GPTVQ: THE BLESSING OF DIMENSIONALITY FOR LLM QUANTIZATION

Mart van Baalen [*1] Andrey Kuzmin [*1] Ivan Koryakovskiy [1] Cedric Bastoul [1] Peter Couperus [1] Eric Mahurin [1] Tijmen Blankevoort [1] Markus Nagel [1] Paul Whatmough [1]

ABSTRACT

Large language model (LLM) accuracy and token generation rate are fundamentally limited by both DRAM footprint and bandwidth, making them challenging to deploy on mobile devices. To improve the size-accuracy trade-off criticial to mobile LLMs, Vector Quantization (VQ) has recently been proposed. However, while previous VQ methods demonstrated footprint reduction, they have failed to demonstrate token rate gains over standard INT4 quantization on Nvidia GPUs, and do not even consider mobile devices. The reason for this is the large codebooks, which are too slow to index at inference time. In this work, we co-design our VQ representation, post-training quantization flow, and LLM software inference engine, to enable efficient inference on mobile devices. We propose a novel post-training quantization algorithm, GPTVQ, that quickly and accurately compresses a wide range of LLMs, specifically resulting in small per-block LUTs, which are fast to decode using existing CPU LUT instructions. Using a custom LLM software inference engine, we demonstrate VQ LLMs running on mobile CPU, and measure a simultaneous DRAM footprint reduction of 19% and token rate improvement of 10% compared to industry standard INT4, at little harm to accuracy, and outperforming llama.cpp. Finally, for task-specific scenarios, we demonstrate that combining GPTVQ base model with the orthogonal approach of LoRA adapters results in a significant improvement of accuracy over previous adapter-based methods.

1 INTRODUCTION

Large language models (LLMs) enable unprecedented improvements in usability on mobile devices, providing general AI assistance across a broad swathe of natural language processing use cases. They also form the backbone for multi-modal models that recognize and interpret images (Liu et al., 2023; Lin et al., 2023c), transcribe and analyze audio (Zhang et al., 2023), and even process video content (Lin et al., 2023a; Cheng et al., 2024), making them a central and indispensable tool in modern computing.

However, the sheer size of LLMs makes them challenging to deploy on mobile devices for two reasons, both pertaining to DRAM main memory constraints. Firstly, the required model footprint is prohibitive in mobile devices, limiting achievable accuracy. Typical mobile phones have around 8GB of total DRAM memory (Wikipedia, 2024), with the OS and active apps easily occupying more than half of this, typically leaving less than 4GB for an LLM. Secondly, DRAM bandwidth limits achievable token generation rates, since the autoregressive nature of LLMs requires loading every single weight once for each generated token. This is particularly severe in the common case of moderate context length of up to 8K–16K tokens (Kim et al., 2023; Hooper et al., 2024). *Therefore, reducing the stored model footprint is critical to relaxing both of these impediments.*

To directly address the compute-bound nature of LLM token generation, we explore how to trade a small increase in (surplus) compute for a commensurate decrease in valuable weight footprint and bandwidth. We propose to do this using Vector Quantization (VQ) (Stock et al., 2019; Tseng et al., 2024; Egiazarian et al., 2024; Malinovskii et al., 2024b), which is a SOTA approach that uses a non-uniform number system in multiple dimensions to aggressively reduce LLM footprint. Since we cannot compute on the VQ encoded data directly, we must first decode to a *native* data type. Hence, we use VQ only as a *storage* data type.

VQ provides an improvement in model size vs accuracy, which will ideally improve all of: footprint, bandwidth and token rate. However, although the token rate improvement is expected since the workload is heavily bandwidth bound, it is not guaranteed, due to the overhead for decoding VQ back to a native type before use. If carefully designed, this overhead is more than offset by the reduction in memory bandwidth, enabling either a larger number of parameters (higher accuracy) at the same DRAM footprint and token rate, or a higher token rate and smaller footprint for the

Snapdragon branded products are products of Qualcomm Technologies, Inc. and/or its subsidiaries.
[*] Equal contribution.
[1] Qualcomm AI Research (Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc).

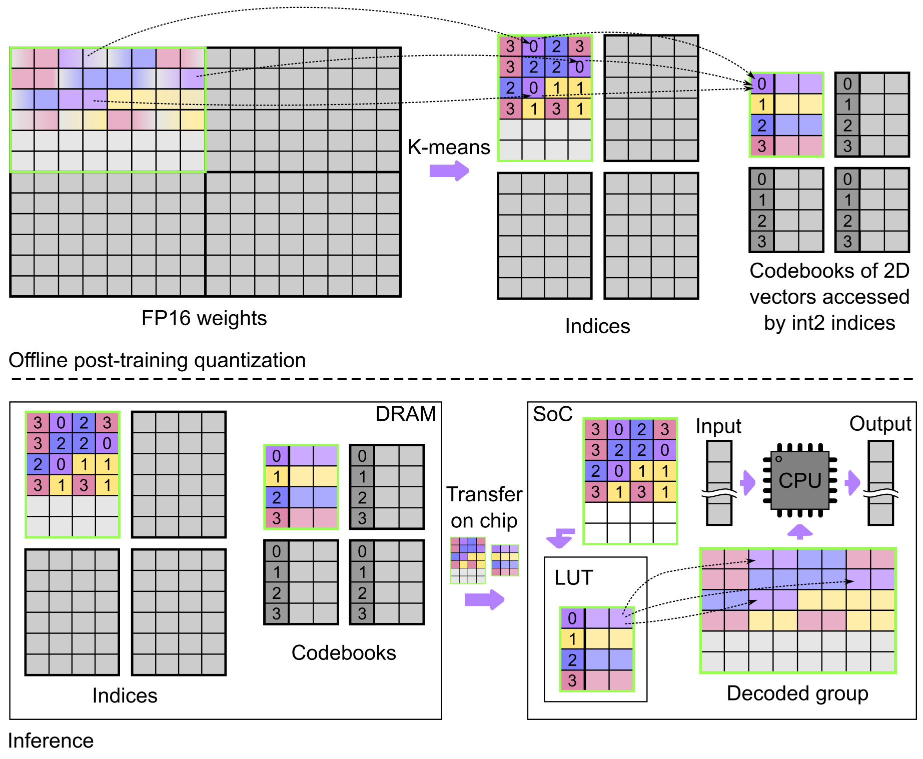

Figure 1. The proposed hardware-friendly representation and GPTVQ method. **Top:** During quantization, the FP16 weights are split into groups with their own small codebook. **Bottom:** During inference, the codebooks and indices are moved from DRAM to SoC independently from each other. The codebook is implemented as a lookup table (LUT) available on modern mobile CPUs.

same number of parameters. However, previous research on VQ targets cloud GPU platforms only, and uses very large codebooks, which are not efficient for implementation on mobile CPUs.

In fact, we find that VQ decoding is most efficiently implemented on mobile devices using existing hardware *lookup table instructions*, present on mobile CPUs and also on many NPUs and GPUs. These instructions typically map a 5- or 6-bit index to an 8-bit value. Critically, this means that very large lookup tables or high VQ dimensions, such as those used by AQLM (Egiazarian et al., 2024), require many calls to the lookup table instruction, leading to significant decoding latency. In this work, we demonstrate the potential of VQ footprint compression running on mobile CPU, by co-designing the VQ compression algorithm with the software implementation.

Figure 1 demonstrates our approach. During post-training quantization, we divide the FP16 weights into groups. For each group, a table of indices and a corresponding codebook is derived. During inference, the indices and codebooks are stored in DRAM, before being transferred to the CPU cache on the SoC, where they are decoded to a native data type and used in matrix multiplication. To minimize the quantization error due to our VQ representation, we introduce a novel post-training quantization method, *GPTVQ*. The quantization of each weight matrix is implemented as a single pass from left to right, which is highly efficient. The quantization error is compensated by updating the remaining unquantized weights. Finally, we implement a custom LLM inference software engine for mobile CPU, to demonstrate that our end-to-end VQ method can reduce footprint and increase token rate, without compromising accuracy.

The contributions of this work are summarized as follows:

- We describe an optimized VQ representation that is not only footprint efficient, but also fast to decode. This is achieved by co-designing the VQ parameters, including the bitwidths, number of dimensions and LUT size, with the mobile CPU software implementation, to efficiently leverage existing hardware ISA extensions for fast LUT decoding.
- We implement and benchmark a full LLM inference stack supporting VQ decompression on mobile CPU. Results demonstrate that VQ reduces DRAM footprint by 19% while increasing the token rate by 10%, compared to a 4-bit integer baseline.
- We propose a fast and accurate algorithm for post-training VQ compression (GPTVQ), which achieves

favorable size vs accuracy trade-offs on a wide range of LLMs, while having a practical offline run time of only 3 to 11 hours on a 70B parameter model.

- We also show that our VQ approach is complementary to the popular use of adapters with LLMs, which gives an additional opportunity to recover accuracy loss from aggressive quantization on mobile devices.

2 Background and Motivation

In this section we motivate the use of VQ as a storage data type. We will first establish notation for VQ and explain why VQ provides better representational accuracy than traditional uniform quantization. Then, we will discuss existing VQ methods and their drawbacks. Lastly, we discuss requirements of on-device implementation of VQ storage type decoding to a native compute data type, and how existing approaches aimed at cloud GPU environments fall short for mobile application.

2.1 Uniform, non-uniform, and vector quantization

A symmetric uniform quantizer approximates an original floating point vector $\mathbf{x} \in \mathbb{R}^D$ as $\mathbf{x} \approx s\mathbf{x}_{int}$, where each element in $\mathbf{x}_{int}$ is a b-bit integer value and s is a higher precision quantization scale, shared across the components of $\mathbf{x}$.

A more flexible quantization approach is non-uniform quantization, in which floating point numbers are discretized to arbitrary scalar centroids stored in a codebook $C : C = \{c_1, c_2, \ldots, c_k\}$. Each high precision value in $\mathbf{x}$ is then represented by the index j of a centroid c_j. Each index is stored in $\lceil \log_2 k \rceil$ bits. Even more flexible quantizer can be constructed using a higher-dimensionality for the centroids of C. In this case, each centroid in C encodes d values, e.g., pairs of values if $d = 2$, and each group of d values in $\mathbf{x}$ is represented by a single index into C_d, where C_d denotes a codebook with elements of dimensionality d (Gersho & Gray, 2012).

Increasing the dimensionality of the codebook via VQ, increases the flexibility of the quantization grid. Figure 2 *(top)* gives a visual representation of this. In this example, where we quantize each value in the original to a 3-bit representation, i.e., 6 bits for each pair of values, we can see that the number of points stays the same, i.e., $2^6 = 64$, but the distribution of the centroids with VQ can more closely match the underlying distribution, increasing the accuracy of the representation.

The representational accuracy increases the more the dimensionality of the codebook increases. Figure 2 *(bottom)* shows signal-to-quantization-noise ratio (SQNR) as the accuracy measure. SQNR is defined in the log scale

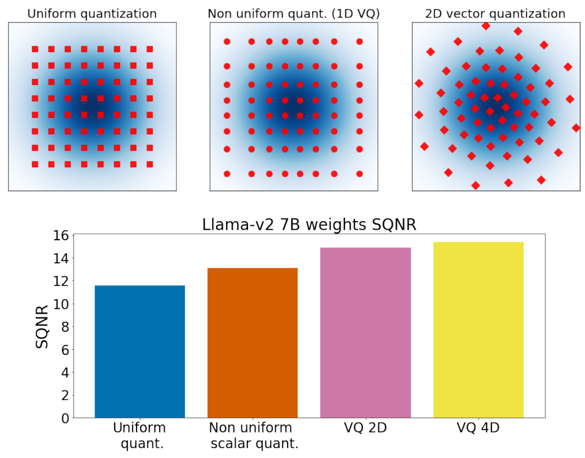

Figure 2. **Top:** Illustration on how vector quantization can fit better 2D normal data, compared to uniform and non-uniform grids. **Bottom:** SQNR increases with quantization dimensionality on Llama-v2 7B weights, due to additional flexibility in the quantization grid.

as: $\text{SQNR}_{dB} = 10 \log_{10} \left(\mathbb{E} \left[W^2 \right] / \mathbb{E} \left[(W - Q(W))^2 \right] \right)$, where $Q(\cdot)$ is the quantization function. We can see the improvement in representational accuracy of higher d for Llama-v2 7B weights. Note that, as d grows, so does the codebook size. For fair comparison, we ensure the codebook overhead is always equal to 0.25 bit per weight for each quantization method, i.e., improved SQNR is not caused trivially by using more bits in total for our representations.

2.2 Prior work on vector quantization

Due to its ability to represent data more flexibly at very low bit-widths, vector quantization has received increased attention recently. The AQLM method introduced in (Egiazarian et al., 2024) employs vector quantization to compress LLMs down to approximately 2 bits per weight, with significantly improved accuracy compared to uniformly quantized models at the same bit width. Their method consists of 3 stages. 1) A codebook initialization step, where a weight tensor is reshaped into a matrix with d columns and the codebook is initialized using k-Means; 2) 100 iterations of an EM-style phase, in which one epoch of a gradient-based codebook fine-tuning is followed by a beam search that updates codebook indices to minimize layer-wise reconstruction error; and 3) a full block fine-tuning step in which the entries of the codebooks for all layers in a decoder block are fine-tuned to minimize the output error of the block. AQLM uses vector dimensions $d = 8$. At 2 bit per dimension, this means that each codebook contains 2^{16} 8-bit vectors.

While AQLM shows good quantization accuracy, the EM-style phase and the following block fine-tuning are expensive to run. In our experiments on a single H100, Llama v2-7B quantization takes approximately 35 hours.

More recently, PV-tuning (Malinovskii et al., 2024b) introduces an end-to-end training method for non-uniform and VQ quantization. Similar to uniform quantization, gradients cannot be backpropagated through centroid indices. In uniform quantization, a method commonly used to circumvent this issue is the *straight-through estimator* (STE). With STE, during the backward pass, the gradient of the loss with respect to the quantized weights is passed 'straight-through' to the unquantized (shadow) weights without modification. The PV-tuning authors show that this approach does not work for VQ, as updating all weights simultaneously leads to too-large updates. Instead, to learn centroid indices, the authors introduce a trust ratio that restricts weight updates to a small subset. Using this method the authors show improved model accuracy compared to models quantized using AQLM.

2.3 VQ model deployment and limitations of existing methods

To avoid introducing extra latency in the decoding step from storage data type to compute data type, decoding needs to be extremely efficient and faster than the DRAM bandwidth. The most efficient way to decode VQ weights is by using the *lookup table instruction* (LUT) which is present in all modern mobile CPUs. The LUT instruction maps a 6-bit index to an 8-bit value. This means that, for 2D VQ, 2 LUT instructions must be called, one for each dimension. The 6-bit index implies that VQ codebooks should contain at most 64 entries. For this reason the setting used by recent VQ methods such as AQLM is not conducive to good on-device performance: Using, e.g., 16-bit indices (the 2.29 bpv in (Egiazarian et al., 2024)) precludes the use of the efficient LUT instruction, and instead requires the use of the less performant SVE gather instruction.

Armed with this knowledge, we design a VQ representation using fewer bits per index and lower dimensionality, and show that this can achieve model accuracy competitive with traditional INT4 quantization and other VQ approaches. Furthermore, we show that this setting can be implemented efficiently for inference on mobile CPU.

3 On-device implementation

In this section we describe our on-device implementation.

3.1 Mobile-friendly quantized tensor representation

The most common approach to LLM quantization on mobile CPU platforms is to use 4-bit integer for each element in a tensor with a scale factor shared among a group of elements. This approach is widely adopted, e.g., by the open source Llama.cpp[1] project. Our VQ implementation follows a similar scheme, with tensors split into groups of elements, each with a scale factor. In addition, we store for each block a lookup table that is used to decode VQ elements. Hence, each group of weight elements is stored as a tuple of 1) the VQ encoded element indices, 2) the associated LUT for decoding, and 3) a quantization scale factor. While the GPTVQ algorithm presented in the next section can be applied to any dimensionality or index bitwidth, for our practical implementation we choose a fixed configuration of 2D VQ, with 6-bit indices, i.e., 3 bits per weight. This allows our LLM inference software implementation to leverage the 6-bit to 8-bit LUT instruction.

3.2 LLM Inference Software Implementation

The inference software we use to benchmark our VQ approach is an in-house implementation of a highly parameterizable transformer architecture that supports major large language models. It is written in C with vector intrinsics for accelerating matrix multiply and similar kernels on mobile CPU using SIMD extensions and the like. Furthermore, it leverages the structural properties of transformers for efficient coarse-grain parallelization and high-level polyhedral compiler capabilities for fine-grain vectorization.

To support VQ in the inference engine, the 6-bit indices are packed tightly and stored in memory along with the lookup tables and quantization scales, organized to enable efficient vectorization. During inference, each block is decoded as follow: The tuple of block data is loaded from DRAM onto the SoC, and into the CPU cache, c.f. Figure 1. Here, the VQ decode kernel uses the native mobile CPU LUT instructions to efficiently perform lookups quickly, converting 6-bit index to signed 8-bit integer data. These integer values are then used in the downstream matrix-vector multiplications.

4 The GPTVQ algorithm

In this section, we introduce our GPTVQ algorithm, a novel method for efficient and accurate post-training vector-quantization of LLMs, which extends the GPTQ (Frantar et al., 2022) algorithm to VQ. Appendices A, F ,G, and H present extensions to GPTVQ, including Codebook SVD, Blockwise Data Normalization, an extended EM initialization algorithm, and a codebook update procedure.

Neural network quantization reduces model size as well as compute and energy requirements, but introduces quantization noise. A large body of literature exists with methods to alleviate the effects of quantization noise on model accuracy, see (Nagel et al., 2021; Gholami et al., 2022) for recent surveys. Post-training quantization (PTQ) approaches aim to mitigate the adverse effects of quantization noise

[1]https://github.com/ggerganov/llama.cpp

Algorithm 1 GPTVQ algorithm: Quantize a weight tensor $\mathbf{W} \in \mathbb{R}^{r\times c}$ given the inverse Hessian $\mathbf{H^{-1}}$, the block size B, VQ dimensionality d, the number of centroids k, and the group size l. To simplify the notation, we assume one group per column.

```
1: $N_b \leftarrow \frac{c}{B}$                          ▷ the number of blocks
2: $m \leftarrow \frac{l}{r}$                            ▷ the number of columns in a group
3: $\mathbf{Q} \leftarrow \mathbf{0}_{r,c}$
4: $\mathbf{E} \leftarrow \mathbf{0}_{r,c}$
5: $N_g \leftarrow \frac{rc}{l}$                         ▷ the number of groups/codebooks
6: $\mathbf{C}_i \leftarrow \mathbf{0}_{d,k}, i = 1, \ldots, N_g$
7: $\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^T$
8: for $i = 0, B, 2B, \ldots, N_bB$ do
9:     if i % m = 0 then
10:        $g \leftarrow \frac{i}{m}$                    ▷ the group index
11:        $\mathbf{C}_g \leftarrow \text{init\_codebook}\left[\mathbf{W}_{:,i:i+m-1}\right]$
12:    end if
13:    $\mathbf{Q}_{:,i:i+m-1} \leftarrow \text{QuantGroup}(\mathbf{W}_{:,i:i+m-1})$
14:    $\mathbf{W}_{:,i+B:} \leftarrow \mathbf{W}_{:,i+B:} - \mathbf{E}\cdot[\mathbf{H}^{-1}]_{i:i+B,i+B:}$
15: end for
```

on pre-trained networks, without having to resort to costly quantization-aware training (QAT). A popular and effective approach in PTQ, introduced by AdaRound (Nagel et al., 2020), is to modify weights to minimize a layer's output error as an approximation to the full network's loss:

$$\mathbb{E}\left[\mathcal{L}(\theta+\epsilon) - \mathcal{L}(\theta)\right] \approx \sum_{\ell} ||\mathbf{W}^{\ell}\mathbf{X}^{\ell} - \widehat{\mathbf{W}}^{\ell}\mathbf{X}^{\ell}||_F^2, \quad (1)$$

where $\mathbf{W}^{\ell}$ is the weight for layer ℓ, $\widehat{\mathbf{W}}^{\ell} = \mathbf{W}^{\ell} + \epsilon^{\ell}$ is the (quantized) approximation to this weight tensor, and $\mathbf{X}^{\ell}$ of shape $R\times N$ denotes the input data for layer ℓ from a calibration dataset, with N individual data points of dimensionality R along its columns.

Algorithm 2 QuantGroup: VQ quantization for a group of weights $\mathbf{W} \in \mathbb{R}^{r\times m}$ given the inverse Hessian $\mathbf{H^{-1}}$, the block size B, VQ dimensionality d

```
1: function QuantGroup($\mathbf{W}$)
2:     for $j = 0, d, 2d, \ldots, l$ do
3:         $P = j, \ldots, j+d-1$
4:         $\mathbf{Q}_{:,P} \leftarrow \text{VQ\_quant}\left[\mathbf{W}_{:,P}, \mathbf{C}_g\right]$
5:         $\mathbf{E}_{:,P} \leftarrow (\mathbf{W}_{:,P} - \mathbf{Q}_{:,P})\left[\mathbf{H}^{-1}\right]_P$
6:         $\mathbf{U} \leftarrow \sum_{p=0}^{d-1} \mathbf{E}_{:,j+p}[\mathbf{H}^{-1}]_{p,j+d-1:B}$
7:         $\mathbf{W}_{:,j+d-1:B} \leftarrow \mathbf{W}_{:,j+d-1:B} - \mathbf{U}$
8:     end for
9: end function
```

Preliminary: GPTQ GPTQ follows Optimal Brain Quantization (OBQ; (Frantar & Alistarh, 2022)), which uses the Hessian of Equation 1. This Hessian can be efficiently computed as $\mathbf{H}^{(\ell)} = \mathbf{X}^{(\ell)}\mathbf{X}^{(\ell)T}$. Like OBQ, GPTQ aims to minimize the Hessian-weighted error introduced by quantizing weights in $\mathbf{W}^{(\ell)}$:

$$E = \sum_q |E_q|_2^2; \quad E_q = \frac{(\mathbf{W}_{:,q} - \text{quant}(\mathbf{W}_{:,q}))^2}{\left[\mathbf{H}^{-1}\right]_{qq}}. \quad (2)$$

GPTQ extends OBQ in the following ways. First, GPTQ exploits the fact that $\mathbf{H}^{(\ell)}$ is shared over all rows of $\mathbf{W}^{(\ell)}$ by quantizing all weights in a column in parallel, from left to right. This obviates the need for independent Hessian updates for different rows. After quantizing a column q, all remaining (unquantized) columns $q' > q$ are modified with a Hessian-based update rule δ that absorbs the error introduced by quantizing column q on the layer's output:

$$\delta = -\frac{\mathbf{W}_{:,q} - \text{quant}(\mathbf{W}_{:,q})}{\left[\mathbf{H}^{-1}\right]_{qq}}\mathbf{H}_{:,(q+1):} \quad (3)$$

To reduce data transfer, GPTQ applies the update of Equation 3 only to a small block of B columns in which column q resides. Note that such blocks enable an efficient weight update implementation. To update the columns outside of block B, the error E_q in Equation 2 is accumulated while the columns in block B are processed, and are applied in one go to all columns outside of block B after all columns in block B are processed. Lastly, GPTQ uses a Cholesky decomposition for updating the inverse Hessian $\mathbf{H}^{-1}$, which introduces a more numerically stable alternative to the inverse Hessian row and column removal operations of OBQ.

Codebook Initialization Unlike traditional GPTQ, VQ requires a codebook for each group of weights. To initialize the codebook for a group of weights, we propose the following variant of the EM algorithm. Given the set of d-dimensional vectors $\mathbf{x}^{(i)}$, our goal is to find k centroid vectors $\mathbf{c}^{(m)}$ and the corresponding sets of indices I_m pointing at the centroid m. The objective is the following sum of weighted distance functions:

$$\min_{\mathbf{I},\mathbf{c}^{(0)},\ldots,(k)} \sum_{m=0}^{k}\sum_{i\in I_m}\left(\mathbf{x}^{(i)} - \mathbf{c}^{(m)}\right)^T \mathbf{D}^{(i)}\left(\mathbf{x}^{(i)} - \mathbf{c}^{(m)}\right), \quad (4)$$

where $\mathbf{D} = \text{diag}\left(1/[\mathbf{H}^{-1}]_{11}, \ldots, 1/[\mathbf{H}^{-1}]_{cc}\right)$ and $\mathbf{D}^{(i)}$ is a $d\times d$ subset of $\mathbf{D}$ corresponding to the data point $\mathbf{x}^i$. E.g. for 2D vector quantization, these matrices are share among pairs of columns. For the case of $\mathbf{D}^{(i)}$ equal to identity, the clustering method is equivalent to K-means. The objective can be minimized using E- and M-steps as follows.

E-step: keeping the values of the centroids fixed, find centroid assignment $\mathbf{c}^{(i)}$ for each unquantized d-dimensional vector $\mathbf{x}^{(i)}$ that minimizes the objective (4). Using this distance function assigns optimal centroids based on the data-aware loss.

$$\mathbf{c}^{(i)} = \arg\min_{m} \left(\mathbf{x}^{(i)} - \mathbf{c}^{(m)}\right)^{T} \mathbf{D}^{(i)} \left(\mathbf{x}^{(i)} - \mathbf{c}^{(m)}\right), \quad (5)$$

M-step: keeping the assignments found in the E-step fixed, find the centroid value $\mathbf{c}^{(m)}$ that minimizes

$$\mathbf{c}^{(m)} = \arg\min_{\mathbf{c}} \sum_{i \in I_m} \left(\mathbf{x}^{(i)} - \mathbf{c}\right)^{T} \mathbf{D}^{(i)} \left(\mathbf{x}^{(i)} - \mathbf{c}\right). \quad (6)$$

This objective is a quadratic form w.r.t $\mathbf{c}^{(m)}$. The optimal value is computed in a closed form as $\mathbf{c}^{(m)} = \left(\sum_{i \in I_m} \mathbf{D}^{(i)}\right)^{+} \left(\sum_{i \in I_m} \mathbf{D}^{(i)}\mathbf{x}^{(i)}\right)$, where $(\cdot)^{+}$ is a Moore–Penrose pseudoinverse. During the vector quantization operation on line 4 in Algorithm 2, we use the assignment step defined in Equation 5 as well. Practically, we find no performance difference between using the inverse Hessian diagonal, or the full d-dim inverse sub-Hessian.

GPTVQ To efficiently and accurately quantize pre-trained floating point LLMs to our VQ storage format, we generalize the GPTQ framework to non-uniform and vector quantization.

Following the GPTQ framework we perform quantization of the weight tensor in a greedy manner starting from the first column. The details of the method are given in Algorithm 1. For a VQ dimensionality d, we quantize d columns at a time. Although quantizing d elements of a single column instead of d columns would absorb the quantization error more effectively, we choose to quantize d columns at a time as this layout allows for a much more efficient on-device implementation.

When quantizing a d-dimensional vector, we find the centroid in the codebook for the current block that minimizes the objective in Equation 4. After quantizing d columns, we update the remaining weights using the update rule (3). We accumulate the update along d coordinates and apply it to the remaining weights as a single operation. To allow for a hardware-friendly representation, we use several codebooks per layer, each assigned to a *group* of weights (see Algorithm 1). We use group sizes of at most 256 columns, to ensure codebook initialization can capture the previous updates of (3). For example, a group of 2,048 weights is 8 rows by 256 columns.

Total bits per value As a measure of total model size, we compute *bits per value* (bpv), given by $\log_2(k)/d + kdb_c/l$, where k is the number of centroids, d is the VQ dimensionality, b_c is the codebook bit-width, and l is the group size, i.e., the number of weights sharing a codebook. We choose values for k s.t. $\log_2(k)$ is an integer.

5 Experiments and Results

In this section we compare on-device token generation rate of our VQ method to uniformly quantized models. Furthermore, we evaluate GPTVQ and compare the performance of vector quantization in 1, 2 and 4 dimensions against uniform quantization baseline methods.

5.1 On-device VQ inference evaluation and comparison

To investigate the effect of VQ quantized models on model DRAM footprint and token rate, we run our optimized VQ decoding kernel with our in-house implementation of a highly parameterizable transformer architecture, as described in Section 4. We benchmark the latency and DRAM footprint on a Snapdragon® X Elite platform, and further compare our implementation to the open source llama.cpp implementation baseline on the same platform. The mobile platform we used runs Windows, and we used Clang 18.1 with Polly enabled.

We measure end-to-end inference token rate for a Llama3-8B model using three different quantization scenarios: 1) llama.cpp Q4_0 INT4 quantization; 2) our implementation with INT4 g128; and 3) Our implementation with 2D VQ, at 3.125 effective bits per weight. The latter uses 3 bits per dimension, i.e. 6-bit indices, and a group size of 8,192.

Results of this experiment can be found in Table 1. Firstly, we note that our INT4 implementation baseline greatly improves token rate compared to llama.cpp. Secondly, we note that, within our optimized implementation, VQ provides a 10% increase in token rate compared to INT4 g128 quantization. Lastly, we note that our VQ model has significantly lower footprint than the INT4 quantized models.

In conclusion, when deployed on a mobile CPU, our VQ setting yields significantly lower footprint and significantly higher token rates than an INT4 model, even when compared to a highly optimized INT4 implementation.

5.2 GPTVQ evaluation

Models We use Llama-1 (Touvron et al., 2023a), Llama-2 (Touvron et al., 2023b), and Llama-3 as well as Mistral-7B-v0.1 (Jiang et al., 2023) and Mixtral-MoE-8x7B-v0.1 (Jiang et al., 2024). Additionally, we run a single ablation on BLOOM-560M (Workshop et al., 2022).

Datasets Following Shao et al. (2023), we use 128 sequences of 2048 tokens from the WikiText2 (Merity et al., 2016) training set as calibration data for all experiments. We evaluate our models on token perplexity for the WikiText2 validation set for a sequence length 2048, as well as zero-shot language tasks: PIQA (Bisk et al., 2020), ARC-

Table 1. **LLM token generation on mobile device for Llama-v3-8B.** The top row shows model footprint and token rate numbers for a model deployed using llama.cpp, the two bottom rows show deployment using our implementation.

Bits per value	Format	Block size	Engine	Footprint [GB] $\downarrow$	Throughput [tok/s] $\uparrow$
4.5	INT4	32	llama.cpp	4.64	$17.95^{\pm 1.01}$
4.125	INT4	128	Ours	4.33	$23.81^{\pm 0.27}$
3.125	VQ 2D	8,192	Ours	**3.52** (-19%)	$\mathbf{26.15}^{\pm \mathbf{0.31}}$ (+10%)

easy/-challenge (Clark et al., 2018), BoolQ (Clark et al., 2019), HellaSwag (Zellers et al., 2019), and WinoGrande (Sakaguchi et al., 2021). For Llama3, following (Huang et al., 2024), we omit BoolQ from the zero-shot average to allow fair comparison to the zero-shot results in (Huang et al., 2024). For all evaluation tasks except WikiText2 perplexity we use the LLM-evaluation-harness (Gao et al., 2023).

Baselines We compare GPTVQ to various uniform quantization methods with different group sizes, at the same overall bits-per-value (bpv). We include Round-to-Nearest (RTN) and several recent state-of-the-art PTQ approaches for LLMs: GPTQ (Frantar et al., 2022), AWQ (Lin et al., 2023b), and OmniQuant (Shao et al., 2023). We take AWQ and OmniQuant baseline numbers from (Shao et al., 2023), all Llama3 baseline numbers from (Huang et al., 2024), and generate all other baseline numbers ourselves. In Section 5.2 and Appendix E.1 we provide a more detailed comparison to other VQ approaches, most notably AQLM (Egiazarian et al., 2024), recent work that applies VQ to LLMs in a different manner.

Codebook overhead For a given bits per index b and VQ dimensionality d, we set group size l to reach an overhead of 0.125 bits per value for all values of b, and additionally consider an overhead 0.25 bits per value for $b = 2$. These are chosen to match the overhead incurred by a 16-bit quantization scale for the commonly used group size of 128 (e.g., (Frantar et al., 2022)) and the group size of 64 used by (Shao et al., 2023).

Comparison to scalar quantization Table 2 summarizes results for GPTVQ, where we report WikiText 2 perplexity and an average over zero-shot task scores for the PIQA, BoolQ, ARC-easy, ARC-challenge, HellaSwag and WinoGrande tasks. We include all Llama-v2 models, Mistral-7B-v0.1 and Mixtral-8x7B-v0.1. More results are in appendix D: Table 10 and Table 11 contain individual scores for the zero-shot tasks, Table 8 contains WikiText2 perplexity for all Llama-v1 models, and Table 9 shows perplexity on 4 bit quantization. Full VQ configurations can be found in Table 7.

These tables show that non-uniform quantization using GPTVQ generally yields improved results over uniform PTQ methods. This gap becomes especially large at low bitwidths and for very large models. For example, comparing GPTVQ 2D on Llamav2-70B to OmniQuant W2@g128, we see an improvement of nearly 2 perplexity points. Furthermore, in nearly all cases, 2D VQ outperforms 1D VQ, while 4D VQ shows even more significant improvements.

Comparison to other VQ methods Table 3 compares the results of GPTVQ to AQLM and QuIP# (Tseng et al., 2024). We use the same calibration data for AQLM and GPTVQ (SlimPajama, 4096 samples $\times$ 2048 tokens). Furthermore, we use the the original AQLM source code and closely follow the AQLM evaluation protocol. In particular, we use sequences of 4096 tokens from the WikiText2 test dataset, and average zero-shot accuracy among five LLM-evaluation-harness tasks (WinoGrande, PiQA, HellaSwag, ArcE, ArcC).

In terms of size vs accuracy, GPTVQ is on par with AQLM when full block fine-tuning is not used. Extra block fine-tuning allows AQLM to achieve better performance but increases the overall compression time significantly. Note that GPTVQ was specifically designed for mobile CPU use case. For this reason, we use smaller codebooks and a smaller vector dimension, even though this may lead to slightly worse perplexity and accuracy compared to the 8-dimensional case of AQLM and QuIP#, c.f. Figure 2. Further details and comparison can be found in Appendix E.1.

5.3 4 bit codebooks

Reducing the bit width of the codebooks to 4 bit can bring further benefits to VQ implementations: 1) 2 values can be decoded with 1 LUT instruction, 2) codebook overhead is reduced, and 3) decoded tensors take up less space on on-chip cache memory. However, reducing the codebook bit width potentially lowers accuracy. To mitigate the accuracy degradation we decode to groupwise INT4, with groups of 128 values. We refer to these groups as *codebook quantization* groups, to distinguish them from the groups used in GPTVQ, which we denote *GPTVQ groups*. We then modify the GPTVQ algorithm in the following ways: 1) First, for each group of weights $\mathbf{w}_g$ we find the scale s_g that minimizes $|(\mathbf{w}_g - Q(\mathbf{w}_g, s, b))\mathbf{h}_b|_2^2$, where $Q(\mathbf{w}_g, s, b))$

Table 2. **Weight-only quantization results of Llama-v2/v3, Mistral, and Mixtral-MoE Models**. We report WikiText2 perplexity and average zero-shot accuracy; Models marked L2 denote Llama-v2, L3 denote Llama-v3, M denotes Mistral, and 8x7B denotes Mixtral-MoE 8x7B. Numbers marked in bold are SOTA or surpass it, numbers underlined are on par with or outperform at least one VQ variant. * Following (Huang et al., 2024), Llama3-8B zeroshot average omits BoolQ.

		WikiText2 perplexity ↓							Zeroshot avg acc. ↑				
		L2-7B	L2-13B	L2-70B	L3-8B	L3-70B	M-7B	8x7B	L2-7B	L2-13B	L3-8B*	M-7B	8x7B
FP16		5.47	4.88	3.31	6.1	2.9	5.25	3.84	70.5	73.2	68.6	75.7	75.9
2.125 W2 g128	RTN	4e3	122	27.3	2e3	5e5	1e3	4e3	36.9	42.1	36.0	37.8	38.3
	GPTQ	36.8	28.1	6.74	2e2	11.9	15.7	14.1	41.4	46.6	36.2	41.9	44.5
	AWQ	2e5	1e5	-	2e6	2e6	-	-	-	-	-	-	-
	OQ	<u>11.1</u>	8.26	6.55	-	-	-	-	-	-	-	-	-
	Ours 1D	<u>12.2</u>	**7.40**	**5.03**	**15.9**	**9.37**	**14.0**	**8.37**	**47.8**	**61.8**	**41.1**	**42.8**	**54.9**
	Ours 2D	**7.77**	**6.52**	**4.72**	**11.3**	**7.37**	**7.53**	**5.92**	**58.6**	**64.5**	**53.9**	**64.5**	**64.4**
	Ours 4D	**7.18**	**6.07**	**4.44**	**9.94**	**6.59**	**6.89**	**5.28**	**60.5**	**65.7**	**57.3**	**65.7**	**68.7**
2.25 W2 g64	RTN	432	26.2	10.3	-	-	71.5	156	42.4	46.4	-	44.8	46.9
	GPTQ	20.9	22.4	NAN	-	-	14.2	10.1	47.5	54.2	-	51.8	48.8
	AWQ	2e5	1e5	-	-	-	-	-	-	-	-	-	-
	OQ	<u>9.62</u>	7.56	6.11	-	-	-	-	-	-	-	-	-
	Ours 1D	10.1	**6.99**	**4.85**	**14.1**	**8.31**	**9.69**	**7.75**	**52.8**	**63.3**	**57.3**	**56.3**	**57.4**
	Ours 2D	**7.61**	**6.41**	**4.58**	**10.8**	**6.83**	**7.24**	**5.58**	**61.5**	**64.8**	**60.3**	**65.3**	**65.7**
	Ours 4D	**6.99**	**5.98**	**4.36**	**9.59**	**6.21**	**6.66**	**5.16**	**62.9**	**67.5**	**62.3**	**68.2**	**69.3**
3.125 W3 g128	RTN	6.66	5.51	3.97	27.9	11.8	6.15	5.18	67.3	70.8	40.2	71.8	72.4
	GPTQ	6.29	5.42	3.85	8.2	5.2	5.83	4.71	66.2	<u>71.4</u>	61.7	<u>72.2</u>	72.7
	AWQ	6.24	5.32	-	8.2	4.8	-	-	-	-	-	-	-
	OQ	6.03	5.28	3.78	-	-	-	-	-	-	-	-	-
	Ours 1D	**5.95**	**5.19**	**3.64**	**7.29**	**4.29**	**5.79**	**4.59**	**66.9**	71.4	**65.7**	71.0	**73.5**
	Ours 2D	**5.83**	**5.12**	**3.58**	**7.00**	**4.04**	**5.51**	**4.27**	**68.3**	**71.2**	**66.1**	**73.9**	**75.1**

Table 3. **Comparison with existing SOTA VQ methods.** No BFT option in AQLM denotes no block fine-tuning. This setting is the most comparable to Our GPTVQ algorithm. For fair compression comparison, we used the original AQLM code adapting it to our GPTVQ setup as close as possible, and also evluation protocol which is slightly different from Table 2.

Model	Algorithm	bpv	Format	WikiText2 perplexity ↓	Zero-shot avg. acc. ↑	Compression time, h ↓
L2-7B	QuIP#	2.02	8D	8.22	52.2	-
	AQLM (no BFT)	2.29	8D	7.49	58.9	18.3
	AQLM	2.29	8D	**6.65**	**60.3**	35.2
	Ours	2.25	4D	7.11	59.2	**2.5**
L3-8B	AQLM (no BFT)	2.27	8D	9.86	60.9	19.9
	AQLM	2.29	8D	**8.71**	**62.7**	40.4
	Ours	2.25	4D	9.40	60.0	**2.8**
M-7B	QuIP#	2.01	8D	6.02	62.2	-
	AQLM (no BFT)	2.27	8D	6.87	64.0	19.4
	AQLM	2.29	8D	**5.77**	**65.4**	76.0
	Ours	2.25	4D	6.68	62.7	**2.8**

denotes b-bit quantization of $\mathbf{w}_g$, using scale s, and $\mathbf{h}_g$ is the Hessian diagonal corresponding to block $\mathbf{w}_g$. 2) Then, EM codebook initialization is applied to the GPTVQ group of weights. These groups are generally larger than the codebook quantization groupsize of 128. E.g., for $d = 2, b = 3$, a group size of 8192 is used. Before EM, each quantzation group g (i.e., row) in a block is scaled using the scale s_g found in the previous step, but not clipped or rounded. This ensures that the codebook found is already on the INT4 scale, but that no further error is introduced. I.e., each row is scaled as $\mathbf{w}_g^s = \frac{1}{s_g}\mathbf{w}_g$, where the scaled rows $\mathbf{w}_g^s$ are used in EM. 3) After EM, the codebook is clipped and rounded, but not scaled, since it is already scaled before initialization. 4) Lastly, during GPTVQ, each codebook quantization group is scaled using the scale found in step 1 before the nearest centroid is found: Equation 4 is modified as $\mathbf{c}^{(i)} = \arg\min_m \left(\frac{1}{s^{(i)}}\mathbf{x}^{(i)} - \mathbf{c}^{(m)}\right)^T \mathbf{D}^{(i)} \left(\frac{1}{s^{(i)}}\mathbf{x}^{(i)} - \mathbf{c}^{(m)}\right)$, where $s^{(i)}$ is the scale corresponding to vector $\mathbf{x}^{(i)}$.

In Table 4 we show the effect of quantizing codebooks to

Table 4. **INT4 codebooks.** Perplexity for on WikiText2 for models with 2D, 3B VQ, blocksize 4096, with codebook entries quantized to INT4. INT4 CB column also shows PPL increase relative to INT8 CB. *For Mistral-7B better results were achieved by avoiding block quantization and instead quantizing per VQ block.

Model	INT4 b128	INT8 CB	INT4 CB
L2-7B	5.72	5.95	5.99 (+0.05)
L3-8B	6.73	7.00	7.29 (+0.29)
M-7B	5.34	5.51	5.77* (+0.26)

4 bits. For Mistral-7B, a slightly modified version of the procedure enumerated above is used, in which the quantization group is set equal to the GPTVQ group. For all models considered, quantizing the codebook to INT4 yields very minor increase in perplexity.

5.4 Combination of GPTVQ with LoRA adapters

Table 5 shows the results of fine-tuning LoRA adapters on top of GPTVQ representation. We do not present the WikiText2 perplexity from the LoftQ paper because it was calculated using a different protocol than the one currently adopted in quantization literature (Shao et al., 2023; Egiazarian et al., 2024). First, we observe that LoRA adapters not only help to improve WikiText2 perplexity and GSM8k (Cobbe et al., 2021) accuracy, but also helps to recover performance from aggressive quantization. Second, we note that the GPTVQ and LoRA combination achieves a noteworthy improvement over previous methods making it a top choice for on device LLM serving with adapters, e.g., (Gunter et al., 2024).

6 Related work

Vector quantization A number of works propose vector quantization of CNN weights (Gong et al., 2014; Martinez et al., 2021; Fan et al., 2020; Stock et al., 2019; Wu et al., 2016; Martinez et al., 2021; Cho et al., 2021). The most common approach is to reshape the weights of convolutional or fully connected layers into a matrix, and then apply K-means clustering directly on the columns. Typically, the clustering is applied on scalars or vectors of dimension 4 or higher. Some of these works consider data-aware optimization of the quantized weights. Most often, a variant of the EM algorithm is used in order to update centroids and indices (Stock et al., 2019; Gong et al., 2014). An alternative approach is using a differentiable K-means formulation, which enables fine-tuning using SGD with the original loss function in order to recover the network accuracy (Cho et al., 2021; Fan et al., 2020; Tang et al., 2023). Finally, most recent SOTA works (Tseng et al., 2024; Egiazarian et al., 2024; Malinovskii et al., 2024b) apply layer-wise, block-wise and even end-to-end fine-tuning on a calibration dataset to recover accuracy loss.

LLM quantization Applying DNN quantization approaches to recent LLMs often poses significant computational challenges. Therefore, even uniform post-training quantization methods must be optimized for scalability (Frantar et al., 2022). Since vector quantization approaches have even higher computational complexity, applying them to LLM weights compression may be expensive. The most similar to our work is a method (Deng et al., 2024), which uses gradient-based layer sensitivities to update the codebooks and a reduced complexity LoRA-based approach (Hu et al., 2021) to partially recover the accuracy.

Hessian-based compression methods Several classical works suggest second-order approximation of the neural network loss function for accurate unstructured pruning (LeCun et al., 1989; Hassibi et al., 1993). A more recent line of work extends this family of methods to PTQ (Singh & Alistarh, 2020; Frantar & Alistarh, 2022; Frantar et al., 2022).

7 Conclusions

In this work, we have shown that vector quantization, when properly designed, can yield significant positive impact on token generation speed, DRAM footprint, and perplexity on a mobile CPU. Furthermore, we have shown that VQ in one or more dimensions progressively improves quantized large language model accuracy. By co-designing the VQ representation with the mobile CPU implementation, we created a fast decode kernel, leveraging existing hardware LUT instructions, and demonstrated an increase in decode token rate. Lastly, we introduced GPTVQ, a fast method for post-training quantization of LLMs that achieves competitive model size vs accuracy trade-offs on a wide range of LLMs and zero-shot tasks, using mobile friendly VQ configurations.

References

Arthur, D. and Vassilvitskii, S. K-means++ the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 1027–1035, 2007.

Bishop, C. M. Pattern recognition and machine learning. *Springer google schola*, 2:1122–1128, 2006.

Bisk, Y., Zellers, R., Gao, J., Choi, Y., et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 7432–7439, 2020.

Cheng, Z., Leng, S., Zhang, H., Xin, Y., Li, X., Chen, G., Zhu, Y., Zhang, W., Luo, Z., Zhao, D., and Bing,

Table 5. **Results of fine-tuning LoRA adapters.** We fine-tuned LoRA adapters on WikiText2 train split for WikiText2 task, and on GSM8k train split for GSM8k task. Subsequent evaluation was done on the corresponding test splits. Frozen adapter is taken from LoRA with FP16 base model (second row) and applied to the vector-quantized base models. We added 0.127 bpv overhead to LoftQ (Li et al., 2024) model mentioned in QLoRA (Dettmers et al., 2024). N.a. indicates that the training diverged.

				WikiText2 perplexity ↓			GSM8k accuracy ↑		
Method	Base bpv	Adapter	Format	L2-7B	L3-8B	M-7B	L2-7B	L3-8B	M-7B
FP	16	-	-	5.47	6.14	5.25	2.4	6.5	16.4
LoRA	16	Trained	-	4.91	5.82	5.19	40.3	62.3	56.5
QLoRA	2.127	Trained	NF2	-	-	-	n.a.	-	-
LoftQ	2.127	Trained	NF2	-	-	-	20.9	-	-
Ours	2.125	Frozen	VQ 2D	7.18	10.73	7.25	15.5	19.3	33.4
Ours	2.125	Trained	VQ 2D	**6.04**	**8.52**	**6.19**	**33.4**	**50.0**	**50.2**
Ours	2.125	Frozen	VQ 4D	6.53	9.65	6.55	18.2	29.9	39.6
Ours	2.125	Trained	VQ 4D	**5.83**	**8.07**	**6.00**	**32.5**	**51.0**	**51.8**
Ours	2.25	Frozen	VQ 2D	6.99	10.23	6.97	17.1	24.8	35.4
Ours	2.25	Trained	VQ 2D	**5.97**	**8.32**	**6.13**	**34.8**	**50.2**	**49.3**
Ours	2.25	Frozen	VQ 4D	6.27	9.13	6.36	19.9	32.2	42.8
Ours	2.25	Trained	VQ 4D	**5.77**	**7.94**	**5.94**	**35.0**	**52.5**	**50.4**
QLoRA	3.127	Trained	NF2/NF4	-	-	-	32.1	-	-
LoftQ	3.127	Trained	NF2/NF4	-	-	-	32.9	-	-
Ours	3.125	Frozen	VQ 2D	5.25	6.71	5.46	34.8	53.9	52.3
Ours	3.125	Trained	VQ 2D	**5.18**	**6.62**	**5.42**	**37.2**	**55.4**	**57.0**

L. Videollama 2: Advancing spatial-temporal modeling and audio understanding in video-llms. *arXiv preprint arXiv:2406.07476*, 2024.

Cho, M., Vahid, K. A., Adya, S., and Rastegari, M. Dkm: Differentiable k-means clustering layer for neural network compression. *arXiv preprint arXiv:2108.12659*, 2021.

Clark, C., Lee, K., Chang, M.-W., Kwiatkowski, T., Collins, M., and Toutanova, K. Boolq: Exploring the surprising difficulty of natural yes/no questions. In *NAACL*, 2019.

Clark, P., Cowhey, I., Etzioni, O., Khot, T., Sabharwal, A., Schoenick, C., and Tafjord, O. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv:1803.05457v1*, 2018.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Deng, J., Li, S., Wang, C., Gu, H., Shen, H., and Huang, K. LLM-codebook for extreme compression of large language models, 2024.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

Egiazarian, V., Panferov, A., Kuznedelev, D., Frantar, E., Babenko, A., and Alistarh, D. Extreme compression of large language models via additive quantization. *arXiv preprint arXiv:2401.06118*, 2024.

Fan, A., Stock, P., Graham, B., Grave, E., Gribonval, R., Jegou, H., and Joulin, A. Training with quantization noise for extreme model compression. *arXiv preprint arXiv:2004.07320*, 2020.

Frantar, E. and Alistarh, D. Optimal brain compression: A framework for accurate post-training quantization and pruning. *Advances in Neural Information Processing Systems*, 35:4475–4488, 2022.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Gao, L., Tow, J., Abbasi, B., Biderman, S., Black, S., DiPofi, A., Foster, C., Golding, L., Hsu, J., Le Noac'h, A., Li, H., McDonell, K., Muennighoff, N., Ociepa, C., Phang, J., Reynolds, L., Schoelkopf, H., Skowron, A., Sutawika, L., Tang, E., Thite, A., Wang, B., Wang, K., and Zou, A. A framework for few-shot language model evaluation, 12 2023. URL `https://zenodo.org/records/10256836`.

Gersho, A. and Gray, R. M. *Vector quantization and signal compression*, volume 159. Springer Science & Business Media, 2012.

Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*, pp. 291–326. Chapman and Hall/CRC, 2022.

Gong, Y., Liu, L., Yang, M., and Bourdev, L. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

Gunter, T., Wang, Z., Wang, C., Pang, R., Narayanan, A., Zhang, A., Zhang, B., Chen, C., Chiu, C.-C., Qiu, D., Gopinath, D., Yap, D. A., Yin, D., Nan, F., Weers, F., Yin, G., Huang, H., Wang, J., Lu, J., Peebles, J., Ye, K., Lee, M., Du, N., Chen, Q., Keunebroek, Q., Wiseman, S., Evans, S., Lei, T., Rathod, V., Kong, X., Du, X., Li, Y., Wang, Y., Gao, Y., Ahmed, Z., Xu, Z., Lu, Z., Rashid, A., Jose, A. M., Doane, A., Bencomo, A., Vanderby, A., Hansen, A., Jain, A., Anupama, A. M., Kamal, A., Wu, B., Brum, C., Maalouf, C., Erdenebileg, C., Dulhanty, C., Moritz, D., Kang, D., Jimenez, E., Ladd, E., Shi, F., Bai, F., Chu, F., Hohman, F., Kotek, H., Coleman, H. G., Li, J., Bigham, J., Cao, J., Lai, J., Cheung, J., Shan, J., Zhou, J., Li, J., Qin, J., Singh, K., Vega, K., Zou, K., Heckman, L., Gardiner, L., Bowler, M., Cordell, M., Cao, M., Hay, N., Shahdadpuri, N., Godwin, O., Dighe, P., Rachapudi, P., Tantawi, R., Frigg, R., Davarnia, S., Shah, S., Guha, S., Sirovica, S., Ma, S., Ma, S., Wang, S., Kim, S., Jayaram, S., Shankar, V., Paidi, V., Kumar, V., Wang, X., Zheng, X., Cheng, W., Shrager, Y., Ye, Y., Tanaka, Y., Guo, Y., Meng, Y., Luo, Z. T., Ouyang, Z., Aygar, A., Wan, A., Walkingshaw, A., Narayanan, A., Lin, A., Farooq, A., Ramerth, B., Reed, C., Bartels, C., Chaney, C., Riazati, D., Yang, E. L., Feldman, E., Hochstrasser, G., Seguin, G., Belousova, I., Pelemans, J., Yang, K., Vahid, K. A., Cao, L., Najibi, M., Zuliani, M., Horton, M., Cho, M., Bhendawade, N., Dong, P., Maj, P., Agrawal, P., Shan, Q., Fu, Q., Poston, R., Xu, S., Liu, S., Rao, S., Heeramun, T., Merth, T., Rayala, U., Cui, V., Sridhar, V. R., Zhang, W., Zhang, W., Wu, W., Zhou, X., Liu, X., Zhao, Y., Xia, Y., Ren, Z., and Ren, Z. Apple intelligence foundation language models, 2024.

Hassibi, B., Stork, D. G., and Wolff, G. J. Optimal brain surgeon and general network pruning. In *IEEE international conference on neural networks*, pp. 293–299. IEEE, 1993.

Hooper, C., Kim, S., Mohammadzadeh, H., Mahoney, M. W., Shao, Y. S., Keutzer, K., and Gholami, A. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Huang, W., Ma, X., Qin, H., Zheng, X., Lv, C., Chen, H., Luo, J., Qi, X., Liu, X., and Magno, M. How good are low-bit quantized Llama3 models? An empirical study. *arXiv preprint arXiv:2404.14047*, 2024.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. d. l., Hanna, E. B., Bressand, F., et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

Kim, S., Hooper, C., Gholami, A., Dong, Z., Li, X., Shen, S., Mahoney, M. W., and Keutzer, K. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.

LeCun, Y., Denker, J., and Solla, S. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

Li, Y., Yu, Y., Liang, C., He, P., Karampatziakis, N., Chen, W., and Zhao, T. Loftq: Lora-fine-tuning-aware quantization for large language models. In *International Conference on Learning Representations*, 2024.

Lin, B., Zhu, B., Ye, Y., Ning, M., Jin, P., and Yuan, L. Video-llava: Learning united visual representation by alignment before projection. *arXiv preprint arXiv:2311.10122*, 2023a.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., and Han, S. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023b.

Lin, J., Yin, H., Ping, W., Lu, Y., Molchanov, P., Tao, A., Mao, H., Kautz, J., Shoeybi, M., and Han, S. Vila: On pre-training for visual language models, 2023c.

Liu, H., Li, C., Wu, Q., and Lee, Y. J. Visual instruction tuning. In *NeurIPS*, 2023.

Malinovskii, V., Mazur, D., Ilin, I., Kuznedelev, D., Burlachenko, K., Yi, K., Alistarh, D., and Richtarik, P. Pv-tuning: Beyond straight-through estimation for extreme llm compression. *Advances in Neural Information Processing Systems*, 37:5074–5121, 2024a.

Malinovskii, V., Mazur, D., Ilin, I., Kuznedelev, D., Burlachenko, K., Yi, K., Alistarh, D., and Richtarik, P. Pv-tuning: Beyond straight-through estimation for extreme llm compression, 2024b.

Martinez, J., Shewakramani, J., Liu, T. W., Bârsan, I. A., Zeng, W., and Urtasun, R. Permute, quantize, and fine-tune: Efficient compression of neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 15699–15708, 2021.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models, 2016.

Nagel, M., Amjad, R. A., Van Baalen, M., Louizos, C., and Blankevoort, T. Up or down? adaptive rounding for post-training quantization. In *International Conference on Machine Learning*, pp. 7197–7206. PMLR, 2020.

Nagel, M., Fournarakis, M., Amjad, R. A., Bondarenko, Y., Van Baalen, M., and Blankevoort, T. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.

Sakaguchi, K., Bras, R. L., Bhagavatula, C., and Choi, Y. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Shao, W., Chen, M., Zhang, Z., Xu, P., Zhao, L., Li, Z., Zhang, K., Gao, P., Qiao, Y., and Luo, P. Omniquant: Omnidirectionally calibrated quantization for large language models. *arXiv preprint arXiv:2308.13137*, 2023.

Singh, S. P. and Alistarh, D. Woodfisher: Efficient second-order approximation for neural network compression. *Advances in Neural Information Processing Systems*, 33: 18098–18109, 2020.

Soboleva, D., Al-Khateeb, F., Myers, R., Steeves, J. R., Hestness, J., and Dey, N. SlimPajama: A 627B token cleaned and deduplicated version of RedPajama, 2023.

Stock, P., Joulin, A., Gribonval, R., Graham, B., and Jégou, H. And the bit goes down: Revisiting the quantization of neural networks. *arXiv preprint arXiv:1907.05686*, 2019.

Tang, X., Wang, Y., Cao, T., Zhang, L. L., Chen, Q., Cai, D., Liu, Y., and Yang, M. Lut-nn: Empower efficient neural network inference with centroid learning and table lookup. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pp. 1–15, 2023.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

Tseng, A., Chee, J., Sun, Q., Kuleshov, V., and Sa, C. D. QuIP#: Even better LLM quantization with hadamard incoherence and lattice codebooks. In *Forty-first International Conference on Machine Learning*, 2024.

Wikipedia. iPhone 16, 2024. URL `https://en.wikipedia.org/wiki/IPhone_16`.

Workshop, B., Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ilić, S., Hesslow, D., Castagné, R., Luccioni, A. S., Yvon, F., et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

Wu, J., Leng, C., Wang, Y., Hu, Q., and Cheng, J. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4820–4828, 2016.

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., and Choi, Y. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

Zhang, D., Li, S., Zhang, X., Zhan, J., Wang, P., Zhou, Y., and Qiu, X. Speechgpt: Empowering large language models with intrinsic cross-modal conversational abilities, 2023.

A GPTVQ Algorithm details

Codebook update After the procedure in Algorithm 1 is complete, we found that the output reconstruction error can be further reduced through a *codebook update*. Recall that, in line 4 of Algorithm 2, **Q** is incrementally constructed from the elements of **C**. Since this construction constitutes a lookup of values in **C**, the layer-wise objective can still be minimized w.r.t **C**. The objective is a quadratic program and is convex:

$$\min_{\mathbf{C}_0,\ldots,\mathbf{C}_N} ||\mathbf{WX} - \mathbf{QX}||_F^2, \tag{7}$$

where $\mathbf{Q}(\mathbf{C}_0, \ldots, \mathbf{C}_N)$ is a look-up operation, reconstructing the quantized weights from the centroids. While this objective can be minimized in a closed form, we find that the PyTorch implementation using gradient descent is considerably faster while the solutions are equally good. The gradient of **Q** w.r.t. **C** can be defined simply, as constructing Q only involves a look-up operation. In each GD step, the values in **C** are updated, and **Q** is reconstructed using the new values in **C**, keeping the indices fixed.

B Further on-device results

To investigate the effect of VQ quantized models on model DRAM footprint and latency, we implemented optimized kernels for both a popular mobile CPU architecture, and Nvidia® GeForce RTX 3080 GPU.

The CPU kernel employs the table lookup (`TBL`) instruction to translate an index of (at most) 5 bits to an 8 bit integer, with two `TBL` instructions chained for 2D VQ. On GPU, we use native CUDA vector types to load and unload data quickly from GPU memory into the registers and back, such as `char4`/`uchar4`, and custom agglomerations of those, up to `char128`. The code for these kernels will be made available in the future.

We measure the time to transfer and unpack/decode the weights of a Llamav2-7B `gate_proj` layer (11008×4096), for VQ and to uniformly quantized data, and also FP16 on GPU. Furthermore, we integrate our CPU kernel with a matmul operation for an end-to-end token generation experiment on Llamav2-7B quantized using 1D VQ.

Table 6 shows that for both data transfer and token generation, VQ can achieve significant footprint reductions, with strictly positive latency impact on CPU, and negligible to positive latency impact on Nvidia® GPU.

Table 6. **Measured VQ data transfer/decoding, and LLM token generation on mobile device.** Exp: experiment, Data Transfer (T) or Token Generation (G). Ptfm: platform, Mobile CPU or NVIDIA® GPU. Format: either Uniform or VQ. Rel. FP: relative footprint. Rel. lat: relative latency.

Exp	Ptfm	bpv	Format	d	Rel. FP ↓	Rel. lat. ↓
T	CPU	4	Unif	1D	1.00×	1.00×
T	CPU	8	Unif	1D	2.00×	1.93×
T	CPU	3	VQ	2D	0.75×	0.98×
T	CPU	2.75	VQ	2D	0.69×	0.96×
T	CPU	2.25	VQ	2D	0.56×	0.87×
G	CPU	3.125	VQ	1D	0.78×	0.96×
T	GPU	4	Unif	1D	1.00×	1.00×
T	GPU	8	Unif	1D	2.00×	1.47×
T	GPU	16	FP	1D	4.00×	2.72×
T	GPU	2.125	VQ	2D	0.53×	1.03×
T	GPU	2.125	VQ	4D	0.53×	0.71×
T	GPU	3.125	VQ	2D	0.78×	1.06×

C VQ Configurations

Table 7 details the studied VQ configurations.

Table 7. **VQ configurations.** Group shape $(r \times c)$ indicates (rows$\times$columns)

bpv	d	b	group size	group shape	codebook bw
2.125	1D	2	256	(1×256)	8
2.125	2D	2	2,048	(4×256)	8
2.125	4D	2	65,536	(256×256)	8
2.25	1D	2	128	(1×128)	8
2.25	2D	2	1,024	(4×256)	8
2.25	4D	2	32,768	(128×256)	8
2.75	2D	2.5	2,048	(4×256)	8
3	2D	2.5	512	(2×256)	8
3.125	1D	3	8,192	(32×256)	8
3.125	2D	3	32,768	(128×256)	8
4.125	1D	4	1,024	(4 ×256)	8
4.125	2D	4	65,536	(256×256)	8

Table 8. **Weight-only quantization results of Llama-1**. We report WikiText2 perplexity in this table; lower is better.

		L1-7B	L1-13B	L1-30B	L1-65B
FP16		5.68	5.09	4.10	3.53
2.125 bpv (W2@g128)	RTN	1.9e3	781.20	68.04	15.08
	GPTQ	44.01	15.60	10.92	9.51
	AWQ	2.6e5	2.8e5	2.4e5	7.4e4
	OmniQuant	9.72	7.93	7.12	5.95
	GPTVQ 1D (ours)	16.29	**6.93**	**6.04**	**5.19**
	GPTVQ 2D (ours)	**9.64**	**6.58**	**5.63**	**4.91**
2.25 bpv (W2@g64)	RTN	188.32	101.87	19.20	9.39
	GPTQ	22.10	10.06	8.54	8.31
	AWQ	2.5e5	2.7e5	2.3e5	7.4e4
	OmniQuant	8.90	7.34	6.59	5.65
	GPTVQ 1D (ours)	16.64	**6.78**	**5.97**	**5.05**
	GPTVQ 2D (ours)	9.90	**6.43**	**5.56**	**4.86**
	GPTVQ 4D (ours)	**8.76**	**6.33**	**5.42**	**4.74**
3.125 bpv (W3@g128)	RTN	7.01	5.88	4.87	4.24
	GPTQ	6.55	5.62	4.80	4.17
	AWQ	6.46	5.51	4.63	3.99
	OmniQuant	6.15	5.44	4.56	3.94
	GPTVQ 1D (ours)	6.60	**5.34**	**4.48**	**3.85**
	GPTVQ 2D (ours)	**6.32**	**5.31**	**4.38**	**3.79**

D Extended results

Table 8 shows GPTVQ results on Llama-1. Table 9 shows GPTVQ results for 4.125 bpv on various models. Tables 10 and 11 present detailed LM-eval results.

E Mean and standard deviation over multiple runs

Table 12. **Mean and standard deviation over 10 random seeds**. Setting used: Llamav2-7B, 2D VQ, 8-bit codebook.

BPV	Mean and Std. Dev.
3.125	5.82 ± 0.01
4.125	5.59 ± 0.01

Table 9. **Weight-only 4 bit quantization results of Llama-1, Llama-2, and Mistral-7B models**. We report WikiText2 perplexity in this table; lower is better. Models marked 'L1' or 'L2' denote Llama-v1 and Llama-v2, respectively. M-7B denotes Mistral.

		L1-7B	L1-13B	L1-30B	L1-65B	L2-7B	L2-13B	L2-70B	M-7B	L3-8B
FP16		5.68	5.09	4.10	3.53	5.47	4.88	3.31	5.25	6.5
4.125 bpv (W4@g128)	RTN	5.96	5.25	4.23	3.67	5.72	4.98	3.46	5.42	6.73
	GPTQ	5.85	5.20	4.23	3.65	5.61	4.98	3.42	5.35	7.32
	AWQ	5.81	5.20	4.21	3.62	5.62	4.97	-	-	-
	OmniQuant	**5.77**	5.17	4.19	3.62	**5.58**	6.5	**4.95**	-	-
	GPTVQ 1D (ours)	5.96	**5.15**	**4.18**	**3.60**	5.62	**4.97**	**3.39**	**5.32**	-
	GPTVQ 2D (ours)	5.94	5.20	**4.18**	3.64	**5.59**	**4.94**	**3.38**	**5.32**	**6.41**

E.1 Comparison to AQLM

Additive Quantization for Language Models (Egiazarian et al., 2024) (AQLM) is a recent method that also uses vector quantization to compress LLMs to very low effective bit widths and achieves impressive bits per value vs accuracy results, as shown in Table 3. While both GPTVQ and AQLM employ VQ for LLM compression, our methods differ in several significant ways, which affects inference deployment, compression time, and on-disk model size.

AQLM uses larger vector dimension d, with $d = 8$, scale their codebooks exponentially in d, similar to us. E.g., for 2-bit results AQLM uses codebooks with 2^{15} or 2^{16} 8-dimensional entries, where each entry is stored in FP16. While the authors have shown that these configurations can be employed on Nvidia® GPUs, codebooks of these sizes would be harder to employ efficiently on a popular mobile platform. This is caused by the fact that many calls to the (5-bit) `TBL` instruction would be required, leading to significant additional latency during inference time. For example, decoding a single 16-bit index to an 8-bit FP16 would require $2 \times 8 \times 2^{11}$ 5-to-8-bit lookup tables (LUTs), where each lookup requires $2 \times 8 \times 11$ instructions to decode.

The full AQLM algorithm requires significant time to compress models. Compressing Llamav2-7B requires up to 35 hours on H100, while GPTVQ takes between 30 minutes and 3 hours on a single H100 GPU. It should however be noted that our method becomes significantly slower for higher quantization dimensionality, mainly due to the EM codebook initialization.

The long runtime of AQLM is caused in part by a block-wise fine-tuning step. This step allows the model to correct intra-layer effects of quantization error. While GPTVQ has no mechanism to correct intra-layer error effects, its results are competitive with AQLM. AQLM without the additional block fine-tuning step (i.e., Table 3), achieves a perplexity of 7.49 for the WikiText2 test set on Llama-v2-7B, a degradation of 0.38 point compared to 7.11 for GPTVQ under the same conditions.

F Ablations on hyperparameter choices

EM initialization To find seed centroids for EM initialization, we compare k-Means++ (Arthur & Vassilvitskii, 2007) to a quick and effective initialization method dubbed *Mahalanobis initialization*. In the latter method, we initialize EM for a matrix of N d-dimensional points $\mathbf{X}$ by first sorting all points by Mahalanobis distance (Bishop, 2006) to the mean of the data, then sampling k points spaced $\lfloor \frac{N}{k-1} \rceil$ apart from the sorted list. Intuitively, this method ensures that points are sampled at representative distances from the mean. Table 13 shows perplexity after GPTVQ for different EM initialization seed methods, and find that Mahalanobis initialization performs comparably to k-Means++, at increased speed.

EM iterations We explore the effect of the number of EM initialization iterations on the final perplexity of GPTVQ. Table 14 shows that even up to 100 iterations, results keep improving slightly, therefore we use 100 iterations as default.

Codebook update Table 15 includes an ablation on including codebook updates as described in Section A. We find that, in all cases, updating the codebook after running Algorithm 2 improves final perplexity, at the expense of moderately increased (though still reasonable) run time. We thus include codebook update in all training runs.

Method runtime GPTVQ can quantize large language models efficiently. Exact runtime depends on model, quantization setting (groupsize, bitwidth, VQ dimensionality), and several hyperparameters (EM iterations, codebook update iterations). As An indication of realistic run-times on a single H100: Llamav2-7B takes between 30 minutes and 1 hour, while Llamav2-70B takes between 3 and 11 hours.

G Codebook compression

While we find that 8 bit quantization of codebooks provides best results for the same overhead, we explore a different approach to codebook compression in this section.

For the case where $d = 1$, we could further compress the codebook $\mathbf{C}$ by stacking all codebooks for multiple blocks (e.g. all blocks in a tensor) and rank-reducing the resulting matrix. For a single tensor, $\mathbf{C}$ has shape $N_G \times k$, where N_G is the number of groups in the corresponding weight tensor, k is the number of centroids per codebook. We first sort values in each codebook in $\mathbf{C}$, and reassign the indices in $\mathbf{I}$ accordingly. Then, we perform SVD on $\mathbf{C}$, leading to matrices $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$, of shapes $N_G \times k$, $k \times k$ and $k \times k$, respectively. $\mathbf{U}' = \mathbf{U}\mathbf{\Sigma}$, and reduce the rank of this matrix to r, yielding a $N_G \times r$ shaped matrix $\mathbf{U}''$. We also reduce the rank of $\mathbf{V}$ accordingly, yielding $r \times r$ matrix $\mathbf{V}'$. Then, we perform gradient descent (GD) on the loss of equation 7, but with respect to the codebook tensor factors $\mathbf{U}''$ and $\mathbf{V}'$. In each GD step, $\widehat{\mathbf{C}}$ is created as $\widehat{\mathbf{C}} = \mathbf{U}''\mathbf{V}'^T$, and the rest of the codebook up procedure as described earlier is followed. Lastly, only the codebook tensor factor $\mathbf{U}''$ is quantized, as $\mathbf{V}'$ gives very little overhead. During inference, $\widehat{\mathbf{C}}$ is quantized per codebook after construction.

For higher dimensions, Tucker factorization could be employed. However, in this case there is no natural ordering in which to sort the elements of each codebook.

In Table 18 we compare the effect of either rank reducing by 50%, or quantizing the codebook to 8-bit (our default approach), to keeping the codebook in FP16 and increasing the group size. In all three settings the overhead of the codebook is the same. We see that, for the same overhead, quantization gives best results. For this reason, and because codebook SVD does not easily apply to $d > 1$, we have not explored codebook SVD further, and instead use INT8 quantization as our default approach.

H Blockwise data normalization

In order to lower the error of vector quantization, we apply blockwise data normalization to the data before the codebook initialization. For each group corresponding to a new codebook we perform element-wise division $\mathbf{W}_i \oslash \mathbf{S}_i$ of the weight sub-matrix matrix $\mathbf{W}_i$ by the corresponding scales $\mathbf{S}_i$. The scale is computed block-wise for every sub-row of $\mathbf{W}_i$, e.g. for a block size of 16, 32, or 64.

Given a set of blocks (sub-rows) $\mathbf{w}^{(i)}$, the scale $s^{(i)}$ for each of them is computed as $s^{(i)} = \max_j |w_j^{(i)}|$. In order to minimize the overhead, the scales are quantized to 4-bit integer.

We found that it is beneficial to perform quantization in log-scale to capture several orders of magnitudes in weights. The quantized scales are computed as $s_{int}^{(i)} = \lceil \frac{\log_2[s^{(i)}]-z}{a} \rfloor a$, where a is the quantization scale shared among the group of weights. In order to accurately represent zero in log-space which corresponds to unit scaling, we use the floating point offset z. In practice the value of z is shared within the columns of $\mathbf{W}$ and thus has negligible overhead. Finally the scaled sub-row is normalized as $\mathbf{w} \cdot 2^{-a s_{int} - s_0}$, where $s_0 = \log_2(z)$. The scaled data is used for codebook initialization. The inverse scaling is applied at VQ decoding step.

I Centroid fine-tuning

The GPTVQ algorithm has two reasons for being not optimal. First, it processes layers sequentially, minimizing the local error in each layer. Second, the codebook is selected per group that usually spans across multiple columns. Once the first d columns are quantized (step 4 in Algorithm 2), the remaining weights of the *same* group are updated (step 7 in Algorithm 2). Therefore, the selected codebook will not optimally represent the remaining weights in the group.

One straight-forward solution to improve the performance is to fine-tune the selected codebook entries end-to-end. To this end, we fine-tuned the codebooks and scales on SlimPajama (Soboleva et al., 2023) dataset for 1000 steps with batch size 32 and sequence lengths of 1024. We used Adam optimizer with warmup and cosine scheduler for both codebooks and scales. We selected the codebook learning rate using a grid search. The scales learning rate was reduced by 10^3 times.

Table 19 compares the fine-tuning results with the original GPTVQ results. We can see that for all models and bits, fine-tuning consistently improves the results except zero-shot accuracy at 3.125 bits.

Note that we also tried to update the indices similarly to PV-Tuning (Malinovskii et al., 2024a), but we did not see any significant improvement of results. We hypothesize that due to the nature of GPTVQ update of the weights, the selected indices are very close to the optimal one. This is partially confirmed by the fact that AQLM algorithm, which is used by PV-tuning, requires layer-wise and block-wise fine-tuning of centroids, while the original GPTVQ algorithm produces comparable results right after initialization.

Table 10. **LM-eval results of quantized Llama-v2 7B and 13B, and Llama-v3 8B models.**

	#Bits	Method	PIQA	ARC-e	Arc-c	BoolQ	HellaSwag	Winogrande	**Avg.**↑
Llama-v2-7B	FP16		79.11	74.58	46.25	77.74	75.99	69.14	70.47
	2.125 bpv (W2@g128)	RTN	51.09	27.95	25.00	41.13	26.57	49.88	36.94
		GPTQ	54.84	30.64	25.09	53.43	33.09	51.54	41.44
		VQ-1D	**61.21**	**38.76**	**24.66**	**62.78**	**45.78**	**53.83**	**47.84**
		VQ-2D	**71.33**	**57.41**	**32.94**	**65.60**	**59.85**	**64.72**	**58.64**
		VQ-4D	**73.34**	**60.44**	**34.39**	**65.50**	**63.99**	**65.04**	**60.45**
	2.25 bpv (W2@g64)	RTN	58.76	36.66	24.83	41.87	40.38	51.93	42.40
		GPTQ	60.83	39.02	25.17	59.33	45.82	55.49	47.61
		VQ-1D	**64.80**	**49.33**	**28.24**	**65.87**	**53.37**	**54.93**	**52.76**
		VQ-2D	**72.36**	**63.47**	**35.41**	**72.14**	**60.92**	**64.72**	**61.50**
		VQ-4D	**73.99**	**64.73**	**36.77**	**71.19**	**64.84**	**65.75**	**62.88**
	3.125 bpv (W3@g128)	RTN	76.77	70.50	42.92	71.71	73.96	67.64	67.25
		GPTQ	77.37	68.14	40.70	71.04	72.50	67.25	66.16
		VQ-1D	**77.86**	**68.64**	**40.96**	**73.85**	**72.29**	**67.80**	**66.90**
		VQ-2D	**77.64**	**73.15**	**43.17**	**74.22**	**72.61**	**69.06**	**68.31**
Llama-v2-13B	FP16		80.52	77.53	49.23	80.52	79.38	72.14	73.22
	2.125 bpv (W2@g128)	RTN	58.43	32.32	25.51	47.86	39.40	48.86	42.06
		GPTQ	59.52	40.15	27.65	57.06	41.56	53.43	46.56
		VQ-1D	**73.23**	**64.10**	**35.75**	**71.38**	**60.71**	**65.43**	**61.77**
		VQ-2D	**75.24**	**68.27**	**38.99**	**69.91**	**65.81**	**68.98**	**64.53**
		VQ-4D	**75.46**	**71.93**	**42.92**	**67.86**	**69.26**	**66.93**	**65.73**
	2.25 bpv (W2@g64)	RTN	61.59	41.58	25.43	49.79	48.24	51.85	46.41
		GPTQ	70.13	56.65	31.57	51.10	56.62	58.88	54.16
		VQ-1D	**72.36**	**67.63**	**37.37**	**74.13**	**62.89**	**65.27**	**63.28**
		VQ-2D	**74.97**	**67.63**	**40.53**	**69.24**	**67.11**	**69.30**	**64.80**
		VQ-4D	**76.66**	**69.87**	**43.00**	**74.68**	**70.81**	**69.69**	**67.45**
	3.125 bpv (W3@g128)	RTN	78.89	74.28	46.76	77.25	76.51	70.80	70.75
		GPTQ	79.33	75.84	47.01	78.90	77.16	70.40	71.44
		VQ-1D	**78.94**	**75.04**	**46.76**	**79.42**	**75.85**	**72.45**	**71.41**
		VQ-2D	**79.27**	**74.33**	**46.67**	**77.40**	**77.21**	**72.45**	**71.22**
Llama-v3-8B	FP16		79.9	80.1	50.4	-	60.2	72.8	68.6
	2.125 bpv (W2@g128)	RTN	53.1	24.8	22.1	-	26.9	53.1	36.0
		GPTQ	53.9	28.8	19.9	-	27.7	50.5	36.2
		VQ-1D	**56.58**	**35.10**	**18.26**	**60.00**	**38.25**	**57.06**	**41.05**
		VQ-2D	**69.48**	**62.58**	**29.01**	**72.29**	**43.05**	**65.51**	**53.93**
		VQ-4D	**71.93**	**69.19**	**32.68**	**69.45**	**45.62**	**67.17**	**57.32**
	2.25 bpv (W2@g64)	**VQ-1D**	**71.16**	**70.24**	**34.04**	**74.13**	**45.71**	**65.27**	**57.29**
		VQ-2D	**74.27**	**71.30**	**37.54**	**69.24**	**49.07**	**69.30**	**60.30**
		VQ-4D	**75.68**	**72.60**	**41.04**	**74.68**	**52.22**	**69.69**	**62.25**
	3.125 bpv (W3@g128)	RTN	62.3	32.1	22.5	-	29.1	54.7	40.2
		VQ-1D	**77.31**	**77.90**	**43.43**	**79.42**	**57.28**	**72.45**	**65.68**
		VQ-2D	**77.80**	**76.68**	**45.14**	**77.40**	**58.16**	**72.45**	**66.05**

Table 11. **LM-eval results of quantized Mistral-7B and Mixtral-8x7B models.**

	#Bits	Method	PIQA	ARC-e	Arc-c	BoolQ	HellaSwag	Winogrande	**Avg.↑**
Mistral-7B	FP16		82.10	79.59	53.92	83.58	81.07	73.88	75.69
	2.125 bpv (W2@g128)	RTN	53.05	29.42	26.62	38.56	29.26	49.57	37.75
		GPTQ	57.73	35.65	26.62	46.06	36.06	49.49	41.93
		VQ-1D	**55.22**	**35.94**	**25.51**	**54.01**	**34.35**	**52.01**	**42.84**
		VQ-2D	**73.78**	**69.02**	**37.80**	**76.57**	**64.52**	**65.35**	**64.51**
		VQ-4D	**75.90**	**71.63**	**41.98**	**69.85**	**68.59**	**66.46**	**65.73**
	2.25 bpv (W2@g64)	RTN	60.72	38.47	27.56	44.83	46.10	51.07	44.79
		GPTQ	65.83	46.21	30.20	62.11	50.64	55.56	51.76
		VQ-1D	**67.41**	**59.01**	**33.79**	**67.74**	**53.80**	**55.96**	**56.28**
		VQ-2D	**74.86**	**69.23**	**40.53**	**74.07**	**65.93**	**67.40**	**65.34**
		VQ-4D	**76.61**	**73.15**	**42.41**	**77.95**	**69.48**	**69.30**	**68.15**
	3.125 bpv (W3@g128)	RTN	80.79	74.62	48.46	80.00	78.66	68.19	71.79
		GPTQ	79.82	75.51	49.40	81.22	77.34	70.17	72.24
		VQ-1D	**78.84**	**75.29**	**47.87**	**79.57**	**75.32**	**69.30**	**71.03**
		VQ-2D	**81.12**	**78.70**	**51.02**	**82.39**	**78.05**	**72.06**	**73.89**
Mixtral-8x7B	FP16		83.46	73.74	55.89	84.74	82.45	75.30	75.93
	2.125 bpv (W2@g128)	RTN	51.90	27.27	25.85	47.98	27.07	49.64	38.29
		GPTQ	59.79	35.44	27.30	52.08	41.80	50.83	44.54
		VQ-1D	**68.93**	**50.93**	**33.02**	**62.51**	**52.52**	**61.17**	**54.85**
		VQ-2D	**76.39**	**57.87**	**38.91**	**74.95**	**67.03**	**71.03**	**64.36**
		VQ-4D	**78.13**	**65.57**	**46.42**	**78.59**	**72.40**	**71.11**	**68.70**
	2.25 bpv (W2@g64)	RTN	62.08	38.68	28.41	54.46	44.40	53.12	46.86
		GPTQ	66.05	42.93	28.58	50.12	49.59	55.41	48.78
		VQ-1D	**69.42**	**50.55**	**36.09**	**64.95**	**59.51**	**63.93**	**57.41**
		VQ-2D	**77.42**	**62.12**	**42.66**	**72.39**	**70.74**	**68.90**	**65.71**
		VQ-4D	**79.16**	**67.68**	**48.04**	**76.09**	**73.43**	**71.11**	**69.25**
	3.125 bpv (W3@g128)	RTN	81.50	68.77	50.60	80.92	79.71	72.93	72.40
		GPTQ	80.85	69.32	52.05	81.35	78.40	74.43	72.73
		VQ-1D	**80.90**	**71.34**	**52.73**	**84.83**	**77.62**	**73.64**	**73.51**
		VQ-2D	**82.59**	**72.94**	**54.86**	**84.46**	**80.61**	**74.82**	**75.05**

Table 13. **Effect of EM initialization**. Setting used: Llamav2-7B, 2D 3-bit VQ, blocksize 2048.

Lookup method	BPV	Setting	PPL	Time (s)
1D 3B 1024	3.125	Mahalanobis	6.05	605
		K++	6.16	3328
2D 3B 16384	3.125	Mahalanobis	5.65	756
		K++	5.63	3168
1D 4B 2048	4.125	Mahalanobis	5.86	1272
		K++	5.88	2116
2D 4B 65536	4.125	Mahalanobis	5.59	3816
		K++	5.57	6644

Table 14. **Effect of number of EM interations**. Setting used: BLOOM-560m 2D 3-bit VQ with blocksize 4096, perplexity on WikiText2 test set.

EM iterations	PPL
10	24.49
30	24.18
50	24.12
75	24.11
100	24.09

Table 15. **Effect of codebook fine-tuning on final PPL for Llamav2-7B**.

d	b	gs	Update	PPL	Runtime (s)
1	2	512	N	43.14	625
			Y	14.02	1857
	3	1024	N	6.05	712
			Y	6.01	1916
2	2	2048	N	8.64	723
			Y	8.21	1335
	3	8192	N	5.93	1585
			Y	5.88	2195

Table 16. **Effect of scaling block size on perplexity for Llamav2-7B**. d: VQ-dimension; b: VQ bitwidth per dimension; gs: block size; Codebooks are quantized to 8 bits.

d	b	gs	Scaling BS					
			None	128	64	32	16	8
1	2	512	14.01	16.74	2744.9	480.8	15.36	13.79
	3	1024	6.02	5.97	6.00	5.87	5.82	5.72
2	2	2048	8.23	8.38	8.04	7.97	7.56	6.89
	3	8192	5.91	5.82	5.78	5.73	5.74	5.66

Table 17. **Effect of scaling on perplexity for different models**. Configurations with equal overhead with or without the scaling are considered. d: VQ-dimension; b: VQ bitwidth per dimension; gs: block size; Codebooks are assumed to be quantized to 8 bit.

d	b	gs	Scale	Llamav2-7B	Llamav2-13B	Mistral-7B	Mixtral-8x7B
1	2	256	N	14.01	7.34	15.03	8.56
		512	Y	171.29	7.44	87.60	8.11
	3	512	N	5.98	5.21	5.76	4.60
		1024	Y	6.01	5.17	5.77	4.59
2	2	2048	N	8.23	6.69	10.98	6.73
		4096	Y	8.49	6.50	10.28	6.37
	3	8192	N	5.91	5.19	8.63	4.52
		16384	Y	5.56	5.11	5.53	4.30

Table 18. **Choices in experimental setup leading to comparable bits per value**. d: VQ-dimension; b: VQ bitwidth per dimension; gs: block size; Q: 8-bit codebook quantization yes/no; SVD: codebook SVD yes/no. BPV: bits per value.

d	b	gs	Q	SVD	BPV	PPL
1	2	512	N	N	2.125	14.01
		256	Y	N	2.125	11.57
		256	N	Y	2.125	44.99
	3	1024	N	N	3.125	6.01
		512	Y	N	3.125	5.98
		512	N	Y	3.125	5.98
2	2	4096	N	N	2.125	8.37
		2048	Y	N	2.125	8.23
	3	16384	N	N	3.125	5.93
		8192	Y	N	3.125	5.87

Table 19. **Weight-only quantization results of Llama-v2, Llama-v3, Mistral, and Phi-3 mini 4k instruct models with centroid fine-tuning (+FT)**. We report WikiText2 perplexity at the context length of 2048 and average zero-shot accuracy. Models marked L2 denote Llama-v2, L3 denote Llama-v3, M denotes Mistral, and P3-mini denotes Phi-3 mini 4k instruct.

		WikiText2 perplexity ↓					Zeroshot avg acc. ↑				
		L2-7B	L2-13B	L3-8B	M-7B	P3-mini	L2-7B	L2-13B	L3-8B	M-7B	P3-mini
FP16		5.47	4.89	6.14	5.25	6.36	70.4	73.3	74.2	75.7	76.1
2.125 W2 g128	**GPTVQ 2D**	7.60	6.38	10.99	7.69	10.18	59.9	65.0	60.3	64.0	62.4
	GPTVQ 2D + FT	**6.57**	**5.70**	**8.65**	**6.22**	**8.00**	**64.2**	**67.2**	**66.8**	**69.1**	**67.5**
	GPTVQ 4D	7.14	5.97	9.61	6.75	9.01	62.2	67.4	61.4	67.7	65.3
	GPTVQ 4D + FT	**6.39**	**5.58**	**8.25**	**6.04**	**7.75**	**64.4**	**68.4**	**68.9**	**70.0**	**68.4**
2.25 W2 g64	**GPTVQ 2D**	7.42	6.25	10.45	7.36	9.70	60.6	66.7	61.9	64.6	63.9
	GPTVQ 2D + FT	**6.49**	**5.65**	**8.51**	**6.16**	**7.91**	**64.1**	**68.7**	**67.1**	**70.4**	**67.8**
	GPTVQ 4D	6.92	5.88	9.26	6.59	8.86	62.0	67.4	63.1	68.2	64.6
	GPTVQ 4D + FT	**6.31**	**5.54**	**8.13**	**5.99**	**7.59**	**65.6**	**70.0**	**68.9**	**70.2**	**68.9**
3.125 W2 g128	**GPTVQ 2D**	5.79	5.11	6.97	5.53	6.93	67.8	**71.4**	71.1	**73.9**	72.3
	GPTVQ 2D + FT	**5.73**	**5.10**	**6.96**	**5.49**	**6.56**	**68.7**	71.2	**72.5**	73.8	**73.7**