Degree Project in Technology

Second cycle, 30 credits

Exploring Inefficiencies in Implementations Utilizing GPUs for Novel View Synthesis of Dynamic Scenes

Limitations of Modern Computer Vision Models and Possible Enhancements

LUKAS S. BEINLICH

Exploring Inefficiencies in Implementations Utilizing GPUs for Novel View Synthesis of Dynamic Scenes

Limitations of Modern Computer Vision Models and Possible Enhancements

LUKAS S. BEINLICH

Date: March 25, 2025

Supervisors: Mårten Björkman, Marcel Büsching, Simon Matern
Examiner: Patrick Jensfelt
School of Electrical Engineering and Computer Science
Swedish title: Utforskning av ineffektivitet i GPU-baserade implementationer för syntes av tidigare osedda vyer i dynamiska scener
Swedish subtitle: Begränsningar i Moderna Datorseendemodeller och Möjliga Förbättringar

Abstract

This thesis investigates the computational inefficiencies in existing machine learning models for novel view synthesis, which is the task of generating images of observed scenes from new view points. Modern models are analyzed, and three models are selected for a detailed examination of their implementation. The goal is to identify factors that limit the efficiency of these models during both inference and training phases and to optimize them. Inefficiencies can arise from poor implementations or suboptimal resource usage, especially when memory is not properly reused across training iterations or when hardware, particularly Graphics Processing Units (GPUs), are not fully utilized.

The thesis addresses the question: What are the limiting factors in current implementations of dynamic scene novel view synthesis, and how can they be mitigated? While many studies present unoptimized models to demonstrate capabilities, this research focuses on improving computational efficiency without altering the underlying model architecture, which would require extensive retraining and benchmarking—beyond the scope of this project.

This problem was addressed by utilizing tools such as the PyTorch profiler to measure the time spent in various functions, helping to identify performance bottlenecks. Additionally, custom kernels were analyzed using the NVIDIA Nsight suite to uncover inefficiencies in their execution. These insights allowed for targeted optimizations that significantly improved runtime performance.

The findings indicate substantial improvements when tensor operations, typically written in PyTorch, are translated into custom CUDA kernels, yielding up to an 80% reduction in runtime. However, implementing a backward function for integration with PyTorch's automatic differentiation engine presents a challenge. Additionally, the optimization of a specific CUDA kernel resulted in a 75% reduction in its runtime, translating into a nearly 20% reduction in total training time for the model. These results highlight that even modest efforts to optimize existing models can yield significant improvements, underscoring the importance of GPU programming knowledge for developers aiming to build more efficient machine learning systems.

Keywords

Novel View Synthesis, Machine Learning optimization, GPU efficiency, Custom CUDA kernels, Performance profiling

Sammanfattning

Denna avhandling undersöker de beräkningsmässiga ineffektiviteteterna i befintliga maskininlärningsmodeller för ny vy-syntes, vilket är uppgiften att generera bilder av observerade scener från nya vyer. Moderna modeller analyseras, och tre modeller väljs ut för en detaljerad granskning av deras implementation. Målet är att identifiera faktorer som begränsar effektiviteten hos dessa modeller under både inferens- och träningsfaser och att optimera dem. Ineffektivitet kan uppstå från bristfälliga implementationer eller suboptimal resursanvändning, särskilt när minne inte återanvänds på ett effektivt sätt mellan träningsiterationer eller när hårdvaran, särskilt grafikkort (GPU:er), inte utnyttjas fullt ut.

Uppsatsen behandlar frågan: Vilka är de begränsande faktorerna i nuvarande implementeringar av ny vy-syntes för dynamiska scener, och hur kan de åtgärdas? Även om många studier presenterar ooptimerade modeller för att demonstrera kapabiliteter, fokuserar denna forskning på att förbättra beräkningsmässig effektivitet utan att ändra den underliggande modellarkitekturen, vilket skulle kräva omfattande omträning och benchmarking—utöver detta projekts omfattning.

Problemet adresserades genom att använda verktyg som PyTorch profiler för att mäta den tid som spenderades i olika funktioner, vilket hjälpte till att identifiera prestandaflaskhalsar. Dessutom analyserades anpassade kärnor med hjälp av NVIDIA Nsight-sviten för att avslöja ineffektivitet i deras körning. Dessa insikter gjorde det möjligt att genomföra riktade optimeringar som avsevärt förbättrade körtidsresultaten.

Resultaten visar på betydande förbättringar när tensoroperationer, vanligtvis skrivna i PyTorch, översätts till anpassade CUDA-kärnor, vilket ger en upp till 80% minskning av körtid. Att implementera en bakåt funktion för integration med PyTorchs automatiska differentiationsmotor är dock en utmaning. Dessutom resulterade optimeringen av en specifik CUDA-kärna i en 75% minskning av dess körtid, vilket resulterade i en nästan 20% minskning av den totala träningstiden för modellen. Dessa resultat belyser att även blygsamma ansträngningar för att optimera befintliga modeller kan ge betydande förbättringar, vilket understryker vikten av GPU-programmeringskunskaper för utvecklare som vill bygga mer effektiva maskininlärningssystem.

Nyckelord

Ny synsättsynthes, Maskininlärningsoptimering, GPU-effektivitet, Anpassade CUDA-kärnor, Prestandaprofilering

Zusammenfassung

Diese Masterarbeit untersucht die rechnerischen Ineffizienzen in bestehenden maschinellen Lernmodellen für die Synthese neuer Ansichten, also die Aufgabe, Bilder beobachteter Szenen aus neuen Blickwinkeln zu generieren. Moderne Modelle werden analysiert, und drei Modelle werden für eine detaillierte Untersuchung ihrer Implementierung ausgewählt. Das Ziel ist es, Faktoren zu identifizieren, die die Effizienz dieser Modelle sowohl in der Inferenz- als auch in der Trainingsphase begrenzen, und diese zu optimieren. Ineffizienzen können durch schlechte Implementierungen oder eine suboptimale Ressourcennutzung entstehen, insbesondere wenn Speicher nicht effizient über Trainingsiterationen hinweg wiederverwendet wird oder wenn die Hardware, insbesondere Grafikkarten (GPUs), nicht vollständig ausgelastet sind.

Die Arbeit behandelt die Frage: Was sind die begrenzenden Faktoren in aktuellen Implementierungen der Synthese neuer Ansichten dynamischer Szenen und wie können sie gemildert werden? Während viele Studien unoptimierte Modelle präsentieren, um deren Fähigkeiten zu demonstrieren, konzentriert sich diese Arbeit auf die Verbesserung der Recheneffizienz ohne die zugrunde liegende Modellarchitektur zu verändern, da dies ein umfangreiches Retraining und Benchmarking erfordern würde, was über den Rahmen dieses Projekts hinausgeht.

Das Problem wurde durch den Einsatz von Werkzeugen wie dem PyTorch Profiler angegangen, um die Zeit, die in verschiedenen Funktionen verbracht wird, zu messen und Engpässe in der Leistung zu identifizieren. Zusätzlich wurden benutzerdefinierte Kernel mit der NVIDIA Nsight Suite analysiert, um Ineffizienzen in ihrer Ausführung aufzudecken. Diese Erkenntnisse ermöglichten gezielte Optimierungen, die die Laufzeitleistung erheblich verbesserten.

Die Ergebnisse zeigen erhebliche Verbesserungen, wenn Tensoroperationen, die üblicherweise in PyTorch geschrieben werden, in benutzerdefinierte CUDA-Kernel übersetzt werden, was zu einer Reduzierung der Laufzeit um bis zu 80% führt. Die Implementierung einer Rückwärtsfunktion für die Integration mit PyTorchs automatischer Differenzierungs-Engine stellt jedoch eine Herausforderung dar. Darüber hinaus führte die Optimierung eines spezifischen CUDA-Kernels zu einer Reduzierung der Laufzeit um 75%, was zu einer fast 20%-igen Reduzierung der gesamten Trainingszeit des Modells führte. Diese Ergebnisse verdeutlichen, dass selbst bescheidene Anstrengungen zur Optimierung bestehender Modelle erhebliche Verbesserungen bringen

können und unterstreichen die Bedeutung von GPU-Programmierkenntnissen für Entwickler, die effizientere maschinelle Lernsysteme entwickeln möchten.

Schlüsselwörter

Synthese neuartiger Ansichten, Optimierung des maschinellen Lernens, GPU-Effizienz, benutzerdefinierte CUDA-Kernel, Performanzanalyse

Acknowledgments

I would like to express my gratitude to my supervisors at KTH, Mårten Björkman and Marcel Büsching, for their guidance and support throughout this thesis.

I am also deeply thankful to my mother, Gabriele Beinlich, for her unwavering support and for enduring countless sleepless nights proofreading and checking my work.

Berlin, Germany, March 2025
Lukas S. Beinlich

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

3DGS	3D Gaussian Splatting
4DGS	4D Gaussian Splatting
AI	artificial intelligence
API	application programming interface
CPU	central processing unit
DynIBaR	Neural Dynamic Image-based Rendering
FPS	frames per second
GAN	generative adversarial network
GB	gigabyte
GPU	graphics processing unit
KB	kilobyte
MAD	multiply-add
MB	megabyte
MLP	multilayer perceptron
NeRF	Neural Radiance Fields
NVS	novel view synthesis
RAM	random-access memory
SH	spherical harmonics
SM	Streaming Multiprocessor
TPU	tensor processing unit
VRAM	video RAM

List of Symbols Used

The following symbols are commonly used within the body of the thesis.

α	Accumulated transparency along a ray during volumetric rendering.
$\Delta\mathcal{G}$	The deformation applied to a Gaussian at a given time-step.
γ	Positional encoding function that maps inputs to a higher-dimensional space.
$\hat{\mathbf{C}}(\mathbf{r})$	The rendered color for a ray $\mathbf{r}$ using discrete volumetric rendering.
$\mathbf{c}$	The predicted color at a point $\mathbf{x}$.
$\mathbf{C}(\mathbf{r})$	The expected color for a ray $\mathbf{r}$.
$\mathbf{d}$	The viewing direction, defined by angles (θ, ϕ) or as a 3D unit vector.
$\mathbf{r}(t)$	A ray parameterized by origin $\mathbf{o}$ and direction $\mathbf{d}$.
$\mathbf{x}$	A 3D point in space.
$\mathcal{F}$	A radiance field mapping $(\mathbf{x}, \mathbf{d})$ to $(\mathbf{c}, \sigma)$.
$\mathcal{L}$	The objective function in a machine learning model.
$\mathcal{X}$	Position of a Gaussian in space.
Σ	Covariance matrix of a Gaussian, defining its shape and orientation.
σ	The density of a point or a Gaussian.
Σ'	Covariance matrix of the Gaussian projected onto the image plane.
f	Features used as input to an multilayer perceptron (MLP).
$G(\boldsymbol{x})$	The Gaussian distribution function at a point $\boldsymbol{x}$.

J	Jacobian of the affine projection matrix.
R	Rotationmatrix of a Gaussian, typically stored as a unit quaternion.
S	Scalematrix of a Gaussian, stored as a 3D vector.
W	Viewing transformation matrix.
T	Transmittance of a ray between two points.

Chapter 1

Introduction

This chapter introduces the general problem that this thesis addresses. First of all, a short background of the context of novel view synthesis (NVS) will be outlined. Moreover, a description of the problem the thesis addresses follows, as well as related work on the problem. The purpose of this thesis, as well as the scope and methodology, will then be described, too. Finally, the structure of the thesis will be outlined.

1.1 Background

The task of generating novel views from a sparse set of input images is a core problem in computer vision and computer graphics.

Applications of NVS span various fields. In medical imaging, NVS can be used to reduce the number of required images, which in turn reduces the amount of radiation a patient receives and also enables freely viewable visualizations of medical points of interest [7, 8]. In interactive media, such as movies or games, NVS allows viewers to freely navigate a scene on their own instead of being bound to a given viewpoint. This could be interesting for virtual reality environments, where a user takes a few pictures of an object, which can then be seamlessly inserted into the virtual environment as a full 3D object. This would help create realistic experiences, especially when sharing the environment with other people.

Modern work is based on various methods. The first optimizes a radiance field from which an image can be recovered using volumetric rendering, as outlined in Neural Radiance Fields (NeRF) [38]. A different method is to model a scene using primitives, such as Gaussians, which can be

projected onto a 2D plane, constructing an image, as described in 3D Gaussian Splatting (3DGS) [25]. Another approach is to use image-based rendering, where information from the source images is aggregated and combined to form a novel view, as described in Neural Dynamic Image-based Rendering (DynIBaR) [30].

Extending NVS models to dynamic scenes is a difficult task, as the additional dimension vastly increases the complexity of the task. Oftentimes, existing models are simply extended by mapping the dynamic scene to a static scene, which is then referenced for rendering. This can be seen in various publications based on the NeRF model [28, 46], and also for models based on 3DGS [35, 59].

1.2 Problem

To achieve high performance of learning-based methods, accelerators like graphics processing units (GPUs) are often used, but also more specialized hardware like tensor processing units (TPUs) are used. A multitude of interfaces exist to program these accelerators, leveraging their highly parallelized processing capabilities optimized for linear algebra to accelerate training massively.

However, popular frameworks like PyTorch [1] highly abstract the usage of such accelerators, simplifying it to a single function call to transfer the model and data to the memory of such a device. This enables execution on these devices, leveraging them without requiring any knowledge of best practices to code efficiently on these devices. On the other hand, the push to utilize models that can render novel views in real time for static and dynamic scenes requires a highly efficient approach and implementation. This is unlikely to be done when using frameworks like PyTorch to implement complex data structures instead of developing these in custom C++ CUDA kernels. Therefore, this project aims to analyze three-state-of-the-art models, HexPlane [9], a simple but fast model utilizing six planes to store a dynamic scene, DynIBaR [30], a very complex and computationally intensive model based on image-based rendering, that produces very high-quality results, and 4D Gaussian Splatting [59], a model utilizing Gaussian Splatting extending a similar model for static scenes. The aim is to unveil inefficiencies in their implementations while attempting to improve them, thus answering the question:

Research Question *What are the limiting factors in current implementations of novel view synthesis for dynamic scenes, and how can they be mitigated?*

1.3 Related Work

Most machine learning models utilize CUDA kernels for their implementations, either directly or through frameworks like PyTorch. Past research has shown that combining kernels together improves speed significantly, showing a speedup of over 50x in compute patterns often used in machine learning models [2]. Another attempt at improving performance of kernels is through better scheduling of kernels utilizing different cores on the GPU and fusing kernels together utilizing multiple types of cores at the same time [65].

For NVS various attempts have already been made to implement highly efficient models. One such work was made by researchers from NVIDIA, which focused on building a model that utilizes NVIDIA GPUs to their full extent, capable of rendering novel views of a scene in seconds [39]. Their code is specifically crafted for the GPU in use, ensuring high performance.

Other work on static scenes is also done, when scientists optimize an already published model, as seen in [56]. There, the authors revisited their model and optimized it by writing custom kernels to better utilize the GPU, achieving a 2-3x speed increase.

However, only very limited work has been done on dynamic scenes, as models in this domain often extend existing models. A good example for this is 4D Gaussian Splatting (4DGS) [59]. There, the Gaussians are first transformed to fit the given time of the dynamic scene. Subsequently, the renderer from the static model from 3DGS is used to render a novel view. The challenge here is that the new transformation code might be inefficient, and the model also inherits inefficiencies from the underlying static model.

1.4 Purpose

The purpose of this thesis is to explore and improve implementations of modern artificial intelligence (AI) models. Specifically, the focus will be on reducing the computational power required to train, deploy, and maintain these models. This will make AI technology more available to the public, lowering the barriers of time and cost associated with their use.

Furthermore, increasing the efficiency of AI models will also decrease their energy consumption, addressing global challenges such as sustainability and climate change. As AI technology becomes an increasingly significant contributor to global energy demands, such optimizations are important to limit the environmental impact of such technology.

However, making this technology easily available also comes with risks, as it can also be misused. This can already be seen today in deepfakes, depicting persons in unwanted or even incriminating situations. Therefore, special care has to be taken when improving these models, although, this is ultimately the responsibility of lawmakers regulating the use of AI.

1.5 Scope and Research Methodology

In this work, three state-of-the-art implementations provided by the authors in the field of novel view generation for dynamic scenes will be analyzed and benchmarked. For this, profilers, which measure the time various components of the model take, will be used. After identifying potential bottlenecks, an attempt will be made to understand the limitations of these bottlenecks. If possible, solutions to these will be provided and implemented, while making sure the results are consistent to those of the original implementation. A further comparison of the developed improvements will be made to the original implementation, showing how researchers can benefit from investing time in developing efficient models. However, changes to the model architectures, such as reducing the neural network size to improve the speed, will not be explored, as this would require a quality evaluation.

1.6 Structure of the Thesis

In this project, the development of various approaches to novel view synthesis will be discussed in chapter 2, introducing the necessary background knowledge of the topic. The evolution of models for static scenes will be shown, outlining the strengths and weaknesses in the quality of early models. An extension to dynamic scenes is presented, too. There it becomes obvious that models often employ approaches from static models, featuring additional components to model the time dependence.

In chapter 3, the architecture of modern GPUs is outlined, representing an overview of how cores and memory are organized and used in order to execute a function. There will also be a short section demonstrating best practices regarding memory transfers and how to interface with a GPU.

In chapter 4, the necessary tools to analyze the performance of code will be outlined. Aspects like simple profiling on the central processing unit (CPU) side, as well as a more complex timing of code on multiple processing units, are of importance. Also, a visualization of executions of models in the form

of traces is shown.

Another important tool is the NVIDIA Nsight suite, which provides various programs to analyze individual kernels. A short introduction to analyzing memory transfers will also be given in chapter 4.

In chapter 6 three selected models will be described in detail, their implementations analyzed and if possible improvements made. There will also be a comparison of the performance between the improved and original models.

Finally, chapter 7 summarizes the conducted thesis.

Chapter 2

Novel View Synthesis

NVS is the task of creating images of an observed scene from novel view points. Usually, the given source images are only available from a few sparse view points for a select scene. These images are then used to create a representation from which new images from unseen perspectives are generated.

Traditionally, this problem has been approached geometrically, estimating the relationship between points in two images [34]. Over time, methods combined geometric priors with image-based rendering (IBR) to recover 3D structures or modeled scenes using 4D implicit functions representing radiance fields in free space [12, 19, 26]. Neural networks, like generative adversarial networks (GANs), later enabled NVS by generating 3D shapes in voxel grids or learning dense volumetric scene representations for reconstructing novel views [18, 33, 60].

Nowadays, methods often rely on machine learning based methods in order to generate an internal representation of the scene. This is done either through feed forward neural networks such as multilayer perceptrons (MLPs) [3, 38], through data structures, such as voxels [14, 56], or through collections of primitives modelling the scene, such as points or Gaussians [25, 61]. These internal representations are then optimized to fit the given images and are used to produce novel views.

This chapter introduces current methods of novel view generation. Firstly, static scenes and their various representations will be described. Secondly, the challenges posed in dynamic environments will be discussed, where the addition of a temporal dimension will add further complexity.

2.1 Mathematical Background

There are two major ways of generating novel views. The first is through volumetric rendering, which uses a radiance field to produce an image. This radiance field can be modeled as an MLP, where the inputs are often encoded harmonically, to ensure a better representation of high frequency variation [47].

The other major method to produce novel views is splatting, where primitives are simply transformed and projected to an image plane, then blended together, which also produces an image.

2.1.1 Volumetric Rendering

A radiance field of color $\mathbf{c}$ and density σ is usually modeled as a 5D vector-valued function:

$$\mathcal{F} : (\mathbf{x}, \mathbf{d}) \rightarrow (\mathbf{c}, \sigma) \tag{2.1}$$

The input to this function is a position $\mathbf{x}$ and a viewing direction $\mathbf{d}$, defined by the angles $\mathbf{d} = (\theta, \phi)$. In practice, the viewing direction is often expressed as a 3D unit vector.

To synthesize an image, a ray is cast through each pixel of the desired view. The process of rendering the color of a ray $\mathbf{C}$ is achieved through volume rendering [24]. The expected color $\mathbf{C}(\mathbf{r})$ for a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ originating at $\mathbf{o}$ and traveling in direction $\mathbf{d}$ is given by:

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt \text{ , where } T(t) = \exp\Big(- \int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\Big) \tag{2.2}$$

Here t_n and t_f describe the near and far limits of the ray, and $T(t)$ corresponds to the transmittance from t_n to t, i.e. the fraction of light that travels through space without being absorbed.

In practice, rather than evaluating the continuous integral, color $\mathbf{c}_i$ and density σ_i are sampled at N discrete points along the ray. There are various sampling strategies, ranging from uniform sampling in NeRF [38] up to the use of a learned predictor in order to determine at which points to query for $\mathbf{c}_i$

and σ [40]. This results in the discrete volumetric rendering equation [37]:

$$\hat{\mathbf{C}}(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i\delta_i))\mathbf{c}_i, \text{where } T_i = \exp(-\sum_{j=1}^{i-1}\sigma_j\delta_j) \tag{2.3}$$

Here, $\delta_i = t_{i+1} - t_i$ is the distance between consecutive samples. This equation is easily differentiable, enabling machine learning methods to be applied through gradient descent.

2.1.2 Positional Encodings

When an MLP is used to model a radiance field, the inputs are often encoded because neural networks usually struggle to represent high-frequency variations [47]. To overcome this, the inputs are usually mapped from $\mathcal{R}$ to a higher dimensional space $\mathcal{R}^{2L}$, before passed to the network. Usually, sine and cosine functions are used for this mapping, with varying frequency factors from 1 up to 2^{L-1}. The encoding γ could then be described as, with the input p normalized to $p \in [-1, 1]$:

$$\gamma(p) = (sin(2^0\pi p), cos(2^0\pi p), ..., sin(2^{L-1}\pi p), cos(2^{L-1}\pi p)) \tag{2.4}$$

For the MLP used to represent a radiance field, usually both input, the position **x** and the viewing direction **d** is encoded.

2.1.3 Gaussian Splatting

When the scene is modeled using primitives, they usually have a position, color and some form of translucency, either modeled as density or transparency.

In Gaussian Splatting methods [25, 35, 59, 62] each Gaussian $\mathcal{G} = \{\mathcal{X}, R, S, \sigma, \mathcal{C}\}$ is described by several attributes, such as position $\mathcal{X}$, scale S and rotation R. They define the Gaussian in space. They also have a density σ and color coefficients $\mathcal{C}$ for the spherical harmonics, which are used to generate a view-dependent color. A single Gaussian in 3D space is centered around a center point $\mathcal{X}$ with a covariance Σ.

The distribution of a single Gaussian can then be modeled as the following:

$$G(\boldsymbol{x}) = e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\Sigma(\boldsymbol{x}-\mathcal{X})} \tag{2.5}$$

To ease computation, the covariance matrix is split into a rotation, stored as a unit quaternion and a scale, stored as a 3D vector, which can be combined to

form the covariance matrix Σ as follows:

$$\Sigma = RSS^{-1}R^{-1} \tag{2.6}$$

A Gaussian can then be projected to the image plane by applying the viewing transformation W as well as the Jacobian of the affine projection matrix J:

$$\Sigma' = JW\Sigma W^{-1}J^{-1} \tag{2.7}$$

As this matrix now represents the Gaussian on a 2D plane, only the upper left 2×2 matrix is needed, when calculating gradients for optimization in the backward pass.

All Gaussian are blended together by being rendered from front to back, i.e. the Gaussians closest to the image plane are splatted first using their respective transparency α_i, reducing the remaining transmittance T_i:

$$\alpha_i = (1 - \exp(-\sigma_i \delta_i)) \tag{2.8}$$

$$T_i = \prod_{j=1}^{i-1}(1 - \alpha_j) \tag{2.9}$$

The final color $\hat{\mathbf{C}}$ can then be calculated using the following equation:

$$\hat{\mathbf{C}} = \sum_{i \in N} \mathbf{c}_i \alpha_i \prod_{j=1}^{i-1}(1 - \alpha_j) \tag{2.10}$$

When the remaining transmittance falls below a threshold the splatting for the given pixel stops. Alternatively, if all Gaussians are rendered, the remaining transmittance is multiplied by the background color and added to the image. This process is also differentiable, which enables the application of gradient descent for machine learning.

2.1.4 Spherical Harmonics

Another difficult aspect of NVS is the use of view-dependent color. To realize this, color is often represented by spherical harmonics (SH) coefficients. SH are a set of functions that form an orthogonal basis of degree l_{max} over a sphere. Low degree harmonics can then represent smooth transitions in color over the sphere, to represent Lambertian surfaces [5, 48], while higher frequency

harmonics encode high frequency changes, such as specular highlights [54]. Evaluating these harmonic functions Y_l^m for a given viewing direction $\mathbf{d}$ then sums to the specific color $\mathbf{c}$. For RGB color, each color channel c_i has its own SH coefficients and can be expressed as the following with the given coefficients $\mathbf{k} = (k_l^m)$ with $m \in \{-l, ..., l\}$:

$$c(\mathbf{d}, \mathbf{k}) = S\Big(\sum_{l=0}^{l_{\text{max}}} \sum_{m=-l}^{l} k_l^m Y_l^m(\mathbf{d})\Big) \tag{2.11}$$

S is used to normalize the colors, which could be a sigmoid function $S(x) = (1 + \exp(-x))^{-1}$. For a detailed explanation of SH, refer to the appendix of PlenOctrees [63].

2.2 Static Scenes

NVS for static scenes aims to generate novel views of a static object or scene. The data usually consists of multiple images taken from various viewpoints or even video footage. From the input data, a representation is then created, which is used to generate novel views.

Traditionally, methods focused on generating explicit 3D models, such as triangle meshes, use multiview geometry [21], which triangulates points from multiple views to reconstruct the scene. However, such methods require at least two source images for each triangulated point, a condition that cannot always be met, especially in occluded regions. A frequently used implementation of this approach is COLMAP [52, 53], which first estimates camera poses using structure from motion and then reconstructs the scene using a sparse point cloud, which is then refined to a dense point cloud and possibly converted to a triangular mesh.

Other methods attempt to reconstruct 3D models from either silhouettes [50] or by refining a mesh from simple shapes like ellipsoids into complex 3D models, leveraging neural networks to refine the geometry [58].

Some approaches also rely on depth information which have to be captured through specialized camera equipment. However, modern techniques using machine learning can infer depth maps from 2D images[29, 31, 49] nullifying the need for expensive hardware.

2.2.1 NeRF: Radiance Fields based on MLPs

Modern approaches represent the scene using radiance fields. In 2020 Mildenhall et al. introduced *NeRF* [38], a groundbreaking approach to novel view synthesis. NeRFs represent the radiance field of a scene using a large MLP as described in eq. (2.1). This representation ensures a smooth transition of $\mathbf{c}$ and σ across space and viewing directions, which can then be used to infer the color of a camera ray $\mathbf{r}$ via volumetric rendering (eq. (2.3)). The neural network is queried multiple times along each ray, once for each sampled point to compute the pixel's final color. In this formulation, the scene is implicitly encoded within the parameters of the MLP $\mathcal{F}_\Theta$.

NeRF incorporates several optimizations to improve efficiency and quality. They use a coarse-to-fine sampling strategy, which involves a faster "coarse" neural network, that is used first to query density and therefore filter out queries to the large MLP $\mathcal{F}_\Theta$ in empty space. In addition to that, another optimization is the usage of a positional encoding of the input in

order to increase the network's capacity, and to capture high-frequency details increasing rendering quality.

2.2.2 Extensions of NeRF

NeRF revolutionized the field by producing high-quality results were unprecedented at the time. However, there were limitations, such as quality issues like blurry results when rendering at different resolutions. Super-sampling (casting multiple rays per pixel) could have mitigated this issue, but it significantly increases computational cost, too. The high computational costs present another issue, as rendering a single novel view involves querying the large MLP hundreds of times per ray to produce only a single pixel of an image. This results in training times lasting approximately 1-2 days on an NVIDIA V100 GPU for a single scene.

The simplest way to speed up NeRF was to use more computational resources. *JaxNeRF* [13] is a NeRF reimplementation in Jax [6, 16], a Python library for high-performance computation with support for execution across multiple accelerators. This reimplementation achieved modest speed improvements per used accelerator unit and demonstrated scalability, rendering a single image in 0.35 seconds using 128 TPUv2 units from Google.

While such brute-force solutions highlight the potential for hardware-accelerated optimizations, they do not scale well economically or ecologically. Thus, more sophisticated methods are needed to address NeRF's computational bottlenecks. Several other methods have also been proposed so far to enhance NeRF's efficiency by reducing the number of queries to its large MLP. These strategies generally fall into two categories, better sampling along the ray and precomputing and storing intermediate representations for faster interpolation. An overview of these methods will now be given.

First of all, *DONeRF* [40] replaces NeRF's coarse MLP with a depth oracle prediction network. This network predicts the points along the ray where the MLP should be queried, effectively minimizing evaluations of empty space. By reducing the number of MLP queries to a fraction of the original, DONeRF achieves speed improvements of up to 48x, reaching near real-time performance on a single GPU at 20 frames per second when rendering a fully trained model. This is achieved without increasing the memory footprint, keeping it similar to NeRF.

Another approach to improve speed is to evaluate a NeRF model over a bounded volume and storing the results in a matrix, allowing computationally

efficient trilinear interpolation to recover $\mathbf{c}$ and σ instead of repeated MLP queries during rendering. Storing a full 5D matrix, however, is infeasible due to memory constraints. Some methods split the MLPs into separate networks to manage memory requirements efficiently and to use caching [17].

SNeRG [22] stores a radiance field produced by NeRF in a sparse neural radiance grid. It "bakes" a trained NeRF by evaluating its output across a sparse voxel grid and by storing the results as features for density and view-dependent color. A small MLP then uses these precomputed features in order to recover the final view-dependent color for NVS. To keep model size manageable, this approach uses modern image and video compression techniques when storing the model on disk. SNeRG achieves significant performance gains while incurring a slight drop in rendering quality compared to the original NeRF, effectively trading stored model size for speed.

Similarly, *PlenOctrees* [63] adopt a hierarchical octree structure to store SH instead of raw color and density values. This allows the recovery of view-dependent color from view direction and SH coefficients while reducing computational costs. After training a modified NeRF which outputs SH, the authors of PlenOctrees further optimize the stored PlenOctrees using the source images. This secondary optimization allows the NeRF training to be stopped early, reducing training time, as it is faster to optimize a PlenOctree. By rendering novel views from the prepared PlenOctree, this model achieves extraordinary speed-ups, rendering novel views 3000x faster than the original NeRF implementation, with frame rates exceeding 150 frames per second. However, these benefit comes with a significant model size increase to nearly 2 GB. Additionally, PlenOctrees enable unique features like rendering cross-sections and combining multiple scenes into one.

All these methods work well when rendering at given resolutions. However, when new images are synthesized at resolutions not used during training, the results are often blurry. This can be countered by using supersampling, but that increases the computational resources needed massively. *Mip-NeRF* [3] enables rendering at arbitrary resolutions without blurry artifacts or the need for super sampling. This is achieved by casting cones instead of rays for each pixel. The cone's geometry is modeled as a Gaussian, which is then used to query the NeRF-like MLP for density and color. This approach reduces aliasing artifacts and allows rendering at any resolution while halving the model size and slightly increasing rendering speed. However, Mip-NeRF still requires several seconds to render a single image, thus being far from real-time performance.

While many improvements to NeRF focus on speed for bounded scenes, all these approaches face significant challenges when applied to unbounded environments. Such scenes require specialized representations to handle objects at varying distances as near objects dominate the source images, leaving far objects underrepresented. This imbalance results in slower training and inadequate reconstruction for distant parts of the scene, creating blurry backgrounds or artifacts.

One approach to unbounded scenes is to transform the scene into a bounded representation, allocating more capacity to objects near the camera and less to distant objects. *NeRF++* [64] addresses unbounded scenes by partitioning the scene into two volumes, a foreground represented as a spherical volume and a background represented as an inverted sphere. Objects in the background are then reparameterized into inverted sphere coordinates before being processed by a separate MLP. This approach improves training efficiency and reconstructs details in distant regions more adequately.

Building on Mip-NeRF and NeRF++, *Mip-NeRF 360* [4] enhances results for unbounded scenes by introducing a new ray parametrization in view space, increasing quality, while also requiring less computational resources compared to NeRF++.

With all these methods improving quality, the underlying problem of the need to query a large MLP multiple times per pixel limits the performance of this approach. Some tricks, like separating the density and color outputs into independent MLPs, as proposed in *FastNeRF* [17] or caching the results to provide faster rendering can help improve performance. However, these are only treating the symptoms, while the reliance on large MLPs remains the fundamental limitation.

To achieve real-time performance for both training and rendering, a shift away from MLPs towards hybrid models or explicit models is essential. Hybrid models combine data structures with usually small supportive neural networks to decode the stored data. In contrast, explicit models rely solely on the scene being represented in data structures. The following section now outlines several important advances using those approaches.

2.2.3 Representation through Voxel Grids and other Data Structures

One promising direction of improving the speed of NVS is to replace the implicit encoding of density and color in an MLP with a voxel grid

representation instead of using an MLP. These hybrid and explicit methods either store density and color directly in the grid or adopt a hybrid approach where intermediate features are stored in the grid, and a tiny MLP, often only consisting of two layers, is used to extract the final density and color. In this section several hybrid methods will be presented.

First, in *NSVF* [32] the authors propose a sparse voxel grid to represent the scene. Instead of directly storing the scene in an MLP, features are stored in the grid focusing on occupied regions of space. These features are trilinearly interpolated and fed into an MLP which computes color and density being used in volumetric rendering (eq. (2.3)). This grid structure enables the model to skip empty space and significantly reduces the number of MLP queries. Despite a similar-sized MLP as NeRF's, NSVF achieves a 10x-20x speedup while maintaining comparable or a slightly better quality. The usage of a supporting data structure, namely the voxel grid storing features, classifies this method as a hybrid method, combining explicit methods with implicit methods.

Plenoxels [51], builds on the ideas of PlenOctrees [63] while eliminating the need for a trained NeRF. Instead, a framework for training a model from scratch is being provided, which dramatically reduces training time. For bounded scenes, Plenoxels store density and spherical harmonic coefficients in a voxel grid which directly represents the scene. For unbounded scenes, the grid is replaced with progressively larger spheres. This approach reduces training time down to 10-30 minutes compared to the days required for NeRF. This is because the reliance on a large MLP is removed.

Another model, proposed in *DVGO* [55, 56], adopts a coarse-to-fine voxel grid in order to represent density and color features, which are decoded using a small MLP to regress the final $\mathbf{c}$ and σ. This approach achieves comparable results to NeRF in a fraction of the training time, but it is currently limited to bounded scenes, unlike methods such as Mip-NeRF360 that handle unbounded environments.

In a follow-up technical report [56], the authors implemented key components as a CUDA kernel, achieving a further 2x-3x speedup compared to their earlier version. This demonstrates the significant gains being achievable by using custom GPU kernels to fully utilize the available hardware.

Another novel approach, NVIDIA's *Instant-NGP* [39], replaces the dense voxel grid with multiple hash tables at varying resolutions accessed by a pseudo random function that maps 3D coordinates to indices. The distinguishing feature of this model is that hash collisions are not handled at

all. Instead, the model is solely relying on the training process to optimize and ensure that only relevant features are encoded. This way, the hash table serves as a memory-efficient representation of the scene.

A key advantage is the hardware-conscious design, showing the expertise of the developers regarding NVIDIA hardware. The hash table sizes are carefully tuned to fit entirely within the L2 cache of the GPU, minimizing memory access latency. Additionally, the model processes all inputs for each resolution level in a batch, avoiding reloading of hash tables and thus further improving efficiency.

Because of these design choices, Instant-NGP achieves exceptional performance, producing results comparable to NeRF after only 15 seconds of training time and reaches the quality of Mip-NeRF within a few minutes. These advances show the importance of optimizing data structures to available hardware combined with efficient memory management, which leads to drastically accelerated training times while maintaining high-quality results.

An alternative to representing a scene using voxel grids is to store the necessary data in point clouds. In *Point-NeRF* [61], the authors propose a model that encodes the scene as a point cloud where each point is described by its position, a feature vector, and a confidence value indicating its likelihood of being near a surface in the scene. To compute $\mathbf{c}$ and σ for a queried point $\mathbf{x}$, first, a k-nearest neighbors search is done to determine the closest points $(\mathbf{x}_1, ..., \mathbf{x}_k)$. Afterward, the features of those points are combined using a weighted average based on distance to the queried point $\mathbf{x}$, while being processed by multiple MLPs to produce the final $\mathbf{c}$ and σ. Applying the volumetric rendering equation 2.3 can then generate novel views.

Training the model involves two steps, the first initializes a point cloud using depth maps and sets an initial confidence value for each point. Depth maps can be created either by using specialized hardware to capture training images or by using techniques to estimate depth through multi-view stereo techniques such as COLMAP [53]. In the second step, the point cloud undergoes refinement through pruning, attribute optimization, and addition or removal of points to improve scene representation. Point-based representations are inherently unbounded and require only short optimization phases rather than the full training of a model from scratch.

All these proposed models have shown how hybrid models using data structures such as voxel grids, sometimes in combination with small MLPs, can benefit computational performance leading to accelerated training and

rendering times. However, their reliance on structured grids imposes limitations, particularly when dealing with unbounded scenes. The following section presents another approach based on point based models.

2.2.4 Representation of a scene using Gaussians

Another approach, namely *3D Gaussian Splatting (3DGS)* [25], builds on a similar principle as Point-NeRF [61] but replaces discrete points with Gaussian primitives. In this approach each Gaussian is represented by its position, covariance, opacity, and SH coefficients for view-independent color description. To render a novel view, the Gaussians are first transformed into canonical view space by applying perspective transformation and then ordered based on distance to the image plane. Then, the Gaussians are blended together using their opacity and color evaluated from the SH coefficients. This technique is called splatting (cf. section 2.1.3).

For this model, the authors implemented a custom CUDA kernel which handles the creation of a new image based on the Gaussians. Part of this implementation is also a complex backward kernel, which propagates the derivatives from the loss function back to the input parameters.

The initialization of the Gaussians can start with point clouds generated through structure-from-motion techniques, such as COLMAP [52]. Initial values can also be chosen completely at random. To optimize the scene representation, the model jointly refines the attributes of each Gaussian—including position, covariance (stored as separate rotation and scale matrices), spherical harmonics, and translucency, which fulfills a similar role as density. To improve coverage of underrepresented regions of the scene, a densification process handles cloning, pruning, and splitting of Gaussians.

This approach achieves remarkable quality for indoor and unbounded outdoor scenes, outperforming state-of-the-art methods like Mip-NeRF360, Instant-NGP, and Plenoxels at the time. Training times are relatively short, around 40 minutes for their high-quality model, while rendering far over 100 frames per second (FPS)—all without the reliance on neural networks.

2.3 Dynamic scenes

As seen in the previous chapter, various methods have been developed to render novel views of static scenes, with the fastest approaches using tiny neural networks, if any at all. However, extending these methods to dynamic scenes is not trivial, as incorporating the time dimension introduces additional complexity. This is especially true when using monocular video, where only a single source image is available at each time step. Each image in such a video features only small changes over time in perspective. Some methods address this by transforming a static scene representation into a dynamic one (or vice versa) [20, 28, 46], while others optimize a compressed discrete 4D space-time continuum and interpolate novel views from the compressed data [9, 15].

On the whole, training a model for dynamic scenes is inherently ill-posed and requires strong assumptions realized through various loss functions to achieve a good representation, especially for monocular video. Additionally, the inclusion of time as a dimension increases the storage demands for the trained models—sometimes substantially. For instance, storing a dynamic scene representation as a dense 4D voxel grid requires significantly more storage than a 3D voxel grid for a static scene, with the additional storage demand depending on the duration and number of frames. This becomes impractical for long duration and large scenes, due to the curse of dimensionality.

Most models for dynamic scenes also inherit their abilities from similar approaches for static scenes, only augmented with specialized mechanisms to account for temporal dependencies. This allows these models to build upon and to benefit from the optimizations made in static scene modeling.

2.3.1 Expanding Implicit Models to Dynamic Scenes

Early efforts to extend NeRF to dynamic scenes focused on adding time as an additional input to the MLP, which regresses color and density from position and viewing direction. This straightforward approach, referred to as T-NeRF [46], serves as a baseline for many subsequent models.

D-NeRF [46] improves on this baseline by dividing the task into two distinct stages. The first stage employs a deformation network that maps scene deformations to a canonical configuration. The second stage is a canonical NeRF model that regresses position and viewing direction to color and density to be used in volumetric rendering eq. (2.3). This decomposition enhances

performance compared to T-NeRF for monocular video.

A related approach to D-NeRF is presented in *NSFF* [28] which adds additional outputs to the first stage for forward and backward scene flow as well as disocclusion weights. These outputs are used for the computation of additional losses that enforce scene flow to be consistent between adjacent time frames and help the model learn the various weights in an unsupervised way, like the disocclusion and blending weights used for blending the output from both a static and a dynamic model. NSFF was in general one of the first to render novel views for monocular dynamic scenes, but struggles with longer scenes or scenes with significant motion, while achieving similar training and rendering times to NeRF.

Similarly, in *Nerfies* [44], a deformation network that maps points from a dynamic scene to a static scene using latent encodings has been introduced. Here, instead of using time as an input, a latent vector modulates the output to handle appearance variations between frames. After transforming a point to static coordinates, a conventional NeRF model is used to render a novel view.

Despite being proposed largely to combine multiple casually captured photos from mobile phones ("selfies") into a single 3D representation, Nerfies is able to interpolate and synthesize novel views for dynamic scenes. By linearly combining latents, Nerfies can interpolate motion between two input images. It struggles, however, when faced with topological changes, such as opening mouths. Another downside is that the model requires extensive training, that is training time reaching up to a week on multiple NVIDIA V100 GPUs when training for Full HD images.

To address the limitation of changes in topology, *HyperNeRF* [43], a follow-up model to Nerfies, splits the deformation network into two separate MLPs. One is used to generate warped coordinates as before. The other is a slicing network that yields an additional coordinate, effectively elevating the canonical representation into a higher-dimensional space. A modified NeRF model then takes both coordinates and viewing direction as an input to regress $\mathbf{c}$ and σ, which in conjunction with volumetric rendering, is used to produce novel views. This representation enables HyperNeRF to capture topological changes, while being able to interpolate between them, by changing only the input to the slicing network. However, the reliance on a NeRF model still results in high computational costs for training and rendering.

DyNeRF) [27] takes a different approach, discarding deformation networks in favor of custom learnable latent encoding as temporal inputs. Here a single MLP regresses color and density from position, viewing direction, and these temporal latents. While the model performs well for moderately dynamic

scenes, it struggles with rapid or large-scale motions and requires multiple static cameras for training. However, its hierarchical training strategy, which focuses on keyframes and ray importance sampling, encouraging training on pixels with high color variance, significantly accelerates convergence. Due to this, training times are reduced by a factor of 10 compared to previously published NeRF-based methods, but it still requires thousands of GPU-hours.

Despite the qualitative improvements achieved by these models, their training and inference times remain high due to the reliance on slow, large MLP. Rendering still requires evaluating millions of rays at multiple points, compounding the computational burden, similar to their static counterparts. One way to increase speed is to replace the MLP with more efficient mechanisms, leading to hybrid models that combine an MLP with an underlying explicit representation, like a voxel grid.

2.3.2 Explicit and Hybrid Approaches to Dynamic Scenes

All the methods introduced in the last chapter, as well as many others, achieve qualitatively decent results for NVS for dynamic scenes. However, their dependence on NeRF results in long training and rendering times, due to the high number of evaluations of the MLP to produce the color of a single pixel. When optimizing on multiple high-resolution images, the process can take days to train a NeRF model on a single scene. To address this bottleneck a different architecture needs to be employed. One promising direction includes hybrid methods applied to dynamic scenes, which in turn are inspired by methods for static scenes.

While explicit methods could store density and color, usually in the form of view-independent SH coefficients directly in a 4D voxel grid, the size requirements quickly exceed the storage capacity of modern computing machines when stored in the necessary resolution. Hybrid methods offer a tradeoff, storing only intermediate features in a lower resolution voxel grid, which are then decoded and refined using small MLPs.

In *TiNeuVox* [14], the authors propose a fast and efficient extension of NeRF that combines a 3D voxel grid with multiple small MLPs. Their approach begins with a deformation network that maps point coordinates and embedded time representations to time-dependent positions, thus eliminating the need to store features in the voxel grid across the time dimension.

These deformed coordinates query voxel features at various resolutions using trilinear interpolation. The resulting features are then combined with the time embedding and the undeformed coordinates and form the input to a radiance network—a small MLP that predicts color and density to be used with volumetric rendering (eq. (2.3)).

A key advantage of this method is the compactness of the MLPs, typically comprising only two to four layers, enabling much faster training speeds while achieving comparable quality to prior methods. At the time of the publication of this model, it delivered state-of-the-art performance for NVS for dynamic scenes. However, it also struggles with scenes involving long-range motions or reflections, where reconstruction quality degrades.

To address the scaling issues of high-dimensional voxel grids, *K-Planes* [15] introduces a factorized representation that breaks down 3D, 3.5D, 4D, or even higher-dimensional spaces into collections of 2D planes. Specifically for dynamic scenes, six 2D planes are used: three for spatial information and three for spatiotemporal variations. Each plane stores features that are interpolated using bilinear sampling based on the relevant coordinates. Features from planes that do not share a dimension are combined using the Hadamard product to increase spatial smoothness and coherence, which are then concatenated from all three plane pairs. These features are then processed using either a learned color basis or a small MLP to predict $\mathbf{c}$ and σ for volumetric rendering (eq. (2.3)).

The authors employ several optimizations, such as a total variation loss that enforces smoothness and sparsity of gradients in a plane. Furthermore, they use a sampling strategy to reduce the number of points needed for volumetric rendering. While K-Planes is neither the fastest nor does it belong to the highest quality methods, it achieves competitive performance. Moreover, relatively low training times for dynamic scenes are required, demonstrating the power of its simple yet efficient design.

A concurrent approach, *HexPlane* [9], similarly factorizes 4D spaces into planes but also uses a learned basis vector when combining features from planes that do not share a dimension. This approach maintains simplicity while achieving results on par with state-of-the-art methods—all with reduced training times. Both K-Planes and HexPlane highlight efficient and memory-conscious ways of handling high-dimensional data not limited to NVS.

An alternative approach to dynamic scenes is to use points to represent a dynamic scene, as described in Point-NeRF [61] for static scenes. *Point-DynRF* [42] extends this model to dynamic scenes with a monocular video

source. Point-DynRF encodes features using points, each defined by a position, time step, feature vector, and a rigidity value, indicating whether the point is part of the static background or dynamic foreground. To render a novel view, a subset of points comprising either the background or matching the current time step has to be selected, then volumetric rendering eq. (2.3) is to be applied. Similarly to in Point-NeRF, a k-nearest-neighbor search is employed on the time-dependent subset of points, which are then combined using MLPs and distance-based weights to predict color and density for that point.

Point-DynRF demonstrates robust results. It especially avoids the production of artifacts in distant spatio-temporal regions, which is a problem in wide-range camera trajectories. Despite optimizations to reduce ray-marching through empty space, rendering times remain high due to the computational overhead of the k-nearest neighbor searches for each point, resulting in high rendering and therefore also in high training times.

Another different approach that does not use an explicit representation is *DynIBaR* [30]. This model adopts an image-based rendering framework rather than voxel grids to store intermediate features. Due to the learned parameters stored in various components, DynIBaR can still be classified as a Hybrid model, utilizing explicit features recovered from the input images with implicit features stored in a motion basis and MLPs in the model.

DynIBaR utilizes information from multiple source images in a neighborhood from the desired time step. To correct for motion, they also utilize a motion trajectory field, using a learned motion basis and coefficients. Motion-corrected details from the source images are then aggregated into a single feature vector for each ray point. An MLP then transforms these features into density and color, enabling volumetric rendering eq. (2.3) to generate novel views.

This model features a static and dynamic component, capturing background and moving objects separately, while a learned mask combines the output of both components. To prevent overfitting on single input images, DynIBaR uses multiple losses, most importantly a loss term that uses cross-time rendering, which renders images based on temporally adjacent frames transformed by the motion trajectory field. Despite requiring days of training on high-end hardware, i.e., eight NVIDIA A100 GPUs, to train reconstruction for a 10-second scene, DynIBaR achieves unparalleled quality, especially for long videos with complex motion, setting a new benchmark for dynamic scene synthesis.

2.3.3 Hash-table based approach

An alternative representation for NVS on dynamic scenes builds upon hash-table-based encoding methods inspired by Instant-NGP [39]. While straightforward extensions to a 4D hash encoding might seem intuitive, this approach either suffers from a high number of hash collisions, not manageable by the optimization, or requires an unfeasibly large model size due to storing data of the added temporal dimension. To address these challenges, *MSTH* [57] introduces two separate hash tables, one time-variant 4D hash encoding for the dynamic parts of the scene and one time-invariant 3D hash encoding for the static part. A learnable mask indicates whether a position is part of the static or dynamic part of the scene and is guided by an uncertainty voxel field.

One advantage of the model is the rendering speed. For example, for rendering videos from a static viewpoint, the static parts of the model need to be computed only once and can then be reused, accelerating rendering. This approach produces high-quality novel views while maintaining a compact model size while only needing 20 minutes of training time per scene. However, the method has limitations. It struggles to reconstruct novel views from monocular video sources due to the absence of auxiliary losses to help train it. This results in artifacts when objects are not fully visible or lack sufficient viewpoint diversity in the input data.

2.3.4 Gaussian Splatting and Point Based Methods for Dynamic Scenes

Building on the success of 3DGS [25], various methods have been proposed to adapt this approach to dynamic scenes.

In *Dynamic 3D Gaussians* [35], the authors extended the position and rotation attributes of the Gaussians to be time-dependent while keeping the other attributes, such as scale, color, and an opacity and background logit, time-invariant. To render an image at a specific time, the model chooses position and rotation according to the input time and proceeds with rendering as in the 3D Gaussian Splatting model, allowing for very fast rendering speeds. To counteract the ill-posed problem of dynamic scene reconstruction, the authors introduce several loss functions. A rigidity loss enforces local rigidity, i.e., neighboring Gaussians maintain the same rotation and position relative to each other at each time step, stopping Gaussians from freely moving through the scene. Another loss, namely a rotational loss, explicitly enforces neighboring Gaussians to have the same relative rotations at every time step,

whereas a long-term isotropy loss maintains consistent distances between nearby Gaussians. The model itself is optimized sequentially, starting with an initialization based solely on the first frame, while only the time-dependent attributes are optimized frame by frame successively.

Although this method achieves impressive speed and quality, it has limitations. As the Gaussians are initialized using only the first frame, this model completely fails to capture objects that are not visible in the first frame. Additionally, the method requires a multi-camera setup and cannot work with monocular video.

Another approach, *4D Gaussian Splatting (4DGS)* [59], opts to transform position, rotation, and scale through small time-dependent MLPs. The inputs for these deformation networks are stored in six multi-resolution 2D planes, similar to HexPlane [9] and K-Planes [15]. To address challenges posed by monocular video sparsity, the authors incorporate a total variational loss to smooth features across these planes in addition to standard L1 color loss. Although the model achieves slightly lower quality than state-of-the-art models like DynIBaR and slightly lower speed than 3D-Gaussian splatting [25] and Dynamic 3D Gaussians [35], it still offers real-time rendering speed at over 30 FPS and short training times of under an hour per scene while still providing good quality without the limitations of Dynamic 3D Gaussians.

A more recent work, namely *Deformable 3D Gaussians* [62], employs a larger MLP as a deformation network. This network estimates the temporal changes in position, rotation, and scale of the Gaussians. Once the Gaussians are transformed, the rendering process is identical to 3DGS. While this model achieves quality comparable to other methods, it struggles with short monocular sequences. The use of a large MLP also slows rendering and training speed compared to approaches with smaller neural networks like 4DGS.

2.4 Summary

This section provided a brief history and background of the various methods for novel view synthesis. Some approaches rely on neural networks to model an implicit function of a scene; others leverage diverse data structures or even leverage the input data directly.

The most complex task—rendering novel views from monocular video of a dynamic scene—requires sophisticated solutions. These range from high-quality but computationally expensive models, that is to say DynIBaR [30], to faster approaches that extend techniques for static scenes [57, 59] or use efficient lower-dimensional scene representations [9, 15].

To explore the efficiency of these diverse approaches and potentially to improve them, three models, HexPlane [9], DynIBaR [30] and 4DGS [59], were selected in this work for further analysis in chapter 6.

These models cover a wide spectrum of approaches, with the aim of showing best practices, uncovering inefficiencies when adapting existing models to dynamic scenes, and exploring opportunities for optimizing complex architectures like DynIBaR.

Chapter 3

GPU Architecture

Historically, GPUs emerged as specialized hardware to alleviate the computational cost of processing 2D and 3D computer graphics on the CPU. Initially, GPUs only took over basic tasks, such as drawing sprites, rendering geometric primitives, and managing the video signal output to the monitor. They operated as coprocessors, offloading specific repetitive tasks to free up the CPU for other computations.

Over time, GPUs became a lot more sophisticated, taking over more and more tasks. Nowadays, they are used in a wide array of tasks and are not just used for computer graphics anymore, instead they have become a cornerstone of high-performance computing, excelling in workloads that benefit from their massive parallelism. Examples include scientific simulation, deep learning, and other applications that require large-scale parallel execution.

While CPUs were developed to be able to interact with large amounts of different components and to handle all kinds of processing, GPUs were mainly developed to handle mathematical operations. However, they are only able to access local memory, so data has to be first transferred to the GPU global memory, called video RAM (VRAM). This enabled them to focus on optimizing those operations, especially the important multiply-add (MAD) operation. MAD operations compute the product of two numbers b and c and add that product to an accumulator a, which is performed in a single clock cycle:

$$a \leftarrow a + (b \cdot c) \tag{3.1}$$

The simplicity of each single core allows GPUs to scale up core counts to massive numbers; thus, modern GPUs often have more than 10,000 processing units. In the last few years, specialized tensor cores and even TPUs have

emerged that support entire matrix MAD operations in a single clock cycle, boosting performance even higher. Due to this specialization, however, these specialized devices have lost their capabilities in rendering computer graphics and are therefore often classified under the broader term accelerator units.

There are two established providers of GPUs that are used in compute tasks, namely NVIDIA and AMD, with the former being the dominant manufacturer. In consequence of this, this chapter generally focuses on the terminology dedicated to NVIDIA's general-purpose compute technology, CUDA.

3.1 Modern GPU-architecture

Today, GPUs are made up of a number of different accelerator units. CUDA Cores are used for general computation for parallel workloads such as rendering, physics simulations, and compute tasks. Tensor Cores are specialized processing units for matrix multiplications, which are often found in machine learning workloads. Ray Tracing Cores are dedicated to ray tracing, a rendering technique that simulates the physical behavior of light. There are other specialized processing units, such as texture mapping units, render output units, and video encode/decode units. However, these specialized units are mainly used in their respective tasks in the graphical rendering pipeline and have limited uses in compute tasks and machine learning.

3.1.1 Core Layout

CUDA Cores are generally grouped into Streaming Multiprocessors (SMs). Here, each SM has its own fast and dedicated memory, called L1 cache, while the GPU core has a much larger L2 cache, accessible to all SMs. The physically separate VRAM is connected to the GPU cores through the L2 cache and is much slower, taking multiple clock cycles to transfer data from VRAM into the L2 cache. However, its large capacity can store vast quantities of data.

Each SM thereby shares resources such as registers, L1 cache, as well as Tensor Cores. Each of those SMs has 128 simple CUDA Cores, which execute threads in blocks of 32, being called a warp. All cores in a warp execute the same instruction simultaneously, a paradigm known as Single Instruction, Multiple Threads (SIMT).

In general, warps are designed to switch execution to a different block without latency when execution stalls, for example, when waiting for data to be loaded from global memory (VRAM). As each thread in a warp executes the same instruction, special care has to be taken when they diverge, i.e., they need to execute different instructions in a warp due to, for example, a branch instruction. The warp scheduler then disables all diverging threads in a warp, which are then executed in a second pass on the diverging execution path. This means processing speed is the highest when there are no divergences within a warp, as otherwise parallel processing capabilities are reduced.

3.1.2 Tensor Cores

With the introduction of the Volta architecture by NVIDIA, they added a new type of core to their GPUs, Tensor Cores. These cores can multiply two full 4×4 matrices together and add them to a third 4×4 matrix in a single clock cycle. This increases the throughput of these GPUs massively. Over the last few years, NVIDIA has improved their Tensor Cores, now on the 4th generation, adding support for many different data types and increasing performance even further.

3.1.3 Memory Layout

The memory hierarchy of a modern GPU consists of several levels, with each level consisting of different capacities and latencies, similar to that of a CPU. In the following list, the different memory levels are defined, sorted from generally the fastest accessible memory to the slowest, with cache sizes for an NVIDIA RTX 4090 shown in brackets [11]:

- Registers (256 KB per SM, maximum of 256 registers of 4 bytes each per thread): The registers are the fastest memory level. They are private to each thread and are used for temporary values, similar to registers in a CPU. As the registers are shared for all threads scheduled on an SM, executing many threads with heavy usage of registers means fewer threads can be scheduled, which reduces occupancy.
- L1 Cache (128 KB per SM): A slightly slower memory level can be seen in the form of an L1 level cache, which caches data recently read or written to. It also serves as an overflow region when the amount of active data exceeds the available registers, called a register spill. Each core also has its own L1 cache.
- Shared Memory: Shared memory, generally, resides in L1 cache. The difference to L1 cache is that it can be read by all threads in a single thread block. For that, data needs to be explicitly declared as shared to make use of this feature. This enables threads to work together on the same data.
- Constant Cache: These are specialized caches that can only be read during execution but can be read by any thread in a thread block.
- L2 Cache (72 MB): A slower cache that is shared by all SMs. It retains recent read and write operations between SMs and VRAM in order to

speed up subsequent reloads. Its size is significantly larger than L1 cache. However, due to its shared nature, each SM is only assigned a fraction of it when the GPU is fully utilized. L2 cache is also used when using multiple GPUs together with NV-Link to quickly synchronize and share data between distinct GPUs or when transferring data from random-access memory (RAM) to the GPU.

- Global Memory (VRAM, 24 GB): Global memory is the bulk of the memory of a GPU, similar to RAM for a CPU.
- Local Memory: This refers to the specific part of global memory reserved for each SM.
- Texture and Constant Memory: Specific regions of global memory that can be filled with data marked as texture or constant data. These are then cached in a constant cache or in the shared memory and can be read by any thread in a thread block.

Data stored in memory itself is organized in memory lines, with each read operation reading 128 bytes of data from a single memory line in L1 cache. This is exactly the size needed to supply a full warp with a single 32-bit datatype at once.

Ideally, a program has all the necessary data in the registers. If that is not possible, data overspills into the slower L1 cache, then into the L2 cache, and finally into global memory. The by far slowest operation involving memory is the transfer between main memory (RAM) and global memory (VRAM). For that, the data is transferred through the L2 cache, from where it can be copied into the faster memory levels when used or transferred to global memory if it does not fit into the L2 cache.

3.1.4 Execution and Best Practices

When writing a kernel, i.e., code that is executed on the GPU, one has to take into concern the core layout and the memory management, as bad practices can degrade performance massively.

A CUDA kernel is launched using two predefined parameters, the first defining how many thread blocks are to be launched and the other defining the size of each thread block. A single thread block is executed on a single SM and can consist of up to 1024 threads. These are then executed in groups of 32, called warps.

This already shows the first important detail, namely that the number of threads per thread block should be divisible by 32 to fully saturate all warps. Multiple thread blocks are then distributed over the SMs, utilizing the GPU while being able to flexibly run on various models.

The second important detail is control flow divergence. When some threads in a warp take a different execution path than others, both execution paths have to be taken by all threads. In this case, only the relevant threads are active for their respective execution paths. This leads to degraded performance when control divergences happen, as the same code has to be executed multiple times. Ideally, all threads in a warp should follow the same execution path, which would not incur any additional computational cost.

A different important aspect is coalesced memory access. Here, data is loaded in memory lines with a size of 128 consecutive bytes for L1 cache and 32 consecutive bytes for L2 cache, being called sectors. To fully utilize this parallel loading of data, the developer should structure their data in such a way that as few loads as possible supply all 32 threads in a warp with good data, i.e., the data is accessed in a coalesced way and memory transfers are fully used. This becomes especially apparent when data is stored in slower memory, such as L2 cache or global memory, as each line read incurs a large amount of waiting cycles. If no other warp is ready for execution, the GPU is effectively idle during that time.

Another effect occurs when the GPU is underutilized due to not executing enough threads in parallel. This can either happen when there are not enough thread blocks to fully saturate all SMs or when an individual SM is not saturated with enough warps, i.e., the thread blocks do not contain enough threads. In this case, parts of the GPU are idle, which also hinders performance. Similarly, when executing a function on the GPU, the number of thread blocks should be several times larger than the number of available SMs. Each SM is equipped to handle multiple thread blocks at the same time, which can be advantageous if a thread block has to wait for results from other thread blocks or are stalled while loading memory from global memory. A sufficiently large number of thread blocks thus ensures good utilization of the GPU.

A related aspect is the tail effect, where at the end of a function execution, only a few active thread blocks remain. These can then no longer saturate the GPU, reducing efficiency. However, there is no limit to the number of kernels being launched simultaneously. Multiple kernels could be launched on different data, which can then be used to fully saturate the GPU. Due to this modular structure, one should consider whether their program fully utilizes

the largest GPU available to them, as varying models have different amounts of cores and SMs. The hardware then handles code execution on the GPU automatically.

A useful metric that measures the utilization of a GPU is the occupancy defined by the ratio of active warps per SM to the maximum number of warps per SM. A low occupancy indicates poor utilization of the GPU, while an extremely high occupancy could lead to resource contention, which also reduces efficiency.

3.2 Interfacing with a GPU

The code that is to be run on the GPU has to be compiled using a special C++ compiler provided by NVIDIA, `nvcc`. `nvcc` generates, similar to other C++ compilers, executable code but can handle precisely defined functions named kernels as code to be run on the GPU. These kernels are then launched using a user-defined number of threads per block and thread blocks.

Typically, and first of all, one has to declare and transfer data to the GPU. Then one has to call a function that utilizes this data. Finally, one has to transfer the data back to the main memory (RAM) so that evaluation can happen and the results can be displayed or saved to disk.

There are also low-level application programming interfaces (APIs) that provide support and optimized implementations for various basic functions, especially for linear algebra. Some examples, among others, are cuBLAS, cuFFT, cuSPARSE, cuRAND, Thrust, and NPP.

Using these, it is also possible to build specific Python modules that call C++ code which executes a kernel on the GPU. However, this requires that the data is present on the GPU.

Most developers, however, use even higher-level APIs, such as TensorFlow [36] or PyTorch [1], where data transfers and function calls are handled by the API and efficiently dispatched to the available GPUs or accelerator units. However, this also abstracts the contact with the GPU away, therefore tempting one to ignore best practices discussed above, despite these APIs being able to optimize utilization of the GPU.

Today, most machine learning models are developed using PyTorch, although some include their own kernels to efficiently implement a desired function tailored specifically to their problem.

Chapter 4

Methods

Analyzing code is not a trivial task, as there are many aspects that determine whether code is good or bad. Especially given that there are different targets code can be optimized for, from memory-efficient design minimizing memory footprint to code that focuses on execution speed, running as quickly as possible.

In this section, various methods of code analysis will be introduced, ranging from using simple tools, like simply measuring the time different tasks take, up to more sophisticated tools graphically visualizing which functions a program is executing on what kind of computing devices.

There are some limitations to code analysis, as not all qualities of good code can be easily seen. One such example would be the memory layout or kernel launch parameters, which are heavily specific to the device it is optimized for. These are difficult to analyze, but can have a major influence on the execution speed. For these things, testing different configurations is necessary.

4.1 Benchmarking Python Code

Today, most machine learning models are implemented in Python. They often use a machine learning framework such as PyTorch [1] to implement the algorithms. Even though Python is an interpreted language in its native form, which is quite slow compared to compiled languages like C++, many frameworks like PyTorch and NumPy call compiled C++ code from Python in order to efficiently run mathematical operations. PyTorch specifically uses the CUDA stream mechanism to execute CUDA kernels parallel to the execution of code on the CPU. This ensures that the GPU is fully utilized, even with a slow interpreted programming language like Python [45].

There are various tools to time the runtime of sections of code. One module in Python, `cProfile`, is written in C++ and offers a lightweight profiler without much overhead. It works well for code run exclusively on the CPU, but falls short when analyzing code that also runs on accelerator devices. For this one should use the profiler offered by PyTorch to analyze one's code. In this way, the code executed on the GPU can be analyzed as well, which helps to show inefficiencies clearly.

The NVIDIA Nsight software suite also provides a plethora of tools for benchmarking, debugging, and developing efficient code. But these tools are better suited to analyze individual kernels instead of entire machine learning models.

Another large area—besides timing the duration of code, which is equally important—is the management of memory, specifically the amount of necessary and unnecessary memory transfers.

By using only the minimum amount of transfers and developing a machine learning model in such a way that data is optimized to stay in the fast caches instead of needing to be reloaded often from slow global memory, the speed can be drastically increased. The limiting factor nowadays is often memory speed instead of computational power. This is due to the very high amount of cores available in modern accelerator devices, which all need to be supplied with data, sharing the available memory bandwidth. The actual available bandwidth to each core is therefore quite small.

4.1.1 Using cProfile

The simplest way to benchmark Python code is to use an efficient and native profiler like cProfile. Its usage is simple by just including the module `cProfile`. Afterward, a `cProfile.Profile` object can be enabled and

```
import cProfile, pstats

def is_prime(prime):
    if prime <= 1:
        return False
    for i in range(2, int(prime**0.5)+1):
        if prime % i == 0:
            return False
    return True

def sum_primes(n):
    sum = 0
    for i in range(2, n+1):
        if is_prime(i):
            sum += i
    return sum

profiler = cProfile.Profile()
profiler.enable()
n = 10000
print(f"Sum of first {n} primes is {sum_primes(n)}.")
profiler.disable()
stats = pstats.Stats(profiler)
stats.sort_stats('cumulative').print_stats(10)
```

Listing 4.1: A simple Python script that calculates the sum of the first 10000 primes by checking each number if they are prime. The code is profiled using `cProfile` and the output can be seen in fig. 4.1.

disabled for varying regions of code. Then, the data can be processed further or printed using a helper module like `pstats`. An example considering this can be in listing 4.1. This script calculates the sum of the first 10000 primes and is immediately benchmarked there using cProfile.

The output of this code can be seen in fig. 4.1, which lists the following information described below.

1. Number of calls (ncalls): The number of times this function was called.
2. Total time (tottime): The total amount of time spent inside this function.
3. Time per call (percall): The average amount of time spent inside this function per single call.
4. Cumulative time (cumtime): The cumulative time spent inside the function plus all functions that this function called.

```
 Sum of first 10000 primes is 5736396.
      10002 function calls in 0.046 seconds
Ordered by: cumulative time
ncalls  tottime  percall  cumtime  filename:lineno(function)
     1    0.005    0.005    0.046  main.py:11(sum_primes)
  9999    0.041    0.000    0.041  main.py:3(is_prime)
     1    0.000    0.000    0.000  {built-in method builtin
s.print}
```

Figure 4.1: Output of the code seen in listing 4.1. Here, the time spent in the two functions can be seen.

5. Filename, line number, and function name (filename:lineno(function)): The filename, line number, and name of the function.

In this example, the three functions called can be seen. The first is the print function, which is a built-in method. This one takes an insignificant amount of time.

The next one is the function `is_prime`, which is called 9999 times. Each individual call only takes a small amount of time. However, due to the high number of calls, it adds up to the majority of the time the program takes.

Last but not least, the function `sum_primes` is called exactly once, and it takes around 5ms. This function is the one that calls `is_prime` 9999 times. For this reason, its cumulative time is very high, making up almost the total runtime of the program.

By using this information, one could focus on improving the method `is_prime`, as that function is called many times, and therefore efficiency gains are very useful. But also the method `sum_primes` could be targeted for optimization. And despite being called only once, it still makes up a significant portion of the total computation time.

4.2 Benchmarking and Analyzing Code with CUDA-kernels

Profiling the memory and compute usage of code run on a single CPU core is usually straightforward. However, in machine learning, often code is run in parallel on accelerator devices, such as GPUs. When using those, profiling is often quite difficult, as timing is not always reported accurately on the CPU side. This is due to non-blocking function calls that just queue up a kernel and return immediately, without the kernel having fully run at that point. A solution could be the usage of synchronization after queuing a kernel. However, that could also impact performance negatively regarding total runtime, as the CPU would be idle whenever a kernel is being run.

The profiler provided by PyTorch does also have the capability to collect information about runtime on the GPU. So using that instead of cProfile is often simpler when facing complex models implemented in PyTorch. One could also use NVIDIA's Nsight tools. However, PyTorch launches a kernel for basically every operation on a tensor that is stored on the GPU, which the NVIDIA tools were not built to handle. Instead, they are made for analyzing a small amount of kernels, which is important when writing more complex custom kernels.

4.2.1 The PyTorch profiler

The PyTorch profiler behaves similarly to the cProfile profiler, but it can also natively measure the time spent in kernel execution and the amount of memory transfers between CPU memory and GPU memory, as well as being able to record traces (see section 4.4). Using the profiler, one can quickly establish an understanding of the bottlenecks and identify the slowest parts of the implementation to either improve them or to make architectural changes by replacing them.

To demonstrate the functions of the PyTorch profiler, the script seen in listing 4.2 implements a simple neural network model in PyTorch. It features a convolutional layer, as well as multiple fully connected layers, besides some reshaping and activation functions. The profiler profiles the activity on the CPU, indicated by the parameter `activities=[ProfilerActivity.CPU]`. Additionally, a warm-up phase is needed, as otherwise, the CPU and GPU are not prepared. During the warm-up phase, code is usually executed in a much slower way than after a few loops, as the branch predictors inside the CPU and GPU are not optimized

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.profiler import profile, record_function
from torch.profiler import ProfilerActivity

class Simple(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Simple, self).__init__()
        self.conv1 = nn.Conv1d(1, 5, 5)
        self.fc1 = nn.Linear((input_size-4)*5, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.unsqueeze(1)
        x = F.relu(self.conv1(x))
        x = x.flatten(start_dim=1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return F.softmax(self.fc3(x), dim=1)

in_size = 100
hidden_channel = 5
hidden_size = 1000
out_size = 20
model = Simple(in_size, hidden_channel, hidden_size, out_size)
inputs = torch.rand([50,100])

with profile(activities=[ProfilerActivity.CPU]) as prof:
    with record_function("model_inference"):
        model(inputs)
        torch.cuda.synchronize()

# Print results ordered by total CPU time
print(prof.key_averages().table(sort_by="cpu_time_total"))
```

Listing 4.2: A simple neural network model implemented in PyTorch demonstrating the PyTorch profiler capabilities.

Name	Self CPU	Self CPU	CPU total	CPU total
model_inference	8.77 %	475.000 µs	99.22 %	5.375 ms
aten::conv1d	0.31 %	17.000 µs	61.38 %	3.325 ms
aten::convolution	1.38 %	75.000 µs	61.07 %	3.308 ms
aten::_convolution	0.57 %	31.000 µs	59.68 %	3.233 ms
aten::mkldnn_convolution	58.69 %	3.179 ms	58.89 %	3.190 ms
aten::linear	0.35 %	19.000 µs	23.04 %	1.248 ms
aten::addmm	20.45 %	1.108 ms	21.99 %	1.191 ms
aten::relu	0.68 %	37.000 µs	3.66 %	198.000 µs
aten::clamp_min	2.97 %	161.000 µs	2.97 %	161.000 µs
aten::copy_	1.31 %	71.000 µs	1.31 %	71.000 µs
aten::unsqueeze	0.72 %	39.000 µs	0.96 %	52.000 µs
aten::softmax	0.17 %	9.000 µs	0.87 %	47.000 µs
aten::zeros	0.59 %	32.000 µs	0.78 %	42.000 µs
aten::t	0.42 %	23.000 µs	0.70 %	38.000 µs
aten::_softmax	0.70 %	38.000 µs	0.70 %	38.000 µs
aten::flatten	0.30 %	16.000 µs	0.63 %	34.000 µs
aten::as_strided	0.41 %	22.000 µs	0.41 %	22.000 µs
aten::_reshape_alias	0.33 %	18.000 µs	0.33 %	18.000 µs
aten::empty	0.31 %	17.000 µs	0.31 %	17.000 µs
aten::transpose	0.17 %	9.000 µs	0.28 %	15.000 µs
aten::expand	0.13 %	7.000 µs	0.17 %	9.000 µs
aten::squeeze	0.07 %	4.000µs	0.09 %	5.000 µs
aten::zero_	0.07 %	4.000µs	0.07 %	4.000 µs
aten::as_strided_	0.06 %	3.000 µs	0.06 %	3.000 µs
aten::resolve_conj	0.06 %	3.000 µs	0.06 %	3.000 µs

Table 4.1: Performance metrics of the model in listing 4.2 using the PyTorch profiler, ”CPU time avg %” and ”# of Calls” omitted. The function calls in yellow in the forward function of the model defined in listing 4.2 are highlighted in yellow.

for the code yet. In the code shown in listing 4.2, the profiler executes the code for `wait=1` loops but is inactive. Then, `warmup=2` loops follow, where the profiler is active but discards the recorded data. Finally, `active=1` loops are run, where the profiler records and saves performance metrics.

Running this short Python script produces the output seen in table 4.1. It details the time spent in each function, similarly to how `cProfile` did in the previous chapter. The yellow lines highlight the different layers and top level functions used in the model.

It is clear that the 1D convolutional layer, as well as the functions in the implementation of PyTorch that are called indirectly, take roughly 60% of the total time, while the dense layers take only 23% of the total time. It is interesting to see that the forward call of the model itself takes around 8.7% of

Name	CPU total	CPU total (%)	GPU total	GPU total (%)
model_inference	335.554 µs	100.00 %	124.735 µs	100.00 %
aten::linear	87.785 µs	26.16 %	107.103 µs	85.86 %
aten::conv1d	70.644 µs	21.05 %	7.648 µs	6.13 %
aten::relu	43.081 µs	12.84 %	7.424 µs	5.95 %
aten::softmax	12.584 µs	3.75 %	2.560 µs	2.05 %
aten::unsqueeze	10.670 µs	3.18 %	0.000 µs	0.00 %
aten::flatten	3.446 µs	1.03 %	0.000 µs	0.00 %

Table 4.2: Performance profiling results with CPU and GPU metrics of the model in listing 4.2. Percentage results are normalized in a way where ”model_inference” equals 100%. As the model does not incorporate synchronization after every function call, the results on the CPU are distorted, capturing only the total time in ”model_inference.” Only the highlighted functions in table 4.1 are shown here.

the total time, despite only calling PyTorch operations on the input parameters. That is likely due to the small model and input data size making the time spent in this function noticeable. However, when increasing input size, then this will become less significant.

A disadvantage of using the PyTorch profiler in this form is that the ordering is slightly confusing, because it is not always obvious which functions are called by others. To counteract that, multiple labels could be introduced by wrapping the appropriate code in a `with record_function("step_xxx"):` block in the code. This helps to group larger sections of code into a single label, of which the cumulative time can be used for comparison as well.

To execute the same model on the GPU, the code for the model has to be slightly changed in a way that allows the transfer of the model and input data to the GPU prior to calling the forward function. The transfer is done by calling the `cuda()` function on the model and input data tensor, which transfers them to global memory. Additionally, the profiler also needs to record activity on the GPU at this point. This is done by appending the array provided to `activities` with `ProfilerActivity.CUDA`, resulting in the timings in table 4.2 showing a much lower total runtime.

These results show that actually only four functions utilize the GPU, namely the linear and convolutional layers as well as the activation functions. The other two operations are merely implemented by changing the view of the underlying data.

Looking at the timings, it is also noticeable that the linear layers now take the majority of the time, hinting at a very efficient implementation of the convolution on the GPU.

One problem that might not be obvious at first sight is that the time spent on the CPU in the various functions does not add up to 100%. This is likely due to the functions simply calling a CUDA kernel, which is executed in parallel, and then waiting for the results to be ready, which then is not recorded. As already mentioned, this issue can be circumvented by synchronization, i.e., by waiting for all currently queued kernels to finish execution. If this is done in combination with custom labels from `record_function` after every important operation, the reported results in the output are accurate. However, this disables the advantages of parallel execution.

Another way that depicts the time spent in various parts of the script much better is to use GPU traces, as described in section 4.4.

Memory and the allocation and deallocation of it are oftentimes also important aspects of code. Especially when a loop is executed hundreds of times when training a model, it is beneficial not to allocate and free memory every iteration. Instead, already reserved memory should just be reused. To measure memory usage, the PyTorch profiler has the capability to show the amount of memory allocated and freed in each function. This can be done by setting the parameter `profile_memory=True` when creating it. This can uncover inefficiencies when parts of the code allocate and free large amounts of memory instead of reusing already prepared memory. However, optimizing memory management likely only leads to small performance improvements, as PyTorch already manages and reuses memory, reducing the overhead of repeated reserving and freeing memory.

4.3 Using NVIDIA Nsight to Analyze Memory Access Patterns

The NVIDIA Nsight suite offers a plethora of tools for benchmarking, debugging, and developing efficient code. NVIDIA Nsight is better suited to analyze individual kernels compared to the PyTorch profiler, which can be used to analyze entire models written in Python. NVIDIA Nsight records various statistics for every kernel run in the measured program, albeit at a significant overhead for each kernel. This in turn makes it unsuitable to analyze entire AI models written in PyTorch, but it works very well when said models incorporate custom kernels that can be analyzed in an isolated environment. These custom kernels are often not limited by processing power, as modern GPUs feature thousands of individual cores. Instead, they are often limited by memory throughput.

Even though there exists high-bandwidth memory on modern GPUs, that bandwidth is shared by all cores, with each core only having a fraction of the total bandwidth available. Different levels of caches somewhat attenuate this problem. However, the faster caches have only a very limited memory capacity. To fully utilize those caches and the available memory bandwidth, special care has to be taken. For example, preventing frequent reloading of data from global memory. This can be achieved by dividing datasets into chunks of memory that fit into the L2 cache and then doing work on the different chunks of data consecutively. This way, the needed data resides in L2 cache and is only being fetched from global memory when working on a new chunk of data.

When benchmarking kernels with the tools available in the NVIDIA Nsight software suite, one important metric to measure is the cache hit rate, i.e., the amount of data accesses that are present in cache and that do not have to be reloaded. A low hit rate indicates inefficient memory accesses that fetches data often from slower memory. To demonstrate this, an example is shown in listing 4.3. It is a 2D matrix add kernel that simply adds two matrices elementwise, as published in the NVIDIA blog [41]. It is run with 32×32 blocks and 32×32 threads per block.

At first glance, there appears to be no problems within this kernel. However, when taking the memory layout of arrays and the order in which `threadIdx` is counted into account, it becomes obvious that memory access is not coalesced.

In a warp, `threadIdx.x` is increased first, while `threadIdx.y` is

```
const size_t size_w = 1024;
const size_t size_h = 1024;
typedef unsigned mytype;
typedef mytype  arr_t[size_w];

__global__ void matrix_add_2D(
    const arr_t * __restrict__ A,
    const arr_t * __restrict__ B,
    arr_t * __restrict__ C,
    const size_t width, const size_t height)
{
    size_t idx = threadIdx.x+blockDim.x*(size_t)blockIdx.x;
    size_t idy = threadIdx.y+blockDim.y*(size_t)blockIdx.y;

    if ((idx < height) && (idy < width))
        C[idx][idy] = A[idx][idy] + B[idx][idy];
}
...
```

Listing 4.3: Simple CUDA kernel that adds two matrices together. Here, the memory access pattern is inefficient, as each load instruction fetches an entire memory line that only supplies a single thread. It would be better to only increase the second index in a single warp.

constant in a warp in this launch configuration. As each warp contains 32 threads, ideally memory accessed by these 32 threads is consecutive, so that each loaded memory line can supply multiple threads. This leads to a quick execution of all relevant code in the warp, taking only eight loads and four store instructions per warp, as each memory line loaded has a size of 32 bytes when fetched from L2 cache.

However, here memory access is not coalesced, resulting in uncoalesced memory access and therefore in inefficiencies. In this specific case, each warp would need to two load instructions and one store instruction for each thread, resulting in 64 loads and 32 stores per warp. This can also be seen when using tools like NVIDIAs Nsight Compute, which show various statistics when they are used to analyze the program. Relevant statistics in this case would be:

1. `l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum`: The number of memory load requests made when running the profiled function.

2. `l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum`: The actual number of sectors transferred necessary to run the profiled function.

Here, a sector describes 32 bytes of memory. Dividing the number of actual transfers by the number of requests made shows the necessary number of memory transfers per request. In this case, there are $65,536$ requests and $2,097,152$ actual transfers, resulting in 32 transfers per request. This means that in each warp, 64 sectors have to be loaded to execute all threads. This is highly inefficient, as each sector contains 32 bytes, of which only four are used.

Changing the access pattern in order to coalesce memory access requires changing the way the arrays are accessed. In the kernel above, `threadIdx.x` is increased first in a thread block, advancing `threadIdx.y` only when `threadIdx.x` resets. This means that each warp has a constant `threadIdx.y`, while `threadIdx.x` ranges from 0 to 31. Now, when accessing the array, the first index is not constant in an array, which results in memory access with a stride of $1024 \cdot 4$ Bytes. In this case, it would be much more efficient if neighboring threads would access neighboring values in the memory, making each sector supply multiple threads. Changing the indexing would result in each warp accessing a continuous block of memory of $32 \cdot 4$ Bytes, which is the size of exactly four sectors of L2 cache. This can be achieved by changing the following line:

```
C[idx][idy] = A[idx][idy] + B[idx][idy];
```

to:

```
C[idy][idx] = A[idy][idx] + B[idy][idx];
```

This reduces the number of actual transfers from $2,097,152$ to $262,144$, one-eighth of the original value. Each sector can now supply eight threads instead of just one.

This optimization does not just reduce the amount of necessary memory transfers massively, but it also increases execution speed. In the NVIDIA blog [41], they report a 68% decrease in runtime. However, on the available NVIDIA RTX 3070Ti Laptop GPU, it only resulted in an 8% decrease in runtime. This could be due to improvements of the compiler since publication or better hardware due to hardware improvements, such as better thread scheduling, faster memory transfers—or simply due to improved hardware with larger caches.

4.4 GPU Traces

Another very powerful tool is to visualize the runtime of various parts of the code. This can be done by using GPU traces, created by either NVIDIA Nsight or by using the PyTorch profiler by saving a trace using the function `export_chrome_trace(filename)` on a PyTorch profiler object. These are similar to the tables created by `cProfile` by the PyTorch profiler. However, they are graphical and interactive, showing most individual function calls and their duration among multiple threads and devices.

An example trace of the code in listing 4.2 after warm-up viewed in the Chrome trace viewer (`chrome://tracing`) can be seen in fig. 4.2.

At the top, activities on the CPU can be seen in column ”thread 5024.” At the bottom, activities on the GPU can be seen in column ”stream 7.” In this case, it becomes clear that the convolution layer executes two very fast kernels on the GPU while taking much longer on the CPU. In contrast to that, the three linear layers each launch a more computationally intensive kernel, which takes longer to execute on the GPU than on the CPU, making the entire process wait slightly at the end for these kernels to finish execution. This can be seen by the last function on the CPU, which is `cudaDeviceSynchronize`, which waits for all active kernels.

More generally, this trace also shows the limitations of the model. The first half with the convolution is limited by the CPU, while the latter half is limited by the GPU. This means improvements in the first half would have to target the code of the convolutional layer on the CPU, as improvements on the GPU would not increase the speed, as the CPU is the slower part. Similarly, the second half would have to implement a faster matrix multiply operation on the GPU to further increase execution speed. However, from the name of the kernel, it can already be seen that the kernel is a specific implementation from PyTorch targeting the available GPU architecture for optimal performance.

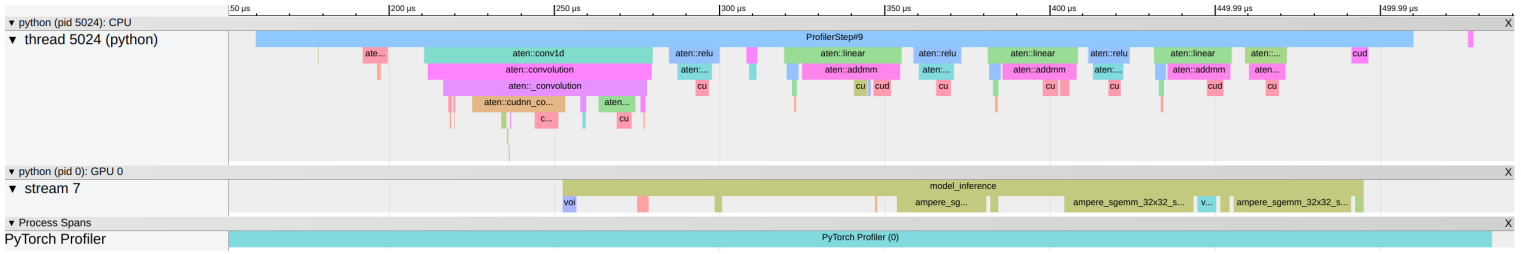

Figure 4.2: Chrome trace for the code in listing 4.2 after warm-up. At the top, activities on the CPU can be seen in column ”thread 5024,” and at the bottom, activities on the GPU can be seen in column ”stream 7.”

4.5 System Documentation

The available hardware for this project features an AMD Ryzen 7 6800H CPU, 64 gigabyte (GB) of RAM, and an NVIDIA GeForce RTX 3070 Ti laptop GPU featuring 8 GB of dedicated memory. All projects in this work were conducted on a system running Manjaro Linux with kernel version 6.6.

The software environment included Python versions 3.8 and 3.11, CUDA versions 12.4 and 12.5, and appropriate versions of PyTorch. These versions were chosen to be compatible with the projects analyzed and to ensure compatibility with the hardware.

Experiments were conducted within isolated virtual environments to ensure reproducibility and avoid software conflicts.

Chapter 5

Model Selection

This chapter introduces the three selected models for an analysis. First, the selection process is outlined. Then, the three models selected for a thorough analysis are described in detail. For this, each model is first introduced, followed by an explanation of their mathematical background, and finally their implementation is presented.

5.1 Model selection

In this thesis, various models have been presented in chapter 2. A superficial analysis of a subset of these models has revealed that many of these models use similar components in their respective implementations.

Almost all models utilize neural networks in the form of MLPs in their implementation. These rely on PyTorch, which has a highly optimized implementation for these neural networks. Therefore, short of architectural changes, which are not covered in this thesis, there is only minimal potential for improvement regarding these.

A different function that is utilized by many models that use 2D, 3D, or 4D data structures is PyTorch's `torch.nn.functional.grid_sample`. This function is used to interpolate the values for a given position in such a data structure. Methods that utilize this function are HexPlane [9] and K-Planes [15] to bilinearly interpolate in their 2D planes. MSTH [57] and TiNeuVox [14] also utilize `grid_sample` to interpolate values in their 3D and 4D data structures. Similarly to MLPs, this function is implemented very well in PyTorch, launching different kernels that are optimized for various GPU architectures.

However, many of these models make heavy use of masking, indexing, reshaping, and concatenation of tensors in their implementation. While PyTorch offers fast solutions to these operations in the form of views, these operations often result in non-contiguous tensors, which means that the data of the tensor is no longer stored in a single easily accessible chunk but can potentially be at distinct memory addresses. This becomes a problem when kernels that expect a continuous array of data are called on these tensors, as they have to be made continuous again prior to execution. This can lead to inefficiencies.

Custom CUDA kernels are another interesting component to analyze. They are utilized in InstantNGP [39] and 4DGS [59], which inherits the rendering kernels from 3DGS [25]. Due to the already very high level of optimization in Instant-NGP, there is very little reason for an analysis, except to show why the model is so fast. It is much more interesting to analyze a model like 4DGS that also utilizes custom CUDA kernels, but that is not yet fully optimized.

Due to these reasons, three models that cover many of the mentioned aspects above were chosen for a detailed analysis. The three models are:

- HexPlane [9], chosen for its fast model and usage in other models, such

as 4DGS.

- DynIBaR [30], selected for its high-quality results but complex and slow architecture.
- 4D Gaussian Splatting [59], an extension to the fast 3D Gaussian Splatting [25], to understand how the change from static to dynamic scenes influences efficiency and to analyze a complex custom kernel.

5.2 HexPlane

HexPlane [9] has been published in 2023 and represents a dynamic 3D scene by using six 2D planes instead of storing data in a 4D matrix. It introduces a good way of storing high-dimensional sparse data factorized as lower-dimensional data. Despite using small neural networks in the form of MLPs, the model is relatively simple and fast, making it suitable for analysis to demonstrate a good implementation to solve the problem of novel view synthesis.

In this section, first the mathematical fundamentals are described, followed by a description of the authors' implementation. The next step is an analysis of the code, using a profiler. Then, possible improvements are discussed, followed by a final section summarizing the analysis on HexPlane.

5.2.1 Mathematical Background

HexPlane [9] is a hybrid model that features an explicit 4D representation storing features of a dynamic scene and a small MLP to decode said features into color $\mathbf{c}$ and density σ at any given point $\mathbf{x}$. A novel view of the scene can then be rendered using volumetric rendering eq. (2.3), as described in section 2.1.

The explicit volume can be described as a 4D matrix $\mathbf{V}_{4D} \in \mathbb{R}^{N^3 \times T \times F}$, where N is the amount of points for each spatial dimension, T is the amount of temporal time steps, and F is the amount of features per point. Instead of using a single matrix, HexPlane uses factorization and utilizes six 2D planes, one for each combination of spatial dimensions $\mathbf{M}^{XY}, \mathbf{M}^{XZ}, \mathbf{M}^{YZ} \in \mathbb{R}^{N^2 \times F}$ and three spatio-temporal planes $\mathbf{M}^{XT}, \mathbf{M}^{XT}, \mathbf{M}^{XT} \in \mathbb{R}^{N \times T \times F}$. This factorization reduces memory requirements massively and allows for a higher spatial and temporal resolution. Additionally, the model employs learned feature basis vectors $v^1, v^2, v^3 \in \mathbb{R}^F$.

The 4D volume $\mathbf{D}$ can then be recovered by combining these attributes in the following way:

$$\mathbf{D} = \sum_{r=1}^{R1} \mathbf{M}_r^{XY} \circ \mathbf{M}_r^{ZT} \circ \mathbf{v}_r^1 + \sum_{r=1}^{R2} \mathbf{M}_r^{XZ} \circ \mathbf{M}_r^{YT} \circ \mathbf{v}_r^2 + \sum_{r=1}^{R3} \mathbf{M}_r^{XT} \circ \mathbf{M}_r^{YZ} \circ \mathbf{v}_r^3 \tag{5.1}$$

Here, $\circ$ is the outer product.

To allow a better representation, there are multiple matrices and feature

vectors for each plane, specifically $R1, R2, R3$ many.

To now query a HexPlane, one could also formulate $\mathbf{D}$ as a function that maps a position $\mathbf{x} = (x, y, z)$ and time t to a feature vector:

$$\begin{aligned} D :&\mathbb{R}^4 \rightarrow \mathbb{R}^F, \\ D(x, y, z, t) =&(\mathbf{P}^{XYR1}_{x,y,\bullet} \odot \mathbf{P}^{ZTR1}_{z,t,\bullet})\mathbf{V}^{R1F} + \\ &(\mathbf{P}^{XZR2}_{x,z,\bullet} \odot \mathbf{P}^{YTR2}_{y,t,\bullet})\mathbf{V}^{R2F} + \\ &(\mathbf{P}^{XTR3}_{x,t,\bullet} \odot \mathbf{P}^{YZR3}_{y,z,\bullet})\mathbf{V}^{R3F} \end{aligned} \tag{5.2}$$

Here, $\mathbf{P}^{XYR1}$ are all $\mathbf{M}^{XY}_i$ matrices stacked to a 3D tensor, while $\mathbf{V}^{R1F}$ stacks all $\mathbf{v}^1_i$ vectors to a 2D tensor; the other terms are defined analogously. The $\odot$ operator defines an elementwise product, while the subscript $x, y, \bullet$ describes a splicing operation using bilinear interpolation. For example, $\mathbf{P}^{XYR1}_{x,y,\bullet}$ describes the vector interpolated from neighbors at index x and y, resulting in a vector of dimension $\mathbb{R}^{R1}$.

In total, querying a HexPlane consists of bilinearly interpolating each plane at the given position, resulting in a vector, which is elementwise multiplied to the plane with orthogonal axes. These vectors are then multiplied with the feature vector matrix and added together, resulting in the final feature vector.

This formulation allows density σ to be regressed from one HexPlane D_σ using a small MLP $\mathcal{F}_\sigma$, while color is regressed from another HexPlane $D_\mathbf{c}$ using a different small MLP $\mathcal{F}_\mathbf{c}$ with features vector and viewing direction $\mathbf{d}$ as input:

$$\sigma = \mathcal{F}_\sigma(D_\sigma(x, y, z, t)) \tag{5.3}$$

$$\mathbf{c} = \mathcal{F}_\mathbf{c}(D_\mathbf{c}(x, y, z, t), \mathbf{d}) \tag{5.4}$$

The objective to optimize a HexPlane is stated as follows:

$$\mathcal{L} = \frac{1}{|\mathcal{R}|} \sum_{r \in \mathcal{R}} ||C(r) - \hat{C}(r)||^2_2 + \lambda_{reg}\mathcal{L}_{reg} \tag{5.5}$$

Here, $\mathcal{L}_{reg}$ describes a combination of a total variational loss, enforcing smooth gradients in each plane, and a depth smoothness loss, enforcing smoothness in depth. λ_{reg} is a simple scaling coefficient for the regularization loss.

Other optimizations to speed up training are a coarse-to-fine training scheme, which starts training with low-resolution planes, increasing their resolution gradually over training, and a tiny, time-independent emptiness

voxel, crafted by querying densities in its respective area and saving the maximum to skip evaluations in empty space.

5.2.2 Implementation Details

In the provided implementation by the authors of HexPlane, they implemented their model in PyTorch using six tensors for the six planes. Interpolation of the plane is achieved by using the `torch.nn.functional.grid_sample` function, which bilinearly interpolates values on a grid using input positions and supports handling of entire batches in a single function call. In their model, they make heavy use of the `torch.cat` and `torch.stack` functions, though mostly for combining small vectors such as coordinate inputs. They also use a binary mask to mask out invalid ray positions for training, either through the emptiness voxel or by encountering a low-density point.

Training is handled by a dedicated `Trainer` class that follows a typical scheme for machine learning consisting of the following steps:

1. Data sampling: A random batch of rays is chosen from all available rays.
2. Rendering: The colors of the selected rays are calculated using the HexPlane model.
3. Loss: All necessary losses are calculated from the rendered rays and the expected outcome.
4. Optimization: Backpropagation is applied using PyTorch, and the weights of the model are updated accordingly.
5. Statistics: The current training state is calculated and feedback is given to the user. At predetermined steps, images are rendered and recorded to disk.

The model can be customized using a multitude of configuration options, which can turn on certain extra features and losses, for example, using SH from which color is regressed instead of using an MLP or activating/deactivating certain losses.

5.3 DynIBaR

DynIBaR [30] was published in 2023 and tackles the disability of NeRF-based approaches to reconstruct novel views from longer videos with shaky monocular viewpoints, producing blurry results and failing to capture motion from faraway times. DynIBaR is a model that utilizes image-based rendering, using the input images directly to reconstruct novel views, instead of relying solely on a learned inherent representation.

To handle the dynamics of a scene, the authors propose a motion trajectory field, which is used to project a ray at the requested time into multiple neighboring frames while correcting for the motion of the ray over these frames. This way, features from multiple source images can be combined. In a second step, these extracted features, along with an encoded time embedding, are fed into a complex neural ray transformer network. This network outputs a color and a density, which are used in volumetric rendering to generate a novel view. These two steps are jointly optimized using cross-time rendering for temporal consistency. Additionally, DynIBaR also features a static model, which is combined using segmentation masks using Bayesian learning. All these features make DynIBaR a computationally intensive model that is very demanding to train, often taking multiple days to optimize the model for a given scene.

Despite its high training time, researchers use it as a benchmark because of the high reconstruction quality the model offers. Its ability to use shaky, 30-second-long, monocular videos while providing good results is unachieved by many other methods. This makes it a state-of-the-art benchmark for NVS from long monocular videos.

5.3.1 Mathematical Background

DynIBaR uses monocular video with N images of a given scene $(I_1, I_2, ..., I_N)$ with camera parameters $(P_1, P_2, ..., P_N)$. To render a frame at time i, it uses the neighboring frames $\mathcal{N}(i)$ in a temporal range r, such that the total source frames are $\mathcal{N}(i) = \{I_{i-r}, ..., I_{i+r}\}$. For each source view I_j, a convolutional neural network extracts features F_j which are used together with the camera parameters P_j and the image I_j to form a tuple (I_j, P_j, F_j). To now render a ray, all tuples in the temporal radius r can be used.

To calculate the motion trajectory field, coefficients are encoded in an MLP

G_{MT}, which uses position $\mathbf{x}$ and time in the form of frame number i as input:

$$(\phi_i^1, ..., \phi_i^L) = G_{MT}(\gamma(\mathbf{x}), \gamma(i)) \tag{5.6}$$

Here, γ denotes a positional encoding. The output of G_{MT} at time i are L coefficients $\phi_i^l \in \mathcal{R}^3$.

Separately from the coefficients, a motion trajectory basis $(h_i^1, ..., h_i^L)$ is also learned. Both combined form the motion trajectory:

$$\Gamma_{x,i}(j) = \sum_{l=1}^{L} h_j^l \phi_i^l(\mathbf{x}) \tag{5.7}$$

The displacement of a point $\mathbf{x}$ from time i to j can now be calculated as the following:

$$\Delta_{\mathbf{x},i}(j) = \Gamma_{\mathbf{x},i}(j) - \Gamma_{\mathbf{x},i}(i) \tag{5.8}$$

Transforming a query position to a neighboring input requires only a single evaluation of the MLP G_{MT} in this way. Each point $\mathbf{x}$ of a ray at time i is then calculated for each input image in the neighborhood $N(i)$ at time j as $\mathbf{x}_{i\to j} = \mathbf{x} + \Delta_{\mathbf{x},i}(j)$. Each warped point $\mathbf{x}_{i\to j}$ is then projected to the corresponding 2D source view I_j plane using the camera parameters at that time P_j to extract the features f_j for that pixel location. The features are then combined and used as input along with a time embedding $\gamma(j)$ to the ray transformer MLP, which infers color $\mathbf{c}$ and density σ to be used with volumetric rendering (eq. (2.3)) to produce the final pixel color $\hat{\mathbf{C}}_i$ for the ray.

To prevent overfitting of the model and to properly learn the motion trajectory field, the authors use cross-time rendering, i.e., first transforming a straight ray from time i to a motion-adjusted ray at time j, from which an image is rendered using the above-described procedure, producing the final color $\hat{\mathbf{C}}_{j\to i}$. This pixel color is then used to produce a motion-disocclusion-aware RGB reconstruction loss:

$$L_{pho} = \sum_{\mathbf{r}} \sum_{j\in\mathcal{N}(i)} W_{j\to i}(\mathbf{r}) \rho(\mathbf{C}_i(\mathbf{r}), \hat{\mathbf{C}}_{j\to i}(\mathbf{r}) \tag{5.9}$$

Here, ρ is a generalized Charbonnier loss [10] and $W_{j\to i}(\mathbf{r})$ describes a motion disocclusion weight, which is calculated by the difference of the accumulated transparency α between time i and j of the ray, with $W_{i\to i}(\mathbf{r}) = 1$.

To render static scenes in high quality, DynIBaR uses a separate static model for the static part of the scene. This static model renders the scene similarly but skips the motion adjustment and simply uses the perspective transformation between input images. The resulting colors of the static model $\hat{\mathbf{C}}^{\text{st}}$ and the color of the dynamic model $\hat{\mathbf{C}}_i^{\text{dy}}$ are then combined to the full color $\hat{\mathbf{C}}_{j\to i}^{\text{full}}$, which rewrites eq. (5.9) as the following:

$$L_{pho} = \sum_{\mathbf{r}} \sum_{j \in \mathcal{N}(i)} W_{j\to i}(\mathbf{r}) \rho(\mathbf{C}_i(\mathbf{r}), \hat{\mathbf{C}}_{j\to i}^{\text{full}}(\mathbf{r})) \tag{5.10}$$

When initializing the model, the motion mask M_i is estimated using a lightweight model based on Bayesian learning techniques. Using the mask, a separate color loss can then be applied to regions with motion rendered by the dynamic model and static regions rendered by the static model:

$$L_{mask} = \sum_{\mathbf{r}} (1 - M_i(\mathbf{r})) \rho(\hat{\mathbf{C}}^{st}(\mathbf{r}), \mathbf{C}_i(\mathbf{r})) + \sum_{r} M_i(\mathbf{r}) \rho(\hat{\mathbf{C}}_i^{dy}(\mathbf{r}), \mathbf{C}_i(\mathbf{r})) \tag{5.11}$$

The authors use multiple additional losses as additional regularization. The first loss is a data-driven loss based on depth and optical flow. A second loss is a motion trajectory term that enforces trajectory fields to be cycle-consistent and spatial-temporally smooth. Finally, a compactness loss prevents floaters—low density points near the origin of a ray that have no corresponding object in the scene.

5.3.2 Implementation Details

The implementation of the model provided is mainly done in the class `DynibarMono`, which encapsulates the model. It has several important attributes, which are used to render an image. First, the `feature_net` is a simple residual convolutional network, which calculates the features f_i of a source image. Secondly, the `motion_mlp` G_{MT} is an MLP, which infers the motion coefficients of the input points. This is used in conjunction with the learnable motion trajectory basis `trajectory_basis`. Lastly, the ray transformer network consists of several `torch.nn.Linear` layers as well as periodic embeddings and multiple matrix operations.

The `feature_net` is a convolutional neural network implemented using PyTorch in its default state. It consists of two convolutional layers and three blocks of two convolutional layers, each with a residual connection. It also

features instance normalization as well as a ReLU activation function.

The `motion_mlp` is an MLP implemented in PyTorch, which generates the coefficients for the motion trajectories. In its default state, it consists of an embedding function that generates a periodic embedding of the input and nine linear layers that are activated using the ReLU function, making it quite deep. There also exists a single skip connection, which concatenates the input to the output of the fourth layer again.

The ray transformer MP also features a periodic embedding for time and several fully connected layers, also implemented using PyTorch.

The majority of the rendering process is handled by the auxiliary function `render_rays_mono`. This complex function performs all the necessary steps to produce a color for a ray. It also provides additional parameters for the loss functions in the learning process.

To derive the color of a given pixel, points are first sampled along the ray cast by the pixel to be rendered. For each of these points, the density and color need to be calculated to be used in volumetric rendering. Using the `motion_mlp` of the model, the motion trajectory coefficients of the points in time are calculated, which are multiplied on the `trajectory_basis` inside the `compute_traj_pts` function. Using the motion trajectory, the position of the points on nearby input frames is then computed. These calculations all use simple matrix operations.

Using the `Project` class, the points are projected to the source images in order to aggregate the features extracted by the `feature_net`. These combined features and an embedding of the current time are then used as input to the dynamic and static ray transformer network together. This is done with the help of PyTorch's `torch.nn.functional.grid_sample` function, which takes a two- or three-dimensional grid and interpolates the values at the given positions.

In the following, the static and dynamic ray transformer network is applied to extract the raw RGB colors and density features. These outputs of both models are then combined according to the dynamic and static masks and form the actual colors and density values. In a last step, the optical flow is calculated from the outputs using simple matrix operations.

During training, however, one of the algorithm's losses involves cross-time rendering. Large parts of the procedure above are repeated, in this case using only the dynamic ray transformer network to calculate colors and densities from which the occlusion weights can then be generated.

The losses are calculated using various matrix operations. PyTorch's automatic differentiation engine is then used to propagate the loss backwards

to the learnable parameters, optimizing them to fit the target images better.

In summary, the implementation features multiple neural networks that are implemented using PyTorch. In addition, it also uses functions for sampling like the `torch.nn.functional.grid_sample` function, as well as various matrix operations, which are all already executed on the GPU using CUDA kernels. Also, the implementation makes heavy use of the `torch.cat` function, which combines multiple tensors by allocating a new one and copying the two tensors into it.

5.4 4D Gaussian Splatting

4D Gaussian Splatting [59] is an extension of 3D Gaussian Splatting [25], where the scene is represented by a set of Gaussian distributions. Each Gaussian is represented by different properties, such as position and covariance. To derive an image from the set of Gaussians, they are simply projected onto the 2D image plane via a custom Gaussian Rasterizer kernel inherited from the 3DGS approach. During optimization, several mechanisms prune or create additional Gaussians to represent the given scene in a better way.

To model dynamic scenes, a time-dependent deformation field is introduced, explained in section 5.4.1, which computes a time-dependent offset to the Gaussian's positions and covariance while leaving the other parameters like color or density untouched. The deformation field is based on a K-Planes [15] or HexPlane [9] approach, where out of six multi-resolution planes, deformation features are extracted, which are then decoded using small MLPs to the actual deformations of the Gaussian.

This model is interesting as it achieves good results while keeping training times and memory footprints low. The biggest advantage of 4DGS is that a trained scene can be rendered in real time, with the authors claiming to achieve more than 30 FPS. This was not achieved at the time for dynamic scenes with high-quality reconstructions.

5.4.1 Mathematical Background

The foundations of Gaussian Splatting have been covered in section 2.1.3. 4DGS extends Gaussian splatting to dynamic scenes by introducing a Gaussian deformation field $\mathcal{F}$, which calculates deformed Gaussians using the deformation $\Delta\mathcal{G}$ at that time-step:

$$\mathcal{G}' = \mathcal{G} + \Delta\mathcal{G} \tag{5.12}$$

The Gaussian deformation $\Delta\mathcal{G}$ is calculated in two steps.

First, a spatial-temporal structure encoder $\mathcal{H}$ encodes both the spatial and temporal features of the Gaussians:

$$f_d = \mathcal{H}(\mathcal{G}, t) \tag{5.13}$$

This is achieved by using an approach based on HexPlane [9] (see sections 2.3.2 and 5.2) where six planes $R_l(i, j)$ at two different resolutions

and a tiny MLP ϕ_d are used to interpolate intermediate features f_h from the current Gaussians position and time:

$$f_h = \bigcup_l \prod \text{interp}(R_l(i,j)), \\ (i,j) \in \{(x,y),(x,z),(y,z),(x,t),(y,t),(z,t)\} \tag{5.14}$$

"interp" here describes the bilinear interpolation given the current position in the HexPlane. The small MLP ϕ_d is applied to these features to get the actual spatial and temporal features of the Gaussians:

$$f_d = \phi_d(f_h) \tag{5.15}$$

Secondly, a multi-head Gaussian deformation decoder $D = \{\phi_\mathcal{X}, \phi_r, \phi_s\}$ generates the deformations of position $\Delta\mathcal{X}$, rotation ΔR, and scale ΔS using three separate MLPs:

$$\Delta\mathcal{X} = \phi_\mathcal{X}(f_d) \tag{5.16}$$
$$\Delta R = \phi_r(f_d) \tag{5.17}$$
$$\Delta S = \phi_s(f_d) \tag{5.18}$$

Having all the deformations, the time-aware Gaussians can then be recovered as:

$$\mathcal{G}' = \{\mathcal{X} + \Delta\mathcal{X}, R + \Delta R, S + \Delta S, \sigma, \mathcal{C}\} \tag{5.19}$$

The model is initialized using a random point cloud first, trained for 3000 iterations as a static model, and then further optimized using the dynamic model. The optimization uses a simple L1 color loss function and an additional total-variation loss to smooth out the features on the planes of the HexPlane model.

During optimization, Gaussians can be removed if their density decreases below a threshold. To better reconstruct underrepresented areas, Gaussians are split and shrunk when they cover a large portion of the rendered image. Similarly, Gaussians are cloned when they are small and underrepresent the given geometry.

5.4.2 Implementation Details

The model and training procedure are implemented using Python and PyTorch but also feature a custom-written kernel from the 3D Gaussian implementation [25] for the forward and backward rendering pass. The training procedure consists of two training loops. The first one is a coarse training for a small number of iterations, which does not use time-dependent Gaussians, i.e., no deformation is used, followed by a fine training using the full model.

A single training iteration can be further characterized by multiple steps:

1. Data sampling: The current training batch is sampled, either by using PyTorch's `torch.utils.data.DataLoader` class or by constructing the batch manually, when the whole dataset can fit into memory.

2. Rendering: An image is rendered in two parts, first by calculating and applying the deformation, followed by calling the Gaussian Rasterizer, which actually renders the image from the deformed Gaussians.

3. Loss calculation: Using the rendered image, the L1 color loss is calculated, as well as the total variation loss of the planes of the HexPlane model.

4. Backpropagation: Using the loss, PyTorch calculates the gradients of each variable in the call graph.

5. Densification and pruning: Gaussians that have a low density are removed. Large Gaussians in view space are split into multiple smaller ones, and Gaussians in regions where the image is not reconstructed well are cloned and moved slightly in the direction of their gradient.

6. Optimization: The optimizer changes the model's parameters to fit the target image in a better way.

7. Statistics and logging: Over several steps during training, the model is saved to disk, depending on the configuration, and the model's fitness is saved and displayed.

The deformation network's first step to extract the hidden features is implemented similarly to HexPlane in section 5.2 and also makes use of PyTorch's `torch.nn.functional.grid_sample` function to interpolate the features of the planes, followed by a small two-layer MLP with a single ReLU activation function. Afterward, the multi-head deformation

decoder, another set of small MLPs, is applied to get the deformations $\Delta\mathcal{X}$, Δr, and Δs for each Gaussian. Each of those small MLP also consists of only two linear layers, with a ReLU activation function before each layer.

The Gaussian Rasterizer is implemented using the PyTorch class `torch.autograd.Function`, which invokes a custom CUDA kernel for the forward and backward pass. For the kernel, the image to be rendered is split into 16×16 pixels large tiles, with each tile representing a thread block. Each thread then operates on a single pixel, calculating the final color and depth.

The forward call of the Gaussian Rasterizer first prepares all the variables, such as calculating the covariance in 2D screen coordinates, the colors from the SH coefficients, and assigning only the needed Gaussians to each pixel using frustum culling. The main kernel then first collects all the needed values into a shared memory. Afterward it iterates through the relevant Gaussians, by which transparency, color, and depth of each single Gaussian are calculated. Then at last the final color and depth are calculated. The backward call of the kernel works similarly, but in reverse, calculating the gradients for every input variable.

In the backpropagation step, PyTorch's automatic differentiation engine is used. It propagates the loss back through the call graph, executing the above-described Gaussian Rasterizers backward kernel.

Before calling the optimizer, the densification process is executed. Afterward, the optimizer updates the model. In a final step, progress is saved to disk, depending on the configuration.

Chapter 6

Analysis and Optimizations

This section will present the analysis conducted on various models for NVS. First, the experimental setup is explained for each model, where potential changes to the recommended launch options and used frameworks are explained and their effects on each model's performance are explained. In the next step, the computational performance is analyzed. After this, possible optimizations are suggested, implemented, and evaluated. Finally, the improvements are summarized.

6.1 HexPlane

HexPlane is profiled on the available hardware, as described in section 4.5. All tests were run using Python 3.8.19 and PyTorch 1.12.1. The default `dnerf_slim.yaml` configuration is used to train the model. However, the number of training iterations is reduced to 250. This reduction does not have any influence on the measured computational performance of the model, as the training process is static and does not change over the training process.

Performance metrics are collected during training after a warm-up phase. Performance was measured while training because training includes the rendering process. Thus, inefficiencies during the rendering process still appear during training.

To actually collect performance data, first, the training process runs for ten iterations without the profiler being active. This is followed by five iterations where the profiler is active but discards measured performance data. Finally, performance metrics are collected during a single iteration.

6.1.1 Performance Analysis

The training process is measured in the following five sections, already described above, namely data sampling, rendering, loss, optimization, and statistics. In this iteration, no expensive computations are done in the statistics section, such as rendering a progress image, updating the progress bar, upsampling the model, or updating the emptiness voxel. Upsampling and updating the emptiness voxel are rare events, usually only happening three times during full training. Therefore, they do not represent the average training iteration well and are not chosen for profiling.

The rendering of an image as well as updating the progress bar are optional. They are only used to share information with the users. When concerned with speed, one would reduce these to a minimum. In consequence, they are not chosen for analysis.

In table 6.1 the time each section takes is shown, including waiting for all kernels to have finished execution by calling `torch.cuda.synchronize`. It is apparent that the total CPU time recorded by the PyTorch profiler is not equal to the sum of all five sections named before, even though there are no extra commands or function calls inside the section being profiled. However, this is likely either due to the implementation of the profiler or due to some intricacies of the Python programming language, such as being an interpreted language, causing slight delays.

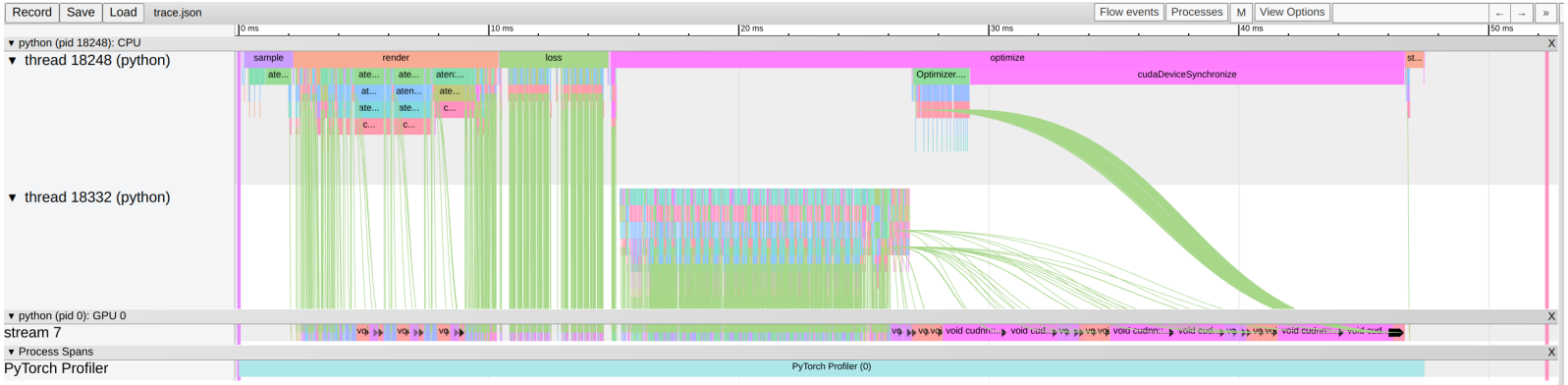

Figure 6.1: GPU trace of a single iteration of HexPlane. The green lines show the association between the launch of a kernel on the CPU and the time of actual execution of the kernel on the GPU. Here, the backpropagation takes the majority of the time, due to the expensive calculations to calculate the gradients of the `grid_sample` function, which alone take roughly 50% of the time of the entire iteration.

	CPU time	CPU time %
Total time	47.426 ms	100.00 %
Data sampling	1.958 ms	4.13 %
Rendering	8.195 ms	17.28 %
Loss	4.365 ms	9.20 %
Optimization	31.775 ms	67.00 %
Statistics	0.731 ms	1.54 %

Table 6.1: Time spent in each section in a single training iteration of HexPlane. At the end of each section, synchronization is enforced through the `torch.cuda.synchronize` function.

Looking at the actual durations of each section, it is obvious that the optimization and the rendering steps are the two most computationally intensive tasks, with the optimization step taking almost two-thirds of the total time. The data sampling and the statistics section are almost irrelevant, making up barely more than five percent together.

Checking the trace of a single iteration in fig. 6.1 reveals that the majority of the time is spent in the optimization step. However, a close look shows that the majority of this step is actually spent in the backward pass of the `grid_sample` function. This is visible in the trace by the three sets of long kernels in the render pass, which take roughly 4.8 ms. Three corresponding sets of kernels can be seen during the backward pass in the optimize section, which take almost 17 ms. Together, the `grid_sample` function makes up almost 50% of the time of a single iteration. All these kernels are limited by the GPU.

In contrast to the `grid_sample` function, the loss calculation is mostly limited by the CPU. In the provided implementation, a majority of the loss calculation features loops and operations on small tensors. As PyTorch queues a kernel for each operation, the small overhead from launching a kernel adds up, which limits the speed of this section. This can be seen in the trace as well, where the loss calculation in the forward and backward pass together takes around 15 ms, a third of the total time.

The small MLPs employed by HexPlane take in comparison almost no time, mainly due to their small size.

The remaining time is spent in the sampling process, the evaluation and statistics, as well as some preparations in the rendering process. The timings measured by the profiler also show a significant time for functions such as `aten:to`, `aten::copy_`, `aten::zeros`, and `aten::fill_`, which are memory allocation operations or operations that transfer memory between

CPU memory and GPU memory. However, inspecting the trace closer indicates that these functions often require more time because they have to wait for the results of previously queued kernels instead of taking the time for themselves.

6.1.2 Possible Optimizations

A large part of HexPlane relies on very efficiently implemented functions in PyTorch, such as `torch.nn.functional.grid_sample`, and neural networks, such as MLPs.

A possible optimization for the `grid_sample` function would be to look at the cache usage and input data. For example, one could sort the list of coordinates where to sample in such a way that the relevant data of the grid in cache is used multiple times, reducing the amount of data fetched from slower caches. On the other hand, sorting also takes a significant amount of computing power. When attempting this, one has to consider the added time of the sorting in contrast to a possible time reduction of the `grid_sample` kernel, especially in the backward kernel, as the majority of the time is spent in it. Attempting this optimization promises very little reward for a large amount of effort, which is why this is not attempted in this project.

A simpler way of improving the speed of HexPlane would be to make architectural changes to reduce the computational demands of the model. An example of this could be to not use the emptiness voxel, which saves one out of three expensive `grid_sample()` calls. However, the purpose of the emptiness voxel is to skip evaluations of regions in empty space, which saves time. Again, one has to compare these two aspects when making architectural changes. One also has to ensure the quality of HexPlane stays the same with these changes. This requires extensive validation, which is why architectural changes are not in the scope of this project.

This leaves the loss calculation, which is CPU limited, an indication that the available resources are not used to their full extent. The loss calculation relies on a high amount of basic tensor operations, implemented by PyTorch. Each operation adds a small overhead when the respective kernel for that operation is launched. To reduce this overhead, one could combine them into a single custom CUDA kernel, which is attempted for the larger model DynIBaR in section 6.2.2.

HexPlane's implementation also features heavy use of tensor concatenations and accesses of tensors using a mask. Usually, PyTorch simply uses new views, which redirect accesses to the tensor to the specific positions of

the data in memory. This minimizes the need to allocate and copy tensors combined using the e.g., `torch.cat` function. However, this also creates discontiguous tensors, which are not suited as parameters for kernels, so they have to be made contiguous again before being passed as a parameter. This necessitates the allocation of new memory and the copying of all tensors that were combined into the new memory.

For small tensors, these operations might not add significant overhead, but HexPlane features six large 2D planes that are stored in a tensor. These are combined into a single entry prior to being passed to the `grid_sample` function using the `torch.cat` function. This necessitates the rearrangement of the data of these tensors in memory, as outlined above. To prevent this issue, a developer could create these 2D planes initially as a 3D tensor, featuring six 2D planes in an array. This would ensure that the data is contiguous in memory. The individual planes could then be saved as a view, which simply changes the bounds this tensor is accessed for each plane and prevents reorganization of the memory. This could then increase the speed of the `grid_sample` function when the stacked planes are used as an input, as the memory is already contiguous.

6.1.3 Conclusion

All in all, HexPlane is already a small and efficient model, benefitting heavily from its simple architecture. As seen in section 6.1.1, a large portion of the time is spent in various functions implemented efficiently by PyTorch.

There is some potential in optimizations, such as sorting the sampling coordinates to improve cache locality or creating internal data structures in a contiguous way. However, they are offset by the increased cost of the sorting operation or might not increase efficiency. This could be due to optimizations for masked tensor accesses already in place and PyTorch simply keeping the memory of the data structures already internally contiguous for future iterations. One optimization, writing a custom kernel to consolidate operations for the loss calculation, will be described in section 6.2.2.

6.2 DynIBaR

DynIBaR is a very computationally and memory-intensive model, which is usually run on multiple NVIDIA A100 GPUs. In this project, only an NVIDIA RTX3070 Ti Laptop GPU is available, as described in section 4.5. The RTX 3070 Ti is obviously vastly outperformed by multiple GPUs specifically designed for developing AI models. The most important difference is that the available GPU only features 8 GB of dedicated video card memory instead of the 80 GB offered by an A100. This necessitates adjustments to the following two training parameters:

- Batch size: The batch size needs to be reduced greatly so that the current data batch, in addition to the model, fits into the much smaller dedicated GPU memory. Reducing the batch size could affect performance, making overheads in various operations more pronounced.
- Number of training iterations: Due to the reduced amount of computing power available, the number of training iterations has to be reduced massively. However, this does not affect performance, as each iteration follows the same optimization process independent of how often it was already run.

A smaller batch size could affect model performance since smaller batches lead to less efficient kernel execution due to a higher relative overhead of launching and managing the kernels. This change could also lead to underutilization of the GPU, as it might lack the data necessary to fully saturate its parallel processing capabilities. The model can not even perform multiple steps in parallel, as the individual steps usually depend on the current step's results.

It is also remarkable that a solution to performance bottlenecks in this analysis could be hardware specific. Adjustments on the available hardware may lead to increased performance, but the same adjustments could have less—or even detrimental—effect when using the recommended hardware configuration.

The second parameter that can be greatly reduced is the number of iterations to train the model, as this project is not interested in achieving or validating good results. Instead, it is mainly focused on analyzing the performance of the model and identifying potential improvements. This is possible as the training process is static and does not change over time. Consequently, to get good performance readings, only a short warm-up period

is needed. Performance measurements can take place afterward and the results are representative for longer training runs, as the training algorithm does not change.

Despite these changes, inefficiencies can still be identified, although they might be less relevant when training on the recommended hardware.

From a software perspective, the model is executed using Python 3.8.19, with PyTorch 2.4.0 and CUDA 12.4. Everything else is as described by the authors of DynIBaR.

6.2.1 Performance Analysis

Training the model consists of two loops executed one after another. In the first loop, only a lightweight model is used to initialize the motion mask M_i using just the static RGB loss, which compares the predicted color with the target color. This also means that all the calculations for cross-time rendering are not taking place, as it is only used in the loss function of the main model. This coarse training solely serves the purpose of bootstrapping the model and is only run for a short amount of iterations in the beginning.

The main training loop has similar characteristics but enables all features of the model and computes all losses for optimization. On the other hand, the first loop is simply a subset of operations of the second loop. This report will only benchmark the main training of the model in the second loop.

In general, one training iteration can be divided into five major steps:

1. Sampling: A new batch is being loaded and transferred to GPU memory.
2. Feature: Using the `model.feature_net` MLP, the static and dynamic features from the current and adjacent frames are extracted.
3. Rendering: Using the input data and calculated features, a color for each ray is determined, and cross-time rendering is performed.
4. Loss: All different losses are calculated from the rendering results and combined into a final single loss.
5. Backward: The backpropagation handled by PyTorch takes place, and the weights are updated according to the optimizer.
6. (Optionally) Logging: Periodically a complete image is rendered to log the training progress, and different metrics are written to disk. For optimal performance, this step can be skipped and is not included in this analysis.

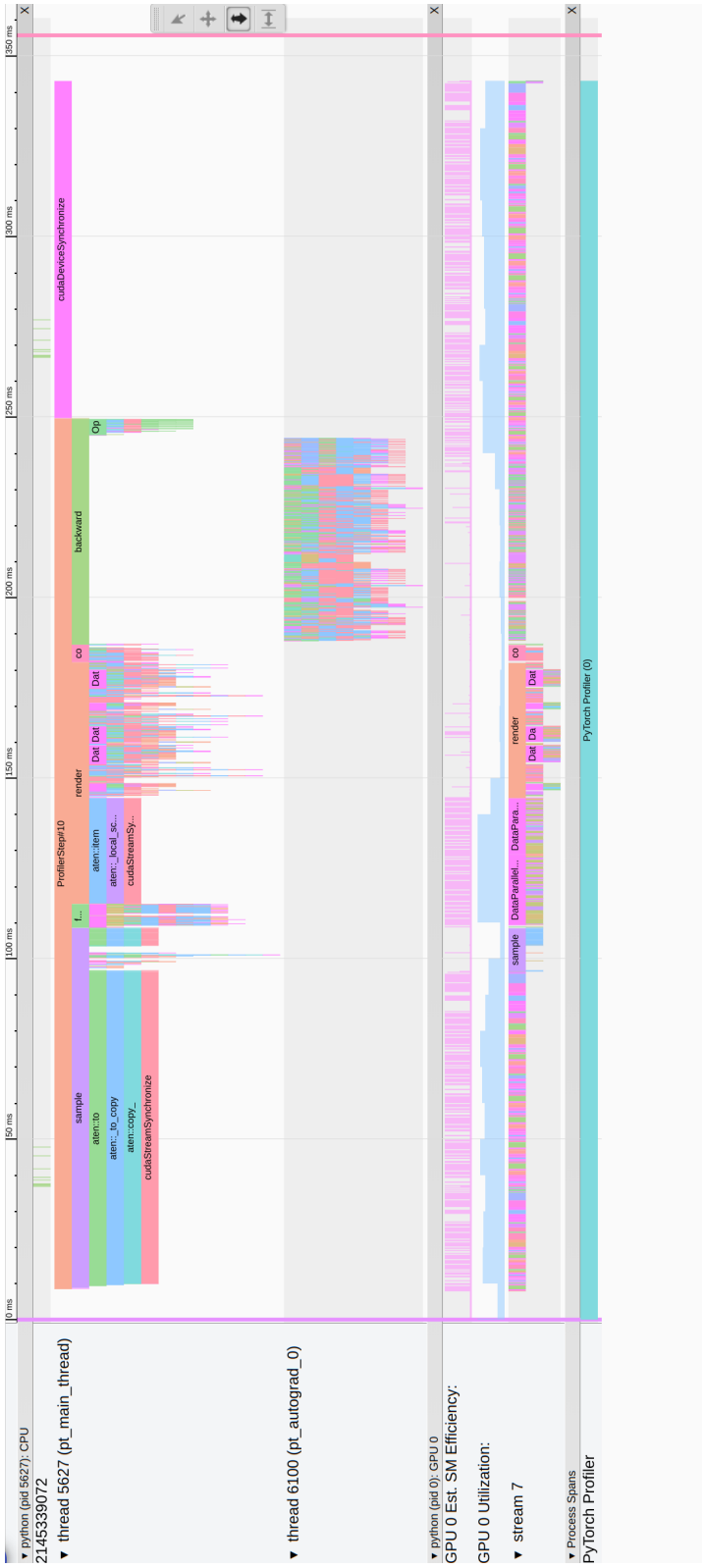

Figure 6.2: Trace of a single iteration during training of DynIBaR with no synchronization. In the beginning, the kernels from the last iteration still need to finish before the next batch is transferred into GPU memory.

Figure 6.3: Trace of a single iteration during training of DynIBaR with a single synchronization at the end of the iteration. This shows the execution of the kernels lagging behind the CPU and that the model is GPU limited at the end of the iteration.

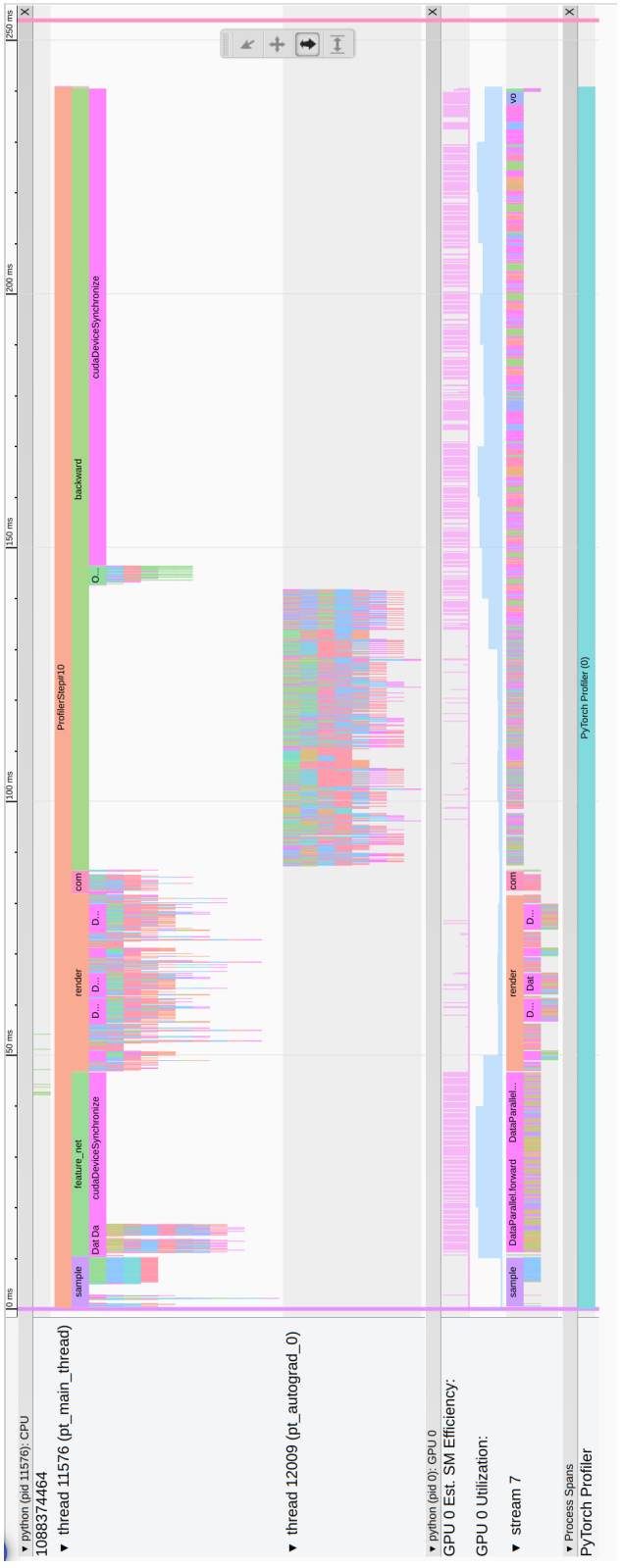

Figure 6.4: Trace of a single iteration during training of DynIBaR with a single synchronization at the end of each major step in a single iteration. Here, it can be seen that the first four steps are not limited by the GPU, as there is no need to wait for the synchronization at each step. This indicates inefficiencies and underutilization of the GPU.

As can be seen in the stack trace depicted in fig. 6.2, some steps are deceptively large, like the sampling step. However, this is due to the last iteration not having finished execution on the GPU, i.e., some kernels are still queued from the last step.

Synchronizing the GPU using `torch.cuda.synchronize()` after each step as seen in fig. 6.4 shows that most time is actually spent computing the gradients through the backpropagation and updating the model's learnable parameters in the `optimizer.step()` function. This takes roughly two-thirds of the entire iteration. Introducing synchronization could lead to less efficient execution. However, the difference in total execution time when only using one synchronization at the end of the loop (46.584ms), compared to synchronizing after every step (46.232ms), is not even one percent.

This indicates that the code before the backward step is either well-balanced or limited by the CPU, the latter hinting at an underutilization of the GPU due to inefficient programming.

This can also be seen when inspecting the GPU utilization, as denoted in fig. 6.4 in the rows "GPU 0 Est. SM Efficiency" and "GPU 0 Utilization." There, only the feature net evaluation, the final part of the backpropagation, and the `optimizer.step()` function achieve high utilization.

For large parts of the backpropagation process and during the entire rendering and loss calculation, the occupancy and utilization of the GPU are near zero, meaning the processing power of the CPU is the limiting factor. As already mentioned, this could be due to the small batch size, which might not take advantage of the high parallel processing capabilities of the GPU.

Similar to HexPlane in the previous section, DynIBaR also utilizes numerous memory operations, such as allocations and copies, which could indicate a potential memory throughput bottleneck. However, since these operations are executed directly on the GPU, such a bottleneck is unlikely, as the GPU would be fully utilized in such a case. It is much more likely that the high amount of operations on small tensors that cannot achieve high occupancy and thus good efficiency are throttling the model. Many of these tensors are too small to fill even a single warp.

When examining specific steps, the sampling step mainly involves memory allocations, memory copies, and setting up the data on the GPU so that it can be used in the subsequent steps. This accounts for approximately 6% of the total iteration time.

The second step uses the `model.feature_net` to calculate input features for the rendering process from the current and nearby frames. For this,

it evaluates the convolutional neural network, which is handled by PyTorch. It takes one-sixth of the total time, achieving high occupancy on the GPU while only requiring minimal CPU time. As shown in fig. 6.4, a significant block of CUDA synchronization indicates that the CPU needs to wait for the GPU to complete its tasks.

Because the rendering step relies on the extracted features, the idle time of the CPU can not be taken advantage of. This is evident in fig. 6.3, where only one synchronization is used at the end of the iteration. Here, the first rendering instruction must wait for the feature extraction to finish, as illustrated by the long wait during `aten::item`, which accesses an extracted feature early in the rendering process.

In contrast to the high GPU utilization observed in the feature extraction step, the render step is mostly limited by the CPU. This limitation arises from the extensive calculations performed on small inputs, which results in a high number of small kernel launches interspersed with some occasional neural networks invocations, namely the G_{MT} MLP.

Examination of these small kernels reveals that they take roughly 30 μs on the CPU, while GPU execution time is only around 2 μs. Consequently, the rendering step does not only achieve low occupancy, due to the small amounts of data the kernels operate on, but also low utilization, as the GPU is mostly idle. This is illustrated in fig. 6.4, where the row ”GPU 0 Utilization” is low, even during the execution of the small neural networks.

The subsequent loss calculation step exhibits similar behavior, achieving only very low streaming multiprocessor efficiency, as can be seen in the row ”GPU 0 Estimated Streaming Multiprocessor Efficiency” in fig. 6.4.

The final major step is backpropagation, implemented using PyTorch’s automatic differentiation engine, starting from the loss. During this step, many small kernels are launched to traverse the computational graph backwards, propagating the gradients to the learnable parameters. High GPU utilization is only achieved at the end of this process when the feature networks are traversed backwards and when the optimizer updates the weights.

This step is bound by both the CPU and GPU at different stages. Initially, the CPU limits performance. But as execution progresses to the feature extractor network and weight updates, many kernels are queued quickly. These kernels take longer to execute, leaving the CPU idle in the final third of this step, waiting for synchronization, as shown in fig. 6.4.

6.2.2 Improvements

As highlighted in the previous analysis, the DynIBaR model faces several bottlenecks. Some, such as loading of the data from disk in the first step, are inherently constrained by hardware limitations. A faster hard drive or a RAM disk, storing the data in memory, could potentially alleviate this issue. However, the data still has to be transferred to GPU memory, which is limited by the memory bandwidth between the CPU and GPU.

If all data could fit into the GPU memory, this step could be significantly accelerated, as it would only involve selecting the batch. On the other hand, the GPU already needs substantial memory capacity, just to hold the model. Even with the recommended NVIDIA A100 Tensor Core GPU, which features 80 GB of memory, the authors of the model still chose to load the data at the start of each iteration from disk rather than keeping the entire dataset in memory.

The second step, feature extraction from the source images, fully saturates the GPU while leaving the CPU mostly idle. This step utilizes a convolutional neural network implemented in PyTorch, which already provides a highly optimized framework for such tasks. Additionally, the large input data ensures full GPU utilization, leaving little room for further and possible improvements.

The next step, the rendering itself, is clearly CPU limited. A part of the trace can be seen in fig. 6.5, where the problem becomes apparent. This section consists of a high amount of small operations, which are all executed on the GPU. But these operations do not actually saturate the GPU. The overhead of queuing up kernels and waiting for them to finish executing is much larger than the actual execution of these kernels. To improve the performance of this step, one can consolidate multiple steps into one custom kernel to reduce the overhead.

An example of this optimization could be the computation of the motion trajectory field. There, the motion coefficients are multiplied on the motion basis. The steps executed to calculate the motion of the points over time using the motion coefficients and the learnable motion basis are depicted in the Python source code in listing 6.1.

This part of the code makes heavy use of slicing and therefore generating views of the underlying tensors. It also concatenates the sum of the coefficients multiplied with the basis in all three directions together. All these operations can be combined into a single kernel, which could look like listing 6.2.

Through clever usage of the size of the thread blocks and the number of threads per block, the slicing can be achieved through those. In this case,

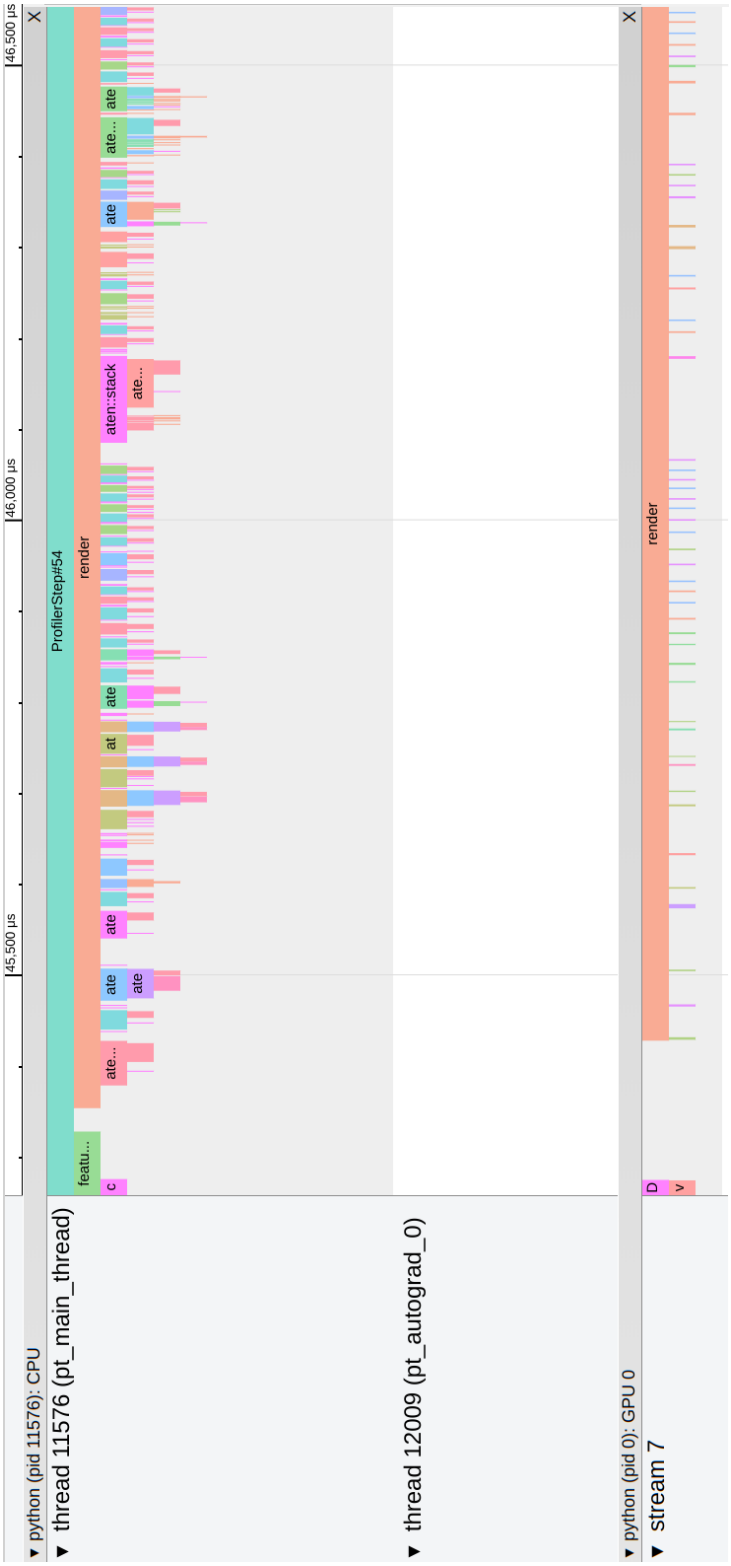

Figure 6.5: Partial trace of the render function in DynIBaR. It becomes obvious that a lot of kernels are launched. However, actual utilization of the GPU is very small, as can be seen in the row ”stream 7.”

```
num_basis = model.trajectory_basis.shape[1]
raw_coeff_x = raw_coeff_xyz[..., 0:num_basis]
raw_coeff_y = raw_coeff_xyz[..., num_basis : num_basis*2]
raw_coeff_z = raw_coeff_xyz[..., num_basis*2 : num_basis*3]

ref_traj_pts_dict = {}
# always use 6 nearby source views for dynamic model.
for offset in [-3, -2, -1, 0, 1, 2, 3]:
  traj_basis_i = model.trajectory_basis[
    None, None, ref_frame_idx+offset, :
  ]
  ref_traj_pts_dict[offset] = torch.cat(
    [
      torch.sum(
        raw_coeff_x*traj_basis_i, axis=-1, keepdim=True
      ),
      torch.sum(
        raw_coeff_y*traj_basis_i, axis=-1, keepdim=True
      ),
      torch.sum(
        raw_coeff_z*traj_basis_i, axis=-1, keepdim=True
      ),
    ],
  dim=-1,
  )
```

Listing 6.1: DynIBaR's original Python code to multiply the motion coefficients to the motion basis to form the motion trajectory. As can be seen, it makes use of splicing and concatenating the results together again afterward, which could be prevented by using a custom kernel.

```
__global__ void motion_kernel(
    const float* __restrict__ trajectory_basis,
    const float* __restrict__ raw_coeff_xyz,
    float* offset_n3,
    float* offset_n2,
    float* offset_n1,
    float* offset_0,
    float* offset_p1,
    float* offset_p2,
    float* offset_p3,
    const int ref_frame_idx
    )
{
    float* offsets[] =
        {offset_n3, offset_n2, offset_n1, offset_0,
         offset_p1, offset_p2, offset_p3};
    float* offset = offsets[blockIdx.x];

    const int ref_idx_offset =
        (ref_frame_idx + blockIdx.x - 3)*6;
    const int base = (blockIdx.y * 128 + threadIdx.x);

    float sum = 0.0f;
    for (int i =0;i<6;i++)
        sum += raw_coeff_xyz[base*18 + threadIdx.y*6 + i]
               * trajectory_basis[ref_idx_offset + i];
    offset[base*3 + threadIdx.y] = sum;
}
```

Listing 6.2: The same code as in listing 6.1 implemented as a single efficient forward CUDA kernel.

for example, the `blocksize.x` is defined by the size of the neighborhood, which is seven by default. This allows assigning the results to the correct array by using `blocksize.x` as an index, which are combined into the `dict` in the wrapper for the kernel. `blocksize.y` and `threadsize.x` are defined by the number of rays to be rendered, i.e., the batch size, and the number of points per ray, respectively. `threadsize.y` is simply set to three for the three directions in 3D space.

Table 6.2 compares the runtime of the original Python implementation and the custom CUDA kernel that consolidates all operations into a single kernel. As shown, reducing the overhead by launching a single kernel instead of multiple significantly enhances performance, with the custom kernel taking less than 20% of the original runtime.

Type	Time (ms)	Time (%)
PyTorch + Python	5.511	100 %
Custom kernel + C++	1.041	18.9 %

Table 6.2: Performance comparison of the original Python implementation and a custom CUDA kernel for calculating the motion trajectory field in DynIBaR. The time displayed comprises both the forward and backward pass.

Even combining just a few operations into a single kernel yields substantial performance improvements. One downside of implementing a custom kernel is that a backward pass must also be provided to be able to use the function in PyTorch's automatic differentiation engine. This can be tricky to implement, especially for complex functions. That said, grouping only a few operations together makes this task relatively straightforward.

An important issue, that should not be forgotten, is that the batch size was drastically lowered in this analysis. It could very well be, that DynIBaR is GPU limited all the time, when the recommended batch size is used. This would diminish the advantage of employing a custom CUDA kernel, but might still lead to a slight improvement.

6.2.3 Conclusion

DynIBaR is a complex model that is often limited by the GPU during training. This indicates it utilizes the GPU well in most areas, but some areas, such as the rendering process, are heavily limited by the CPU.

To address such limitations, the use of custom kernels for a complex function, combining multiple steps into one, can significantly improve efficiency and can reduce the required computing resources. However, an increase in batch size could also address this limitation, but this cannot be verified in this thesis due to the missing hardware.

In spite of that, a well-designed implementation that combines multiple operations into a single kernel can eliminate unnecessary memory allocations and improve throughput by better utilizing the GPU. On the other side, implementing a backward function for such kernels can be challenging.

Reimplementing all of DynIBaR with custom kernels could improve performance massively, but as DynIBaR is a very complex model, it is out of the scope of this project but could be studied in a future project.

6.3 4D Gaussian Splatting

The hardware used to conduct the experiments on the 4DGS model is the same as outlined in section 4.5. In addition, Python 3.11.10 was used, as well as PyTorch 2.5.0 and CUDA 12.5.

The model was run using the unmodified configuration given to train on the bouncing balls scene from the D-NeRF dataset [46].

6.3.1 Performance Analysis

The measured performance of the training varies greatly depending on the amount of Gaussians in the model. As the model learns a good representation of the scene, the amount of Gaussians usually increases quickly at the beginning. Because of this, a single iteration of the early fine training has been selected for an analysis, where the system is already warmed up, but the number of Gaussians is still variable. This means that the impact from the densification process is still visible and not drowned out by a high amount of Gaussians.

In table 6.3, the durations of the individual steps are presented. Additional steps, like network, where data is potentially downloaded from a server, and steps grouped under the statistics step in the previous chapter, are also included. Obviously, some sections are not wanted when chasing performance, like network, progressbar, logging, and saving, and should be turned off for maximal performance. But these steps only account for only

Task	Description	Time (ms)	Percent (%)
Network	Network communication	0.046	0.12 %
Sampling	Sampling the current batch	0.152	0.38 %
Rendering	Synthesizing novel view	6.495	16.36 %
Loss	Calculating all losses	3.080	7.76 %
Backpropagation	Calculating gradients	26.155	65.88 %
Progressbar	Updating the progress bar	0.347	0.87 %
Logging	Save image and metrics	0.308	0.78 %
Densification	Adding and removing Gaussians	1.084	2.73 %
Optimization	Updating weights	1.998	5.04 %
Saving	Saving current model	0.005	0.01 %
Total		39.698	100.00 %

Table 6.3: Section breakdown of a single training iteration in 4DGS. The rendering, loss and backpropagation sections make up for almost 90% of the total time.

about 2% of the total time per iteration, so the potential time savings are marginal.

Much more interesting, however, is the backpropagation section, which makes up a large part of the entire iteration. The largest contributor to it is by far the Gaussian Rasterizer backward kernel, which alone takes 16.1 ms, that is about 40% of the total iteration, as can also be seen in the trace in fig. 6.6.

When looking at the rendering section, the small neural networks to infer the time-dependent deformation also take a significant amount of time, especially on the CPU. However, they are implemented using the tools offered by PyTorch, making them already quite efficient. Due to their small size, they are bottlenecked by the CPU, with each kernel finishing much quicker than the next one being queued. This is similar to the problem solved in section 6.2.2, where the overhead of the kernel is larger than the execution of said kernel.

The forward pass of the Gaussian Rasterizer evokes multiple custom kernels. On the other side, these are not as computationally intensive as the backward kernel and require only slightly over 2 ms of computation time.

Similarly, the loss section consists simply of PyTorch's vector operations, but these are also CPU-bound, indicating inefficient programming as the GPU is not fully utilized. Calculating the loss in a custom kernel that combines these instructions might be more efficient, as has been shown in the previous section 6.2.2.

Almost at the end of the iteration, the densification process splits, clones, or removes malformed Gaussians. This also does not take a large amount of time, only around 2.7% of the total time. The final section optimization takes around 5%, which is again handled by PyTorch's automatic differentiation engine.

All in all, the model is not very complex, the different sections being either simple and efficient or relying on established models, such as HexPlane. Therefore, they do not require a large amount of computational power, or they are already optimized by using a customized kernel.

While the dynamic part of 4DGS is handled mostly by an implementation similar to HexPlane, the custom-written kernel is a target for improvement, which will be attempted in the next section.

However, there is one important thing to keep in mind when attempting to improve the backward kernel of the Gaussian Rasterizer: The CPU only has to wait around 20 ms on it, which means that an improvement reducing computation time by more than 20 ms makes the code limited by the CPU CPU-bound again and would not yield further speed improvements on the test machine.

Figure 6.6: Trace of a single training iteration of 4DGS. Here, it can be seen that the backward Gaussian Rasterizer kernel takes a significant amount of time of the entire iteration and could be a target for optimization.

6.3.2 Improving the Gaussian Rasterizer Backward Kernel

As discussed in the previous section, the most time-consuming task of the training process is the backward pass of the Gaussian Rasterizer. This consists of multiple steps. The most significant step of these is the backward kernel. Its purpose is to calculate the gradients for all the Gaussians depending on the input. The mathematical details of it are thoroughly explained in the appendix of the original paper [25].

To make improvements regarding the backward kernel, tests have to be done in isolation without the rest of the 4DGS model. For that, during training of the full model, the inputs and outputs of the kernel have to be recorded and saved to disk. Then, the kernel can be tested alone using the recorded data as input when running the kernel. It is also important to check for correctness after changes are applied to it.

Here, the kernel is tested by using a Python script, which calls the backward routine of the compiled CUDA kernel. This script is then profiled using either the PyTorch profiler to measure execution times or NVIDIA Nsight Compute to measure various metrics. Performance is recorded after a warm-up phase of ten executions of the kernel, followed by a single execution where data is recorded. This is repeated 30 times to get a good average and to ensure representative data. The standard deviation of the measured metrics over the 30 repeats should be low.

In the original implementation of the kernel, each thread handles a pixel. The image is processed in tiles of 16 by 16 pixels, so that each thread block consists of 256 threads. Each thread block is executed in warps of 32 threads per warp.

The kernel itself first calculates some necessary variables, such as the position of the current pixel and the number of Gaussians affecting it, using information from the forward pass and the position of the current block and thread, as well as initializing variables used for calculating the backward pass. Following that, the main loop of the kernel is started. The main loop consists of two phases.

In the first phase, all threads in the current block work together to copy all necessary information regarding the Gaussians into shared arrays. Ideally, these are located in the fast L1 cache, so that the slow global memory has to be accessed less during the gradient calculation. The original implementation uses multiple shared arrays for ID, position, depth, and conic opacity, which is

just a wrapper for the covariance and opacity of the Gaussians. Before and after populating the shared array, all threads in the thread block are synchronized to ensure that all threads are ready and help create the arrays in the shared memory without accessing wrong or overwriting still relevant data.

The second phase of the main loop then iterates over all relevant Gaussians that affect the thread's pixel and calculates the backward gradients using simple mathematical operations, such as addition and multiplication, as well as one exponent. The resulting gradients are then added to an array using multiple `atomicAdd` instructions.

After all pixels and Gaussians affecting these gradients are processed, the kernel finishes executions, and they are accessible in the array populated by the `atomicAdd` operations.

There are multiple opportunities to optimize the kernel for improved efficiency. The first possible improvement concerns the data in the shared memory. In the original kernel, there are five distinct arrays that hold the Gaussian data in shared memory. As all five variables are necessary to calculate the effects of the currently processed Gaussian, they have to be loaded from five distinct memory positions. As they are stored in separate arrays, this process needs to load multiple cache lines to access the data. An idea for improvement is to group the variables into a single `struct`, storing the attributes of a single Gaussian consecutively in 32 Bytes of memory. Ideally, the data for a Gaussian can then be loaded using only a single cache line for this thread. However, testing this idea only leads to a very small improvement of 1% reduced execution time.

Another possible improvement is to have each thread process multiple pixels—such as four—instead of each thread processing exactly one pixel. This reduces the total number of threads needed by a factor of four. By doing this, some variables can be reused for multiple pixels. The GPU can also be utilized in a better way, as most variables can be stored efficiently as `float4` instead of as a normal `float`.

This way, the compiler can optimize the code better and can make use of vectorized operations, such as parallel loads and stores. But only a few of these operations exist on the GPU, and the parallelism comes mainly from having multiple threads execute the same instruction on different data in parallel.

Another large advantage of each thread handling four pixels is the reduction in `AtomicAdd` instructions, as each thread can accumulate the gradients for four pixels before adding them to the gradient arrays storing the results.

Version	Mean CPU Time (ms)	Mean GPU Time (ms)	Min CPU Time (ms)	Min GPU Time (ms)	CPU σ	GPU σ
original	9.379	12.091	8.805	11.403	0.433	0.578
struct	9.243	11.985	8.704	11.327	0.436	0.573
4 pixel	2.860	3.566	2.525	3.135	0.424	0.555

Table 6.4: Performance results of the backward kernel of the Gaussian Rasterizer comparing CPU and CUDA times across different versions. The original version uses the original kernel, as used in 3DGS [25] and 4DGS [59], the struct version tries to improve memory coalescence by packing the variables of the Gaussians into a single `struct` and the 4 pixel version calculates the gradients of the Gaussians for four pixels in parallel in each kernel instead of each kernel only handling one pixel, resulting in only one fourth of the number of necessary threads.

The runtime for each version of the kernels can be seen in table 6.4. The grouping of the attributes of the Gaussians into a single structure does improve the code only slightly but consistently, showing an improvement of around one percent. This makes sense, as, although all attributes of the Gaussians are needed, they are each only needed once per processed Gaussian, with the computations using these variables taking the major amount of processing time. Improving this single load therefore does not really contribute to a major reduction in computation time.

The second improvement of each thread processing multiple pixels reduces the runtime of the backward kernel significantly, to slightly above 25% of the original computation time. This could indicate that each thread processing four pixels needs roughly the same amount of time as a thread processing one pixel. Therefore, the reduction in processing time can be attributed to the reduction of necessary threads that need to be executed.

Looking at the metrics recorded with NVIDIA Nsight Compute in table 6.5 reveals that the improved kernel is not four times as long when counting SASS instructions, which are comparable to assembly instructions but specific for NVIDIA CUDA kernel binaries. This indicates that the improved kernel can reuse a large amount of instructions. A similar indicator is the maximum registers per thread, which shows that a large amount of variables stored in those registers are used for supporting calculations. Both of these changes suggest that the improved kernel is more efficient, as less non-arithmetic instructions need to be executed per handled pixel.

Metric	Original version	4 pixel version	Change (%)
Block size	256	64	-75.00 %
Registers per thread	64	122	+90.62 %
Duration	5.51 ms	14.66 ms	- 62.42 %
Number of SASS instructions	393	920	+134.10 %
Theoretical Occupancy	66.67 %	29.17 %	-56.25 %
Achieved Occupancy	65.49 %	25.74 %	-60.70 %
Shared memory per block	11.26 Kbyte	12.29 Kbyte	+9.09 %
Shared memory config size	65.54 Kbyte	102.40 Kbyte	+56.50%
L1 Sector requests	2,492,294	1,032,469	-58.57 %
L1 Sectors transferred	8,311,784	8,744,291	+5.20 %

Table 6.5: Selected performance metrics measured by NVIDIA Nsight Compute for the original Gaussian Rasterizer backward kernel and the improved version that handles four pixels per thread. The difference in runtime compared to table 6.4 can be explained to be the usage of NVIDIA Nsight Compute adding an overhead to the duration.

Similarly, the number of requested sectors decreased massively, however the actual amount of sectors transferred increased slightly. The reason for this is that each kernel needs to request less sectors, as they can be requested together. At the same time, the higher number of necessary transfers leaves potential room for further improvements, as the memory has not been optimized to support the handling of four pixels per thread.

Another interesting finding is that the amount of available shared memory increased. This could indicate that the available hardware does simply not have enough memory for a kernel where each thread handles only one pixel. But due to the lower number of threads per block the actual memory per thread could be increased leading to efficiency gains.

Finally, the achieved occupancy is close to the theoretical occupancy. This indicates good utilization of the GPU. The improved version has slightly lower achieved occupancy compared to the theoretical occupancy, but that can be explained due to higher branching of the kernel as each of the four pixel can cause a branch in each thread as opposed to just one.

An interesting observation while testing the backward kernel is, that execution of the kernel is not deterministic. The results of the original kernel only match the expected results closely in around two-thirds of the runs. Exact matches could not be reproduced in either case, neither in the original version nor in any improved kernel. Using the first improved version, where Gaussian data is grouped together, the output matches in around 80% of the runs, while

in the version that processes four pixels at the same time, it matches the expected output almost 97% of the time.

To validate the results of the improved kernel, it is used to train the entire model. For this, it is compiled into a Python module and replaces the original implementation of the Gaussian Rasterizer from 3DGS.

Full training of the model using the original kernel takes around eight and a half minutes, while full training using the improved kernel needs only around seven minutes, an improvement of almost 20%.

The model trained with the improved kernel reaches the same level of quality when reconstructing views, as indicated by a visual comparison of the resulting pictures and by similar metrics, such as the L1 loss and the peak signal-to-noise ratio. There are only slight differences due to the aforementioned changes in the output of the kernel. This also indicates that the original kernel's inconsistent outputs are not a necessary feature when training the model.

As mentioned before, the massive improvement to the backward kernels runtime is not fully reflected in the almost 20% improvement in computation time when training the full model. This is due to the training now being bound by the CPU of the machine the model was trained on.

6.3.3 Conclusion

4DGS [59] is not a complex model that relies on a complex custom kernel inherited from 3DGS [25]. The extension to dynamic scenes is done fairly efficiently, as it relies on another model's efficient architecture, namely HexPlane [9].

An analysis of the model revealed that the majority of training time is consumed by the backward kernel of the Gaussian Rasterizer. Further investigation into this kernel identified multiple opportunities for optimization. Implementing these improvements resulted in a 75% reduction in the runtime of the backward kernel, which in turn lead to an almost 20% decrease in the overall training time of the model.

The model still holds potential for further optimization. One promising direction is the optimization of the loss calculation, which could be done in a custom CUDA kernel. Similar improvements, as discussed in section 6.2.2, suggest that this approach could yield another significant reduction in training time and could be a direction for future research.

6.4 Summary

The analysis of the three distinct model have revealed interesting properties when relying on a framework like PyTorch. Certain features and functions, such as neural networks and operations on structured data, like bilinear interpolation on grids, are useful and well implemented. They can be used without spending much thought on the underlying implementation, provided they operate on sufficiently large data. However, due to the necessity to execute a kernel for every operation involving tensors that are stored on the GPU, large amounts of these operations introduce a significant overhead, reducing the efficiency. In these cases it was demonstrated that combining these operations into a single custom kernel improves performance greatly. But when writing large custom kernels special care has to be taken, as performance there can still not be optimal, as seen on the complex backward kernel for Gaussian rendering. There, despite following best practices when necessary, small changes in the organization and the grid layout lead to remarkable improvements on the available hardware. This shows that notable performance improvements are still possible in modern machine learning models for NVS.

Chapter 7

Conclusion and Future Works

This concluding section summarizes the key insights gained from this project. It also highlights its limitations and discusses potential extensions and future directions. Finally, the real-world applicability and implications of the research are reflected upon.

7.1 Conclusions

This thesis examined various models used for novel view synthesis for dynamic scenes. Three of those were selected for detailed analysis, resulting in achieving significant improvements for two of them, while possible enhancements for the third have been discussed.

The research showed that utilizing functions from frameworks like PyTorch on data structures or implementing small neural networks, results in highly efficient models. But oftentimes researchers also need to use large amounts of operations on tensors directly, for example to implement complex loss functions or rendering algorithms. These are often inefficient and limited by the CPU, not fully utilizing the available resources on the GPU, despite the optimization of frameworks like PyTorch. This highlights the importance of adhering to best practices when developing code that runs on the GPU.

The high level of abstraction offered by PyTorch, while convenient, can inadvertently lead to inefficient implementations and slower models. Nevertheless, tools offered by frameworks like PyTorch are essential for allowing a broad community of scientists to apply their knowledge to experiment, to innovate, and to advance their respective fields. Even an initially inefficient implementation can often serve as a stepping stone for more advanced and optimized approaches, as exemplified by NeRF's novel approach

to NVS in 2020.

By highlighting bottlenecks and improving prominent NVS models, this project has successfully achieved the goals. It underlines the importance of specialized hardware knowledge and the knowledge of best practices when working with it. An important insight is, as conjectured, that many researchers do not prioritize an optimized approach in their models but focus instead on novel ideas. Follow-up work occasionally addresses these inefficiencies. However, such efforts remain rare, especially as the field of NVS advances very quickly. In most cases, the core contribution of new projects lies in the innovation to solve a specific problem, with a provided implementation only serving as a proof of concept. However, in NVS, the longtime goal of achieving real-time rendering and training performance, researchers must be aware of custom, hardware-specific implementations, instead of relying solely on frameworks like PyTorch.

Reflecting on this work, a future direction could be to solely focus on a single project, such as DynIBaR. It provides ample opportunities to address bottlenecks in various components including neural networks and implementing custom code for, e.g., learnable bases or complex loss functions.

7.2 Limitations

The primary limitation of this project was the available hardware, which fell short of being able to run complex models—particularly DynIBaR—at their intended configuration. This constraint potentially skewed results, as improvements might only be viable for low-end systems and not scale to high-performance computing environments.

While it would have been possible to dissect DynIBaR and to test individual components in synthetic environments that provide enough data to saturate the GPU, this approach was not pursued due to the limited time available and the focus on three models. The chosen methodology has still yielded valuable insights, even if some might not apply directly to DynIBaR. Researchers can still apply these to other projects with data-lean pipelines that do not fully saturate GPU resources.

However, this problem of not fully saturating a GPU becomes increasingly relevant because of the rapid development of AI accelerators. For example, new hardware is set to be released in early 2025 by NVIDIA, projected to deliver more than double the processing power and memory speed of current top-of-the-line models.

7.3 Future work

Future work could focus on extensively improving only a single complex project, such as DynIBaR, in order to see how much the rendering and training speed can be improved upon.

Additionally, several possible improvements have been identified in this project but could not be tested due to time constraints. They provide a promising direction for further research. On the other hand, some of these improvements are model specific. Therefore, they do not offer as much value as the general improvements as those shown in this project.

7.4 Reflections

Helping to advance the field of AI research, especially when increasing its efficiency, has significant real-world implications. For instance, it is extremely relevant for energy consumption, as AI research is a particularly power-hungry field. AI technology is projected to reach global energy consumption that equals that of a large country like Sweden or Germany in the year of 2026, according to an analysis of the International Energy Agency in 2024 [23]. Raising awareness about efficient implementation practices can directly combat the climate crisis by reducing the computational power—and therefore also the energy demand—necessary when training and deploying AI models. Especially as the research done in this thesis can also be applied to other AI models.

Moreover, accelerating training and rendering models for NVS opens up applications in high-fidelity virtual and augmented reality environments and enables autonomous systems, such as self-driving cars, to develop a robust understanding of their environment. However, ethical considerations and risks might become relevant with advanced technology as well.

Improving NVS models can facilitate the creation of images depicting real individuals in misleading or harmful situations, such as deepfakes. Therefore, ensuring the responsible usage of such technologies is crucial. While researchers need to account for these ethical considerations, it is ultimately the responsibility of today's policymakers to establish clear regulations in order to prevent the misuse of these technologies without restricting innovations in the field.

References

[1] J. Ansel *et al.*, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf.

[2] A. Ashari *et al.*, "On optimizing machine learning workloads via kernel fusion," *SIGPLAN Not.*, vol. 50, no. 8, pp. 173–182, Jan. 2015, ISSN: 0362-1340. DOI: 10.1145/2858788.2688521. [Online]. Available: https://doi.org/10.1145/2858788.2688521.

[3] J. T. Barron, B. Mildenhall, M. Tancik, P. Hedman, R. Martin-Brualla, and P. P. Srinivasan, "Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields," *ICCV*, 2021.

[4] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman, "Mip-nerf 360: Unbounded anti-aliased neural radiance fields," *CVPR*, 2022.

[5] R. Basri and D. W. Jacobs, "Lambertian reflectance and linear subspaces," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 25, no. 2, pp. 218–233, Feb. 2003, ISSN: 0162-8828. DOI: 10.1109/TPAMI.2003.1177153. [Online]. Available: https://doi.org/10.1109/TPAMI.2003.1177153.

[6] J. Bradbury *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.3.13, 2018. [Online]. Available: http://github.com/jax-ml/jax.

[7] Y. Cai, J. Wang, A. Yuille, Z. Zhou, and A. Wang, "Structure-aware sparse-view x-ray 3d reconstruction," in *CVPR*, 2024.

[8] Y. Cai *et al.*, "Radiative gaussian splatting for efficient x-ray novel view synthesis," in *ECCV*, 2024.

[9] A. Cao and J. Johnson, *Hexplane: A fast representation for dynamic scenes*, 2023. arXiv: 2301.09632 [cs.CV].

[10] P. Charbonnier, L. Blanc-Féraud, G. Aubert, and M. Barlaud, "Two deterministic half-quadratic regularization algorithms for computed imaging," *Proceedings of 1st International Conference on Image Processing*, vol. 2, 168–172 vol.2, 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:38030033.

[11] N. Corporation, "Nvidia ada lovelace architecture," NVIDIA, Tech. Rep., 2022. [Online]. Available: https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf.

[12] P. E. Debevec, C. J. Taylor, and J. Malik, "Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '96, New York, NY, USA: Association for Computing Machinery, 1996, pp. 11–20, ISBN: 0897917464. DOI: 10.1145/237170.237191. [Online]. Available: https://doi.org/10.1145/237170.237191.

[13] B. Deng, J. T. Barron, and P. P. Srinivasan, *JaxNeRF: An efficient JAX implementation of NeRF*, version 0.0, 2020. [Online]. Available: https://github.com/google-research/google-research/tree/master/jaxnerf.

[14] J. Fang *et al.*, "Fast dynamic radiance fields with time-aware neural voxels," in *SIGGRAPH Asia 2022 Conference Papers*, ser. SA '22, ACM, Nov. 2022. DOI: 10.1145/3550469.3555383. [Online]. Available: http://dx.doi.org/10.1145/3550469.3555383.

[15] S. Fridovich-Keil, G. Meanti, F. Warburg, B. Recht, and A. Kanazawa, *K-planes: Explicit radiance fields in space, time, and appearance*, 2023. arXiv: 2301.10241 [cs.CV].

[16] R. Frostig, M. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," 2018. [Online]. Available: https://mlsys.org/Conferences/doc/2018/146.pdf.

[17] S. J. Garbin, M. Kowalski, M. Johnson, J. Shotton, and J. Valentin, " FastNeRF: High-Fidelity Neural Rendering at 200FPS," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2021, pp. 14 326–14 335. DOI: 10.1109/ICCV48922.2021.01408. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCV48922.2021.01408.

[18] I. Goodfellow *et al.*, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27, Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf.

[19] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen, "The lumigraph," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '96, New York, NY, USA: Association for Computing Machinery, 1996, pp. 43–54, ISBN: 0897917464. DOI: 10.1145/237170.237200. [Online]. Available: https://doi.org/10.1145/237170.237200.

[20] X. Guo *et al.*, "Forward flow for novel view synthesis of dynamic scenes," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, Oct. 2023, pp. 16 022–16 033.

[21] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd. Cambridge University Press, 2003, ISBN: 9780521540513.

[22] P. Hedman, P. P. Srinivasan, B. Mildenhall, J. T. Barron, and P. Debevec, "Baking neural radiance fields for real-time view synthesis," *ICCV*, 2021.

[23] International Energy Agency (IEA), *Electricity 2024*, Licence: CC BY 4.0, Paris, 2024. [Online]. Available: https://www.iea.org/reports/electricity-2024.

[24] J. T. Kajiya and B. P. Von Herzen, "Ray tracing volume densities," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 165–174, Jan. 1984, ISSN: 0097-8930. DOI: 10.1145/964965.808594. [Online]. Available: https://doi.org/10.1145/964965.808594.

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

[26] M. Levoy and P. Hanrahan, "Light field rendering," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '96, New York, NY, USA: Association for Computing Machinery, 1996, pp. 31–42, ISBN: 0897917464. DOI: 10.1145/237170.237199. [Online]. Available: https://doi.org/10.1145/237170.237199.

[27] T. Li *et al.*, "Neural 3d video synthesis," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2022.

[28] Z. Li, S. Niklaus, N. Snavely, and O. Wang, "Neural scene flow fields for space-time view synthesis of dynamic scenes," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2021, pp. 6498–6508.

[29] Z. Li and N. Snavely, "Megadepth: Learning single-view depth prediction from internet photos," in *Computer Vision and Pattern Recognition (CVPR)*, 2018.

[30] Z. Li, Q. Wang, F. Cole, R. Tucker, and N. Snavely, "Dynibar: Neural dynamic image-based rendering," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

[31] Z. Li *et al.*, "Learning the depths of moving people by watching frozen people," *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4516–4525, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:131775632.

[32] L. Liu, J. Gu, K. Z. Lin, T.-S. Chua, and C. Theobalt, "Neural sparse voxel fields," *NeurIPS*, 2020.

[33] S. Lombardi, T. Simon, J. Saragih, G. Schwartz, A. Lehrmann, and Y. Sheikh, "Neural volumes: Learning dynamic renderable volumes from images," *ACM Trans. Graph.*, vol. 38, no. 4, Jul. 2019, ISSN: 0730-0301. DOI: 10.1145/3306346.3323020. [Online]. Available: https://doi.org/10.1145/3306346.3323020.

[34] H. C. Longuet-Higgins, "A computer algorithm for reconstructing a scene from two projections," *Nature*, vol. 293, pp. 133–135, 1981. [Online]. Available: https://api.semanticscholar.org/CorpusID:4327732.

[35] J. Luiten, G. Kopanas, B. Leibe, and D. Ramanan, *Dynamic 3d gaussians: Tracking by persistent dynamic view synthesis*, 2023. arXiv: 2308.09713 [cs.CV].

[36] Martín Abadi *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[37] N. Max, "Optical models for direct volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, 1995. DOI: 10.1109/2945.468400.

[38] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, *Nerf: Representing scenes as neural radiance fields for view synthesis*, 2020. arXiv: 2003.08934 [cs.CV].

[39] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *ACM Transactions on Graphics*, vol. 41, no. 4, pp. 1–15, Jul. 2022, ISSN: 1557-7368. DOI: 10.1145/3528223.3530127. [Online]. Available: http://dx.doi.org/10.1145/3528223.3530127.

[40] T. Neff *et al.*, "DONeRF: Towards Real-Time Rendering of Compact Neural Radiance Fields using Depth Oracle Networks," *Computer Graphics Forum*, vol. 40, no. 4, 2021, ISSN: 1467-8659. DOI: 10.1111/cgf.14340. [Online]. Available: https://doi.org/10.1111/cgf.14340.

[41] NVIDIA, *Using nsight compute to inspect your kernels*, https://developer.nvidia.com/blog/using-nsight-compute-to-inspect-your-kernels/, Accessed: 2024-12-10, Sep. 2019.

[42] B. Park and C. Kim, "Point-dynrf: Point-based dynamic radiance fields from a monocular video," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, Jan. 2024, pp. 3171–3181.

[43] K. Park *et al.*, "Hypernerf: A higher-dimensional representation for topologically varying neural radiance fields," *ACM Trans. Graph.*, vol. 40, no. 6, Dec. 2021.

[44] K. Park *et al.*, "Nerfies: Deformable neural radiance fields," *ICCV*, 2021.

[45] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[46] A. Pumarola, E. Corona, G. Pons-Moll, and F. Moreno-Noguer, "D-NeRF: Neural Radiance Fields for Dynamic Scenes," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[47] N. Rahaman *et al.*, "On the spectral bias of neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, PMLR, Jun. 2019, pp. 5301–5310. [Online]. Available: https://proceedings.mlr.press/v97/rahaman19a.html.

[48] R. Ramamoorthi and P. Hanrahan, "On the relationship between radiance and irradiance: Determining the illumination from images of a convex lambertian object," *Journal of the Optical Society of America A*, vol. 18, pp. 2448–2459, Oct. 2001. DOI: 10.1364/JOSAA.18.002448.

[49] R. Ranftl, K. Lasinger, D. Hafner, K. Schindler, and V. Koltun, " Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-Shot Cross-Dataset Transfer," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, vol. 44, no. 03, pp. 1623–1637, Mar. 2022, ISSN: 1939-3539. DOI: 10.1109/TPAMI.2020.3019967. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TPAMI.2020.3019967.

[50] A. Rivers, F. Durand, and T. Igarashi, "3d modeling with silhouettes," *ACM Trans. Graph.*, vol. 29, no. 4, Jul. 2010, ISSN: 0730-0301. DOI: 10.1145/1778765.1778846. [Online]. Available: https://doi.org/10.1145/1778765.1778846.

[51] Sara Fridovich-Keil and Alex Yu, M. Tancik, Q. Chen, B. Recht, and A. Kanazawa, "Plenoxels: Radiance fields without neural networks," in *CVPR*, 2022.

[52] J. L. Schönberger and J.-M. Frahm, "Structure-from-motion revisited," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4104–4113.

[53] J. L. Schönberger, E. Zheng, M. Pollefeys, and J.-M. Frahm, "Pixelwise view selection for unstructured multi-view stereo," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016, pp. 501–518.

[54] P.-P. Sloan, J. Kautz, and J. Snyder, "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 527–536, Jul. 2002, ISSN: 0730-0301. DOI: 10.1145/566654.566612. [Online]. Available: https://doi.org/10.1145/566654.566612.

[55] C. Sun, M. Sun, and H. Chen, "Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction," in *CVPR*, 2022.

[56] C. Sun, M. Sun, and H.-T. Chen, *Improved direct voxel grid optimization for radiance fields reconstruction*, 2022. arXiv: 2206.05085 [cs.GR]. [Online]. Available: https://arxiv.org/abs/2206.05085.

[57] F. Wang, Z. Chen, G. Wang, Y. Song, and H. Liu, *Masked space-time hash encoding for efficient dynamic scene reconstruction*, 2023. arXiv: 2310.17527 [cs.CV].

[58] N. Wang, Y. Zhang, Z. Li, Y. Fu, W. Liu, and Y.-G. Jiang, "Pixel2mesh: Generating 3d mesh models from single rgb images," in *Proceedings of the European Conference on Computer Vision (ECCV)*, Sep. 2018.

[59] G. Wu *et al.*, *4d gaussian splatting for real-time dynamic scene rendering*, 2023. arXiv: 2310.08528 [cs.CV].

[60] J. Wu, C. Zhang, T. Xue, W. T. Freeman, and J. B. Tenenbaum, "Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16, Barcelona, Spain: Curran Associates Inc., 2016, pp. 82–90, ISBN: 9781510838819.

[61] Q. Xu *et al.*, "Point-nerf: Point-based neural radiance fields," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 5438–5448.

[62] Z. Yang, X. Gao, W. Zhou, S. Jiao, Y. Zhang, and X. Jin, "Deformable 3d gaussians for high-fidelity monocular dynamic scene reconstruction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 20 331–20 341.

[63] A. Yu, R. Li, M. Tancik, H. Li, R. Ng, and A. Kanazawa, "PlenOctrees for real-time rendering of neural radiance fields," in *ICCV*, 2021.

[64] K. Zhang, G. Riegler, N. Snavely, and V. Koltun, "Nerf++: Analyzing and improving neural radiance fields," *arXiv:2010.07492*, 2020.

[65] H. Zhao *et al.*, "Tacker: Tensor-cuda core kernel fusion for improving the gpu utilization while ensuring qos," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 800–813. DOI: 10.1109/HPCA53966.2022.00064.

TRITA-EECS-EX-2025:93
Stockholm, Sweden 2025