POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Politecnico di Torino

Master's Degree Thesis

Optimization of Variational Bayes Gaussian Splatting Algorithms on Embedded GPU

Supervisors

Prof. Alessio BURRELLO

Prof. Daniele Jahier PAGLIARI

Dr. Matteo RISSO

Dr. Miguel DE PRADO

Dr. Toon VAN DE MAELE

Candidate

Iván ZAINO

S324565

Academic Year 2024-2025

Abstract

The deployment of demanding computer vision algorithms on embedded devices is a complicated challenge due to limited memory, computational capabilities and energy constraints. In many low-latency robotic and autonomous applications, edge deployment is indispensable, as real-time response cannot rely on remote servers.
This work focuses on the optimization of the Variational Bayes Gaussian Splatting algorithm, a state-of-the-art probabilistic 3D scene modelling method originally developed for server-grade GPUs with more than 20 GBs of RAM and not previously implemented on edge platforms. We target the NVIDIA Jetson Orin Nano, an embedded GPU platform with just 8 GBs of RAM.
Rather than modifying the mathematical model itself, this work aims at optimizing the existing implementation to meet the edge devices limitations and improve the overall latency.
The implementation is based on JAX, a high-performance numerical computing library that enables GPU acceleration and just-in-time (JIT) compilation. JIT compilation proved to be a pivotal optimization tool: by staging large portions of the computation into fused, statically optimized kernels, we significantly reduced Python-level overhead and improved data throughput on the Nano's limited GPU.
The optimization process began with an in-depth memory and latency profiling to identify the main computational bottlenecks. The first improvement targeted a suboptimal memory allocation scheme: by avoiding unnecessary intermediate allocations during matrix multiplications, we reduced peak memory usage by up to 75%.
The second major improvement was achieved through data quantization. Since the original algorithm was designed using double-precision (fp64) arithmetic for numerical stability, many operations used excessively high precision relative to their actual requirements, leading to significant overhead in both memory and computation. To address this, we developed an automatic mixed-precision search algorithm. It systematically evaluates each operation and selectively lowers its precision whenever this does not compromise output accuracy. This strategy yielded substantial gains in both training speed and memory efficiency, while maintaining output quality with minimal, and often negligible, loss. Mixed precision proved

pivotal in the optimization process, boosting training speed by a factor of five while preserving output quality.
Importantly, the mixed-precision search algorithm was developed in order to be generic and not tied to VBGS specifically. It provides a systematic framework for identifying precision-sensitive operations and can therefore be applied to a wide range of numerical algorithms. This makes it a valuable tool for future works aiming to deploy computationally demanding models on memory and power constrained devices.

Acknowledgments

I would like to express my deepest gratitude to Prof. Alessio Burrello and Prof. Daniele Jahier Pagliari, Dr. Matteo Risso, Dr. Miguel De Prado, and Dr. Toon Van de Maele for their invaluable guidance, feedback, and continuous support throughout the development of this thesis. Their expertise has been fundamental to shaping both the direction and the depth of my research, and this experience has greatly contributed to my personal growth and professional development.

A mia madre e mio padre
Voglio ringraziare voi che che mi avete sostenuto in ogni momento di questo percorso, con amore, pazienza e fiducia incondizionata. A voi devo e dedico ogni traguardo che ho raggiunto e raggiungerò. Nei vostri occhi vedo che siete fieri della persona che sto diventando; il merito è tutto dei due esempi che mi trovo davanti.

Ai miei amici (di giù)
A voi, che mi conoscete ormai da tempo e mi avete visto cambiare, ringrazio di esserci sempre stati. Si dice che gli amici fatti all'università durino per una vita, ma anche avere amicizie che sono sopravvissute agli studi è un traguardo non da poco.
Voglio ringraziare in particolare Mattia, Marco e Teresa che, nonostante mi vedano una manciata di giorni alla volta, mi fanno sentire come se non me ne fossi mai andato. A Fabio, Marco e Giacomo ringrazio le innumerevoli volte in cui mi hanno dimostrato vicinanza, sostegno ed amicizia sincera durante questi anni.

Ai miei amici (di sù)
Ringrazio voi che avete condiviso con me il bello ed il brutto degli anni da giovane adulto. E' anche grazie a voi che sono riuscito ad arrivare a questa tappa della mia vita e spero, in qualche modo, di aver contribuito in qualche modo alle vostre.
Voglio fare un ringraziamento speciale a Luca, Giacomo, Nicola, Raf, Feno, Guglielmo e Ludovico che, oltre ad aver condiviso lacrime (e, occasionalmente, risate) sui libri, per me sono ormai diventati come una seconda famiglia.

A chi c'è stato

Concludo questa sezione smielata e fuori personaggio con un ringraziamento a chi, in un modo o nell'altro, ha incrociato la sua strada con la mia. Ad ogni persona che è stata nella mia vita e ne ha modificato la traiettoria, anche se solo di poco, voglio dire un sincero grazie, perchè mi ha portato qui dove sono ora.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

CAVI
Coordinate Ascent Variational Inference

DNN
Deep Neural Network

ELBO
Evidence Lower Bound

EM
Expectation–Maximization

GS
Gaussian Splatting

GPU
Graphical Processing Unit

JIT
Just in time

JSON
JavaScript Object Notation

KL
Kullback-Liebler

LSE
Least Square Estimation

MSE

Mean Squared Error

ML

Machine Learning

MLE

Maximum Likelihood Estimation

NeRF

Neural Radiance Fields

NVS

Novel View Syntesis

NIW

Normal-Inverse-Wishart

OOM

Out-of-Memory

PSNR

Peak Signal-to-Noise Ratio

RGB

Red-Blue-Green

SoA

State of the Art

SLAM

Simultaneous localization and mapping

SGD

Stochastic Gradient Descent

SfM

Structure from Motion

VBGS

Variational Bayes Gaussian Splatting

VI

Variational Inference

Chapter 1

Introduction

The ability to model and reconstruct three-dimensional environments from visual data is a fundamental challenge in computer vision and graphics. Accurate 3D representations are essential for a wide range of applications, from immersive media to autonomous systems. In recent years, a class of methods known as *novel view synthesis* (NVS) has emerged as a central solution to this challenge. NVS focuses on generating realistic views of a scene from previously unseen viewpoints, given only a limited set of observations. By learning the underlying structure of the environment, these algorithms enable consistent and photorealistic rendering without requiring full geometric models.

The development of novel view synthesis has undergone rapid evolution. Early approaches were based on geometry-driven methods, where explicit meshes or point clouds were reconstructed and rendered. These techniques offered interpretability but often struggled to reproduce complex lighting effects and fine-grained scene appearance. With the rise of deep learning, the field shifted toward neural implicit representations, the most influential being Neural Radiance Fields (NeRF) [1]. NeRF represents a scene as a continuous volumetric function parameterized by a neural network, mapping spatial coordinates and viewing directions to emitted radiance and density. This formulation enabled unprecedented levels of photorealism and detail in novel view synthesis, setting a new benchmark for fidelity. However, NeRF and its many extensions come at a high computational cost: training requires millions of gradient descent steps, inference is slow, and the reliance on full backpropagation makes continual learning infeasible.

A more recent line of work, *Gaussian Splatting* (GS) [2], represents the scene as a collection of anisotropic Gaussian primitives positioned in 3D space. This formulation provides a balance between efficiency and quality: rendering can be performed by projecting the Gaussians to the image plane and combining them through differentiable rasterization, while training remains tractable through gradient-based optimization. GS significantly reduced both training time and

rendering latency compared to NeRF, making real-time performance feasible on high-end hardware. Nevertheless, gradient-based Gaussian Splatting still inherits key drawbacks of neural optimization, such as catastrophic forgetting when data is streamed sequentially.

To address these limitations, *Variational Bayes Gaussian Splatting* (VBGS) [3] was introduced. Instead of treating Gaussian parameters as deterministic quantities optimized through backpropagation, VBGS frames the problem in a Bayesian setting: each Gaussian primitive is associated with a posterior distribution over its parameters, updated through closed-form rules derived from variational inference. This formulation not only provides uncertainty estimates but also supports continual learning. New data can be integrated incrementally without retraining from scratch, avoiding catastrophic forgetting and making VBGS particularly attractive for real-time applications where data arrives sequentially.

The ability to perform continual, uncertainty-aware scene reconstruction opens promising directions for robotics. In autonomous navigation, exploration, or mapping tasks, robots often operate under strict memory, energy, and latency constraints. They must build and update a 3D model of their environment while interacting with it in real time. Traditional gradient-based methods are not suitable for such settings, as they require extensive recomputation and lack robustness when presented with non-stationary data streams. VBGS, by contrast, offers a probabilistic mechanism to adapt continuous inputs, ensuring that representations remain consistent as the environment evolves. When optimized for embedded GPUs, VBGS has the potential to provide a practical backbone for onboard perception in autonomous systems.

In summary, this work focuses on the optimization of the (VBGS) algorithm, with the goal of enabling its deployment on constrained hardware platforms. Originally designed for server-grade GPUs with abundant resources, VBGS has not been adapted to operate within the limitations of embedded devices. This work addresses shortens that gap by conducting an extensive characterization of the algorithm's computational profile, identifying the main bottlenecks. By systematically analyzing these constraints, it becomes possible to pinpoint where optimization is both feasible and impactful.

A key contribution of this thesis is the introduction of an automated mixed-precision search procedure. Since VBGS was originally designed to operate in double precision for stability, its default arithmetic requirements far exceed what embedded GPUs can efficiently support. The mixed-precision search algorithm developed here selectively reduces the numerical precision of individual operations while rigorously controlling output fidelity. This not only reduces memory consumption and execution time but also establishes a general framework that can be applied to other ML algorithms facing similar deployment constraints.

The optimization process is further complemented by profiling and memory management strategies. By inspecting allocation patterns during training, the implementation was restructured to reduce intermediate buffer usage, lowering peak memory consumption by up to 75%. Together with quantization, these optimizations bring the algorithm within the limits of devices such as the NVIDIA Jetson Orin Nano, which offers only a fraction of the memory and power budget of workstation GPUs.
Beyond advancing the SoA in view synthesis, this work highlights a broader direction for real-world robotics. Autonomous systems must learn and adapt in dynamic, uncertain environments while operating under strict latency, energy, and memory constraints. Traditional neural methods, while powerful, are poorly suited for such scenarios because they require replay buffers, frequent retraining, and large GPU resources. In contrast, VBGS naturally supports continual Bayesian updates, integrating new data without the issue of catastrophic forgetting and providing uncertainty estimates that are crucial for safe decision-making.
The thesis is organized as follows:
Chapter 2 provides background on the context of needed Bayesian statistics concepts that are a core component of the VBGS algorithm.
Chapter 3 explains briefly the original Gaussian Splatting work mechanisms, the VBGS algorithm itself and how this two differ.
Chapter 4 describes the characterization process, the modifications brought to VBGS and the functioning of the automated mixed-precision search algorithm.
Chapter 5 presents a comprehensive visualization and analysis of the key results obtained throughout the project.
Chapter 6 offers concluding remarks and outlines potential directions for future works.

Chapter 2

Background

2.1 Bayesian statistics

Bayesian statistics [4] is a probabilistic framework for modeling uncertainty. In this framework, we define an a-priori belief on the parameters of a generative model, and we update it after seeing the data. Opposed to classical methods(e.g. MLE [5], LSE [6]) which usually return point estimates, Bayesian statistics outputs a distribution over parameters, better describing the degree of belief we have on our answer.

2.1.1 Bayes theorem

Central to Bayesian statistics is the Bayes theorem, which relates prior knowledge about parameters with new evidence from observed data:

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta)\, p(\theta)}{p(\mathcal{D})} \tag{2.1}$$

where θ denotes model parameters and $\mathcal{D}$ the observed data.
In this formulation:

- $p(\theta)$ is the *prior*, encoding assumptions or knowledge about parameters before seeing data.
- $p(\mathcal{D} \mid \theta)$ is the *likelihood*, describing how likely the observed data is under given parameters.
- $p(\mathcal{D})$ is the *evidence*, describing the total probability to observe the data.
- $p(\theta \mid \mathcal{D})$ is the *posterior*, the updated belief about parameters after incorporating data.

In practice, exact computation of the posterior is often intractable [7]; this means the denominator $p(\mathcal{D})$ cannot be computed exactly. Approximate methods such as *variational inference* are used, which reframe the posterior computation as an optimization problem to find the best approximating distribution, selected from a given family, of the true posterior. This principle underlies the Variational Bayes Gaussian Splatting (VBGS) approach described later in section 3.2.

2.1.2 Variational inference

The mathematical expression for the evidence is given by:

$$p(\mathcal{D}) = \int p(\mathcal{D} \mid \theta)\, p(\theta)\, d\theta. \tag{2.2}$$

Unfortunately most of the times (and especially when working in a high dimensional parameter space) this integral is not guaranteed to have a closed form solution. Variational Inference [7] solves this issue by offering, in general, a non-exact, iterative solution to the problem.
Given a set of latent variables z (random variables that cannot be measured, but influence the data output of a generative model) and a set of data points x, the main idea of Variational Inference is to approximate the posterior distribution $p(x \mid z)$ with an approximate distribution $q(z)$. The latter is chosen from a family of "proper" distributions in order to obtain some desired mathematical features.
In order to measure the difference between two distributions a metric called *Kullback-Liebler* (KL) divergence is used:

$$D_{\mathrm{KL}}(q(z) \,\|\, p(x \mid z)) = \mathbb{E}_q\Big[\log \frac{q(z)}{p(x \mid z)}\Big] \tag{2.3}$$

From equation (2.3) it is possible to derive two important properties:

- The KL divergence is always non-negative.
- The KL divergence is zero if and only-if $p = q$.

Through some mathematical manipulation it is possible to bypass the use of the posterior distribution $p(x \mid z)$ entirely and obtain an equivalent expression using the *Evidence Lower Bound* (ELBO) [8]:

$$\log p(x) = D_{\mathrm{KL}}(q(z) \,\|\, p(x \mid z)) + \underbrace{\mathbb{E}_q[\log p(x, z)] - \mathbb{E}_q[\log q(z)]}_{\mathcal{L}(q)=\mathrm{ELBO}} \tag{2.4}$$

In the form of (2.4) it's easy to see that $\mathcal{L}(q) \leq \log p(x)$ always holds true and $\mathcal{L}(q) = \log p(x)$ only when the KL divergence is zero. As $p(x)$ is fixed, minimizing

the KL divergence is the same as maximizing the ELBO. The original problem of finding a closed form solution can be rewritten as the optimization problem:

$$q(z) = min_q \mathcal{L}(q) \tag{2.5}$$

where every term in the ELBO can be computed.

2.1.3 Exponential Family Distributions

A large number of probability distributions commonly used in statistics and ML belong to the *exponential family of distributions* [9]. A distribution is said to be in the exponential family if its probability density function can be expressed in the following canonical form:

$$p(x \mid \theta) = h(x) \exp\Big(\theta^\top T(x) - A(\theta)\Big), \tag{2.6}$$

where:

- θ are the *natural parameters* of the distribution,
- $T(x)$ are the *sufficient statistics* of the data,
- $A(\theta)$ is the log-partition function ensuring normalization,
- $h(x)$ is the base measure.

This formulation yields several important theoretical properties. First, the sufficient statistics $T(x)$ summarize all the information in the data that is relevant for parameter estimation, a fact that underlies the efficiency of maximum likelihood methods. Second, exponential family distributions admit *conjugate priors*: if the likelihood is in the exponential family, one can select a conjugate prior such that the posterior (calculated via Bayes formula) belongs to the same family (and, thus, mantains the same form). This leads to closed-form Bayesian updates, avoiding the need for costly integration.

2.1.4 Conjugate Priors in VBGS

VBGS exploits these properties in order to derive efficient update rules. In particular:

- the Gaussian likelihoods over spatial positions and colors admit Normal-Inverse-Wishart (NIW) conjugate priors:

$$\mu, \Sigma \sim \mathrm{NIW}(m_0, \kappa_0, V_0, \nu_0). \tag{2.7}$$

- The categorical assignments over mixture components admit a Dirichlet prior:

$$\pi \sim \mathrm{Dir}(\alpha_0). \tag{2.8}$$

Because these pairs are conjugate members of the exponential family, the posterior distributions retain the same functional form as the priors, and their natural parameters can be updated in closed form. This allows VBGS to perform coordinate ascent variational inference (CAVI) [10] with simple update equations that aggregate sufficient statistics of the observed data over time, yielding, as will be discussed in section 2.1.5, an efficient *continual learning* algorithm without the need for replay buffers.

2.1.5 Uncertainty Propagation and Continual Learning

One of the main motivations for adopting a Bayesian framework in VBGS is the ability to naturally propagate uncertainty. Instead of storing fixed Gaussian parameters, VBGS maintains posterior distributions over them. This allows for:

- **Sequential updates:** new data refines the posterior without discarding prior knowledge.
- **Continual learning:** catastrophic forgetting is mitigated, as updates are accumulative by construction.

This probabilistic treatment forms the mathematical foundation for the methods described in later chapters.

2.2 Novel View Synthesis

Novel view synthesis (NVS) refers to the task of generating images of a scene from camera viewpoints that were not observed during training. Over the last few years, this field has experienced rapid progress.
A major breakthrough came with Neural Radiance Fields (NeRF) [1], which achieved unprecedented fidelity in reconstructing complex scenes from sparse input views. NeRF models represent scenes as continuous volumetric fields parameterized by neural networks, trained via differentiable volume rendering. Despite their quality, NeRFs suffer from long training times and slow rendering speeds, making them impractical for real-time applications.
To address these limitations, *3D Gaussian Splatting* (GS) was introduced by Kerbl et al. [2]. In contrast to NeRF, Gaussian Splatting represents a scene as a collection of anisotropic 3D Gaussian primitives that can be efficiently rasterized. This

formulation drastically reduces training and rendering time, enabling interactive frame rates while preserving competitive visual quality.
Recently, several surveys have highlighted the importance of Gaussian Splatting in the context of NVS and beyond. Chen and Wang [11] provide a comprehensive overview of methods and applications, while Fei et al. [12] position Gaussian Splatting as a paradigm shift for real-time 3D scene representation. These works underline how Gaussian-based methods are quickly becoming a central tool in computer vision and graphics research.

Chapter 3

Related Works

3.1 Gaussian Splatting

3D Gaussian Splatting (GS) [2] is a technique for novel-view synthesis and 3D environment reconstruction. In contrast with its predecessors (such as NeRFs) Gaussian Splatting represents the scene as a set of anisotropic 3D Gaussian primitives (usually referred as *splats*) positioned in space and serve as the atomic unit of the representation; this formulation makes the rasterization process very efficient, while mantaining differentiability for gradient-based optimization.
Each Gaussian component is associated with a set of parameters:

- **Mean and covariance:** define the position, scale, and orientation of the Gaussian in 3D space. The covariance encodes anisotropy, allowing the representation of ellipsoidal rather than spherical shapes.
- **Spherical harmonic coefficients:** represent view-dependent color. This compact encoding allows Gaussians to capture appearance variations due to lighting and viewing angle.
- **Opacity coefficient:** controls the transparency of the primitive and enables differentiable alpha-blending during rendering.

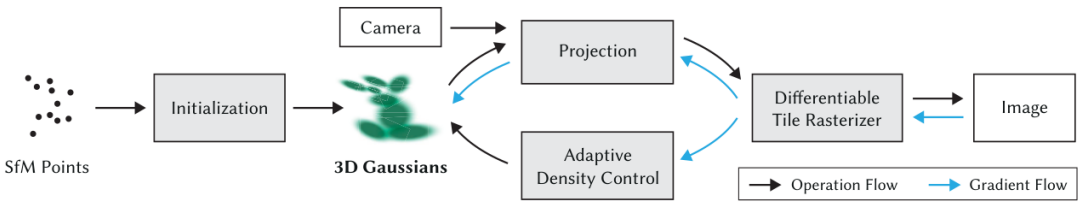

Figure 3.1: 3D Gaussian Splatting training flow, taken from [2]

Figure 3.1 illustrates the training pipeline, where black arrows denote the forward operation flow and blue arrows indicate the backward gradient flow used during optimization. First the splats get initialized on a sparse cloud of Structure from Motion (SfM) points. Then the GS algorithm alternates between an *projection phase* and an *adaptive-control phase*.

3.1.1 Projection phase

During the projection, the 3D Gaussians get projected onto the image plane using the camera parameters of the training image; then, using the fast rasterizer, a simulated image gets created by the splats. The loss gets calculated using the photometric error between the rendered image and the ground truth image. The gradient of this loss is used to update Gaussian parameters through Stochastic Gradient Descent SGD.

3.1.2 Adaptive-control phase

During the optimization phase, due to the natural ambiguity of 3D-to-2D projections, different 3D Gaussian can generate the same 2D projection on one plane. For this reason an adaptive-control algorithm runs every 100 iterations. The algorithm distinguishes spatial sections of the 3D environment in 2 categories, *under-reconstruction* and *over-reconstruction* sections. This categorization, together with the *pruning* mechanism, helps GS keep the 3D space's Gaussian density under control.

- **Pruning:** Gaussians that are essentially transparent, i.e., with opacity below a threshold $\alpha < \epsilon_\alpha$, are removed. This prevents wasted computation on primitives that do not contribute to the rendering.

- **Cloning (under-reconstruction):** In regions where geometric detail is missing, the algorithm identifies Gaussians with high positional gradients. Such Gaussians are duplicated, and the clone is mooved to the direction of the gradient. This increases the number of Gaussians and expands coverage of underrepresented areas.

- **Splitting (over-reconstruction):** In regions where one large Gaussian covers too much area (corresponding to high positional gradients and high variance), the Gaussian is split into two smaller Gaussians. Their scale is reduced by a factor $\phi \approx 1.6$, and their positions are initialized by sampling from the parent Gaussian's probability density function. This preserves total scene volume while improving local detail.

This heuristics ensure the number of Gaussians does not explode (note a high number of Gaussians could significantly slow down the rasterization process and, consequently, the overall training speed) and helps improve detail in low density zones.

3.2 Variational Bayes Gaussian Splatting

Variational Bayes Gaussian Splatting (VBGS) [3] is a probabilistic extension of 3D Gaussian Splatting that frames the learning process as *variational inference* over the parameters of the Gaussian mixture model. Instead of relying on backpropagation through a differentiable renderer, VBGS makes use of the conjugacy properties exponential family distributions and derives closed-form update rules, enabling efficient parameter estimation from sequential or partial observations. This approach naturally supports *continual learning*, avoiding the catastrophic forgetting [13] problem that affects gradient-based optimization when data arrives as a stream. Each Gaussian primitive in VBGS is not associated with a fixed set of parameters, but with a posterior distribution over them. This probabilistic treatment allows VBGS to integrate uncertainty directly into the scene representation and update components incrementally as new data arrives.

3.2.1 The Generative Model

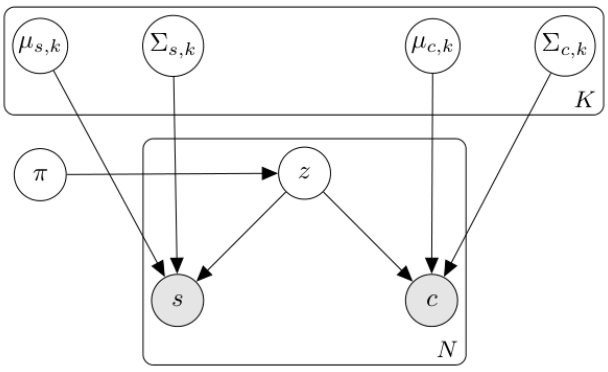

Figure 3.2: VBGS generative model; gray and white circles denote observed and unobserved data respectively. Taken from [3]

VBGS models the scene as a mixture of K Gaussian components. Each data point consists of a spatial coordinate s (2D pixels or 3D points) and a color vector c (RGB). The generative process is:

- A latent assignment $z \sim \text{Cat}(\pi)$ selects a component.

- The spatial observation is drawn as $s|z = k \sim \mathcal{N}(\mu_{s,k}, \Sigma_{s,k})$.
- The color is drawn as $c|z = k \sim \mathcal{N}(\mu_{c,k}, \Sigma_{c,k})$.

$\mu_{s,k}$ and $\mu_{c,k}$ represent the spatial and color means of the k^{th} distribution, while $\Sigma_{c,k}$ and $\Sigma_{s,k}$ their respective covariance matrices. The component parameters themselves are random variables: the parameters (μ, Σ) follow a Normal–Inverse–Wishart prior (NIW), $\mu, \Sigma \sim \text{NIW}(m, \kappa, V, n)$ while the mixture weights follow a Dirichlet prior, $\pi \sim \text{Dirichlet}(\alpha)$.
More in detail, the parameters of the NIW are:

- m: the prior mean of the Gaussian distribution.
- κ: the scaling factor controlling the strength of the prior on the mean.
- V: the scale matrix of the inverse Wishart distribution, encoding the prior covariance structure.
- n: the degrees of freedom of the inverse Wishart distribution.

This hierarchical model (Fig. 3.2) allows inference to change confidence along all levels of the representation.

3.2.2 Coordinate Ascent Variational Inference

Posterior inference in VBGS is performed via *Coordinate Ascent Variational Inference* (CAVI) [10]. The ELBO is maximized with respect to a mean-field factorization over latent assignments $q(z)$ and parameters $q(\mu, \Sigma, \pi)$.
The update cycle mirrors the Expectation–Maximization (EM) [4] algorithm:

- **E-step:** Compute the responsibilities $\gamma_{n,k}$, i.e., the probability that component k generated observation (s_n, c_n).
- **M-step:** Update the natural parameters of the posterior distributions by aggregating sufficient statistics of the data, weighted by responsibilities.

Because conjugate priors are used, these updates are closed-form and accumulate over time.

3.2.3 Continual Updates

Unlike gradient descent, VBGS supports continual learning through sequential Bayesian updates:

$$\eta_{t,k} = \eta_{t-1,k} + \sum_{x \in D_t} \gamma_{k,x} T(x), \qquad \nu_{t,k} = \nu_{t-1,k} + \sum_{x \in D_t} \gamma_{k,x} \tag{3.1}$$

where $T(x)$ are sufficient statistics and (η, ν) are natural parameters of the conjugate prior. This ensures that new data refines the posterior without overwriting prior knowledge, avoiding catastrophic forgetting.

3.2.4 Component Reassignments

In continual learning, uniform initialization may lead to unused components or poor coverage of the data space. VBGS introduces a *component reassignment* heuristic: unused Gaussians are reassigned to unexplained regions by sampling from data points with a probability proportional to the negative ELBO; this way poorly explained points have the highest probability of being reassigned. This increases coverage while keeping the number of components (and consequantially the model complexity) constant.

3.2.5 Rendering

For images, rendering reduces to computing the expected color given pixel coordinates. For 3D data, VBGS employs the GS rasterizer [2], projecting Gaussians to the image plane and combining them via alpha blending. Unlike GS, opacity is not optimized; all Gaussians are treated as opaque, simplifying rendering while still supporting novel view synthesis.

3.3 Advantages over Gradient-based GS

VBGS offers several improvements over gradient-based Gaussian Splatting:

- **Continual learning:** No need for replay buffers or retraining, as Bayesian updates are inherently accumulative.
- **Closed-form updates:** Eliminates the need for backpropagation through the renderer.
- **Uncertainty representation:** Each Gaussian has a posterior distribution, enabling an accurate description of the confidence level.
- **Efficiency in streaming data:** Single-step updates make VBGS suitable for robotics and online perception tasks.

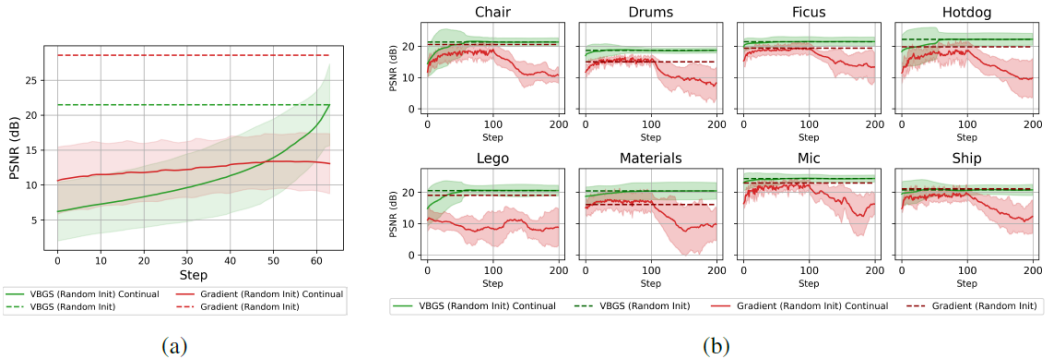

Figure 3.3: Continual learning performance. (a) Evolution of image reconstruction performance measured as PSNR (dB) after feeding image patches of size 8x8 sequentially to the model. Confidence intervals are the Z95 interval computed over the 10k validation images of the Tiny ImageNet dataset [14] of size 64x64. (b) Evolution of the reconstruction performance, measured as PSNR (dB), after feeding in consecutive images of an object. The shaded area indicates the 95% confidence interval computed over the 100 frames from the validation set. Taken from [3]

Figure 3.3 compares the performance of GS ad VBGS on the Tiny Imagenet dataset [14]. The graphs confirm our expectations: When data is fed in block the two algorithms have comparable performances, while when GS receives partial information about the image (in (a) images are fed in sub-patches while in (b) the 3D object's images are provided one-by-one) it clearly underperforms VBGS. In contrast VBGS's performance remains consistent in the continual learning setting.

Chapter 4

Methods

Ther core og this thesis explores a set of optimization strategies for the VBGS algorithm. The main objectives are **(1)** to reduce the memory footprint of the algorithm, both in terms of peak allocation and average utilization and **(2)** to accelerate training by lowering overall execution time.
Section 4.1 focuses on a detailed characterization of the baseline VBGS implementation. Through systematic profiling of memory usage and runtime behavior, this section identifies critical memory and latency bottlenecks. Such analysis provides the foundation for developing targeted optimization strategies aimed at adressing this bottlenecks without compromising model accuracy.
Section 4.2 will describe initial algorithmic optimization targeted at removing intermediate allocations and improve inefficient implementations.
Sections 4.4 and 4.5 investigate numerical precision related optimizations. First, the rationale for quantization is presented, emphasizing its role in reducing computational overhead especially considering GPU acceleration and memory footprint. Finally, an automatic mixed-precision search algorithm tailored for statically compiled functions will be described. This mechanism systematically explores precision trade-offs, ensuring reduced precision does not impact the quality of the final output and the training metrics.
Taken together, these contributions outline a comprehensive optimization framework for VBGS, balancing memory efficiency and runtime performance while maintaining the algorithm's probabilistic robustness.

4.1 Profiling

Before applying targeted optimizations to the VBGS algorithm, it is essential to first obtain a detailed understanding of its computational behavior. Profiling aims at achieving this groundwork by measuring both memory utilization and execution

latency. These measurements reveal where the algorithm stresses device resources most heavily, and highlight which operations dominate runtime.

Such insights are particularly critical in the context of embedded devices. On this hardware, peak memory allocations may exceed device capacity, leading to execution failures, while inefficient implementations directly limit the feasibility of real-time training or inference.

In the following subsections, two complementary profiling analyses are discussed: memory profiling, which exposes allocation patterns and peak usage, and latency profiling, which quantifies the runtime cost of individual operations and functions. Together, these analyses establish the performance baseline against which all subsequent optimizations are evaluated.

4.1.1 Memory profiling

The memory usage of the VBGS algorithm can be divided in two groups:

- **Base memory:** the static portion that is always in memory. This includes the model parameters and the memory required to store the frame data. Base memory constitutes the minimum footprint of the algorithm, independent of the training process.

- **Training memory:** the dynamic portion that is allocated during a training cycle and freed after finishing. It primarily consists of intermediate results produced by JAX operations, intermediate arrays created during function execution, and data structures used in batch processing. This part is typically responsible for large memory fluctuations during execution.

  The algorithm always keeps two models in memory, the *prior model* and the *approximate posterior model*. The first represents our initial beliefs on the parameters, while the second is the version updated by the observed data. The distinction between the two is fundamental for enabling continual learning.

  Tables 4.1 and 4.2 report the size of the parameters of the two models, parametrized by the number of components n_c.

Table 4.1: Parameter sizes of the prior model, parametrized by the number of components n_c. Parameters are in the canonical form of the corresponding distribution.

Distribution	Parameter name	Size
$p(\mu_x, \Sigma_x),\ x \in \{s, c\}$	m_x	$3 \cdot n_c$
	κ_x	$1 \cdot n_c$
	V_x	$9 \cdot n_c$
	n_x	$1 \cdot n_c$
$p(\pi)$	α	$1 \cdot n_c$

Table 4.2: Parameters sizes of the approximate posterior model, parametrized by the number of components n_c. Parameters are in the canonical form of the corresponding distribution.

Distribution	Parameter name	Size
$q(\mu_x, \Sigma_x),\ x \in \{s, c\}$	m_x	$3 \cdot n_c$
	κ_x	$1 \cdot n_c$
	V_x	$9 \cdot n_c$
	n_x	$1 \cdot n_c$
$q(\pi)$	α	$1 \cdot n_c$

The function of this parameters is explained in 3.2.1. The constant values refer to the dimension of the parameter of the k^{th} distribution; the prior mean m has the same dimension of the spatial and color means μ (3). V is the same dimension as $sigma$ (3×3); $finally$,$\kappa, nand\alpha$ are all numbers (1).
With this information, the models' base memory occupation can be calculated as:

$$memory\ [GB] = (\sum parameter\ sizes) \cdot \frac{precision\ bits}{GB\ to\ bit\ conversion\ rate} \tag{4.1}$$

Where the conversion rate from GB to bits is equal to $8 \cdot 1024^3$.
Since the original implementation of VBGS uses `fp64` precision, for $n_c = 100000$ the models' base memory equals to ~ 0.02 GB. As we will see in this section, this can be considered negligible with respect to the other memory components.

Listing 4.1: Example of JAX's profiler usage.

```
training_loop()
    jax.profiler.start_trace(logdir)

    function_to_profile_1()
    function_to_profile_2()

    jax.profiler.stop_trace()
```

In this work, memory profiling was done using JAX's native profiling tool in conjunction with **XProf** [15], a visualization software designed for analyzing program traces. JAX provides an easy mechanism to record traces of execution [16]; an example is shown below:

The trace obtained with 4.1 can then be analyzed in XProf. XProf offers a series of visualization tools to help profiling. In particular, the *memory viewer* is the most valuable among them, as it shows the memory usage and fragmentation (occurs when free memory is split into small non-contiguous blocks, preventing large allocations despite sufficient total space over execution time).

Algorithm 1 VBGS main training loop.

```
Initialize prior_model, model
Initialize training_frames
Initialize stats at 0

for each frame ∈ training_frames do
    prior_model = reassign(prior_model, model, frame)
    model, stats = fit_gmm_step(model, stats, frame)
end for
```

Algorithm 2 VBGS `reassign()` definition.

Divide the frame in batches
Initialize elbos array empty

for each batch ∈ batches **do**
    elbo, _ = **compute_elbo_delta**(prior_model, batch)
    Append elbo to elbos
end for

$p_{\text{elbo}} = -\text{elbos}$
Normalize p_{elbo} so that it sums to 1

n_{reassign} = fraction of number of the components with no previous assignments
p_{reassign} = sample n_{reassign} points from data with probability p_{elbo}
c_{reassign} = n_{reassign} number of components with no previous assignments

Change the means of c_{reassign} to p_{reassign}
Update prior_model

return prior_model

Algorithm 3 VBGS `fit_gmm_step()` definition.

Divide the frame in batches

for each batch ∈ batches **do**
    _, assignments = **compute_elbo_delta**(prior_model, batch)

    $\text{stats}_{\text{raw}}$ = **compute_statistics**(batch)
    $\text{stats}_{\text{sum}}$ = **sum_stats_over_samples**($\text{stats}_{\text{raw}}$, assignments)
    Append $\text{stats}_{\text{sum}}$ to stats
end for

model = **update_model**(model, stats)

return model

Algorithm 1 shows the main training loop of the VBGS algorithm; it can be roughly divided into two main phases that alternate iteratively: the *reassign* step (2) and the *fit* step (3). In both steps the image (or frame) is divided into smaller

sub-patches during ELBO computation; the dimension of the sub-patches depends from the user-defined size parameter (b_s). This is done to avoid memory issues due to having to process the whole image at the same time. The reassign step provides a useful heuristic for component redistribution; all points in the frame get assigned a score based on the negative ELBO, which gives a higher score to points that are poorly explained by the model; random points are now sampled with probability equal to their score and a small number of components that have not received any assignments yet get their mean updated to those points' mean. The fit step is the true training phase. Each point in the images gets assigned to the component that explains it best (line 4). Then, sufficient statistics of the data are computed (line 6-7) and the model gets updated based on the previous assignments. Identifying which of the two phases is the most critical for memory usage is key to designing effective optimizations.

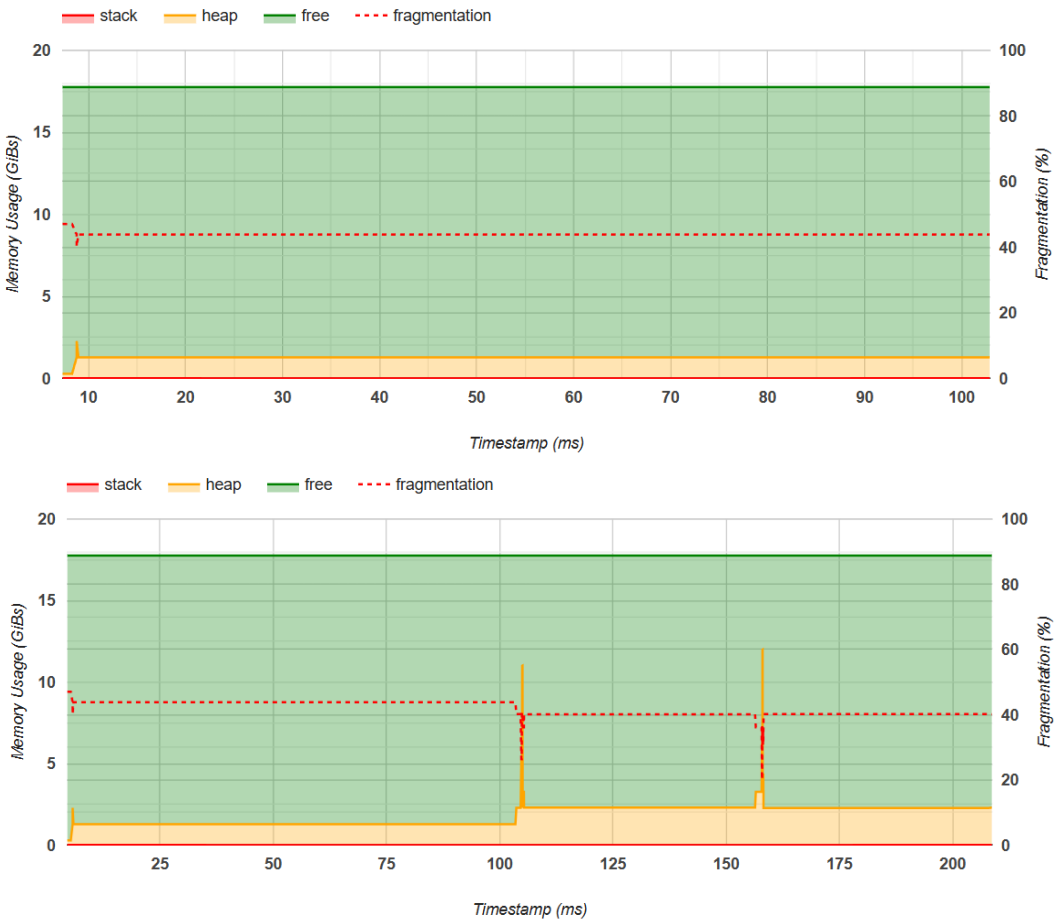

Figure 4.1: Memory profile over time of a single batch computation window during the reassign() step (top) and the fit_gmm() step (bottom)

.

Figure 4.1 illustrates the memory profiles of these two phases. It is evident that the `fit_gmm_step()` function is the primary source of memory pressure. During its execution, two distinct spikes in memory usage appear, each rising sharply above the baseline for only a brief duration. Such behavior hints at a large intermediate allocation that is created and freed within the scope of a single function call. By comparing the timing of these spikes with other XProf tools that relate the execution time with the python functions execution, it was possible to attribute the issue to the function `sum_stats_over_samples()`. Even if just transient, this big memory allocation can be detrimental for embedded devices, where the limited memory could prevent the function execution in its entirety. A solution to this point will be discussed and formulated in section 4.2.

4.1.2 Latency profiling

While memory profiling exposes how the algorithm interacts with device capacity, an equally important aspect is the time required to execute each part of the computation. Reducing latency is a critical factor when targeting real-time applications: embedded devices do not own the computational power of GPU, so every small improvement on GPU can become a huge time save on the device. Even when memory usage remains within acceptable bounds, excessive latency could make the algorithm impractical for target applications.

Latency profiling provides a detailed breakdown of execution time across the different stages of training, making it possible to pinpoint where the majority of computational effort is spent. By correlating timing information with the program's functional structure, bottlenecks can be identified and linked to specific functions or operations. This knowledge is essential for introducing subsequent optimizations: it highlights not only the regions where improvements may bring the greatest impact, but also whether inefficiencies from algorithmic design, redundant computations, or hardware underutilization could be present.

In the case of VBGS, profiling execution translates in understanding the impact of the sub-processes of the `reassign()` and the `fit_gmm_step()` routines. Understanding their relative contribution to overall runtime establishes the baseline against which optimization strategies can be developed and evaluated.

Measuring execution latency in JAX is not straightforward, since computations are executed asynchronously on the device. When a function is called, JAX typically only adds the operations to a queue, returning control to Python immediately while the actual kernels are executed in the background. As a result, normal timing measurements cannot capture true device runtime. Moreover, the first call to a JIT compiled function includes compilation overhead, which must be separated from steady-state execution. To obtain reliable latency values, explicit synchronization is required, typically by calling `block_until_ready()` on the returned arrays. This ensures that the host process waits until all queued operations have finished and the result is returned, making the measured time representative of the actual device computation. Once again, the XProf program is of great help in our goal.

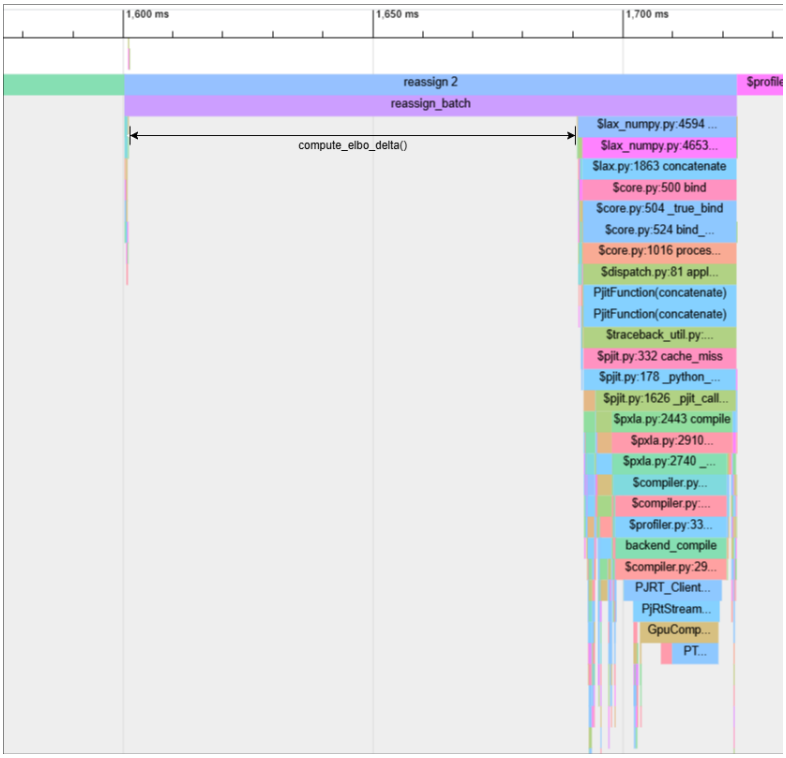

Figure 4.2: Python function-level profile of a single batch computation window during the `reassign()` function execution.

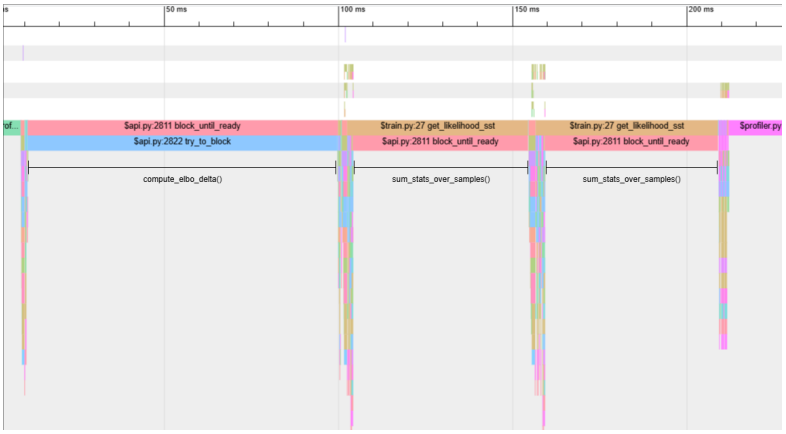

Figure 4.3: Python function-level profile of a single batch computation window during the `fit_gmm_step` function execution.

Figure 4.4: Visualization of the latency split during a batch of the reassign() (left) and fit_gmm_step() (right) functions.

The timeline in Figures 4.2 and 4.3 highlight the functions `compute_elbo_delta()` and `sum_stats_over_samples()` as the primary contributors to the overall training time. Both functions are invoked repeatedly in the training loop and involve computationally intensive operations, which makes their inefficiencies particularly impactful on runtime. A complete analisys of the implementations of these functions will be conducted in section 4.2 in order to improve the training speed.

Figure 4.4 provides a visualization of the computation time split in the `reassign()` and `fit_gmm_step()` functions. The left chart shows that during `reassign()` about 75% of the time is spent computing the ELBO. The right chart shows close to the full computation step is taken by the `compute_elbo_delta()` and the `sum_stats_over_samples()` functions.

At last, the correlation between training speed and batch size was investigated.

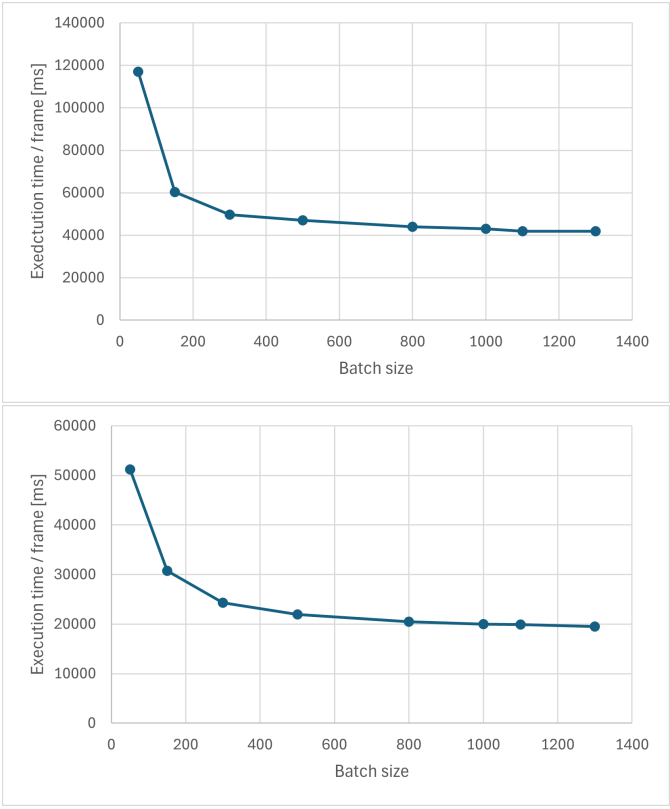

Figure 4.5: Correlation between latency and batch size in `fit_gmm_step()` (top) and `reassign()` (bottom).

Figure 4.5 reveals the batch size really affects training speed for values below 500. For higher values, the curve becomes flat and increasing batch size only yields an increase in memory occupation.

4.2 Initial code optimizations

The original VBGS algorithm produces an OOM error when ported to the Jetson Orin Nano. For this reason, the first goal has been reducing the peak memory utilization in order to comply with the devices constraints.
Section 4.1.1 revealed that the function `sum_stats_over_samples()` was the primary source of transient memory spikes, making it a key optimization target.
The function 4.2 performs an element-wise multiplication between two matrices and a reduction. The issue is that the results of the multiplication are not immediatley accumulated along the summing axis, but the multiplication result is first stored in memory, and then accumulated. For large matrices, this temporary allocation can be several times larger than the input tensors, leading to the short but extreme

Listing 4.2: `sum_stats_over_samples()` original implementation.

```
def sum_stats_over_samples(
    self, stats, weights, sample_dims):

    def sum_samples_over_leaves(tree):
        return jtu.tree_map(
            lambda leaf_array: (leaf_array * weights).sum(sample_dims
    ),
            tree,)

    return jtu.tree_map(
        lambda stats_tree: sum_samples_over_leaves(stats_tree), stats
    )
```

spikes observed in the memory in figure 4.1. While such spikes may be tolerable on a high-end GPU, they can exceed the available memory on embedded devices. To address this, the function was rewritten using `jax.lax.dot_general()` [16], JAX's low-level primitive for generalized tensor contractions. Unlike the previous approach, `dot_general` fuses the multiplication and reduction into a single operation. The key difference is that partial results are accumulated directly along the contracting axis, eliminating the need to allocate the full intermediate product in memory. This design not only reduces peak memory usage but also better aligns with how XLA lowers contractions to efficient device kernels.

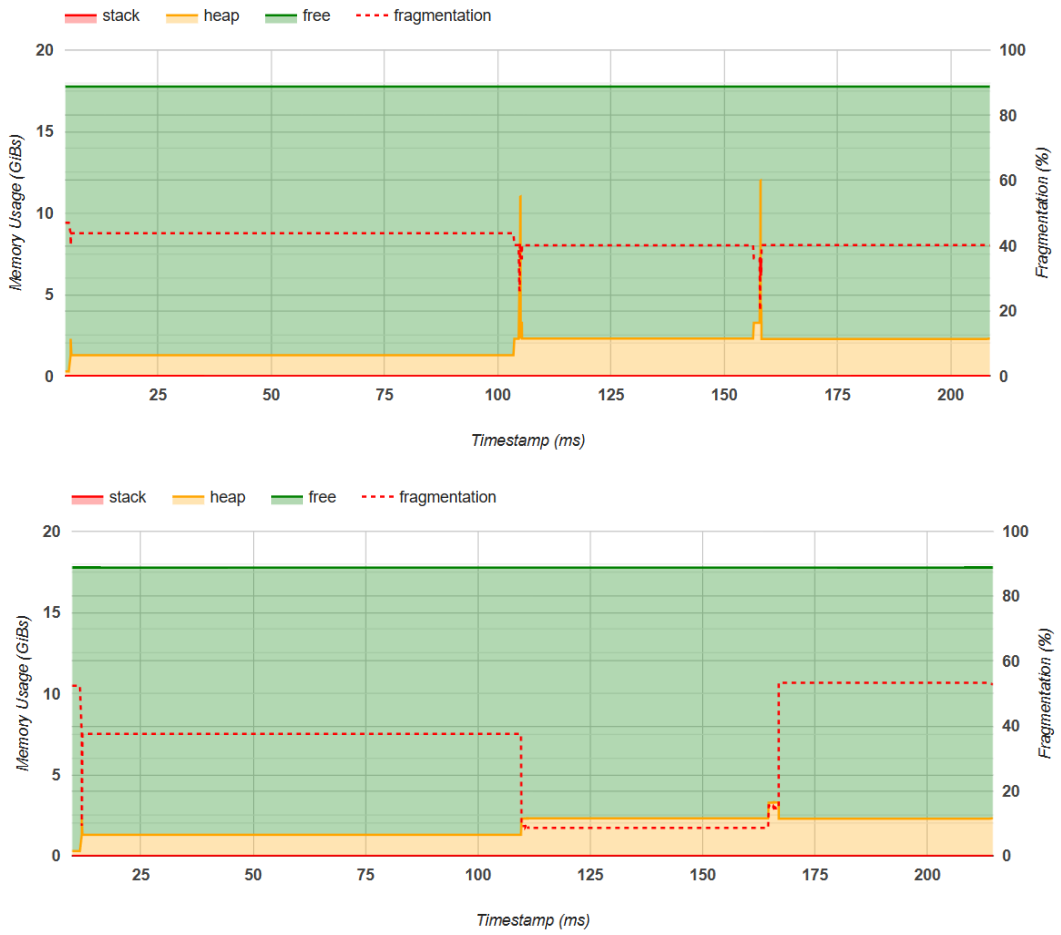

Figure 4.6: Memory profile over time of a single batch computation window during `fit_gmm_step()`. The top panel represents the memory usage of the baseline VBGS, the bottom panel shows the improvement brought by the new implementation of `sum_stats_over_samples()`

The effect of this change is illustrated in Figure 4.6. The bottom panel shows the improved implementation with `jax.lax.dot_general()`, where the spikes are completely eliminated.
As seen in Figure 4.4 the `sum_stats_over_samples()` function is also responsible for a significant fraction of the overall training time; Improving the efficiency of this function therefore has the potential to bring substantial speedups. Two concrete modifications were identified:

- In the current implementation, operations like `squeeze()` and `transpose()` (see Listing 4.3) are used to reshape outputs so they remain consistent with the original function shape. While functionally correct, these transformations

Listing 4.3: `sum_stats_over_samples()` second implementation.

```
def sum_stats_over_samples(self, stats, weights, sample_dims):

    def sum_stats_over_leaves(tree):
        return jtu.tree_map(
            lambda leaf_array: jax.lax.dot_general(
                ((sample_dims,), (sample_dims,)), ((),())
            ).squeeze(sample_dims,4,5)
            .transpose(2, sample_dims, 1),
            tree,)

    return jtu.tree_map(
        lambda stats_tree: sum_samples_over_leaves(stats_tree), stats
    )
```

introduce overhead. A more efficient solution uses the `einsum` function [17], which directly encodes the intended contraction and transposition pattern and avoids redundant reshaping operations.

- The function `sum_stats_over_samples()` is not JIT compiled, because the sub-function `sum_stats_over_leaves()` receives as input sufficient statistics of four different natural parameters, each with distinct shapes. However, since the number of natural parameters is small and fixed (only four), the function can be split into four specialized sub-functions, each dedicated to one natural parameter. This strategy makes the function static and enables effective JIT compilation, providing a great runtime reduction as will be shown in chapter 5.

Listing 4.4: `sum_stats_over_samples()` third implementation.

```
@jax.jit
def sum_stats_over_samples(stats, weights):

    def weighted_sum_eta_1(eta_1, weights):
        return jnp.einsum("adf, ai -> idf",
                          jnp.squeeze(eta_1, axis=1),
                          jnp.squeeze(weights))

    def weighted_sum_eta_2(eta_2, weights):
        return jnp.einsum("adf, ai -> idf",
                          jnp.squeeze(eta_2, axis=1),
                          jnp.squeeze(weights))

    def weighted_sum_nu_1(nu_1, weights):
        return jnp.einsum("adf, ai -> idf",
                          jnp.squeeze(nu_1, axis=1),
                          jnp.squeeze(weights))

    def weighted_sum_nu_2(nu_2, weights):
        return jnp.einsum("adf, ai -> idf",
                          jnp.squeeze(nu_2, axis=1),
                          jnp.squeeze(weights))

    # Flatten the pytree of stats
    leaves, tree_def = tree_flatten(stats)

    # Apply the weighted sums to each leaf
    leaves[0] = weighted_sum_eta_1(leaves[0], weights)
    leaves[1] = weighted_sum_eta_2(leaves[1], weights)
    leaves[2] = weighted_sum_nu_1(leaves[2], weights)
    leaves[3] = weighted_sum_nu_2(leaves[3], weights)

    # Reconstruct the tree
    new_stats = tree_unflatten(tree_def, leaves)
    return new_stats
```

4.3 Modified Training Loop

Another optimization attempt focused on reducing the overhead of the `reassign()` procedure by restructuring the training loop. In the baseline algorithm, reassignment requires computing the ELBO and posterior responsibilities twice (once for the reassignment step, and once for the fit step), introducing redundant computations and memory usage.

4.3.1 Motivation

The key idea was to merge the reassignment logic into the main EM cycle. Instead of recalculating the ELBO and responsibilities separately for the reassigned components, these values would be computed once and then selectively modified to reflect reassignments. This was expected to reduce latency, while still preserving the statistical meaning of the variational updates.

4.3.2 Implementation

The modified loop was structured as follows:

1. Compute ELBO and posterior responsibilities once for the full frame.
2. Compress the posterior matrix into an `argmax` representation of the size of the frame, avoiding the need to explicitly store the full posterior matrix of size $(512 \cdot 512) \times n_c$) (considering a frame of dimension $(512 \cdot 512)$, which for $(n_c = 100\,000)$ would result in an OOM error.
3. Identify components selected for reassignment and update the posterior representation so that these components receive one-hot assignments to their target points.
4. Re-batch the modified posterior matrix and recompute sufficient statistics, followed by the M-step update.

4.3.3 Results and Limitations

This merged training loop produced a clear speed-up compared to the baseline implementation. The modified training loop yielded a 25% to 50% latency decrease (depensing on the batch size value).
Despite these gains, a fundamental limitation emerged: reassigned components in this scheme received only a single hard assignment from their designated poorly explained point. In contrast, in the baseline algorithm these components acquired hundreds of assignments through full ELBO recomputation. This imbalance led

to worse updates and degraded the overall quality of the representation. Since recomputing the ELBO for reassigned components would negate the speed advantages, the modified training loop was ultimately not adopted in the final optimized framework.

4.4 Quantization

Some parameters of the model of the VBGS codebase makes use of very small numberical values to describe fine-grained differences between Gaussians. To avoid precision issues, led the original implementation was fully developed in `float64` and the quantization possibility was never explored.
While utilizing such a high-precision options ensures robustness, it also translates in an important use of computational resources and a high memory footprint, even when a lower-precision would be acceptable. On modern accelerators, particularly GPUs, lower-precision arithmetic such as `float32` or `float16` is not only more memory-efficient but can also exploit specialized hardware units (e.g., Tensor Cores [18]) for substantial speedups.
For these reasons a significant part of this work focuses on a rigorous quantization exploration in order to balance hardware resources and final quality.

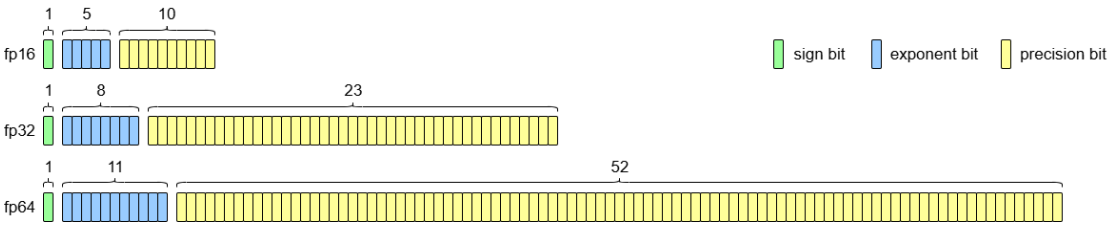

Figure 4.7: Comparison of IEEE 754 [19] floating-point bit representation. The sign bits are represented in green, the exponent bits in blue and the precision bits in yellow

Table 4.3: Comparison of IEEE 754 [19] floating-point ranges and precisions.

Format	Range	Precision (decimal digits)
`fp16`	6.1×10^{-5}-6.5×10^{4}	~ 3.3
`fp32`	1.2×10^{-38}-3.4×10^{38}	~ 7.2
`fp64`	2.2×10^{-308}-1.8×10^{308}	~ 15.9

Figure 4.7 and table 4.3 compare the bit representation of the three most popular floating-point precisions. Altough `fp64` offers the finest precision and the biggest dynamic range, it also occupies the biggest space in memory (2 times the space of an `fp32` number and 4 times the space of an `fp16` number). Moreover, the more bits are used to represent a number, the slower data transfer and its use for computation will be. The purpose of quantization is to reduce the numerical precision in order to balance the size throughput and final quality of resource intensive algorithms. Quantization is a well documented topic in ML, especially for DNNs ([20]), but research is lacking in the specific case of probabilistic ML.

4.5 Automatic mixed-precision search algorithm

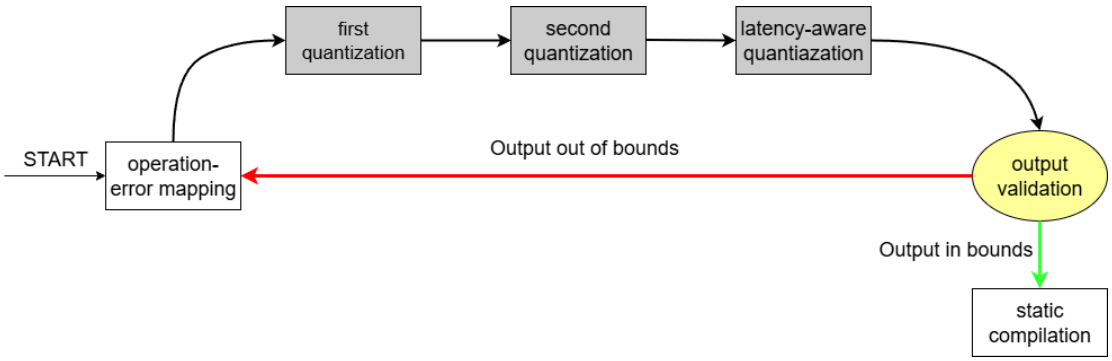

Figure 4.8: Automatic mixed-precision search algorithm's workflow diagram. the green and red arrows indicate respectively a succesfull and unsuccesfull validation.

Achieving the best balance between precision and performance requires a systematic strategy. For this purpose, an *automatic mixed-precision search algorithm* was designed. Its goal is to assign to every operation in a certain function the optimal precision given a certain tolerance; In essence, the algorithm seeks to minimize computational and memory costs while ensuring that the deviation from a high-precision reference output remains within acceptable bounds.

To start, a way to identify the atomic operation was needed. As the project already used JAX, adopting the **jaxpr** was convenient. The jaxpr is a low-level intermediate representation of a function, capturing the entire computation as a dataflow graph. Each equation in the jaxpr corresponds to a single primitive operation (e.g., addition, multiplication, or matrix multiplication); figure 4.9 provides the jaxpr representation of an example function. By operating at this level of granularity, the algorithm can reason about precision assignment in a fine-grained and systematic manner.

```python
def example_fn(in1, in2, in3):
    intr1 = jnp.log(in1)
    intr2 = in2 * in3
    intr3 = in3 / jnp.abs(in2)
    return intr1 + intr2 * intr3

in1 = 1.0
in2 = -2.0
in3 = 3.0
jaxpr = jax.make_jaxpr(example_fn
    )(in1,in2,in3)
```

```
{ lambda ;
    a:f32[] b:f32[] c:f32[].
    let
    d:f32[] = log a
    e:f32[] = mul b c
    f:f32[] = abs b
    g:f32[] = div c f
    h:f32[] = mul e g
    i:f32[] = add d h
  in (i,)
  }
```

Figure 4.9: Python function and its corresponding jaxpr representation, shown side by side.

Figure 4.8 shows a summary of the search algorithm's main functional blocks. These will be discussed one by one in the following sections.

4.5.1 Operation-error mapping

The first step of the algorithm is understanding how modifying an operation precision changes the overall function's output. The optimal approach would involve exhaustively testing all possible combinations of precision assignments across operations; such an approach is computationally intractable even for moderately sized functions, as the number of combinations grows exponentially with the number of operations. Algorithm 4 provides pseudocode for this implementation.

1. **Reference Computation:** First, the function of interest is executed with all operations forced to the highest available precision, in our case `fp64`. The resulting output serves as the *reference output*, representing the ground truth against which all further approximations are compared (line **1**).

2. **Operation-wise Perturbation:** For each operation in the jaxpr, the algorithm constructs a modified version of the function (line **5**) in which *only that specific operation* is executed at a lower precision (e.g., `fp32` or `fp16`), while all remaining operations are kept at `fp64`. The function is then executed again, producing an *altered output* (line **6**).

3. **Error Measurement:** The altered output is compared to the reference output using maximum relative error (line **7**). The result quantifies the sensitivity of the overall function output to lowering the precision of that particular operation.

4. **Error–Operation Mapping:** By repeating the perturbation process for every operation, the algorithm constructs a mapping between each jaxpr equation and its associated error impact (line **8**). This highlights which operations are robust to precision reduction and which ones are critical for maintaining numerical stability.

Algorithm 4 Compute operation-error mapping

1: $y_{ref} = f(x)$ with all operations in precision hp
2: Initialize empty map M
3:
4: **for** each $op \in O$ **do**
5: $\quad f_{op} =$ function f where op runs in precision lp
6: $\quad y_{op} = f_{op}(x)$
7: $\quad \text{err} = \max\Big(|y_{ref} - y_{op}|\Big)/\text{scale}$
8: $\quad M[op] = \text{err}$
9: **end for**
10: **return** M

4.5.2 First quantization pass

The operation-error mapping obtained in the previous section is now used to construct a quantization map Q. The goal is to downcast as many operations as possible to a lower precision while ensuring that the final output remains within a specified error tolerance compared to the high-precision reference. Algorithm 5 provides pseudocode for this implementation.

1. **Initialization:** We begin with a fully high-precision reference computation $f(x)$, executed with all operations at precision `fp64`. This establishes the reference output y_{ref} (line **3**). In parallel, we define a quantized candidate function f_Q (line **4**) where, by default, all operations are executed in a lower precision (e.g., `fp16`).

2. **Error Ranking:** From the operation–error mapping M, the operations are ordered in descending order of their individual error contributions. This ranking reflects the criticality of each operation: those with higher error scores are more likely to destabilize the output if kept in low precision.

3. **Progressive Upcasting:** Starting from the fully quantized function f_Q, the algorithm progressively reverts operations back to the reference precision (line **8**). The reversion (or "upcasting") of operation precision follows the error ranking order: the most critical operation is restored first, then the second most critical, and so forth. After each reversion, the function is re-evaluated on the input x, producing a candidate output y_Q (line **10**).

4. **Stopping Criterion:** At each iteration, the relative error between y_Q and y_{ref} is measured:

$$err = \frac{max(|y_{ref} - y_Q|)}{scale} \tag{4.2}$$

   If the error drops below a user-defined tolerance, the process terminates. This ensures the minimum number of required operations get reverted at this stage.

5. **Quantization Map:** The final outcome of this procedure is a quantization map Q, which associates each operation with the lowest possible precision. This map can then be used to consistently evaluate the function in future runs under mixed precision.

Algorithm 5 First quantization pass

1: Q = operation-precision mapping
2: M_{or} = M mapping ordered by decreasing error
3: $y_{ref} = f(x)$ with all operations in precision hp
4: $f_Q = f$ with all operations in precision $Q[op]$
5:
6: **while** err $<$ tolerance **do**
7: **for** each op in M_{or} **do**
8: $Q[op] = hp$
9: f_Q = function f using Q for ops precision
10: $y_Q = f_Q(x)$
11: err = max $\Big(|y_{ref} - y_Q|\Big)/scale$
12: **end for**
13: **end while**
14: **return** Q

It is important to note the mapping M does not account for error accumulation caused by non-linear function error accumulation. This translates to the mapping Q not truly describing the number of possible quantizations, but only a sub-optimal set; for this reason, a further pass is required to increase the number of quantized operations.

4.5.3 Second quantization pass

As discussed in the previous section, the operation–error mapping M only captures the local effect of lowering the precision of individual operations, without considering how errors propagate and accumulate in the computational graph. Consequently, the quantization map Q derived from the first pass represents a conservative, sub-optimal assignment: many operations that could, in fact, be quantized remain unnecessarily at high precision.

To address this, a second pass is performed in which the algorithm explores the graph dependencies of the jaxpr representation. Each operation is represented as a node, and edges denote input–output relationships between operations. The search starts from operations that have already been successfully quantized to low precision (treating the results of the first pass as "seeds") and then recursively propagates along their graph neighbors. Algorithm 6 provides pseudocode for this implementation.

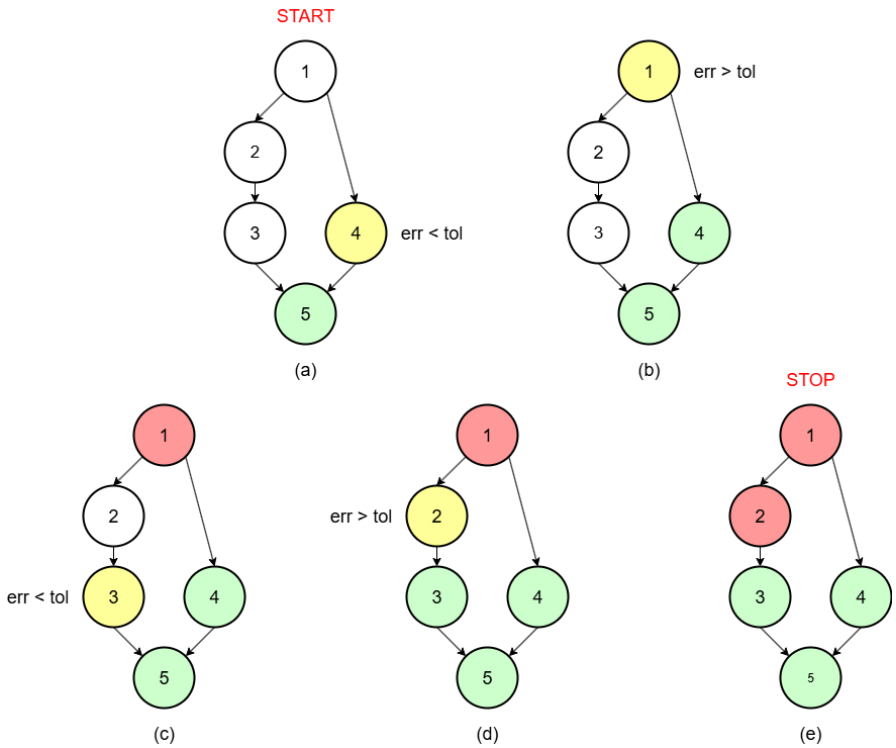

Figure 4.10: Example of the second quantization pass downcasting logic. Nodes in white represent unexplored nodes; nodes in yellow represent nodes currenlty in exploration. Nodes in green and red represent respectively quantized and unquantized explored nodes. Arrows show the input-output relationships between nodes. Graph **(a)** represents the initial graph configuration from the previous pass. In this configuration node **5** is the inital seed and node **4** is the first node that attempts quantization. As the error output error with the quantized node **4** is under tolerance, node **4** is now the *current* node. Graph **b** attempts quantization of node **1**, but fails due to output exceeding tolerance. As the predecessors and successors of **4** have already been searched, the last predecessor of **5**, node **3**, is next. In graph **(c)** we can see node **3** quantization is succesfull, node **2** is next node. Graph **(d)** attempts downcasting of node **2**, but fails due to the error exceeding tolerance. Graph **(e)** represents the final configuration after the second quantization pass. As the search space of the graph is now empty, the algorithm stops.

1. **Graph traversal:** Each operation is treated as a node in a directed graph. Starting from a node that is already assigned to low precision, we traverse recursively through all connected nodes following their input–output edges.

2. **Candidate downcasting:** When a connected node is found that is not yet in low precision, we attempt to downcast it (line **8**).

3. **Error evaluation:** The modified function output is computed and compared to the reference output. The relative error is computed in the same way as in equation 4.2 (line **10**).

4. **Acceptance criterion:** If the error remains below the tolerance threshold (line **12**), the candidate node is permanently added to the quantization map Q with assignment $Q[op] = lp$. If not (line **14**), the node is reverted to high precision.

5. **Termination:** The process repeats until all low-precision nodes and their neighbors have been visited, ensuring that quantization has been pushed as far as possible through the computational graph while respecting the tolerance constraint.

Figure 4.10 offers an example of the algorithm's behaviour on an toy graph.

Algorithm 6 Second quantization pass

```
1: y_ref = f(x) with all operations in precision hp
2: Q = quantization mapping from previous pass
3:
4: for each op with Q[op] = lp do
5:     Explore neighbors of op recursively
6:
7:     for each neighbor n not in lp do
8:         Temporarily set Q[n] = lp
9:         y_Q = f_Q(x)
10:        err = max(|y_ref − y_Q|)/scale
11:
12:        if err < tolerance then
13:            Keep Q[n] = lp
14:        else
15:            Revert Q[n] = hp
16:        end if
17:
18:    end for
19: end for
20: return Q
```

This recursive strategy complements the first pass by explicitly leveraging graph dependencies. It promotes a more aggressive reduction of precision in regions of the computation that are structurally robust, while avoiding error propagation

across sensitive edges. The combination of local error-driven (first pass) and recursive graph-based (second pass) downcasting provide a balanced and systematic exploration of the precision landscape, without having to analyze every possible quantization possibility.

4.5.4 Latency aware quantization pass

Operation quantization can, sometimes, be non-beneficial in the context of training speed; when introducing casting operations at precision boundaries some effects (e.g., higher memory traffic, conversion overhead) can decrease the computation speed of graphs with isolated low-precision nodes with respect to high-precision consistent operations. A simple cost function for a cluster can be written as:

$$C = \sum t^i_{computation} + \sum t^i_{casting} \tag{4.3}$$

To decrease training latency, only clusters of low-precision nodes that have an associated cost lower than the high-precision counterpart should be kept. In practice, it's not possible to precompute a mapping between operation type and computation time as after JIT compilation it's not guaranteed to be constant. The solution adopted directly measures the latency of the function with and without the cluster quantization; this way it's possible to say definitively if a cluster is worth quantizing or not. The clusters are first identified using a recursive strategy: for each low-precision node, we check recursively if its predecessors and successors are in the same precision. If they are, tehy get added to that cluster. Then, for each cluster, a function f_{tmp} gets created using $Q[cluster] = hp$. This function and the function f_Q get JIT compiled and compared and profiled. if f_{tmp} is faster, we update Q as:

$$Q[\text{cluster}] = hp \tag{4.4}$$

As a result of this process, the fastest possible quantization mapping Q for the avaiable low-precision nodes should be returned. Algorithm 7 provides pseudocode for this implementation.

Algorithm 7 Latency aware quantization pass

```
Q = quantization mapping from previous pass
f_Q = f with all operations in precision Q[op]
improved = True

while improved == True do
    cluster = define_clusters()

    for cluster ∈ clusters do
        f_tmp = f_Q with Q[cluster] = hp
        t_tmp = measure_latency(f_tmp)
        t_Q = measure_latency(f_Q)

        if t_tmp < t_Q then
            Q[cluster] = hp
            break
        end if

        improved = False
    end for
end while
return Q
```

4.5.5 Offline static compiling

Although greatly reducing the search space with respect to checking every combination of the jaxpr operations quantization, the algorithm still takes a non-negligible amount of time to run for large functions. Furthermore, memory usage can get really high during some parts of the algorithm execution (e.g., when computing the reference output we are basically computing the full function in `fp64`) and remove the possibility of it running on embedded devices where memory constraints are more strict. Both problems were solved by running the search algorithm in another process, before starting the true VBGS algorithm run. This is implemented via a function decorator and a setting option in a configuration file:

- **Compilation mode:** When enabled, the decorator triggers the automatic mixed-precision search before the true VBGS execution begins. To ensure robustness, the search is not performed on a small batch of five frames. This multi-frame evaluation prevents overfitting the quantization map Q to a specific image and provides a check to the quantization capability to generalize across data instances. If during algorithm execution a numerical issue is encountered

(e.g., overflows or underflows) in any of the subsequent frames, the search is automatically re-run. The resulting new quantization map replaces the previous one. Finally, the final Q map is serialized into a JSON file and stored for future use.

- **Running mode:** In this setting, the search algorithm is completely bypassed. The decorator looks for the precomputed JSON file containing the quantization map; the map is directly applied to the function at load time, effectively compiling a statically quantized version of the function without incurring the overhead of rerunning the search.

This workflow offers a practical solution to our issues: it allows VBGS to run with moderate memory usage, making the method suitable for embedded devices. The algorithm search only needs to be performed once, avoiding the latency overhead issue without compromises. Importantly, the use of multiple frames ensures that the stored quantization map remains stable and does not degrade when exposed to unseen inputs.

4.5.6 Decorator implementation

For practical integration into the VBGS codebase, the entire mixed-precision search procedure was encapsulated in a *Python decorator* named `mixed_precision`. A decorator in Python refers to a structure that changes the behaviour of a function without modifying it's code. For example, JIT compilation can also be applied to a function definition as a decorator. This allows any statically compilable JAX function to be automatically quantized without requiring changes to its interface. The decorator implements the two modes of operation described in section 4.5.5 using a user defined flag.
Inside the decorator the quantization map Q gets applied inside a JAX JIT compilation, so from the user perspective the decorated function behaves exactly like the original one. Importantly, the decorator also accepts a `static_argnums()` parameter; these are used to treat some function arguments as compile-constants (usually when they are used to manipulate array shapes) when they could technically change but the user knows they will not.

Listing 4.5: Example usage of the `mixed_precision` decorator

```
@mixed_precision()
def example_fn(in1, in2, in3):
    intr1 = jnp.log(in1)
    intr2 = in2 * in3
    intr3 = in3 / jnp.abs(in2)
    return intr1 + intr2 * intr3
```

Chapter 5

Experimental Results

This chapter presents the experimental evaluation of the optimized Variational Bayes Gaussian Splatting algorithm. The goal of this section is to demonstrate how the proposed profiling, memory optimizations, and quantization strategies impact the performance of VBGS across different environments and hardware platforms. Section 5.1 describes the datasets used in the training experiments; section 5.2 focuses on the metrics utilized to determine the quality of the algorithm, followed by section 5.3 containing details of the hardware used for training. Subsequently, we report quantitative and qualitative results in section 5.4, comparing the optimized implementation with the baseline version in terms of reconstruction quality, memory usage, and training latency. Finally, in section 5.6 the results are discussed with respect to scalability, efficiency, and applicability to real-world embedded deployment scenarios.

5.1 Datasets

Although the proposed algorithm is capable of reconstructing both images and 3D objects, in this work we focus on evaluating performance in indoor environments. To this end, three scenes from the Habitat test suite [21] were selected: the *Van Gogh Room*, *Apartment 1*, and *Skokloster Castle*. Environments have been preferred to objects since they are representative of typical robotic use cases, where embedded devices are deployed for tasks such as autonomous navigation, exploration, and mapping, which enabling is the point of this work. Unlike synthetic object-centric datasets, Habitat scenes provide complex room layouts with complicated geometry, diverse lighting conditions, and diverse camera positions, making them a more faithful benchmark for embedded deployment. The datasets were preprocessed using the DUSt3R pipeline [22].
The three rooms present different levels of difficulty, allowing us to determine the

robustness of the algorithm in increasingly challenging scenarios. Skokloster Castle represents the most complex setting, followed by the Van Gogh Room and, finally, Apartment 1. This varied complexity provides a useful spectrum for analyzing how quantization and optimization strategies scale with task difficulty.
For each environment, the dataset is divided into training and validation sets. A total of 200 randomly sampled frames are used for training, while an additional 200 unseen frames are used for validation. The use of non-overlapping splits ensures that the evaluation correctly measures generalization rather than overfitting of training viewpoints. Each frame includes RGB data, depth information and camera parameters.

5.2 Evaluation metrics

The performance assessment of the training is divided in 3 categories, each with a specific evaluation metric.

- **Rendering quality:** the most important criterion is the quality of the rendered images, as the primary objective of VBGS is to reconstruct realistic and accurate views of 3D environments. We utilize the *Peak Signal-to-Noise Ratio* (PSNR) [23], a widely used metric in image reconstruction. PSNR measures the logarithmic ratio between the maximum possible pixel intensity and the error introduced by reconstruction. It is derived from the Mean Squared Error (MSE) between a reference image x and its reconstruction $\hat{x}$:

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \|x(i,j) - \hat{x}(i,j)\|^2 \tag{5.1}$$

  where m and n denote the number of rows and columns in the image. Using MSE, PSNR can be computed as:

$$PSNR = 20 \log_{10}(\frac{MAX_f}{MSE}) \tag{5.2}$$

  where MAX_f represents the maximum possible pixel value (255 for 8-bit images). Higher PSNR values correspond to better similarity between original and reconstructed images.

- **Training speed:** the training speed is be evaluated as the average execution time per frame. Specifically, we record the total time required to process all training frames and divide this duration by the number of frames.

- **Memory usage:** memory efficiency is quantified as the peak RAM usage observed on the Jetson Orin Nano during algorithm execution.

5.3 Hardware

The algorithm performance is, naturally, heavily reliant on the hardware used. The results presented in 5.4 have been obtained on Jetson Orin Nano or Nvidia RTX A5000 GPU.

The two platforms used for evaluation differ substantially in terms of raw computational power and memory resources. The Jetson Orin Nano is designed for embedded and power-constrained environments, offering only 8 GB of LPDDR5 memory and a peak performance of about 1.28 TFLOPS in single precision, with a strict 15 W power budget. In contrast, the RTX A5000 is a workstation-class GPU with 24 GB of high-bandwidth GDDR6 memory, thousands of CUDA cores, and more than 20 times the floating-point computational power of the Jetson.

Table 5.1: Jetson Orin Nano technical specifications taken from [24].

Jetson Orin Nano	
Architecture	Nvidia Ampere
GPU memory	8 GB LPDDR5
Memory bandwidth	102 GB/s
NVIDIA Ampere architecture-based CUDA Cores	1024
NVIDIA third-generation Tensor Cores	32
Single-precision performance	1.28 TFLOPS
Tensor performance	40 TFLOPS
Power consumption	7-15 W

Table 5.2: Nvidia RTX A5000 technical specifications, taken from [25].

RTX A5000	
Architecture	Nvidia Ampere
GPU memory	24 GB GDDR6
Memory bandwidth	768 GB/s
NVIDIA Ampere architecture-based CUDA Cores	8,192
NVIDIA third-generation Tensor Cores	256
Single-precision performance	27.8 TFLOPS
Tensor performance	222.2 TFLOPS
Power consumption	230 W

5.4 Results

Figures 5.1, 5.2 and 5.3 show the evolution of reconstruction quality during training. For each environment, the optimized implementation closely tracks or surpasses the baseline in terms of reconstruction fidelity. The shaded areas indicate the 95% confidence interval, confirming that improvements are consistent across iterations and not due to random fluctuations. In the Van Gogh Room and Skokloster Castle the two algorithms start the same but, around frame 25, the original implementaion reaches a plateau while the optimized version's average PSNR continues growing.

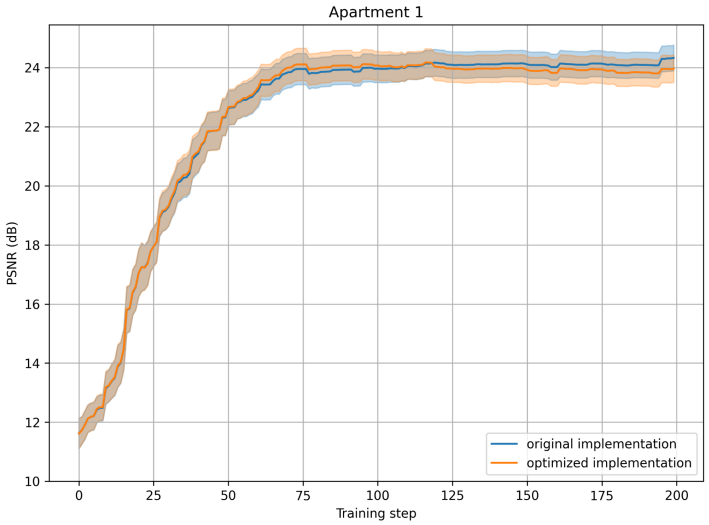

Figure 5.1: Comparison of the VBGS baseline implementation and the optimized algorithm's performances on the *Apartment* test environment; the continuous line represent the average PSNR at iteration i, the shaded area the 95% confidence interval.

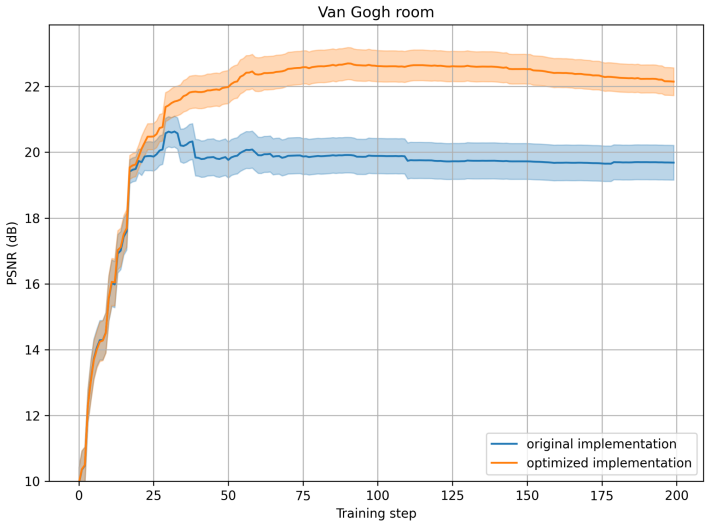

Figure 5.2: Comparison of the VBGS baseline implementation and the optimized algorithm's performances on the *Van Gogh Room* test environment; the continuous line represent the average PSNR at iteration i, the shaded area the 95% confidence interval.

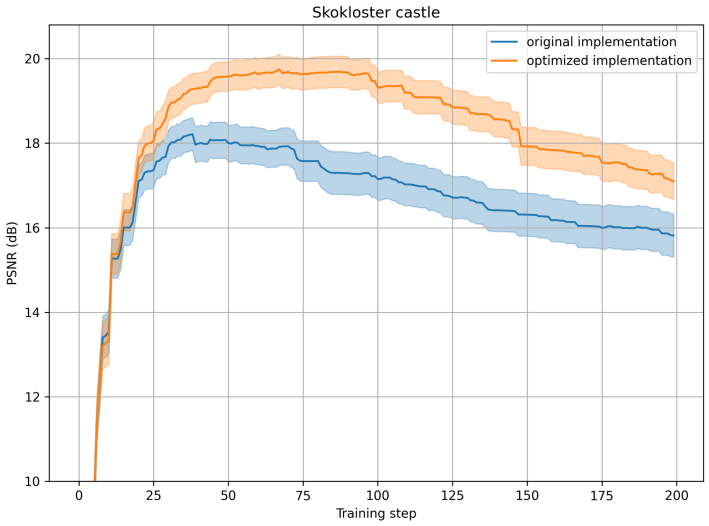

Figure 5.3: Comparison of the VBGS baseline implementation and the optimized algorithm's performances on the *Skokloster Castle* test environment; the continuous line represent the average PSNR at iteration i, the shaded area the 95% confidence interval.

Table 5.3: Peak PSNR comparison across environments.

Environment	Original	Optimized
Apartment 1	**24.31 dB**	24.18 dB
Van Gogh Room	20.08 dB	**22.70 dB**
Skokloster Castle	18.21 dB	**19.75 dB**

Table 5.3 compares peak PSNR values across environments. The optimized implementation preserves reconstruction quality overall, with negligible changes in Apartment 1, and clear improvements in both the Van Gogh Room and Skokloster Castle.

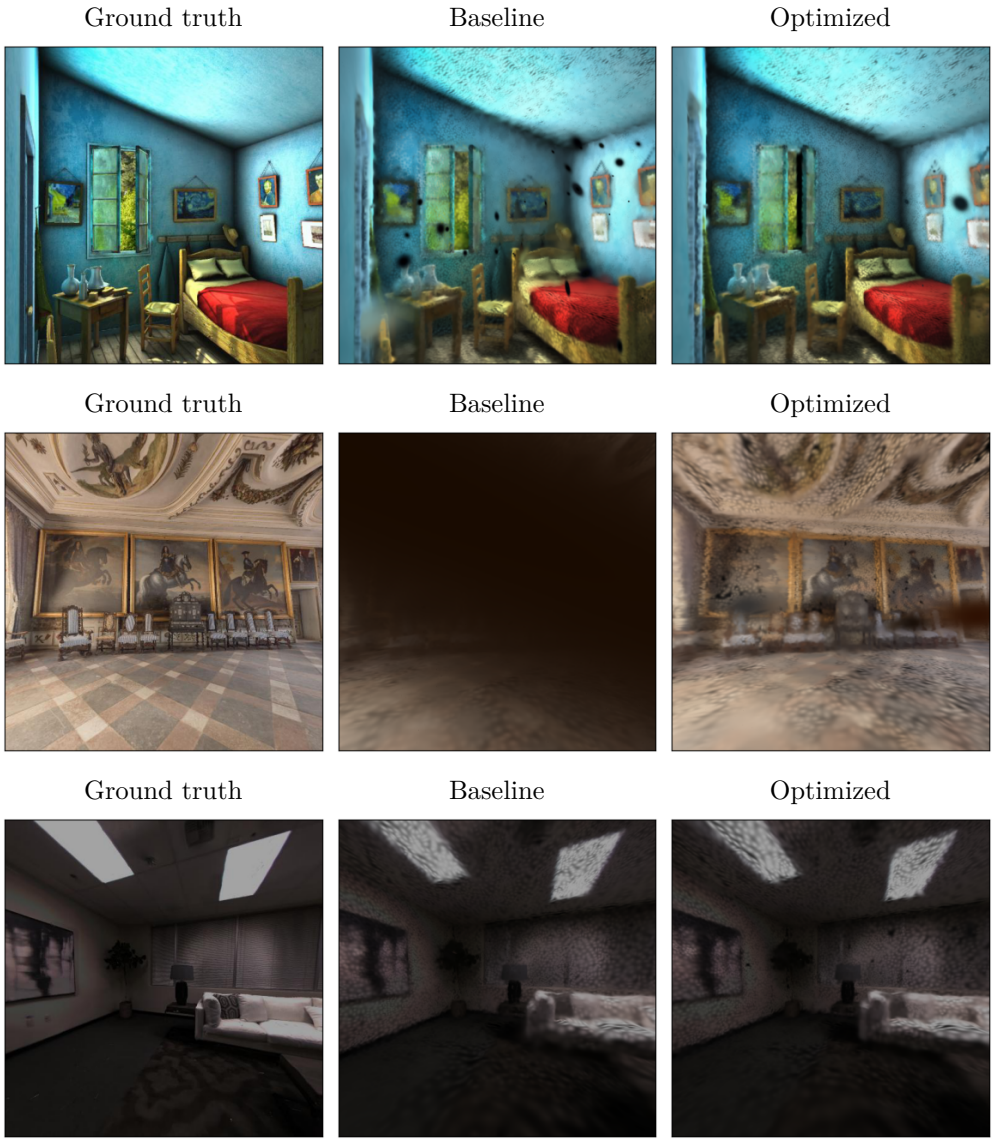

Figure 5.4: Image reconstruction comparisons for models trained over 200 frames; the left column images are the ground truth, the center column the VBGS baseline while the right column our optimized implementation.

Figure 5.4 presents the reconstructed images. The optimized version produces sharper details and fewer artifacts, particularly in scenes with complex lighting and clutter. The difference in detail across the rooms is also a function of environment scale: as the number of components is kept constant, bigger environments tend to be more sparse as the same number of components needs to cover a more space. This is clearly visible in the Sklokoster Castle renderings.

Table 5.4: Latency and memory metrics comparison. The Original* implementation refers to the baseline implementation after the optimization in listing 4.3. This result is presented here in place of the baseline algorithm as the baseline produces an OOM error on the Jetson.

Device	Implementation	Per-frame average latency [s]	Peak memory usage [GB]
RTX A5000	Original	70.12	9.44
	Optimized	18.33	1.11
Jetson Orin Nano	Original*	~900	4.95
	Optimized	~180	1.11

Table 5.4 reports latency and memory metrics on both the RTX A5000 workstation GPU and the Jetson Orin Nano embedded device. On the A5000, the average per-frame latency decreases from 70.12 s in the baseline implementation to 18.33 s after optimization, corresponding to a speedup of nearly 4 times. On the Jetson Orin Nano, the improvement is even more pronounced, with latency reduced from approximately 900 s to 180 s per frame, a 5 times acceleration. Peak memory usage, which previously prevented execution on the embedded platform, was also substantially reduced, allowing the algorithm to run reliably within the 8 GB limit.

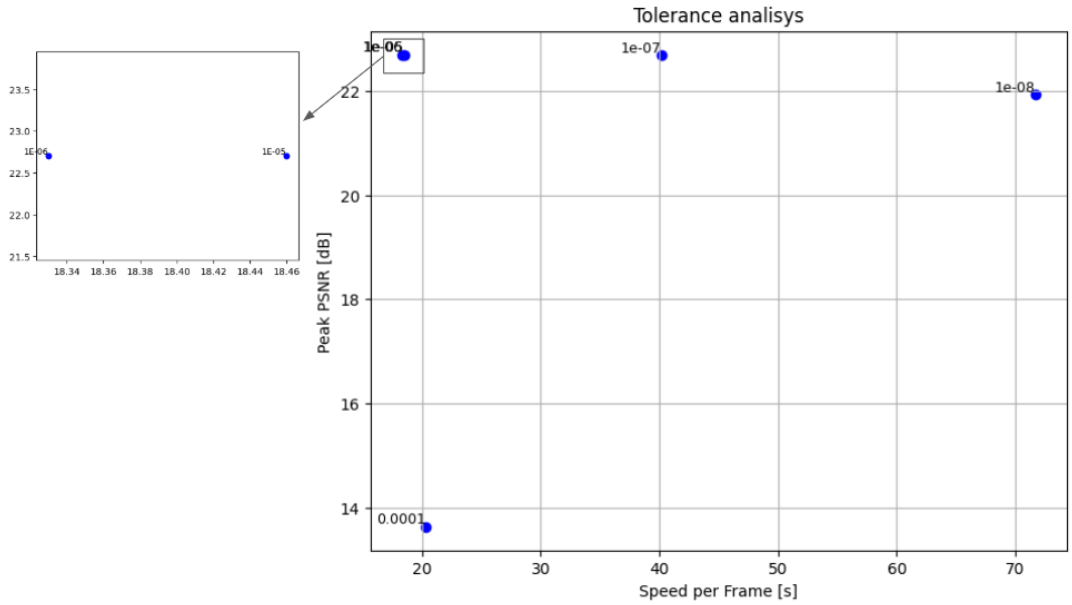

Figure 5.5: Tolerance swipe of the optimized VBGS algorithm. The y-axis indicates the PSNR in dB, while the x-axis the per-frame latency in seconds. The value over the points correspond to the tolerance value used when running the mixed-precision search. The experiment was conducted with the Van Gogh Room environment

The best tolerance was selected after testing different values in the Van Gogh Room. The results of the experiment can be seen in figure 5.5 and the best result was obtained with a value of 1e-06. Outliers can be seen in the 1e-04 point and the relative position of the 1e-05 and 1e-06 positions. 1e-05 seems to be the biggest sensitivity that is precise enough to guarantee the correct algorithm quality. Selecting a higher precision (like in the case of 1e-04) leads to an incorrect reconstrucion and, consequently, a low PSNR. Altough point 1e-05 has higher precision than point 1e-06 it is slower. This can be explained when considering the last pass in the mixed search: the higher precision point could be forming a bigger `fp16` cluster, that the algorithm considers cumulatively slower than the same cluster in fp32, but there could be sub-clusters of that cluster that are not.

5.5 Ablation Study on Floating-Point Precision

To understand the impact of numerical precision on the reconstruction quality of the VBGS training process, we performed an ablation study comparing multiple floating-point configurations for (Figure 5.6). The tested configurations include

`fp64` (double precision), `fp32` with both `HIGH` and `HIGHEST` accumulation precisions, and an adaptive version of `fp32` where a search algorithm dynamically selects the precision of each matrix multiplication based on its numerical sensitivity.
The baseline model, `fp64`, represents the reference implementation, providing high numerical accuracy but also the greatest computational and memory cost. The `fp32 HIGH` configuration corresponds to the TensorFloat-32 (`TF32`) standard, where accumulation precision is limited to a 10-bit mantissa, but receives a great speed-up due to the use of GPU *tensor cores* [18]. In contrast, `fp32 HIGHEST` maintains a full 23-bit mantissa in matrix multiplication accumulations, offering a closer approximation to the `fp64` baseline. Finally, the `fp32 + search` configuration leverages the automatic mixed-precision search algorithm, which applies the lowest possible precision to each matrix multiplication without degrading the final output. These results highlight the improvement capabilities of the use of quantization strategies in VBGS, not only from a computational and memory point of view, but also in terms of final reconstruction quality.

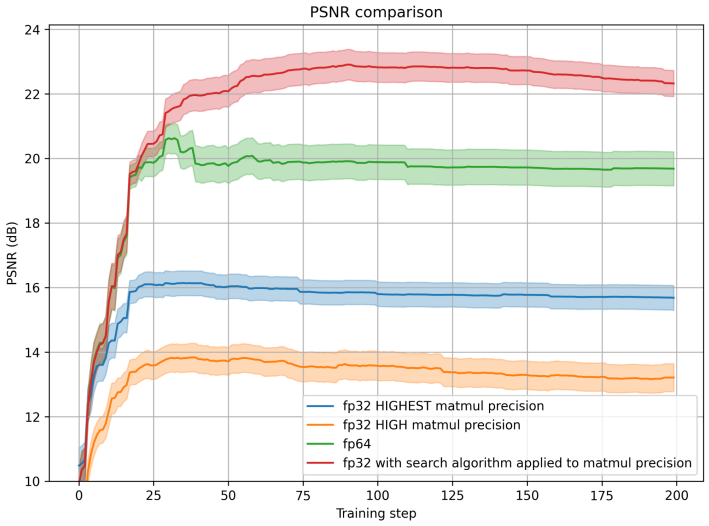

Figure 5.6: PSNR comparison between floating-point precision configurations. The adaptive `fp32 + search` mode achieves the best reconstruction quality, outperforming even the `fp64` baseline, while the `fp32 HIGH (TF32)` variant shows a significant quality drop. Full `fp16` training is omitted as it leads to precision errors and incomplete training.

5.6 Discussion

The results presented in section 5.4 demonstrate that quantization and memory optimizations do not degrade the visual fidelity of the reconstructions; on the

contrary, they can sometimes improve generalization by reducing overfitting to the training frames.
From a quality perspective, the optimized implementation consistently maintained the PSNR value of the baseline algorithm. Interestingly, the optimized implementation sometimes outperforms the baseline in terms of PPSNR, particularly in the Van Gogh Room and Skokloster Castle. This suggests that the aggressive use of double precision in the original version may have introduced unnecessary numerical rigidity, potentially leading to overfitting on the training frames. By selectively lowering precision, the optimized version appears to generalize better to unseen views.
Another interesting result can be found looking at the latency decreases of the optimized implementations across the two GPUs. The relative speed increase, in fact, does not match and seems to be higher on the embedded platform. This point could be attributed to multiple reasons:

- **Memory bandwidth:** on the Jetson is much lower than on the A5000, which could mean VBGS memory bottlenecked the board while the problem was less present on the server GPU.
- **CUDA and Tensor cores performance ratio:** while the ratio of Tensor cores and CUDA cores is the same between the 2 platforms (the A5000 has 8 times the cores of the Jetson), the performance ratio is not. The CUDA cores on the A5000 are 22 times more powerful than the ones on the Jetson, while the Tensor cores have only 5 times more throughput. Using mixed precisions, many more operations used `tensorfloat32`, which could have led to this difference in speed increase.

Overall, the results validate the optimization strategy: through targeted memory reductions and mixed-precision quantization, advanced probabilistic algorithms such as VBGS can be deployed on constrained platforms. Beyond the specific case of VBGS, these findings suggest that systematic profiling and mixed precision tuning can provide a general recipe for adapting complex numerical models to embedded environments.

Chapter 6

Conclusions

This thesis presented an optimization framework for Variational Bayes Gaussian Splatting with the specific goal of enabling deployment on embedded GPU platforms. Originally developed for high-end GPUs with abundant memory and computational power, VBGS was not previously adapted to devices with stringent resource constraints. By systematically profiling the algorithm, identifying its bottlenecks, and introducing targeted optimizations, this work demonstrated that SoA probabilistic 3D scene reconstruction methods can be made practical on devices such as the NVIDIA Jetson Orin Nano.

6.1 Summary of contributions

The first contribution of this work was a detailed profiling of the VBGS training loop. Memory and latency analyses revealed critical inefficiencies and non-essential use of double precision arithmetic. These investigations provided a clear roadmap for optimization.

The second contribution consisted of algorithmic modifications designed to reduce memory footprint. By rewriting the statistics accumulation routiness, peak allocations were eliminated, reducing maximum memory consumption by up to 75%. This improvement alone was essential to make execution feasible on the Nano's 8 GB memory budget.

The third contribution was the introduction of a rigorous quantization strategy. An automatic mixed-precision search algorithm was developed to assign the most efficient precision to every operation while preserving output quality within a user-defined tolerance. This algorithm, based on the `jaxpr` representation, allowed systematic exploration of the trade-off between precision and accuracy, and proved to be both effective and generalizable.

The optimized implementation achieved substantial improvements. On the RTX

A5000, the training latency decreased from 70.12 s to 18.33 s per frame, while peak memory usage dropped from 9.44 GB to 1.11 GB. On the Jetson Orin Nano, where deploymennt of the baseline algorithm results in an OOM error, the optimized implementation reduced latency from approximately 900 s to 180 s per frame and enabled stable execution within the 8 GB memory limit. Importantly, reconstruction quality was maintained or improved, with PSNR results confirming the effectiveness of mixed-precision optimization.

6.2 Broader implications

The results highlight two broader lessons. First, high-precision computation is often unnecessarily conservative in complex algorithms: careful downcasting not only reduces latency and memory use, but can also improve generalization by preventing overfitting to training frames. Second, embedded GPUs are particularly sensitive to inefficient memory usage and precision choices. Diversely from a workstation-class GPU that can mask inefficiencies with computational power, on constrained hardware such inefficiencies dominate execution time.
The mixed-precision search algorithm is a notable outcome in its own right. Although developed for VBGS, it is generic by design and can be applied to a wide range of JAX-based numerical models. Its ability to automatically adapt precision assignments makes it a valuable tool for future research on edge deployment of probabilistic or machine learning algorithms.

6.3 Limitations

Despite the improvements, several limitations remain. The optimized VBGS implementation, although significantly faster, still operates far from real-time on the Jetson Nano. The offline search procedure for mixed-precision assignments is computationally demanding and must be executed on a more powerful device before deployment. Furthermore, the number of Gaussian components was fixed across environments, which led to poor performance in larger scenes such as Skokloster Castle.

6.4 Future work

Future research could extend this work along several directions. From a systemic perspective, integrating dynamic model resizing, as done in gradient-based Gaussian Splatting, would allow the model to better manage 3D space across environments of varying scale; furthermore, on-device depth estimation could avoid the need for

depth data, reducing the requirements to only RGB data and camera parameters. From an algorithmic perspective, reducing the cost of the mixed-precision search through fast heuristics and dividing the graphs into memory manageable subgraphs could make it viable directly on embedded hardware. From an hardware point of view, the use of embedded boards with support of tensor cores for `fp16` could substantially increase the performance compared to the Jetson Nano.

6.5 Final remarks

In conclusion, this thesis demonstrated that by combining systematic profiling and mixed-precision quantization, it was possible to adapt an advanced probabilistic reconstruction framework for embedded deployment. The optimized VBGS implementation not only makes training feasible on resource-constrained devices, but also opens the door to future developments in edge robotics and real-time perception.

Chapter 7

Appendix

7.1 VBGS Update Rules

In this section, we provide an insight on the closed-form updates used in VBGS. The rules are derived from the general theory of variational inference with conjugate exponential family distributions [8].

7.1.1 General setting

Given observed data $\mathcal{D} = \{(s_n, c_n)\}_{n=1}^{N}$ consisting of spatial coordinates s_n and color vectors c_n, the generative model is defined as:

$$z_n \sim \mathrm{Cat}(\pi), \tag{7.1}$$
$$s_n \mid z_n = k \sim \mathcal{N}(\mu_{s,k}, \Sigma_{s,k}), \tag{7.2}$$
$$c_n \mid z_n = k \sim \mathcal{N}(\mu_{c,k}, \Sigma_{c,k}), \tag{7.3}$$

with priors

$$\mu_{s,k}, \Sigma_{s,k} \sim \mathrm{NIW}(m_{0,s}, \kappa_{0,s}, V_{0,s}, \nu_{0,s}), \tag{7.4}$$
$$\mu_{c,k}, \Sigma_{c,k} \sim \mathrm{NIW}(m_{0,c}, \kappa_{0,c}, V_{0,c}, \nu_{0,c}), \tag{7.5}$$
$$\pi \sim \mathrm{Dir}(\alpha_0). \tag{7.6}$$

7.1.2 Variational approximation

We adopt a mean-field factorization of the variational posterior:

$$q(z, \mu, \Sigma, \pi) = \Big(\prod_{n=1}^{N} q(z_n) \Big) \Big(\prod_{k=1}^{K} q(\mu_{s,k}, \Sigma_{s,k}) q(\mu_{c,k}, \Sigma_{c,k}) \Big) q(\pi). \tag{7.7}$$

Each factor is chosen from the same family as the prior, ensuring tractability:

$$q(z_n) = \mathrm{Cat}(\gamma_n), \tag{7.8}$$
$$q(\mu_{s,k}, \Sigma_{s,k}) = \mathrm{NIW}(m_{t,s}, \kappa_{t,s}, V_{t,s}, \nu_{t,s}), \tag{7.9}$$
$$q(\pi) = \mathrm{Dir}(\alpha_t). \tag{7.10}$$

7.1.3 Coordinate ascent updates

The variational parameters are optimized by maximizing the ELBO. Using the conjugacy properties of the exponential family, the following updates are obtained:

E-step (responsibilities). For each data point (s_n, c_n), the responsibility $\gamma_{n,k}$ of component k is:

$$\log \gamma_{n,k} \propto \mathbb{E}_{q(\mu_{s,k}, \Sigma_{s,k})}[\log p(s_n \mid \mu_{s,k}, \Sigma_{s,k})] \tag{7.11}$$
$$+ \mathbb{E}_{q(\mu_{c,k}, \Sigma_{c,k})}[\log p(c_n \mid \mu_{c,k}, \Sigma_{c,k})] \tag{7.12}$$
$$+ \mathbb{E}_{q(\pi)}[\log \pi_k]. \tag{7.13}$$

M-step (parameter updates). For each component k, the natural parameters are updated as:

$$\eta_{t,k} = \eta_{0,k} + \sum_{n=1}^{N} \gamma_{n,k} T(x_n), \tag{7.14}$$
$$\nu_{t,k} = \nu_{0,k} + \sum_{n=1}^{N} \gamma_{n,k}, \tag{7.15}$$

where $T(x_n)$ are the sufficient statistics of the Gaussian likelihood:

$$T(s_n) = \{s_n,\, s_n s_n^\top\}, \quad T(c_n) = \{c_n,\, c_n c_n^\top\}. \tag{7.16}$$

Dirichlet update. The mixture weights posterior parameters are updated as:

$$\alpha_{t,k} = \alpha_{0,k} + \sum_{n=1}^{N} \gamma_{n,k}. \tag{7.17}$$

These closed-form updates make it possible to integrate new data sequentially without gradient descent, forming the basis of continual learning in VBGS.

7.2 Initialization strategies

The baseline VBGS experiments considered two initialization strategies:

- **Data Init:** Means initialized from randomly sampled datapoints (s_n, c_n), providing good coverage of the dataset from the start.

- **Random Init:** Means initialized from uniform distributions ($s \sim U[-1,1]$, $c \sim \delta(0)$), independent of data.

Data-based initialization generally improves convergence speed and reconstruction quality. The experiments of our optimized version only consider random initialization, as this is the closest setting to a real-world environment.

7.3 Hyperparameters Initialization values

Tables 7.1 and 7.2 summarize the canonical hyperparameters initialization values used in our experiments. Values are consistent with the ones in the original VBGS paper [3].

Table 7.1: Parameters of the conjugate priors over likelihood parameters. n_c is the number of components, I is the identity matrix of size 3×3. Parameters are in the canonical form of the corresponding distribution.

Distribution	Parameter name	Initialization value
$p(\mu_{k,s}, \Sigma_{k,s})$	$m_{s,k}$	0
	$\kappa_{s,k}$	$10^{-2} \cdot 1$
	$V_{s,k}$	$2.25 \cdot 10^{6} \cdot n_c \cdot I$
	$n_{s,k}$	5
$p(\mu_{k,c}, \Sigma_{k,c})$	$m_{c,k}$	0
	$\kappa_{c,k}$	$10^{-2} \cdot 1$
	$V_{c,k}$	$10^{8} \cdot I$
	$n_{c,k}$	5
$p(\pi)$	α_k	$\frac{1}{n_c}$

Table 7.2: Parameters of the initial approximate posteriors over likelihood parameters. n_c is the number of components, I is the identity matrix of size 3×3. Parameters are in the canonical form of the corresponding distribution.

Distribution	Parameter name	Initialization value
$q(\mu_{k,s}, \Sigma_{k,s})$	$m_{s,k}$	$m_{s,k,\text{init}}$
	$\kappa_{s,k}$	$10^{-6} \cdot 1$
	$V_{s,k}$	$2.25 \cdot 10^6 \cdot n_c \cdot I$
	$n_{s,k}$	5
$q(\mu_{k,c}, \Sigma_{k,c})$	$m_{c,k}$	$m_{c,k,\text{init}}$
	$\kappa_{c,k}$	$10^{-2} \cdot 1$
	$V_{c,k}$	$10^8 \cdot I$
	$n_{c,k}$	5
$q(\pi)$	α_k	$\frac{1}{n_c}$

7.4 Implementation details

This section provides additional details about the implementation of the automatic mixed-precision framework described in Section 4.5. The following utilities were developed in order to manipulate `jaxpr` representations, evaluate functions under different precision assignments, and build consistent quantization maps. These details are reported here to give a complete view of the software engineering required to realize the methods, while the main text focuses on the algorithmic aspects.

7.4.1 Jaxpr helper class

The `MixedJaxprHelper` class acts as a central interface for managing multiple versions of a function's `jaxpr` at different numerical precisions. It provides:

- Storage of `ClosedJaxpr` objects for different {dtype, matmul precision} pairs.
- A quantization dictionary mapping operation indices to assigned precisions.
- Utilities to mark which operations have already been processed during the quantization passes.

- A deep-copy method that carefully avoids duplicating JAX internal source information and tracebacks, reducing memory overhead when exploring alternative quantization configurations.

This helper encapsulates the otherwise verbose calls to `jax.make_jaxpr` and `default_matmul_precision`, and exposes a unified interface to the quantization pipeline.

7.4.2 Evaluation under mixed precision

To assess the effect of quantization, two main functions are used:

- `evaluate_with_replacement()`: evaluates the function output by replacing only one equation of the `jaxpr` with its quantized counterpart, while keeping all other operations at reference precision. This isolates the non-compound sensitivity of a single operation.
- `evaluate_mixed_jaxpr()`: reconstructs the function execution by traversing the `jaxpr` and applying the current quantization dictionary. Operations not listed in the dictionary default to fp64.

These functions implement a custom evaluation loop, where inputs and outputs are explicitly written to and read from a temporary environment. This design mirrors the execution semantics of the `jaxpr`, but allows overriding the dtype of individual primitives.

7.4.3 Normalization and pruning

The function `normalize_floats()` reconstructs a `Jaxpr` in which all floating-point variables, constants, and outputs are retargeted to a single `dtype`. During this pass, redundant `convert_element_type` operations are eliminated to prevent spurious casts. Similarly, `strip_unused_eqns()` applies dead-code elimination to prune unused variables or intermediate computations. These utilities are important to keep the quantized `jaxpr` graphs compact and avoid over-counting unnecessary operations during sensitivity analysis.

7.4.4 Memory safety

The helper `remove_unused_environment_variables()` ensures that temporary values are deleted from the evaluation environment once they are no longer required by subsequent operations. This prevents unbounded growth of the variable environment.

7.4.5 Casting strategy

Finally, the `safe_cast()` function standardizes conversions to lower precision. It only casts when necessary:

- Floating arrays and scalars are converted if their dtype differs.
- Integers and booleans are returned unchanged to preserve semantics.
- Python floats are wrapped in JAX arrays only if the target dtype requires it.

This approach avoids introducing redundant same-precision conversions and keeps the evaluation loop as close as possible to the original function behavior.

Bibliograpy

[1] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. «NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis». In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer, 2020, pp. 405–421. DOI: 10.1007/978-3-030-58452-8_24. URL: https://arxiv.org/abs/2003.08934 (cit. on pp. 1, 7).

[2] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. «3D Gaussian Splatting for Real-Time Radiance Field Rendering». In: *ACM Transactions on Graphics (TOG)* 42.4 (2023), pp. 1–14. DOI: 10.1145/3592433 (cit. on pp. 1, 7, 9, 13).

[3] Toon Van de Maele, Ozan Catal, Alexander Tschantz, Christopher L. Buckley, and Tim Verbelen. *Variational Bayes Gaussian Splatting*. 2024. arXiv: 2410.03592 [cs.CV]. URL: https://arxiv.org/abs/2410.03592 (cit. on pp. 2, 11, 14, 59).

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. ISBN: 978-0-387-31073-2 (cit. on pp. 4, 12).

[5] Ronald A. Fisher. «On the mathematical foundations of theoretical statistics». In: *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character* 222 (1922), pp. 309–368 (cit. on p. 4).

[6] Carl Friedrich Gauss. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*. Hamburg: Friedrich Perthes and I.H. Besser, 1809 (cit. on p. 4).

[7] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. «An Introduction to Variational Methods for Graphical Models». In: *Machine Learning* 37.2 (1999), pp. 183–233. DOI: 10.1023/A:1007665907178 (cit. on p. 5).

[8] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. «Variational Inference: A Review for Statisticians». In: *Journal of the American Statistical Association* 112.518 (2017), pp. 859–877. DOI: 10.1080/01621459.2017.1285773 (cit. on pp. 5, 57).

[9] Lawrence D. Brown. *Fundamentals of Statistical Exponential Families with Applications in Statistical Decision Theory*. Hayward, CA: Institute of Mathematical Statistics, 1986 (cit. on p. 6).

[10] Matthew James Beal. «Variational Algorithms for Approximate Bayesian Inference». PhD thesis. University College London, 2003 (cit. on pp. 7, 12).

[11] Guikun Chen and Wenguan Wang. «A Survey on 3D Gaussian Splatting». In: *arXiv preprint arXiv:2401.03890* (2024). URL: https://arxiv.org/abs/2401.03890 (cit. on p. 8).

[12] Ben Fei, Jingyi Xu, Rui Zhang, Qingyuan Zhou, Weidong Yang, and Ying He. «3D Gaussian Splatting as New Era: A Survey». In: *IEEE Transactions on Visualization and Computer Graphics* (2024), pp. 1–20. DOI: 10.1109/TVCG.2024.3397828 (cit. on p. 8).

[13] Robert M. French. «Catastrophic forgetting in connectionist networks». In: *Trends in Cognitive Sciences* 3.4 (1999), pp. 128–135 (cit. on p. 11).

[14] Ya Le and Xuan Yang. *Tiny ImageNet*. https://zenodo.org/doi/10.5281/zenodo.10720916. 2015 (cit. on p. 14).

[15] OpenXLA Project. *XProf: Performance Analysis Tool for XLA and Machine Learning Workloads*. https://openxla.org/xprof. Accessed: 2025-09-24. 2024 (cit. on p. 18).

[16] Roy Frostig, Matthew Johnson, and Chris Leary. *JAX: composable transformations of Python+NumPy programs*. https://github.com/google/jax. 2018 (cit. on pp. 18, 26).

[17] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, and et al. «Array programming with NumPy». In: *Nature* 585 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2 (cit. on p. 28).

[18] *NVIDIA V100 Tensor Core GPU Architecture*. https://resources.nvidia.com/en-us-tensor-core/nvidia-volta-v100-whitepaper. Whitepaper. 2017 (cit. on pp. 31, 52).

[19] IEEE Standard Association. *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2019 (Revision of IEEE 754-2008). 2019. DOI: 10.1109/IEEESTD.2019.8766229. URL: https://doi.org/10.1109/IEEESTD.2019.8766229 (cit. on p. 31).

[20] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. «A Survey of Quantization Methods for Efficient Neural Network Inference». In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 16952–16973 (cit. on p. 32).

[21] Manolis Savva et al. «Habitat: A Platform for Embodied AI Research». In: *arXiv preprint arXiv:1904.01201* (2019). arXiv: `1904.01201 [cs.CV]` (cit. on p. 43).

[22] Shuzhe Wang, Vincent Leroy, Yohann Cabon, Boris Chidlovskii, and Jerome Revaud. «DUSt3R: Geometric 3D Vision Made Easy». In: *arXiv preprint arXiv:2312.14132* (2023). URL: `https://arxiv.org/abs/2312.14132` (cit. on p. 43).

[23] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. 3rd. Prentice Hall, 2008. ISBN: 9780131687288 (cit. on p. 44).

[24] *NVIDIA Jetson Orin Nano Module Series Data Sheet*. `https://developer.nvidia.com/embedded/jetson-orin-nano`. 2023 (cit. on p. 45).

[25] *NVIDIA RTX A5000: Technical Specifications*. `https://www.nvidia.com/en-us/design-visualization/rtx-a5000/`. 2023 (cit. on p. 46).