Hardware Acceleration for Real-Time Compression of 3D Gaussians

by

Kailas B. Kahler

S.B. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by: Kailas B. Kahler
Department of Electrical Engineering and Computer Science
May 16, 2025

Certified by: Vivienne Sze
Professor of Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Hardware Acceleration for Real-Time Compression of 3D Gaussians

by

Kailas B. Kahler

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

3D Gaussian Splatting (3DGS) is a technique for novel view synthesis, where images of a scene from a specific viewpoint are generated using images from different viewpoints, that has gained popularity for its reduced computational overhead, resulting in faster training and rendering times compared to other methods like Neural Radiance Fields (NeRFs). Its applications outside of strictly novel view synthesis have also been explored, with monocular simultaneous localization and mapping (SLAM) in robotics being an emergent application. However, because of limited on-board battery capacity, the computer hardware used in small robots is much less capable than the high-powered GPUs that the 3DGS algorithm was originally developed on, having both less compute and memory capacity and bandwidth. While there has been work developing specialized compute for the rendering pipeline of 3DGS, memory remains an obstacle to deployment. The Gaussian map can occupy from $1\text{MB} - 700\text{MB}$ in memory, which is both too large to store on-chip within micro-robots and such that moving Gaussians from memory to compute can dominate power consumption. While there has been prior work on algorithms for compressing Gaussian representations, they are not yet capable of running in real-time on the hardware present in these robots, as would be required for SLAM. Thus, this thesis explores the limits of these compression methods on current hardware, resulting in an optimized CUDA implementation with better than $100\times$ the throughput of prior work and achieving real-time operation on workstation-class hardware. However, after concluding that custom hardware is necessary for further improvement, this thesis also presents a hardware accelerator that nears real-time compression performance within a reduced power budget, outperforming an NVIDIA Jetson Orin Nano with 64% higher throughput while using 1/16th of the multipliers and drawing 38% of the power when running at 100MHz on an AMD UltraScale+ FPGA.

Thesis supervisor: Vivienne Sze
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Professor Vivienne Sze for supporting my work in the LEAN group for the last several years.

I would also like to thank Peter Li and Zih-Sing Fu, who have mentored me throughout my time in the LEAN research group, initially as a UROP and now as an MEng student, and have helped develop my academic writing skills. In addition, I would like to thank Dasong Gao for his help with understanding Gaussian splatting and its applications to SLAM.

Lastly, I would also like to thank my friends and family, my students, and anyone else who lets me give my two cents about my work and all things related.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

3D Gaussian splatting (3DGS) [1] performs novel view synthesis by blending the color contributions of a map of 3D Gaussian distributions. Compared to other novel view synthesis methods like neural radiance fields (NeRFs), it is less computationally expensive, with rendering speeds of $10 - 1000\times$ those of leading NeRF implementations [1], which has lead to recent popularity [2]. 3DGS's performance largely stems from its explicit representation of the environment; the map in 3DGS is an unordered array of parameters representing the position, size, orientation, and color of each 3D Gaussian. Representing obstacles directly through Gaussians makes both training and rendering in 3DGS more friendly to highly parallel hardware like CUDA-capable GPUs [1]. Even so, the algorithm is typically run on workstation-class hardware, and it has not been ported to smaller, lower power hardware. Offshoots of the algorithm have applications in simultaneous localization and mapping (SLAM) for robotics, which drives the motivation to make the algorithm feasible in the kinds of area and power-constrained hardware used in robots, especially small ones.

This thesis presents a solution to the memory size and bandwidth bottlenecks of the algorithm given said constraints through compression of the map accelerated by specialized hardware. The real-time compression performance required for SLAM can be achieved by re-implementing compression methods from prior work to best, and more directly, utilize current hardware. However, custom hardware is required to reach similar performance within the power and area restrictions of micro-robots.

1.2 Prior Work

In this section, we present prior work that has reduced the memory and compute demands of 3DGS through better optimized implementations, hardware acceleration of the rendering pipeline, and compression of the Gaussian map in software. However, current compression work has not been developed to operate in real-time as would be necessary for SLAM applications.

1.2.1 Optimized Implementations

The original 3DGS algorithm was implemented in Python, using PyTorch and CUDA for GPU-based acceleration of the compute bottlenecks [1]. There have since been various re-implementations and ports of the algorithm to C++ and other languages seeking to improve its performance, with some implementations touting up to 4× reduction in memory usage in training [3]. However, these ports do not fundamentally change the algorithm nor the hardware that is being used to execute it, which imposes a limit on further gains.

1.2.2 Hardware Acceleration of Gaussian Rendering

Some inefficiencies stem from the use of general-purpose GPUs for compute. For example, rendering ends for a given pixel once its α value exceeds a threshold, which leads to threads becoming inactive while they wait for the rest of the rendering tile to finish, meaning hardware is underutilized.

Thus, prior research has looked to solve this shortcoming so that the rendering pipeline can be deployed on smaller hardware than the workstation-class GPUs which the original 3DGS paper used. One such paper, called GSCore [4], was able to demonstrate 15.86× performance improvement over a NVIDIA Jetson Xavier NX-based system while using less than 2% of the on-chip area.

However, the GSCore design does not change the memory representation of the Gaussians in the map, which can remain in the hundreds of megabytes in size. So, especially given the increased rendering throughput capabilities of the hardware, the memory throughput and capacity of the systems stand to remain, or become more of, a bottleneck to deployment on constrained hardware.

1.2.3 Compact Gaussian Maps

While there has yet to be any hardware developed to compress the 3DGS map, there has been work to reduce the size of the Gaussian map in memory. Such papers have proposed multiple techniques: additional culling of redundant Gaussians [5][6], alternative representation of the spherical harmonic (SH) color parameters [5], and vector quantization of a subset of the Gaussian parameters, ranging from just scale and rotation [5] to all parameters aside from position and opacity [6][7]. For SLAM applications, where the number of Gaussians is already much smaller and SH parameters may not be included [8], the third shows the most promise for further gains.

Prior work has explored multiple forms of vector quantization (VQ) for compressing 3D Gaussians, which compresses Gaussian parameters by replacing them with references to an approximation of the parameter, called a *centroid*. There are various ways of determining an optimal set of centroids, but in its simplest form, VQ utilizes k-means clustering to generate centroids [6][7]. The reference, which is often implemented as a index into an array of centroids called a *codebook*, is smaller than the parameter itself, so the per-Gaussian memory cost is reduced. When the parameter's value is needed, it is decompressed by indexing into the codebook to find its assigned centroid, and returning that value in its

stead. These compression and decompression steps are also referred to as *encoding* the parameter into and *decoding* it from the codebook. In addition to more compact storage, using the compressed map also reduces the amount of data transferred from system memory through on-chip buses, which speeds up the memory-bound preprocess step [7] and reduces energy consumption.

Residual vector quantization (R-VQ), which reduces the total number of centroids by representing parameters as the sum of several centroids from different codebooks, has also been proposed [5]. Each set of centroids in R-VQ quantizes the error between the true Gaussian parameter and the quantized value only using centroids from previous stages. The training process for R-VQ, however, is more involved than k-means, and an index needs to be stored for each parameter per codebook, and thus index size, which determines the size of the compressed map, is often larger than if a single set of centroids was used.

There are, however, some limitations to current work on Gaussian compression that this thesis addresses. Firstly, it has been based of the original 3DGS paper's Python codebase and retains memory overhead that make it too large to run on systems with less memory, like a NVIDIA Jetson series device. Though there is some work that quantizes the map during Gaussian training [6][5], a non-quantized version of the map is still kept in memory, so the memory benefits of compression are not realized until after training of the 3DGS model is complete. This limits application to SLAM, which continually re-trains the map with every new keyframe from the camera. This shortcoming is, at least in part, because prior implementations have not achieved real-time compression performance, and thus the map is only encoded once per set interval, e.g. every 100 iterations of 3DGS training. If the Gaussians could be immediately encoded after every iteration, the non-quantized map would no longer be necessary. Decompression, i.e. decoding, of the Gaussians, on the other hand, is not a bottleneck to performance and thus is not addressed in this thesis.

1.3 Contributions

The principal contribution of this thesis is the design of a vector quantization engine that encodes 3 and 4-dimensional Gaussians parameters, which encompasses the scale, rotation and DC, i.e. constant, color parameters most critical in SLAM applications, to compress their representation in memory at minimal detriment to quality. In service to this, the following contributions were made:

1. The design of an optimized, high-performance implementation of k-means vector quantization on the current most-suitable hardware platform, i.e. GPUs via CUDA.
2. The design of an encoder core that can generate equivalent encoding results compared to software in a fraction of the clock cycles and memory accesses while also making efficient use of area resources.
3. The design of supplementary modules which aid to gather several encoder cores into a single compression engine that is software-controllable through a memory-mapped interface.

4. The deployment of said engine onto an FPGA development platform to measure compression speed and accuracy on parameters from live Gaussian splatting training.

1.4 Outline

This thesis is organized into the following chapters:

- **Chapter 2** introduces the 3D Gaussian splatting algorithm, its representation of 3D Gaussian distributions, and its applications to localization and mapping in robotics. It will also detail further the k-means vector quantization method from prior work that this thesis builds upon.
- **Chapter 3** establishes an optimized software baseline that is later used as the benchmark for the dedicated hardware.
- **Chapter 4** details the encoder core, which is the principal building block developed in this thesis to compress Gaussian parameters.
- **Chapter 5** describes the compression engine, which is built around an array of encoder cores with additional arbitration and glue logic to provide a memory-mapped interface to software.
- **Chapter 6** presents the verification methodology employed to check hardware outputs against software models in both RTL simulation and FPGA implementation. It also evaluates the power, performance, and area of the compression engine compared to the software baseline.
- **Chapter 7** summarizes the contributions of this thesis and previews future work that can improve upon this work.

Chapter 2

Background

In this chapter, we provide necessary background information about Gaussian splatting and its application to robotics in SLAM. This includes, in Section 2.1, the memory representation of 3D Gaussian distributions and, in Section 2.2, how Gaussian distributions are rendered to and trained from a set of 2D images. Lastly, in Section 2.3, we explain k-means vector quantization, which is the algorithm both the optimized software and specialized hardware implementations use to compress Gaussian parameters.

2.1 Gaussian Representation

In this section, we detail the representation of 3D Gaussians in memory according to the original work on 3D Gaussian splatting.

The map in 3DGS is an unordered array of 3D Gaussian distributions. Unlike in NeRFs, this map is explicit, i.e. each Gaussian is represented exactly by a set of parameters. Each Gaussian has a mean μ, covariance Σ, opacity α, and color c. In the original paper [1], the memory representations of each Gaussian parameter are the following:

- μ is directly stored as 3×32-bit floats representing the position of the center of the Gaussian distribution.
- Σ is derived from two sub-parameters:
  - 3×32-bit floats representing the scale s of the Gaussian along each axis.
  - 4×32-bit floats representing the rotation r of the Gaussian along each axis as a quaternion.

  Using the scale matrix S and rotation matrix R derived from s and r, respectively, $\Sigma = RSS^TR^T$.
- α is directly stored as 1×32-bit float representing the opacity of the Gaussian, used for alpha blending. The opacity at a given point x for the Gaussian is this opacity

times the Gaussian distribution sampled at that point:

$$\alpha(x) = \alpha \cdot \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right).$$

- c is derived from 48×32-bit floats representing the color of the Gaussian. These can be further divided into the DC color components (3 floats) that represent the constant color in each of the red, green, and blue channels, and the spherical harmonic (SH) color components representing the 1st (9 floats), 2nd (15 floats), and 3rd (21 floats) spherical harmonics of the color in each of the three channels.

In novel view synthesis models, there are typically on the order of $\approx 1,000,000$ Gaussians in the map, meaning that the map can occupy up to 700MB with the original representation of 236B per Gaussian [1]. For SLAM applications, the spherical harmonic parameters are sometimes omitted, and there are less Gaussians, typically around $50,000$, in the map, each with a memory cost of 56B per Gaussian due to the omission of SH parameters [8]. This results in a smaller map size of around 2MB, but even this can be too large for DRAM-less embedded systems with only kilobytes of SRAM.

2.2 Gaussian Rendering and Training

In this section, we detail the Gaussian rendering and training processes and how they are modified in SLAM applications.

The 3DGS algorithm consists of two main passes: a forward pass which renders a selected view from the current map, and a backward pass which computes the loss of the rendered image compared to its ground truth image and back-propagates the loss gradients to update the Gaussians' parameters.

Both the forward and backward passes can be further divided into a projection (a.k.a. preprocess) stage and rasterization (a.k.a. render) stage. In the preprocess stage, the 3D Gaussians are transformed into the view space given by the viewpoint and direction and then projected to a 2D Gaussian distribution as shown in Fig. 2.1. In the render stage,

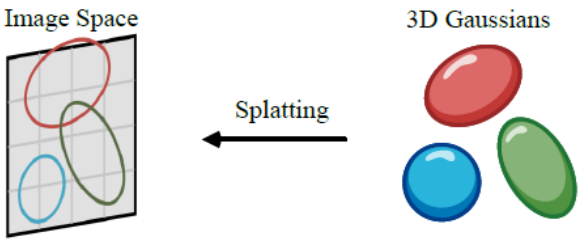


Figure 2.1: Depiction of projection of 3D Gaussians, represented as ellipsoids, to a 2D plane as ellipses [2].

Gaussians are batched into groups that overlap with a given screen-space tile, are sorted by

depth, and then the alpha-blended contributions of each Gaussian, from nearest to farthest, are summed to find the color C of each pixel in the tile via the following equation [1]:

$$C = \sum_{i=1}^{N} c_i \alpha_i \prod_{j=1}^{i-1} (1 - \alpha_j).$$

The backward pass computes the gradients through the same stages as the forward pass, except in reverse, and is about $2 - 4\times$ slower than the forward pass. Typical training for novel view synthesis consists of up to $30,000$ iterations of this process over $20 - 50$ minutes [1].

The training process for SLAM is somewhat different. Whereas for novel view synthesis the Gaussian parameters are trained based on a fixed set of images, in SLAM both the Gaussian parameters and the camera pose are trained based on a live feed of images from a monocular camera setup. As such, this process must occur in real-time. For every frame, of which there are 30 per second, the 3DGS algorithm is generally run 100 times to update the position of the robot (via the camera pose) in the tracking phase. In these iterations, the Gaussian parameters are held constant while the gradients of the error with respect to the camera pose are computed and back-propagated. In addition, two to eight of those 30 frames are identified as keyframes for which 3DGS is then run for another 150 iterations to update the Gaussian parameters in the map [8]. These mapping iterations differ from those in traditional Gaussian splatting, as forward passes through the entire window of training images are run with a single backwards pass afterward [8]. Each iteration thus requires more rendering, but the Gaussians are still updated once per iteration. This imposes a throughput requirement of up to around 4000 iterations of 3DGS per second for 30FPS operation, which has not been achieved even with high-end hardware, which has reached approximately 3FPS [8]. However, only at most 1200 of the iterations update the Gaussians, so the compression engine needs to reach a throughput of 60 million Gaussians encoded per second to re-encode the map after each update.

2.3 K-means Vector Quantization

In this section, we describe k-means-based vector quantization [6], which is the algorithm used in this thesis to compress several of the Gaussian parameters. The implementations of the algorithm vary, from that in prior work described in Section 1.2.3, to the optimized software baseline in Chapter 3, and then the specialized hardware in Chapter 4. However, in all implementations, the general principles of k-means quantization and how it is used to compress the Gaussians remain the same.

K-means quantization replaces a set of parameters with a set of indices into an array of *centroids* called a *codebook*. It is an iterative algorithm with two steps per iteration. First, parameters are assigned to and store the index of their nearest centroid in the codebook by squared Euclidean distance. The squared Euclidean distance d between two n-dimensional points a and b is given by

$$d = \sum_{i=1}^{n} (a_i - b_i)^2.$$

Second, the centroids are updated to be the average value of the parameters assigned to them. Thus, a centroid c that is the nearest centroid to n parameters P would be computed as

$$c = \left(\sum_{i=1}^{n} P[n] \right) / n.$$

This means that in addition to the indices of the nearest centroids to each parameter, the number of parameters assigned to each centroid must also be stored.

By the end of all iterations, each of the Gaussian parameters has been replaced with the index of the closest centroid in the final codebook, which has in turn been trained to be a decent approximation of all of the parameters assigned to it. So, the index, typically around 12 to 16 bits for a single parameter, replaces a 96 or 128-bit value, saving memory. Thus, the memory footprint of the map can be reduced by $6 - 10\times$ for the quantized parameters, or $3\times$ including non-quantized parameters.

Prior work has identified the index assignment step of k-means as the performance bottleneck to quantization [6], but there are no proposed solutions aside from quantizing less often, which is not an option when there is not enough system memory to store the non-quantized map between quantizations. In addition, it has also been shown that once sufficiently trained, the codebook can be held constant, and thus the second step can be omitted all together with only minimal detriment to the quality of the quantized 3D Gaussian map [6]. Thus, our encoder only performs the index assignment step of the algorithm.

Summary

In this chapter, we explained the memory representation of 3D Gaussian distributions, how Gaussians are trained from a set of input images, and how they are rendered to output images. We also introduced ways Gaussian splatting can be modified for usage in SLAM and how the memory representation can be compressed using k-means vector quantization.

Chapter 3

Software Baseline

In this chapter, we outline the bottlenecks to the performance of prior work, and we subsequently showcase an improved implementation that performs substantially better on the same hardware while achieving similar quantization results. Though prior work has demonstrated that k-means vector quantization of Gaussian parameters can effectively preserve rendered scene quality while reducing memory footprint, real-time performance has not yet been achieved. This is in part due to the nature of prior implementations, rather than the algorithms themselves. To confirm the necessity and efficacy of hardware acceleration, we first optimize existing software implementations on the most suitable existing hardware for the workload. We also provide an alternative algorithm which results in equivalent quantization results in fewer memory and arithmetic operations but find that it would be best implemented by dedicated hardware.

3.1 Choosing GPGPU and CUDA

In this section, we discuss the two types of hardware that are most often deployed in modern performance-critical workloads.

The first candidate is the central processing unit (CPU) of the system. Programming for CPUs is relatively simple: code can be written in a variety of languages, from higher-level interpreted languages like Python, to lower-level compiled languages like C++, and even directly in assembly for the given instruction set architecture. Required knowledge of the hardware implementation is dependent on the language used and its level of abstraction, e.g. Python generally requires less knowledge than C++, which requires less than assembly. Now, with modern out-of-order CPUs, even assembly fails to directly correspond to the actions taken by the hardware. Though originally designed to execute a single instruction at a time, current high-performance processors can execute multiple instructions per cycle and often include single-instruction-multiple-data (SIMD) extensions to accelerate parallel workloads. Many systems also include multiple cores, now sometimes reaching over 100 cores, which further improves parallel performance.

The second candidate is the graphics processing unit (GPU), which is present in some form

in all modern PC and mobile systems to render and display graphics to the screen. However, the GPU is also capable of being used more similarly to a CPU through a technique called general-purpose computing on graphics processing units, a.k.a. GPGPU. Tooling for GPGPU is generally less flexible than that for CPUs and is specific to the hardware vendor. The most widely used GPGPU framework is NVIDIA's CUDA, which executes C++-like code, packaged into functions called *kernels*, directly on compatible hardware, e.g. an NVIDIA RTX 4090 GPU. This gives the illusion to the kernel programmer of a general-purpose processor like a CPU, but instead, many instances of the kernel, called *threads*, execute in parallel on the GPU through its single-instruction-multiple-thread (SIMT) architecture. Threads are grouped into *thread groups*, which all execute in the same streaming multiprocessor (SM) allowing for inter-thread communication within the group and access to a 32-bank shared memory. Threads are further partitioned into *warps* of 32 threads, which are executed together in the SIMT units in the SMs. This being said, proper understanding of the SIMT architecture is required to obtain optimal usage of the GPU hardware, being perhaps more essential than with CPU programming.

With levels of parallelism that far exceed comparably sized and priced CPUs [9], and libraries providing interfaces to CUDA for Python, using GPUs through CUDA has become increasing popular for parallel workloads like machine learning but also Gaussian splatting [1] and quantization [6] [10] [5]. This is for good reason, but the details of said CUDA implementations sometimes leave performance on the table that a more optimized implementation could leverage.

It should be noted that a third, less but increasingly popular, alternative exists. Custom processing units, designed specifically for their workloads, are already present in consumer systems for common workloads like media encoding and decoding. By baking the algorithm directly into the hardware, which can be either an application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA), these accelerators trade off flexibility for maximum performance and efficiency. This thesis will later detail such a design in Chapter 4.

3.2 Existing Bottlenecks

In this section, we describe the existing bottlenecks in quantization implementations from prior work.

Two main factors render prior work as a subpar baseline for evaluating the performance improvement of hardware acceleration of the k-means process:

1. Prior implementations have built off of the original 3DGS codebase, which is written in Python, a high-level interpreted language, with PyTorch/CUDA acceleration for computation-intensive portions of the algorithm. However, bare-metal or RTOS applications are typically written in lower-level compiled languages like C/C++, which give more fine-grained control of device hardware.

2. Using PyTorch to implement the quantization algorithm, as done in [6], forces con-

structs like conditional logic, which can be expressed directly in the C++ dialect used in CUDA kernels, to instead be adapted into matrix operations, which introduces both memory and computation overhead. The breadth of tensor operations trades flexibility for performance, and work that could be achieved in a single CUDA kernel is broken into several smaller kernels, adding additional overhead.

Thus, for proper evaluation of a hardware-accelerated implementation of Gaussian compression, we require an optimized software implementation comparable to what would be deployed on existing hardware, i.e. GPUs, in performance-critical applications.

3.3 Optimized Baseline

The optimized quantization implementation is based on an existing port of Gaussian splatting from the Python and PyTorch-based codebase of the original to a C++ and LibTorch-based one that keeps the original's CUDA backend used to render Gaussians and backpropagate gradients [11]. The k-means algorithm is split into two CUDA kernels, one for each of the two main steps of each iteration, which directly access the underlying memory on the GPU of the LibTorch tensors containing the Gaussian parameters:

1. assign_indices: this kernel spawns one thread for every four Gaussians. The choice of four parameters per thread allows for centroid loads to be amortized such that utilization is equal between the GPU's load-store-units (LSU), which can support one warp per cycle, and fixed-multiply-add (FMA) units, which can support four warps per cycle. Each thread iterates through all centroids in the codebook, and tracks and stores the index of the centroid with the lowest squared Euclidean distance from its parameters. This process is shown for a single parameter in pseudocode format in Alg. 1. After it determines the closest centroids, each thread submits an atomic add operation to increment tallies of the centroids its four parameters were assigned to.

   Replacing a single parameter of a single Gaussian with the index of the closest centroid in the codebook will be referred to in this thesis as *an encode*. For example, 300 scale encodes would refer to searching for and returning the indices of the closest centroids for the scale parameters of 300 Gaussians.

2. update_centroids: this kernel spawns one thread per Gaussian. Each parameter is atomically added to a sum of the all parameters assigned to each centroid. The new centroid value then becomes this sum divided by the number of parameters assigned to the centroid, as tallied in assign_indices. To simplify inter-thread communication, the division is computed by a separate kernel which spawns one thread per centroid after all threads of update_centroids are finished.

Separating the two kernels allows for the steps of k-means to be executed asymmetrically, e.g. indices can assigned once every 10 updates to the centroids based on new parameter values from recent training iterations. This capability was also present in CompGS [6], and was leveraged to reduce training time at the cost of a minor reduction in final model quality.

The optimized CUDA quantization strongly outperforms the implementation from CompGS

Algorithm 1: Nearest-neighbor search iteratively through codebook array

Input: c, a D-dimensional centroid
p, a D-dimensional parameter
Output: d_{calc}, the squared Euclidean distance between c and p

```
1  function calculate_distance(c, p)
2      d_calc ← 0
3      for j = 1, ..., D do
4          d_calc ← d_calc + (c[j] − p[j])^2
5      end
6      return d_calc
7  end
```

Input: C, an array of N D-dimensional centroids
p, a D-dimensional parameter
Output: i_{best}, the index of the closest centroid in C to p

```
8  function search_iterative(C, p)
9      i_best ← 0
10     d_best ← ∞
11     for i = 1, ..., N do
12         c ← C[i]
13         d ← calculate_distance(c, p)
14         if d < d_best then
15             i_best ← i
16             d_best ← d
17         end
18     end
19     return i_best
20 end
```

[6], as shown in Table 3.1. Is also makes highly-efficient usage of the GPU's compute and memory bandwidth, evidenced by a utilization around 81% of peak throughput and equal utilization of the LSU and FMA units as measured by NVIDIA's NSight Compute tool. This

	Time per iteration per parameter (ns)		
Implementation	Scale (3-d)	Rotation (4-d)	DC color (3-d)
CompGS	373.72	373.72	373.72
Optimized CUDA	2.45	2.95	2.52

Table 3.1: Time elapsed per parameter per iteration of the k-means clustering algorithm implemented directly on CUDA versus the original implementation from CompGS [6]. Performance is averaged over 10 iterations of k-means run after 16000 iterations of 3DGS training with three parameter types: 3-dimensional scale, 4-dimensional rotation, and 3-dimensional DC color. Codebook size was set to 4096. Implementing k-means directly with CUDA results in, at minimum, a 127× reduction in time per iteration per parameter for all types.

implementation, which will be referred to in this thesis as the "optimized" or "CUDA" implementation, thus represents the best performance when can be achieved without modifications to the quantization algorithm or hardware platform.

3.4 Algorithmic Improvements

In this section, we showcase changes to the quantization algorithm, rather than the implementation or hardware, that can reduce the amount of memory and arithmetic operations required to quantize a single parameter.

Though this thesis' CUDA quantization implementation optimally executes the algorithm's nearest-neighbor search, i.e. how the closest centroid for a parameter is found, the search algorithm itself is not optimally efficient. In prior work [6], and in the optimized CUDA implementation, the closest centroid is found by iterating through all of the centroids and calculating the distance from the parameter. This process was shown in Alg. 1.

However, the same closest centroid can be found without doing full comparisons to the entire codebook. This requires a way of determining centroids with no chance of being the closest centroid given the current closest centroid in the search, which in turn generally mandates partitioning the data being searched, i.e. centroids, into a tree. Many tree variants exist, but for its relative simplicity, the k-d tree [12] was chosen as the tree structure to partition the codebook with. The k-d tree splits the centroids in half repeatedly along an alternating dimension, which will be referred to as the *split dimension*. The centroid with the median value in the split dimension becomes the parent node, with all centroids with values less than the median sorted to the left and all centroids with values above the median sorted to the right. For example, for 3-dimensional DC color parameters, the centroids would be sorted and split by their values in the red channel, then green, blue, and red again. A graphical representation of this process for 2D parameters is shown in Fig. 3.1. Given that the codebook will not be modified, the tree implementation does not need to support insertion and deletion

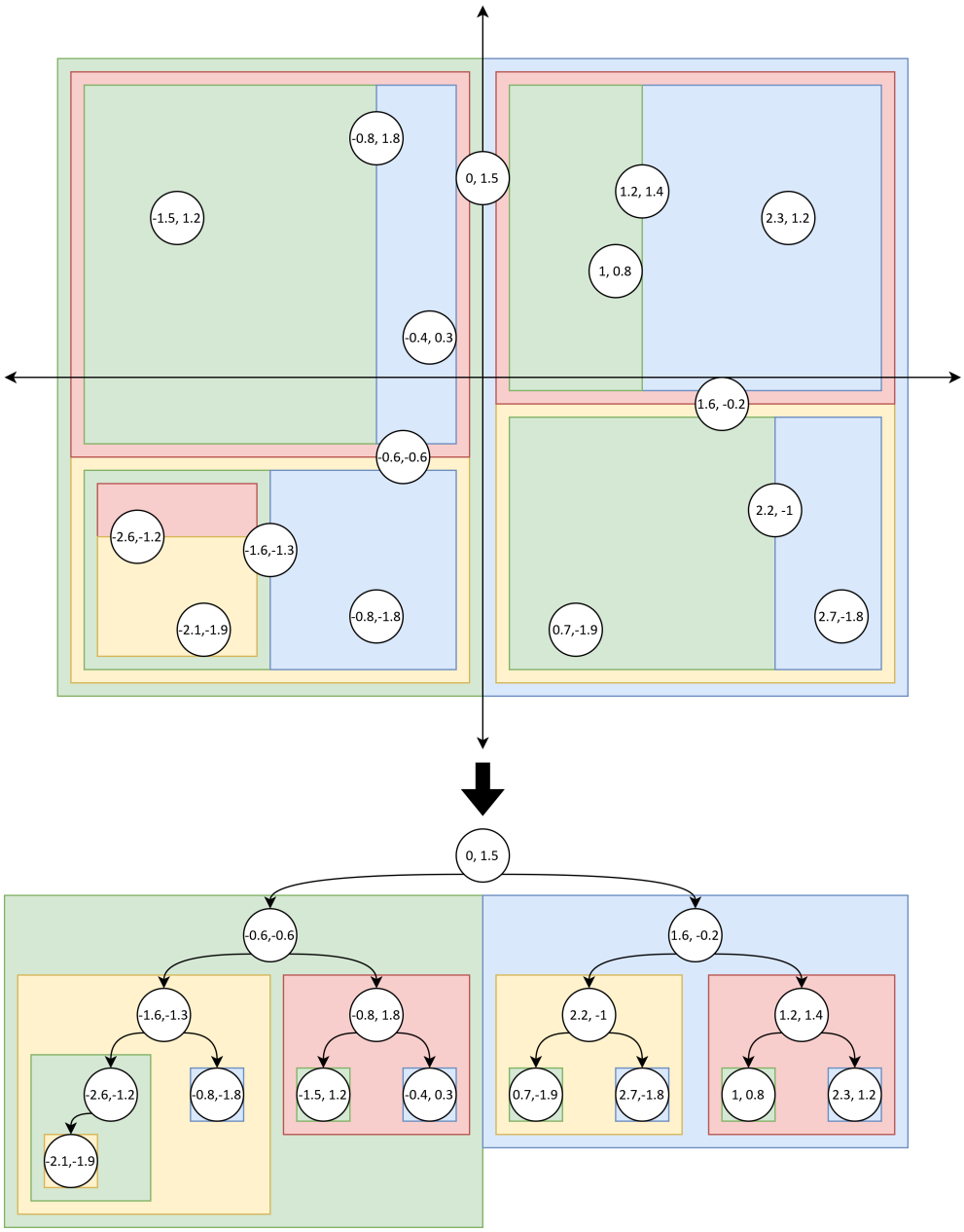
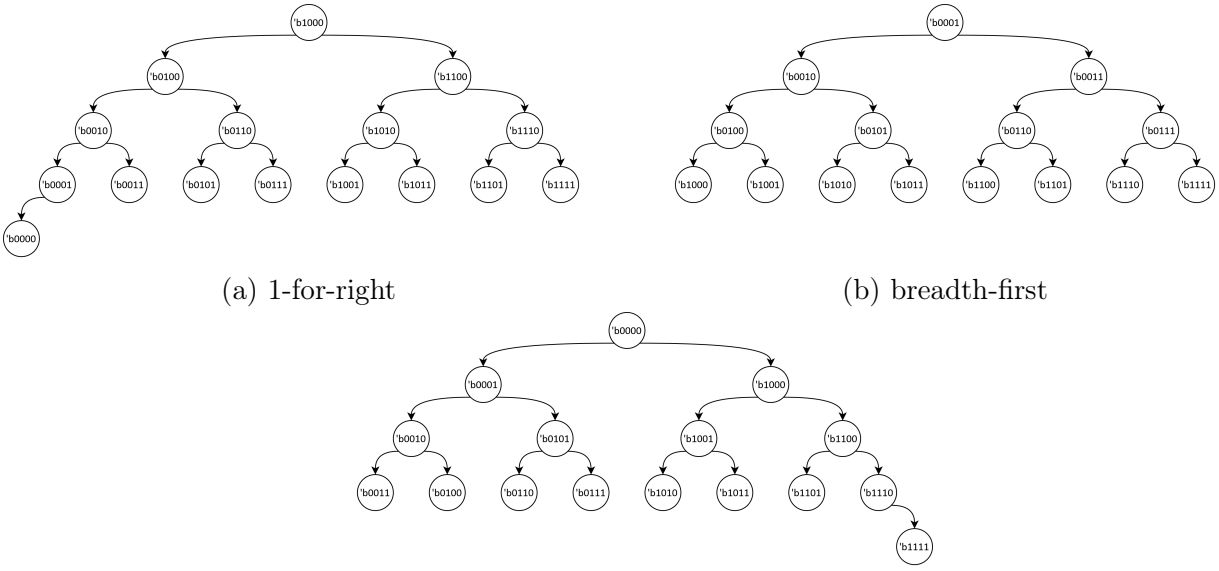


Figure 3.1: A 2-dimensional k-d tree, right, constructed from 16 xy points, left. Divides along the x direction are represented by the green (less than) and blue (greater than) regions. Divides along the y direction are represented by the yellow (less than) and red (greater than) regions.

operations; the k-d tree as implemented only supports two operations: first, creation from an array of centroids through the sorting and splitting process as previously described, and second, nearest-neighbor search given an input parameter to find the closest centroid.

3.4.1 Indexing

The limited set of operations allows for a more lightweight representation of the tree, where it is flattened and stored as a one-dimensional array with the tree structure represented in the indexing scheme rather than with pointers to nodes in a heap. The *depth* of a centroid in the codebook tree, which is defined as the number of edges between the centroid and the root of the tree, can also be ascertained by the index alone. Three indexing schemes, illustrated in Fig. 3.2, were considered:



(a) 1-for-right

(b) breadth-first

(c) depth-first

Figure 3.2: Three indexing schemes that flatten the codebook tree structure so that it can be stored as a continuous array. All are shown with a codebook size of 16 centroids, i.e. depth of 4. The 16-centroid codebook with breadth-first indexing is actually 15 centroids, or more generally $2^n - 1$ centroids instead of 2^n for a tree of depth n.

1. **1-for-right (1fr)**: this indexing scheme, shown in Fig. 3.2a, is the natural result of sorting and splitting the codebook in place to generate the tree. It stores which direction to branch at each depth in order to reach the centroid. For a 2^n-sized codebook, bit $n - 1 - i$ of the index, where bit 0 is the least significant bit, is set to 1 if the centroid is on the right of its parent centroid at depth i. For the bit corresponding to the depth where the centroid resides, it is always set to 1. All bits for further (higher) depths are set to 0.
2. **breadth-first (bf)**: this indexing scheme, shown in Fig. 3.2b, matches a breadth-first traversal of the codebook tree. In other words, the centroids at the top of the tree

occupy the lowest indices. If the root is stored at index 1, then for the centroid at a given index i, its left-child is at index $2i$, and its right-child is at index $2i + 1$, which can be implemented cheaply in hardware with a shift-and-add operation. Thus the root's children are stored in index 2 and 3, and their children are stored in indices 4 through 7. This scheme precludes having a centroid at index 0, so a 2^n-sized codebook is actually $2^n - 1$ centroids.

3. **depth-first (df)**: this indexing scheme, shown in Fig. 3.2c, matches a depth-first search of the tree. For a centroid at index i, its left-child is at index $i + 1$, and its right-child is at index $i + j$, where j is the number of children and grandchildren on the centroid's left. For example, for the root of a 16-centroid tree at index 0, its left-child resides at index 1 and its right-child resides at index 8.

Though the calculation of indices is more complicated in 1-for-right and depth-first, they have higher *locality*, defined here as a favorable property that similar centroids have similar indices, even if they are in different depths of the tree.

Regardless of indexing scheme, the stride between parents and children are depth-dependent, and additionally sometimes direction dependent, which complicates caching the tree with traditional caches. This stride is most rarely a power of two in breadth-first, followed by depth-first, so these designs perform better with a direct-mapped cache because parent centroids conflict less often with their descendants. However, because more performant cache architectures can be designed and tailored to the tree traversal pattern, this advantage is rendered moot. These cache designs are covered in Section 4.5.

More importantly, the indexing scheme should also be amenable to banking, where the codebook is divided among multiple memory banks that can each fulfill read requests simultaneously. Banking typically uses the bottom bits to select the bank, so the access pattern should result in an even distribution of the bottom index bits among the indices loaded. For example, the two least-significant bits would be used to select the bank in a 4-banked memory. In 1-for-right indexing, all centroids before the last two depths in the codebook tree have zeros for their two least-significant bits, which means that the whole top of the tree maps to the same bank, which is an undesirable result. On the other hand, in breadth-first indexing, each depth of the tree after the first two has its centroids evenly distributed between the banks. This results in a more even bank access pattern, which allows for peak memory bandwith to be supplied.

Thus, despite the superior locality of 1-for-right indexing, breath-first indexing was chosen for its friendliness to caching and banking.

3.4.2 Tree Search

Nearest-neighbor search through a k-d tree is easiest represented as a recursive process, akin to the pseudocode in Alg. 2. When implemented on a CPU, it results in a roughly two-orders-of-magnitude reduction in loads and distance calculations, which provides improved performance versus an iterative implementation on the same hardware. This reduction is visualized on a per-depth basis in Fig. 3.3. However, the additional index calculation logic,

Algorithm 2: Nearest-neighbor search through codebook tree

Input: C, a k-d sorted array of N D-dimensional centroids
p, a D-dimensional parameter

Output: i_{best}, the index of the closest centroid in C to p

```
1  function search_tree(C, p)
2      i_best ← i_root
3      d_best ← ∞
4      i_best, d_best ← search_recursive(i_best, d_best, i_curr, 0, C, p)
5      return i_best
6  end
7  function search_recursive(i_best, d_best, i_curr, s, C, p)
8      c ← C[i_curr]
9      d ← calculate_distance(c, p)
10     if d < d_best then
11         i_best ← i_curr
12         d_best ← d
13     end
14     if not is_leaf(i_curr) then
15         i_left ← left_child(i_curr)
16         i_right ← right_child(i_curr)
17         s_next ← (s + 1) mod D
18         if p[s] < c[s] then
19             i_best, d_best ← search_recursive(i_best, d_best, i_left, s_next, C, p)
20         else
21             i_best, d_best ← search_recursive(i_best, d_best, i_right, s_next, C, p)
22         end
23         if (c[s] − p[s])^2 < d_best then
24             if p[s] < c[s] then
25                 i_best, d_best ← search_recursive(i_best, d_best, i_right, s_next, C, p)
26             else
27                 i_best, d_best ← search_recursive(i_best, d_best, i_left, s_next, C, p)
28             end
29         end
30     end
31     return i_best, d_best
32 end
```

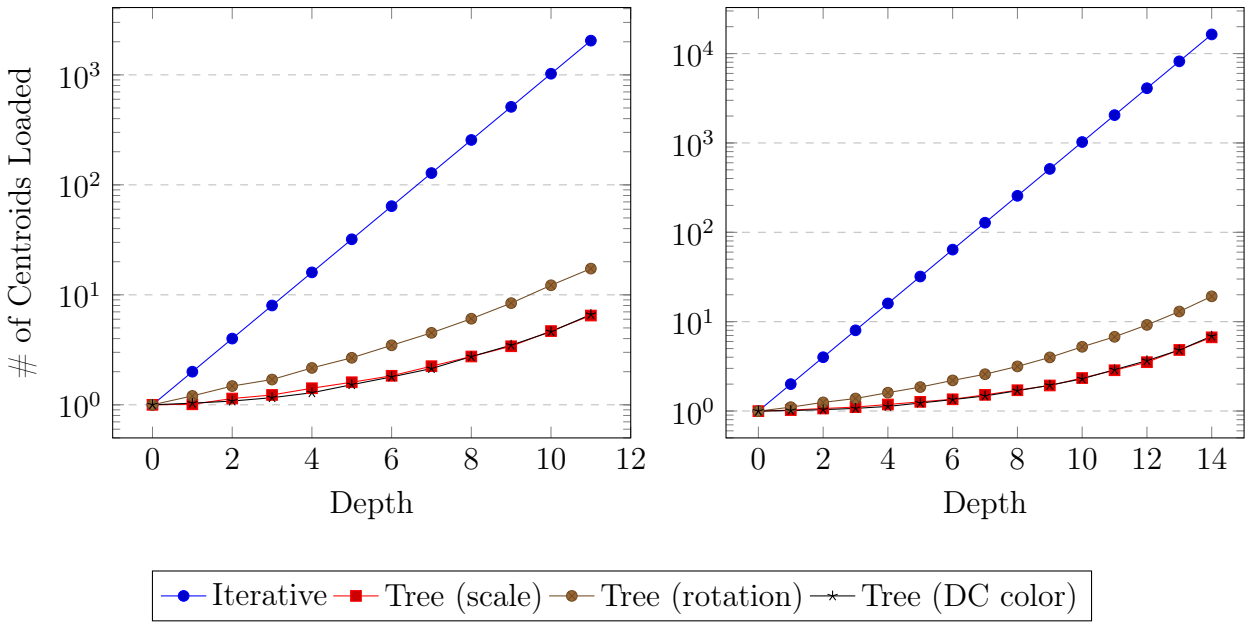


Figure 3.3: Average total number of centroids loaded at each depth of the codebook tree during an encode of a single parameter using codebook sizes of 4096 (left) and 32768 (right) centroids, i.e. depths of 12 and 15, respectively. Number of loads in tree search varies based on the type of parameter encoded and is compared to number of loads from iterative search. The 3-dimensional scale and DC color parameters benefit from a larger reduction in centroids loaded in tree search compared to iterative search than the 4-dimensional rotation parameter.

distance comparisons, and branching add some overhead, meaning that execution time per centroid loaded is increased. The overall performance uplift, measured in time per parameter encoded, is thus closer to one order-of-magnitude. The performance of the iterative and tree searches running on a single CPU core is documented and compared with the optimized CUDA implementation in Table 3.2. Despite comparing to between $65 - 141\times$ fewer centroids, the tree search implementation yields identical encoding results to the iterative search implementation, sans mismatches that less than once every several million encodes when two centroids tie for closest to the parameter.

3.4.3 Architectural Mismatch

Though the reduction in comparisons from a tree-based nearest-neighbor search leads to a performance improvement in CPU architectures in line with the reduction, this would not be the case if the same algorithm was ported to a GPU-based CUDA architecture. This is because the GPU single-instruction-multiple-thread (SIMT) model upon which CUDA is based does not map well to highly recursive and divergent code paths for several reasons:

- To start, CUDA's support for recursion is limited. It does have some support for recursion through its dynamic parallelism feature, which allows kernels to launch other kernel calls similar to how host C++ code does. However, this feature is optimized

	Scale (3-d)		
Implementation	Time/encode (ns)	Centroids loaded/encode	Time/centroid (ps)
CPU Iterative	7946	4095	1941
CPU Tree	318	29.1	10944
GPU Iterative	2.4	4096	0.6

(a) 3-dimensional scale parameters

	Rotation (4-d)		
Implementation	Time/encode (ns)	Centroids loaded/encode	Time/centroid (ps)
CPU Iterative	6994	4095	1708
CPU Tree	633	63.4	9998
GPU Iterative	3.1	4096	0.8

(b) 4-dimensional rotation parameters

	DC color (3-d)		
Implementation	Time/encode (ns)	Centroids loaded/encode	Time/centroid (ps)
CPU Iterative	7950	4095	1941
CPU Tree	326	28.8	11325
GPU Iterative	2.3	4096	0.6

(c) 3-dimensional DC color parameters

Table 3.2: Performance metrics of iterative and tree search implementations on each of the 3 quantized Gaussian parameters: 3-dimensional scale, 4-dimensional rotation, and 3-dimensional DC color. The codebook used was 4096 centroids trained on the drjohnson dataset [13]. The tree search implementation compares to about two orders-of-magnitude fewer centroids, but takes longer to do so per centroid. The highly parallel nature of the GPU implementation means that it outperforms both CPU implementations, even if it compares to more centroids than the tree search.

to support recursive paths that the thread group, or at least multiple threads in the group, take together. It thus does not serve as a good candidate for the case where each thread of execution is making its own traversal decision.

- Furthermore, branching in SIMT architectures is implemented with predication. In the original CUDA architectures, all threads share the same program counter and are thus executing the same instruction. To support conditional statements like if-else statements, whether a given thread in the warp *actually* executes the current instruction is determined by a masking bit per thread, which is set/reset upon entering and exiting conditional blocks. So, when a thread in the warp needs to execute an if-statement body to check another side of the tree, the other threads that wouldn't execute the body are forced to sit idle, reducing utilization of the GPU cores. In more recent CUDA architectures, threads in a warp can have their own instruction counter and call stack through CUDA's Independent Thread Scheduling feature, allowing for sub-warp scheduling. However, execution units are still SIMT, meaning that they are underutilized when heavy branching occurs even if the divergent branches can be

scheduled to different SIMT units at the same time.

- Lastly, divergent traversal paths cause divergent load patterns which coalesce less. Large codebooks are too large to store, in full, in CUDA shared memory. In an iterative search implementation, subsets of the codebook can be efficiently batched into shared memory because all threads are comparing their parameters to the same centroids at the same time. However, the same cannot be done when the threads could access any combination of centroids at the same time. This mandates loading centroids with global loads, which are most efficient when the loads coalesce, meaning that they access the same continuous 32, 64, or 128-byte chunk of memory. Threads that load different parts of the tree will make global loads to different sectors, meaning that the loads will coalesce less often, reducing effective memory throughput and bottlenecking performance.

Thus, the desired hardware-software combination, where an algorithm with a minimized amount of computation and data movement per encode is executed by highly parallel yet fully utilized hardware is not feasible with only general-purpose processors like CPUs and GPUs. In order to achieve the most efficient use of area, power, and time resources, a custom hardware accelerator for the tree search algorithm must be created, which is the topic of Chapter 4.

Summary

In this chapter, we showcased an improved software implementation of k-means vector quantization that directly utilizes CUDA kernels to make efficient use of GPU capabilities. In addition, we introduced nearest-neighbor search of a k-d tree as a method to reduce the amount of work required to encode parameters but explained why it cannot be implemented well with CUDA, instead requiring specialized hardware.

Chapter 4

Hardware Encoder Core

In this chapter, we introduce the encoder core, the fundamental unit designed in this thesis, which implements the k-d tree nearest-neighbor search algorithm directly in hardware.

The block diagram of the encoder core is provided in Fig. 4.1. The core executes a finite-state machine (FSM), which is introduced in Section 4.1, that is equivalent to the nearest-neighbor search algorithm through the k-d tree shown in Alg. 2. To support high throughput, the encoder core breaks the FSM into a pipeline with multithreading to mask pipeline latency, which is described further in Section 4.3. The pipeline is largely in-order, except for the out-of-order loader stage (Section 4.4), which fetches centroids to be compared to. To reduce the number of loads made to memory outside of the core, the loader also checks for hits in a centroid cache with a custom depth-strided indexing scheme, described in Section 4.5.

4.1 Tree Search FSM

In this section, we describe the tree search finite-state machine (FSM), which implements nearest-neighbor search through the k-d tree.

To efficiently adapt the tree search algorithm directing into hardware, it must be rewritten as a finite-state machine with a reduced and more rigid set of behaviors than the instructions supported by a CPU or GPU. The tree search FSM contains the following state variables:

- curr_param: this is the value of the current parameter being encoded, and is constant throughout the entire search process.
- curr_state: this is the current execution state, which determines what operations are performed on the other state variables in the current cycle. This takes the place of the program counter (PC) in a general-purpose processor.
- curr_idx and curr_centroid: these are the index and value, respectively, of the current centroid being visited in the tree traversal process.
- curr_depth and curr_split_dim: these are the depth of the current centroid being visited and current dimension that the tree is split on, respectively. curr_depth does

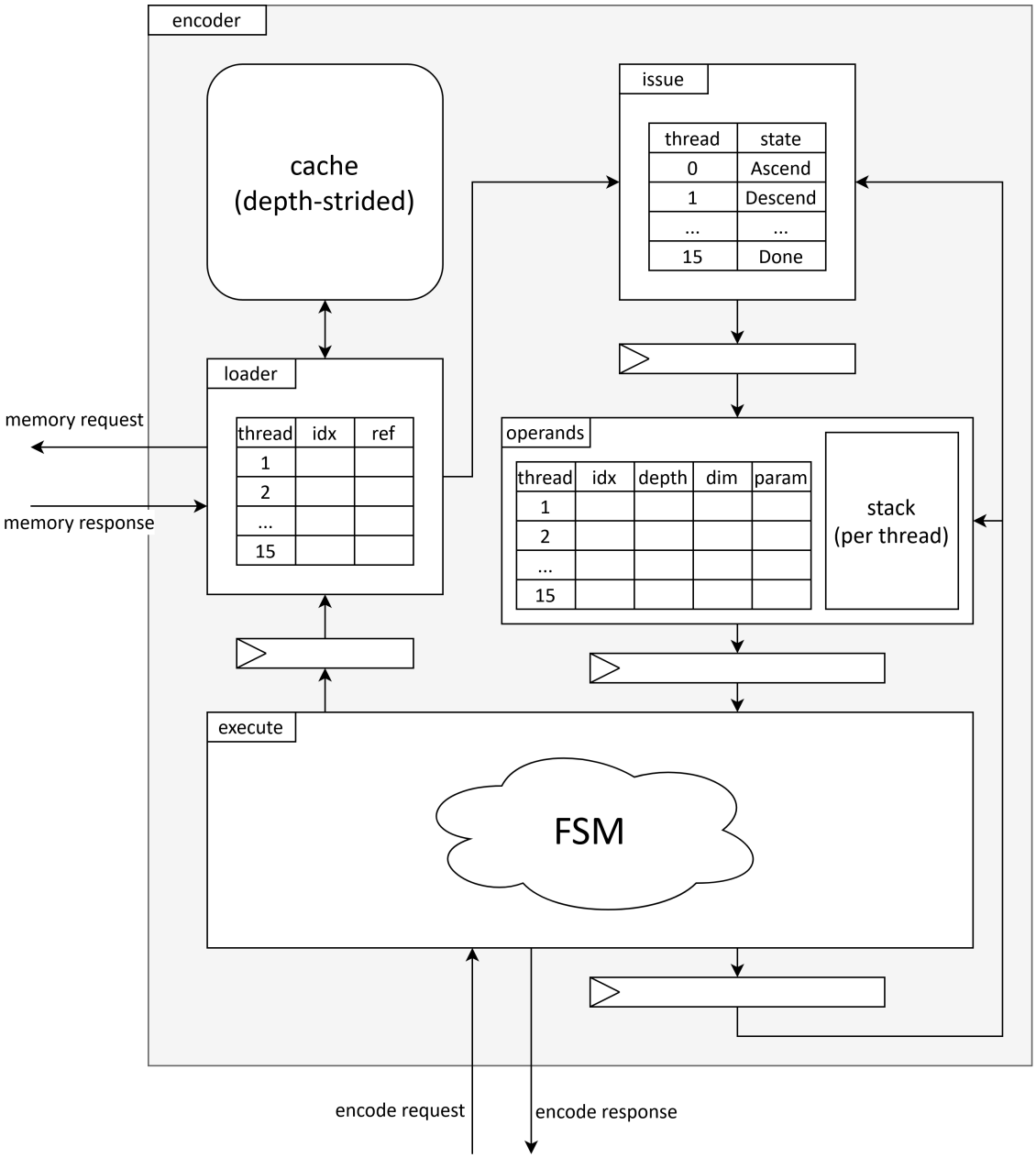


Figure 4.1: Encoder core pipeline with issue, operands, and execute stages and loader with cache. The issue stage chooses a thread to dispatch into the pipeline. The operands stage loads the per-thread FSM state, like current centroid index, the split dimension, and the param to encode. The logic of the FSM execution states, e.g. Descend and Ascend, is carried out in the execute stage. If the next execution stage is Descend, the loader must first fetch the centroid before the issue stage dispatches the thread again.

not exist in Alg. 2, but it is used to determine if the current centroid is a leaf of the codebook tree, analogous to line 14. Though the split dimension and leaf status of the current centroid could also be computed using curr_idx in some indexing schemes, it simplifies logic and is more flexible to have them managed separately.

- best_idx and best_dist: these are the index and distance from curr_param, respectively, of the closest centroid visited thus far in the search.
- stack: one feature of recursion that needs to be maintained is the call stack that is automatically appended to with each recursive function call. In the FSM, this stack contains an entry for all of the parent centroids of the current centroid which have not been revisited yet. Each entry contains the centroid index and depth, split dimension, distance between the parameter and centroid along the split dimension, and whether the encode traversed to the left- or right-child of the centroid after the first visit.

With the aforementioned state variables, the algorithm logic can be divided into four execution states: Idle, Descend, Ascend, and Done. These states incur the following behaviors:

- Idle: this is the idle state of the FSM, where no work occurs until a encode request is received. When one is, store the parameter to encode in curr_param, and then transition to the Descend state with curr_idx set to the index of root of the codebook tree, and curr_depth and curr_split_dim both set to 0. This thus roughly corresponds to lines 2 to 4 of Alg. 2.
- Descend: this replaces the portion of Alg. 2 before and including the first recursive call, i.e. lines 8 to 22. In this state, load curr_centroid from the codebook tree based on its curr_idx. Calculate the squared Euclidean distance between curr_centroid and curr_param, and if it is less than best_dist, it becomes to new best distance and the best_idx is set to curr_idx. If the current centroid is a leaf of the tree (as deduced from curr_depth), go to the Ascend state. Otherwise, if the parameter value along the split dimension is less than the centroid's, traverse to the left-child in the Descend state. If greater-than-or-equal, traverse to the right-child in the Descend state. In both cases where the next state is Descend, push an entry to the top of the stack for the current centroid.
- Ascend: this replaces the portion of Alg. 2 after the first recursive call and including the potential second recursive call, i.e. lines 23 to 29. If there are no entries left on the stack, move to the Done state to return the best_idx as the result of the search. Otherwise, pop the top entry off of the stack. If the distance from the centroid along the split dimension is less than the best_dist, traverse to the opposite, unvisited child, and move to the Descend state. Do not push an entry to the stack.
- Done: this is the done state of the FSM, where all work is finished. Return the encode response, which is the index stored in best_idx. Then transition to the Idle state.

The FSM's execution state transitions are shown in Fig. 4.2. One benefit of the FSM approach is that operations within the same execution state can be executed in parallel, e.g. child indices can be computed and chosen in parallel with the distance calculation from the current centroid to update the best distance.

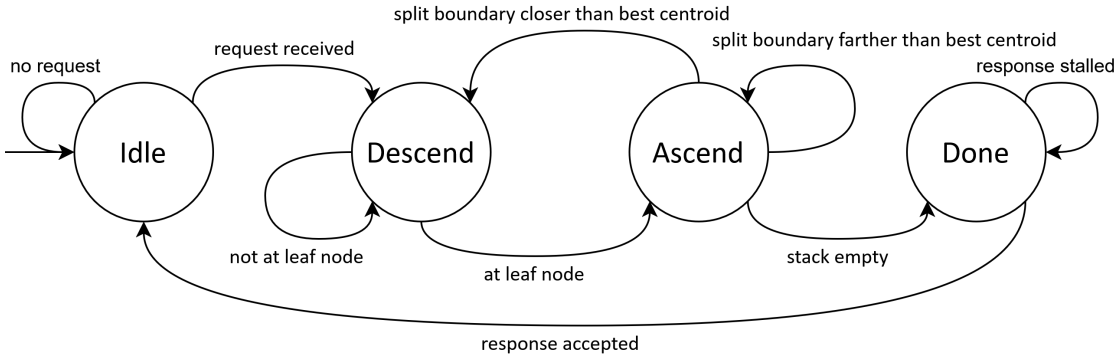


Figure 4.2: Diagram summarizing state transitions for the tree traversal finite-state machine. The FSM starts in the Idle state. Upon receiving an encode request, it proceeds through several transitions to Descend and Ascend until finally transitioning to Done. When the response is accepted, the FSM goes back to Idle to receive a new request.

4.2 Distance Calculation

In this section, we detail the distance calculation module, which is the main arithmetic module in the encoder core.

Another benefit of specialized encoder hardware is that it only needs to implement the subset of arithmetic operations required for the search algorithm. In this case, that primarily entails computing the squared Euclidean distance, as given in Section 2.3, in the Descend state. An architecture could choose to implement this sum of squares of differences in n dimensions in several ways:

1. **Folded / serial**: in this arrangement, the square distance in each dimension is calculated and added to the sum one dimension per cycle. This uses the least hardware resources, only requiring a single multiplier, but it has an n-cycle latency and throughput of $1/n$ per cycle.
2. **Pipelined**: in this arrangement, the square distance in each dimension is calculated and added to the sum one dimension per stage of the pipeline. This uses n distinct multipliers, but it has an n-cycle latency and throughput of 1 per cycle.
3. **Parallel**: in this arrangement, the square distance in each dimension are calculated in parallel and added within the same cycle. This still uses n distinct multipliers, but has a 1-cycle latency and throughput of 1 per cycle.

The parallel version, with the best latency and throughput, is the method of choice for the encoder distance calculation module. The pipelined version does have a slightly smaller critical path by only performing a single add per cycle, but the delay of multiplication typically dominates that of addition, so the benefit is insignificant.

A block diagram of the parallel 4-dimensional distance calculation module present in the encoder is shown in Fig. 4.3. Given that the maximum number of dimensions handled by the encoder is 4, in the case of the rotation parameter, it performs 4 differences in

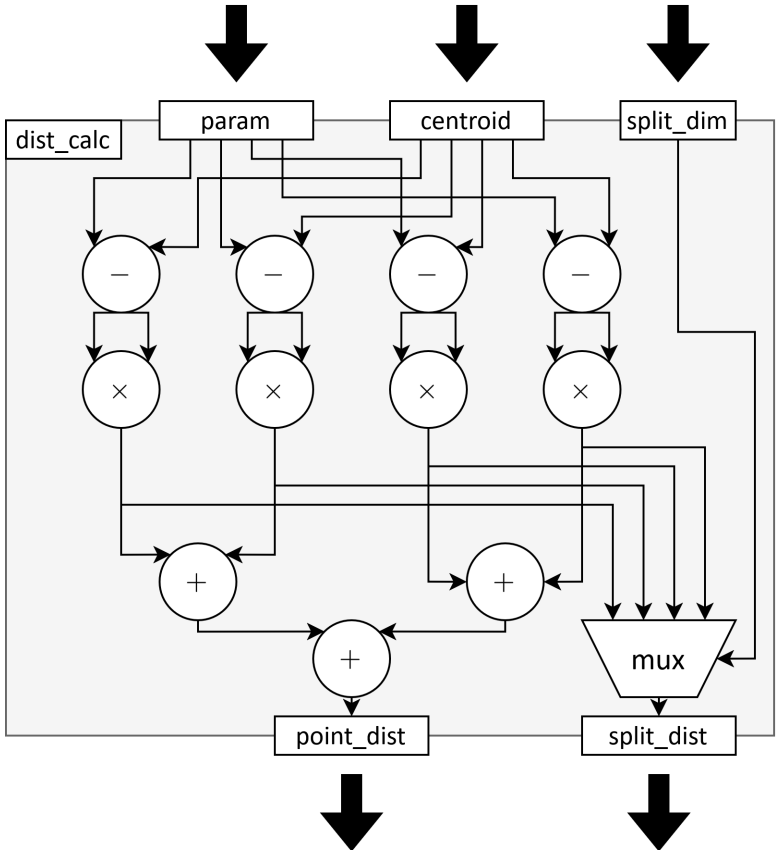


Figure 4.3: Distance calculation module taking in 4-dimensional parameter and centroid values, in addition to split dimension, as inputs. Subtraction and multiplication are performed in parallel for each dimension. The resulting squares of differences are then summed and muxed to obtain the squared Euclidean (point) distance and squared split distance outputs, respectively.

parallel, squares the differences, and then sums the squares. When handling 3-dimensional parameters like scale and DC color, the fourth lane is fed zeroes, and thus has no effect on the final result. This does impose an area penalty, as the fourth lane is not fully utilized, but it is a necessary trade-off to achieve full throughput on the throughput-limiting operation of the search algorithm.

The squared split distance, i.e. the squared distance between the parameter and the centroid along the particular split dimension, is also required so it can be pushed to the stack and popped later when the centroid is revisited. It is obtained by muxing between the products before summation.

4.2.1 Dynamic Fixed-point Arithmetic

In prior vector quantization work and in the optimized CUDA implementation, 32-bit floating-point arithmetic was used, as it is the primary method of representing fractional numbers in general-purpose hardware. However, when designing application-specific hard-

ware, where arithmetic units can be built to operate on arbitrary bit-widths, fixed-point arithmetic can act as a more efficient replacement for floating-point.

Fixed-point has the following advantages over floating-point:

- The location of the fixed-point is implicit in the representation, with tracking of its location done by the designer. Operations on fixed-point numbers are thus implemented identically to operations on integral values of the same bit-width. These operations have a shorter critical path and are more area and energy efficient than floating-point operations because they lack re-normalization and exponent-tracking logic.
- Fixed-point representations more easily allow for storing varying levels of precision between different operations. This can be leveraged to reduce area and energy cost where high levels of precision are not necessary.

There is, however, one disadvantage to the fixed-point representation, which is that finding an optimal location for the fixed-point is difficult, as it sets both the maximum value representable and the precision. To alleviate this issue, the fixed-point is dynamic between codebooks, but constant within a codebook. For example, for a shared 8-bit twos-complement representation, it could represent $[-16, 16)$ with a precision of $1/8$ in one codebook while representing $[-1/2, 1/2)$ with a precision of $1/256$ in another. The encoder core thus operates on fixed-point values normalized to the range of $[-1, 1)$, i.e. one negative integral bit to the left of the fixed-point and the rest, positive and fractional, to the right. Because distances are only compared within the same codebook (and thus between parameters and centroid normalized by the same factor), parameters never need to be de-normalized back to their original values.

The floating-point to fixed-point converter module was created using the Synopsys DesignWare DW_fp_flt2i IP [14] with the dynamic point implemented by subtracting from the exponent field in the floating-point value before feeding it to the converter.

Given the current goal of identical encoding results compared to the CUDA baseline, the fixed-point representation chosen was 32 bits wide, which is enough to express the 24-bit significand of the 32-bit floats with some extra headroom for variance in magnitude within the codebook. In addition, intermediate values are not rounded, and thus the operand size almost doubles after the squaring operation of the distance calculation. These two decisions have the drawback of increasing area usage, but they allow for hardware acceleration with no quality penalty. In future work, this design decision should be revisited.

4.3 Pipelining and Multithreading

In this section, we justify why the finite-state machine must be implemented with a pipeline and how we use multithreading to maintain peak pipeline throughput.

As shown in Fig. 4.1, the encoder core consists of a pipeline with four stages. To mask pipeline and memory latency, the pipeline is filled with states from 16 threads, each handling a single encode, that timeshare the pipeline. When the operands for its next state are ready, each

thread is issued from the issue stage and loads its operands, e.g. current the centroid and its index and depth, in the operands stage. It then executes the FSM logic and determines the next state in the execute stage, which is also where the distance calculation module resides. If the next state is Ascend, the thread is immediately marked as ready and skips the loader stage. Thus, for Descend-then-Ascend and Descend-then-Ascend transitions, the pipeline latency is 4 cycles. If the next state is Descend, the thread moves to the loader stage and is stalled until the centroid is loaded which takes a minimum of a single cycle on a cache hit but several more on a cache miss. So, for Ascend-then-Descend and Descend-then-Descend transitions, the pipeline latency is at least 5 cycles.

This pipeline is necessary because, in general, operands cannot be read and fetched in the same cycle. As discussed in Section 4.2, the distance calculation step can be implemented to complete in a single cycle, but the entire Descend state cannot, as it must also load the centroid from memory before the calculation and write to state variables and the stack afterward. This necessitates breaking up the state into a sequence of sub-states which perform the individual steps in consecutive cycles. However, in a typical FSM implementation, the throughput is inversely proportional to the latency. In other words, if one "execution" of the Descend state is spread across L cycles, then the FSM has a throughput of $1/L$ Descend states executed per cycle. Given that there is little hardware shared between the sub-states, this leads to an overall underutilization of the hardware and requires wasting area to achieve higher throughput, e.g. if running at 1/4 utilization, four arithmetic units must be instantiated to match the throughput of a single fully utilized unit. Most modern processors employ pipelining to achieve full throughput, such that each pipeline stage performs one step in executing each instruction. For the FSM, the equivalent is to give each step in executing either the Descend or Ascend state its own pipeline stage.

However, the next issue is that pipeline stages need separate instructions to execute. In a CPU, this amounts to speculating following instructions, and then stalling if data resulting from an earlier instruction is required to execute a later one. An analogous technique for the encoder would be to speculate future states and indices from the current ones, but every execution state in the FSM is like a branching instruction, unlike in CPUs where branches comprise a small subset of instructions executed. This makes accurately predicting future states to fill the pipeline nearly impossible.

Fortunately, speculative execution is not the only possible technique to fill the pipeline. GPUs, in addition to using parallelism within warps through SIMT execution, also heavily employ multithreading between warps, i.e. different warps are issued into the SM's functional unit pipelines in consecutive cycles such that instruction latency, which is typically 4 to 6 cycles [9], is hidden. This leverages Little's law, which states that

$$\text{throughput} = \frac{\#\text{ in flight}}{\text{latency}}.$$

By having many warps in the SM schedulers, the number of instructions in flight can be increased to match any increases in latency.

For the encoder, this means maintaining states for multiple encodes at a time, and scheduling encodes into the pipeline whenever their next state is ready to execute. This allows for

the latency of the encoder pipeline to be hidden, and the Descend and Ascend execution states can be broken into several pipeline stages without a performance or underutilization penalty. It should be noted that multithreading does not increase maximum throughput; having multiple threads increases the *maximum* number of instructions in flight, so that maximum throughput can be achieved even when latency spikes. If latency is low, then the number of instructions in flight will remain equivalently low, and thus throughput still hits the same cap.

Multithreading can also be used to hide memory latency. Memory latency is more variable than pipeline latency, which is only a function of the current and next execution state, as cache hits and misses and bank arbitration also play a role. For maximum performance, the number of threads should be sized to mask the worst-case average latency of the threads. This latency is somewhat unknown until implementation of the memory subsystem, but as an initial guide, an experiment was conducted measuring execute stage utilization, i.e. what percentage of cycles a valid state is executed, as a function of thread count and a specified constant memory latency. The results from this test are shown in Fig. 4.4. The experiment

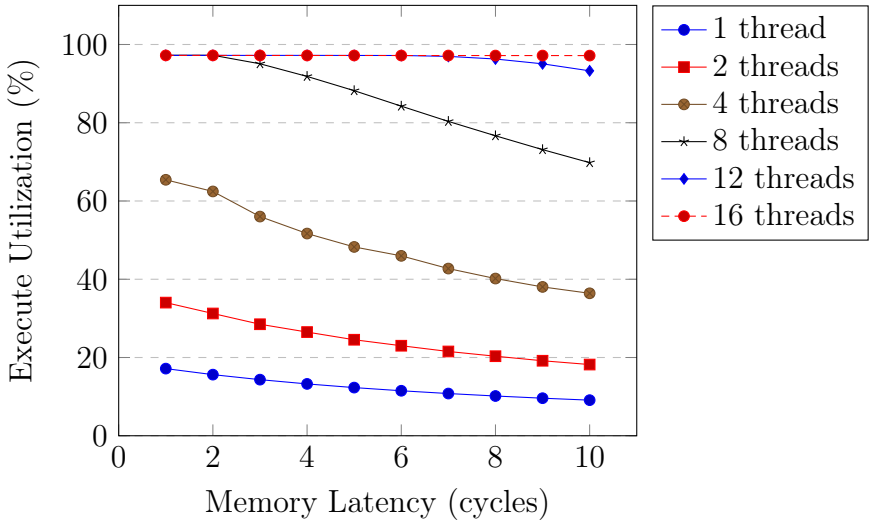


Figure 4.4: Execute unit utilization, as a percentage of total cycles, measured for several multithreading configurations (from single-threaded to 16 threads) and sweeping fixed memory latency from 1 to 10 cycles. The 1-, 2-, and 4-thread configurations fail to hide the pipeline latency. The 8-thread configuration hides the pipeline latency and up to two cycles of memory latency, while the 12-thread configuration hides up to six with a slower fall-off in utilization. The 16-thread configuration fully hides all memory latencies tested.

shows that given the pipeline as designed, the 1, 2, and 4 thread options are insufficient to hide pipeline latency and thus make poor choices for the thread count. On the other hand, the 8, 12, and 16-thread configurations all mask the pipeline latency and some amount of memory latency. Higher thread counts also experience slower decays in performance once they can no longer fully mask latency, as the denominator is higher, e.g. 8/8 to 8/9 is a steeper drop than 12/12 to 12/13. The 8 and 12-thread configurations do have the advantage of a lesser memory cost compared to the 16-thread version, but given that it can mask any

latency the core should plausibly encounter, the encoder core was designed with 16 threads.

4.4 Out-of-Order Loader

In this section, we describe the out-of-order loader, which fetches centroids required to execute the Descend state of the FSM.

In many cache implementations, only one pending miss is allowed at a given time, forcing other loads to stall until the miss is processed. This is often to avoid the additional complexity which comes with an out-of-order or non-blocking cache, where various buffers are required to restore program order and check for conflicts. For the encoder core, given that cache hit-rates can be as low as 40% of centroids loaded (see Section 4.5), cache misses are very common, so being able to support multiple misses and hits under these misses is crucial to achieving peak performance. Furthermore, out-of-order implementation is more simple because accesses from different threads are inherently unordered, and only loads are made from the codebook so read-after-write and write-after-read conflicts don't arise. The encoder core thus includes an out-of-order loader unit that fulfills centroid loads made during the Descend state and supports pending misses in all threads simultaneously.

Because the loader supports a pending miss for each thread, it never needs to stall due to a miss and it always capable of accepting a load request from the execute stage. When a load request comes in from a given thread, the loader first checks if the centroid requested is in the cache. On a cache hit, the centroid is immediately returned and the thread proceeds to execute the Descend state. On a cache miss, the loader first checks if there are any pending misses for the same centroid by making associative compares to the miss states of all of the other threads. If there are not, the loader makes a request for the centroid to external memory, and the thread miss state is updated to reflect this. If there is, the thread becomes a listener and stores a pointer to the thread which most recently asked for the centroid. If many threads ask for the same centroid in subsequent cycles, this grows to form something similar to a linked list. When the loader receives the centroid from external memory, it puts the centroid into the cache and broadcasts the centroid to all listening threads by unrolling the list, reference by reference, until the last thread which requested the centroid.

4.5 Caches

In this section, we present the centroid cache internal to the encoder core and explain how it differs from traditional caches found in CPUs and GPUs.

Given multithreading as discussed in Section 4.3, high execute utilization, and thus area-efficient throughput, is maintained as long as latency can be hidden by the number of threads and centroids can be delivered to the execution unit at sufficient bandwidth. Caching centroid loads can provide reduced average load latency, increased effective bandwidth, and improved energy efficiency, but given its irregular access pattern, the encoder benefits from a specialized cache design that differs from typical CPU and GPU caches. The cache in the encoder core

is a size-configurable design with a novel "depth-strided" mapping between centroid indices and cache lines.

Typical caches are build upon two types of locality, but neither is wholly applicable to the encoder's traversal process:

- **Temporal locality**: this asserts that if an access is made to a given address, then it is likely that another access will be made to that location in the future. However, a single encode never makes accesses to the same centroid twice, as each centroid is descended to at most once per encode. It is true, though, that if one encode loads a given centroid, that an encode of a similar parameter may too, but the delay between the first and subsequent loads may be longer than that in a CPU program.
- **Spatial locality**: this asserts that if an access is made to a given address, then it is likely that accesses will also be made to nearby addresses in the future. This is somewhat true for the encoder; if a parent is loaded, then one or both of the children will also be loaded. But given that the stride between parents and their children is depth-dependent, this can't be leveraged by increasing block size like in a CPU or GPU cache. The largest continuous segment of memory that is consistently accessed together is the parameter itself, so the block size should fit a complete parameter, with all components together, instead of each component in a separate line.

In place of temporal and spatial locality, two more apt properties of the encoder's load pattern can be observed and exploited by the cache design:

- **Path similarity**: this property is defined by how similar the set of loads made by different encodes are. This is similar to temporal locality, but there are some properties of the loading pattern that can be exploited. For example, given that the tree widens at further depths, paths are generally more similar at the beginning before they diverge. In addition, the most similar two parameters are, the further into the tree their paths will remain similar. In the extreme case, encodes of identical parameters will take identical paths through the codebook tree from top to bottom.
- **Access asymmetry**: this property represents how much certain centroids are accessed more often than others. To exploit this, a cache should not evict a frequently accessed centroid to make space for an infrequently accessed one. Replacement policies in set-associative caches often do something to this effect by evicting the least-recently-used (LRU) or least-frequently-used (LFU) line in the set. However, the access asymmetry is also baked into the search algorithm itself. As shown in Fig. 4.5, although more total loads are made further from the root of the codebook tree, individual centroids at the top of the tree are accessed very often compared to those on the bottom, e.g. the root is accessed by every encode and each of its children are accessed by at least half of encodes. It thus rarely makes sense to evict a centroid with a lower depth for a centroid with a higher depth, and a parent should never be evicted for one of its children or grandchildren.

Stemming from the consideration of these properties, five candidate cache designs, some common and some novel, were considered for use in the encoder cache:

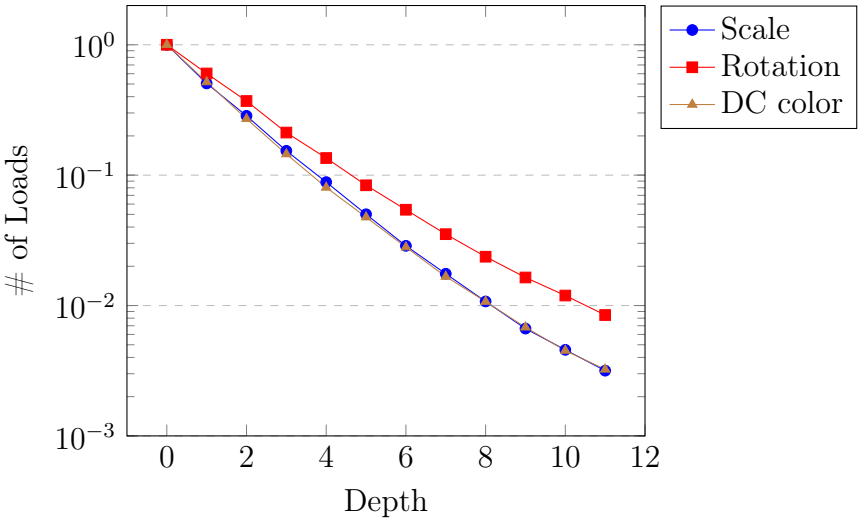


Figure 4.5: Average number of loads per encode of each individual centroid at a given depth, measured for each quantized parameter type using a codebook with 4096 centroids. For all parameter types, the individual centroids at the top of the tree, i.e. those with the lowest depth, are loaded the most because the number of centroids per depth grows exponentially. The fall-off is slower for the 4-dimensional rotation parameter because more loads are made per encode.

- **Direct-mapped**: this is the typical direct-mapped cache design, where the bottom n bits of the centroid index are used to select one of 2^n rows and check for a tag match. On a miss, the previous line in the selected row is evicted. This is the only of the five designs whose performance depends on indexing scheme, and it was tested here using breadth-first indexing.
- **Fully-associative**: this is the typical fully-set-associative cache design, where all lines are read for a tag match simultaneously. On a miss, the line to evict is chosen using the SRRIP replacement policy [15].
- **Depth-associative**: in this design, the depth of the centroid is used to select a set, and all lines in the set are read for a tag match. On a miss, the line within the set to evict is chosen using the SRRIP replacement policy.
- **Depth-strided**: in this design, the cache is arranged into a 2-dimensional array, where the row is the depth of the centroid and the columns are indexed by dividing the address by the stride between centroids at that depth. For example, for a 8-deep by 16-wide depth-strided cache, one of the 8 rows is selected by depth, while the column is selected by dividing the index by the depth-dependent stride, modulo 16. To save memory, the first couple depths of the tree are compacted into the first row, e.g. depths 0 through 3 with 1, 2, 4, and 8 centroids, respectively, would all reside in the first row of a 16-wide depth-strided cache. The stride between columns is dependent on the depth in some indexing schemes and thus adjusted for at each access, hence the name "depth-strided" cache. Similar to the direct-mapped cache, there is no associativity, so on a miss, the

previous line at the selected row and column is evicted.

- **Top only**: this design stores only the top of the tree, which can be indexed using the top bits of the centroid index in 1-for-right indexing and the bottom bits in breadth-first indexing. For example, a 2^n-centroid cache of this variety would use these n bits of the index to select a centroid from the first n depths of the codebook tree. On a miss, centroids are only placed into the cache if they are within the first n depths of the tree. In this way, valid lines are never evicted after being cached.

To benchmark caches designs before RTL implementation, three tests were ran using the encoder RTL testbench (see Section 6.3.1) to document hit-rates across various cache sizes and levels of path similarity and access asymmetry. These tests used C++ models of the cache designs that were connected to the encoder memory interface to monitor the encoder's load pattern, including both the tree traversal pattern but also interleaving between loads from different threads.

In the first test, whose results are shown in Fig. 4.6, parameters are sent to the encoder in random shuffled order, so there is minimal similarity between consecutive parameters and hence low path similarity between consecutive encodes. It also represents a case with both high short-term and long-term access asymmetry. This is generally pessimistic, but gives a good benchmark for minimum cache performance.

In the second test, whose results are shown in Fig. 4.7, parameters are sent to the encoder ordered by their nearest centroid. In the last test, whose results are shown in Fig. 4.8, parameters are batched together by the region of the codebook tree where their centroid resides, with order within the batch remaining random. Both tests represent situations with higher similarity between consecutive parameters and thus high path similarity compared to fully random encodes. In both cases, there is lower short-term access asymmetry that inevitably becomes high in the long-term. These situations may seem unrealistic, but other works have shown that nearby Gaussians have similar parameters [10], and even when that is not the case, prior encoding results or the first several centroids traversed to can be used as hints to batch or sort similar parameters together.

The direct-mapped cache performs well with high path similarity but very poorly in the opposite case. This is because the direct-mapped cache does nothing to leverage access asymmetry. Caches lines, even those which are often used like those at the top of the codebook tree, are evicted whenever a conflicting centroid is loaded. This is made worse by the fact that strides between parent centroids and their children are sometimes such that they all map to the same cache line, causing needless thrashing evictions during descension.

The fully-associative cache performs better that the direct-mapped cache, and in cases of higher path similarity, it can optimally allocate lines resulting in the best performance starting at 64 centroids in Fig. 4.7 and 256 centroids in Fig. 4.8. However, implementation of a fully-associative cache of these sizes is highly impracticable, requiring many associative compares and large amounts of logic dedicated to the cache replacement policy. In addition, its performance in the randomly ordered test is only somewhat better than the direct-mapped and worse than the other three options.

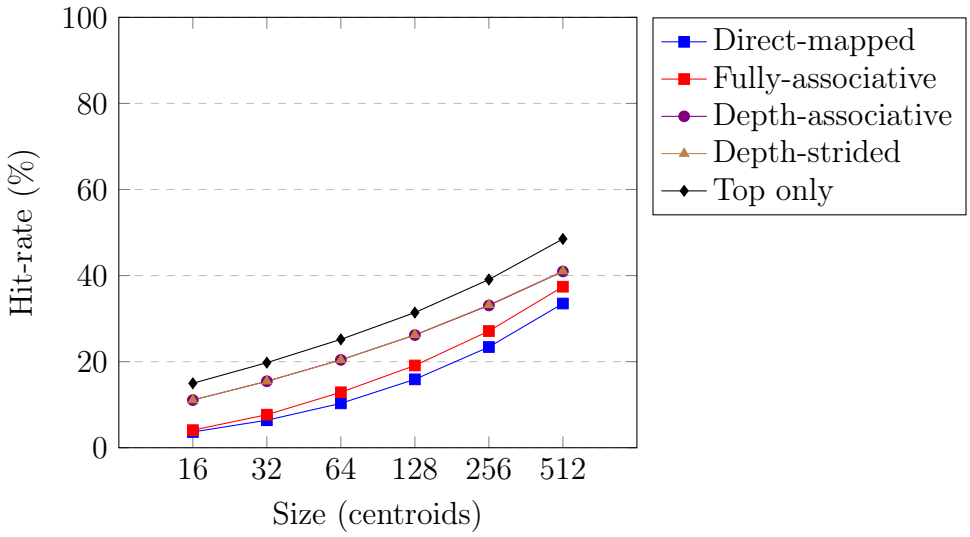


Figure 4.6: Cache hit rates with encodes randomly ordered. With low path similarity, cache designs that leverage access asymmetry perform better. Thus the top-only cache performs best, followed by the depth-associative and depth-strided caches.

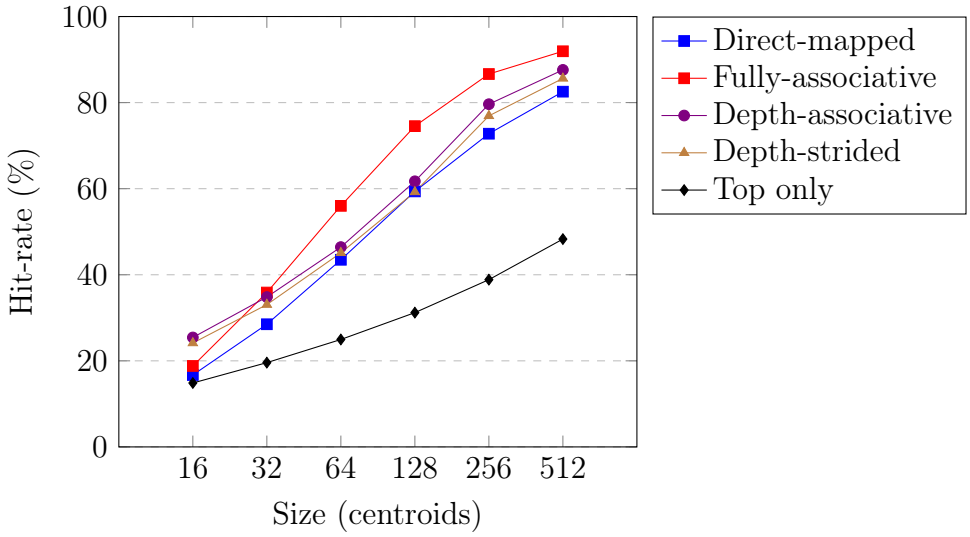


Figure 4.7: Cache hit rates with encodes ordered by their closest centroid, starting with those close to the leftmost centroids and proceeding to the right. With higher path similarity, all caches designs aside from the top-only cache see improved performance compared to randomly ordered encodes. The fully-associative cache sees the steepest increase in hit-rate as cache size grows.

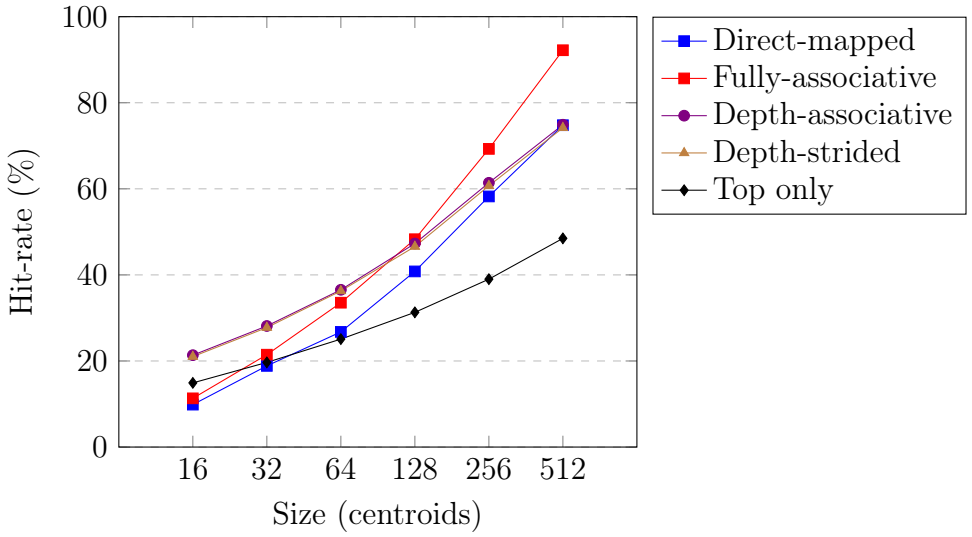


Figure 4.8: Cache hit rates with encodes randomly ordered, but batched into 16 groups of encodes that all encode to centroids in the same 1/16th of the tree. With higher path similarity, all caches designs aside from the top-only cache see improved performance compared to fully randomly ordered encodes. The fully-associative and direct-mapped caches see a faster increase in hit-rate as cache size grows than the others.

Unsurprisingly, the top-only cache performs identically in the three tests, as the top of the tree makes up the same amount of total loads regardless of encode order. However, it lags behind the other cache designs when path similarity is high, as it never caches any centroids beyond the first couple depths. It also scales poorly with size, as adding the next depth layer of the tree imposes an exponentially increasing memory overhead for a slightly better than linear hit-rate improvement, which is slower than the other four designs.

The depth-mapped, i.e. depth-associative and depth-strided, caches perform well in all cases. Because each depth is allocated its own row in the cache, there are no conflicts between parent centroids and their children. And, for the first several depths, the depth-mapped caches act similar to the top only cache, so they retain some performance when path similarity is lower. Perhaps surprisingly, the depth-strided cache was almost on-par with the depth-associative cache across the tests and outperforms the direct-mapped cache across the board. Given that it achieves similar performance while requiring less logic to implement than its associative sibling, the depth-strided cache is the architecture of choice and is implemented in the encoder core design.

Summary

In this chapter, we introduced the encoder core, which implements the k-d tree nearest-neighbor search through a finite-state machine. We described the main parts of the core, which include the core pipeline, execute unit with distance calculation and FSM logic, the out-of-order loader, and the centroid cache. We explained how we employ multithreading in

the encoder core to ensure that arithmetic units in the execute stage remain fully utilized despite pipeline and memory latency. We showcased the out-of-order loader, which exploits the lack of order between threads to reduce effective memory latency by allowing threads to be re-ordered after the execute stage. We also found that the encoder benefits from a custom cache architecture that is indexed by depth rather than directly by index or with associativity.

Chapter 5

Gaussian Compression Engine

In this chapter, we describe the hardware Gaussian Compression Engine, which is abbreviated as the GCE, and the additional modules, supplementary to the encoder core, of which it is comprised.

Overall encoding throughput, as encodes completed per cycle, is given by the following expression:

$$\text{encodes per cycle} = \frac{\text{operations per cycle}}{\text{operations per encode}}.$$

The number of operations per encode is determined by the searching algorithm, the dimensionality of the parameters, and the encoder architecture. By minimizing this number, like by grouping sections of the algorithm into single operations executed at once and by limiting the number of centroids visited by traversing a codebook tree, performance can be improved. These kinds of considerations were the focus of the encoder core design from Chapter 4.

However, once gains can no longer be made by decreasing the operations per encode, the next step is to increase the number of operations executed per cycle, which is in term given by

$$\text{ops per cycle} = (\text{max ops per core per cycle})(\#\text{ of cores})(\text{average core utilization}).$$

An encoder core can only execute one operation (i.e. execute a Descend or Ascend state) per cycle, so, to optimize encoder engine performance, we duplicate encoder cores while maintaining high core utilization, i.e. the percentage of cycles where the core executes an operation, above 90%.

The block diagram of the compression engine is given in Fig. 5.1. The engine includes a configurable number of encoder cores, which are paired into dual encoder units. These units load centroids from the codebooks memory and are configured through writes to configuration registers through a subordinate bridge. Encode requests to the cores are generated from a batch request by the encoder coupler and assigned to a dual encoder unit by the encoder splitter. The coupler also handles fetching parameters and writing out indices through the manager bridge.

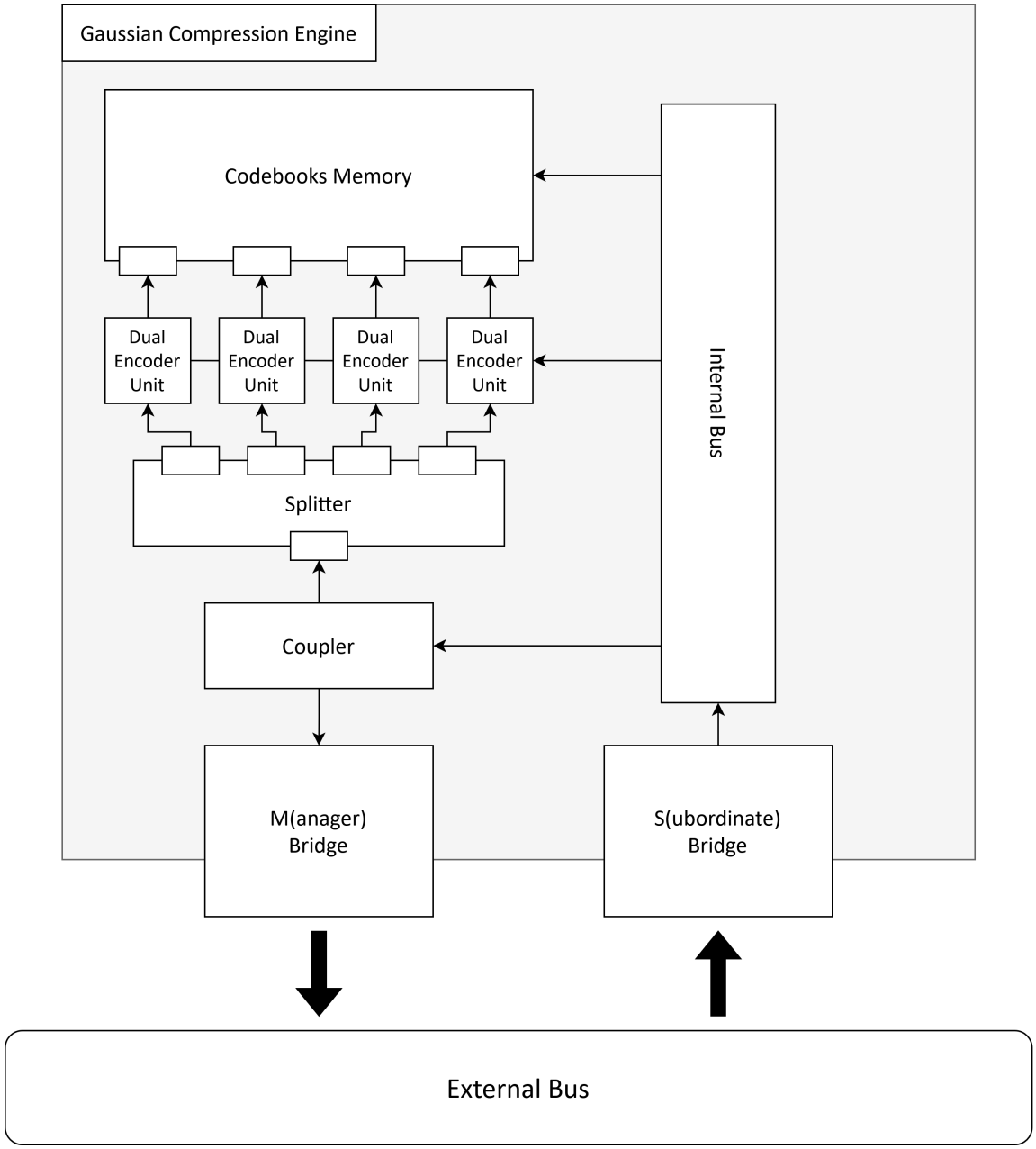


Figure 5.1: Overview of the Gaussian Compression Engine. The engine connects to an external bus via the manager and subordinate bridges. Encoder cores, grouped into dual encoder units, are controlled through the encoder coupler and splitter and fetch centroids from the codebooks memory through a dedicated port per dual encoder.

5.1 Dual Encoder Unit

In this section, we overview the dual encoder unit, which merges a pair of encoder cores to reduce routing complexity in large designs.

In the GCE, encoders are grouped into pairs of two encoder cores which share a memory interface but almost never starve each other of memory bandwidth. The near symmetry of the Descend and Ascend execution states places a limit on the portion of cycles where an encoder core will make a load request. Using the software tree search model, the ratio between the two states was determined to be about 4 Descend states for every 3 Ascend states, so an encoder will make load requests in less than 60% of cycles. With caches, encoders will make even less requests to memory outside the core, even with adversarial encoding orders where only access asymmetry can be leveraged. Grouping cores into dual encoder units helps reduce arbitration complexity between encoder cores accessing shared memory, which is shown in Section 6.5.5 to become a key area cost for high core counts. Because the cores can be made tolerant of memory latency by increasing thread count, additional latency resulting from arbitration stages and/or conflicting loads is masked and does not impose a performance penalty as long as the average memory throughput remains the same.

The implementation of the dual encoder is relatively simple. Internally, it consists of two encoder cores, each with configuration, encode request and response, and memory request and response interfaces in the module ports. For each request and response interface, there is a round-robin arbiter which determines which core is granted control of the single interface in the dual encoder ports. When a encoder core has a request granted, the request data is muxed onto the single outward request channel. The one exception is the memory response interface, where instead of an arbiter, an extra field, which denotes the core that made the corresponding request, is used to signal response valid to the correct encoder.

5.2 Internal Bus and I/O

In this section, we introduce the engine's internal bus and bridges, which are used to configure the submodules and provide access to and from the external system bus.

When placed in a system-on-chip, the Gaussian compression engine acts as both a *manager*, i.e. making requests and receiving responses, and a *subordinate*, i.e. receiving requests and providing responses. In this thesis, the terms manager and subordinate are used in place of master and slave, respectively, to align with the terminology used in the most recent versions of the AXI protocol, which is the external bus interface used in the FPGA subsystem the GCE is integrated into in Section 6.4.

The subordinate bridge converts 32-bit AXI-Lite requests into an APB-like interface for reduced logic complexity. Requests made to the bridge from the external bus raise the select line for a single module on the internal bus, which raises its acknowledge line when it handles the request. The address and write data lines are shared between all modules on the internal bus. This bus is not high-throughput, as it only supports single beat transactions, but that is not required because it is mostly used to write configuration register values. The

most bandwidth-intensive operations handled by the bridge are the writes to the codebooks memory. However, the current design assumes that the codebooks are only written once at boot-up, so this does not impact encoding performance.

The manager bridge handles requests from the encoder coupler to memory on the external bus. These requests can become throughput-limiting, so the manager bridge connects to the external bus via a full AXI interface with a 64-bit data width. This allows it to load up to 2 parameter dimensions and store 4 16-bit indices per cycle. As the encoder coupler is the only module that requests from external memory, there is no arbitration to the manager bridge. The full AXI protocol supports signals and operations that the encoder coupler does not need, so the bridge primarily functions as a translator between a more simple request format from the coupler with little additional buffering or FSM logic.

5.3 Encoder Coupler and Splitter

In this section, we describe the encoder coupler and splitter modules, which divides an encoding batch into several individual requests and assigns them to individual dual encoder units, respectively.

The encoder coupler module splits encoder batch requests into individual encode requests that can be handled simultaneously by separate cores. Though internally there may be many encoder cores present, the encoder coupler acts under the facade of a single encoder to allow for a flexible software interface that can be used for an arbitrary number of encoder cores. With multiple cores and multiple encodes per core, all without any mandated completion order, each encode request is sent along with a serial identifier such that encode responses can be re-ordered and stored in the batch order.

The coupler includes direct-memory-access (DMA) capability so that software can provide a pointer to the start of the parameters to be encoded rather than having to feed them directly to the accelerator. To make the best use of higher-latency memories, including DRAM, the coupler requests multi-transfer transactions that load several parameters at a time. Similarly, on the output side, resultant indices are also written directly, in batches of 64 indices per transaction, to memory starting at a second pointer configurable by the software. Additional information like the number of parameters, dimensionality of the parameters, and the fixed-point bias are also written to configuration registers in the coupler so that it can stitch together requests to encode fixed-point n-dimensional parameters sourced from memory, where they are stored in floating-point representation.

The encoder splitter converts the single encoder request and response interface from the encoder coupler into an interface for each of the dual encoder units. It uses round-robin arbitration to route each request to one of the dual encoder units, and it similarly accepts responses from the dual encoders with another round-robin arbiter. Round-robin arbitration was chosen for its simplicity and lack of stalling when encode times differ between cores, but it may not result in the best allocation of requests to core. As shown in Section 4.5, more intelligent arbitration provides an opportunity to increase cache hit rates by bundling similar parameters together to be handled by the same encoder.

5.4 Codebooks Memory

In this section, we introduce the codebooks memory, which holds the codebooks in their entirety and fulfills centroid load requests from all of the dual encoder units.

The codebooks memory is a multi-ported, multi-banked memory within the engine that is shared by all the encoder cores. Given that the encoder's multithreading hides memory latency, the limiting factor to encoder throughput becomes the memory bandwidth to the encoder cores, as this sets the rate at which Descend states can be executed. So, it is necessary to build a memory hierarchy that can support multiple encoders requesting centroids at the same time. The internal centroid caches within the encoders partially solve this issue, but given that miss-rates can remain around up to 50% for certain encode orders, encoders are likely to make simultaneous loads to the codebooks. Because the interface to the external bus is narrow compared to the centroid size and adds potentially unbounded latency, is necessary to instantiate an internal memory that can fully house the codebooks and can supply multiple centroids at a time. This is the purpose of the codebooks memory, which does so by splitting the codebooks across several memory banks and providing an access port to each of the dual encoder units and hence all of the encoder cores.

The block diagram of the codebooks memory structure is shown in Fig. 5.2. Each dual encoder unit makes requests to the codebooks through its own dedicated read port. Requests are fulfilled by one of the memory banks, which is selected using the bottom bits of the index. Requests to each bank are separated after the read port into a request FIFO for each of the banks for each port. In the event of multiple ports making requests which map to the same bank in the same cycle, the requests are granted using round-robin arbitration. Splitting the requests by bank before the arbitration allows for short-term asymmetries in bank utilization to be evened out, allowing the banks to remain fully utilized even if the ports receive conflicting reads. Furthermore, because backpressure is supported on the response, each port has a response FIFO for each bank which determines whether the port can make a request to the bank that cycle. The port is enabled to make a request to the bank if there is enough space in the FIFO for the new request and any previous requests which have not been returned from the memory yet. Parallel to the memory access, the encoder thread which made the request is propagated through a pipeline of the same length. This thread is passed back to the encoder core in the response, as the arbitration scheme described above does not guarantee that loads to different banks will be returned in the order they came, which becomes another functionality enabled by the out-of-order loaders in the cores.

Centroids are written into the codebooks memory through a single write port on the internal bus. There is no need for multiple write ports as the internal bus only supports one write at a time and because the codebooks are only written once upon initialization on system boot. A register value is set to determine the address to write, while another register address is used for the data to write. Centroids are provided to the codebooks memory as floating-point parameters, but they are converted to fixed-point with the codebook's given bias before being stored in one of the banks. This prevents centroids from needing to be converted on each load, which reduces dynamic power consumption. Given the same 32-bit width of both formats, there are currently no storage savings associated with the conversion, but reducing

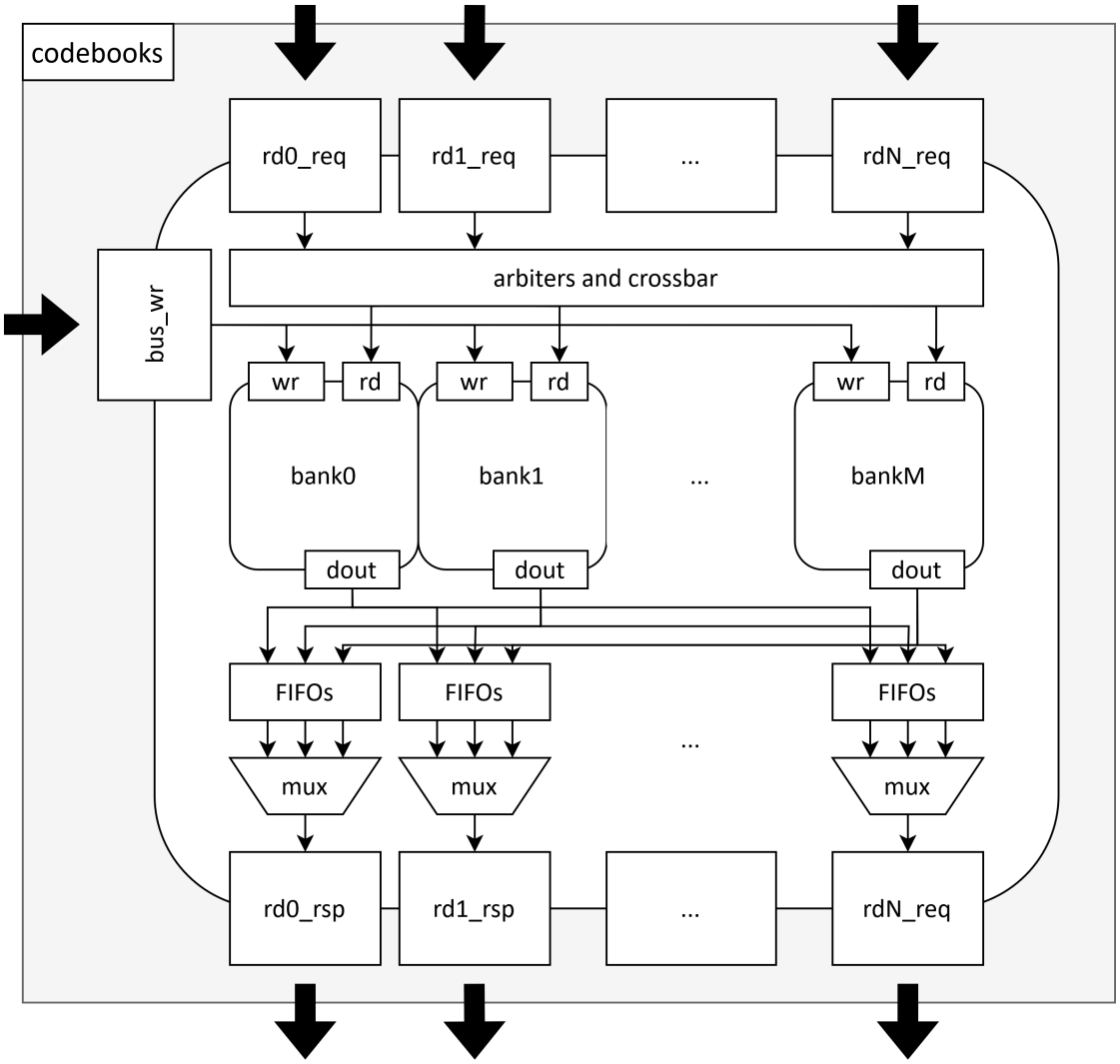


Figure 5.2: Codebooks memory with N read ports serviced by M memory banks. If centroid read requests evenly access the banks, the memory can support a bandwidth of up to M 4-dimensional centroids per cycle. Centroids are written through a single write port that connects to all banks, where they are first converted from floating-point to fixed-point representation.

the fixed-point width would allow for a smaller codebooks memory.

Determining which centroid indices correspond to which bank is crucial in minimizing conflicting bank accesses. Indexing using the top bits would be suboptimal because only one request can be made to each region of the codebook. Indexing with the bottom bits instead makes more sense, as if the distribution of the bottom bits is uniform, given that the requests on each port should not be correlated, this would generally conflict less often. However, this is only the optimal scheme if the distribution of values in the bottom bits is uniform across the load pattern. This is the case with breadth-first indexing but not with 1-for-right indexing.

Summary

In this chapter, we introduced the hardware compression engine, which integrates multiple encoder cores to increase maximum encode throughput through parallelism. We also described the helper modules that glue together the encoder cores and give an interface to control them from software.

Chapter 6

Verification and Evaluation

In this chapter, we detail various software models, including but not limited to software baselines from Chapter 3, the RTL simulation environment used to verify Verilog modules using these models, and the FPGA platform used to test sample configurations against the software models for correctness. Throughout software and RTL development, we performed continuous verification to ensure that software models were correct and consistent, and that hardware implementations returned identical results both in simulation and on the FPGA testing platform.

6.1 RTL Simulation

In this section, we describe the simulations conducted to verify the RTL written for the modules in this thesis. We ran all simulations using the industry-standard Synopsys VCS simulator.

6.1.1 Software Models

Two software encoder models, written in C++, were cross-referenced against the encoder outputs in RTL simulation. These models reuse much of the same logic from the CPU iterative and tree search implementations from Chapter 3, which were run on the CUDA-enabled workstation to inform decisions about indexing and caching. Having one iterative search model and one tree search model in the RTL testing environment allows for easier verification that the models and DUT are configured properly; if the tree and iterative search models mismatch, this indicates a fault in the setup sequence. However, these models are not entirely sufficient as is for RTL testing, as they only output the final encoding result, i.e. the index of the closest centroid.

Another important correctness metric is the path the encoder takes along the tree. If the path the hardware takes diverges from the software model, then 1) performance modeling from the software models may no longer valid and 2) the final encoding result may differ. These two consequences do not strictly imply each other, e.g. the traversal path could be the

same number of centroids but the wrong ones leading to an incorrect encoding, or the encoder could find the correct encoding by traversing more of the codebook than it should, leading to incorrect performance. Because there are many inputs to the traversal logic, traversal order also acts as a good marker, even on its own, of errors in logic outside of strictly the distance calculation and split dimension comparison. For example, a centroid being loaded incorrectly will cause traversal to an incorrect centroid despite correct arithmetic logic. Or, despite the correct centroid and correct arithmetic, incorrect indexing logic would also lead to the wrong centroid. So, the tree model used in RTL simulation also outputs a tree traversal order, i.e. a sequence of execution states and centroid indices, that is used to verify the encoder core's behavior.

However, there is one mismatch that does not imply an actual error in the encoder core logic. This being a difference in traversal resulting from variations between the floating-point arithmetic of the software model and the fixed-point arithmetic of the encoder hardware. To remedy this, the software model was modified to use the same fixed-point representation as the hardware while remaining otherwise equivalent to the floating-point tree search model. C++ does not support arbitrarily-sized fixed-point numbers out-of-the-box, so this was accomplished using the ac_fixed type from a modified version of the ac_types library [16].

6.2 Verification Methodologies

In this section, we overview the most common verification strategies for functional simulation of design RTL, and we justify the choice of UVM as the favored methodology.

For pre-silicon verification, there are several methodologies which have been used in Verilog-based environments for the past several decades. There are also non-Verilog-based verification environments, like SystemC [17] and cocotb [18], which were not considered given a lower level of personal familiarity. These methodologies are the following:

- **Verilog directed tests**: this is the old-school way of simulating designs. A testbench module is created that instantiates the design-under-test (DUT), and the sequences of stimuli into the DUT and subsequent checks are hard-coded by the programmer. This methodology requires a minimal amount of boilerplate, but it requires creating a new testbench whenever the values or order of input and output data changes.
- **Verilog random tests**: this is a more modern approach that builds upon the original Verilog directed tests. In this case, the testbench loads in a set of test vectors, which can be set manually or randomly generated. To support random test vectors, however, the reference outputs must now be generated by a software model instead of by hand. This may be implemented with direct-programming-interface (DPI) calls to a C++ model, with a Verilog model, or by running a C++ model on the test vectors before running the testbench. This methodology has slightly more boilerplate than directed testing, but can scale to support a wider range of tests with a singular testbench. In all cases, though, a fundamental restriction on the form and order of the data remains; though the values may be random, the input data must be driven, and the output data must arrive, in known sequences. In addition, a lack of compartmentalization can

make scaling the testbench and reusing verification logic more difficult.

- **SystemVerilog constrained random tests**: this methodology make use of the modern verification features added to the SystemVerilog language. Aside from quality-of-life additions on the RTL side of the language, SystemVerilog also supports more C++-esque types like classes and queues and new features like interfaces, clocking blocks and constrained randomization. Constrained randomization can be leveraged to generate greater coverage of possible DUT inputs, but with tight constraints, it can also emulate directed testing. This allows for greater code reuse and for easier modeling of complicated interleaving between inputs and outputs through transaction level modeling. Though these features could be employed directly to one's liking, they are often used in accordance to the Universal Verification Methodology (UVM), which is supported by all major SystemVerilog simulators, e.g. Synopsys VCS. Using UVM does admittedly incur a high boilerplate penalty to initial verification efforts, but it allows for a single testbench to be reused in many different, and perhaps dissimilar, tests of the DUT. It also accelerates development of system-level testbenches, as verification logic from unit-level testbenches can be reused.

UVM's flexibility and scalability makes it the most appealing option, as long as the initial work required to layout boilerplate can be justified. With some additional scripting, this burden can be lessened, and so UVM was chosen as the verification methodology for this work.

6.3 UVM Testbenches

In this section, we introduce the general structure of UVM testing environments as it pertains to their application in this thesis. UVM does not strictly require a given hierarchy or even that the programmer use any specific set of its features, so its application can vary between projects.

In the UVM methodology, and thus the verification environments created for this work, testbenches are generally organized in the following hierarchy:

- **Testbench**: this is the top-level SystemVerilog module given to the simulator.
  - **Clock and reset generation**: though this could be handled by a separate module, the testbenches directly generate the clock and reset signals that are passed to the DUT though an `always` process.
  - **Interfaces**: interfaces are a SystemVerilog feature which allows for grouping signals together and defining clocking blocks, i.e. subgroups of signals which are synchronous to the same clock signal. Signals in the interfaces are all defined as nets, instead of variables, so that the same interface can be reused to drive and snoop a set of signals. All of the interfaces for the given DUT are instantiated in the testbench module and passed to the DUT wrapper, where they are connected to the RTL, and testcase, where they are used in verification code.

  - **DUT wrapper**: this module has two purposes: first, it instantiates the DUT, or several DUTs in the case of a multi-module test, and second, it connects the DUT clock and reset signals, and the remaining ports with the interface nets, as only the interfaces are passed out from the wrapper.

  Lastly, in an initial process, it forwards the interface handles to the UVM configuration database so that they can be accessed by the verification logic. It then calls the proper UVM testcase, which is specified to the simulator through a command-line argument.

- **Testcase**: this contains all of the verification logic and inherits from the uvm_test class. The relationship between testcase and testbench is not necessarily one-to-one; there may by several testcases, which all require the same interfaces, for a single testbench that provides said interfaces to the DUT. The testcase instantiates the following:
  - **Environments**: the testcase does not directly drive any of the DUT's inputs or check any of its outputs. Instead, this is handled by a hierarchy of UVM objects which begins with an environment. The environment inherits from the uvm_env class. It may contain more environments, e.g. the DualEncoderEnv includes two EncoderEnvs. It also instantiates the classes which bear the brunt of the responsibility of driving, monitoring, and checking the DUT. These are the following:
    * **Agents**: agents in UVM, inheriting from uvm_agent, are generally created for each DUT interface and are comprised of three parts. All agents include a **monitor**, which is a class that observes activity on the interface and generates transactions, which are sent to the scoreboard through a dataclass, i.e. a SystemVerilog class that is used more like a struct. *Active* agents, defined as agents that drive signals on the interface, also include a **driver** and **sequencer**, which drive signals according to transactions and interface protocol and arbitrate control over the driver between several sequences, respectively. UVM includes a default parametrized sequencer, which was sufficient for all of the agents in this thesis. *Passive* agents, defined as those which only monitor the bus, do not need to initialize their driver and sequencer. Thus, all UVM agents include a flag in the configuration database to specify if an agent is being used as a passive agent or active agent in the given environment.
    * **Scoreboard**: there is typically one scoreboard per environment, inheriting from uvm_scoreboard, which takes data from all of the monitors in the environment and verifies that it matches expected DUT behavior.
  - **Global and local variables**: variables in UVM are typically passed between objects through the UVM configuration database. This database stores values by a key (a string) and scope (in the UVM object hierarchy, relative to the object accessing the database). This provides a mechanism for passing values through several layers of the hierarchy without exact knowledge of the internal representation of the destination object. If a scope is omitted when setting the value, then the value can be accessed by any UVM object, and the value acts essentially as a global variable.

With setup complete, the testcase runs a set of sequences, which encapsulate common series of transactions for better code reuse. Once all sequences complete and the environments drop their exceptions, the testcase finishes.

There are three testbenches at different levels of system integration: the lowest-level encoder core testbench, the middle-level dual encoder testbench, and the top-level compression engine testbench. These testbenches are covered in more detail in the following subsections.

6.3.1 Encoder Core Testbench

The encoder core testbench verifies the RTL of a singular encoder core. The block diagram overview of the testbench is provided in Fig. 6.1. The DUT wrapper provides access to three

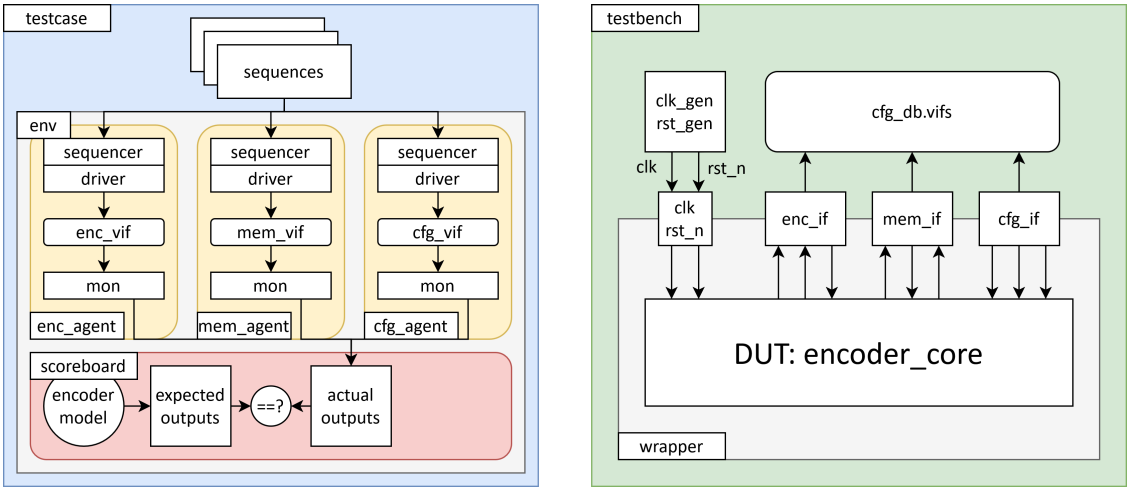


Figure 6.1: Encoder core testing environment with the encoder testbench and a testcase. A single encoder core is instantiated in the wrapper, with all of its interfaces driven by the verification logic. Each testcase instantiates the EncoderEnv class, which has agents for all three interfaces and a scoreboard to check encoder outputs.

interfaces on the encoder core:

1. cfg_if: this interface contains the signals for encoder configuration writes, which are ports on the DUT.
2. enc_if: this interface contains the signals for encode requests and responses, which are ports on the DUT. For additional correctness checking, it can also be connected to internal signals to track the path that the encoder is traversing, even though these are not ports on the DUT.
3. mem_if: this interface contains the signals for centroid load requests and responses, which are ports on the DUT.

Because it is a unit-level testbench that does not test integration with other modules, all of the agents are active and thus all signals into the DUT are driven by verification code through clocking blocks instead of by other RTL.

The configuration agent includes both a monitor and driver-sequencer pair and takes control of the cfg_if. The driver writes configuration to the encoder core configuration registers through the interface. When a configuration value it written, the monitor reports the new settings to the scoreboard, which updates the encoder model settings to match the settings in the RTL.

The encode agent is responsible for the enc_if. The encode monitor tracks all traffic on the request and response ports as well as every valid state which is sent to the execute stage of the pipeline. This allows for correctness testing using both the final output index and traversal path through the codebook tree. Checking the traversal path doubles as performance validation to make sure the encoder does not load more centroids than it should. The driver-sequencer pair drives requests to the encode interface and accepts all responses.

The memory agent has both a monitor and driver, but it lacks a sequencer because it only responds to requests from the DUT and does not drive sequences from the testcases. The memory driver responds to all centroid load requests on the mem_if using a codebook that is passed to the agent through the global configuration database. The memory latency can be set to be either constant or variable, and when memory latency is variable, memory responses are driven out-of-order according to their latencies. The memory monitor is not used for correctness testing, i.e. it does not send transactions to the scoreboard, but it monitors interface utilization and load patterns, which are used for cache modeling and testing.

The encoder scoreboard, which connects to the three monitors, houses the encoder models described in Section 6.1.1 and generates expected outputs from monitored inputs. The tree search model provides both the final index output from the encoder and the order that the encoder traverses the codebook tree centroids. As actual outputs come in from the monitors, they are compared to the expected outputs. The comparison is implemented with queues for both the expected and actual outputs. Because there is no mandated order between the execution of different threads, the queues are divided into queues per encoder thread.

This testbench and environment is used by several testcases. This includes correctness tests using both randomized and approximate Gaussian parameters generated from centroids taken from the CUDA implementation, the threads-latency-utilization tests, and cache modeling tests. In addition to sharing the same environment, i.e. the EncoderEnv class, these tests all share the same general sequence structure. First, a configuration sequence sets the encoder configuration, like codebook size, parameter dimensionality, and whether each encoder thread and the cache are enabled. Then, a request sequence sends a list of encode requests to the encode sequencer. The test concludes when both the testcase is finished with all its sequences and the scoreboard timeout triggers after a period without any activity on the DUT interfaces seen by the monitors.

6.3.2 Dual Encoder Testbench

The dual encoder testbench reuses many of the classes from the core testbench to verify the dual encoder RTL. Like the dual encoder module itself, it is a fairly simple testbench that combines two encoder environments to ensure that each acts properly when making and receiving requests through the dual encoder arbitration logic. This testbench was also

used to verify the hypothesis from Section 5.1 that two encoder cores could share a single memory interface without starving each other of bandwidth.

6.3.3 Compression Engine Testbench

The compression engine testbench verifies the top-level compression engine module. It thus includes and tests all of the RTL written for this thesis integrated together. It is also used for post-implementation simulation for power analysis done in Section 6.5.3.

The testbench checks the engine behavior by modeling and comparing with the AXI transactions made by the engine through the manager bridge to the external bus. The bus load requests are modeled to match the encoder coupler load pattern given the batch configuration written to the engine via the subordinate port. In addition, using the loaded parameters and the encoder models, the expected encode responses are generated and packed into a queue of expected bus store requests.

When run in RTL, i.e. not post-implementation, mode, this testbench also instantiates an EncoderEnv inside the GceEnv so that encoder core and compression engine bugs can be easier distinguished. This also provides a chance for further verification of the encoder internals under real usage by other RTL, as opposed to the more ideal interactions with the encoder core testbench. However, because net names are mangled in the post-implementation netlist, this feature is not supported in post-implementation simulation.

6.4 FPGA Testing

In this section, we present the FPGA testing environment, where the compression engine, implemented on an AMD UltraScale+ VCU118 evaluation board, was tested using Gaussian parameters trained in real-time on a CUDA-enabled workstation.

The block diagram of the testing environment is given in Fig. 6.2.

6.4.1 FPGA Subsystem

For FPGA testing, the compression engine is integrated into an existing SoC platform previously developed for testing hardware accelerator designs on FPGAs before tape-out to ASICs. The platform includes both a CVA6 RISC-V core by OpenHW Group [19] and an interface for control by a more powerful x86 workstation via PCIe, so it is hence referred to as the RISC-V subsystem. The RISC-V subsystem is based upon the CORE-V APU platform [19], which is also developed by the makers of the CVA6. The block diagram of the subsystem is given in Fig. 6.3. The subsystem is centered around a memory-mapped 64-bit AXI4 crossbar, which acts as the main system bus. Manager modules, like a CPU core, make requests to the bus by connecting to one of the crossbar's subordinate ports, while subordinate modules connect to one of the manager ports to respond to requests. Some modules, including the internal DMA and the compression engine itself, connect to the bus both as a manager and a subordinate. In the case of the compression engine, it acts as a

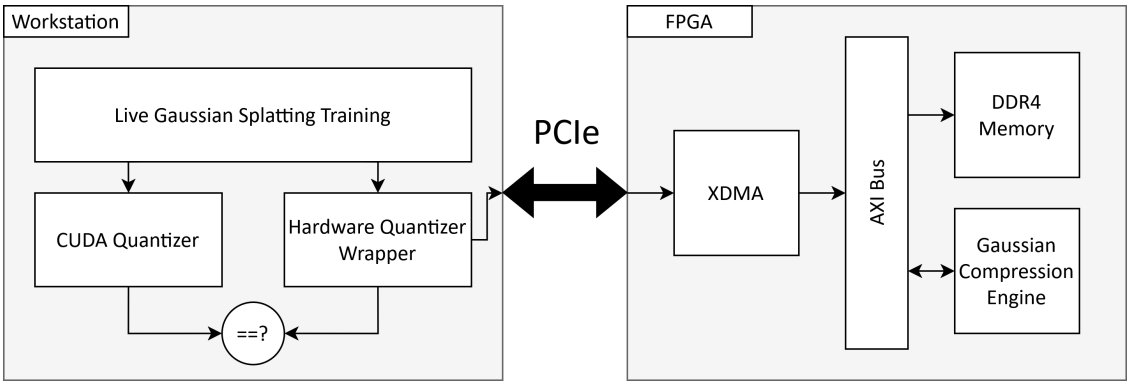


Figure 6.2: FPGA testing environment. Gaussian splatting training is performed with the CUDA quantizer from Chapter 3.3. In parallel, Gaussians are transferred over PCIe to the DDR4 DRAM on the FPGA board, quantized by the compression engine, and then output indices are transferred back to the workstation to be compared to the indices from the CUDA quantizer.

subordinate when its configuration registers are set, and it acts as a manager when it loads parameters from and stores indices to memory.

6.4.2 Reference Models

As opposed to RTL testing, where input parameters are either constrained random floats or constrained random variants of centroids, in FPGA testing true Gaussian parameters are used as the inputs. The CUDA-enabled workstation runs live 3D Gaussian splatting to train both the Gaussians and the codebooks. At a given iteration, most typically after 30K iterations of training, these codebooks and the Gaussians are transferred to the GCE's codebooks memory and the FPGA's DDR4 memory, respectively, via the XDMA IP on the subsystem. Then, batch settings are written into the encoder coupler and encoding commences. The workstation polls the completion state of the engine until the batch is finished.

Given that 3DGS training is already running on the workstation, the optimized CUDA baseline can be used as the reference model for the engine encoding results without any additional modification. The final CUDA indices lay in a PyTorch tensor residing in GPU memory, while final engine indices are stored in the DDR4 memory on the VCU118 board. The tensor is moved to CPU memory while the engine outputs are DMAed into workstation CPU memory over PCIe by the XDMA module, and they are subsequently compared.

6.5 Evaluation

In this section, we evaluate the performance, power, and area metrics of the compression engine. Though accuracy is essential, performance and efficiency are also important justifications for deploying a custom hardware accelerator on a real-world platform.

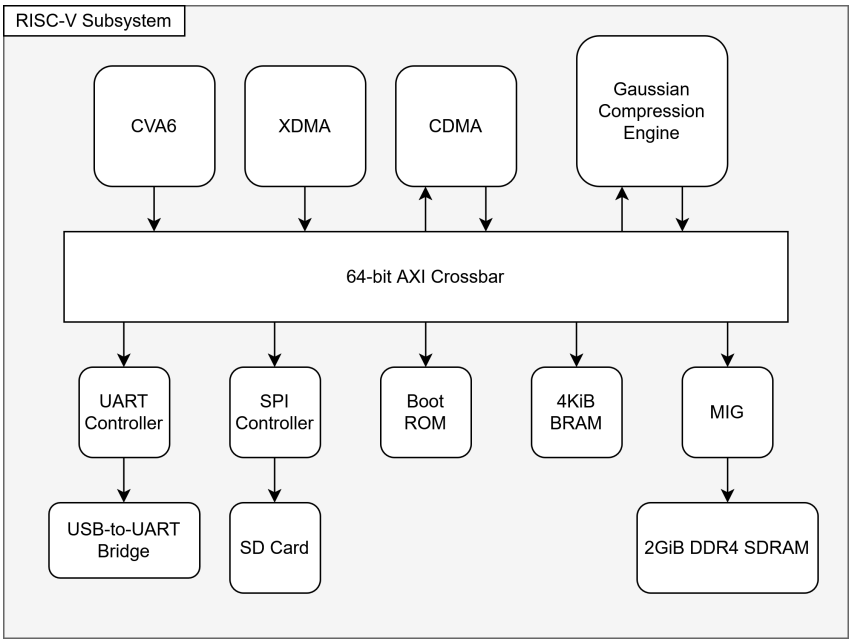


Figure 6.3: FPGA subsystem used to test the compression engine. Arrows point from AXI manager to subordinate. The compression engine connects to the central crossbar both as a manager and subordinate. During operation, Gaussian parameters are read from and codebook indices are written to the DDR4 SDRAM by the engine. The XDMA module transfers data between the subsystem and a workstation PC via PCIe.

Unless otherwise noted, the compression engine configuration tested is one with 8 dual encoder units (16 encoder cores, each with 16 threads and a 512-centroid cache), connected to a 8-ported, 8-banked codebooks memory.

6.5.1 Accuracy

As was the goal of the design, the indices returned by the hardware encoders are identical to those from the optimized CUDA baseline, with a couple minimal exceptions. Given the use of breadth-first codebook tree indexing, the encoder only searches $n - 1$ of the centroids in a n-centroid codebook, while the CUDA implementation, with its shared memory batching, cannot exclude the first centroid in the array from the search. Secondly, very rarely, i.e. about once per several million encodes, there are centroids that tie for closest to the parameter, and thus traversal order, which differs between iterative and tree search, influences the output index. In both of these cases, however, this is an algorithmic artifact, not an hardware implementation bug, as the tree search CPU model returns the same results as the encoder core. These differences in encoding are thus both rare and inconsequential when they do occur.

6.5.2 Performance

Performance is generally measured by throughput and latency. In the compression engine, the batch is only considered finished and ready to access when all of the parameters are encoded. So, latency, i.e. how long it takes to encode a batch of parameters, is directly related to throughput, i.e. how many parameters are encoded per unit time, by the following:

$$\text{latency} = \frac{\#\text{ of params}}{\text{throughput}} .$$

So, engine throughput, and thus average encoder throughput and utilization, can be ascertained by measuring the encode latency for a batch with a known number of parameters.

Performance testing was performed on several datasets [13] [20] [21] using the FPGA testing environment, as described in Section 6.4, against the optimized CUDA baseline running on an NVIDIA RTX 4070 Ti SUPER. Batch encode times, in microseconds, were measured using a CUDA event timer in the workstation code for the CUDA implementation and by reading a internal performance counter for the GCE. Using the counter eliminates delay from DMA transfers between the workstation and the FPGA in the measured latency, as a real system on a micro-robot would have the compute and the compression engine on the same unified bus. The performance counter gives latency in cycles, which is converted to microseconds using the 100MHz main clock frequency. As seen in Table 6.1, the GCE is about an order of magnitude slower than the CUDA implementation in terms of encodes per second. However, given that the 4070 Ti SUPER has a clock speed of 2.34GHz compared to the 100MHz clock speed of the RISC-V subsystem, the GCE has a similar throughput per cycle.

It should be noted that the 4070 Ti SUPER is not a true competitor to the GCE, as it is a part with a 285W TDP, making it far outside the limits of a micro-robot. A more comparable competitor is the NVIDIA Jetson Orin Nano, whose 8GB configuration has 8 SMs running at 625MHz, as opposed to the 4070 Ti SUPER's 66 SMs at 2.34GHz, and a TDP configurable between $7-15$W. Unfortunately, the Orin Nano is not capable of reliably running both 3DGS training and quantization on novel view synthesis datasets without running out of memory. So, the throughput of the Orin Nano was measured after only 5K iterations of training on just the drjohnson dataset, and thus it encoded a fewer number of Gaussians with different codebooks, so the measured throughput is not entirely comparable. However, assuming it achieves the same compute utilization as the 4070 Ti Super, the Orin Nano's performance under the same conditions as the GCE and 4070 Ti SUPER can also be approximated by scaling the 4070 Ti SUPER's throughput down by the reduced SM count and frequency. These two throughput metrics are compared to that of GCE in Table 6.2. The GCE outperforms the Jetson across all parameter types with a minimum of 19%, maximum of 83%, and average of 64% higher throughput.

As discussed in Section 2.2, the encoding engine needs to be able to support 1200 updates to the map each second, or 60 million Gaussians, each with scale, rotation and DC color parameters, encoded in order to reach real-time compression of the map. The optimized CUDA implementation, when run on the 4070 Ti SUPER, surpasses this threshold. Meanwhile, the

		CUDA		GCE	
Dataset	# of params	Latency (µs)	Throughput (s^{-1})	Latency	Throughput
drjohnson	3358506	8518	394M	151238	22M
counter	1051308	2603	403M	46892	22M
train	995554	2447	406M	44147	22M
playroom	2564740	6058	423M	113922	22M

(a) 3-dimensional scale parameters

		CUDA		GCE	
Dataset	# of params	Latency (µs)	Throughput (s^{-1})	Latency	Throughput
drjohnson	3358506	9336	359M	271398	12M
counter	1051308	2958	355M	84350	12M
train	995554	2777	358M	80925	12M
playroom	2564740	7727	331M	205883	12M

(b) 4-dimensional rotation parameters

		CUDA		GCE	
Dataset	# of params	Latency (µs)	Throughput (s^{-1})	Latency	Throughput
drjohnson	3358506	8928	376M	148955	23M
counter	1051308	2519	417M	46625	23M
train	995554	2364	421M	44207	23M
playroom	2564740	5976	429M	113758	23M

(c) 3-dimensional DC color parameters

Table 6.1: Encoding performance of the Gaussian Compression Engine (GCE) and CUDA implementations on the counter [20], train [21], and drjohnson and playroom [13] datasets. The GCE configuration tested has 8 dual encoders with an 8-port, 8-bank codebooks memory. The CUDA implementation is run on an NVIDIA RTX 4070 Ti SUPER. Performance is measured by encode batch latency in microseconds and by encode throughput in parameters encoded per second (where M denotes million). The CUDA implementation encodes about $20 - 30\times$ more parameters per second compared to the GCE.

GCE can support around 130 updates per second, falling short of real-time operation. However, recent 3DGS-based SLAM implementations [22] update only an active set of Gaussians during each of the mapping iterations, so real-time compression can be achieved with the GCE if only 10% of the Gaussians are updated and re-encoded each iteration.

6.5.3 Power

Power measurements can be obtained by measuring the FPGA board power during operation, but, if finer granularity is desired, they can also be determined through simulation of the module netlist deployed to the FPGA.

The power consumption of the engine, and that of its submodules, was estimated in Vivado using an out-of-context post-implementation netlist of the engine with switching activities

	Throughput (s^{-1})		
Parameter	Jetson (est.)	Jetson (meas.)	GCE
Scale (3-d)	13.18M	12.18M	22.08M
Rotation (4-d)	11.38M	10.41M	12.40M
DC color (3-d)	11.43M	12.17M	22.54M

Table 6.2: Throughput of the Gaussian Compression Engine (GCE) and CUDA implementations in encodes completed per second. The GCE configuration tested has 8 dual encoders with an 8-port, 8-bank codebooks memory. The estimated Jetson performance comes from scaling down performance of an RTX 4070 Ti SUPER to approximate that of an NVIDIA Jetson Orin Nano 8GB. The measured Jetson performance comes from limited encoding testing on a map with less Gaussians. The GCE achieves 83% higher throughput in the 3-d parameters and 19% more in the 4-d parameters, compared to CUDA on the Jetson.

derived from simulation in VCS using the netlist on the engine testbench. As shown in Table 6.3, most dynamic power is dissipated in the logic and the signals. There is also a large

Type	Power (mW)
Dynamic	1882
Clocks	239
Logic	505
Signals	836
RAMs	196
DSP48s	106
Static	2498
(adjusted)	55
Total	4380
(adjusted)	1937

Table 6.3: Estimated dynamic and static power consumption of the Gaussian compression engine in post-implementation simulation. Static power is adjusted to match the portion of the FPGA utilized by the design by subtracting consumption from unused regions of the CLB fabric.

static power component which dominates over the dynamic power. However, the FPGA on the VCU118 is much larger than required to fit the design, so static power from always-on components, like the configurable-logic-blocks, that are not used by the design contributes to the total power consumption. So, in addition to overall device static power, an adjusted power metric, which seeks to represent that corresponding only to the components in the design, is reported. The adjusted static power $P_{static,adj}$ is calculated as

$$P_{static,adj} = U \cdot P_{static,int} + P_{static,bram}$$

where U is the total LUT utilization from 0 to 1, $P_{static,int}$ is the static power from the VCCINT voltage rail that powers all of the CLBs in the FPGA, and $P_{static,bram}$ is the static

	Dynamic Power (mW)					
Module	Clocks	Logic	Signals	RAMs	DSP48s	Total
Engine	239	505	836	196	106	1882
Coupler	34	9	11	0	0	53
Splitter	6	1	9	0	0	17
Dual Encoders	160	432	694	133	106	1520
Codebooks	38	64	122	63	0	288

(a) 3-dimensional parameters

	Dynamic Power (mW)					
Module	Clocks	Logic	Signals	RAMs	DSP48s	Total
Engine	238	537	914	202	135	2024
Coupler	34	4	5	0	0	42
Splitter	6	<1	4	0	0	11
Dual Encoders	160	459	753	137	135	1644
Codebooks	38	72	152	65	0	327

(b) 4-dimensional parameters

Table 6.4: Estimated dynamic power consumption of the Gaussian compression engine in post-implementation simulation, by module. For modules that have multiple instances, total power consumption across all instances is reported. Power consumption in the dual encoders and codebooks increases when encoding 4-dimensional parameters.

power for the memory arrays, all as reported by Vivado. With static power adjusted, it becomes a much smaller portion of total design power consumption.

Unsurprisingly, the dynamic power consumed by the engine has a dependence on dimensionality as documented in Table 6.4. Dynamic power increases for the dual encoders and the codebooks memory when encoding 4-dimensional parameters, as the 4th lane of the memory and arithmetic pipelines becomes active. There is a slight drop in dynamic power for the other modules, as encoder throughput is lower when encoding 4-dimensional parameters because more centroids are visited per encode.

The 1.937W adjusted total power consumption of the engine when encoding 3-dimensional parameters is 2.68× less than, or 38% of, the 5.189W power consumption measured on the VDD_CPU_GPU_CV rail of the Jetson Orin Nano during the encoding test that yielded the performance results in Table 6.2.

6.5.4 Area

A singular area metric, which would represent the size of the design when fabricated on an ASIC, does not exist when deploying to FPGAs, as different FPGA primitives have different sizes. Instead, a general idea of area utilization can be drawn from the FPGA utilization report given from Vivado.

As a reference, the utilization of the compression engine is compared to the CVA6 CPU core

also present on the FPGA testing platform, as shown in Table 6.5. The DSP to LUT and

Module	LUTs	LUTRAMs	FFs	RAMs	DSP48s
Engine	66329	9874	82973	6624 Kb	256
Coupler	7055	306	13751	0	0
Splitter	1380	0	2701	0	0
Dual Encoders	47770	3552	61743	2016 Kb	256
Codebooks	9703	6016	4083	4608 Kb	0
CVA6	37771	570	15995	1296 Kb	27

Table 6.5: Utilization of FPGA resources of compression engine, with 8 banks and 8 dual encoders, implemented on VCU118 FPGA board with the CVA6 from the same design as a reference. Resources are divided into logic look-up-tables (LUTs), LUTRAMs, flip-flops (FFs), Block RAMs and UltraRAMs (RAMs), and DSP48 units (DSP48s). For the dual encoders, total utilization across all instances is reported.

FF ratio is higher on the dual encoders than the CVA6. This generally indicates a higher portion of area dedicated to compute compared to control logic, which allows for higher maximum throughput at lower area cost.

However, a low control to compute area ratio is not unique to the engine's encoder architecture. After all, as discussed in Section 3.1, this quality is what makes GPUs favorable over CPUs in highly parallel workloads. When both architectures have little control logic overhead, the arithmetic units themselves dominate the area cost of the design, with a "unit" being defined in this case as a single adder and multiplier. It should be noted that the arithmetic units in GPU are floating-point fused-multiply-add units, but fixed-point units are used in the encoder cores. For the sake of simplicity, this comparison will also exclude the integer arithmetic units on the GPU and the supplementary operations after the difference and multiply on the GCE. Nonetheless, with only 4 adders and multipliers per core in the distance calculation module, the GCE configuration tested has only 64 arithmetic units, compared to the 1024 on the Jetson Orin Nano. This means that the GCE outperforms the Jetson not only at 1/6 of the clock frequency, but while using 1/16 of the arithmetic units.

The last area consideration is memory usage, which can be gauged relative to the arithmetic hardware present. With a 512-centroid cache, traversal stack, and operands register file per encoder core, 252KiB of BRAM is used across all 16 cores. This means that there is about 4KiB of memory instantiated per arithmetic unit in the dual encoders. For perspective, an SM in either the Jetson or the 4070 Ti SUPER uses 128KiB in the unified data cache alone, with another 256KiB used for the register file [9]. With 128 arithmetic units per SM, this results in 3KiB of memory per arithmetic unit. The memory to arithmetic logic ratio in the encoder core is thus a little higher than in GPUs, but this number is partly inflated due to imperfect mappings between the memories in the RTL and the size increments, i.e. 18 and 36Kibit, of BRAMs on the FPGA. The total RAM usage of the codebooks memory is similarly inflated due to the coarse size increments of the URAM primitives.

In conclusion, the Gaussian compression engine is highly area efficient both because it has

# of units	Dual Encoder LUTs	Change	Codebooks LUTs	Change
4	25684		4265	
8	51322	↑ 2.00×	15719	↑ 3.69×
16	103118	↑ 2.01×	53897	↑ 3.43×

Table 6.6: Total LUT utilization of the dual encoders and the codebooks memory as a function of number of units. The number of "units" refers to the number of banks in the memory and the number of dual encoders, which are set to be equal. Codebooks LUT usage scales faster than dual encoder LUT usage.

a low control logic overhead compared to CPUs and because it achieves higher throughput with less arithmetic units than GPUs, even with an optimal CUDA implementation. Lastly, the memory usage per arithmetic unit is comparable to GPUs and has room to be reduced with better mapping to primitives.

6.5.5 Scaling

As noted in previous sections, the ability to use multiple encoder cores to proportionally multiply encoding throughput is a critical property of the design of the encoder core and the compression engine. However, the engine topology has limits to how far performance can be scaled by adding additional encoder units.

The first limitation is memory bandwidth when loading centroids. To support more dual encoder units, the codebooks memory needs to have more banks as well. In other words, if 5 dual encoders can reach full utilization with 4 banks in the codebooks memory, then 8 banks would be adequate for 10 dual encoders. However, with both more dual encoders and more banks, banking and arbitration logic begins to grow in size. Unfortunately, unlike the dual encoder where both throughput and resource usage are proportional to the number of units, because of the complexity of the arbitration logic is related to both the number of input and output ports, its scales quadratically, as seen in Table 6.6. Thus, area cost from the codebooks memory eventually becomes prohibitive to further scaling of encoder count for increased throughput. With the current topology, designs above 32 dual encoders would have more area allocated to the codebooks memory than to the encoders themselves, which would be an inefficient usage of resources.

A second limitation stems from the non-scalable modules in the engine. The relevant ones are the encoder coupler, encoder splitter, and AXI bridges. The encoder splitter can handle a maximum of one request in and one response out per cycle, and it is the last of the three to become a bottleneck. In terms of the AXI bridges, the subordinate interface only handles requests on a batch level, so it is never throughput limiting. For the AXI manager bridge, it uses more read bandwidth than write bandwidth per encode, so the rate at which parameters are read limits performance before the rate at which indices can be written does. The encoder coupler is somewhat beholden to the read bandwidth, as 3-dimensional and 4-dimensional parameters take on average 1.5 and 2 transfers, respectively, to be stitched together from the 64-bit data transfers from the bus. Even with maximum bus bandwidth, because only one

transfer can be processed per cycle, this in term places a hard throughput cap of 0.66 encodes and 0.5 encodes per cycle, for 3-dimensional and 4-dimensional parameters respectively.

Summary

In this chapter, we described the apparatus for verifying the RTL logic of the encoder core and compression engine. After verification, we evaluated the compression engine along the axes of performance, area, and power. We showed that a 8-dual-encoder configuration provided 64% higher encode throughput compared to the optimized CUDA baseline running on a Jetson Orin Nano while using 1/16th of the arithmetic units and consuming 38% of the power. We also discussed the limits to scaling performance by increasing the number of dual encoder units in the engine.

Chapter 7

Conclusion

In this chapter, we summarize the contributions made by this thesis and present future work that addresses remaining shortfalls.

7.1 Contributions

In this section, we recap the contributions from this thesis.

In this thesis, we presented several improvements to k-means vector quantization with the goal of making it feasible for real-time compression of a 3D Gaussian splatting map in SLAM on micro-robots. First, we showed that current implementations of k-means integrated into 3DGS can be sped up by directly using CUDA instead of through a library like PyTorch, similar to how the original 3DGS paper used CUDA kernels to implement the most time-critical portions of the algorithm. However, we also showed that algorithmic modifications can reduce the number of centroid loads and squared Euclidean distance calculations by $65 - 141\times$ while producing the same optimal encoding results.

Given that these modifications do not map well to CUDA architectures, we next implemented a hardware encoder core that implements the modified k-means quantization process through nearest-neighbor search of a codebook sorted into a k-d tree. The core includes several optimizations specific to the search algorithm, like multithreading instead of speculation, a full throughput distance calculation unit, and a novel cache design that best handles the access pattern of the search.

Lastly, to leverage parallelism, we grouped up to 32 encoder cores into a unified Gaussian compression engine that encodes batches of Gaussian parameters with a throughput 64% higher than that of a Jetson Orin Nano, all while using 1/16th the arithmetic units at 1/6th the clock speed and less than 1/2 the power. This engine is made accessible to software through a memory-mapped interface, and directly fetches parameters from memory without intervention from a CPU. The engine was integrated into an FPGA subsystem such that a workstation PC running live 3DGS training can transfer 3- and 4-dimensional floating-point parameters to the engine via PCIe and retrieve the encoded indices once the encoding batch

is complete.

7.2 Future Work

In this section, we reintroduce the limitations of this work and hypothesize future work to fix them. These limitations are listed below:

- **Codebook size limits**: the unpredictable access pattern of the tree search algorithm requires that the codebooks be stored in their entirety within the compression engine. This means that the maximum codebook size that the engine can act upon is limited by the size of this memory, which is limited by how much on-die area can be used for SRAM. Because the tree search algorithm scales well with increased codebook size from a compute complexity standpoint, this becomes a bottleneck to using larger codebooks sizes, e.g. 32768 centroids.

  To support larger codebooks, the codebooks memory in the engine would need to be converted from the endpoint memory to a last-level cache.

- **Naive encoder core scheduling**: currently, the encode splitter arbitrates requests through round-robin arbitration. However, this means that there is no correlation between encodes running in the same encoder core. This degrades cache performance, as a larger portion of the tree is accessed by the active encodes.

  Instead of round-robin arbitration, encodes could instead be partitioned among the encoder cores by their initial traversal path. For example, for a compression engine with 8 encoder cores, all of the encodes that traverse left-left-left can be handled by core 0, all that traverse left-left-right by core 1, left-right-left by core 2, and so forth. Because encodes rarely traverse the other side of their first couple branches, as shown by Fig. 3.3, encodes grouped this way will exhibit high levels of path similarity, which can be leveraged by the depth-strided cache already present in the encoder core.

- **Lossy acceleration**: this thesis demonstrates a hardware accelerator that provides identical encoding results to the software baseline and prior work at a lower area and power cost with comparable performance. However, the design can be tweaked in several ways to provide lossy encoding results with even less area and increased performance.

  First, the tree search algorithm implemented by the encoder is guaranteed to find the centroid with the minimum squared Euclidean distance from the parameter being quantized. In some cases, though, it may be acceptable for an almost-closest centroid to be found instead. This would allow the search algorithm to exclude more sections of the tree earlier in traversal, and thus load fewer centroids during the search.

  Second, the fixed-point arithmetic used by the encoder core is sized to be as precise as the single-precision floating-point arithmetic present in the CUDA quantization implementation. In addition, there is no intermediate rounding performed, so operand size grows in the distance calculation pipeline. If less precision is required by the

application, cores can be instantiated instead with smaller fixed-point representations and with intermediate rounding to limit operand size. This only requires minor tweaks to the current encoder core RTL, but more evaluation of resultant encoded indices is necessary.

A combination of these techniques can also be used to tailor quantization accuracy to the application, where compression quality can be traded for higher encode throughput or lower power consumption.

- **Bit packing**: the index packer in the compression engine presented in this thesis packs indices such that indices are always at least 2-byte aligned. This format is easier to decode for general-purpose hardware, which can read the indices as either an array of 16-bit, 32-bit, or 64-bit integers. However, this limits maximal compression gains, as, for example, a 12-bit index into a 4096-centroid codebook is stored as a 16-bit number.

References

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

[2] G. Chen and W. Wang, *A survey on 3d gaussian splatting*, 2024. arXiv: 2401.03890 [cs.CV]. URL: https://arxiv.org/abs/2401.03890.

[3] V. Ye, R. Li, J. Kerr, *et al.*, "Gsplat: An open-source library for Gaussian splatting," *arXiv preprint arXiv:2409.06765*, 2024. arXiv: 2409.06765 [cs.CV]. URL: https://arxiv.org/abs/2409.06765.

[4] J. Lee, S. Lee, J. Lee, J. Park, and J. Sim, "Gscore: Efficient radiance field rendering via architectural support for 3d gaussian splatting," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 497–511, ISBN: 9798400703867. DOI: 10.1145/3620666.3651385. URL: https://doi.org/10.1145/3620666.3651385.

[5] J. C. Lee, D. Rho, X. Sun, J. H. Ko, and E. Park, "Compact 3d gaussian representation for radiance field," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024, pp. 21 719–21 728.

[6] K. Navaneet, K. P. Meibodi, S. A. Koohpayegani, and H. Pirsiavash, "Compgs: Smaller and faster gaussian splatting with vector quantization," *ECCV*, 2024.

[7] S. Niedermayr, J. Stumpfegger, and R. Westermann, *Compressed 3d gaussian splatting for accelerated novel view synthesis*, 2023. arXiv: 2401.02436 [cs.CV].

[8] H. Matsuki, R. Murai, P. H. J. Kelly, and A. J. Davison, "Gaussian Splatting SLAM," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024.

[9] N. Corporation. "Cuda c++ programming guide." (2025), URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[10] X. Liu, X. Wu, P. Zhang, S. Wang, Z. Li, and S. Kwong, *Compgs: Efficient 3d scene representation via compressed gaussian splatting*, 2024. arXiv: 2404.09458 [cs.CV]. URL: https://arxiv.org/abs/2404.09458.

[11] J. Patas. ""3d gaussian splatting for real-time radiance field rendering" reproduction in c++ and cuda." (2025), URL: https://github.com/MrNeRF/gaussian-splatting-cuda/tree/master.

[12] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975, ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[13] P. Hedman, J. Philip, T. Price, J.-M. Frahm, G. Drettakis, and G. Brostow, "Deep blending for free-viewpoint image-based rendering," *ACM Trans. Graph.*, vol. 37, no. 6, Dec. 2018, ISSN: 0730-0301. DOI: 10.1145/3272127.3275084. URL: https://doi.org/10.1145/3272127.3275084.

[14] I. Synopsys. "Designware library - datapath and building block ip." (2025), URL: https://www.synopsys.com/dw/buildingblock.php.

[15] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, France: Association for Computing Machinery, 2010, pp. 60–71, ISBN: 9781450300537. DOI: 10.1145/1815961.1815971. URL: https://doi.org/10.1145/1815961.1815971.

[16] hlslibs. "Ac_types." (2025), URL: https://github.com/hlslibs/ac_types.

[17] A. S. Initiative. "Systemc.org." (2025), URL: https://systemc.org/.

[18] cocotb. "Cocotb | python verification framework." (2025), URL: https://www.cocotb.org/.

[19] openhwgroup. "Cva6 risc-v cpu." (2025), URL: https://github.com/openhwgroup/cva6.

[20] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman, *Mip-nerf 360: Unbounded anti-aliased neural radiance fields*, 2022. arXiv: 2111.12077 [cs.CV]. URL: https://arxiv.org/abs/2111.12077.

[21] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, "Tanks and temples: Benchmarking large-scale scene reconstruction," *ACM Trans. Graph.*, vol. 36, no. 4, Jul. 2017, ISSN: 0730-0301. DOI: 10.1145/3072959.3073599. URL: https://doi.org/10.1145/3072959.3073599.

[22] D. Gao, P. Z. X. Li, V. Sze, and S. Karaman, *Gevo: Memory-efficient monocular visual odometry using gaussians*, 2024. arXiv: 2409.09295 [cs.RO]. URL: https://arxiv.org/abs/2409.09295.