MASTER'S THESIS 2026

Iterative Tensorwise Quantization of Neural Networks Using MLIR

Albin Nyström, Aron Somi

ISSN 1650-2884
LU-CS-EX: 2026-33

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2026-33

Iterative Tensorwise Quantization of Neural Networks Using MLIR

Iterativ tensorbaserad kvantisering av neurala nätverk med MLIR

Albin Nyström, Aron Somi

Iterative Tensorwise Quantization of Neural Networks Using MLIR

Albin Nyström
albin.s.nystrom@gmail.com

Aron Somi
arsomi00@gmail.com

June 17, 2026

Master's thesis work carried out at Inceptron.

Supervisors: Noric Couderc (LTH), noric.couderc@cs.lth.se
Felix Malmsjö (Inceptron), felix.malmsjo@inceptron.io

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

As Large Language Models (LLMs) continue to grow in size, so do their memory and computational requirements. Quantization is a common method for reducing these requirements to improve efficiency and meet hardware specifications, though often at the cost of model quality. Existing quantization methods typically rely on advanced calibration algorithms and perform one-shot quantization. This thesis explores iterative tensor-wise mixed-precision quantization using Bayesian optimization for two different quantization methods. Our solution leverages a MLIR backend to decrease the cost of exploring a large set of quantization configurations. The results demonstrate both the feasibility and the challenges of applying Bayesian optimization to LLM quantization. In particular, the large search space makes it difficult to consistently identify optimal quantization configurations, indicating that additional methods for constraining or guiding the optimization process are needed.

Keywords: Post-training quantization, mixed-precision quantization, Large Language Models, Bayesian optimization, Llama, MLIR

Acknowledgements

We would like to thank Noric Couderc, our supervisor at LTH. His guidance and the weekly meetings have been of great help during this project. We would also like to thank our LTH examiner, Flavius Gruian, for the thorough feedback. Furthermore, we want to warmly thank our supervisor Felix Malmsjö for his advice when we needed itand the entire Inceptron team for the welcoming atmosphere, the lunch discussions and the opportunity to conduct our thesis at their company. Lastly, we would also like to thank Patrik Persson for his ideas on how to improve our work and Linus Carlsson for his help when technology worked against us.

Contents

Chapter 1
Introduction

Large Language Models (LLMs) are at the forefront of the current AI boom. They consist primarily of stacked transformer layers dominated by large matrix multiplications [33]. In practice, the most computationally expensive operations include large matrix multiplications, where weights are multiplied with an input to produce an intermediary step (activation), this is repeated until the final multiplication becomes interpretable [38]. Weights are usually represented as floating point numbers, and are precomputed when "training" the model. They make up for a majority of the models total parameters, and by extension its storage requirements.

LLMs continue to grow more advanced, this growth is not only reflected in performance but also in model size [20]. The current LLM boom started with GPT-3 in 2020, a model with 175 billion parameters requiring around 350 GB of storage [6]. Parameter count for newer models is a close kept secret but some estimates put the number of parameters in the trillions [10].

Numerical quantization is a common technique for reducing the memory requirements of large language models [8]. This technique is generally preferred to simply reducing the parameter count, as the latter tends to have a greater effect on model behaviour and requires advanced algorithms to be correctly performed [12]. At its core, numerical quantization is simply representing model parameters with data types that use less memory for a single number, thus saving space. A common example is going from 32 bit floating-point numbers to 8 bit integers, cutting the required storage for a single number in four. However, quantization comes at a cost of model performance, storing parameters at a lower precision negatively impacts model accuracy. Thus, quantization becomes an optimization problem where a model's size is weighed against its accuracy and where the optimal solution depends on the requirements (memory-accuracy trade-off).

The simplest way to quantize a LLM would be to change the format of all datatypes uniformly across the model. This has quite often been done for simpler image classification networks, but LLMs require a more careful approach [19][22]. Quantizing some parameters while keeping others in their original format is called mixed-precision quantization and is

commonly used for LLMs [8]. Activation-aware Weight Quantization (AWQ) and Generative Pre-trained Transformer Quantization (GPTQ) are two quantization methods in the spirit of mixed-precision quantization, but for performance reasons keep all weights in one data-type. AWQ looks at the activations of the LLMs and keeps track of which are greater in size. Larger activations lead to larger errors when the corresponding weights are quantized and the AWQ algorithm therefore aims to keep these larger activations intact while quantizing other parts of the model [23]. GPTQ instead looks at the output of the model and how changes to the weights affect this output, both single weights and groups of weights, through covariance. Based on these metrics GPTQ then quantizes the weights that minimize output variance [13].

Both AWQ and GPTQ profiles the network by running data through it, analysing it and performing the quantization once. However, these methods are primarily designed as one-shot quantization approaches and do not iteratively evaluate and refine their quantization.

We address this lack of iterative improvement by implementing a tensor-wise quantization framework with the help of transformation passes in MLIR, a compiler framework that is able to represent neural networks as a set of operations [21]. More specifically, we implement quantization as a MLIR-to-MLIR transformation pass on an intermediate representation of a neural network. Instead of lowering weights to a reduced precision data-type, we simulate quantization in MLIR. This allows quantization parameters to be changed at runtime without recompiling the model for each quantization configuration. By representing quantization as compiler transformations, we can iteratively evaluate different mixed-precision configurations while reusing the same compiled MLIR program.

Using this framework, we investigate whether black-box optimization can be used to guide tensor-wise mixed-precision quantization of large language models. More specifically, we explore if Bayesian optimization can identify tensors that tolerate lower precision while maintaining model quality. We also compare affine and non-uniform quantization schemes, and evaluate how MLIR can be used to efficiently test a large number of quantization configurations.

This work was conducted in collaboration with Inceptron AB, a company developing MLIR-based compiler technology for efficient large language model inference. The company's existing MLIR infrastructure motivated the choice of implementation framework.

1.1 Research Questions

To evaluate the effectiveness of our approach, we formulate the following research questions:

1. **RQ1**: How does a black-box optimization guided mixed-precision quantization method compare to fixed precision quantization with respect to memory-accuracy trade-off?

2. **RQ2**: To what extent can Bayesian optimization be used to select tensor-level quantization policies that improve memory-accuracy trade-off?

3. **RQ3**: How do different group-wise quantization schemes affect the memory–accuracy trade-off in iterative mixed-precision quantization?

4. **RQ4**: To what extent can MLIR be used to evaluate different quantization configurations for large language models?

1.2 Division of Labor

Both authors contributed equally with initial research. For the implementation, Aron focused more on prototyping and experimentation while Albin compiled the solution and ironed out the kinks. For the paper, most content has been a shared effort but Aron has had more focus on benchmarking and results whereas Albin has focused more on the method and solution overview.

1.3 Use of Generative AI

We used ChatGPT as a tool when writing this report. ChatGPT was occasionally used to assist with word choices, grammar and general structure of the text. Additionally, we used Codex (the coding variant of ChatGPT) as a tool when coding. We have manually reviewed all AI generated code and take fully responsibility for its functionality.

1.4 Report Disposition

Chapter 1 – Introduction	This chapter introduces the thesis, presents the problem formulation, and states the research questions.
Chapter 2 – Background	This chapter provides background covering large language model inference, quantization techniques, and optimization methods, as well as an introduction to MLIR.
Chapter 3 – Solution	This chapter covers our method, presents our solution and the optimization framework.
Chapter 4 – Evaluation	This chapter presents our results, describes the experimental setup and evaluates our method against standard benchmarks.
Chapter 5 – Discussion	This chapter discusses the results with respect to the memory-accuracy trade-off, the effectiveness of our Bayesian optimization and the usage of MLIR. It also covers threats to validity and future work.
Chapter 6 – Related Work	This chapter discusses related work in more detail and positions this work within the existing literature.
Chapter 7 – Conclusion	This chapter summarizes the reports findings and revisiting the research questions.

Chapter 2
Background

In this chapter we provide the theoretical and technical knowledge required to understand our solution. It covers LLM inference, quantization techniques, black-box optimization and the MLIR framework used for implementation and evaluation.

2.1 Large Language Models

The structure of current LLMs is based on the transformer architecture from the 2017 paper *Attention Is All You Need* [33]. Transformer models are made up of repeating transformer blocks containing matrix multiplications, normalization operations, and attention mechanisms. Attention is the LLMs' ability to look at the correlation between words in a sentence and determine which ones are connected to each other. Attention blocks require storing large amounts of model weights. These weights contain the majority of the models parameters and constitute the majority of the models footprint. In this work we aim to reduce the memory footprint by altering these weights.

2.1.1 Llama Architecture

The experiments in this thesis use an open source model in the Llama family developed by Meta AI. It is like most LLMs a decoder only model, meaning it's a model that predicts and constructs sentences based on the previous part of said sentence. In contrast with many other LLMs, Llama's weights are open source - meaning the model is open to the public to download and run. The Llama model family ranges from small models which run on consumer grade hardware, to large state-of-the art models.

In this work we use the Llama-3.1-8B model as it is relatively small and can be run on a single GPU, while still being large enough to quantize well. Furthermore Llama-3.1-8B is a model commonly used for benchmarking.

2.1.2 Memory Bandwidth Bottleneck

LLM inference is typically memory-bandwidth bound rather than compute-bound. During inference, model parameters must be repeatedly loaded from memory for each generated token, resulting in large data movement relative to the amount of computation performed. As modern GPUs provide higher compute throughput than memory-bandwidth, the execution is often limited by the rate at which data can be transferred from memory to the compute units. Meaning that GPU cores wait for memory accesses which makes memory bandwidth the primary performance bottleneck in LLM inference [29]. In our work, this observation motivates the use of quantization to reduce model size and thereby reduce memory bandwidth requirements during inference.

2.1.3 Model Output

Inside a LLM words are represented by tokens. When generating text a LLM does so one token at a time by taking an input sequence and producing a tensor of logits with the shape.

`[batch_size, sequence_length, vocab_size]`

This tensor contains the logits for each batch, for each position in the sequence and for each token in the vocabulary. We use these values to compute the negative log-likelihood (NLL) and perplexity, see Section 2.1.4.

2.1.4 Perplexity

A common metric for evaluating language model performance is *perplexity*. Perplexity is computed from the logits produced by the model's final layer which represent the model's predictions for the next token at each position in the input sequence. These logits are converted into probabilities using the softmax function, defined in Equation (2.1), which maps the logits to a probability distribution over the vocabulary. We use perplexity as the performance metric for out optimization loop described in Section 3.4.

Intuitively, perplexity can be understood as how uncertain the model is when predicting the next token. A lower perplexity indicates a less uncertain model, which is achieved when the model assigns a high probability to the correct token.

Given the probability assigned to the correct next token, the NLL is computed as shown in Equation 2.2. The average NLL over all positions in the sequence is given by Equation 2.3, and the perplexity is defined as the exponential of this average, as shown in Equation 2.4.

$$p(y_t \mid y_{<t}) = \frac{e^{z_{t,y_t}}}{\sum_{j=1}^{V} e^{z_{t,j}}} \tag{2.1}$$

$$\mathrm{NLL}_t = -\log p(y_t \mid y_{<t}) \tag{2.2}$$

$$\mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} \mathrm{NLL}_t \tag{2.3}$$

$$\mathrm{PPL} = \exp(\mathcal{L}) \tag{2.4}$$

In the equations above, $z_{t,j}$ denotes the logit assigned by the model to token j at position t, and z_{t,y_t} is the logit corresponding to the correct target token y_t. The term $p(y_t \mid y_{<t})$ is the probability assigned to the correct next token given the preceding tokens $y_{<t}$. The variable V denotes the vocabulary size, and T is the number of predicted positions in the sequence. The quantity $\mathbf{NLL}_t$ is the negative log-likelihood at position t, $\mathcal{L}$ is the average negative log-likelihood over the sequence, and **PPL** is the resulting perplexity.

2.1.5 Attention Masks

The input sequences for the LLMs used in this work are fixed in length. Shorter inputs are padded with padding tokens. These padding tokens should not contribute the loss or NLL calculations. In our solution we use attention masks for determining which tokens are valid.

2.1.6 Large Language Model Evaluation

LLM evaluation benchmarks can be divided into three categories of which the first two are used in this work. The first category consists of language modelling evaluations based on log-likelihood or perplexity (for the correlation between these two metrics see Section 2.1.4). Log-likelihood benchmarks mainly measure how well a model predicts the next token in a sequence.

The second category consists of benchmarks using multiple-choice questions. These benchmarks measure reasoning, knowledge retrieval, and decision-making. Each example in the benchmark consists of a question and a set of alternatives. For example, the question "What is the capital of France?" may be associated with the alternatives "Paris, Madrid, Rome, Berlin". For each alternative, the answer is appended to the prompt, creating a sequences such as "What is the capital of France? Answer: Paris". Each sequence is passed through the model, and the log-likelihood of the answers is computed. The alternative with the highest log-likelihood is selected as the model's prediction. Both language modelling and multiple-choice evaluations rely entirely on log-likelihood and do not require token generation.

The third category consists of generation-based evaluations where the model produces free-form responses. These tasks measure the model's ability to generate coherent and correct outputs and use more complex assessments of correctness then the first two methods. We decided to not include this category as it has substantially higher evaluation costs and variability.

We limit our evaluation to benchmarks of the first two categories. In the first category we use **WikiText** [25], a large dataset of around 100 million tokens from Wikipedia articles, which captures modelling quality.

The second category of benchmarks are useful for capturing model reasoning and knowledge retrieval degradation caused by quantization. **MMLU** [17] evaluates both reasoning and knowledge retrieval on a large set of academic domains. **ARC-challenge** [7] is useful for evaluating multiple-step reasoning while **HellaSwag** [37] evaluates sentence completion, and **PIQA** [5] evaluates physical reasoning. We chose these benchmarks as they are common in quantization literature and cover a wide range of tasks [13, 18, 23].

Additional benchmarks could provide a further spanning evaluation. However, adding more benchmarks increases evaluation time. These benchmarks take around 2 hours to

complete on our hardware. The selected benchmarks are a compromise between evaluation breadth and computational time.

2.2 Quantization

Quantization is the process of mapping numbers from a larger set to a smaller set. In practice, this often means mapping a high-precision value to a low-precision representation. Quantization is a non-linear, irreversible process, and the exact input value cannot in general be recovered from the quantized value. The primary benefit of quantization is that it allows models to run on otherwise insufficient hardware.

In machine learning (ML), quantization of model weights and activations is common practice. The goal of quantization in ML is to reduce a model's memory footprint while maintaining model performance. Model weights are typically stored in 32-bit (FP32) or 16-bit (FP16 or BF16) floating point representation. Quantization maps these representations to lower precision formats, like 8-bit integers (INT8). This reduces model size but also restricts weights to a less precise data-type. Models are commonly deployed at 8-bit precision [8, 19], while lower precision in the 2-4 bit range is actively being explored [23].

Depending on hardware and implementation, quantization can improve inference speed. Quantization improves inference speed by reducing memory bandwidth demands, due to lower-precision weights requiring fewer bytes to be moved between devices [19]. The bandwidth-constrained nature of LLM inference causes the reduced data movement to directly improve throughput [29].

If the target hardware does not natively support the quantized format, weights may be stored in this format and dequantized at runtime. Here, dequantization happens before the model's mathematical operations and the overhead associated with the conversion incurs a small performance cost.

Quantization can be applied during training (quantization-aware training) or after (post-training quantization). In this work only post-training quantization is considered, as quantization-aware training requires retraining models, which is beyond the scope of this work.

2.2.1 Affine Quantization

Affine quantization is a common quantization technique that uses a linear transformation defined by a scale and a zero-point, this is one of the two quantization schemes implemented in this work. This approach allows floating-point values to be represented as integers while approximately preserving the numerical range. The spacing between quantization levels is uniform and determined by the scale parameter, while the zero-point parameter aligns the quantized representation with the original value range. Due to its simplicity and efficient hardware support, affine quantization is widely used in machine learning systems [19].

Affine Quantization Maths

To be able to perform affine-quantization you first have to find the scale S and zero-point Z. The equations below define the quantization transformations our implementation performs at runtime for affine quantization.

Given a starting interval

$$I = [\alpha, \beta]$$

and a target quantized interval

$$I_Q = [q_{\min}, q_{\max}]$$

the scale and zero-point are defined as

$$S = \frac{\beta - \alpha}{q_{\max} - q_{\min}}$$

$$Z = \text{round}\left(\frac{-\alpha}{S}\right) + q_{\min}$$

The quantized value of a weight w_q is calculated as

$$w_q = \text{round}\left(\frac{w}{S}\right) + Z$$

which maps the floating-point value w_q into the quantized representation w_q. To reconstruct and approximation of the original value $\tilde{w}$, dequantization is performed as

$$\tilde{w} = S * (w_q - Z).$$

Figure 2.1 shows the quantization transformation, while Figure 2.2 shows the corresponding dequantization process.

Because quantization is not a lossless process, the maximum error of the quantization can be calculated by using the maximum rounding error of the rounding function used when quantizing.

$$E_{\text{Qmax}} = \frac{max\ rounding\ error\ * S}{2}$$

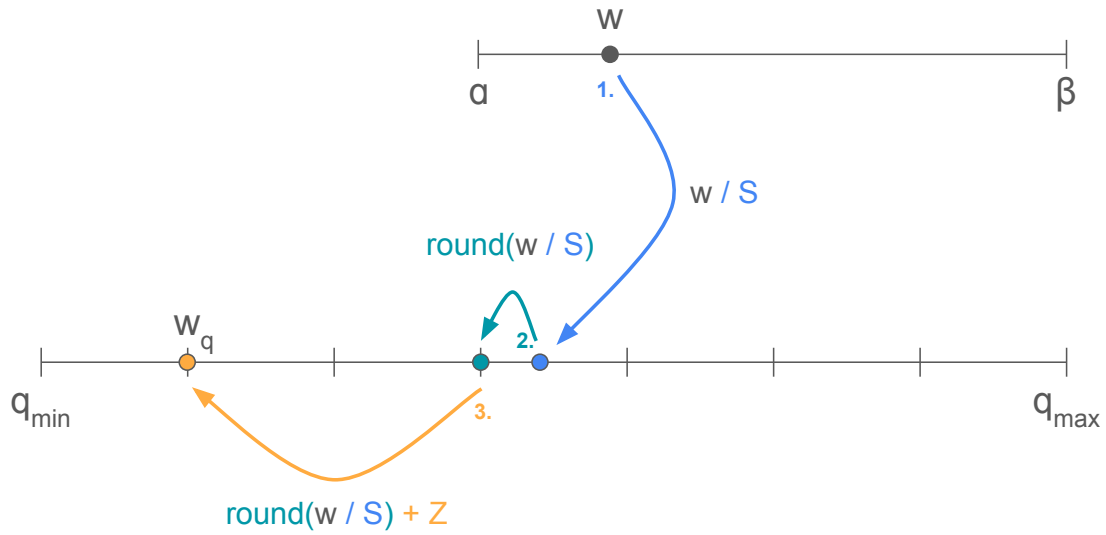

Figure 2.1: Step by step quantization from $[\alpha, \beta]$ to $[q_{\text{min}}, q_{\text{max}}]$

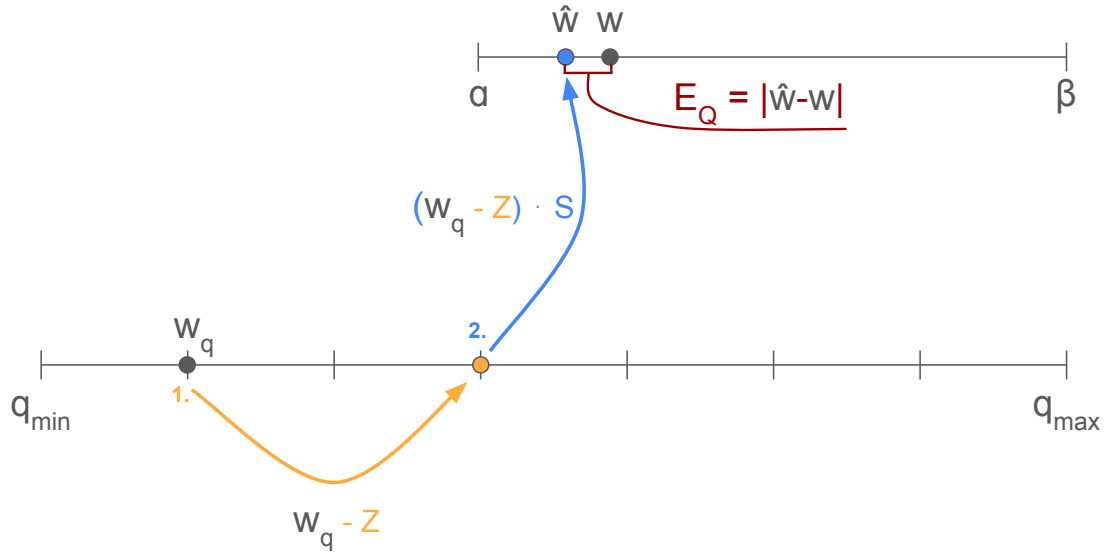

Figure 2.2: Step by step dequantization from $[q_{\text{min}}, q_{\text{max}}]$ to $[\alpha, \beta]$

2.2.2 Non-Uniform Quantization

Non-uniform quantization is another quantization technique implemented in this work. It is a technique that spaces quantization levels in a non-uniform way according to a chosen distribution. Regions with higher density in the distribution are allocated more levels, allowing for these regions to be represented more accurately. The difference in spacing between the levels can be seen in Figure 2.3. Provided the underlying distribution reflects which numerical regions are important, total quantization error is reduced compared to uniform quantization.

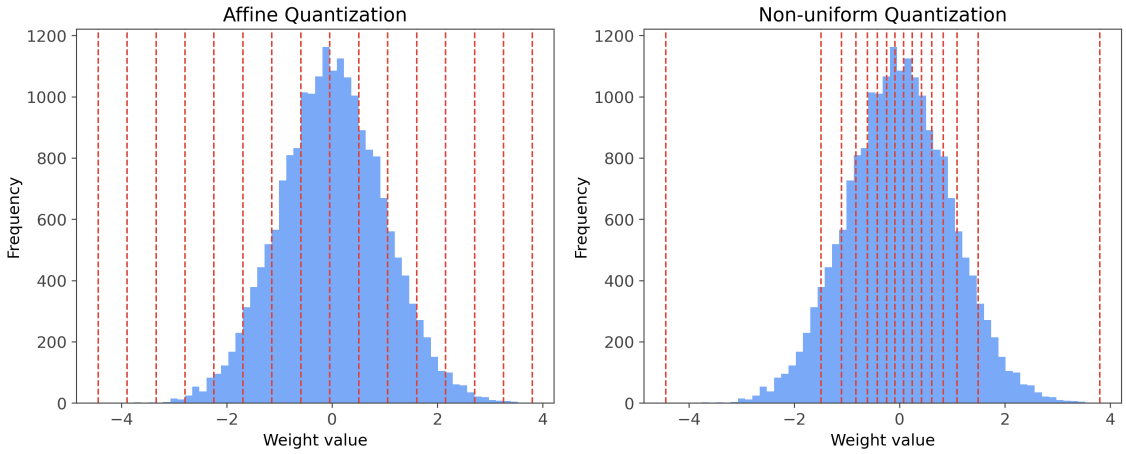

Figure 2.3: Comparison between uniform and non-uniform quantization

In this work we implement non-uniform quantization using codebooks. A codebook C consists of a discrete set of floating-point values

$$C = \{c_0, c_1, \ldots, c_{K-1}\},$$

where $K = 2^{b_q}$ for a b_q-bit quantization scheme. Each weight $w_i \in W$ is quantized by replacing it with the closest codebook value according to

$$\hat{w}_i = \arg\min_{c_k \in C} |w_i - c_k|.$$

One way to create a codebook is to use an importance score for each weight. Weights and importance scores can be used together to define a cumulative distribution function (CDF). The quantization levels are then obtained by taking the inverse CDF at regular intervals. This results in quantization levels concentrating in regions with higher cumulative importance. Figure 2.4 illustrates an example of an importance-weighted CDF and the resulting quantization levels.

Codebooks are associated with a higher overhead than affine quantization, as explained in Section 2.2.4. In this work we evaluate codebook quantization in comparison to affine quantization in order to investigate if the increased overhead of codebook quantization is justified by its performance.

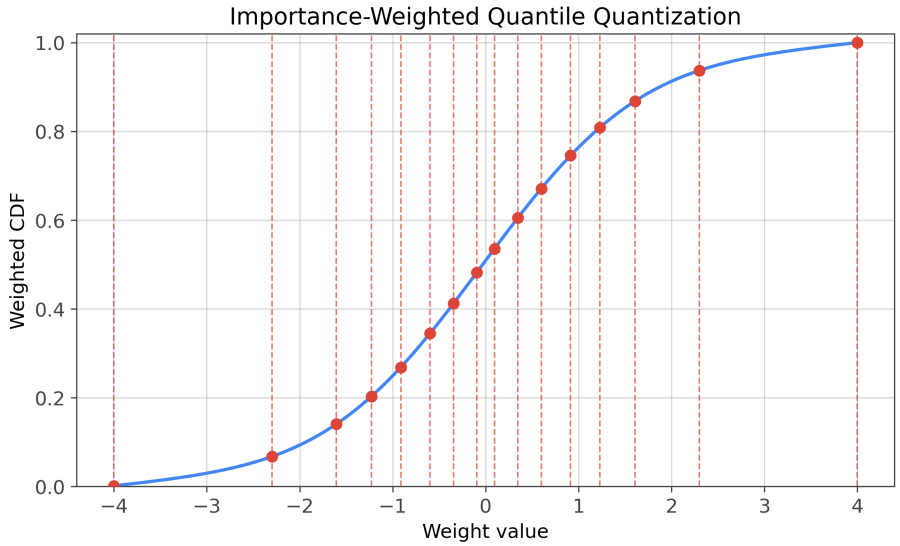

Figure 2.4: Example of importance-weighted quantile quantization using 128 weight values and a 4-bit codebook ($K = 16$ bins).

2.2.3 Tensor Quantization

When quantizing a tensor the quantization can be more or less granular. As demonstrated in Section 2.2, you start of with a set of weights W within an interval and map these weights onto a new discrete quantized interval. The endpoints of the interval are decided by $min(W)$ and $max(W)$ respectively, thus more weights increase the risks of having outliers that makes the interval unnecessarily large. To combat this, it is common to not quantize per tensor, but rather per channel or per group (see figure 2.5).

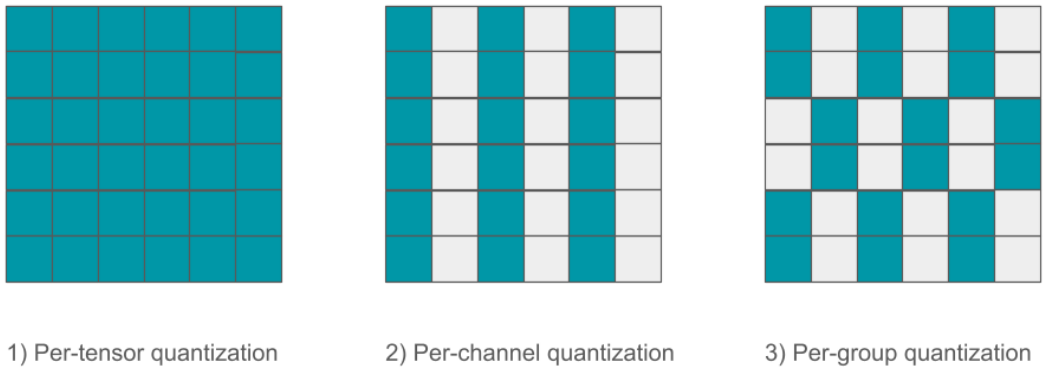

Figure 2.5: Representation of different quantization granularities

When splitting a tensor you get one quantization interval per part. This makes the quantization more resistant to outliers, as they influence a lesser number of weights. However, it creates more memory overhead as the new intervals have to be stored. In Figure 2.6 this observation is shown, each channel and group requires the same amount of metadata overhead as a whole tensor. Additionally, weights within the same channel or row contribute to the same computation and therefore tend to have more similar distributions than other channels of the same tensor [15]. This further motivates group-wise or per-channel quantization, as it allows each group to be quantized according to its own distribution, reducing quantization error. In our work we apply quantization group-wise in order to reduce the effects of outliers.

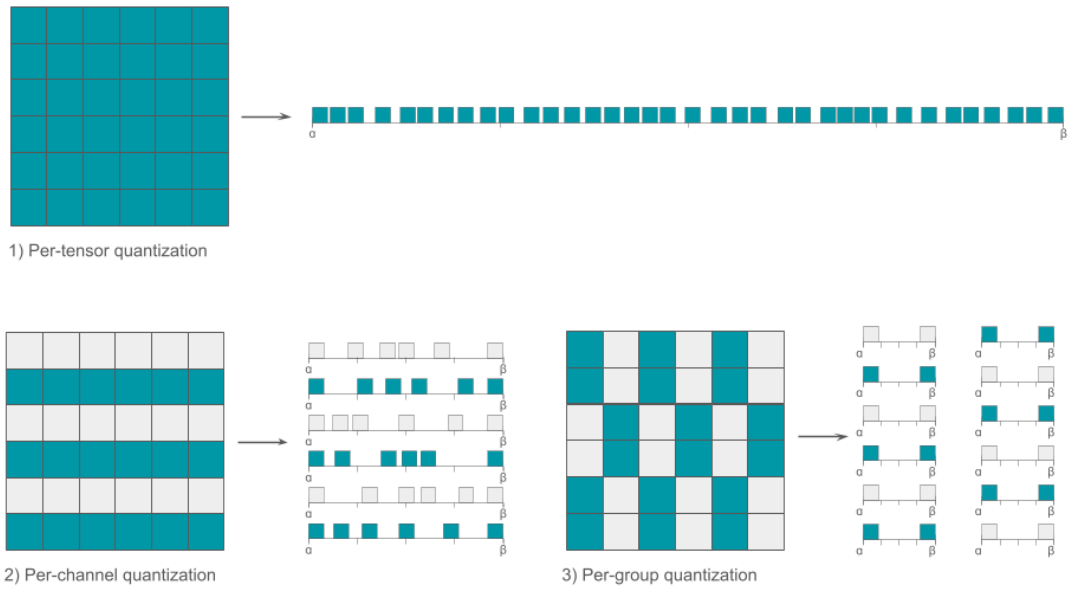

Figure 2.6: Representation of how different quantization granularities influence the number of intervals

2.2.4 Model Quantization

Model compression depends on both the original and the quantized data type. In general, reducing the bit-width of stored weights leads to a proportional reduction in memory usage. However, all practical quantization methods introduce some additional storage overhead.

In affine quantization, this overhead comes from the scale and zero-point parameters. When group-wise quantization is used, each group requires its own scale and zero-point, and the total overhead increases as the group size decreases. The total storage cost for affine quantization is expressed in Equation 2.5.

$$S_{\text{affine}} = \underbrace{N * b_q}_{\text{quantized weights}} + \underbrace{\left\lceil \frac{N}{G} \right\rceil 2 * b_{\text{orig}}}_{\text{quantization overhead}} \tag{2.5}$$

where N is the number of weights, b_q the size in bits of the quantized data type, G is the group size and b_{orig} is the size in bits of the original data type. Scale and zero-point values are both represented using the original precision data type b_{orig}.

In codebook quantization weights are stored as indices into the codebook. Each group is associated with a codebook containing 2^{b_q} entries. The storage cost for codebook quantization is expressed in Equation 2.6.

$$S_{\text{quant}} = \underbrace{N * b_q}_{\text{quantized weights}} + \underbrace{\left\lceil \frac{N}{G} \right\rceil 2^{b_q} * b_{\text{orig}}}_{\text{quantization overhead}} \tag{2.6}$$

The storage overhead associated with codebook-based methods is significantly larger than that of affine quantization due to the additional requirements for storing a codebook for each group. However, as codebooks only require memory look-ups, the computational cost of codebook quantization is smaller than that of affine quantization. In this work we investigate whether this additional metadata overhead associated with codebooks can increase the memory-accuracy trade-off compared to affine quantization.

2.3 Memory-Accuracy Trade-Off

Post-training quantization (PTQ) reduces bit-widths with generally negligible performance penalties down to 8 bits [18]. At 4 bits model quality degrades quickly without additional quantization-aware techniques, at 3 bits even these start showing performance issues [23, 13]. For even fewer bits, specialized ultra-low bit quantization is required [24]. For the Llama models affine quantization starts showing noticeable performance degradation at 4 bits and heavy degradation bellow 4 bits, while methods like AWQ and GPTQ lose less performance at these levels [18]. These methods exploit the fact that not all weights are of equal importance.

To protect or more accurately represent salient weights is a central design principles in modern PTQ methods, like AWQ and GPTQ. For example protecting around 0.1-1% of weights, AWQ manages to quantize at 3 bits with low performance losses [18]. Research also shows that quantization sensitivity is unevenly distributed across the model [24].

In this work, we build on these observations by exploring both codebook quantization and mixed-precision bit-width assignment. We investigate whether codebook quantization can better preserve salient weights and whether Bayesian optimization can identify tensors less sensitive to quantization which are suitable for lower precision.

2.4 Quantization Parameter Search

In affine and codebook quantization each tensor, layer or group has quantization parameters that can be varied to achieve different quantization results. In this work, we formulate the per-tensor assignment of bit-widths as a black-box optimization problem, where the objective is to find a configuration that optimizes the memory-accuracy trade-off.

The search space consists of discrete bit-width choices for each quantizable tensor which makes it a non-differentiable and high-dimensional optimization problem. To find a solution

to this optimization problem we use Bayesian optimization (BO), implemented using the Hyperopt library [3].

2.4.1 Black-Box Optimization

Black box optimization is a method which attempts to find a solution which minimizes an objective function without analysing its derivative. The method is used when it is impossible or impractical to find the derivative of the function that you want to optimize.

The basic problem formulation is: given an objective function $f : A \to \mathbb{R}$, find $x_0 \in A$ such that

$$f(x_0) \leq f(x) \quad \text{for all } x \in A.$$

This is the same as for general numerical optimization. The difference for black-box optimization is that finding the optimal x_0 is not guaranteed without exploring the entire search space.

2.4.2 Bayesian Optimization

Bayesian optimization is a sequential black-box optimization method. The BO models the objective function as a probabilistic process. To optimize the function the BO makes use of prior and posterior probability. The prior is a collection of assumptions on how the function reacts to different inputs. The posterior for a given iteration is the prior for the next iteration. Each iteration, an input is created from the prior, the function is evaluated and the prior is updated with the results.

2.4.3 Choice of Optimization Method

In this paper we use Bayesian optimization to search for the optimal quantization parameters for our LLM. Alternative approaches include grid search, random search, gradient-based optimization, evolutionary algorithms and reinforcement learning.

Our quantization parameter search problem has: a high-dimensional search space, discrete optimization variables, a non-differentiable objective function and an expensive evaluation function requiring LLM perplexity calculations.

Brute force or exhaustive search methods like grid search are infeasible due to the magnitude of configurations. Random search is a better choice and outperforms grid search in high-dimensional problems [2]. However, random search does not exploit information from previous evaluations, making it sample inefficient.

Gradient based optimization methods are not usable as the optimization variables are discrete and the quantization introduces rounding and clipping [16].

Evolutionary algorithms and reinforcement learning have also been used for architecture search, which is a similar discrete optimization problem [28, 39]. However, these methods typically require a large number of evaluations to learn policies which is detrimental when the evaluation function is expensive.

Bayesian optimization has successfully been used for hyper-parameter tuning which is a similar problem to ours. It is a method that is particularly well-suited to optimize expensive cost functions [30]. Evaluating our objective function requires us running LLM inference and

computing perplexity over a set of testing sentences making the evaluating expensive. For this reason we use Bayesian optimization, although evolutionary and reinforcement learning algorithms might be an alternative.

2.4.4 Hyperopt

We implement the BO using the Hyperopt library. Hyperopt is a Python library that provides a simple API for minimizing a loss function using Bayesian optimization [3]. Traditionally, a Gaussian process is used as the surrogate model for the Bayesian optimization process. However, this method struggles in high-dimensional settings due to the curse of dimensionality [4]. Hyperopt's creation was an attempt to solve this problem by using an algorithm called TPE (Tree-structured Parzen Estimator) [1]. While the traditional Gaussian approach essentially attempts to model the entire search space, TPE instead looks at the difference between good and bad results and tries to figure out what makes the "good" runs "good". This approach is much faster for a problem with a large amount of dimensions, but is more prone to getting stuck in local minima.

When we search for quantization parameters our BO's number of dimensions corresponds to the number of tensors, layers or groups depending on granularity. For Llama 3.1-8B's we quantize 224 tensors which gives our BO a relatively high number of dimensions, making an algorithm like TPE a more suitable choice than Gaussian process based BO.

2.5 MLIR (Multi Level Intermediate Representation)

Evaluating different quantization configurations for LLMs requires a large number of combinations of model parameters. Usually, this process requires compiling or reloading a model for each combination of model parameters. After a recompilation or reload the model is only used for evaluating a specific combination, and then discarded. Instead of recompiling or reloading the model at each iteration, injecting quantization parameters at runtime can lessen the effects of this bottleneck.

MLIR is one way to address this problem. It is a reusable and extensible compiler framework designed to support multiple abstraction levels within a single infrastructure. Using MLIR we can apply transformations to the computational graph, which enables us to:

- Modify the computational graph to accept quantization parameters at runtime
- Insert quantization and dequantization operations directly in the intermediate representation
- Reuse the same compiled program across many configurations

These improvements mean that we can evaluate different quantization configurations without recompiling the model, which can decrease the cost of evaluation.

2.5.1 Inceptron Backend

Inceptron is developing a MLIR-based compiler backend that efficiently executes LLMs by an optimized MLIR lowering pipeline. Leveraging this backend, the compiled MLIR representation of the LLM can be used for efficient execution. Since each BO iteration requires running LLM inference and computing perplexity, the cost of inference is a large factor in iteration time.

By using Inceptron's backend we reduce the cost of individual evaluations in the BO loop, increasing the number of configurations we can explore in a limited amount of time.

Chapter 3
Solution

This chapter describes our process developing the solution and the design choices and trade-offs that were made. This description is followed by a detailed explanation of our final solution.

3.1 Method

In this section we give a brief summary of our method and thought process during this project. This section includes work that was part of the process in which our solution was obtained, but not a part of the solution itself. For a more detailed description of our final solution see Section 3.2

3.1.1 Tooling Choices

The given problem specification from Inceptron required that MLIR was to be used as back-end for our solution. It is relatively resource intensive and complicated to implement things in MLIR, therefore we decided on using a more approachable (albeit performance-wise slower) framework for our prototyping, namely PyTorch [27].

Inceptron recommended Hyperopt [3] as a Bayesian optimization framework, as they had previous experience working with it. After verifying that it fit our specifications we decided to follow this advice, but to keep our solution adaptable and open for alternatives.

We chose Python as our main programming languages as this is the main language of both PyTorch and Inceptron's MLIR compiler. The MLIR passes are written in C++.

3.1.2 Quantization Demo

As a first step we constructed as small ML model in PyTorch to explore layer-wise quantization. Using PyTorch, we trained the model for image classification on the image dataset CIFAR-10. Once trained we set up a Bayesian optimization loop with Hyperopt that performed layer-wise bit-width quantization, using TorchAO for the quantization of the PyTorch model [26]. We experimented with the number, size and original data-type of the ML model's layers and with the search space for the Bayesian optimization. We also compared our quantization approach with using the same bit-width across the entire network. The test showed promising results and we achieved a better model memory-accuracy trade-off using our method than the uniform quantization across the entire network.

3.1.3 MLIR Implementation

Next, we created a working implementation of our quantization algorithm using MLIR. MLIR has no easily accessible quantization API such as PyTorch's TorchAO. We also discovered that there was no native support for variable bit-width integers or floats, especially not if you want a performant implementation. We therefore decided to instead simulate the quantization and leave the implementation of actual quantization to future work (more about simulated quantization in Section 3.5).

Running simulated quantization in MLIR still required modification of the MLIR pipeline. We therefore wrote transformation passes which mark relevant weights for simulated quantization (Section 3.3.3) and insertion of operations into the MLIR graph which perform quantization simulation (Section 3.3.4).

On the Python side, we set up a framework to generate the required inputs for the simulated quantization, the search space for the Bayesian optimization loop and logging of the BO loop's trials and results.

We first implemented support for our small CIFAR-10 PyTorch model. Then, we implemented support for a small LLM called OPT-125M, the choice of which was motivated by Inceptron's use of the model as a validation model.

3.1.4 Llama Implementation

Support for Llama was first implemented for Llama 3.2-1B and then Llama 3.1-8B. We started out with the smaller model as compilation times for the larger model were quite high, making debugging very time-consuming.

The MLIR part of our solution stayed widely unchanged but running the Llama models, especially the 8 billon parameter model, required optimizing the memory utilization of our Python script. It also required optimization of the input generation, which amongst other adaptions included caching of possible model inputs.

3.1.5 Search Space Limitations

As our solution evolved it became clear that we had to limit our Bayesian optimization search space. Initially we had a choice of bit-width per group and per tensor, creating a search space

which was unfeasible to sufficiently explore. Our initial approach to limiting the search space was to increase the size of the quantization groups, which in turn reduced the total amount of groups and choices the BO had to make. However, the increase in group size combined with affine quantization (Section 2.2.1) led to poor accuracy of the model. As an alternative solution, we implemented a codebook quantization scheme (Section 2.2.2) which, in theory, could produce a lower quantization error at the cost of increased memory overhead.

3.1.6 Validation and Evaluation

To complete our implementation we made sure to validate our work using PyTorch and TorchAO as a back-end instead of MLIR. This was done to show that the results of our Bayesian optimization loop are generalizable and to validate the MLIR implementation (Section 3.5.1).

We also set up support for evaluation of our models using Language Model Evaluation Harness [14].

3.1.7 Performance Issues

Using the existing lowering pipeline from MLIR to CUDA proved more difficult than initially assumed. We have not managed to achieve a performance in MLIR which is better than our validation set-up in PyTorch. We predict that this is,in part, because of difficulties in running some CUDA specific optimization passes.

The results in Section 4.2 are therefore based on runs using PyTorch.

3.2 Solution Overview

Our system consists of two main parts, *compilation* and *optimization*, of which an overview can be viewed in Figure 3.1. During compilation, our pipeline takes a PyTorch model and converts it to MLIR. After that, it inserts quantization logic into the MLIR representation of the model. The MLIR is then rewritten so that its input signature includes quantization parameters and runtime quantization functionality into the model. This design allows us to evaluate the effect of multiple quantization policies with a single compiled MLIR artifact, instead of recompiling for every quantization configuration.

The optimization uses back-box optimization to find quantization parameters that optimizes the memory-accuracy trade-off. We first define a search space consisting of bit-width choices for the model's weights. These choices are at tensor level, meaning that all weights in a tensor will use the same bit-width. Then, for each trial, the optimization makes choices for each tensor and evaluates the MLIR program with these quantization choices. The optimization uses a combination of perplexity and calculated quantized size as an evaluation metric. This metric is scored according to the loss function in Section 3.4.2. Using this metric better and better choices can be made by the black-box optimization, all without recompiling the model.

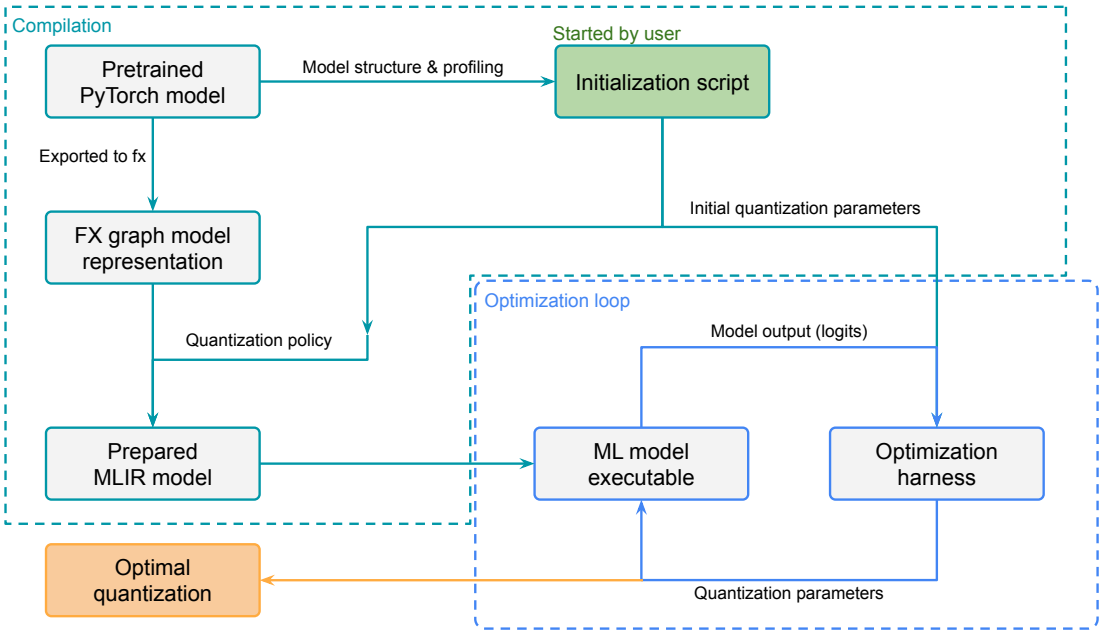

Figure 3.1: Solution overview. Arrows represent flow of data between different steps of the process. The dashed boxes represent Section 3.3 Compilation and Section 3.4 Optimization.

3.3 Compilation

The compilation phase starts with a pre-trained model defined in PyTorch. A `torch.fx` graph representation is generated from this model, which in turn is converted into a custom MLIR representation.

3.3.1 PyTorch Model

A model defined in PyTorch is used as the underlying model for the compilation pipeline. Using Huggingface's transformers API we download both model architectures, weights and the corresponding tokenizer required for creating correctly formatted inputs to the model. Using the transformers API a user can input a formatted message to a model and extract the returned logits from a `ModelOutput` object, decoding it to characters using the tokenizer.

3.3.2 Fx-Graph and MLIR module

We export the model to an FX graph representation, which is then translated into an MLIR module. First, the FX graph is converted into high-level MLIR. We then apply our MLIR transformation passes as seen in Figure 3.2 to introduce quantization. Finally, we apply Inceptron's lowering pipeline to transform the MLIR into an executable artifact. The executable artifact is then loaded and run by the MLIR runtime, with quantization parameters passed alongside the model inputs.

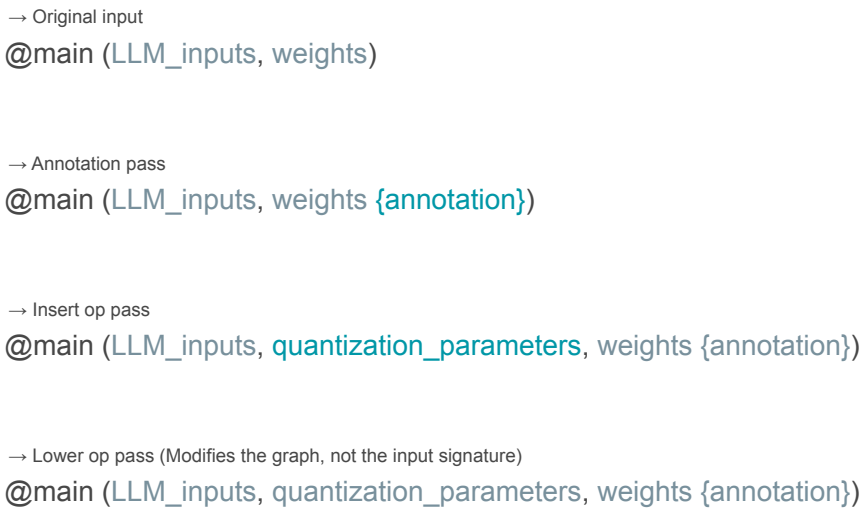

Figure 3.2: Input signature changes during transformation passes. Each row represents the input signature after the corresponding pass. Changes between rows are highlighted.

3.3.3 Annotation Pass

As a part of the creation of the prepared MLIR model (see Figure 3.1) we run three passes, the first of which is an annotation pass. This pass identifies what inputs to MLIR `@main` function are quantizable weight tensors and marks them for later passes. Inputs to the `@main` function include not only quantizable weights, but also model inputs and other parameters that have no quantizable use.

The pass allows filtering which types of tensors will be marked for quantization. The vector-embedding matrix is filtered out by default. Inputs that are of rank-one are optionally filtered, such as bias weights.

3.3.4 Op Passes

These passes are responsible for changing the `@main` function signature to include quantization parameters for the weight tensors marked by the annotation pass. It scans the `@main` function arguments and for each marked argument, it inserts new parameters needed for quantization of that marked weight tensor. The second responsibility of this pass it to insert the operations that perform the dequantization of the marked tensor. These passes do the following things:

1. It changes the input signature of the MLIR function by replacing each annotated argument with quantization parameters.
2. It inserts the calculations needed to reconstruct the original tensor via the new parameters.
3. It changes the networks routing so that it uses reconstructed quantized weights and removes the original full-precision weights.

3.4 Optimization

The optimization stage uses an iterative cycle of black box optimization for quantization parameter suggestion, and evaluation of those choices when applied to the compiled MLIR artifact.

3.4.1 Profiling

For the codebook quantization, the method distributes quantization bins across the range of weight values based on the cumulative importance of the weights. To be able to gather information about the importance of each weight the network needs to be profiled. Therefore we set up a profiling script which uses the PyTorch model's hidden states to acquire the input's activation values throughout the network.

We use activation magnitude as weight importance, meaning we look at the activations of the weight's corresponding channel in the hidden states.

3.4.2 Optimization Loop

Our optimization loop follows an iterative cycle consisting of two main components, as illustrated in Figure 3.1. In each iteration, the BO component selects a new quantization configuration based on previously evaluated configurations. The selection is evaluated with the MLIR program, computing a loss value based on model perplexity and relative size.

The loss value is then fed back into the BO algorithm, which updates its internal model. Based on this updated model, a new configuration is proposed, and the process repeats. This loop explores the search space in a systematic way, focusing on configurations that improve the memory-accuracy trade-off.

In this work, we use Hyperopt's Tree-structured Parzen Estimator (TPE) algorithm to guide the search for optimal quantization parameters. The choice of TPE is motivated by its ability to handle high-dimensional search spaces more effectively than traditional Gaussian process-based Bayesian optimization methods, as discussed in Section 2.4.

Model Performance

To evaluate model performance efficiently, we use perplexity computed over a set of evaluation prompts. Perplexity can be obtained directly from model outputs and requires only a forward pass, making it well-suited for iterative optimization.

The optimizer uses *relative quality*, defined as the ratio between the perplexity of the quantized model and the full-precision model, to measure degradation in performance.

Model Size

In addition to performance, we optimize for model size. We define *relative size* as the ratio between the size of the quantized model and the full-precision model. The trade-off between accuracy and size is handled through the objective function, as desribed later in this Section.

Evaluation Prompts

The evaluation prompts are sampled from the WikiText dataset [25]. During optimization, a subset of these prompts is used to compute perplexity efficiently. This subset-based evaluation reduces computational cost while still providing a reliable signal for the optimization process.

Loss Function

We use a loss function to evaluate the MLIR model for a given quantization configuration and a set of evaluation prompts. The function returns a scalar loss value that reflects the trade-off between model accuracy and model size.
The loss function we use is as follows:

$$\text{loss}\,(\text{relative\_quality}, \text{relative\_size}) = 2^{10\cdot(1-\text{relative\_quality})} - \frac{1}{\text{relative\_size}}$$

We designed this function to balance the memory-accuracy trade-off. The exponential term strongly penalizes reductions in accuracy, while the second term rewards reductions in model size. This reflects the practical constraint that even small decreases in accuracy can have a significant impact on performance, and therefore compression is only desirable when the corresponding loss in accuracy is limited.

The parameters of the function are chosen such that, approximately, a 50% reduction in size is only favourable if the corresponding accuracy loss remains below 10%. This function aims to capture the fact that maintaining quality in LLMs is more important then reducing size. A small degradation in performance accuracy can have a significant impact on downstream model performance. For this reason we designed the loss function to penalize reduction in accuracy, and only allowing for compression when the corresponding loss in model quality is very limited.

Search Space

The black-box optimization needs a predefined search space to function. Our search space consists of a choice for each quantizable tensor in the network. Each choice is a choice between a preselected set of available bit widths (usually between three or four bits).

3.5 Simulating Quantization

Due to previously mentioned (see Section 3.1) hardware limitations in supported data types, quantization must be simulated in order to evaluate model behaviour. This means that the computations that make up the model is performed in its original data type. Quantization is instead performed by calculating the quantization error of the weights and updating them accordingly. This simulated quantization preserves the original execution pipeline, allowing the model to run on the target hardware while accurately injecting the quantization error that true quantization would introduce.

3.5.1 Validation

In order to verify that our solution correctly quantizes a model we create a validation program that compares our MLIR-based quantization pipeline against PyTorch. It verifies that our solution correctly simulates a quantized model.

Chapter 4
Evaluation

The goal of our experiments is to evaluate whether two methods of mixed-precision BO guided quantization can improve the memory-accuracy trade-off compared to fixed-precision quantization for the Llama-3.1-8B model.

4.1 Experimental Setup

We compare four quantization methods: Fixed-precision affine quantization (FAQ), fixed-precision codebook quantization (FCQ), BO-guided affine quantization (BOAQ), and BO-guided codebook quantization (BOCQ). For the fixed-precision methods 4-bit quantization is used as the baseline, as this provides a strong memory-accuracy trade-off for LLM quantization [31]. We selected these four methods as they allow us to isolate the effects of the quantization technique.

When comparing affine vs codebook techniques our goal is to evaluate whether more expensive techniques provide benefits in terms of memory-accuracy trade-off in, both baseline and in combination with BO.

For the BO-guided methods, the BO search space consists of per tensor bit-width choices. Each tensor is assigned 3 or 4 bits. This way the BO can reduce memory usage by assigning 3-bits to less sensitive tensors while at the same time keeping more sensitive tensors in 4-bits. The goal is for the BO to create models which are smaller than their fixed-precision counterparts.

We designed the BO search space to only include the choice between 3 and 4 bits as this is the threshold for where major performance degradation is seen in affine and codebook quantization. Including higher and lower bit-widths would dramatically increase the already large search space without any clear benefits.

We evaluate group sizes **64, 128, 256, 512** as they affect both quantization error and overhead size, as explained in Section 2.2.4. As group size grows, the quantization becomes more coarse, leading to smaller overhead, but worse performance. Evaluating multiple group sizes

allows us to study the effect of group size on the memory-accuracy trade-off.

Each configuration of group size and quantization method is evaluated for 2000 BO trials. The trial that results in the best loss value is selected to be used for the full evaluation. We stop the experiments after 2000 iterations due to the high computational cost of each evaluation. For affine quantization 2000 iterations requires about 8 GPU hours on a NVIDIA 5090 32GB GPU, while codebook is slightly faster due to caching, requiring about 5 hours.

After the BO, the model and its fixed-precision counterpart is evaluated on five evaluation benchmarks. We evaluate perplexity on Wikitext and downstream accuracy on MMLU, ARC-Challange, HellaSwag and PIQA. Together these benchmarks provide a trade-off between evaluation and computational costs, as explained in Section 2.1.6.

The perplexity score used in our objective function is calculated from the model's output on a subset of WikiText. The same subset is used for each BO run so the perplexity score does not change between runs. As there is no current hardware support for mixed bit-width calculations the results are computed in full-precision, but with the quantization error introduced as explain in Section3.5.

Because the quantization is simulated, we have to calculate the model compression instead of reading it. The size of the model is calculated according to Section 2.2.4.

4.2 Results

This sections describes our results. It begins with 3-bit fixed-precision quantization, continues with 3/4 bit mixed-quantization together with 4-bit fixed-precision and finishes with general BO results.

4.2.1 3-Bit Fixed-Precision Quantization

Table 4.1 shows that fixed 3-bit quantization results in a substantial perplexity and downstream accuracy degradation for both methods and all group sizes. This confirms that 3-bits is insufficient for maintaining model quality and motivates our use of 3/4-bit search space for the BO guided methods, and our choice of 4-bits for the fixed-precision baseline.

Interestingly codebook quantization has both perplexity and downstream accuracy benefits over affine quantization for group sizes **64, 128, 256**, which cannot be seen in 4.2

Group size	Method	Relative size	WikiText PPL	Downstream accuracy	Downstream loss
–	Full precision	1.000	7.541	0.640	0.000
64	FAQ	**0.218**	19.100	0.570	**-2.442**
64	FCQ	0.312	**15.696**	**0.577**	-1.227
128	FAQ	**0.203**	22.540	0.547	**-2.167**
128	FCQ	0.250	**18.325**	**0.550**	-1.336
256	FAQ	**0.195**	28.520	**0.523**	**-1.560**
256	FCQ	0.218	**24.839**	0.504	-0.216
512	FAQ	**0.191**	**37.331**	**0.491**	**-0.190**
512	FCQ	0.203	59.031	0.441	3.744

Table 4.1: Evaluation of Affine and codebook quantization at fixed precision 3 bit quantization for all tensor. Best values per group size are highlighted.

4.2.2 3/4-Bit Mixed-Precision Quantization

The results in Table 4.2 show a few different trends for quantization methods and group sizes.

First, the BO-guided methods achieve lower relative model sizes compared to their full precision counterparts for all group sizes. This demonstrates that the BO guided method successfully improves compression. This compression comes at the cost of model performance. In all group sizes, and for both affine and codebook methods the mixed-precision models show higher perplexity and lower accuracy compared to the fixed precision counterparts. This effect becomes more prominent as group size increases.

Second, comparing affine an codebook quantization the latter generally retains better perplexity but has similar downstream accuracy, all while being substantially larger.

Third, increasing group size leads to a improvement in model compression at the cost of quality. For fixed-precision, our loss function yields better values the higher the group size. In contrast, for BO-guided quantization the function yields better values at group sizes 128 and 256 than for 64 and 512.

Group size	Method	Relative size	WikiText PPL	Downstream accuracy	Downstream loss
–	Full precision	1.000	7.541	0.640	0.000
64	FAQ	0.281	10.754	**0.631**	-2.450
64	FCQ	0.500	**10.579**	0.627	-0.846
64	BOAQ	**0.253**	14.179	0.609	**-2.542**
64	BOCQ	0.403	13.841	0.608	-1.060
128	FAQ	0.266	**10.990**	**0.623**	-2.548
128	FCQ	0.375	11.027	0.622	-1.442
128	BOAQ	**0.236**	14.848	0.596	**-2.618**
128	BOCQ	0.322	14.687	0.592	-1.424
256	FAQ	0.258	**11.206**	**0.622**	-2.661
256	FCQ	0.313	13.654	0.604	-1.714
256	BOAQ	**0.228**	15.973	0.582	**-2.502**
256	BOCQ	0.278	15.782	0.584	-1.754
512	FAQ	0.254	**13.850**	**0.615**	**-2.619**
512	FCQ	0.281	14.684	0.598	-1.983
512	BOAQ	**0.232**	16.220	0.582	-2.421
512	BOCQ	0.254	18.147	0.547	-1.193

Table 4.2: Evaluation of four quantization methods on Llama-3.1-8B. Best results per group size are highlighted.

4.2.3 Pareto Frontier

Figure 4.1 shows the observed Pareto frontiers for different group sizes and quantization techniques. The observed Pareto frontiers are based on 2000 data points and represent an approximation of the trade-off. Each colour represents a Pareto frontier of BO-trials, where each point is non-dominated. This means that no other BO configuration has both lower relative size and lower perplexity. The Pareto frontier represents the observed optimal trade-off between perplexity and relative size.

The crosses in Figure 4.1 show the fixed-precision equivalent. The gap between the fixed-precision baseline and the Pareto frontiers shows the model compression. All mixed-precision models are smaller than their fixed-precision counterparts, but also worse performing. The notable exceptions are affine quantization group 64 and 128, which are smaller but have equivalent perplexity.

Furthermore, Figure 4.1 highlights that the memory-accuracy trade-off is better for the affine schemes than the codebook, which is supported by the results from Table 4.2. This indicates that the additional overhead associated with codebooks does not improve memory-accuracy trade-off.

Finally, the concentration of points along the lower end of the Pareto frontiers suggest that the BO searches the configuration space more effectively than random search would.

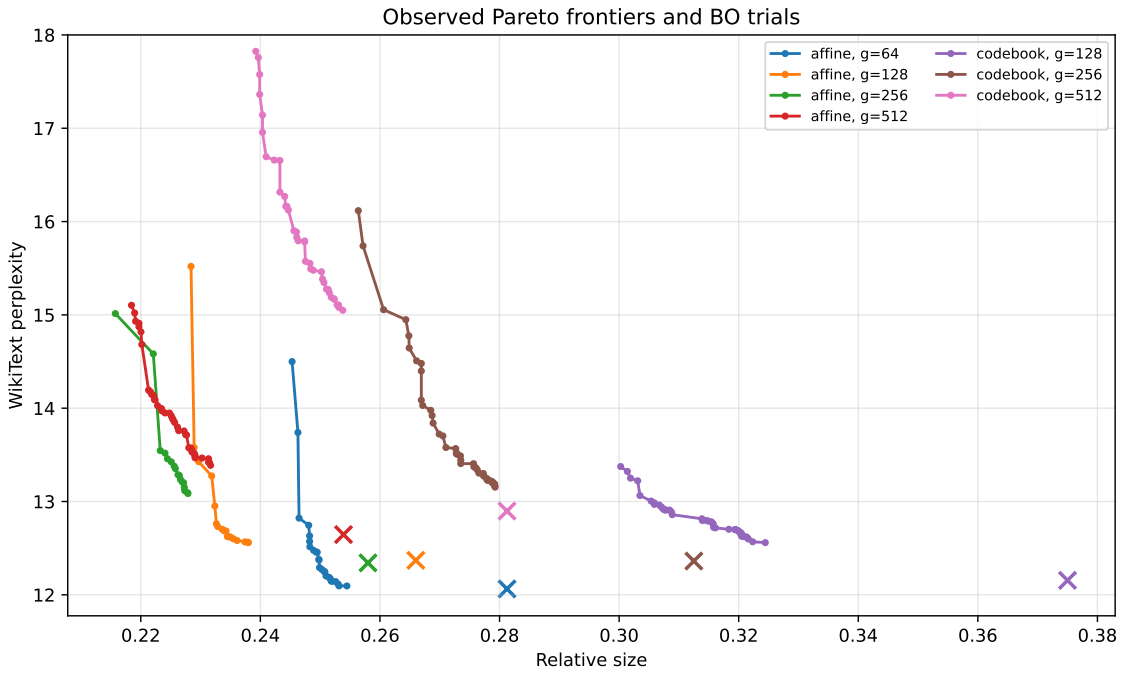

Figure 4.1: The observed Pareto frontier for BO-guided quantization.

4.2.4 BO Performance

The results show that the iterative method improves the memory-accuracy trade-off over time. To explore if BO is capable of finding tensor-level bit-width assignments that improve this trade-off, we plot the best loss over sets of 50 trials in Figure 4.2 for the affine models and in Figure 4.3 for codebook quantization. Both figures show that the loss improves as the number of trials increases.

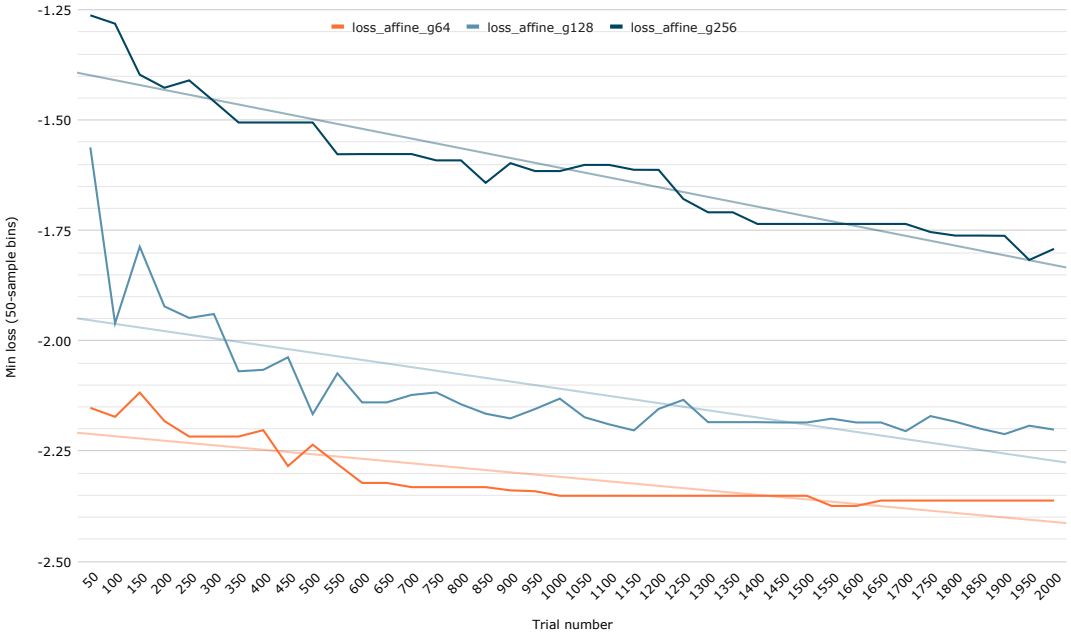

Figure 4.2: The minimum loss per 50-sample bin for affine quantization.

The BO starting point seems to have a strong influence on the final optimization result. In Figure 4.4 we present results of 20 independent BO runs (BOAQ at group size 256) of 2000 iterations each, as well as one run where the bit-width distribution is picked at random. It shows that the initial decision has a large impact on the end results, and that there is large variance in the performance between runs. This may suggest that the optimization function has many local minima.

The BO runs outperform the random baseline as is represented by the purple line in Figure 4.4. This indicates that the BO optimizer can learn about the search space rather than simply picking bit-width assignments are random. Further this suggests that some tensors or combinations of tensors are more sensitive to quantization than others.

To investigate the long term behaviour of the optimization process we also conducted an isolated test where we let the BO loop run for 20 000 iterations. In this experiment, the BO loop continued to discover lower loss bit-width distributions throughout the run. This suggests that the search had not converged after 2000 iterations and that searching further may find better bit-width distributions.

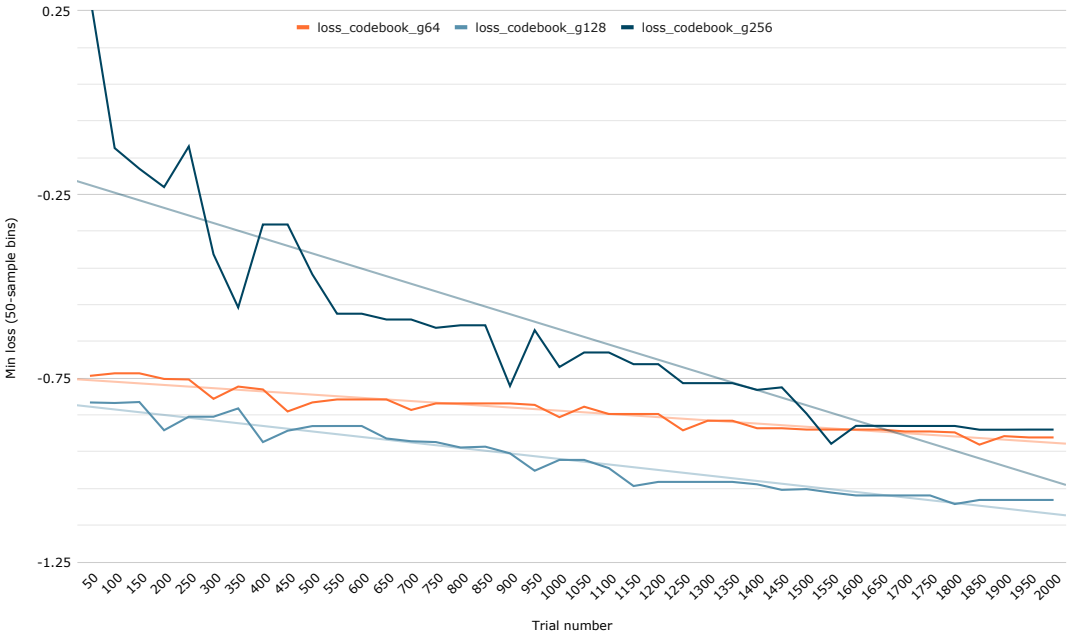

Figure 4.3: The minimum loss per 50-sample bin for codebook quantization.

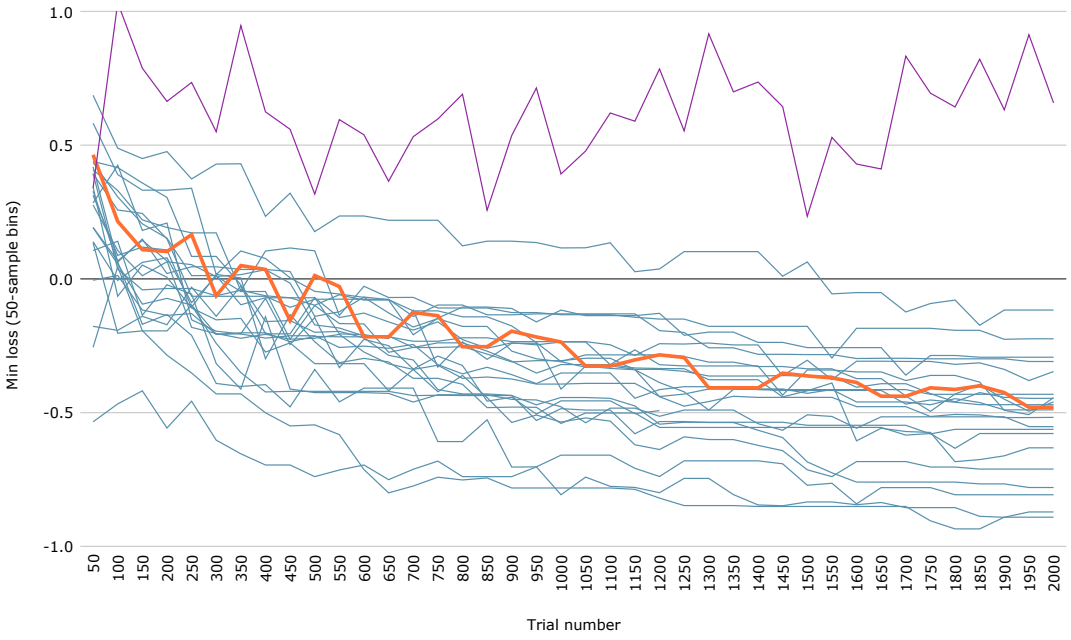

Figure 4.4: 20 runs of codebook optimization with group size 256. Median value is highlighted in orange. Random run is highlighted in purple.

Chapter 5
Discussion

In this section we discuss our results and suggest explanations for our observations. The main themes are memory-accuracy trade-off, comparison between affine and codebook quantization and the effectiveness of BO guided quantization. We also discuss threats to validity and future work.

5.1 Memory-Accuracy Trade-Off

The results demonstrate that our method is capable of reducing the size of models through quantization. They also show that the choice of quantization configuration has a large impact on the memory-accuracy trade-off. Across most evaluated configurations, affine quantization achieves a more favourable memory-accuracy trade-off than codebook quantization. The primary reason for the difference in memory-accuracy trade-off is the additional quantization overhead associated with codebooks, rather than differences in accuracy.

A relationship can be observed between group size, model size and quality. For group sizes 64, 128 and 256, increasing group size reduces model size, worsens perplexity and downstream task accuracy. This behaviour is expected as larger group sizes force larger weights distributions to share the same quantization parameters. This confirms that group size is a vital component in the memory-accuracy trade-off.

The BO-guided mixed-precision models indicate that Bayesian optimization can identify tensors, or combinations of tensors, that are less damaged by 3 bit quantization. Affine quantization at group sizes 64 and 128 show particularly promising results. In these cases BO guided quantization achieves approximately 4% model size reduction compared to their fixed-precision counterparts while retaining perplexity. This suggest that some tensors, or combinations of tensors, can be quantized more aggressively than others. We believe that the shape of the Pareto frontiers provide evidence for this claim, as the frontiers show large improvements in perplexity for relatively small model size increases. This holds true up to a certain point, then further compression causes increased perplexity. This suggest that the

remaining tensors after a certain point are more sensitive to quantization.

Studying the relationship between perplexity and downstream task performance reveals a non-linear correlation between these performance metrics. A worsening of perplexity does not cause a proportional loss of performance in the multiple choice benchmarks. Perplexity is computed as an exponential of the average negative log-likelihood, making it highly sensitive to small changes. On the other hand, multiple choice benchmarks only require that the model assigns higher probability to the correct answer. This means that quantization can reduce model confidence without changing its choice of answer. Further this suggest that quantization may affect model confidence more than its reasoning ability.

The difference between the two metrics raises the question of using multiple choice evaluations as a more direct metric for model usefulness inside the BO loop. The modest degradation of multiple choice accuracy across the evaluated methods may suggest that optimizing exclusively for perplexity may overestimate the models degradation caused by quantization. Exploring a quantization objective that incorporates downstream task benchmarks may yield more useful results, although this comes at a computational cost.

5.2 Effectiveness of Bayesian Optimization

The results suggest that the BO used in the iterative optimization method is capable of finding meaningful quantization policies. The optimization curves shown in Figure 4.2 and Figure 4.3 highlight this. In both affine and codebook quantization, the minimum loss decreases as the number of BO trials increases. This indicates that the BO can find useful information from previous evaluations and improve its configurations over time. Furthermore, by selecting which tensors to quantize to 3 bits the method achieves better results than a random selection does.

However, the configuration search problem is difficult to optimize. There exists a large variance between BO runs which may suggest that the optimization problem contains many local minima. One possible explanation is that tensor quantization decisions are interdependent. There may be interactions between tensors where tensors tolerate lower precision only when certain other tensors remain in higher precision. This possibility further complicates the curse of dimensionality, introducing a combinatorial aspect to the configuration search problem.

The continued improvement observed in the 20000 iteration run motivates the need for an efficient evaluation framework. After these iterations the problem has not converged on a loss value, and beyond 20000 iterations we don't have any data. The total computational costs of running these iterations is large, although the method appears to find structures to learn from in the process. By lowering the computational costs the iterative BO method may prove to be a viable way of solving the configuration search problem.

Another observation is that the improvements in size are relatively small. The BO identifies lower-size models, but their gains over the fixed precision models are relatively modest, just a few precent in relative size. This could be explained by some tensors being "easy wins" for the BO loop, while others cannot be quantized without large model degradation.

5.3 Affine vs Codebook Quantization

The results show that for 4-bit quantization, FCQ is generally outperformed by FAQ across all group sizes, with the exception of group size 64 where FCQ achieves a slightly lower perplexity. For 3-bit quantization, FCQ outperforms FAQ in the accuracy metrics for lower group sizes and performs worse for higher ones. This suggests that codebook quantization is more effective at preserving model quality at very low bit-widths.

The overall memory-accuracy trade-off is in favour of affine quantization. The metadata required for storing codebooks causes large overhead, resulting in significantly larger model sizes at the same bit-width for FCQ compared to FAQ.

A similar pattern is observed for the mixed-precision BO-guided models. BOCQ achieves slightly better perplexity than BOAQ for most group sizes, except at 512. However, these gains are modest and come at the cost of increased model size, meaning that the overall trade-off remains in favour of affine quantization.

Overall, the results suggest that group-level codebook quantization is not preferable to affine quantization using our method. Although codebook quantization can reduce quantization error at low bit-widths, the associated metadata overhead outweighs these gains. For the evaluated Llama-3.1-8B configurations, affine quantization provides a more favourable balance between compression and model quality.

Codebook quantization can in theory reduce quantization error by more effectively utilizing the available bits, as explained in Section 2.2.2. In our implementation we approximate the importance of weights using activation magnitudes. The intuition is that weights associated with larger activations contribute more to the model output and should therefore be represented more accurately. However, we believe this interpretation is too simple. Our method does not preserve individual weights directly, rather it preserves numerical regions associated with weights deemed important. We construct the codebooks from importance-weighted statistics so that the resolution in important numerical regions is higher. However, large activations only show that a weight is included in computations that have large effects, but not necessarily how sensitive the model is to changes to that weight. This means that our way of measuring weight sensitivity might not correspond with the true weight sensitivity (salience).

We believe this provides a possible explanation for the results comparing affine and codebook methods. Our insufficient way of measuring weight sensitivity limits us from preserving the most important weights of the model. Using more accurate measures of weights sensitivity, similar to what AWQ or GPTQ uses may yield better results in terms of memory-accuracy trade-off.

5.4 Usage of MLIR

MLIR can be used to modify the LLM so that quantization parameters can be passed at runtime, instead of being fixed at compile time. We use this for our BO guided solution where thousands of quantization parameter configurations are tested separately. This way, we can reuse the same compiled model for all optimization trials. The alternative is an expensive reload or recompilation at each trial. The dynamically injected quantization parameters allows the iterative quantization search to leverage compiler-level optimizations.

Using MLIR we implement quantization as a compiler pass directly on the intermediate representation. What passes are used is easily changed, making this approach highly modular and reusable. The annotation pass identifies tensors, while the op passes insert the quantization logic. The op passes can be exchanged with other ones that perform quantization in a different way, as we do with affine and codebook quantization. The passes can be exchanged without changing the rest of the compilation pipeline.

Another benefit of MLIR is that our quantization happens at a high abstraction level. Our operations are transformed and lowered to be executed on NVIDIA GPUS, but as MLIR uses progressive lowering this hardware target can easily be changed (as long as its supported in MLIR). In theory our quantization infrastructure should be reusable on many different hardware backends.

Effectively combining quantization with Inceptron's efficient MLIR pipeline was more difficult then initially expected. Therefore, our final experiments were performed using the PyTorch validation implementation, not the MLIR implementation. Instead of showing that MLIR is more performant, we show that MLIR is an alternative for this type of iterative task, while achieving highly optimized runtime performance is beyond the scope of this thesis.

5.5 Threats to Validity

This section presents identified threats to validity. From the general lack of peer-reviewed articles in the field to our simulation of quantization.

5.5.1 arXiv Pre-Prints

A general issue when doing research in the ML field is that the research moves very quickly. If you want to cover the latest algorithms, methods and discoveries, odds are that they aren't peer-reviewed. Pre-prints are usually published on arXiv and when this initial publication is followed by a peer-reviewed one it is usually years later. For example, in the published paper on a method called SmoothQuant (see 6.Related Work), GPTQ is cited as an arXiv pre-print [35]. GPTQ has been peer-reviewed and published between the publishing of SmoothQuant and the writing of this master's thesis, therefore we reference the published article [13].

Having research be built on non-peer-reviewed work is a risk. With the ML field moving as quickly as it does, it is hard to not depend on these papers if you want to refer to the latest work. We have searched for published papers where possible, nevertheless there are quite a few references to arXiv pre-prints in this work.

5.5.2 The Large BO Search Space

With our Llama model and Bayeisan optimization structure, the set of possible quantization configurations contains $2^{224} = 2.7 * 10^{67}$ different variations. This search space is impossibly large for the BO to fully explore and it therefore not expected to find the optimal solution. This is not unusual for black box optimizations methods, they are not expected to find the optimal solution, but rather a sufficiently good one. However, during our experiments we have observed that the algorithm is heavily reliant on its initial guesses and tends to gets stuck

in local minima. Runs with the same configuration get varying results and the TPE algorithm (see Section 2.4.) tends to get stuck and suggest previously evaluated policies during later stages of the run.

5.5.3 Simulated Quantization

Since we simulate end to end quantization of the model, it is possible our simulation will slightly differ from an actual implementation. Our size calculations assume perfect packing and unpacking of 3-bit weights. Although quantization generally has a positive impact on model speed [8, 13], there is no guarantee this holds true for our specific mixed-precision scheme. Therefore, our size calculations and results should not be taken at face value, but should instead be used as a general indication of the methods effectiveness.

5.5.4 Testing on Other LLMs

In this work we have primarily used Meta's Llama 3 model, both its one and eight billion parameter variants. During the development of the method we also created and used a MLP model and the 125 million variant of Meta's OPT. We have not tried our method on other large scale LLMs as it would require extra work adapting our framework. As all LLMs are structurally similar, it is likely that our method would work in a similar ways on non-Llama LLMs, however this has not been tested.

5.6 Future Work

As discussed in the previous section, a limitation of our method is the large search space which the Bayesian optimization method has to explore. With more time, we would have explored ranking the weights by importance and not quantizing those of most importance, instead focusing on the less important weights - similar to the philosophy of *GPT3.int8()* [8]. An alternative solution could be having a choice in the BO impact a group of tensors, instead of a choice per tensor like our method. This would cut down on the size of the BO's search space and possibly improve our method. We leave it to future work to explore how the issue of search space size could be addressed.

Additionally, there is room for exploring if alternative quantization schemes and allowing more granular BO decisions could allow salient weights to be better captured. Although this would increase the size of the search space with the addition of further combinatorial complexity. Exploring this search space could require some other method than BO, and a more efficient computational framework.

Lastly, a main limiting factor of our method and something which future work could improve upon is the iteration speed of the Bayesian optimization. The time of one iteration increases as the method progresses. Improving the iteration time would allow the BO to explore its search space at a faster rate and it would thus be more likely to find a good result.

Chapter 6
Related Work

In this section we present related work divided into three categories; Post-training quantization for LLMs, mixed-precision and layerwise quantization and optimization-based quantization.

6.1 Post-Training Quantization for LLMs

The papers in this section all address post-training quantization for large language models. Their common goal is to reduce memory requirements and inference cost while maintaining model quality. These methods are all based on the observation that quantization sensitivity is unevenly distributed throughout the model, meaning that some weights, channels, or activations are more sensitive to quantization than others.

Our work formulates quantization as an iterative optimization problem rather than relying on a predefined quantization strategy. The following works therefore differ primarily in how they identify quantization-sensitive components and how quantization policies are selected.

6.1.1 GPT3.int8() (2022)

GPT3.int8() introduces an outlier-aware mixed-precision quantization method for transformer models [8]. The authors observe that a small number of activation dimensions contain extremely large values that cause large quantization errors when quantized. To address this, most matrix multiplications are quantized while the outlier dimensions are kept at a higher precision. This allows the model to maintain near full-precision accuracy while reducing memory usage and inference cost. GPT3.int8() was one of the first works to show that large transformer models could be quantized without retraining, while keeping good model performance.

In contrast to our work that uses a more aggressive 3-4 bit quantization, GPT3.int8() quantizes to 8 bits and tries to keep the performance of the model broadly unchanged. Additionally, GPT3.int8() focuses on selectively protecting activation outliers during inference with mixed-precision within operations, while our work instead focuses on tensor-wise mixed-precision quantization of weights.

6.1.2 GPTQ (2022)

GPTQ (Generative Pre-trained Transformer Quantization) is a one-shot post-training quantization method designed for large transformer models [13]. The method enables 3-4 bit quantization while maintaining relatively small accuracy loss. GPTQ only quantizes weights and leaves activations in higher precision. To reduce the quantization error, GPTQ uses second-order information derived from the Hessian matrix to estimate the sensitivity of weights and to compensate for quantization errors. GPTQ showed that 3-4 bit quantization could be achieved without retraining and became a common baseline for later quantization methods.

6.1.3 SmoothQuant (2023)

SmoothQuant proposes a method for stable quantization of both weights and activations in large language models [35]. The method is based on the observation that activation outliers are harder to quantize than weights. SmoothQuant applies a mathematically equivalent transformation that redistributes quantization difficulty from activations into weights through channel-wise scaling. This smooths the activation distribution and enables W8A8 quantization with limited degradation. The method is designed to be hardware friendly and enables efficient INT8 inference on modern accelerators.

6.1.4 AWQ (2024)

AWQ (Activation-Aware Weight Quantization) is a low-bit weight-only post-training quantization method designed for LLMs [23]. The method is based on the observation that only a small subset of weights, approximately 0.1%–1%, are highly sensitive to quantization. These so-called salient weights are identified by analysing activation distributions using a calibration dataset. Instead of storing salient weights in higher precision, which introduces hardware inefficiencies, AWQ scales these weights before quantization and compensates for the scaling during dequantization. This allows the method to preserve model quality while remaining hardware friendly. AWQ has become a commonly used baseline for low-bit LLM quantization frameworks such as HuggingFace and vLLM.

6.2 Mixed-Precision and Layerwise Quantization

The following works are related to our method because they use mixed-precision quantization rather than assigning the same precision throughout the network. These methods are based on the idea that different parts of a model tolerate quantization differently and should be quantized accordingly.

Similar to our work, these methods aim to improve the memory-accuracy trade-off by allocating precision where it has the greatest impact on model quality. Unlike these methods, which determine quantization policies using analytical sensitivity metrics or predefined procedures, our work treats quantization policy selection as an iterative optimization problem.

6.2.1 HAWQ (2019)

HAWQ (Hessian AWare Quantization) introduces a mixed-precision quantization framework that assigns different bit-widths to different layers based on their sensitivity to quantization noise [9]. The method estimates layer sensitivity using second-order information from the Hessian matrix under the observation that layers with larger Hessian eigenvalues are more sensitive to quantization errors. By allocating higher precision to sensitive layers and lower precision to more robust layers, HAWQ achieves better accuracy-efficiency trade-offs than uniform quantization schemes.

A key difference between HAWQ and our work is that HAWQ derives bit-width assignments directly from Hessian-based sensitivity estimates, whereas our method searches for quantization policies through iterative evaluation. HAWQ also primarily targets CNN-style networks, while our work focuses on transformer-based large language models.

6.2.2 ZeroQuant (2022)

ZeroQuant presents a hardware-aware post-training quantization framework designed for transformer-based models such as BERT and GPT [36]. The framework combines fine-grained layerwise quantization, efficient kernel implementations, and knowledge distillation to enable low-bit inference with limited accuracy degradation. ZeroQuant also introduces optimized CUDA kernels and operator fusion techniques to improve runtime performance on modern hardware.

ZeroQuant mainly focuses on optimized inference kernels and deployment efficiency, whereas our work instead focuses on the search and evaluation of mixed-precision quantization configurations.

6.3 Optimization-based Quantization

The following works are related to our method because they formulate quantization, or a closely related problem, as an optimization task. Rather than relying entirely on manually designed solutions, these methods use automated search procedures to identify configurations that balance quality and efficiency.

Similar to our work, these methods operate in large configuration spaces where evaluating a solution is computationally expensive. The primary difference lies in the choice of optimization algorithm and optimization objective. While our work uses Bayesian optimization to search for tensor-wise quantization policies, the following works explore alternative optimization strategies for automated configuration search.

6.3.1 BOHB (2018)

BOHB (Bayesian Optimization and HyperBand) is a hyperparameter optimization method that combines Bayesian optimization with HyperBand resource allocation [11]. The method balances exploration and exploitation by using Bayesian optimization to guide the search toward promising configurations while using partial evaluations to terminate poorly performing candidates early. BOHB is particularly effective for optimization problems where evaluating configurations is computationally expensive.

BOHB is related to our work because both methods use Bayesian optimization in settings where evaluation is computationally expensive. However, BOHB is a general-purpose optimization framework and is not specifically designed for quantization.

6.3.2 HAQ (2019)

HAQ (Hardware-Aware Automated Quantization) introduces a reinforcement learning-based framework for automatically selecting layerwise quantization policies under hardware constraints [34]. The method trains an RL agent to choose per-layer bit-widths while optimizing objectives such as accuracy, latency, energy consumption, and model size. HAQ interacts directly with hardware simulators or devices, allowing the learned quantization policies to adapt to different hardware platforms.

Similar to HAQ, our work searches for tensor-wise quantization policies under memory and accuracy constraints. However, HAQ uses reinforcement learning to learn quantization policies, whereas our work applies Bayesian optimization as a black-box search strategy.

6.3.3 BOMP-NAS (2023)

BOMP-NAS is a multi-objective neural architecture search framework that jointly optimizes neural network architectures and mixed-precision quantization policies [32]. The method formulates model design as a multi-objective optimization problem balancing accuracy, latency, computational cost, and model size. By jointly searching both architecture and quantization configurations, the framework is able to find hardware-efficient neural network deployments.

Similar to BOMP-NAS, our work also searches for mixed-precision quantization configurations instead of applying fixed precision uniformly across the network. However, BOMP-NAS jointly optimizes neural network architectures and quantization policies, whereas our work assumes a fixed pretrained LLM architecture and focuses only on quantization parameter search.

Chapter 7
Conclusion

In this section we present a summary of our work and answer our stated research questions.

7.1 Summary

By quantizing the weights of a LLM, inference and storage becomes more efficient. Quantization comes at a cost of degradation in model performance. The trade-off between size and performance can be improved by mixed-precision quantization, where weights are quantized differently depending on their sensitivity to quantization.

In this thesis we explore iterative mixed-precision quantization using Bayesian optimization. Our method assigns bit-widths at tensor level and evaluates the resulting model iteratively. Each iteration selects a new bit-width distribution based on previous knowledge. LLM evaluation is expensive and we investigate whether MLIR can be used to make this evaluation less expensive. We utilize MLIR to inject quantization parameters at runtime, which allows the same compiled model to be reused between iterations, saving computational resources.

We also evaluate two quantization schemes: affine and codebook quantization. Affine quantization is more size-efficient while codebook can, in theory, reduce quantization error by better approximating the original distribution. These schemes were combined with the iterative BO method. We evaluate both fixed-precision and BO-guided mixed-precision configurations for different group sizes and benchmarks.

Our results show that Bayesian optimization can find meaningful quantization policies. The best memory-accuracy trade-off is achieved by small group size affine quantization, where the BO guided method clearly outperforms fixed-precision quantization. Our codebook implementation suffered from substantial metadata overhead and affine quantization is therefore a better balance between compression and model quality for the Llama-3.1-8B model.

We demonstrate that tensor-wise quantization is a difficult high-dimensional optimization problem. The large variance between optimization runs and a non-converging 20000

iteration run suggest that the search space is complex. It also demonstrates that iterative quantization is feasible, and that Bayesian optimization can identify useful tensor combinations and learn from previous experience. At the same time, we highlight that an efficient runtime and/or limiting the search space is required for making the iterative method computational feasible.

7.2 Research Questions

RQ1: How does a black-box optimization guided mixed-precision quantization method compare to fixed-precision quantization with respect to memory-accuracy trade-off?
BO-guided mixed-precision quantization can improve the memory-accuracy trade-off compared to fixed-precision quantization when selecting between 3-bit and 4-bit tensor distribution. For affine quantization we can see clear improvements where mixed-precision achieves lower model size with very limited model degradation.

RQ2: To what extent can Bayesian optimization be used to select tensor-level quantization policies that improve memory-accuracy trade-off?
The results show that BO is capable of finding meaningful tensor-wise quantization policies and improving the memory-accuracy trade-off. Although the optimization problem is challenging due to a high-dimensional search space, our BO learns to find better results over time. After 20000 iterations the BO had not converged suggesting that BO is capable of iteratively improving the loss, but remains inefficient and limited by evaluation cost and search space complexity.

RQ3: How do different group-wise quantization schemes affect the memory–accuracy trade-off in iterative mixed-precision quantization?
Affine quantization outperforms our codebook quantization. In theory codebook quantization reduces quantization error by adapting to weight distributions, but the additional metadata overhead associated with the codebooks outweighed these benefits for our implementation. Affine quantization achieves better compression and performance for the evaluated Llama-3.1-8B model.

RQ4: To what extent can MLIR be used to evaluate different quantization configurations for large language models?
MLIR proved to be a flexible framework for iterative quantization experimentation. By using transformation passes and runtime parameter injection, the same compiled model could be reused across many quantization configurations, reducing recompilation overhead during optimization. However, our implementation did not outperform the PyTorch validation program. With more extensive compiler and hardware optimization expertise, we believe it can serve as a powerful tool.

References

[1] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[2] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[3] James Bergstra, Daniel Yamins, and David D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, volume 28 of *Proceedings of Machine Learning Research*, pages 115–123. JMLR.org, 2013.

[4] Mickael Binois and Nathan Wycoff. A survey on high-dimensional gaussian process modeling with application to bayesian optimization. *ACM Transactions on Evolutionary Learning and Optimization*, 2(2):1–26, 2022.

[5] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[7] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv:1803.05457v1*, 2018.

[8] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems*, 35:30318–30332, 2022.

[9] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Hawq: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 293–302, 2019.

[10] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International conference on machine learning*, pages 5547–5569. PMLR, 2022.

[11] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International conference on machine learning*, pages 1437–1446. PMLR, 2018.

[12] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[13] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[14] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024.

[15] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-power computer vision*, pages 291–326. Chapman and Hall/CRC, 2022.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[17] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

[18] Wei Huang, Xingyu Zheng, Xudong Ma, Haotong Qin, Chengtao Lv, Hong Chen, Jie Luo, Xiaojuan Qi, Xianglong Liu, and Michele Magno. An empirical study of llama3 quantization: From llms to mllms. *Visual Intelligence*, 2(1):36, 2024.

[19] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[22] Shiyao Li, Xuefei Ning, Luning Wang, Tengxuan Liu, Xiangsheng Shi, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. Evaluating quantized large language models. *arXiv preprint arXiv:2402.18158*, 2024.

[23] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of machine learning and systems*, 6:87–100, 2024.

[24] Zechun Liu, Changsheng Zhao, Hanxian Huang, Sijia Chen, Jing Zhang, Jiawei Zhao, Scott Roy, Lisa Jin, Yunyang Xiong, Yangyang Shi, et al. Paretoq: Improving scaling laws in extremely low-bit llm quantization. *Advances in Neural Information Processing Systems*, 38:91311–91336, 2026.

[25] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[26] Andrew Or, Apurva Jain, Daniel Vega-Myhre, Jesse Cai, Charles David Hernandez, Zhenrui Zheng, Driss Guessous, Vasiliy Kuznetsov, Christian Puhrsch, Mark Saroufim, Supriya Rao, Thien Tran, and Aleksandar Samardžić. Torchao: Pytorch-native training-to-serving model optimization, 2025.

[27] Adam Paszke, Sam Gross, Francisco Massa, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[28] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[29] Pol G. Recasens, Ferran Agullo, Yue Zhu, Chen Wang, Eun Kyung Lee, Olivier Tardieu, Jordi Torres, and Josep Ll. Berral. Mind the memory gap: Unveiling gpu bottlenecks in large-batch llm inference. In *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, pages 277–287, 2025.

[30] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[31] Luke Zettlemoyer Tim Dettmers. The case for 4-bit precision: k-bit inference scaling laws. *arXiv preprint arXiv:2212.09720*, 2022.

[32] David Van Son, Floran De Putter, Sebastian Vogel, and Henk Corporaal. Bomp-nas: Bayesian optimization mixed precision nas. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–2. IEEE, 2023.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[34] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8612–8620, 2019.

[35] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International conference on machine learning*, pages 38087–38099. PMLR, 2023.

[36] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in neural information processing systems*, 35:27168–27183, 2022.

[37] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pages 4791–4800, 2019.

[38] Rui-Jie Zhu, Yu Zhang, Steven Abreu, Ethan Sifferman, Tyler Sheaves, Yiqiao Wang, Dustin Richmond, Sumit Bam Shrestha, Peng Zhou, and Jason K Eshraghian. Scalable matmul-free language modeling. *arXiv preprint arXiv:2406.02528*, 2024.

[39] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Appendices

Appendix A
Popular Science Summary

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2026-06-11

EXAMENSARBETE Iterative Tensorwise Quantization of Neural Networks Using MLIR
STUDENTER Albin Nyström, Aron Somi
HANDLEDARE Noric Couderc (LTH), Felix Malmsjö (Inceptron)
EXAMINATOR Flavius Gruian (LTH)

En kvantiseringsmetod som utvärderar sig själv

POPULÄRVETENSKAPLIG SAMMANFATTNING **Albin Nyström, Aron Somi**

Storleken på AI-modeller växer i rasande fart, vilket skapar ett ökat behov av resursoptimering. En metod för storleksminskning av AI-modeller är så kallad kvantisering. De flesta metoderna utför endast kvantisering i ett steg, i vårt arbete utforskar vi möjligheten att göra detta till en iterativ process.

I takt med att dagens AI-modeller blir allt större krävs allt mer av hårdvaran som kör dem, vilket gör tillhandahållandet av AI till ett dyrt och på sikt ohållbart åtagande. Ett sätt att göra AI mer resurseffektivt är att komprimera modellerna.

Majoriteten av en modells storlek utgörs av s.k. vikter, vilket förenklat kan beskrivas som de siffror som lagrar modellens kunskap. Komprimering av en modells vikter är därför ett effektivt sätt att minska modellens minnesanvändning. Vid komprimering använder man sig av datorers sätt att representera siffror med bitar. Ett vanligt sätt frigöra minne är att gå från ett talsystem med hög precision som använder många bitar till ett med färre bitar och lägre precision. Detta kallas kvantisering.

Kvantisering av en modells vikter har ett pris, minskningen i antal bitar innebär också en minskning i precision vilket i sin tur påverkar prestandan. För att minimera effekten på en modells prestanda utnyttjar moderna kvantiseringsmetoder att alla vikter i en modell inte har lika stor betydelse. Dessa metoder använder olika verktyg för att bedöma vilka delar av en modell som är mer eller mindre lämpliga för kvantisering. Metoderna skapar sedan en s.k. kvantiseringskonfiguration som beskriver hur modellens olika delar ska kvantiseras.

Vårt examensarbete utforskar en metod som steg för steg söker efter den optimala kvantiseringskonfigurationen av en AI-modell. De etablerade kvantiseringsmetoderna utför avancerade analyser av modellen innan kvantisering, men är inte medvetna om resultaten av sin kvantisering. Vi använder oss av en optimeringsmetod som testar en konfiguration, utvärderar den och sedan testar en ny konfiguration baserat på resultaten av den föregående. På sikt "lär" sig metoden vad som gör en konfiguration bra respektive dålig och strävar på så sätt efter en optimal lösning.

Resultatet visar på att vår optimeringsmetod fungerar och att den successivt hittar allt bättre konfigurationer. Vår metod hittar däremot inte konfigurationer som är bättre än de som moderna kvantiseringsmetoder levererar, vilket visar att dessa metoder är svåra att överträffa. Vi bedömer att rymden av möjliga konfigurationer som optimeringsmetoden utforskar är för omfattande för att rymmas inom ramen för vårt examensarbete. Vi lämnar därför åt framtida arbeten att vidare experimentera med och optimera denna metod.