utilized for novel view synthesis or granularity-based thresholds can be selected to choose only a subset of the tree structure. In this way, fewer Gaussians would be utilized to synthesize a novel view. This structure can be further utilized for inter-prediction of leaf nodes, i.e., actual Gaussian primitives.

2.1.3 Gaussian Primitive Pruning

Gaussian primitive pruning and densification are important steps of primitive optimization in 3D-GS algorithm, referred as “adaptive density control”. Since pruning and densification are counter-acting steps where one removes Gaussians and other adds them, the densification step is criticized for being suboptimal and producing redundant Gaussians [20]. These steps have been further investigated by many research [39, 27, 20, 4] to find more robust approaches that do not require hyperparameter tuning and yield more compact representations.
To promote a smaller representation in terms of storage requirements, Gaussian primitive pruning is an important step, since the storage complexity of 3D-GS is linear in number of primitives. For that purpose, RadSplat [27] is a highly relevant work for efficient pruning. Although the main idea of the work extends to an improved version of 3D-GS [15] by providing supervision for optimization through the ZipNeRF [3] prior, they also introduce a novel pruning technique which preserves the visual quality while reducing the number of Gaussian primitives. Their pruning method, which we refer as “RadSplat pruning”, is applied only once or twice throughout the entire training in order to only keep Gaussian primitives that have a maximum pixel contribution over a predefined threshold $t_{prune} \in [0,\ 1]$. Further details of “RadSplat pruning” method will be introduced in Section 3.3.
On the other hand, Lee et al. [20] focuses on pruning Gaussian primitives by using a learnable masking strategy and introduce a compact representation of Gaussian attributes without compromising the quality of rendered images. By utilizing a volume-based learnable masking strategy, they mask out small (low scaling magnitude) and low-opacity Gaussian primitives. A learnable mask m helps to determine whether a Gaussian should be removed. In addition, a masking loss, L_m, is introduced to balance the rendering accuracy and pruning strength by penalizing the mask magnitude. The details of the learned masking method will be elaborated in Section 3.3.

2.1.4 Gaussian Splatting Compression

Since the problem of storage complexity is one of the most important problems related to 3D-GS [11], there has been multitude of recent work [20, 10, 6, 28, 39, 26] on compressing the Gaussian primitive attributes in a short period of time.
As one of the first works on 3D-GS compression, LightGaussian [10] can provide approximately $15\times$ reduction in the scene representation without decreasing the visual quality. Their methodology involves calculating a global significance score per Gaussian based on its contribution to the rendering and pruning of insignificant Gaussians. Additionally, they distill spherical harmonic coefficients, which comprise a large part of the storage requirement. This distillation involves using full-degree spherical harmonics as teacher models and supervising truncated low-degree spherical harmonics with a distillation loss, $L_{distill}$,

$$L_{distill} = \frac{1}{HW} \sum_{i=1}^{HW} \|C_{teacher}(r_i) - C_{student}(r_i)\|^2 \tag{2.7}$$

where $C(r_i)$ corresponds to pixel intensities with *teacher* and *student* models. Furthermore, to compress the Gaussian attributes excluding Gaussian positions, they utilize vector quantization for which they acquire K codes per Gaussian attribute. After initializing K vectors in the codebook via K-means algorithm, they perform batchwise expectation maximization while updating cluster means using moving average rule. Finally, they state that the Gaussian positions are compressed using lossless G-PCC [22] algorithm, although their open-source code utilize DEFLATE compression algorithm [9].
Similarly, Niedermayr et al. [26] focus on creating compact codebooks in a sensitivity-aware manner. The creation of compact codebooks depends again on perform K-means in a sensitivity-aware manner by considering feature gradients so that the cluster means are updated favoring higher maximum sensitivity of

Gaussian primitives. After constructing the codebooks using sensitivity-aware K-means, the codebooks are compressed using DEFLATE algorithm [9] for entropy coding.
Another notable technique involves a focus on reducing the number of Gaussian primitives and keeping compact codebooks for Gaussian attributes [20]. For the pruning of Gaussians, Lee et al. propose using the learnable Gaussian volume mask to prune away low-opacity and small Gaussians. Moreover, they utilize residual vector quantization (R-VQ) [38] in contrast to the vector quantization-based methods discussed above to reduce computational complexity [20]. Finally, instead of using spherical harmonics which require high amount of parameters, Lee et al. utilize a hash-grid based neural field followed by a small MLP to represent view-dependent colors. Instead of training the compression algorithm end-to-end, they apply R-VQ and learn codebooks for the last 1000 iterations.
Since the problem with the memory footprint of 3D-GS has been evident, the authors of seminal work [15] suggested follow-up improvements to reduce the memory footprint [28]. Specifically, they propose size reduction through a better pruning, better choice of spherical harmonic degrees of Gaussian primitives, and codebook-based quantization. Firstly, to eliminate spatial redundancy, the number of overlapping primitives in a spherical region is counted for a specific primitive. If the redundancy score which is calculated based on the number of overlapping primitives is above an adaptive threshold determined for overall scene, the Gaussian primitive is pruned. Secondly, they reduce the degree of spherical harmonics for redundant harmonic coefficients. Since not all Gaussians require view-dependent effects, a base color is sufficient for most of them. For that reason, the color of a Gaussian is evaluated from all viewpoints and only the base color is used if the color has low variance from all viewpoints. Finally, they utilize K-means clustering to store only the index to the closest value where $K = 256$ for their implementation. For the attribute codebooks, one for opacity, one for scaling components, two for real and imaginary parts of rotation, one for the base color, and one per spherical harmonics coefficients are created.
Although previously provided improvements for Gaussian splat compression provide low complexity solutions, follow-up improvements on 3D-GS such as Scaffold-GS [21] offer for higher compression possibilities. One work that uses the compact hierarchy of Scaffold-GS is HAC [6]. HAC leverages hash grids to capture spatial correlation among unorganized "anchor points". This structure allows for efficient context modeling and entropy coding. Unlike previous works, the use of context modeling with learned entropy models allows for higher compression efficiency. Specifically, they assume a Gaussian distribution for the features with hyperprior information which estimates the feature statistics [25] and perform an adaptive quantization step to perform learned compression. Finally, the hash grids are also binarized and included in bit consumption and loss calculation using straight-through estimation [33].
Using a similar approach, ContextGS [35] uses an autoregressive context model at anchor level by building a framework over Scaffold-GS [30] while having a high similarity to Octree-GS [30]. Anchors in coarser levels help to predict the distribution of anchors at finer levels by creating a context that significantly improves entropy coding efficiency. ContextGS [35] also incorporates a quantized hyperprior feature for each anchor to further compress the representation. The proposed method can achieve a compression ratio of more than 100x compared to standard 3D-GS [35].

2.2 Point Cloud Compression

As the Gaussian splat representation draws a high similarity to point-cloud representations, the point-cloud compression methods has been influential in terms of compressing the positioning of 3D-GS representation. Specifically, DEFLATE [9] has been a significant compression algorithm for multiple works [10, 26]. DEFLATE is a lossless data compression algorithm that combines the LZ77 algorithm and Huffman coding. While the LZ77 algorithm replaces repeated occurrences with references to previous copies in the uncompressed data stream, Huffman coding performs entropy coding to assign short codes to frequent symbols.

On the other hand, G-PCC [22] as an MPEG compression standard has been adopted for use by [10] as reported in their paper. However, as a proprietary point cloud compression method, this method is not included in their open-source code. As a result, their shared work has been also using DEFLATE algorithm for Gaussian primitive position compression.

2.3 Learned Entropy Modeling

Although the literature for learned entropy modeling is vast, we are only interested in two early models for our main purpose of Gaussian primitive compression with two extensions: residual coding from traditional video compression literature and multi-rate image compression method proposed by Cui et al. [8].
As one of the first works on learned image compression, Ballé et al. [1] proposed using a learned counterpart of an image with a unique hyperprior entropy model, named fully factorized entropy model. The fully-factorized entropy model utilizes a neural network structure to estimate the cumulative mass function (CMF) of a probability distribution while constraining it to have value 0 at negative asymptote, value 1 at positive asymptote, and a non-negative probability mass function over its domain. Using the learned CMF, the entropy coding can be performed more efficiently by giving shorter bitstrings to symbols that occur frequently as long as probabilities are estimated accurately. On the other hand, the fully-factorized entropy model cannot capture statistical dependencies in latent distribution, leading to an overhead in compressed representations [1]. The model details of fully-factorized entropy model are provided in depth throughout Section 3.4.1.
For an important extension to the fully-factorized entropy model, Minnen et al. [25] propose a hierarchical extension, which we refer to as "mean-scale hyperprior" network. The proposed hierarchical extension introduces a hierarchical prior over the distribution of Gaussian primitive attributes and use the compressed hyperprior as side information about the entropy parameters of the assumed Gaussian distribution. The hierarchical prior is achieved by introducing a second subnetwork which learns a probabilistic model of quantized latents by generating mean and standard deviation parameters of the conditional Gaussian entropy model over quantized latents [25]. The architecture details will be introduced in Section 3.4.2
To achieve further efficiency and make use of similarities between frames, standards-based and learned video compression methods make use of residual frame coding [31]. In the context of video compression, the residual frame is calculated by subtracting a reference or prediction frame from the frame of interest. By subtracting the prediction, the residual frame generally achieves a lower entropy than the target frame itself. This procedure can be further utilized for Gaussian splatting representation compression as it will be discussed in Section 3.5.
Finally, Cui et al. [8] achieve a multi-rate entropy model training using channel-wise quantization steps learned prior to entropy coding. In this way, the Lagrangian parameters for the rate-distortion trade-off can be linked with their respective scaling vectors that adjust the quantization steps while the rest of the model is shared for all the rate-distortion trade-off levels [8]. Although this method does not introduce any gains in terms of coding efficiency or visual quality, it can decrease the training time by more than $4\times$ as only one training becomes sufficient compared to training a separate model for each rate-distortion trade-off level.

Chapter 3

Materials and Methods

3.1 Background: 3D Gaussian Splatting

3.1.1 Gaussian Primitive Structure

3D Gaussian Splatting (3D-GS) [15] builds a scene representation utilizing Gaussian primitives with their mean μ, covariance Σ, and appearance attributes, color c and opacity $\sigma(o)$.

$$\tau(\mathbf{G}) = \exp\left(-\frac{1}{2}(\mathbf{G}-\mu)^\top \Sigma^{-1}(\mathbf{G}-\mu)\right) \tag{3.1}$$

Since it is analogous to represent the covariance matrix Σ, using a scaling matrix S, and quaternion, R, each Gaussian primitive is associated with its own scaling and quaternion to optimize them independently. In that case, scaling parameters are 3 scalar values for the diagonal matrix S, and quaternions are 4 dimensional vectors to capture the rotation of a Gaussian primitive. This way, the positive semi-definiteness constraint on covariance matrix can be imposed while performing gradient descent optimization,

$$\Sigma = RSS^\top R^\top \tag{3.2}$$

where the quaternion is converted into rotation matrix R while making sure to normalize them to have valid quaternion. Scaling matrix S is formed using a scaling vector on the diagonal of an identity matrix.

In addition, instead of using a 3 dimensional color attribute per Gaussian primitive, the appearance of a Gaussian primitive is modeled using spherical harmonics (SH) [14] of a preset maximum degree in order to capture the view-dependent color changes. This helps to improve the visual quality by modeling non-Lambertian effects such as specular reflections.

During rendering, the color of a Gaussian can be efficiently calculated based on the viewing direction. The SHs are evaluated using following formula with respect to SH degree l, viewing angle parameters θ and ϕ,

$$Y_l^m(\theta,\phi) = \frac{(-1)^l}{2^l l!}\sqrt{\frac{(2l+1)(l+m)!}{4\pi(l-m)!}}\exp(im\phi)(\sin\theta)^{-m}\frac{d^{l-m}}{d(\cos\theta)^{l-m}}(\sin\theta)^{2l} \tag{3.3}$$

where $-l \leq m \leq l$ for a given SH degree l. By default, 3D-GS makes use of maximum SH degree $l_{max} = 3$. In turn, a single Gaussian has 3 coefficients for degree 0 (due to red, green, and blue color channels) while degree 3 has 21 coefficients due to the SH coefficients ranging from levels -3 to $+3$.

The final color from the viewing angle θ, and ϕ is acquired through a weighted-average of Y_l^m,

$$f_l^m = \int f(\theta,\phi) y_l^m(\theta\phi)\ d\theta\ d\phi \tag{3.4}$$

$$c = \sum_{l=0}^{l_{max}} \sum_{m=-l}^{l} f_l^m Y_l^m(\theta,\phi) \tag{3.5}$$

3.1.2 Differentiable Rasterization

As in the name of the method, the "splatting" operation is performed to project the Gaussian primitives on the image plane and rasterize them in order to generate the final image. For that purpose, the calculated covariance matrix Σ, and position μ, are projected onto 2D image plane using camera extrinsic parameters W, and camera intrinsic parameters K. The projection of covariance and position to 2D is highly efficient compared to ray-tracing based methods,

$$\mu' = K\left(W\mu/((W\mu)_z)\right) \tag{3.6}$$

$$J = \frac{\partial \mu'}{\partial \mu} \tag{3.7}$$

$$\Sigma' = JW\Sigma W^\top J^\top \tag{3.8}$$

As a result, the impact of a Gaussian primitive on a pixel p, is calculated as,

$$f_i(p) = \sigma(o_i)\exp\left(-\frac{1}{2}(p-\mu_i')^\top(\Sigma_i')^{-1}(p-\mu_i')\right) \tag{3.9}$$

On the other hand the image formation is similar to NeRF-based novel view synthesis [24] in the sense that Gaussian splatting also makes use of volumetric rendering formula to calculate the color for a pixel $C(p)$,

$$C(p) = \sum_{i=1}^{N} c_i(1-\exp(\sigma_i\delta_i))\exp\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right) \tag{3.10}$$

$$= \sum_{i=1}^{N} c_i(1-\exp(\sigma_i\delta_i))\prod_{j=1}^{i-1}\exp(-\sigma_j\delta_j) \tag{3.11}$$

$$= \sum_{i=1}^{N} c_i(1-\exp(-\sigma_i\delta_i))T_i \tag{3.12}$$

where c_i is the color, σ_i is probability of hitting a particle, δ_i is the distance from previous point, and T_i is the transmittance. The volume rendering equation requires some adjustment for Gaussians. To draw a similarity with Gaussian primitives, the density of 2D Gaussian primitives is utilized together with their opacity. Specifically, using Equation 3.9 and opacity, we acquire the following volume rendering function,

$$C(p) = \sum_{i\in N} c_i f_i(p)\prod_{j=1}^{i-1}(1-f_j(p)) \tag{3.13}$$

This process is highly efficient, because projections of 3D Gaussians to 2D image plane are also Gaussian as seen in Equation 3.9. In addition, the projections and sorting of Gaussians can be done on a GPU in a parallelized fashion prior to image rasterization so that a re-projection is not required. The parallelized sorting stage is performed on 16×16 tiles where a single Gaussian can appear in multiple tiles.

Finally, and crucially, since the procedure does not require ray-sampling, each Gaussian along a ray has a significant impact and does not pose redundancy. These results yield an efficient, real-time algorithm which can be highly parallelized since the Gaussian primitives are independent of each other, and the projection is just completed with a couple of matrix multiplications.

3.1.3 Storage Complexity of Gaussian Primitives

Although the computation efficiency and high parallelizability of 3D-GS admits a very feasible approach, the storage of Gaussian primitives yield an important problem. Based on scene complexity, a regular scene might require from a few hundreds of thousands to millions of Gaussians. Recognizing the fact that each Gaussian requires in total 59 parameters which are stored in single-precision floating-point format (*float32*), a scene with 1 million Gaussians requires 225 MB memory. This memory requirement is significantly larger compared to network-based novel view synthesis methods.

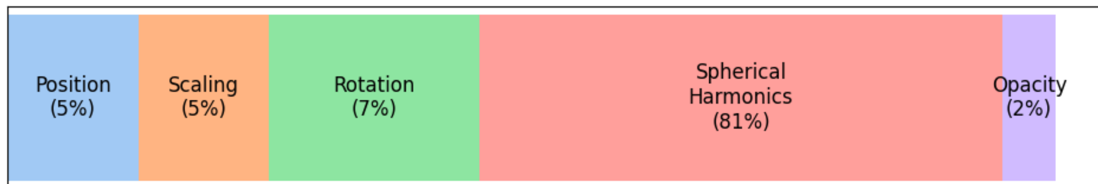

Figure 3.1: **The parameter distribution of Gaussian primitives.** The spherical harmonics with maximum degree of 3 require 48 floating point values while position, scaling, rotation, and opacity require 3, 3, 4, 1 floating point values, respectively. This situation depicts the importance of high compression for spherical harmonics.

When comparing the components of Gaussian primitives based on Figure 3.1, it becomes evident that large part of the Gaussian primitives is composed of spherical harmonics for accurate view-dependent visualization. On the other hand, geometric information such as position, scaling, and rotation tend to be highly important for visually plausible images with correct localization.

3.1.4 Optimization of Gaussian Primitives

The optimization process with adaptive density control is a vital part of 3D-GS since Gaussians need to adhere to certain conditions, i.e. semi-definiteness of covariance while achieving close-to-optimal parameters for rasterization. Ensuring the accuracy of the scene representation, this process preserves the efficiency of the approach by pruning and densifying the Gaussian primitives.
Optimization process is initialized from a set of sparse points obtained from COLMAP [32] with Structure-from-Motion (SfM) [34] algorithm. Using the sparse points as initial positions for Gaussians, the base color is assigned to 0-th degree components of the 0-th degree of the spherical harmonics, while the opacity is initialized with 0.1, scaling initialized with the distance to the closest position, and the rotation is initialized with 0 vector.
Throughout the training process, the differentiable nature of rasterization described in Section 3.1.2 allows the performing of gradient descent updates on the attributes of Gaussian primitives while also spawning new Gaussians and pruning unimportant ones. Specifically, the loss function utilized for optimization is a combination of L1-loss and structural dissimilarity index measure (D-SSIM) on training views synthesized using Gaussian primitives and ground truth view,

$$\mathcal{L} = (1 - \lambda_{dssim}) \times \text{L1}(f(\hat{x}), f(x)) + \lambda_{dssim} \times \text{D-SSIM}(f(\hat{x}), f(x)) \tag{3.14}$$

where $f(\hat{x})$ is the rasterized image from Gaussian primitives and $f(x)$ is the ground truth image. For the adaptive densification and pruning of Gaussian primitives, the optimization process splits, clones or prunes Gaussians at every 100 iterations until $15,000$ iterations.

Pruning. In order to prune Gaussians, the opacity of Gaussian primitives are considered. Specifically, if a Gaussian primitive achieves an opacity below a threshold, 0.005, the Gaussian primitive is removed entirely since it does not contribute much to the novel view synthesis. Furthermore, if a Gaussian has a radius greater than a predefined scene extent, the point is removed to reduce the rasterization complexity and potentially improve the optimization process.

Densification. If a Gaussian has a high gradient with respect to its position, this indicates that the Gaussian can benefit from being split or cloned into two separate Gaussians to simplify the optimization process, i.e., reduce the gradient magnitude. A Gaussian is chosen for densification if the magnitude of the position gradient, $\partial\mathcal{L}/\partial\mu$, is above a threshold, 0.0002. In that case, the Gaussian is splitted if its maximum scale (equivalently, maximum eigenvalue magnitude of covariance) is above a specified threshold, 1% of camera extent. By splitting a Gaussian, the scale of the split parts of the Gaussian is divided by 1.6 while keeping the rotation and the rest of the parameters the same. On the other hand, if the maximum scale is below the specified threshold, the Gaussian is cloned entirely with all its parameter kept same. The overall optimization process can be better visualized in the decision tree in Figure 3.2.

Although the described algorithm works well in practice, it is well-known for producing redundant Gaussians with little contribution. For that purpose, 3D-MCMC [17] is proposed for a better optimization procedure and described in Section 3.2.

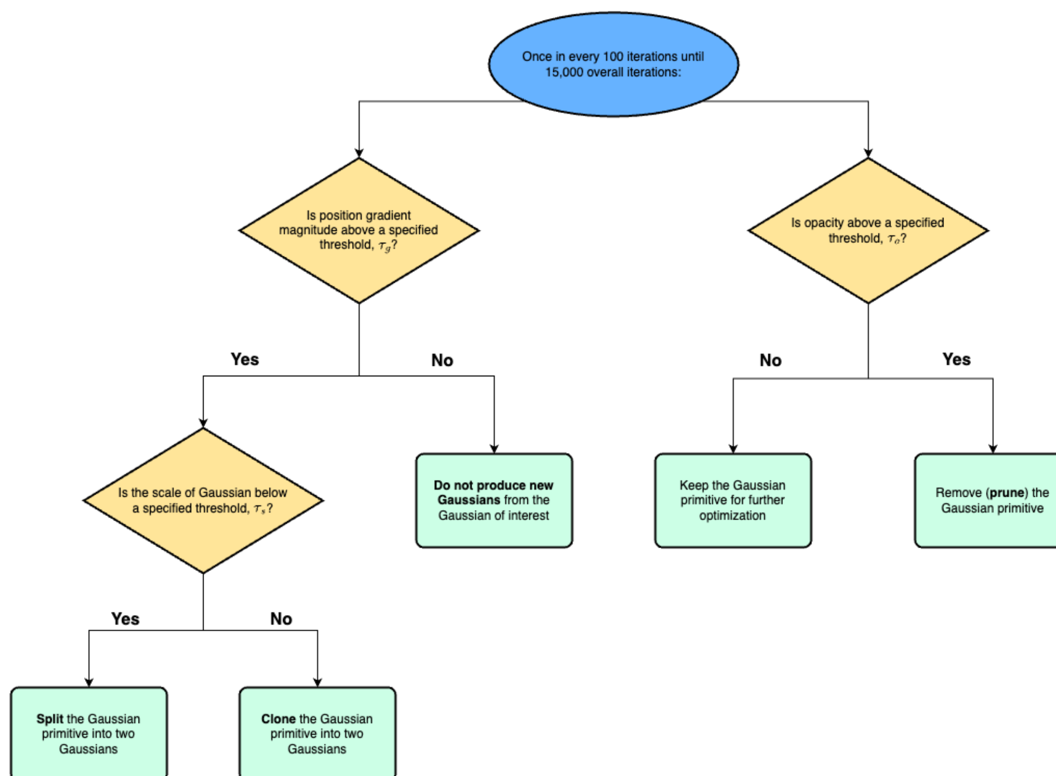

Figure 3.2: **Gaussian splatting densification and pruning algorithm in a decision tree format**. Gaussian splatting algorithm requires adding or removing Gaussian primitives periodically (once in every 100 iterations) using the described algorithm. The hyperparameters for optimization are used as is from the original 3D-GS descriptions [15].

3.2 Improving 3D-GS through Markov Chain Monte Carlo (MCMC)

As the optimization procedure introduced with 3D-GS [15] is prone to generate many redundant or ineffective Gaussians, several works [39, 27, 20, 4] proposed to reduce the number of Gaussians in different ways. Recently, Kheradmand et al. [17] proposed considering Gaussian primitives as samples from a probability distribution through a simpler process involving "relocalizing" Gaussians and regularizing Gaussian scales and opacities to achieve improved rendering quality and a more robust optimization procedure without heuristic-based methods.
As the updating method for Gaussian primitives makes use of the Stochastic Gradient Descent, it can be converted into a Stochastic Gradient Langevin Dynamics (SGLD) update through the introduction of noise in an MCMC framework. As a result, regions in a scene can be explored naturally while reducing the dependency on initialization. The conversion from SGD to SGLD requires updating the update steps in the following form with noise introduction,

$$\theta \leftarrow \theta - \lambda_{lr} \cdot \nabla_\theta \mathbb{E}_{\mathbf{I}\sim\mathcal{I}}\left[\mathcal{L}(\theta;\ \mathbf{I})\right] + \lambda_{noise} \cdot \epsilon \tag{3.15}$$

The full loss function, $\mathcal{L}$, is designed as,

$$\mathcal{L} = (1 - \lambda_{dssim}) \cdot \text{L1}(\hat{x}, x) + \lambda_{dssim} \cdot \text{D-SSIM}(\hat{x}, x) + \lambda_o \cdot \sum_i |o_i|_1 + \lambda_\Sigma \cdot \sum_{ij} \left| \sqrt{\text{eig}_j(\Sigma_i)} \right|_1 \tag{3.16}$$

where $\hat{x}$ is the rasterized image, x is the ground truth image, o_i is the i-th Gaussian's opacity parameter, and Σ_i is the covariance matrix of i-th Gaussian. The hyperparameters are set as $\lambda_{dssim} = 0.2$, $\lambda_{lr} = 1.6e-4$, $\lambda_{noise} = 5 \times 10^5$, $\lambda_\Sigma = 0.01$, and $\lambda_o = 0.01$. Note that the designed loss function is different from 3D-GS [15] loss function in Equation 3.14 as additional terms for opacity, o_i and covariance Σ_i promote lower opacity and lower scaling parameters for Gaussians. Since Gaussians with lower opacity and lower scale Gaussians are pruned later on, the new loss function regularizes non-useful Gaussians to have lower values for opacity and scaling.
In order to design the additive noise term, ϵ, special consideration is required to prevent forcing accurate Gaussians out of support regions while allowing them to move out for inaccurate ones. For that reason, a special noise term is defined to promote low opacity and high scale Gaussians to be exposed to larger noise,

$$\epsilon = \lambda_{lr} \cdot \sigma(-k(o - t)) \cdot \Sigma\eta \tag{3.17}$$

where $\eta \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, σ is sigmoid function, $k = 100$, and $t = (1 - 0.005)$ for sharp transitions.
Another important contribution is through reinterpreting the pruning and densification steps as "relocation" of "dead" ($o_i < 0.005$) Gaussians. Specifically, the dead Gaussians are relocated to live Gaussians using multinomial sampling based on probabilities proportional to opacities of live Gaussians.
However, such cloning operation should preserve the state probabilities (i.e. $\mathcal{P}(\mathbf{g}_{old}) \approx \mathcal{P}(\mathbf{g}_{new})$) to prevent MCMC sampling from collapse into a few states where the state is defined as all attributes of a Gaussian. To satisfy this, the relocation strategy takes $N-1$ Gaussians satisfying the "relocation" condition from their respective states $\mathbf{g}_i$ and relocates them to a state of another Gaussian, $\mathbf{g}_N$. The updated parameters of $N-1$ Gaussians after relocation are derived as following,

$$\mu^{new}_{1,\ldots N} = \mu^{old}_N \tag{3.18}$$

$$o^{new}_{1,\ldots N} = 1 - \sqrt[N]{1 - o^{old}_N} \tag{3.19}$$

$$\Sigma^{new}_{1,\ldots N} = \left(o^{old}_N\right)^2 \left(\binom{i-1}{k} \frac{(-1)^k (o^{new}_N)^{k+1}}{\sqrt{k+1}} \right)^{-2} \Sigma^{old}_N \tag{3.20}$$

For densification, the number of Gaussians are increased by 5% at each iteration until a preset maximum number of Gaussians, N_{max}. Similar to "relocation", new Gaussian primitives are created at locations using multinomial sampling based on probabilities proportional to opacities of live Gaussians.

3.3 Gaussian Primitive Pruning

Following the general consensus with previous compression methods for 3D-GS [20, 26, 6], we apply pruning to Gaussian primitives to reduce the storage complexity, which is linear in number of primitives.
For that purpose, we evaluated the impact of the "RadSplat pruning" [27] technique that preserves visual quality while reducing the number of Gaussian primitives. The idea with RadSplat pruning is to keep Gaussian primitives that have a maximum pixel contribution over a predefined threshold, $t_{prune} \in [0,\ 1]$,

$$f_i(p) = \sigma(o_i) \exp\left(-\frac{1}{2}(p - \mu_i')^\top (\Sigma_i')^{-1}(p - \mu_i')\right) \tag{3.21}$$

$$T_i(p) = \prod_{j=1}^{i-1}(1 - f_j(p)) \tag{3.22}$$

$$m_i = m(p_i) = \mathbb{1}(\max_{r \in I_f} f_i^r T_i^r < t_{prune}) \tag{3.23}$$

where μ_i' is the 2D projected position, Σ_i' is the 2D projected covariance, $r \in I_f$ are all rays forming all pixels in the training dataset, and $f_i^r T_i^r$ is the ray contribution of i-th Gaussian primitive for the ray r during rasterization of splatted Gaussians. We apply RadSplat pruning only twice throughout the entire training, at $16K$ and $24K$ iterations. For the maximum pixel contribution threshold, we set 0.01 to remove Gaussian primitives which contribute less than this amount. For the rest of the training parameters, we keep everything same as proposed in 3D-MCMC [17] for a fair comparison with the case without pruning.
As a second method for pruning, we evaluate the learnable masking strategy [20] and mask out small (low scaling magnitude) and low-opacity Gaussian primitives. A learnable mask m helps to determine whether a Gaussian should be removed. The mask is generated as follows,

$$M_n = \text{stop-gradient}\left(\mathbb{1}\left[\sigma(m_n) > \epsilon\right] - \sigma(m_n)\right) + \sigma(m_n) \tag{3.24}$$

where "stop-gradient" operator ensures gradients are not propagated through binary decision process, σ is sigmoid operator, m_n is the learnable mask parameter, and ϵ is the user specified threshold. Note that M_n is a binary matrix that can achieve the values $M_n \in \{0,\ 1\}$. Using calculated mask, we retrieve the masked scaling and opacity vectors as,

$$\hat{s}_n = M_n s_n, \qquad \hat{o}_n = M_n o_n \tag{3.25}$$

which effectively removes a Gaussian primitive by making it invisible and suitable for pruning. In addition, a masking loss, L_m, is introduced to balance the rendering accuracy and pruning strength by penalizing the mask magnitude,

$$\mathcal{L}_m = \frac{1}{N}\sum_{n=1}^{N} \sigma(m_n) \tag{3.26}$$

3.4 Learned Entropy Models for 3D Gaussian Primitive Compression

Although the previous 3D-GS compression methods discussed in Section 2.1.4 allowed significant compression ratios, the majority of them (except for the concurrent work HAC [6] and Context-GS [35]) do not utilize learned entropy models for Gaussian primitive compression. As appears in image / video compression, the learned codecs managed to surpass the traditional codecs in recent years [13]. Thus, it is sensible to utilize learned codecs also for the purpose of Gaussian primitive compression. Consequently, inspiration

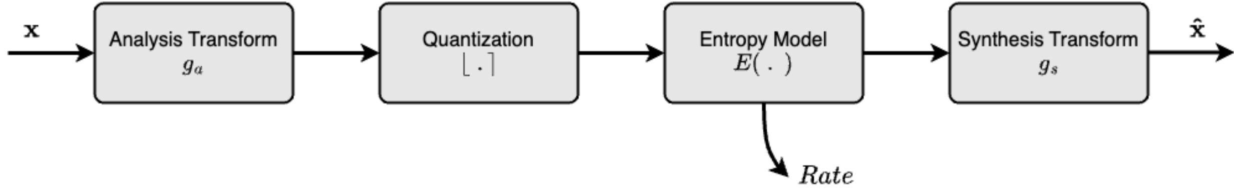

Figure 3.3: **An overview of lossy compression pipeline in high level.** A compression pipeline is composed of an analysis transform to map signals to a low-entropy representation, quantization for applicability, entropy modeling for efficient coding, and synthesis transform to map signal back into domain of interest.

can be drawn from the image compression literature to build learned entropy models and improve the coding efficiency for 3D-GS.

As depicted in Figure 3.3, for lossy image/video compression four main building blocks are required: an analysis transform, quantization, entropy model for entropy coding, and synthesis transform. To optimize the bitrate while also keeping a high quality signal, we require to perform rate-distortion optimization with a join loss function. In the context of rate-distortion optimization, rate is the expected code length which can be written as cross-entropy when using the entropy coding technique [1],

$$Rate = \mathbb{E}_{\mathbf{y} \sim p_{\mathbf{y}}} \left[-\log_2 p_{\hat{\mathbf{y}}} \left(\lfloor \mathbf{y} \rceil \right) \right] \tag{3.27}$$

where $p_{\hat{\mathbf{y}}}$ is the entropy model, and $\lfloor \cdot \rceil$ is the quantization operator. However, as quantization is a non-differentiable operation, the quantization step is approximated with a noisy representation $\tilde{\mathbf{y}}$ during training whereas $\hat{\mathbf{y}}$ stands for actual quantized representation. Accordingly, for a given entropy model $p_{\hat{\mathbf{y}}}$, the rate-distortion loss function is calculated as,

$$\mathcal{L} = D(f(\mathbf{x}), f(\hat{\mathbf{x}})) + \lambda \; Rate \tag{3.28}$$

where $f(\cdot)$ is the rasterization function from Gaussian primitives $\mathbf{x}$, D is the distortion measure between ground truth image and prediction rendered image, and λ controls for the trade-off between rate and distortion. As our distortion metric, we utilize the same distortion metric as 3D-GS [15] and use a weighted combination of image L1-loss together with image D-SSIM loss as in Equation 3.14. In addition, we regularize the Gaussian primitive attributes using the optimized attributes so that training for compression does not cause a large deviation from optimized values,

$$\begin{aligned} \mathcal{L} = (1 - \lambda_{dssim}) \cdot \text{L1}(f(\hat{\mathbf{x}}), f(\mathbf{x})) + \lambda_{dssim} \cdot \text{D-SSIM}(f(\hat{\mathbf{x}}), f(\mathbf{x})) + \lambda \; Rate \\ + \lambda_{rec} \left[\text{L1}(\hat{s}, s) + \text{L1}(\hat{r}, r) + \text{L1}(\hat{o}, o) + \text{L1}(\widehat{SH}, SH) \right] \end{aligned} \tag{3.29}$$

where $\lambda_{dssim} = 0.2$, $\lambda \in \{0.005,\ 0.001,\ 0.0001\}$, and $\lambda_{rec} = 0.2$. Note that for the case of Gaussian primitive compression, $\mathbf{x} \in \mathbb{R}^{59}$ since each Gaussian primitive has 59 scalar attributes and rate is calculated over the quantized representation of these attributes except position, μ. The reason for excluding μ from compression with the learned entropy model is that the position information is highly sensitive to noise. For our experiments, we randomly sub-sample the Gaussian primitives for compression since compressing over $1M$ Gaussian primitives in each iteration is not feasible and slows down training significantly. For that reason, we subsample 10% of Gaussian primitives for compression and only backpropagate over these samples in respective iteration. For the optimization process of both the Gaussian primitive parameters and the learned entropy model parameters, we utilize Adam optimizer [18] with a learning rate of 0.01, $\beta = (0.9,\ 0.999)$ for running averages.

3.4.1 Fully-Factorized Entropy Modeling

A fully-factorized entropy model estimates the probability distribution for a latent representation [1] while assuming independence of features. The probability mass function (PMF) $p_{\tilde{\mathbf{y}}} : \mathbb{Z} \rightarrow [0,\ 1]$ is used for feature-wise distribution modeling of input $\tilde{\mathbf{y}}$. The fully-factorized entropy model is most suitable for independent and identical distributions where features are not correlated. If such a correlation exists, fully-factorized entropy model cannot make use of the correlation.
To build a fully-factorized entropy model, the cumulative mass function (CDF) associated with the PMF is approximated using a fully-connected network of 4 layers where the CDF is not conditioned on any other information. As a result, derivative of the fully-connected network is the PMF function,

$$p_{\tilde{\mathbf{y}}|\psi} = \prod_i \left(p_{y_i|\psi^{(i)}} \left(\psi^{(i)} \right) * \mathcal{U} \left(-\frac{1}{2}, \frac{1}{2} \right) \right) (\tilde{y}_i) \tag{3.30}$$

where $\psi^{(i)}$ stand for the univariate distribution parameters and $\mathcal{U}$ is additive uniform noise to approximate quantization while allowing differentiability during backpropagation. This derivation remains highly efficient thanks to automated differentiation frameworks. As a result, the bitrate R for the loss calculation can be estimated and penalized using the above defined full-connected network with uniform additive noise ("soft quantization" [1]) on latent representation to simulate rounding operation. During inference time, rounding operation is utilized for quantization as usual.
For our purposes of Gaussian primitive compression, the fully-factorized entropy model is utilized for a part of our analysis. During Gaussian primitive compression, the optimization of analysis g_a, and synthesis g_s, transforms depicted in Figure 3.3 are found to be highly unstable since learning a mapping from and to Gaussian primitives is a difficult problem that is solved through direct optimization of Gaussian parameters in the 3D-GS framework [15].

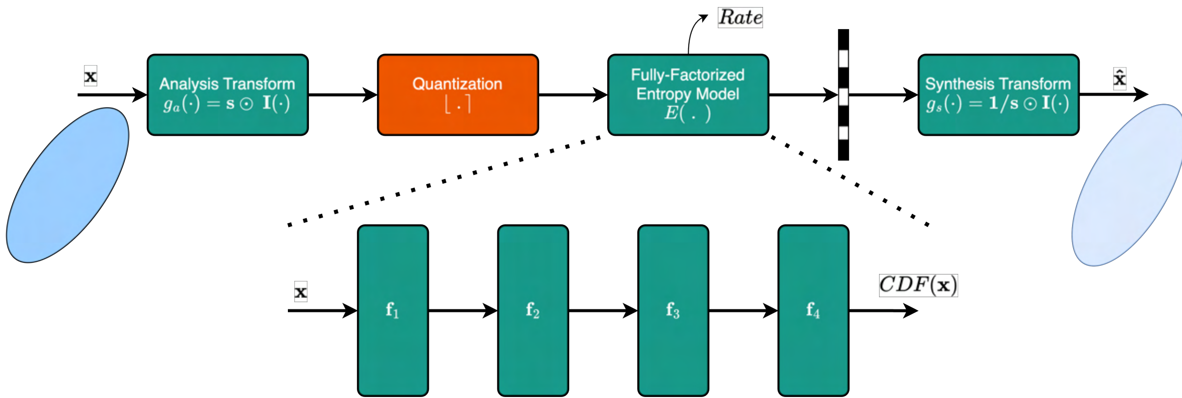

Figure 3.4: **An overview of fully-factorized entropy model.** The fully-factorized entropy model is utilized for a part of the analysis for learned Gaussian primitive compression. The green modules depict the learned components such as MLP layers or quantization parameters $\mathbf{s}$. Note that the quantization scale $\mathbf{s}$ is shared for analysis and synthesis transforms. The orange modules are the fixed operations such as quantization and entropy coding through learned entropy model.

For that reason, we only scale the input Gaussian parameters using learned parameters per input channel prior to applying quantization, denoted as $\mathbf{s}$ as depicted in Figure 3.4,

$$g_a(\mathbf{x}) = \mathbf{s} \odot \mathbf{x} \tag{3.31}$$

where $\mathbf{x}$ is a flattened version of Gaussian primitive attributes: scaling, s, rotation, r, opacity, o, and spherical harmonic coefficients, f. Note that the position of Gaussian attributes, μ, is not compressed using the learned entropy model but compressed through lossless DEFLATE [9] compression algorithm due to the high sensitivity of position information. The learned scaling parameters help us to quantize the sensitive parameters with a smaller quantization step and thus, allowing for lower distortion on them while less sensitive parameters such as spherical harmonics are quantized with larger quantization steps. For our experiments, we initialize the learned scaling parameters from 20 based on our trials.
Furthermore, for entropy modeling, the CMF of input $\mathbf{x}$ is modeled using a $K = 4$ layer MLP with a specific structure, $c = f_K \circ f_{K-1} \circ \cdots f_1$ as described in [1]. Using the network layers f_i, the features are processed so that each feature is processed by a filter with 3 channels. Again, note that the derivative of the CMF yields the PMF which can be calculated through automated differentiation, $p = f'_K \circ f'_{K-1} \circ \cdots f'_1$. For the functions described by each layer, we have,

$$f_k(\mathbf{x}) = g_k\left(\mathbf{H}^{(k)}\mathbf{x} + \mathbf{b}^{(k)}\right) \tag{3.32}$$

$$g_k(\mathbf{x}) = \mathbf{x} + \mathbf{a}^{(k)} \odot \tanh(\mathbf{x}) \tag{3.33}$$

$$f_K(\mathbf{x}) = \sigma\left(\mathbf{H}^{(K)}\mathbf{x} + \mathbf{b}^{(K)}\right) \tag{3.34}$$

In order to have a valid PMF, the derivative of CMF, $\partial c/\partial x$ is constrained by using,

$$\mathbf{H}^{(k)} = \text{softplus}\left(\hat{\mathbf{H}}^{(k)}\right) \tag{3.35}$$

$$\mathbf{a}^{(k)} = \tanh\left(\hat{\mathbf{a}}^{(k)}\right) \tag{3.36}$$

This special definition of MLP helps us to build an approximation of a valid probability distribution with CMF upper-bounded by 1, and PMF lower-bounded by 0. In order to train the learned entropy model, rate-distortion loss calculated in Equation 3.29 is used for gradient descent algorithm where rate is estimated using the learned entropy model parameters.

3.4.2 Hierarchical Entropy Modeling

Although fully-factorized entropy model [1] described in Section 3.4.1 is capable of modeling the probability distribution at relatively simpler cases, a more sophisticated entropy model is built with hierarchical priors [25]. The hierarchical entropy model, also referred as mean-scale hyperprior entropy model, exploits more structure in the latent representation compared to fully-factorized priors [1] by conditioning latent distribution on additional information shared with the decoder.
Hierarchical entropy modeling utilizes hyper-latents, $\mathbf{z}$, in order to provide more information regarding the probability distribution of the latent, $\mathbf{y}$. Specifically, the probability distribution of the latent representation of a signal is modeled using a Gaussian distribution and the parameters of the distribution (per feature mean, μ_i, and standard deviation, σ_i) are estimated using the hyper-latents while the hyper-latent distribution is modeled using a fully-factorized model [1] as described in Section 3.4.1,

$$p_{\tilde{\mathbf{y}}|\tilde{\mathbf{z}},\,\theta_{hd}} = \prod_i \left(\mathcal{N}\left(\mu_i,\ \sigma_i^2\right) * \mathcal{U}\left(-\frac{1}{2}, \frac{1}{2}\right)\right)(\tilde{y}_i) \quad \text{with } \mu_i,\ \sigma_i = g_h(\tilde{\mathbf{z}} \mid \theta_{hd}) \tag{3.37}$$

$$p_{\tilde{\mathbf{z}}|\psi} = \prod_i \left(p_{z_i|\psi^{(i)}}\left(\psi^{(i)}\right) * \mathcal{U}\left(-\frac{1}{2}, \frac{1}{2}\right)\right)(\tilde{z}_i) \tag{3.38}$$

where hyper-latent, $\tilde{\mathbf{z}}$ is estimated using the hyper-encoder network with parameters, θ_{he} and uniform noise instead of hard quantization to not disrupt differentiability. Later on, the quantized hyper-latent is decoded using the hyper-decoder, θ_{hd} to estimate the entropy parameters, $\mu_i,\ \sigma_i$ of the latent, $\tilde{\mathbf{y}}$. The decoded entropy parameters are utilized for entropy coding the latent, $\tilde{\mathbf{y}}$. The full overview of the devised lossy codec with hierarchical entropy modeling is depicted in Figure 3.5.

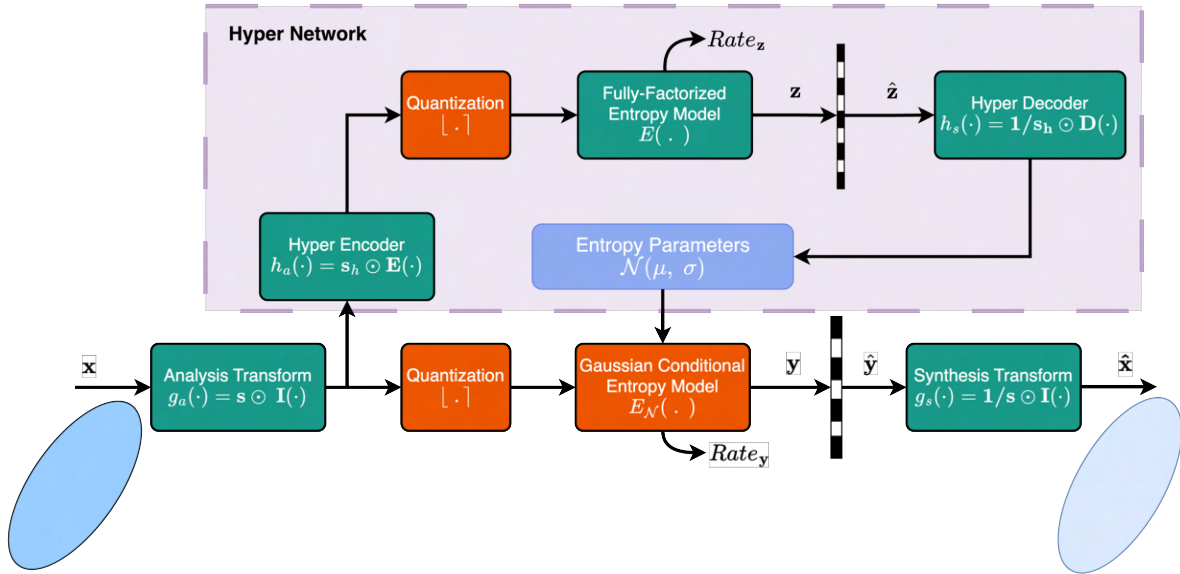

Figure 3.5: **An overview of hierarchical entropy model with Gaussian conditional probability model.** A hyper-network is utilized to model Gaussian primitive parameters using a conditional Gaussian distribution. The green modules depict the learned components such as MLP layers or quantization step sizes like $\mathbf{s}$. Note that the quantization scale $\mathbf{s}$ is shared for analysis and synthesis transforms. The orange modules are the fixed operations such as quantization and entropy coding through learned entropy model. The blue box depicts the entropy parameters estimated by the "Hyper Network".

The hierarchical entropy modeling allows for a higher quality entropy modeling compared to fully-factorized entropy modeling since the fully-factorized model fails to capture complex dependencies, assuming independent and identical distribution [25]. On the other hand, the hyperprior network with hierarchical entropy modeling can capture statistical dependencies between different parts of an image, leading to more efficient entropy modeling.

To calculate the final loss, we sum up the bitrate for both hyperprior latent, $\mathbf{z}$ and scaled Gaussian primitives, $\mathbf{y}$,

$$\mathcal{L} = D(f(\mathbf{x}), f(\hat{\mathbf{x}})) + \lambda \cdot (Rate_{\mathbf{y}} + Rate_{\mathbf{z}}) \tag{3.39}$$

where distortion is estimated same as calculated in Equation 3.29 with additional L1-loss terms to have more guidance on opacity, scaling, and rotation information of Gaussian primitives. Similar to previous derivation of rate-distortion loss, λ stands for the Lagrangian of rate-distortion trade-off. Note that we also refer to hierarchical entropy modeling as "mean-scale hyperprior" entropy model as the hyperprior network's goal is to predict the mean and standard deviation (scale) for the latent representation of Gaussian primitives.

3.4.3 Multi-rate Entropy Modeling and Rate Adaptation

As a further addition to learned entropy models, Cui et al. [8] introduce "gain units" to scale quantization steps and achieve rate-adaptation in one single model instead of training multiple models for multiple bitrates. This method can be analogously applied to Gaussian splatting representation compression.
To achieve multiple bitrates in a single compression model, multiple scaling vectors $\mathbf{s}$ (as depicted in Figures 3.4 and 3.5) are learned in one training instance. Recall that the scaling parameters in Sections 3.4.1 and 3.4.2 have the dimensionality same as the number of attributes, that is, $\mathbf{s} \in \mathbb{R}^{C}$. With multi-rate adaptation, we learn a matrix $\mathbf{S} \in \mathbb{R}^{L \times C}$ where L is the number of rate distortion levels (i.e. bitrate levels). Bitrate levels are associated with the Lagrangian parameters of the compression loss function as in Equation 3.28. Consequently, we set $L = 4$ for $\{\lambda_0 = 0.005,\ \lambda_1 = 0.001,\ \lambda_2 = 0.0005,\ \lambda_3 = 0.0001\}$ and associate each with a different row of the learned scaling matrix $\mathbf{S}$, $\{\mathbf{s}_0,\ \mathbf{s}_1,\ \mathbf{s}_2,\ \mathbf{s}_3\}$. The learned scale vectors must be positive to have a meaningful effect on the importance and quantization steps of the features. For that reason, the scaling operation is performed by multiplying with the absolute value of the scaling vector, $|\mathbf{s}_i|$.
An additional benefit of having multi-rate adaptation through training for multiple bitrates is that we can achieve intermediate bitrate levels through exponential interpolation as well. Specifically, subsequent scaling vectors can be exponentially interpolated in order to achieve intermediate levels. For example, for a bitrate level decomposed as $r = \text{int}(l) \in [0,\ L-1]\,,\ t = l - \text{int}(l) \in [0,\ 1)$, we can perform an exponential interpolation as $\mathbf{s}_{interp} = \mathbf{s}_r^{(1-l)} \cdot s_{r+1}^{(l)}$.
In addition, the multi-rate adaptation with multiple scaling vectors for latent representation is applied for hyperprior network in hierarchical entropy modeling described in Section 3.4.2 as well. As a result, instead of training single scaling vector for each training corresponding to single bitrate, we train $L = 4$ separate scaling vectors in single training which correspond to multiple bitrates covered by multiple Lagrange multipliers, $\{\lambda_0 = 0.005,\ \lambda_1 = 0.001,\ \lambda_2 = 0.0005,\ \lambda_3 = 0.0001\}$.

3.4.4 Disjoint Position Compression

In order to compress the position information for Gaussian primitives, we evaluated the DEFLATE [9] and G-PCC / TMC13 [22] algorithms. Due to the position-sensitive nature of Gaussian primitives, we avoid compressing the Gaussian positions with learned entropy models. For our tests with the DEFLATE algorithm, we reduced the precision of Gaussian primitive positions from a single precision floating point (*float32*) to a half-precision (*float16*) and applied the DEFLATE algorithm. Specifically, the DEFLATE algorithm replaces repeated occurrences with references to previous copies in the uncompressed data stream and applies Huffman coding for entropy coding. For that reason, the position compression with DEFLATE algorithm is not a learned process but a lossless compression process after reducing precision).
Secondly, we experiment with the G-PCC / TMC13 [22] algorithm for position compression. Although the method is proposed for point-cloud compression, it can be utilized for Gaussian primitive positions analogously. For TMC13, the point cloud is initially voxelized and quantized for octree encoding. Later, the octree structure is serialized to efficiently encode the positions of primitives. Predictive and residual coding methods are applied to exploit the spatial redundancy and the octree data with residuals are further compressed using entropy coding. To utilize TMC13, we utilize the out-of-the-box repository provided by MPEG.

3.5 Hierarchy Generation

Although our main objective is compressing the Gaussian splatting representation, the compression of Gaussian primitives can benefit from correlation among them. For that reason, we experiment on building a tree hierarchy on Gaussian primitives. Specifically, we are inspired from [16] and modify their algorithm to build

an octree structure where the scene is covered with a 3D bounding box and the bounding box is segmented into 8 regions at every depth of the octree, as visualized in Figure 3.6. By compressing the node attributes of the octree that is built, we use the information from compressed node attributes to perform residual coding on Gaussian attributes.
For that purpose, the proposed algorithm can be defined in two steps: First of all, the octree is built using the already calculated Gaussian primitives based on the position of the primitives. Later, the attributes of Gaussian primitives lying in the same leaf node are aggregated, and the aggregation is continued upward in the tree structure so that the aggregation continues until the desired depth level starting from leaves to root. For the final part, the aggregated node attributes are utilized for calculating residual attributes for each Gaussian primitive, and the residual information is encoded and decoded separately.

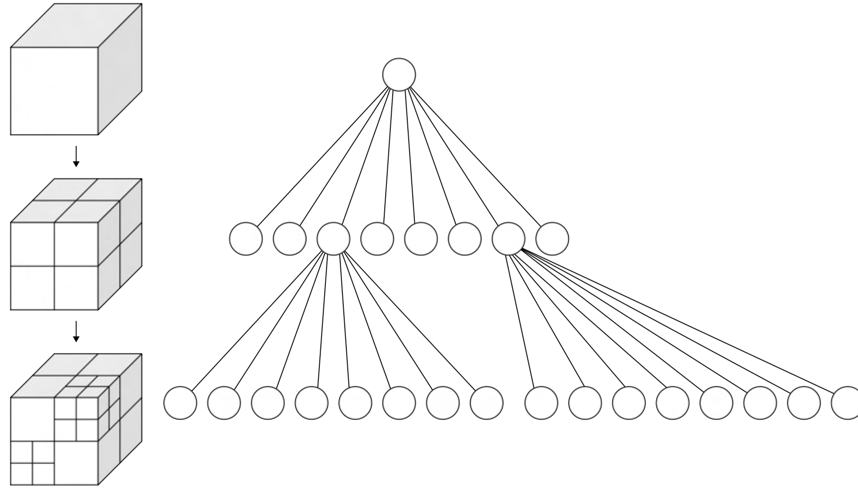

Figure 3.6: **A high-level visualization of octree structure [7].** An octree is created by dividing the 3D space into 8 pieces for each bounding box that is non-empty. As a result, a tree structure is acquired where close Gaussians are children of the same node.

3.5.1 3D Gaussian Octree Building

In order to build an octree hierarchy over Gaussian primitives, we set an initial axis-aligned bounding box (AABB) over Gaussian primitives. The bounding box size is extended by $1.01\times$ along all axis-aligned dimensions and covers all primitives. Later on, each Gaussian primitive is assigned to one of approximately 8^d nodes at depth level d until d_{max}. For our case, we set d_{max} adaptively to avoid over-merging Gaussian primitives since a lower maximum depth causes many Gaussian primitives to collapse into single node. The adaptive calculation for the depth level is calculated as,

$$d_{max} = \left\lceil \frac{\max(l_x,\ l_y\ l_z)}{0.01} \right\rceil \tag{3.40}$$

$$d = d_{max} - 5 \tag{3.41}$$

where l_x, l_y, l_z are the axis-aligned length of AABB over Gaussian primitives and d is the chosen node level to provide predictions for the Gaussian primitives. As each scene representation has between $500K$ and $5M$ Gaussians. The octree building has to be highly parallelized in order to be fast. For that reason, the code for building an octree is prepared in C and CUDA programming languages to allow a low-level implementation. The algorithm for building the octree structure can be viewed in Algorithm 1. As a result

of the process involving octree building, each Gaussian primitive is assigned to a set of intermediate octree nodes at all depths between d and d_{max} are stored for residual coding.

Algorithm 1 Octree Building Kernel. The octree building kernel is utilized in a parallel manner on all Gaussian primitives in order to assign them to the nodes that they belong to and store it in a matrix where rows are for each Gaussian primitive and columns are for each depth level. A point, corners of initial axis-aligned bounding box (AABB), and maximum octree depth are provided to the kernel.

```
1:  procedure RECURSIVE_OCTREE_KERNEL(point, aabb_min, aabb_max, max_depth)
2:      Initialize current_min, current_max ← aabb_min, aabb_max
3:      Init node_idx ← 0
4:      for depth ← 0 to max_depth − 1 do                ▷ Iterate until desired depth level
5:          mid ← (current_min + current_max) / 2        ▷ Middle point of the axis-aligned bounding box
6:          octant ← 0
7:          for i ← 0 to 2 do
8:              if point[i] ≥ mid[i] then
9:                  octant ← octant | (1 ≪ i)            ▷ Shift octant according to point position and mid
10:                 current_min[i] ← mid[i]              ▷ Subdivide the AABB into new pieces for current point
11:             else
12:                 current_max[i] ← mid[i]
13:             end if
14:         end for
15:         node_idx ← node_idx × 8 + octant             ▷ Assign a node id to the intermediate node
16:     end for
17: end procedure
```

3.5.2 3D Gaussian Primitive Aggregation

After forming the octree by utilizing the Algorithm 1, the node attributes need to be propagated from actual Gaussian primitives to octree nodes by weighted averaging. The implementation of this process is again inspired from [16]. In order to aggregate the Gaussian primitive attributes into node attributes, we first calculate a scalar weight per Gaussian primitive which are essentially based on the opacity of primitive and the surface area of the Gaussian,

$$w'_i = o_i \times \left(\sqrt[3]{|\Sigma_i|}\right), \qquad w_i = \frac{w'_i}{\sum_{j=1}^{N} w'_j} \tag{3.42}$$

where $\sqrt[3]{\Sigma_i}$ calculates the characteristic length of the ellipsoid to approximate the surface area upto a factor and scaled later on by the opacity of Gaussian. We use the calculated weights to apply weighted averaging on positions, covariances, spherical harmonics and opacities of Gaussian primitives that are children of same

leaf node of the octree structure,

$$\mu^{(l+1)} = \sum_i w_i \mu_i^{(l)} \tag{3.43}$$

$$\Sigma^{(l+1)} = \sum_i w_i \left(\Sigma_i^{(l)} + (\mu_i^{(l)} - \mu^{(l+1)})(\mu_i^{(l)} - \mu^{(l+1)})^\top\right) \tag{3.44}$$

$$o^{(l+1)} = \sum_i w_i o_i^{(l)} \tag{3.45}$$

$$\mathrm{SH}^{(l+1)} = \sum_i w_i \mathrm{SH}_i^{(l)} \tag{3.46}$$

The process of aggregation is continued until reaching the desired depth level of the octree structure that will be used for predictions before calculating the residual attributes of Gaussian primitives. At each intermediate octree depth level, we utilize the attributes of children nodes to apply weighted averaging instead of the Gaussian primitives that lie in the target node.

3.5.3 Residual Coding Using Lower Hierarchy Levels As Predictions

After computing the intermediate node attributes for the desired depth level, the respective node attributes for each Gaussian are used as predictions. As predictions tend to deviate from actual attributes, the difference between the actual Gaussian attributes and the predicted attributes (node attributes) are compressed in addition. The difference $\mathbf{\Delta} = \mathbf{x} - \mathbf{n}$, where $\mathbf{x}$ stands for Gaussian primitive attributes, $\mathbf{n}$ stands for node attributes at depth level l as calculated in Equation 3.40, has the same dimensionality since $\mathbf{\Delta},\ \mathbf{x},\ \mathbf{n} \in \mathbb{R}^{59}$ as each Gaussian primitive has 59 scalar attributes for position, scaling, rotation, opacity, and spherical harmonics.

Recall that the aggregation of Gaussian covariance information was performed directly over the calculated covariance and not the scaling and rotation components. To compress the predicted Gaussian covariance matrices, we experiment with both compressing directly the covariance matrix or scaling and rotation parameters. For the case of scaling/rotation compression, we decompose the covariance matrix into scaling and rotation matrices after calculating the aggregated node covariance using Equation 3.44. The decomposition is performed utilizing eigen-decomposition,

$$\Sigma = V\Lambda V^{-1}, \qquad S = \Lambda^{1/2}, \qquad R = V \tag{3.47}$$

After decomposing the predicted Gaussian covariance matrix, the rotation and scaling components are compressed. For decompressed rotation R, we ensure that eigenvectors are normalized, and for decompressed scaling S, we ensure that eigenvalues are positive by clamping to a minimum value $1e-6$. For the case of covariance compression, the node covariance is factorized using Cholesky factorization to ensure a positive-semidefinite matrix after compression,

$$\Sigma = LL^T \tag{3.48}$$

The lower half of lower triangular matrix is compressed/decompressed and the covariance is reconstructed using lower triangular matrix again. For both cases, the attribute differences (residuals) are computed by directly subtracting the attributes. Specifically, we calculate the residuals for Gaussian primitive position, covariance (or scaling/rotation), opacity, and spherical harmonics by subtraction. The residuals are encoded independently using the same compression model architecture as described in Section 3.4.2. Finally, we make sure that the reconstructed covariance matrix is positive semidefinite by clamping the eigenvalues to be positive when compressing directly the covariance matrix. As the assignment of Gaussian primitives to intermediate nodes is necessary information to use them as predictions, the node assignments are

compressed separately for decoding at a low cost. A high-level overview of the proposed residual coding algorithm for Gaussian primitives can be seen in Figure 3.7.

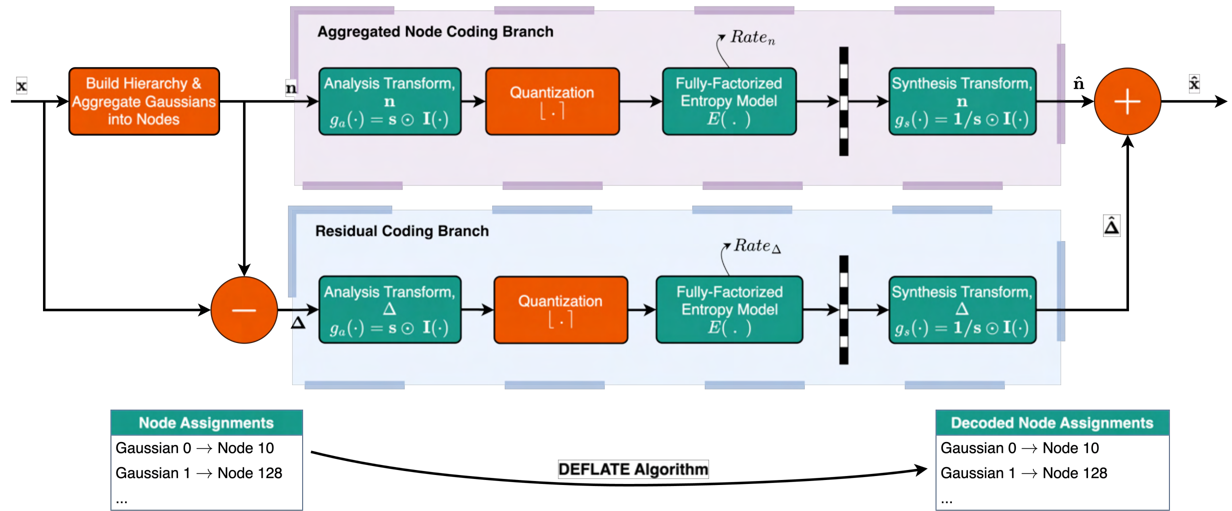

Figure 3.7: **An overview of the residual coding framework for compressing Gaussian primitives.** The green modules depict the learned components such as MLP layers or quantization step sizes like $\mathbf{s}$. Note that the quantization scale $\mathbf{s}$ is shared for analysis and synthesis transforms within each branch. The orange modules are the fixed operations such as quantization and entropy coding. For the entropy model components, mean-scale hyperprior can be utilized instead of fully-factorized entropy model.

3.6 Resources and Framework of Implementation

For the implementations and experimentation during the preparation of this report, the deep learning framework PyTorch [29] is utilized. In addition, compression methods are used with adaptations and templates provided in the CompressAI deep image / video compression library [5]. In addition, custom operations that are suitable for parallelization such as octree generation and attribute aggregation described in Section 3.5 are implemented in C and CUDA programming languages for efficiency.

Throughout the project, Biomedical Image Computing (BMIC) computation resources and compute clusters are utilized. Specifically, all entropy model trainings and Gaussian primitive optimizations are performed on one NVIDIA RTX A6000 GPU for parallelized computing.

Chapter 4

Experiments and Results

After utilizing methods described in Chapter 3, results in current chapter are acquired for utilization of Gaussian splatting, pruning, and compression methods. In section 4.2, a close inspection of 3D-MCMC [17] against seminal 3D-GS [15] will be presented. Later, the impact of pruning and the comparison of learned masking [20] with RadSplat pruning [27] will be depicted in Section 4.3. Finally, fully-factorized [1] and mean-scale hyperprior [25] entropy models will be evaluated and analysis of residual coding for Gaussian primitives will be presented in Section 4.4 and 4.5, respectively.

4.1 Metrics and Datasets

Metrics. For the experiments performed throughout this chapter, the general consensus in Gaussian Splatting literature is followed. For metrics, the methods are evaluated using Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index Metric (SSIM), and Learned Perceptual Image Patch Similarity (LPIPS) [40]. In addition, the size of Gaussian splatting representation is calculated in mega-bytes (MB) and number of Gaussians are reported for Gaussian primitive optimization. As the Gaussian primitives are no longer densified and pruned during compression stage, the number of Gaussians is omitted in tables after Section 4.3 for the sake of conciseness.
Datasets. The evaluation is performed on well-known and public scenes from Tanks & Temples [19], Deep Blending [12], and Mip-NeRF 360 [2] datasets. For all scene related parameters such as scenes, frame IDs, and image resolution, we follow the same parameter choices specified in seminal 3D-GS method [15].

4.2 3D Gaussian Splatting Optimization Improvement

In order to achieve a superior rate-distortion performance, the optimization of actual Gaussians in a 3D-GS framework is essential. For that reason, we utilize an improved version of 3D-GS in order to improve visual quality without increasing the number of Gaussian primitives, namely 3D-MCMC [17]. In contrast to other works such as HAC [6] and Context-GS [35], our use of 3D-MCMC conforms with the highly adopted 3D-GS framework instead of Scaffold-GS. In order to train both 3D-GS and 3D-MCMC, we use the hyperparameters described in Sections 3.1 and 3.2.
As 3D-MCMC framework for 3D-GS also allows for control on number of Gaussian primitives, a fair comparison with similar number of Gaussian primitives depicts the visual superiority of 3D-MCMC quantitatively in Table 4.1. To achieve similar number of Gaussian primitives, we first optimize the 3D-GS [15] Gaussian primitives and then set the N_{max} close to the number of final Gaussian primitives optimized with 3D-GS. The parameters that are set for each scene can be found in Tables A.1, A.2, and A.3.

Note that the change in the optimization framework through 3D-MCMC does not cause any change in the type of parameters of Gaussian primitives. In addition, as both models have similar number of Gaussians with same parameters, the memory requirement are the same for both models.

	Tanks&Temples		DeepBlending		Mip-NeRF 360	
Method	# Gaussians	PSNR / SSIM / LPIPS	# Gaussians	PSNR / SSIM / LPIPS	# Gaussians	PSNR / SSIM / LPIPS
3D-GS [15]	1,834,587	23.68 / 0.848 / 0.176	2,818,819	29.57 / 0.904 / 0.244	3,322,968	27.47 / 0.815 / 0.216
Scaffold-GS [21]	not applicable	23.96 / 0.853 / 0.177	not applicable	**30.21 / 0.906 / 0.254**	not applicable	27.50 / 0.806 / 0.252
3D-MCMC [17]	1,750,000	**24.65 / 0.868 / 0.154**	2,750,000	29.48 / 0.908 / 0.252	3,277,778	**27.97 / 0.834 / 0.191**

Table 4.1: **Comparison of 3D-GS [15] against 3D-MCMC [17].** We evaluate both 3D-GS and 3D-MCMC on Tanks & Temples [19], DeepBlending [12], and Mip-NeRF 360 [2] scenes in order to devise a base Gaussian optimization method. As comparison reveals superiority of 3D-MCMC with negligible overhead during training, we continue with 3D-MCMC in order to compress 3D Gaussian primitives. For evaluation of both methods, the public repositories of 3D-GS and 3D-MCMC are utilized without change in functionality to reproduce the results. For Scaffold-GS, the experiment results are taken as reported [21]. Note that our evaluation involves *treehill* and *flowers* in contrast to reported results in 3D-GS [15] and 3D-MCMC [17].

According to our experiments on Tanks & Temples [19], DeepBlending [12], and Mip-NeRF 360 [2], we confirmed that 3D-MCMC generally appears to be superior compared to 3D-GS. Although superiority is evident in Tanks & Temples [19] and Mip-NeRF 360 [2], 3D-GS performs slightly better in DeepBlending [12]. This result is also confirmed by the results reported by 3D-MCMC [17]. Furthermore, noting samples from Figure 4.1, the noise term and exploration with 3D-MCMC clearly help in terms of improving quality at convoluted regions, while 3D-GS appears blurry in most difficult regions when initialization is not particularly helpful. Some examples of such regions on scenes are from mountain parts of the *train* scene and flowers in the *flowers* scene. As a result of the quantitative comparison based on Table 4.1 and qualitative comparisons on Figure 4.1, our primary choice is to continue taking 3D-MCMC [17] as a base for the rest of compression evaluations.

4.3 3D Gaussian Splatting Primitive Pruning

The second stage towards compressing 3D Gaussian primitives is pruning the primitives so that redundancy in number of parameters is reduced with fewer primitives. This stage is essential as 3D-GS is known to be highly redundant in optimization process even when modified with 3D-MCMC algorithm. Thus, an effective yet efficient pruning method is adopted by comparing learnable masking of 3D Gaussian primitives [20] against the importance based pruning method proposed with RadSplat [27] in Table 4.2.

For the training and evaluation of Gaussian primitives with learned masking and RadSplat pruning methods, we train the Gaussian primitives from scratch for $30K$ iterations, same as the case without these additional pruning methods. All other training hyperparameters are kept same as in 3D-MCMC [17] for fair evaluation. In addition, we apply RadSplat pruning at iterations $16K$, and $24K$ same as proposed in RadSplat [27] with a threshold of 0.01. For the learned masking, we apply the masking on 3D-MCMC [17] with a masking threshold of 0.01 and a learning rate of 0.001 for the learned mask parameters. Furthermore, we prune the Gaussian primitives once in every $1,000$ iterations and continue pruning primitives even after $15,000$ iterations which is different from 3D-GS and 3D-MCMC without learned masking.

To compare both methods and see their similarity, we first compare the density of Gaussian primitives in important regions of *train* scene from Tanks & Temples [19] in Figure 4.2. To understand whether learned

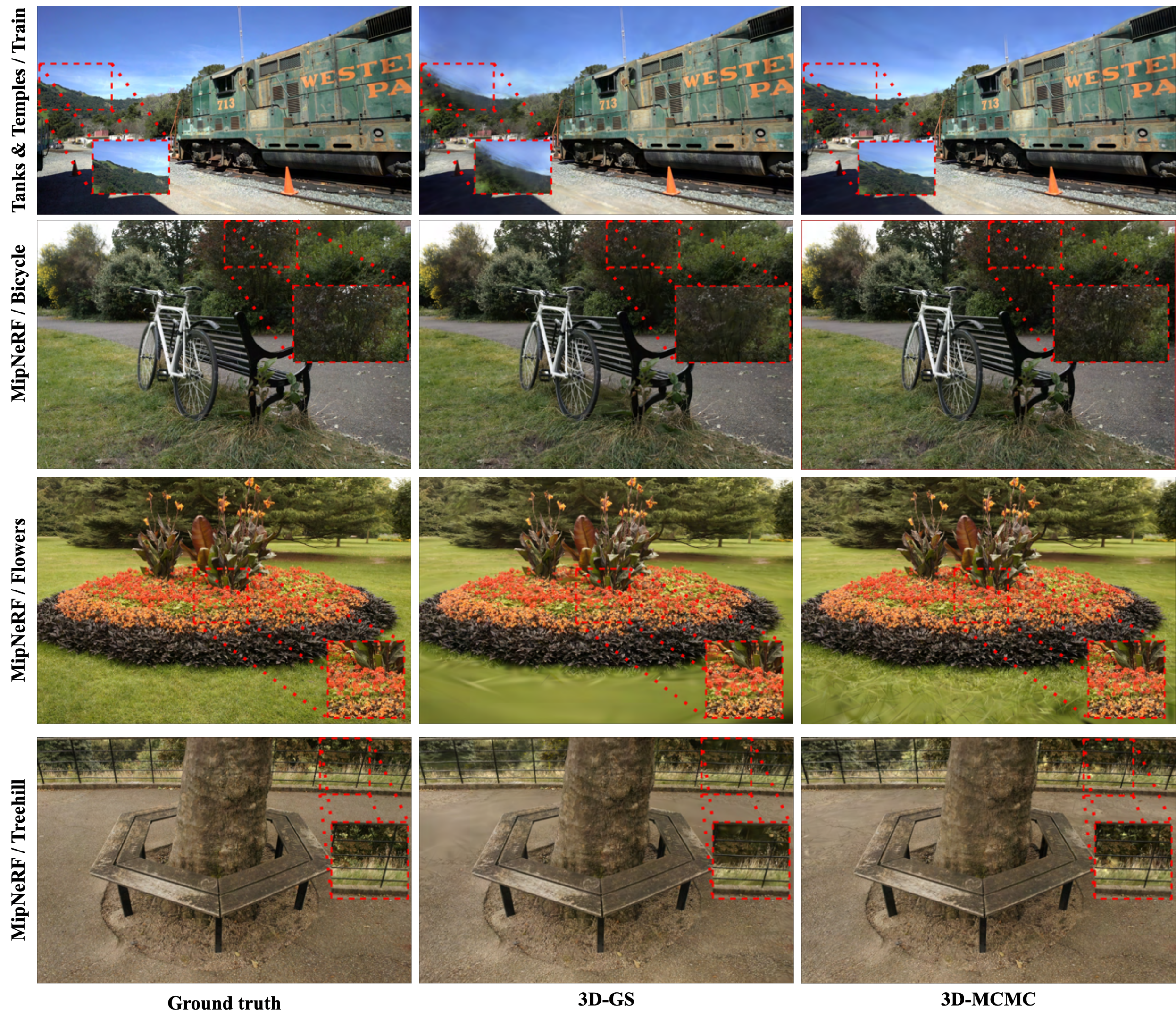

Figure 4.1: **Qualitative comparison of 3D-GS [15] against 3D-MCMC [17].** The qualitative results on *train*, *bicycle*, *flowers*, and *treehill* scenes. As per seen from crops on rasterized images, 3D-MCMC yields superior visual quality over 3D-GS.

masking and RadSplat pruning pay importance to similar regions, we plot a histogram for the count of Gaussian primitive positions falling into the same 3D bins. For that purpose, we look at most important where majority of primitives reside. As can be seen in Figure 4.2, both methods have very similar frequency histograms for the number of Gaussian primitives falling into same bins. For that reason, we can deduce that although RadSplat pruning has external guidance through maximum image contribution formulation, learned masking also learns a similar feature (scalar) which denotes the importance of a Gaussian for all training images.

Based on the results on Table 4.2 and more detailed view of experiments in Appendix A, application of RadSplat pruning method appears slighly better compared to applying learned masking in overall. Although the results are similar in terms of PSNR, SSIM, and LPIPS metrics, RadSplat pruning method is relatively simpler as it needs to be applied only twice during training. In addition, RadSplat pruning is agnostic to number of views since it requires taking $\max$ of primitive contribution over all views and it allows for use

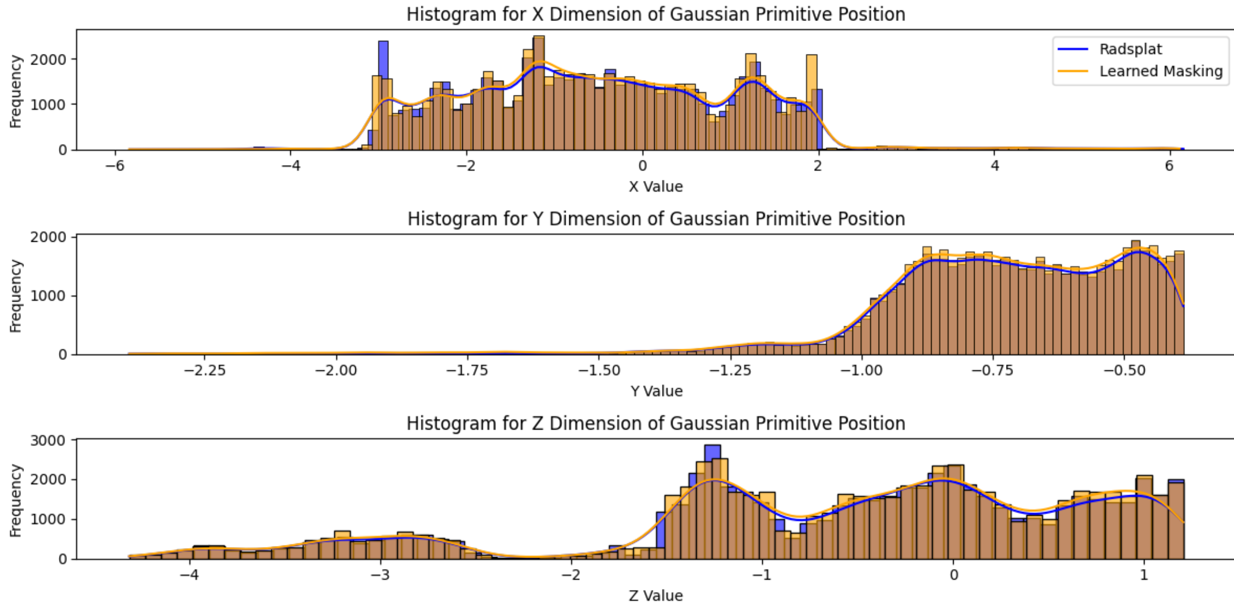

Figure 4.2: **Gaussian primitive position histogram comparison of learned masking and RadSplat pruning on *train* scene from Tanks & Temples [19] dataset.** We draw histograms of Gaussian primitive positions along each dimension. Specifically, we take the $0.2 \times \sigma$ range for each dimension around the mean position to reduce the sample size for histogram. We observe that both RadSplat pruning [27] and learned masking [20] achieve similar distributions for Gaussian primitives.

	Tanks&Temples		DeepBlending		Mip-NeRF 360	
Method	# Gaussians	PSNR / SSIM / LPIPS	# Gaussians	PSNR / SSIM / LPIPS	# Gaussians	PSNR / SSIM / LPIPS
3D-MCMC	1,750,000	24.65 / 0.868 / 0.154	2,750,000	29.48 / 0.908 / 0.252	3,277,778	27.97 / 0.834 / 0.191
with Masking	856,210	24.60 / 0.868 / 0.154	624,490	29.26 / 0.906 / 0.254	2,170,155	27.98 / 0.835 / 0.192
with Pruning	788,980	24.61 / 0.867 / 0.157	505,365	29.33 / 0.905 / 0.256	2,058,623	27.99 / 0.835 / 0.192

Table 4.2: **Comparison of learned masking against RadSplat pruning method on 3D-MCMC.** We evaluate two different pruning methods, namely learned masking proposed in [20] and importance-aware pruning method proposed in [27]. Pruning is an important step of Gaussian splatting compression since the redundancy in number of Gaussians should be first reduced through lowering number of Gaussians. The comparison reveals slightly better results with pruning method proposed with RadSplat in given two scenes.

of relatively fewer Gaussians for all scenes. Therefore, based on general consensus and reduced number of Gaussians with RadSplat pruning method, our choice for further analysis and compression of Gaussian Splatting will utilize RadSplat pruning for reducing the redundancy in scene representation with Gaussian Splatting.
In addition, note that a lower number of Gaussian primitives is also beneficial in terms of rasterization speed. For a simple yet important comparison, the regular 3D-MCMC model with $3,700,000$ Gaussian primitives is capable of rasterizing 85 frames per second while its counterpart with 103 FPS after RadSplat pruning method is applied to reduce the number of Gaussian primitives to $2,426,631$. The reported values are acquired when rasterizing on single NVIDIA RTX A6000.

4.4 Gaussian Primitive Compression using Learned Entropy Models

For compression of Gaussian primitives, our trials mainly focused on compressing attributes of Gaussian primitives without complex analysis and synthesis transforms which are described in Section 3.4. For the rest of this chapter, the experiments with fully-factorized entropy model, mean-scale hyperprior entropy model, and hierarchical structure will be presented. In addition, the application of multi-rate compression models for Gaussian primitives will be presented.
For our experiments in this section, we refer to hierarchical entropy model as "mean-scale hyperprior" model as it includes a hyperprior network to estimate the entropy parameters, mean and standard deviation (scale). As for our entropy model learning rates, we always use a learning rate of 0.01 both for the entropy model parameters, and also for the auxiliary parameters of fully-factorized model to calculate quantiles. For entropy model training, we tested three different cases:

1. Training and comparing fully-factorized and mean-scale hyperprior (hierarchical) entropy models,
2. Training the entropy model for $5K$ iterations after optimizing Gaussian primitives $25K$ iterations against training the entropy model for $10K$ iterations after optimizing Gaussian primitives $30K$ iterations,
3. Training the entropy model with and without optimizing geometry (position, scaling, rotation),
4. Training multi-rate compression model against training separate models for different bitrates.

4.4.1 Fully-Factorized and Mean-Scale Hyperprior Entropy Models

In this subsection, we compare the fully-factorized entropy model with mean-scale hyperprior to compare in terms of rate-distortion optimization. For the experiments, we optimize Gaussian primitives for $30K$ iterations with 3D-MCMC [17] algorithm and then introduce entropy models in order to compress the Gaussian primitives. Furthermore, we keep optimizing the geometry related attributes of Gaussians, i.e., the position, scaling, and rotation attributes for compressed Gaussian primitives. The results acquired for our tests can be visualized in Table 4.3 and rate-distortion curves displayed in Figure 4.3.

	Mip-NeRF360 [3]				**Tanks&Temples [16]**				**DeepBlending [14]**			
	PSNR↑	**SSIM↑**	**LPIPS↓**	**Size↓**	**PSNR**	**SSIM↑**	**LPIPS↓**	**Size↓**	**PSNR↑**	**SSIM↑**	**LPIPS↓**	**Size↓**
Fully-Factorized ($\lambda = 0.005$)	25.46	0.778	0.251	25.16	23.02	0.822	0.204	9.27	26.67	0.875	0.304	5.13
Mean-Scale Hyperprior ($\lambda = 0.005$)	25.61	0.781	0.248	25.16	23.09	0.824	0.203	9.21	26.82	0.876	0.303	5.13
Fully-Factorized ($\lambda = 0.001$)	26.84	0.809	0.219	39.97	24.06	0.848	0.178	13.99	27.73	0.888	0.285	7.72
Mean-Scale Hyperprior ($\lambda = 0.001$)	26.95	0.810	0.218	38.82	24.08	0.849	0.176	13.79	27.76	0.889	0.283	7.58
Fully-Factorized ($\lambda = 0.0001$)	27.53	0.823	0.201	63.71	24.33	0.860	0.164	23.11	28.15	0.893	0.276	13.13
Mean-Scale Hyperprior ($\lambda = 0.0001$)	27.54	0.823	0.201	61.27	24.35	0.860	0.164	22.02	28.17	0.893	0.275	12.57

Table 4.3: **The quantitative compression results obtained from the proposed entropy models when trained for 10K iterations after 30K iterations of optimization for Gaussian primitives.** We compare the results with fully-factorized and mean-scale hyperprior entropy models to deduce which appears superior. As a result, mean-scale hyperprior has slightly better compression ratios with very similar performance.

Note that the size of a scene in 4.3 includes three parts: 1) Compressed Gaussian primitive parameters except position, 2) Entropy model parameters, 3) Gaussian primitive position parameters. We compress the Gaussian primitive position separately because of the high sensitivity of Gaussian primitives to noise. This is a recurring theme observed in other models [20, 10, 26] as well. Based on our comparison between

fully-factorized and mean-scale hyperprior entropy models, it is evident that mean-scale hyperprior achieves superior compression performance by a small margin against fully-factorized entropy model. In almost all scenes of provided datasets, mean-scale hyperprior requires smaller memory while acquiring a slightly better visual quality. This result is sensible with the main motivation for the derivation of mean-scale hyperprior entropy model: As the entropy model allows for providing additional information regarding the distribution of Gaussian primitive attributes, it achieves a more efficient entropy coding, resulting in a higher efficiency in terms of bitrate and visual quality.

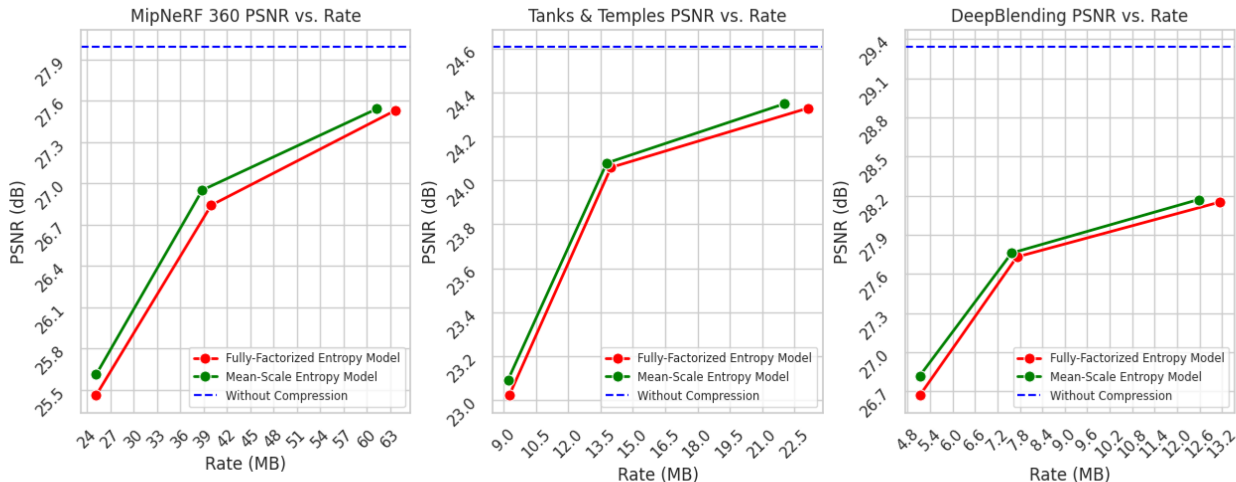

Figure 4.3: **Rate-Distortion Curve for fully-factorized and Meanscale-Hyperprior entropy models.** For both fully-factorized and mean-scale hyperprior entropy models, we first optimize the Gaussian primitives for $30K$ iterations and then train the respective entropy model for $10K$ iterations. During the training, we continue optimizing all Gaussian attributes.

For that reason, we continue our experiments only with mean-scale hyperprior entropy model rather than fully-factorized entropy model.

4.4.2 Entropy Penalization Start Iteration and Training Duration

Secondly, we inspect the effect of training duration and reducing the computational complexity of training. For generalizable compression networks in image/video compression literature, a longer training duration on multiple images/videos is required. However, since we are overfitting a compression model on a single scene, training the entropy model for a shorter duration might suffice. For that reason, we repeat our experiments to train the entropy model for $5K$ iterations after optimizing the Gaussian primitives for $25K$ iterations using the 3D-MCMC algorithm.

Repeating the experiments for training $5K$ iterations after optimizing Gaussian primitives for $25K$ iterations, we observe that the change in quantitative results is not vast and a similar visual quality is achieved even with $5K$ iterations instead of $10K$ iterations. Since we are in an overfitting setting, this result is understandable. Having a single scene per training is less demanding in terms of optimization compared to generalization setting.

Thus, after investigating Table 4.4, we opt for training entropy model for only $5K$ iterations after optimizing the Gaussian primitives for $25K$ iterations over the training entropy model for $10K$ iterations after optimizing the Gaussian primitives for $30K$ iterations. The reason behind our choice is the reduced training

		Mip-NeRF360 [3]				Tanks&Temples [16]				DeepBlending [14]			
		PSNR↑	SSIM↑	LPIPS	Size↓	PSNR	SSIM↑	LPIPS↓	Size↓	PSNR↑	SSIM↑	LPIPS↓	Size↓
$5K$ after $25K$	**Fully-Factorized** ($\lambda = 0.005$)	25.40	0.778	0.251	24.88	22.91	0.822	0.206	9.22	26.62	0.876	0.302	5.17
	Mean-Scale Hyperprior ($\lambda = 0.005$)	25.54	0.781	0.248	24.12	23.00	0.824	0.204	9.09	26.67	0.876	0.303	5.17
	Fully-Factorized ($\lambda = 0.001$)	26.67	0.809	0.219	38.60	23.96	0.848	0.179	13.71	27.73	0.890	0.281	7.50
	Mean-Scale Hyperprior ($\lambda = 0.001$)	26.89	0.810	0.218	37.38	24.01	0.848	0.178	13.36	27.78	0.891	0.281	7.28
	Fully-Factorized ($\lambda = 0.0001$)	27.50	0.823	0.202	58.75	24.32	0.859	0.166	22.86	28.31	0.895	0.272	12.70
	Mean-Scale Hyperprior ($\lambda = 0.0001$)	27.50	0.823	0.202	54.15	24.32	0.859	0.166	20.80	28.30	0.896	0.272	11.69
$10K$ after $30K$	**Fully-Factorized** ($\lambda = 0.005$)	25.46	0.778	0.251	25.16	23.02	0.822	0.204	9.27	26.67	0.875	0.304	5.13
	Mean-Scale Hyperprior ($\lambda = 0.005$)	25.61	0.781	0.248	25.16	23.09	0.824	0.203	9.21	26.82	0.876	0.303	5.13
	Fully-Factorized ($\lambda = 0.001$)	26.84	0.809	0.219	39.97	24.06	0.848	0.178	13.99	27.73	0.888	0.285	7.72
	Mean-Scale Hyperprior ($\lambda = 0.001$)	26.95	0.810	0.218	38.82	24.08	0.849	0.176	13.79	27.76	0.889	0.283	7.58
	Fully-Factorized ($\lambda = 0.0001$)	27.53	0.823	0.201	63.71	24.33	0.860	0.164	23.11	28.15	0.893	0.276	13.13
	Mean-Scale Hyperprior ($\lambda = 0.0001$)	27.54	0.823	0.201	61.27	24.35	0.860	0.164	22.02	28.17	0.893	0.275	12.57

Table 4.4: **The quantitative compression results obtained from the proposed entropy models when trained for 5K iterations against $10K$ iterations with varying Gaussian primitive optimization durations.** As training for **10K** iterations is computationally demanding compared to training for an overall $30K$ iterations, we test having a total training time of $30K$ iterations where last $5K$ iterations are utilized for entropy model training. As a result, the improvement with training for an additional $10K$ iterations remains negligible.

complexity while achieving almost same results without any degradation due to reduced number of training iterations.

4.4.3 Freezing Geometry Parameters

During the training of entropy models, one can have two different approaches: 1) Continuing optimizing Gaussian primitive geometry attributes, 2) Only training the entropy model and appearance attributes (i.e., opacity and spherical harmonics). This choice is born from the fact that the Gaussian primitives are more sensitive to deviations in geometry attributes such as position, scaling, and rotation when compared with appearance attributes such as opacity, and spherical harmonics. To test this hypothesis, we repeat our experiments with mean-scale hyperprior entropy model with and without optimizing the geometry attributes.

		Mip-NeRF360 [3]				Tanks&Temples [16]				DeepBlending [14]			
		PSNR↑	SSIM↑	LPIPS	Size↓	PSNR	SSIM↑	LPIPS↓	Size↓	PSNR↑	SSIM↑	LPIPS↓	Size↓
Optimize geometry	**Mean-Scale Hyperprior** ($\lambda = 0.005$)	25.54	0.781	0.248	24.12	23.00	0.824	0.204	9.09	26.67	0.876	0.303	5.17
	Mean-Scale Hyperprior ($\lambda = 0.001$)	26.89	0.810	0.218	37.38	24.01	0.848	0.178	13.36	27.78	0.891	0.281	7.28
	Mean-Scale Hyperprior ($\lambda = 0.0001$)	27.50	0.823	0.202	54.15	24.32	0.859	0.166	20.80	28.30	0.896	0.272	11.69
Frozen geometry	**Mean-Scale Hyperprior** ($\lambda = 0.005$)	25.73	0.795	0.234	25.83	23.35	0.834	0.194	9.71	27.23	0.885	0.287	5.80
	Mean-Scale Hyperprior ($\lambda = 0.001$)	27.28	0.824	0.206	38.45	24.38	0.858	0.169	13.74	28.38	0.900	0.267	7.69
	Mean-Scale Hyperprior ($\lambda = 0.0001$)	28.01	0.834	0.193	59.69	24.73	0.866	0.159	21.35	28.82	0.903	0.260	11.37

Table 4.5: **The quantitative compression results obtained from mean-scale hyperprior entropy model when optimization for geometry attributes continues against the case when they are fixed.** Mean-scale hyperprior entropy model is evaluated when it is trained for $5K$ iterations after optimizing the Gaussian primitives for $25K$ iterations. We compare the cases where geometry attributes are further optimized against the case when they are no longer optimized after $25K$ iterations.

As a result of our experiments, we gathered quantitative results for scene sizes and visual qualities in Table 4.5 and Figure 4.4 for the rate-distortion performance. Specifically observing Figure 4.4, it becomes evident

that rate-distortion loss hurts the visual quality significantly compared to jointly optimizing the geometry attributes. For that reason, our choice for training the entropy model is to train it without optimizing the geometry attributes.

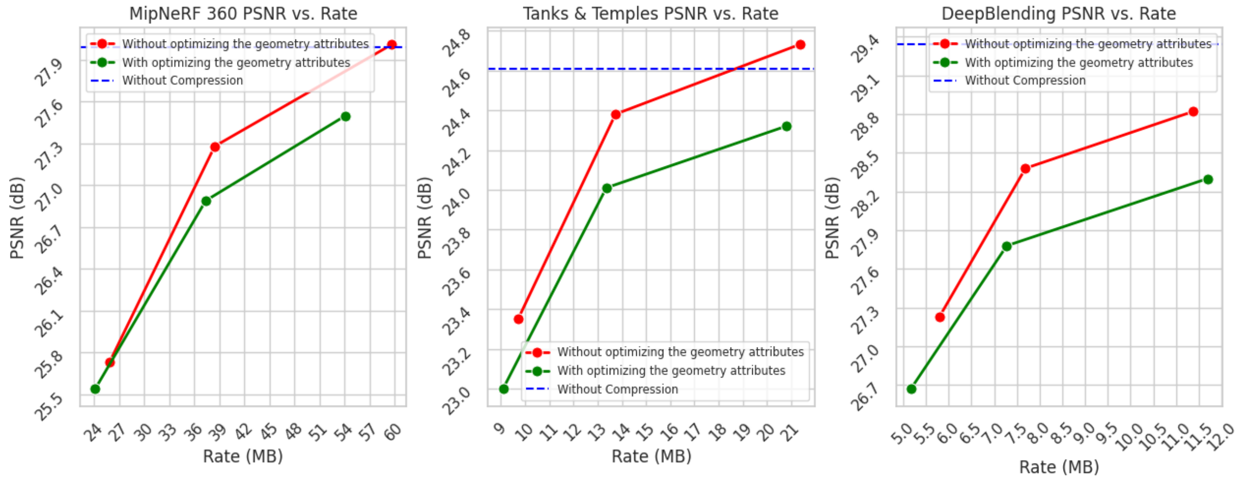

Figure 4.4: **Rate-distortion comparison of training mean-scale hyperprior entropy model with and without optimizing the Gaussian geometry attributes.** We train the entropy model for $5K$ iterations after optimizing Gaussian primiitves for $25K$ iterations. As a result, we deduce that Gaussian geometry attributes are more prone to noise and should not be optimized.

At this point, a comparison with other works [10, 20, 26, 6, 35] is beneficial to observe where our compression approach falls in Table 4.6 and Figure 4.5.

	Mip-NeRF360 [3]				Tanks&Temples [16]				DeepBlending [14]			
	PSNR↑	SSIM↑	LPIPS	Size↓	PSNR	SSIM↑	LPIPS↓	Size↓	PSNR↑	SSIM↑	LPIPS↓	Size↓
3DGS [15] (SIGGRAPH'23)	27.49	0.813	0.222	744.7	23.69	0.844	0.178	431.0	29.42	0.899	0.247	663.9
Scaffold-GS [21] (CVPR'24)	27.50	0.806	0.252	253.9	23.96	0.853	0.177	86.50	30.21	0.906	0.254	66.00
LightGaussian [10]	27.00	0.799	0.249	44.54	22.83	0.822	0.242	22.43	27.01	0.872	0.308	33.94
Compact3DGS [20] (CVPR'24)	27.08	0.798	0.247	48.80	23.32	0.831	0.201	39.43	29.79	0.901	0.258	43.21
Compressed3D [26] (CVPR'24)	26.98	0.801	0.238	28.90	23.32	0.832	0.194	17.28	29.38	0.898	0.253	25.30
HAC [6]	27.53	0.807	0.238	15.26	24.04	0.846	0.187	8.10	29.98	0.902	0.269	4.35
Context-GS [35]	27.75	0.811	0.231	18.41	24.29	0.855	0.176	11.80	30.39	0.909	0.258	6.60
Ours ($\lambda = 0.005$)	25.73	0.795	0.234	25.83	23.35	0.834	0.194	9.71	27.23	0.885	0.287	5.80
Ours ($\lambda = 0.001$)	27.28	0.824	0.206	38.45	24.38	0.858	0.169	13.74	28.38	0.900	0.267	7.69
Ours ($\lambda = 0.0001$)	28.01	0.834	0.193	59.69	24.73	0.866	0.159	21.35	28.82	0.903	0.260	11.37

Table 4.6: **The quantitative compression results obtained from the proposed entropy models and other works in the literature.** 3D-GS [15] and Scaffold-GS [21] are non-compressed Gaussian splatting results included for reference. The results of other works are obtained from Context-GS [35]. Compared with other works in the literature, our approach with fully-factorized and mean-scale hyperprior entropy models achieves competitive results.

Compared with other works, our model with mean-scale hyperprior entropy model trained for $5K$ iterations after optimizing all Gaussian attributes for $25K$ iterations achieves competitive results. In the Mip-NeRF 360 dataset, our model only performs worse compared to HAC [6] and Context-GS [35] which employ

context models for compression and use Scaffold-GS [21] as a base Gaussian splatting method. For that reason, their method is not applicable to the generally adopted Gaussian splatting framework. In contrast, our approach performs better than other works that employ more general-case Gaussian splatting approaches such as LightGaussian [10], Compact3DGS [20], and Compressed3D [26]. This is a recurring theme on Tanks & Temples as well. Although it is difficult to compare our approach on DeepBlending, since our method achieves a higher compression rate with a higher quality drop, it is evident that our compression approach remains competitive against them.

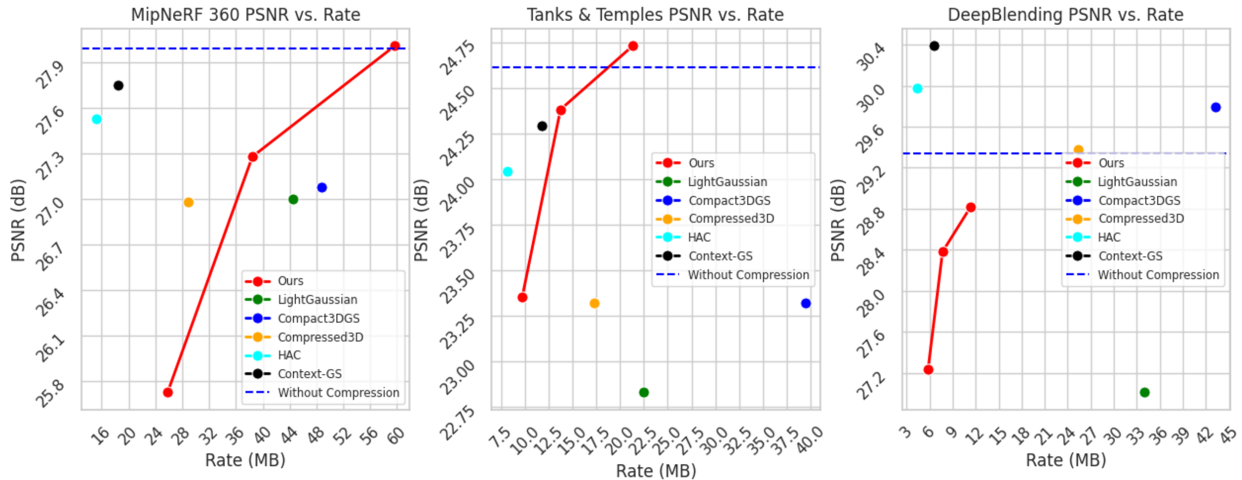

Figure 4.5: **Comparison of mean-scale hyperprior network against other works in the literature.** Our method achieves competitive results compared to LightGaussian [10], Compact3DGS [20] and Compressed3D [26] while achieving inferior results compared to HAC [6] and Context-GS [35].

As an important additional point, note that our compression approach is suitable as a post-training method for mainstream Gaussian splatting frameworks where Gaussians are composed of same attributes. As a result, our method is applicable to many cases where users can choose their desired use-case from low bandwidth to high visual quality.

4.4.4 Position Compression using DEFLATE and G-PCC

Although the compression of Gaussian primitive attributes except the Gaussian position (mean) has been handled by the learned entropy models so far, the position information was excluded from compression until now. The reason behind this choice was the fact that the Gaussian position information is highly sensitive to noise, and additional noise on position significantly degrades the visual quality. For that purpose, other works in the literature such as [26, 10] utilize the DEFLATE [9] algorithm for the compression of position information.

Similarly, our approach so far has been utilizing the DEFLATE algorithm. As a first step to compress the position information, we reduce the precision of the floating point from single precision (*float32*) to half precision (*float16*). Afterwards, we employ DEFLATE algorithm to reduce the size of position information. On the other hand, G-PCC [22] is a highly influential algorithm developed by MPEG for point cloud compression. Although Gaussian primitives have significantly different attributes compared to point cloud data which only has color and position attributes, the G-PCC algorithm can be utilized for compression of position information solely. For that purpose, we compare the use of the TMC13 algorithm, which is the position compression component of the G-PCC algorithm for Gaussian primitive position compression, against the

DEFLATE algorithm. Although our experiments are not matured into a complete algorithm, we evaluated the short-comings of both methods and compared the compression efficiency. For our comparisons, we report our results on Tanks & Temples [19] dataset in Table 4.7.

Method	Scene	# Gaussians	Encode Duration	Decode Duration	Size
DEFLATE [9]	Train	564,884	0.195	0.001	3.03
	Truck	1,013,077	0.252	0.001	5.50
TMC13 [22]	Train	564,884	7.63 + 34.92	5.59	0.78
	Truck	1,013,077	13.79 + 125.42	10.08	1.72

Table 4.7: **Comparison of DEFLATE and TMC13 position compression algorithms on Tanks & Temples [19] dataset.** We compare DEFLATE [9] and TMC13 [22] compression algorithms for position compression. For our method, we found that DEFLATE algorithm is preferable due to undesired permutation performed by TMC13 during encoding. Since TMC13 permutes Gaussian primitive orders, we need to match Gaussian primitive orders in an ad-hoc manner, causing an increase in encode time depicted in red.

When utilizing DEFLATE algorithm for position compression, the algorithm is considerably faster compared to TMC13 algorithm, even though TMC13 yields a lower memory requirement. One reason for the slower encode speed of TMC13 is the permutation of Gaussian primitives' order during encoding. In order to have matching orders for positions and attributes of Gaussians, we find the closest points in pre-encoded and post-encoded positions to perform the same permutation on attributes as well. For that reason, the encode duration increases significantly. Although this increase can be alleviated with a more parallelized and complex method, the encode duration even without post-permutation is significantly higher, forcing us to continue with DEFLATE algorithm.

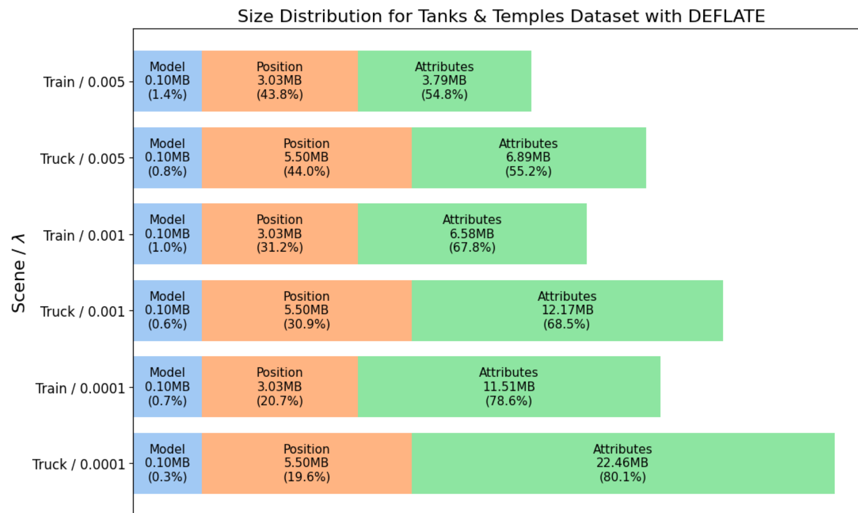

Figure 4.6: **The size distribution of compressed Gaussian splat representations with DEFLATE algorithm.** Using DEFLATE algorithm, we evaluate the size distribution of Gaussian primitives, decomposed into *entropy model parameters*, *Gaussian primitive positions*, and *Gaussian primitive attributes*. Our evaluation captures only Tanks & Temples since results are only scaled with respect to number of Gaussians for other datasets.

Further investigating Figure 4.6, it shows that position information captures a significant portion of the memory requirement. For that reason, further investigation and more caution is required for the compression of position information jointly with Gaussian primitive attributes. However, for our concerns, DEFLATE algorithm has been a simple yet effective compression algorithm alleviating the high sensitivity of Gaussian position information.

4.4.5 Multi-rate Gaussian Primitive Compression

During all our experiments, we treat the quantization steps as learnable parameters encapsulated by parameter s in the analysis and synthesis transforms depicted in Figure 3.3.
The quantization steps are scaled up and down with respect to the rate-distortion (RD) trade-off parameter, λ, in the loss function, $L = D(\hat{x},\ x) + \lambda R$. Due to the fact that we are using single RD trade-off parameter in all previous experiments, we required to train a different model for each RD point. To alleviate this problem and improve parameter-efficiency, we propose utilizing the "gain units" proposed by Cui et al. [8] as described in Section 3.4.3. Specifically, instead of training for a single RD trade-off parameter, λ, we take a list of parameters, $\lambda = \{\ 0.005,\ 0.001,\ 0.0005,\ 0.0001\ \}$ and associate each with a different learnable scaling vector, $\mathbf{S} = \{\ \mathbf{s}_0,\ \mathbf{s}_1,\ \mathbf{s}_2,\ \mathbf{s}_3\ \}$.
To run our training and optimize for all RD trade-off parameters, we sample a random RD trade-off parameter, λ' at each iteration, and optimize for the chosen RD trade-off parameter and its scaling vector, s' using loss, $L = D(\hat{x},\ x) + \lambda' R$. At each iteration, we element-wise multiply Gaussian attributes with the selected scaling vector for the analysis transform, and re-scale for the synthesis transform using the multiplicative inverse of the same scaling vector.

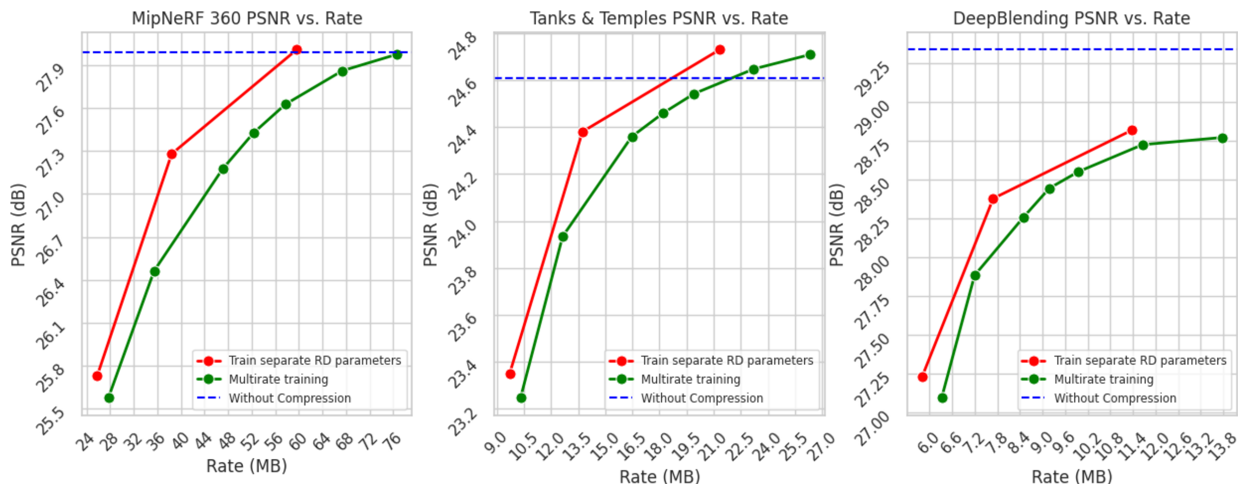

Figure 4.7: **Rate-distortion curves for multi-rate entropy model against training the entropy model separately for each rate-distortion parameter.** Rate-distortion performance of entropy model slightly falls when multi-rate training is performed.

Note that intermediate RD trade-off points can be achieved at test time by performing exponential interpolation on scaling parameters as explained in Section 3.4.3 which can be seen in Figure 4.7. In mentioned figure, every second RD point is obtained by performing exponential interpolation which does not correspond to a learned scaling vector but the interpolated version of two closest scaling vectors.
The rate-distortion performance of multi-rate training against single-rate training is compared in Figure 4.7. Based on the comparison on all datasets, it is evident that there is a degradation in rate-distortion

performance. Even though the visual quality degrades slightly at lower bitrates, the gap widens at higher bitrates and becomes significant. For instance, the results on Mip-NeRF 360 [2] dataset reveals that the gap is around 16 MB at highest bitrate. On the other hand, the gap at lowest bitrate on same dataset is relatively little at around 2 MB. However, the results are promising in terms of lower than expected degradation and an impressive generalization among different bitrates for entropy models given the fact that training of the entropy model took only $5,000$ iterations. Despite the promising results, we continue our analysis with our best-performing models that are achieved by training one model per each RD trade-off parameter.

4.4.6 Learned Quantization and Attribute Importance

During all of our experiments, we treat the quantization steps as learnable parameters encapsulated by the parameter $\mathbf{s}$ in the analysis and synthesis transforms depicted in Figure 3.3. Since the quantization step adjusts the amount of noise injected in the respective Gaussian attribute, the scaling parameter $\mathbf{s}$ can be treated as an indicator of attribute importance for Gaussian splatting, where a larger scaling value for the attribute (s_i) would be expected to indicate higher importance, while a lower scaling value would be expected to indicate lower importance. Note that a lower scaling parameter results in larger distortion when quantization is applied, while a larger scaling parameter yields a higher bitrate.
In the light of this hypothesis and expectation, Figure 4.8 reveals which attributes are more important based on the optimization process. In Figure 4.8, the mean absolute value of each Gaussian primitive attribute is drawn with a bar. Accordingly, there exists 16 bars for spherical harmonics since we utilize degree 3 for them where degree 0 corresponds to base color without any view-dependency and other coefficients represent view-dependent color properties. In addition, the attributes have 3 scaling coefficients for 3 dimensions, 4 rotation coefficients for quaternions, and 1 opacity attribute.
Observing three subplots in Figure 4.8 for each RD trade-off parameter $\lambda \in \{\ 0.005,\ 0.001,\ 0.0001\ \}$, one common theme reveals that scaling parameters are highly important compared to all other attributes since the optimization process forces them to be less affected by the quantization noise while other attributes are allowed to be more affected from quantization. On the other hand, another important observation is that spherical harmonic coefficients are relatively unimportant except for the base-degree coefficient. This result reveals the fact that most Gaussian primitives do not require view-dependent color as they collapse to value 0 during quantization anyways. Finally, note that the first coefficient of the rotation vector (quaternions) is deemed to be more important compared to other coefficients. This result is logical in the sense that the 0 -th coefficient is the real part of the quaternion and represents the angle of rotation.
As a result, our findings show that taking the scale magnitude as an attribute importance metric for Gaussian primitives, Gaussian scaling parameters are the most important parameters, followed by rotation, opacity, and spherical harmonics. In this comparison, position attributes are disregarded as we observed that they are highly prone to quantization noise when compared with other attributes.

Figure 4.8: **Effect of learned scaling on Gaussian primitive attributes and quantization on Tanks & Temples dataset [19].** Each bar represents the mean value of one attribute of Gaussian primitives. The standard deviation is visualized with black lines per attribute. The spherical harmonics are aggregated for R, G, B color channels to have 16 attributes instead of $16 \times 3 = 48$.

4.5 Hierarchy Generation for 3D Gaussian Primitives

To perform residual coding and aim achieving a reduced entropy for residual Gaussian attributes, the utilization of additional compression techniques is required. For that reason, we experiment using the hierarchy generation idea from [16] to perform residual coding as described in Section 3.5.3.

4.5.1 Effect of Depth Selection

In order to compress all Gaussian primitives, we draw inspiration from low-delay video compression and utilize intermediate nodes of the Octree structure as prior information for actual Gaussian primitives in the manner described in Section 3.5. Although the duration of the hierarchy generation with octree structure depends on the number of Gaussians, the duration for $\sim 600K$ Gaussian primitives on *train* scene from Tanks & Temples [19] dataset is approximately 10 milliseconds.
The octree structure has exponentially increasing number of nodes at each depth up until the actual number of Gaussian primitives as depicted in Figure 4.9. To choose a depth level for prior information, which has relatively few nodes for the prior information and forms a representative prediction of actual Gaussian primitives, we use Equation 3.40. Note that, the number of Gaussians that are required to be encoded and decoded are much fewer at lower depth levels.

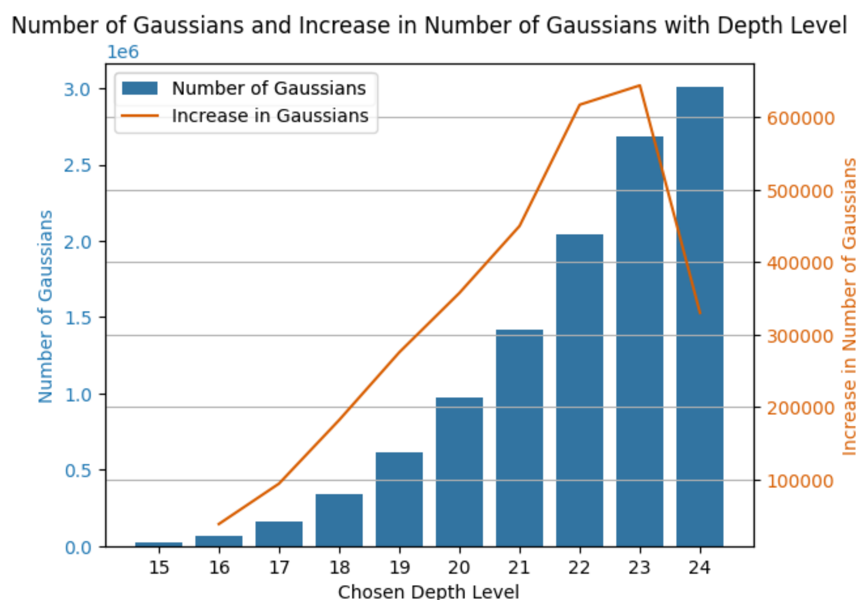

Figure 4.9: **Histogram for the increase in number of Gaussian primitives with increasing depth level in octree structure for *bicycle* scene from Mip-NeRF 360.** Coding of predicted Gaussians (intermediate nodes) enforces a trade-off between representativeness of intermediate nodes and number of intermediate nodes. For given scene, we utilize depth 20 as predictions for actual Gaussians, encoding $1/3$ of primitives as predictions.

Similarly, for different levels of the octree structure, renderings in Figure 4.10 can be acquired on *playroom* scene from DeepBlending [12] dataset. Note that the depth level for *playroom* scene is very low compared to larger scenes since *playroom* is a small scene and our adaptive depth selection method accounts for the maximum distance of an axis-aligned bounding box as stated in Equation 3.40.

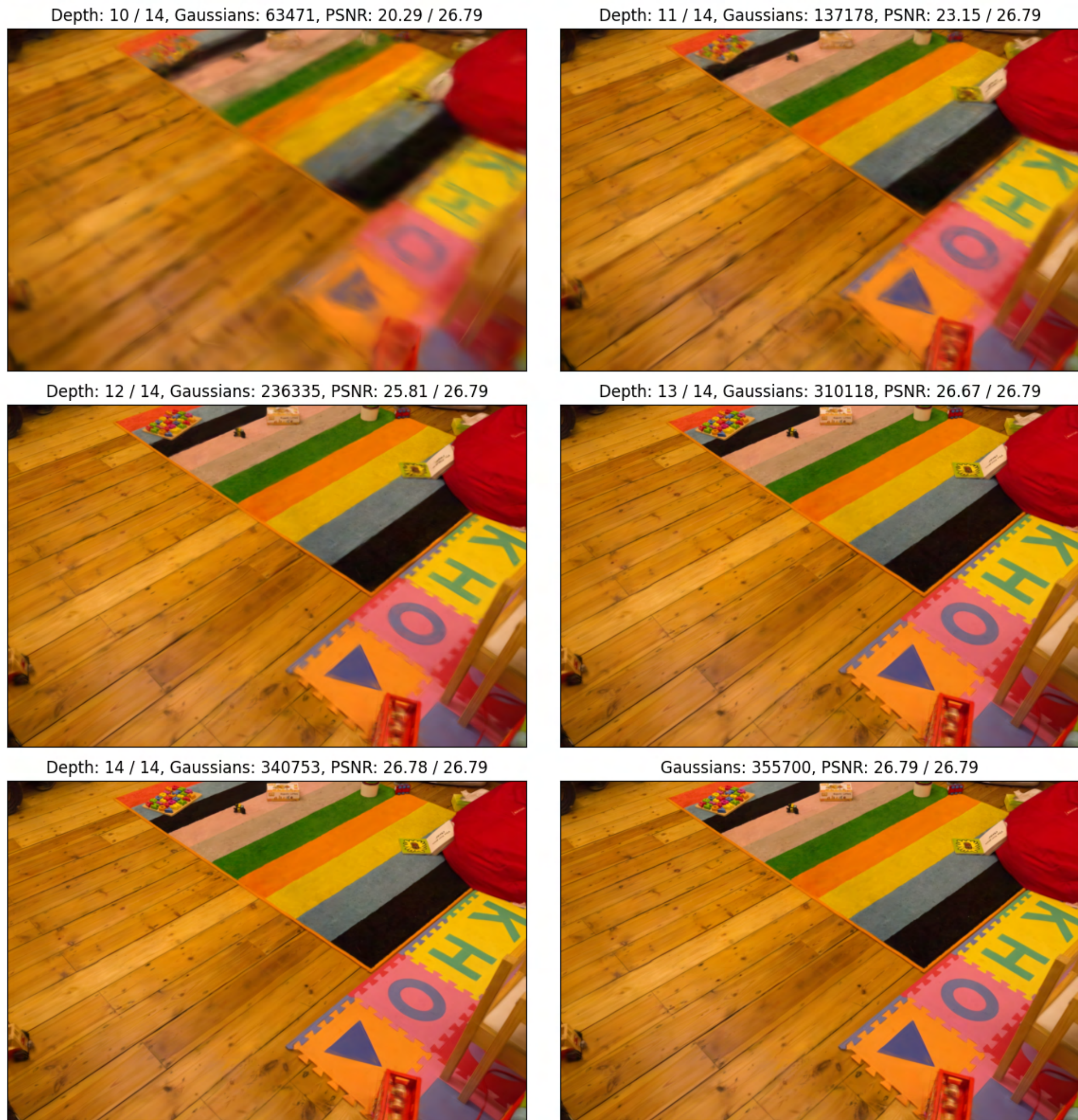

Figure 4.10: **Effect of depth variation on visual quality on *playroom* scene from DeepBlending [12].** Using different octree levels results in changing granularity on images. Specifically, close points appear more blurry while far away points are less affected from averaging of Gaussian primitives due to the resolution.

4.5.2 Residual Coding for Gaussian Primitives

Before analyzing the results for residual coding as described in Section 3.5.3, we analyze the distribution change for Gaussian primitive positions and covariance matrices to understand the effect of subtracting aggregated attributes from actual attributes to acquire residuals.

In Figure 4.11, we observe the change in distribution for Gaussian primitive positions before and after calculating residuals. As can be seen from the narrowing down of the probability density functions approximated using kernel density estimation, the entropy coding for position residuals can benefit from residual coding. While it is challenging to model a broad distribution, such as the distribution for actual positions and aggregated positions, the number of aggregated positions is significantly lower than that for actual positions, which should allow for lower bitrates.

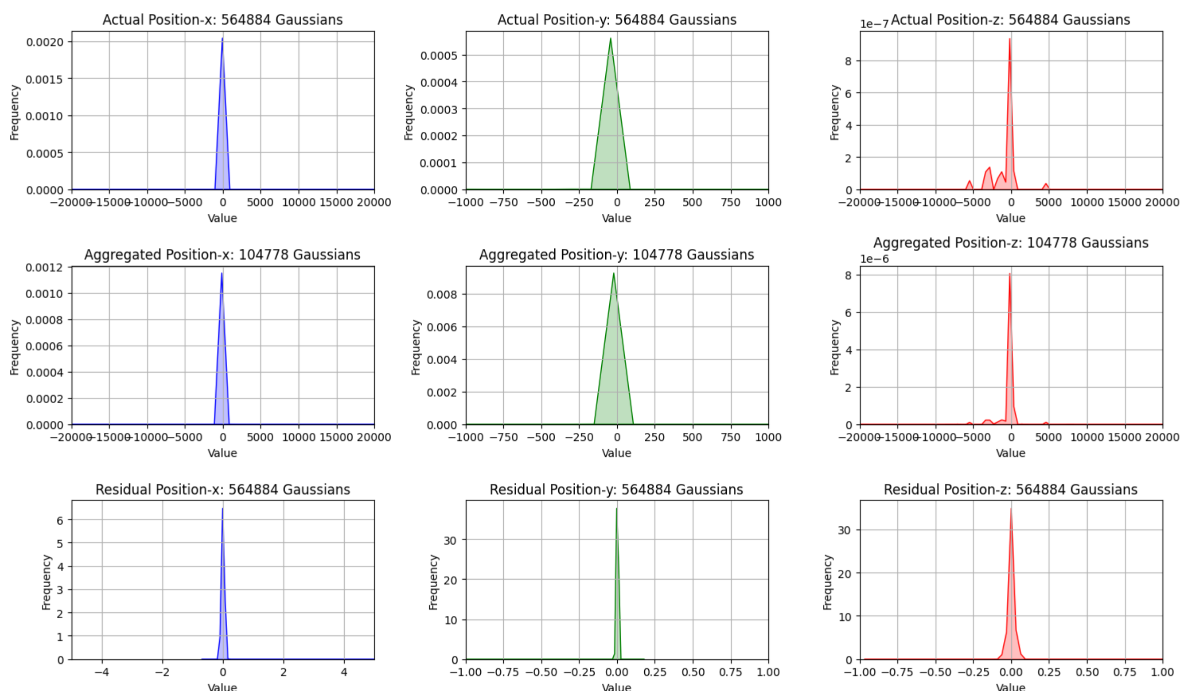

Figure 4.11: **Distribution of Gaussian primitive position, aggregated position, and residual position attributes.** The distribution of position per dimension is acquired using kernel density estimation. Note that the distribution gets narrower for residual position elements, indicating a lower entropy.

Similar to our approach in previous sections without residual coding, we compress the aggregated positions using DEFLATE algorithm [9] in order to prevent large distortion in position information due to its sensitivity. For the residual position, we compress them using learned entropy models similar to other attributes. For that reason, the bitrate consumed by the DEFLATE algorithm is reduced to $1/5$-th due to the lowered number of Gaussians after aggregation, while the residual position brings additional bitrate consumption which cannot be measured due to joint compression with other attributes.

On the other hand, for the compression of Gaussian primitive covariances either through direct compression of covariance matrices or compression of scaling and rotation attributes, we visualize the distribution change with residual calculation in Figure 4.12. Recall from Section 4.4.6, the scaling information is also highly sensitive to compression. For that reason, correct entropy modeling for Gaussian covariance information is of great importance. However, observing Figure 4.12, the distribution of residual covariance matrix elements has a wider distribution compared to aggregated or actual Gaussian primitive covariance matrix elements.

This result indicates that residual coding while using aggregated Gaussian primitives as predictions might not be very beneficial in terms of compression efficiency.

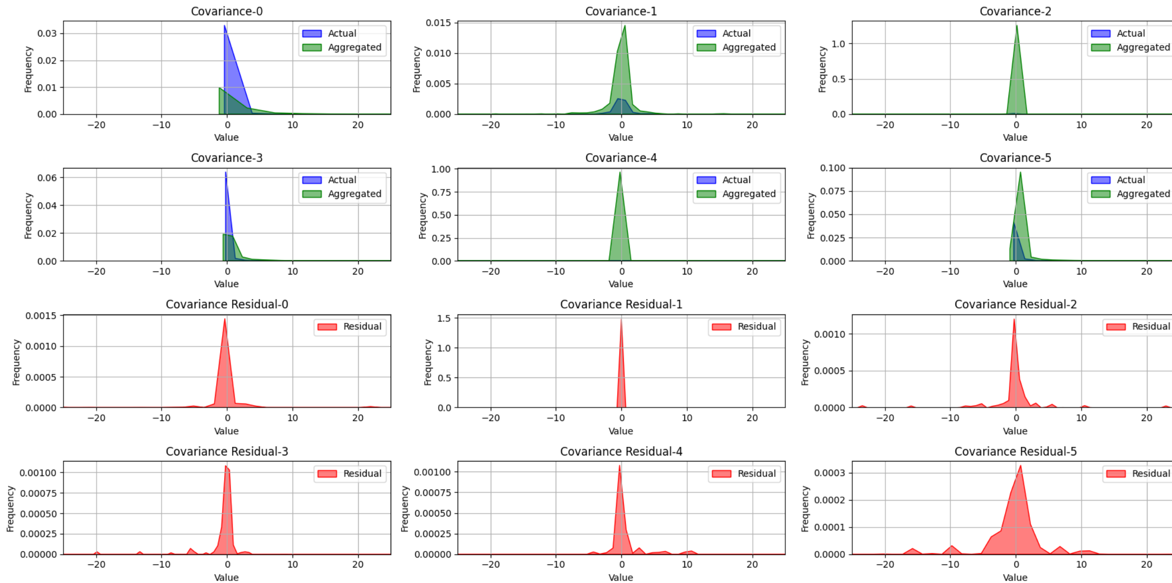

Figure 4.12: **Distribution of Gaussian primitive, aggregated, and residual covariance matrices.** The distribution of covariance is calculated using kernel density estimation. Note that the distribution of residual covariance matrix elements gets wider, indicating a larger entropy and higher bitrates for compression.

As explained in Section 3.5.3, a design choice is to compress either the covariance matrix while ensuring positive semidefiniteness through Cholesky decomposition or compressing scaling and rotation components after decomposing it usign eigen-decomposition. The comparison of these two approaches for residual coding, are depicted in Figure 4.13 and Table 4.8.

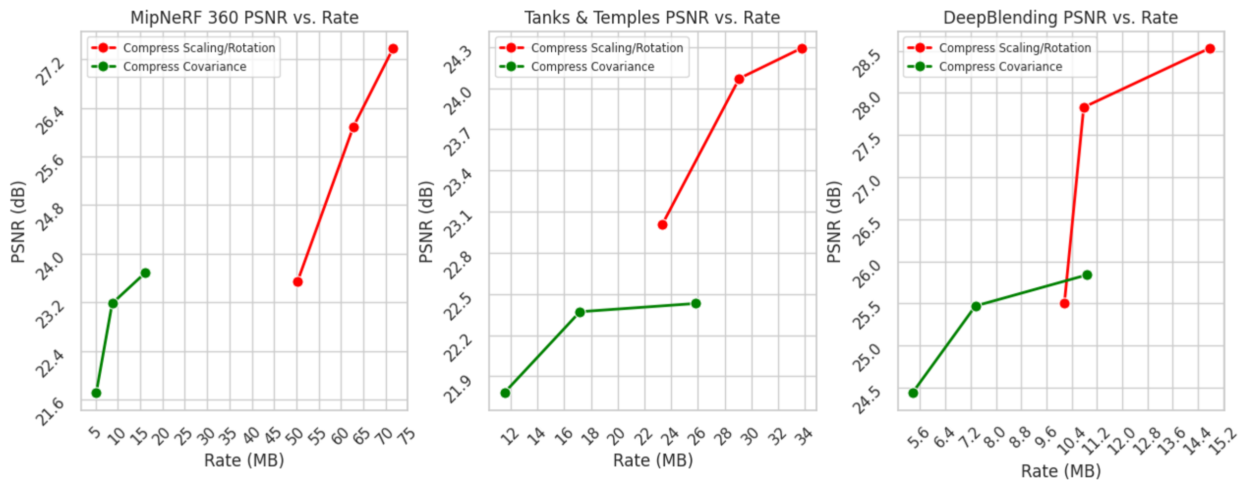

Figure 4.13: **Rate-distortion curve comparison for residual compression with compressing covariance matrix against compressing scaling and rotation decomposition.** Two approaches for compressing covariance information yields curves at significantly different bitrates despite using same hyperparameters.

		Mip-NeRF360 [3]				Tanks&Temples [16]				DeepBlending [14]			
		PSNR↑	SSIM↑	LPIPS	Size↓	PSNR	SSIM↑	LPIPS↓	Size↓	PSNR↑	SSIM↑	LPIPS↓	Size↓
Single Bottleneck	**Compress Scaling/Rotation** ($\lambda = 0.005$)	25.73	0.795	0.234	25.83	23.35	0.834	0.194	9.71	27.23	0.885	0.287	5.80
	Compress Scaling/Rotation ($\lambda = 0.001$)	27.28	0.824	0.206	38.45	24.38	0.858	0.169	13.74	28.38	0.900	0.267	7.69
	Compress Scaling/Rotation ($\lambda = 0.0001$)	28.01	0.834	0.193	59.69	24.73	0.866	0.159	21.35	28.82	0.903	0.260	11.37
Residual coding	**Compress Covariance** ($\lambda = 0.005$)	21.71	0.681	0.406	5.17	21.78	0.761	0.280	11.52	24.44	0.810	0.366	5.36
	Compress Covariance ($\lambda = 0.001$)	23.18	0.751	0.372	8.73	22.37	0.781	0.261	17.17	25.47	0.836	0.346	7.35
	Compress Covariance ($\lambda = 0.0001$)	23.68	0.771	0.353	16.08	22.43	0.783	0.259	25.79	25.84	0.843	0.338	10.89
	Compress Scaling/Rotation ($\lambda = 0.005$)	23.54	0.694	0.307	50.26	23.01	0.824	0.207	23.35	25.51	0.826	0.342	10.17
	Compress Scaling/Rotation ($\lambda = 0.001$)	26.09	0.791	0.231	62.79	24.07	0.851	0.175	29.09	27.83	0.890	0.276	10.78
	Compress Scaling/Rotation ($\lambda = 0.0001$)	27.38	0.817	0.206	71.69	24.29	0.857	0.167	33.73	28.53	0.898	0.266	14.76

Table 4.8: **Comparison of residual coding methods and single bottleneck compression.** We compare residual coding either with compressing covariance or scaling and rotation components. Single bottleneck model is the previously reported mean-scale hyperprior network without optimizing the geometry attributes during entropy model training.

In both cases where compressing either covariance matrix or the scaling and rotation attributes, the compression of residuals together with aggregated Gaussians work significantly worse than compressing actual Gaussian attributes in a single bottleneck as shown in Table 4.8. Although the reason for this situation is not crystal clear, the adverse change in the distribution for covariance is a possible explanation of this result. Since the residual Gaussian attributes do not have a probability distribution that is significantly simpler to compress, the bitrate is increased due to the increased entropy of residuals and requirement of additional aggregated Gaussian compression. Further evaluating the size distribution born from residual coding in Figure 4.14, we confirm that the residual Gaussian attributes require a significant bitrate which is not conforming to observations from video compression and frame residuals. For residual compression to be efficient, the residual attributes need to consume a much lower bitrate.

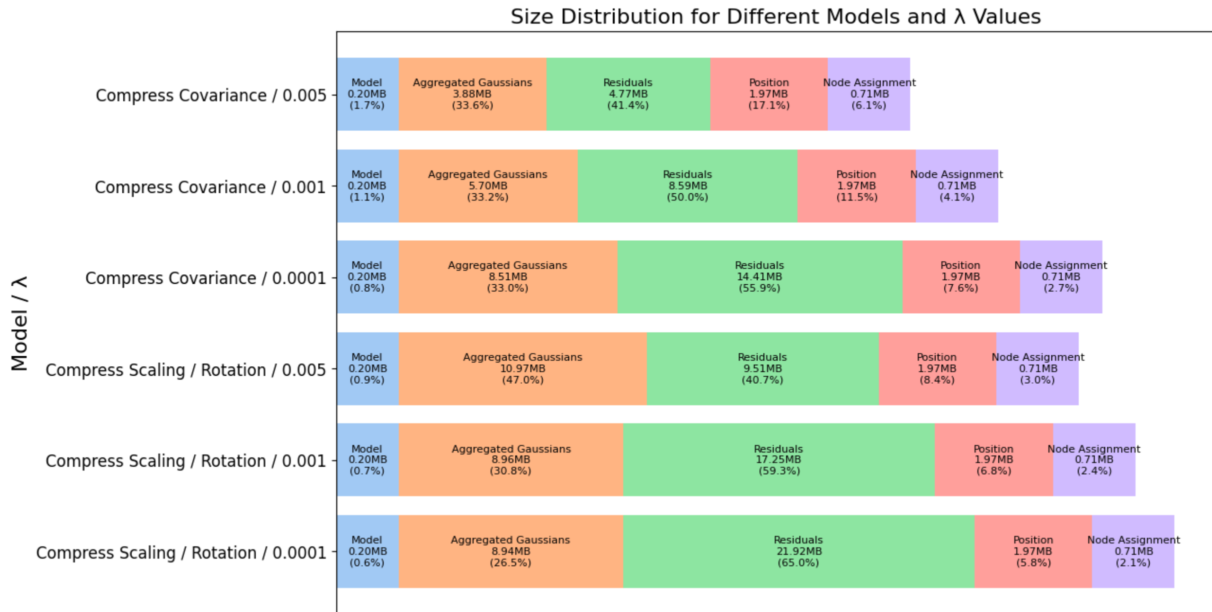

Figure 4.14: **Size distribution for residual compression.** Most of the bitrate requirement is first due to the residuals and then the aggregated Gaussians. This result is in contrast with the expectations based on the video compression literature as residuals are expected to require small amount of memory.

Chapter 5

Discussion

In this chapter, we interpret and discuss the findings from our experiments on Gaussian splatting, pruning, and compression utilizing learned entropy models and hierarchical structure. The results discussed here are drawn from the detailed experiments and evaluations performed in Section 4.

5.1 3D Gaussian Splatting Optimization Improvement

The comparison between 3D-MCMC [17] and the seminal 3D-GS [15] method shows that 3D-MCMC offers superior visual quality without increasing the number of Gaussian primitives. This improvement is particularly evident in complex regions of the scenes where 3D-GS tends to produce blurry results. The main difference between 3D-GS and 3D-MCMC is the optimization algorithm without changing the general framework. Thus, using 3D-MCMC is a costless improvement while remaining in sync with generally adopted Gaussian primitive properties. Quantitative and qualitative analysis, as demonstrated in Table 4.1 and Figure 4.1, confirm the superiority of 3D-MCMC. Since higher visual quality from Gaussian primitives directly translates into better rate-distortion performance, our choice to use it as a baseline for learned entropy modeling is justified.
Compared to Scaffold-GS [21], 3D-MCMC generally achieves higher quality while following a framework similar to 3D-GS. However, both 3D-GS and Scaffold-GS outperform 3D-MCMC in the DeepBlending [12] dataset. This result aligns with the results reported by 3D-MCMC [17]. On the other hand, Scaffold-GS employs a significantly different Gaussian Splatting framework with anchor primitives and neural Gaussians. Thus, the Scaffold-GS framework does not align well with many applications that are adopted by a large portion of end-users. For that reason, employing 3D-MCMC is a more reasonable approach from both the perspective of visual quality and adoption.

5.2 3D Gaussian Splatting Primitive Pruning

Our experiments with learned masking [20] and RadSplat pruning [27] methods revealed that both approaches are effective in reducing the number of Gaussian primitives while maintaining visual quality. The RadSplat method showed a slight edge over the learned masking in terms of simplicity and effectiveness, as seen in Table 4.2. By applying the pruning only twice throughout training and being invariant to number of training views, RadSplat pruning appears as a simpler pruning method compared to learned masking which requires learning one additional mask parameter for each Gaussian primitive and applying pruning throughout whole training.
Inspecting Figure 4.2, it is reassuring to see that the distribution of Gaussian primitives is highly overlapping in terms of position for both learned masking and RadSplat pruning. This result confirms the redundancy of

the primitive Gaussian optimization methods and that both approaches target similar redundant Gaussians. Furthermore, having a reduction in the number of Gaussians not only reduces the memory requirements but also improves the rasterization speed, making it more practical for real-time applications.

5.3 Gaussian Primitive Compression using Learned Entropy Models

Fully-factorized and mean-scale hyperprior entropy models demonstrated significant compression capabilities, while the mean-scale hyperprior model consistently outperformed the fully-factorized model in compression rates and visual quality, as indicated in Table 4.3 and Figure 4.3. Training the entropy model for $5K$ iterations after optimizing Gaussian primitives for $25K$ iterations proved to be computationally efficient without compromising performance significantly, as shown in Table 4.4. This result is intuitive when considering the overfitting setting of our entropy models. Despite the increased compression efficiency of learned entropy models in generalizable image compression domain with increased training duration, such long training duration appears redundant for our single scene setting. Although longer training of the entropy model improves rate-distortion performance, it increases computational cost and training time as expected. Furthermore, optimizing geometry attributes with entropy penalty negatively affects performance due to their sensitivity. Therefore, keeping the geometry attributes fixed significantly improves rate-distortion performance.

5.3.1 Position Compression

For the compression of Gaussian primitive positions, our experiments show that the position information is more sensitive to quantization errors. This result is consistent with other works in the literature [10, 26]. Thus, we prefer the DEFLATE algorithm with precision reduction from single-point precision to half-point precision. DEFLATE algorithm was found to be effective and computation-efficient compared to the TMC13 algorithm due to the permutation performed by TMC13. Although TMC13 provided better compression ratios, its significant encoding time and the need for post-permutation adjustments made DEFLATE a more practical choice, as detailed in Table 4.7.

As indicated, many of the works except HAC [6] and Context-GS [35] use the same approach as we preferred with DEFLATE algorithm. The reason for HAC and Context-GS to not use the same approach is primarily the significantly different Gaussian splatting framework they use with taking Scaffold-GS as a base for compression.

5.3.2 Multi-rate Compression

Introducing multi-rate compression using gain units allowed us to achieve multiple rate-distortion trade-offs with a single model. Although this approach slightly degraded the rate-distortion performance compared to training separate models for each rate, it offered a more parameter-efficient and flexible solution, as illustrated in Figure 4.7. Although the multi-rate implementation with multiple scaling parameters and single entropy model results in a slight degradation, training multiple models for the full rate-distortion curve remains as primary choice since the training overhead is negligible with learned entropy modeling. However, multi-rate compression is a promising avenue to follow for future research since results from image compression display results without degradation while achieving the same compression ratios as training single model for each rate-distortion trade-off.

5.3.3 Limitations and Relation to Previous Work

Comparing our approach with previous work depicted in Table 4.6 and Figure 4.5, HAC [6] and Context-GS [35] appear superior to our model, while our approach with mean-scale hyperprior is competitive against LightGaussian [10], Compact3DGS [20], and Compressed3D [26]. Our work is concurrent with others due to the temporal proximity of the 3D Gaussian splatting approach.
Firstly, aligning with the observations of Papantonakis et al. [28], we observe that view-dependent part of the spherical harmonics are redundant for most cases since the learned importance with scale parameters of our learned entropy models are very low for the corresponding coefficients. Thus, they are quantized heavily as it induces negligible cost for visual quality.
When compared with LightGaussian [10], Compact3DGS [20] and Compressed3D [26], their approach does not require any explicit entropy modeling and rate-distortion optimization. These approaches depend on clustering and codebook creation from vector quantization, requiring less computation. However, their approach is significantly more complex with multiple components with ad-hoc methods such as importance scoring and knowledge distillation. Our method, with a single hyperprior network, achieves either superior or competitive results on Mip-NeRF 360 [2], Tanks & Temples [19], and DeepBlending [12] datasets.
Among other provided related compression works, HAC [6] and Context-GS [35] also utilize a learned entropy model with a similar approach where they model the Gaussian parameters with a Gaussian distribution. However, as their approach takes Scaffold-GS as basis, their approach utilizes latent features for Gaussian primitive attributes which allows a better modeling during optimization. On the other hand, their approach has a higher computational cost due to the MLPs that are used to decode the latent features into Gaussian attributes. Furthermore, as the underlying structure with anchor primitives in Scaffold-GS provides an inherent context model, both HAC and Context-GS make use of the spatial correlation. Context-GS employs additional structure with an octree hierarchy among anchor primitives, resulting in even further increase in compression efficiency with a small cost of computational complexity. On the other hand, since the adoption for Scaffold-GS framework is low in the literature and among end-users, both methods might lack integration with actual 3D-GS [15] approach. Our approach, however, can be seamlessly integrated with most works in literature and industry, as it adopts an additional learned entropy model that can be trained with regular 3D Gaussian primitives after $25K$ iterations.
As a result, our approach has a proper balance between computational complexity and bitrate reduction making it suitable for end-user adoption to reduce the inherent storage complexity of 3D Gaussian splatting. Furthermore, it can be easily integrated to most preferred 3D Gaussian splatting representation as a post-optimization algorithm after $25K$ iterations of Gaussian primitive optimization.
As a final limitation of our work with applying learned entropy modeling for Gaussian primitives, the instant rendering is prevented due to the imposed decoding duration. Although the decoding of the Gaussian primitives takes around 3 seconds for $\sim 600K$ Gaussian primitives for the *train* scene in Tanks & Temples [19] dataset, this might be still a concern for some applications requiring instant rendering. This problem is avoided in LightGaussian [10], Compressed3D [26], and Compact3DGS [20] as they do not utilize learned entropy models. On the other hand, HAC [6] and Context-GS [35] require to spend the same duration for the decoding process as well.

5.4 Residual Coding for Gaussian Primitive Compression

The results achieved for residual coding were against the expectations as they yield lower compression efficiency and visual quality compared to single bottleneck learned entropy models presented in Section 4.4. We experimented with compressing the covariance matrix and compressing the scaling and rotation attributes of Gaussian primitives for inter-predictions; both yielded inferior results compared to single bottleneck entropy models.

One possible explanation for this result is the higher entropy of residuals for Gaussian covariance matrices as observed in Figure 4.12. Although video compression typically requires lower entropy for residuals, we observe the opposite, likely due to faults in inter-prediction with hierarchical structure and Gaussian aggregation. Figure 4.14 shows that shows that Gaussian primitive residuals require a large portion of bitrate, contrary to expectations for residual coding in video compression. This indicates that although the hierarchical structure proposed in [16] provides decent aggregated Gaussians, they are not predictive for actual Gaussians. We achieve a lower entropy for residual Gaussian positions as expected, since the hierarchy generation is based on the Gaussian positions, while other attributes might vary within a set of Gaussians belonging to the same node.

In an ablation study on compressing either the covariance matrices or the scaling and rotation attributes, we observe that the two approaches yield significantly different bitrates, with covariance compression yielding slightly worse compression efficiency. This result can be explained by the differing importance of scaling and rotation attributes, which are multiplied to calculate covariance matrices. Since scaling attributes usually have larger value ranges, compressing them separately requires larger bitrates even with the same rate-distortion parameter λ. Additionally, directly compressing the covariance matrix is susceptible to noise, causing random rotations for Gaussians, whereas compressing scaling and rotation attributes avoids this problem. Therefore, we choose to compress the scaling and rotation attributes separately rather than the covariance matrix.

Chapter 6

Conclusion

In conclusion, our exploration into compressing Gaussian splatting representation through improving optimization framework with Markov Chain Monte Carlo (MCMC), applying primitive pruning, and compressing Gaussian primitives with learned entropy models has yielded promising results. The integration of the MCMC framework enhances the visual quality of novel view synthesis without increasing the computational cost significantly, making it a cost-effective improvement over seminal 3D-GS. Additionally, our comparative analysis indicates that although methods like Scaffold-GS can outperform in DeepBlending dataset, 3D-MCMC maintains broader applicability and superior visual quality in general use cases.

Our experiments with pruning techniques, particularly the pruning method proposed by RadSplat, demonstrate effective reduction of Gaussian primitives while maintaining visual quality, thereby reducing storage complexity and improving rasterization speed. The compression of Gaussian primitives using learned entropy models, specifically mean-scale hyperprior models, shows substantial reductions in storage requirements while preserving visual quality. This approach, which does not necessitate modifications to the Gaussian primitives, ensures easy adoption and integration into existing deployments.

The multi-rate compression using multiple scale vectors introduced flexibility in achieving various rate-distortion trade-offs, albeit with a slight degradation in performance compared to training separate models for each rate. This indicates a potential avenue for future research to balance parameter efficiency and compression performance.

However, our investigation of residual coding for Gaussian primitive compression reveals limitations. By visualizing the estimated distribution of Gaussian covariance matrices, we observe a higher entropy in residuals, suggesting that more advanced prediction schemes may be necessary to enhance compression efficiency. The DEFLATE algorithm, with precision reduction, proves to be a more practical choice over TMC13 for position compression due to its computational efficiency, despite TMC13's better compression ratios.

Overall, our findings suggest that while significant advancements have been made in optimizing 3D Gaussian splatting for reduced storage complexity and improved visual quality, there remain opportunities for further research. We note that better prediction schemes and exploration of more sophisticated hierarchical structures could provide additional gains in storage efficiency.

Appendix A

Experiment Results of Pruning Methods

This chapter includes training details and extensive results for pruning methods. To apply pruning using learned masking [20] and RadSplat pruning [27], we follow slightly varying training strategies without changing the hyperparameters for Gaussian primitive optimization. Only thing that is differing is the pruning related hyperparameters for two experiments for fair evaluation.

As described in Section 4.3, we use apply RadSplat pruning with importance score threshold 0.01 and apply RadSplat pruning at every $16K$ and $24K$ iterations. For learned masking, we apply masking at every 1000 iterations for the full training until $30K$ iterations with mask learning rate of 0.01, mask magnitude penalty $\lambda = 0.001$, and mask pruning threshold 0.01.

	3D-MCMC		with Masking		with Pruning	
Scene	# Gaussians	PSNR / SSIM / LPIPS	# Gaussians	PSNR / SSIM / LPIPS	# Gaussians	PSNR / SSIM / LPIPS
train	1,000,000	22.80 / 0.838 / 0.190	601,214	22.79 / 0.838 / 0.192	564,884	22.82 / 0.838 / 0.192
truck	2,500,000	26.44 / 0.898 / 0.117	1,111,206	26.41 / 0.899 / 0.117	1,013,077	26.40 / 0.898 / 0.119
average	1,750,000	24.65 / 0.868 / 0.154	856,210	24.60 / 0.868 / 0.154	788,980	24.61 / 0.867 / 0.157

Table A.1: **Comparison of learned masking against RadSplat pruning method on 3D-MCMC on Tanks & Temples [19] scenes.**

	3D-MCMC		with Masking		with Pruning	
Scene	# Gaussians	PSNR / SSIM / LPIPS	# Gaussians	PSNR / SSIM / LPIPS	# Gaussians	PSNR / SSIM / LPIPS
playroom	2,300,000	30.32 / 0.913 / 0.251	415,697	29.94 / 0.911 / 0.252	355,700	29.90 / 0.906 / 0.255
drjohnson	3,200,000	28.63 / 0.902 / 0.253	833,282	28.57 / 0.902 / 0.255	655,030	28.77 / 0.903 / 0.256
average	2,750,000	29.48 / 0.908 / 0.252	624,490	29.26 / 0.906 / 0.254	505,365	29.33 / 0.905 / 0.256

Table A.2: **Comparison of learned masking against RadSplat pruning method on 3D-MCMC on Deep-Blending [12] scenes.** The Gaussian primitives for each method are optimized for $30,000$ iterations.

	3D-MCMC		with Masking		with Pruning	
Scene	# Gaussians	PSNR / SSIM / LPIPS	# Gaussians	PSNR / SSIM / LPIPS	# Gaussians	PSNR / SSIM / LPIPS
bicycle	6,000,000	25.66 / 0.797 / 0.174	3,538,333	25.64 / 0.797 / 0.174	3,205,464	25.69 / 0.797 / 0.174
bonsai	1,200,000	32.54 / 0.947 / 0.195	702,349	32.49 / 0.947 / 0.196	693,450	32.52 / 0.946 / 0.196
counter	1,200,000	29.28 / 0.916 / 0.188	750,779	29.31 / 0.916 / 0.189	739,895	29.27 / 0.915 / 0.190
flowers	3,500,000	21.99 / 0.642 / 0.292	2,686,961	22.02 / 0.643 / 0.292	2,609,128	22.06 / 0.643 / 0.290
garden	5,800,000	27.79 / 0.877 / 0.096	4,346,106	27.77 / 0.877 / 0.096	4,090,208	27.75 / 0.876 / 0.096
kitchen	1,800,000	31.99 / 0.933 / 0.123	1,221,990	31.99 / 0.932 / 0.123	1,197,397	31.99 / 0.932 / 0.123
room	1,500,000	32.13 / 0.928 / 0.203	598,284	32.03 / 0.927 / 0.205	570,579	32.14 / 0.928 / 0.205
stump	4,800,000	27.33 / 0.811 / 0.174	3,150,719	27.46 / 0.812 / 0.173	2,994,861	27.43 / 0.812 / 0.173
treehill	3,700,000	23.05 / 0.660 / 0.277	2,535,880	23.06 / 0.661 / 0.278	2,426,631	23.04 / 0.660 / 0.277
average	3,277,778	27.97 / 0.834 / 0.191	2,170,155	27.98 / 0.835 / 0.192	2,058,623	27.99 / 0.835 / 0.192

Table A.3: **Comparison of learned masking against RadSplat pruning method on 3D-MCMC on Mip-NeRF 360 [2] scenes.**

Appendix B

Experiment Results of Single Lagrangian Training and Compression

This chapter includes training details and extensive results for single Lagrangian compression models. Specifically, we provide details for following cases:

1. Training and comparing fully-factorized entropy model and mean-scale hyperprior (hierarchical) entropy model,
2. Training the entropy model after optimizing Gaussian primitives $25K$ iterations and training for $5K$ iterations against training the entropy model after optimizing Gaussian primitives $30K$ iterations and training for $10K$ iterations.

For both cases, we jointly optimize the Gaussian parameters including geometry attributes such as position, scaling, and rotation) while training the entropy model. Training details and hyperparameters for both models can be found in Section 3.4. Our experiment results are provided for both fully-factorized [1] and mean-scale hyperprior [25] entropy models.

B.0.1 Training Entropy Model for 5K After Optimizing Gaussian Primitives for 25K Iterations

For each scene from Tanks & Temples [19], DeepBlending [12], and Mip-NeRF 360 [2], we train one model for each rate-distortion tradeoff. Specifically, we train one model for each scene and λ where $\lambda \in \{0.005,\ 0.001,\ 0.0001\}$.

3D-MCMC with RadSplat pruning with $\lambda = 0.005$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
train	6.61	21.34 / 0.791 / 0.240	6.50	21.49 / 0.794 / 0.236	127.14	22.82 / 0.838 / 0.192
truck	11.82	24.48 / 0.852 / 0.172	11.68	24.50 / 0.853 / 0.171	228.01	26.40 / 0.898 / 0.119
average	9.22	22.91 / 0.822 / 0.206	9.09	23.00 / 0.824 / 0.204	177.58	24.61 / 0.868 / 0.156

Table B.1: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.005$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.005$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
playroom	3.63	26.51 / 0.876 / 0.303	3.70	26.52 / 0.875 / 0.305	80.06	29.90 / 0.906 / 0.255
drjohnson	6.71	26.72 / 0.876 / 0.300	6.65	26.82 / 0.876 / 0.301	147.43	28.77 / 0.903 / 0.256
average	5.17	26.62 / 0.876 / 0.302	5.17	26.67 / 0.876 / 0.303	113.75	29.34 / 0.905 / 0.256

Table B.2: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.005$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.005$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
bicycle	39.32	24.08 / 0.739 / 0.229	37.43	24.22 / 0.743 / 0.223	721.45	25.69 / 0.797 / 0.174
bonsai	7.93	27.24 / 0.886 / 0.259	7.81	27.50 / 0.891 / 0.257	156.07	32.52 / 0.946 / 0.196
counter	8.73	26.22 / 0.853 / 0.257	8.71	26.41 / 0.857 / 0.254	166.53	29.27 / 0.915 / 0.190
flowers	33.08	21.34 / 0.598 / 0.326	32.65	21.37 / 0.600 / 0.326	587.23	22.06 / 0.643 / 0.290
garden	46.28	25.69 / 0.811 / 0.164	45.29	25.73 / 0.813 / 0.161	920.57	27.75 / 0.876 / 0.096
kitchen	12.78	27.69 / 0.874 / 0.186	12.46	28.12 / 0.880 / 0.180	269.49	31.99 / 0.932 / 0.123
room	6.06	28.18 / 0.875 / 0.267	6.08	28.32 / 0.879 / 0.265	128.41	32.14 / 0.928 / 0.205
stump	37.11	25.75 / 0.748 / 0.237	36.84	25.77 / 0.748 / 0.236	674.04	27.43 / 0.812 / 0.173
treehill	31.67	22.37 / 0.617 / 0.337	30.83	22.41 / 0.619 / 0.332	546.15	23.04 / 0.660 / 0.277
average	24.88	25.40 / 0.778 / 0.251	24.12	25.54 / 0.781 / 0.248	463.44	28.10 / 0.834 / 0.192

Table B.3: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.005$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.001$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
train	9.71	22.36 / 0.818 / 0.213	9.37	22.42 / 0.819 / 0.212	127.14	22.82 / 0.838 / 0.192
truck	17.70	25.55 / 0.877 / 0.144	17.34	25.60 / 0.877 / 0.143	228.01	26.40 / 0.898 / 0.119
average	13.71	23.96 / 0.848 / 0.179	13.36	24.01 / 0.848 / 0.178	177.58	24.61 / 0.868 / 0.156

Table B.4: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.001$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
playroom	5.49	27.61 / 0.889 / 0.285	5.53	27.61 / 0.890 / 0.285	80.06	29.90 / 0.906 / 0.255
drjohnson	9.51	27.84 / 0.890 / 0.277	9.02	27.94 / 0.891 / 0.276	147.43	28.77 / 0.903 / 0.256
average	7.50	27.73 / 0.890 / 0.281	7.28	27.78 / 0.891 / 0.281	113.75	29.34 / 0.905 / 0.256

Table B.5: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.001$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
bicycle	60.15	25.03 / 0.770 / 0.200	56.59	25.12 / 0.773 / 0.197	721.45	25.69 / 0.797 / 0.174
bonsai	12.77	29.61 / 0.920 / 0.224	12.37	30.11 / 0.922 / 0.223	156.07	32.52 / 0.946 / 0.196
counter	13.86	28.06 / 0.889 / 0.220	13.74	28.14 / 0.890 / 0.219	166.53	29.27 / 0.915 / 0.190
flowers	50.61	21.75 / 0.625 / 0.303	49.58	21.75 / 0.625 / 0.303	587.23	22.06 / 0.643 / 0.290
garden	72.76	26.94 / 0.849 / 0.127	72.08	26.95 / 0.849 / 0.126	920.57	27.75 / 0.876 / 0.096
kitchen	20.77	30.17 / 0.909 / 0.149	20.19	30.29 / 0.911 / 0.148	269.49	31.99 / 0.932 / 0.123
room	9.35	30.12 / 0.903 / 0.237	9.15	30.24 / 0.905 / 0.236	128.41	32.14 / 0.928 / 0.205
stump	58.03	26.46 / 0.776 / 0.207	57.01	26.50 / 0.777 / 0.206	674.04	27.43 / 0.812 / 0.173
treehill	49.12	22.88 / 0.641 / 0.306	45.73	22.88 / 0.643 / 0.304	546.15	23.04 / 0.660 / 0.277
average	38.60	26.67 / 0.809 / 0.219	37.38	26.89 / 0.810 / 0.218	463.44	28.10 / 0.834 / 0.192

Table B.6: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.0001$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
train	15.96	22.65 / 0.829 / 0.201	14.30	22.65 / 0.829 / 0.201	127.14	22.82 / 0.838 / 0.192
truck	29.76	25.98 / 0.888 / 0.130	27.30	25.98 / 0.888 / 0.130	228.01	26.40 / 0.898 / 0.119
average	22.86	24.32 / 0.859 / 0.166	20.80	24.32 / 0.859 / 0.166	177.58	24.61 / 0.868 / 0.156

Table B.7: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.0001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.0001$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
playroom	9.19	28.32 / 0.895 / 0.276	8.93	28.30 / 0.895 / 0.276	80.06	29.90 / 0.906 / 0.255
drjohnson	16.20	28.29 / 0.895 / 0.268	14.45	28.30 / 0.896 / 0.267	147.43	28.77 / 0.903 / 0.256
average	12.70	28.31 / 0.895 / 0.272	11.69	28.30 / 0.896 / 0.272	113.75	29.34 / 0.905 / 0.256

Table B.8: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.0001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.0001$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
bicycle	99.15	25.40 / 0.784 / 0.179	87.55	25.39 / 0.784 / 0.182	721.45	25.69 / 0.797 / 0.174
bonsai	21.70	31.60 / 0.937 / 0.205	19.96	31.62 / 0.937 / 0.205	156.07	32.52 / 0.946 / 0.196
counter	22.94	28.77 / 0.904 / 0.202	21.17	28.77 / 0.904 / 0.202	166.53	29.27 / 0.915 / 0.190
flowers	82.77	21.92 / 0.636 / 0.293	75.93	21.92 / 0.636 / 0.293	587.23	22.06 / 0.643 / 0.290
garden	76.87	27.40 / 0.865 / 0.108	77.61	27.41 / 0.865 / 0.108	920.57	27.75 / 0.876 / 0.096
kitchen	35.58	31.26 / 0.922 / 0.133	32.56	31.26 / 0.923 / 0.132	269.49	31.99 / 0.932 / 0.123
room	16.24	31.34 / 0.917 / 0.221	15.13	31.36 / 0.917 / 0.221	128.41	32.14 / 0.928 / 0.205
stump	94.38	26.78 / 0.790 / 0.189	87.12	26.78 / 0.790 / 0.189	674.04	27.43 / 0.812 / 0.173
treehill	79.13	23.01 / 0.653 / 0.285	70.28	23.01 / 0.653 / 0.285	546.15	23.04 / 0.660 / 0.277
average	58.75	27.50 / 0.823 / 0.202	54.15	27.50 / 0.823 / 0.202	463.44	28.10 / 0.834 / 0.192

Table B.9: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.0001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations.

B.0.2 Training Entropy Model for 10K After Optimizing Gaussian Primitives for 30K Iterations

For each scene from Tanks & Temples [19], DeepBlending [12], and Mip-NeRF 360 [2], we train one model for each rate-distortion tradeoff. Specifically, we train one model for each scene and λ where $\lambda \in \{0.005,\ 0.001,\ 0.0001\}$.

3D-MCMC with RadSplat pruning with $\lambda = 0.005$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
train	6.71	21.41 / 0.790 / 0.238	6.65	21.49 / 0.793 / 0.237	127.14	22.82 / 0.838 / 0.192
truck	11.83	24.62 / 0.853 / 0.170	11.76	24.68 / 0.855 / 0.169	228.01	26.40 / 0.898 / 0.119
average	9.27	23.02 / 0.822 / 0.204	9.21	23.09 / 0.824 / 0.203	177.58	24.61 / 0.868 / 0.156

Table B.10: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.005$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.005$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
playroom	3.61	26.36 / 0.874 / 0.307	3.69	26.58 / 0.874 / 0.306	80.06	29.90 / 0.906 / 0.255
drjohnson	6.65	26.98 / 0.875 / 0.301	6.56	27.06 / 0.877 / 0.300	147.43	28.77 / 0.903 / 0.256
average	5.13	26.67 / 0.875 / 0.304	5.13	26.82 / 0.876 / 0.303	113.75	29.34 / 0.905 / 0.256

Table B.11: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.005$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.005$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
bicycle	40.53	24.22 / 0.741 / 0.226	39.35	24.29 / 0.744 / 0.223	721.45	25.69 / 0.797 / 0.174
bonsai	7.85	27.14 / 0.888 / 0.259	7.66	27.69 / 0.891 / 0.256	156.07	32.52 / 0.946 / 0.196
counter	8.67	26.28 / 0.855 / 0.257	8.53	26.42 / 0.858 / 0.255	166.53	29.27 / 0.915 / 0.190
flowers	33.81	21.31 / 0.598 / 0.325	33.39	21.33 / 0.601 / 0.323	587.23	22.06 / 0.643 / 0.290
garden	46.77	25.72 / 0.810 / 0.163	45.80	25.80 / 0.812 / 0.161	920.57	27.75 / 0.876 / 0.096
kitchen	12.69	28.00 / 0.877 / 0.184	12.58	28.26 / 0.881 / 0.180	269.49	31.99 / 0.932 / 0.123
room	6.00	28.48 / 0.876 / 0.270	6.01	28.67 / 0.879 / 0.268	128.41	32.14 / 0.928 / 0.205
stump	37.87	25.67 / 0.745 / 0.239	36.80	25.72 / 0.747 / 0.237	674.04	27.43 / 0.812 / 0.173
treehill	32.25	22.33 / 0.617 / 0.336	31.31	22.30 / 0.618 / 0.332	546.15	23.04 / 0.660 / 0.277
average	25.16	25.46 / 0.778 / 0.251	24.60	25.61 / 0.781 / 0.248	463.44	28.10 / 0.834 / 0.192

Table B.12: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.005$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.001$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
train	9.98	22.46 / 0.818 / 0.212	9.91	22.47 / 0.820 / 0.210	127.14	22.82 / 0.838 / 0.192
truck	18.00	25.66 / 0.877 / 0.143	17.67	25.68 / 0.878 / 0.142	228.01	26.40 / 0.898 / 0.119
average	13.99	24.06 / 0.848 / 0.178	13.79	24.08 / 0.849 / 0.176	177.58	24.61 / 0.868 / 0.156

Table B.13: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.001$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
playroom	5.58	27.46 / 0.885 / 0.292	5.64	27.52 / 0.887 / 0.289	80.06	29.90 / 0.906 / 0.255
drjohnson	9.86	27.99 / 0.891 / 0.277	9.51	27.99 / 0.891 / 0.276	147.43	28.77 / 0.903 / 0.256
average	7.72	27.73 / 0.888 / 0.285	7.58	27.76 / 0.889 / 0.283	113.75	29.34 / 0.905 / 0.256

Table B.14: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations.

	3D-MCMC with RadSplat pruning with $\lambda = 0.001$					
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
bicycle	63.06	25.09 / 0.770 / 0.198	60.20	25.16 / 0.772 / 0.196	721.45	25.69 / 0.797 / 0.174
bonsai	12.84	29.65 / 0.921 / 0.225	12.33	30.19 / 0.923 / 0.223	156.07	32.52 / 0.946 / 0.196
counter	14.08	28.13 / 0.890 / 0.220	13.79	28.24 / 0.891 / 0.219	166.53	29.27 / 0.915 / 0.190
flowers	52.66	21.71 / 0.625 / 0.301	51.63	21.74 / 0.627 / 0.300	587.23	22.06 / 0.643 / 0.290
garden	75.39	27.02 / 0.849 / 0.126	75.10	27.03 / 0.850 / 0.125	920.57	27.75 / 0.876 / 0.096
kitchen	21.15	30.46 / 0.912 / 0.148	20.94	30.53 / 0.913 / 0.146	269.49	31.99 / 0.932 / 0.123
room	9.35	30.34 / 0.903 / 0.241	9.14	30.52 / 0.904 / 0.240	128.41	32.14 / 0.928 / 0.205
stump	60.16	26.40 / 0.773 / 0.209	58.62	26.42 / 0.773 / 0.208	674.04	27.43 / 0.812 / 0.173
treehill	51.00	22.73 / 0.640 / 0.305	47.59	22.76 / 0.641 / 0.303	546.15	23.04 / 0.660 / 0.277
average	39.97	26.84 / 0.809 / 0.219	38.82	26.95 / 0.810 / 0.218	463.44	28.10 / 0.834 / 0.192

Table B.15: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations.

	3D-MCMC with RadSplat pruning with $\lambda = 0.0001$					
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
train	16.31	22.62 / 0.830 / 0.198	15.50	22.66 / 0.830 / 0.198	127.14	22.82 / 0.838 / 0.192
truck	29.91	26.03 / 0.889 / 0.129	28.54	26.03 / 0.889 / 0.129	228.01	26.40 / 0.898 / 0.119
average	23.11	24.33 / 0.860 / 0.164	22.02	24.35 / 0.860 / 0.164	177.58	24.61 / 0.868 / 0.156

Table B.16: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.0001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations.

	3D-MCMC with RadSplat pruning with $\lambda = 0.0001$					
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
playroom	9.68	27.89 / 0.890 / 0.284	9.38	27.88 / 0.890 / 0.283	80.06	29.90 / 0.906 / 0.255
drjohnson	16.57	28.41 / 0.896 / 0.267	15.76	28.45 / 0.896 / 0.267	147.43	28.77 / 0.903 / 0.256
average	13.13	28.15 / 0.893 / 0.276	12.57	28.17 / 0.893 / 0.275	113.75	29.34 / 0.905 / 0.256

Table B.17: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.0001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations.

3D-MCMC with RadSplat pruning with $\lambda = 0.0001$						
	Fully-Factorized		Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
bicycle	99.15	25.40 / 0.784 / 0.179	92.34	25.40 / 0.784 / 0.179	721.45	25.69 / 0.797 / 0.174
bonsai	21.49	31.68 / 0.937 / 0.205	20.81	31.72 / 0.938 / 0.205	156.07	32.52 / 0.946 / 0.196
counter	22.81	28.83 / 0.905 / 0.201	22.23	28.82 / 0.905 / 0.201	166.53	29.27 / 0.915 / 0.190
flowers	84.52	21.89 / 0.636 / 0.290	80.65	21.89 / 0.636 / 0.290	587.23	22.06 / 0.643 / 0.290
garden	120.94	27.45 / 0.865 / 0.107	119.74	27.44 / 0.865 / 0.107	920.57	27.75 / 0.876 / 0.096
kitchen	34.67	31.50 / 0.925 / 0.130	33.83	31.51 / 0.926 / 0.130	269.49	31.99 / 0.932 / 0.123
room	16.12	31.52 / 0.916 / 0.224	15.41	31.51 / 0.916 / 0.224	128.41	32.14 / 0.928 / 0.205
stump	95.32	26.67 / 0.785 / 0.191	91.30	26.68 / 0.785 / 0.192	674.04	27.43 / 0.812 / 0.173
treehill	78.36	22.89 / 0.651 / 0.283	75.12	22.89 / 0.651 / 0.282	546.15	23.04 / 0.660 / 0.277
average	63.71	27.53 / 0.823 / 0.201	61.27	27.54 / 0.823 / 0.201	463.44	28.10 / 0.834 / 0.192

Table B.18: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.0001$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations.

B.0.3 Experiment Results of Freezing Gaussian Primitive Geometry Attributes During Entropy Model Training

In contrast to experiments in previous sections of Appendix, we keep the Gaussian primitive geometry attributes, i.e. position, scaling, rotation, fixed in this section. The reason we try this is that the Gaussian splatting is more sensitive to geometry attributes of Gaussian primitives in contrast to visual attributes such as spherical harmonics as explained in Section 4.4.6.

For each case in this subsection, we train one model for each rate-distortion tradeoff. Specifically, we train one model for each scene and λ where $\lambda \in \{0.005,\ 0.001,\ 0.0001\}$. Since mean-scale hyperprior appears as slightly superior to fully-factorized entropy model, we repeat experiments with frozen geometry attributes only for mean-scale hyperprior entropy model.

3D-MCMC with RadSplat pruning with $\lambda = 0.005$				
	Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
train	6.93	21.73 / 0.802 / 0.230	127.14	22.82 / 0.838 / 0.192
truck	12.48	24.97 / 0.866 / 0.157	228.01	26.40 / 0.898 / 0.119
average	9.71	23.35 / 0.834 / 0.194	177.58	24.61 / 0.868 / 0.156

Table B.19: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.005$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives.

3D-MCMC with RadSplat pruning with $\lambda = 0.005$				
	Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
playroom	4.13	26.87 / 0.883 / 0.290	80.06	29.90 / 0.906 / 0.255
drjohnson	7.46	27.58 / 0.886 / 0.283	147.43	28.77 / 0.903 / 0.256
average	5.80	27.23 / 0.885 / 0.287	113.75	29.34 / 0.905 / 0.256

Table B.20: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.005$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives.

3D-MCMC with RadSplat pruning with $\lambda = 0.005$				
	Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
bicycle	40.76	24.12 / 0.758 / 0.211	721.45	25.69 / 0.797 / 0.174
bonsai	8.36	28.15 / 0.902 / 0.244	156.07	32.52 / 0.946 / 0.196
counter	9.00	26.59 / 0.868 / 0.239	166.53	29.27 / 0.915 / 0.190
flowers	35.28	21.59 / 0.610 / 0.320	587.23	22.06 / 0.643 / 0.290
garden	48.07	25.67 / 0.831 / 0.146	920.57	27.75 / 0.876 / 0.096
kitchen	13.27	28.24 / 0.891 / 0.170	269.49	31.99 / 0.932 / 0.123
room	6.47	28.64 / 0.891 / 0.244	128.41	32.14 / 0.928 / 0.205
stump	38.62	26.16 / 0.775 / 0.215	674.04	27.43 / 0.812 / 0.173
treehill	32.65	22.43 / 0.630 / 0.321	546.15	23.04 / 0.660 / 0.277
average	25.83	25.73 / 0.795 / 0.234	463.44	28.10 / 0.834 / 0.192

Table B.21: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.005$.** The entropy models are trained for $10,000$ iterations after optimizing the Gaussian primitives for $30,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives.

3D-MCMC with RadSplat pruning with $\lambda = 0.001$				
	Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
train	9.71	22.72 / 0.827 / 0.207	127.14	22.82 / 0.838 / 0.192
truck	17.76	26.03 / 0.889 / 0.130	228.01	26.40 / 0.898 / 0.119
average	13.74	24.38 / 0.858 / 0.169	177.58	24.61 / 0.868 / 0.156

Table B.22: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives.

3D-MCMC with RadSplat pruning with $\lambda = 0.001$				
	Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
playroom	5.81	28.15 / 0.899 / 0.269	80.06	29.90 / 0.906 / 0.255
drjohnson	9.57	28.61 / 0.900 / 0.264	147.43	28.77 / 0.903 / 0.256
average	7.69	28.38 / 0.900 / 0.267	113.75	29.34 / 0.905 / 0.256

Table B.23: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives.

3D-MCMC with RadSplat pruning with $\lambda = 0.001$				
	Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
bicycle	59.25	25.35 / 0.787 / 0.186	721.45	25.69 / 0.797 / 0.174
bonsai	12.14	30.86 / 0.931 / 0.212	156.07	32.52 / 0.946 / 0.196
counter	13.76	28.54 / 0.901 / 0.205	166.53	29.27 / 0.915 / 0.190
flowers	51.90	21.96 / 0.635 / 0.300	587.23	22.06 / 0.643 / 0.290
garden	72.76	27.19 / 0.862 / 0.114	920.57	27.75 / 0.876 / 0.096
kitchen	20.28	30.61 / 0.920 / 0.139	269.49	31.99 / 0.932 / 0.123
room	9.21	30.94 / 0.917 / 0.216	128.41	32.14 / 0.928 / 0.205
stump	58.51	27.11 / 0.801 / 0.188	674.04	27.43 / 0.812 / 0.173
treehill	48.25	22.93 / 0.651 / 0.295	546.15	23.04 / 0.660 / 0.277
average	38.45	27.28 / 0.824 / 0.206	463.44	28.10 / 0.834 / 0.192

Table B.24: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives.

3D-MCMC with RadSplat pruning with $\lambda = 0.0001$				
	Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
train	14.64	22.99 / 0.835 / 0.198	127.14	22.82 / 0.838 / 0.192
truck	28.05	26.46 / 0.897 / 0.120	228.01	26.40 / 0.898 / 0.119
average	21.35	24.73 / 0.866 / 0.159	177.58	24.61 / 0.868 / 0.156

Table B.25: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on Tanks & Temples [19] scenes for $\lambda = 0.0001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives.

3D-MCMC with RadSplat pruning with $\lambda = 0.0001$				
	Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
playroom	9.10	28.69 / 0.903 / 0.262	80.06	29.90 / 0.906 / 0.255
drjohnson	13.63	28.94 / 0.903 / 0.258	147.43	28.77 / 0.903 / 0.256
average	11.37	28.82 / 0.903 / 0.260	113.75	29.34 / 0.905 / 0.256

Table B.26: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on DeepBlending [12] scenes for $\lambda = 0.0001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives.

3D-MCMC with RadSplat pruning with $\lambda = 0.0001$				
	Mean-Scale Hyperprior		Uncompressed	
Scene	Size (MB)	PSNR / SSIM / LPIPS	Size (MB)	PSNR / SSIM / LPIPS
bicycle	92.63	25.79 / 0.797 / 0.173	721.45	25.69 / 0.797 / 0.174
bonsai	20.33	32.49 / 0.945 / 0.198	156.07	32.52 / 0.946 / 0.196
counter	21.31	29.30 / 0.913 / 0.192	166.53	29.27 / 0.915 / 0.190
flowers	78.82	22.12 / 0.643 / 0.291	587.23	22.06 / 0.643 / 0.290
garden	114.49	27.75 / 0.875 / 0.098	920.57	27.75 / 0.876 / 0.096
kitchen	32.44	32.01 / 0.931 / 0.125	269.49	31.99 / 0.932 / 0.123
room	15.11	32.08 / 0.93 / 0.206	128.41	32.14 / 0.928 / 0.205
stump	89.34	27.43 / 0.811 / 0.175	674.04	27.43 / 0.812 / 0.173
treehill	72.77	23.09 / 0.660 / 0.279	546.15	23.04 / 0.660 / 0.277
average	59.69	28.01 / 0.834 / 0.193	463.44	28.10 / 0.834 / 0.192

Table B.27: **Comparison of learned Fully-Factorized [1] and Mean-Scale Hyperprior Entropy Models [25] on MipNeRF 360 [2] scenes for $\lambda = 0.001$.** The entropy models are trained for $5,000$ iterations after optimizing the Gaussian primitives for $25,000$ iterations. The $5,000$ iterations are performed without optimizing position, scaling, and rotation attributes of Gaussian primitives.

Appendix C

Hierarchy Generation Examples on Different Datasets

In this chapter of the Appendix, we depict some of the visuals for hierarchy generation. Hierarchy generation aggregates the Gaussian primitives into intermediate nodes for a more compressed representation with fewer intermediate nodes compared to actual Gaussians. As we go to higher depth levels of the octree hierarchy as explained in Section 3.5, we have fewer Gaussians, resulting in lower rendering quality. In following Figures C.1, C.2, and C.3, we visualize the effect of changing depth level for different scenes.

Figure C.1: **Effect of depth variation on visual quality on *train* scene from Tanks & Temples [19] without compression.** Using different levels of octree results in changing granularity of scenes. Specifically, close points occur more blurry while far away points are less affected from averaging of Gaussian primitives.

Figure C.2: **Effect of depth variation on visual quality on *truck* scene from Tanks & Temples [19].** Using different levels of Octree results in changing granularity on images. Specifically, close points occur more blurry while far away points are less affected from averaging of Gaussian primitives.

Figure C.3: **Effect of depth variation on visual quality on *room* scene from Mip-NeRF 360 [2].** Using different levels of Octree results in changing granularity on images. Specifically, close points occur more blurry while far away points are less affected from averaging of Gaussian primitives.

Bibliography

[1] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior. *arXiv preprint arXiv:1802.01436*, 2018.

[2] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields, 2022.

[3] Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. Zip-nerf: Anti-aliased grid-based neural radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 19697–19705, 2023.

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

[5] Jean Bégaint, Fabien Racapé, Simon Feltman, and Akshay Pushparaja. Compressai: a pytorch library and evaluation platform for end-to-end compression research, 2020.

[6] Yihang Chen, Qianyi Wu, Jianfei Cai, Mehrtash Harandi, and Weiyao Lin. Hac: Hash-grid assisted context for 3d gaussian splatting compression. *arXiv preprint arXiv:2403.14530*, 2024.

[7] Wikimedia Commons. Octree, 2010.

[8] Ze Cui, Jing Wang, Shangyin Gao, Tiansheng Guo, Yihui Feng, and Bo Bai. Asymmetric gained deep image compression with continuous rate adaptation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10532–10541, 2021.

[9] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. Technical Report RFC 1951, Request for Comments, 1996.

[10] Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, and Zhangyang Wang. Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps. *arXiv preprint arXiv:2311.17245*, 2023.

[11] Ben Fei, Jingyi Xu, Rui Zhang, Qingyuan Zhou, Weidong Yang, and Ying He. 3d gaussian as a new vision era: A survey. *arXiv preprint arXiv:2402.07181*, 2024.

[12] 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.

[13] Sonain Jamil, Md Jalil Piran, MuhibUr Rahman, and Oh-Jin Kwon. Learning-driven lossy image compression: A comprehensive survey. *Engineering Applications of Artificial Intelligence*, 123:106361, 2023.

[14] James T Kajiya and Brian P Von Herzen. Ray tracing volume densities. *ACM SIGGRAPH computer graphics*, 18(3):165–174, 1984.

[15] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4):1–14, 2023.

[16] 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*, 44(3), 2024.

[17] 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, 2024.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[19] 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.

[20] Joo Chan Lee, Daniel Rho, Xiangyu Sun, Jong Hwan Ko, and Eunbyung Park. Compact 3d gaussian representation for radiance field. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 21719–21728, 2024.

[21] Tao Lu, Mulin Yu, Linning Xu, Yuanbo Xiangli, Limin Wang, Dahua Lin, , and Bo Dai. Scaffold-gs: Structured 3d gaussians for view-adaptive rendering. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

[22] K. Mammou. PCC test model category 3 v0. Technical Report N17249, ISO/IEC JTC1/SC29/WG11 MPEG, Macau, Oct. 2017.

[23] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[24] 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.

[25] David Minnen, Johannes Ballé, and George Toderici. Joint autoregressive and hierarchical priors for learned image compression, 2018.

[26] Simon Niedermayr, Josef Stumpfegger, and Rüdiger Westermann. Compressed 3d gaussian splatting for accelerated novel view synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10349–10358, 2024.

[27] Michael Niemeyer, Fabian Manhardt, Marie-Julie Rakotosaona, Michael Oechsle, Daniel Duckworth, Rama Gosula, Keisuke Tateno, John Bates, Dominik Kaeser, and Federico Tombari. Radsplat: Radiance field-informed gaussian splatting for robust real-time rendering with 900+ fps. *arXiv preprint arXiv:2403.13806*, 2024.

[28] 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.

[29] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[30] 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.

[31] Iain E Richardson. *The H. 264 advanced video compression standard*. John Wiley & Sons, 2011.

[32] Johannes Lutz Schönberger, True Price, Torsten Sattler, Jan-Michael Frahm, and Marc Pollefeys. A vote-and-verify strategy for fast spatial verification in image retrieval. In *Asian Conference on Computer Vision (ACCV)*, 2016.

[33] Seungjoo Shin and Jaesik Park. Binary radiance fields, 2023.

[34] Shimon Ullman. The interpretation of structure from motion. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 203(1153):405–426, 1979.

[35] Yufei Wang, Zhihao Li, Lanqing Guo, Wenhan Yang, Alex C. Kot, and Bihan Wen. Contextgs: Compact 3d gaussian splatting with anchor level context model, 2024.

[36] Lee Alan Westover. *Splatting: a parallel, feed-forward volume rendering algorithm*. The University of North Carolina at Chapel Hill, 1991.

[37] Tong Wu, Yu-Jie Yuan, Ling-Xiao Zhang, Jie Yang, Yan-Pei Cao, Ling-Qi Yan, and Lin Gao. Recent advances in 3d gaussian splatting. *arXiv preprint arXiv:2403.11134*, 2024.

[38] Neil Zeghidour, Alejandro Luebs, Ahmed Omran, Jan Skoglund, and Marco Tagliasacchi. Soundstream: An end-to-end neural audio codec. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 30:495–507, 2021.

[39] Jiahui Zhang, Fangneng Zhan, Muyu Xu, Shijian Lu, and Eric Xing. Fregs: 3d gaussian splatting with progressive frequency regularization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 21424–21433, 2024.

[40] 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.