ELUTQ: Efficient LUT-Aware Quantization for Deploying Large Language Models on Edge Devices

Xin Nie [1] **Liang Dong** [1] **Haicheng Zhang** [1] **Jiawang Xiao** [1] **G. Sun** [1]

Abstract

Weight quantization effectively reduces memory consumption and enable the deployment of Large Language Models on CPU-based edge devices, yet existing hardware-friendly methods often rely on uniform quantization, which suffers from poor weight-distribution fitting and high dequantization overhead under low-bit settings. In this paper, we propose **ELUTQ**, an efficient quantization framework featuring a novel quantization format termed *Hierarchical Linear Quantization* (HLQ). HLQ is designed to better capture the statistical characteristics of weights without increasing the computational cost of Bit-serial LUT-based GEMM operations, thereby eliminating dequantization overhead. HLQ is orthogonal to existing quantization algorithms. For the LLaMA3.1-8B model, when combined with the post-training quantization framework, HLQ enhances uniform quantization by achieving approximately 8% perplexity reduction at 3-bit precision and 85% perplexity reduction at 2-bit precision. When combined with efficient finetuning techniques, HLQ further improves model accuracy. A novel disk-offload technique is integrated into ELUTQ, enabling it to complete the quantization of LLaMA 3.1–70B using only 64 GB of CPU memory and 48 GB of VRAM, significantly reducing the hardware requirements for large-scale model quantization. To enable efficient deployment on edge devices, ELUTQ designs high-performance CPU kernels to support end-to-end inference. Under a 4-thread configuration with batch size = 1, our 2-bit quantized LLaMA2-7B model achieves a throughput of over 25 tokens per second on an Apple M2 chip. All the code is available at `https://github.com/Nkniexin/ELUTQ`.

[*]Equal contribution [1]College of Electronic Information and Optical Engineering, Nankai University, Tianjin, China. Correspondence to: G. Sun <sungl@nankai.edu.cn>.

1. Introduction

Large Language Models (LLMs) have demonstrated exceptional performance across diverse tasks, including natural language understanding, image recognition, and multimodal reasoning. Traditionally deployed in cloud environments with abundant computational resources, these models are now increasingly being adapted for edge devices, such as smartphones, IoT systems, and autonomous vehicles, to meet growing demands for low-latency inference, privacy preservation, and real-time intelligent services.

Unlike high-performance GPU servers, on-device hardware typically operates under stringent resource constraints, characterized by limited memory capacity and computational power. These devices predominantly employ ARM or x86 CPUs, which offer restricted vectorization support and limited parallelism. To address these challenges, model quantization (Frantar et al., 2022; Xiao et al., 2023; Lin et al., 2024; Kim et al., 2023; Shang et al., 2023; Chen et al., 2024b) has emerged as a widely adopted compression technique, where high-precision weights are mapped to discrete integer values and stored using low-bit representations. This approach drastically reduces memory footprint while preserving model accuracy. Recent advances demonstrate that 8-bit weight quantization achieves near-lossless performance (Xiao et al., 2023; Yao et al., 2022). Furthermore, 4-bit quantization techniques (Frantar et al., 2022; Lin et al., 2024; Shao et al., 2023; Kim et al., 2023; Chen et al., 2024c) typically incur less than 3% accuracy degradation, with ongoing research further improving robustness. However, activation quantization remains more challenging due to the prevalence of outlier values. While some studies (Xiao et al., 2023; Liu et al., 2024) explore low-bit activation quantization, most practical implementations retain activations in FP16 precision (e.g., W4A16, W3A16) to ensure accuracy. Meanwhile, some research (Chen et al., 2024a; Shang et al., 2023; Huang et al., 2024) focuses on extreme low-bit weight quantization, pushing the boundaries of efficiency without sacrificing model quality.

Quantization can be categorized as uniform (Frantar et al., 2022; Lin et al., 2024; Shao et al., 2023; Chen et al., 2024c) or non-uniform (Chee et al., 2023; Park et al., 2024; Kim et al., 2023; Xu et al., 2023; Zhao & Yuan, 2025) depend-

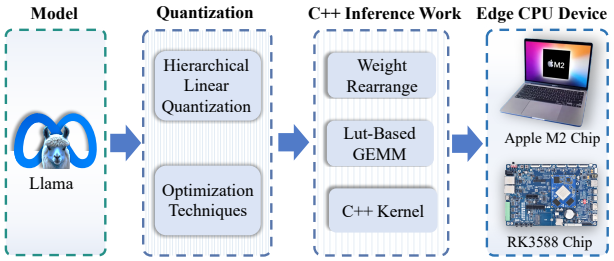


Figure 1: Overview of ELUTQ.

ing on whether the quantization intervals are equal. Uniform quantization divides the weight space into equal intervals, making it hardware-friendly and efficient for acceleration (Lin et al., 2024; Frantar et al., 2022). However, as noted in (Dettmers et al., 2023a; Kim et al., 2023), this approach poorly matches the bell-shaped distribution of typical weights, incurring substantial approximation error. Non-uniform quantization addresses this by adaptively allocating bins, via clustering (Kim et al., 2023; Zhao & Yuan, 2025) or codebooks (Chee et al., 2023) to better fit the weight distribution. While this improves representational efficiency, it often sacrifices hardware compatibility due to irregular memory access patterns. To address these limitations, We propose Hierarchical Linear Quantization, a non-uniform quantization format that reduces weight quantization error while maintaining hardware compatibility.

Many large model quantization methods aim to efficiently complete model quantization, where efficiency primarily refers to time and memory consumption, particularly video memory (VRAM), which is typically more constrained and expensive than CPU memory. Achieving quantization with low memory usage and short processing time is critical, as it enables running quantization algorithms even on consumer-grade GPUs such as the RTX 3090 or RTX 4090. Most post-training quantization approaches (Frantar et al., 2022; Lin et al., 2024; Xiao et al., 2023; Huang et al., 2024; Shang et al., 2023; Zhao & Yuan, 2025; Kim et al., 2023) can complete quantization quickly and with modest memory requirements. These train-free methods operate without retraining and typically quantize the model layer by layer. For example, GPTQ quantizes one linear layer at a time. With sufficient CPU memory, even large models such as LLaMA3.1-70B can be quantized on a single A6000 GPU. In contrast, efficient finetuning–based methods (Shao et al., 2023; Dettmers et al., 2023b;a; Chee et al., 2023; Li et al., 2023; Xu et al., 2023; Chen et al., 2024b) require partial retraining, which increases time cost but often mitigates VRAM pressure by freezing most parameters, training only a small subset, or performing quantization in batches. This strategy allows quantization to be completed within limited video memory. However, while prior studies have primarily focused on reducing GPU memory consumption, the CPU memory overhead has been largely neglected. Existing quantization frameworks typically require loading the entire model into CPU memory during processing, which poses a critical bottleneck for large-scale models. For instance, quantizing a LLaMA 3.1–70B model requires at least 140 GB of CPU RAM, and the actual usage is often much higher, exceeding the capacity of most standard servers. In this paper, we integrate our proposed Hierarchical Linear Quantization with efficient quantization techniques to enhance model accuracy while maintaining low GPU memory usage, reduced CPU memory footprint, and short quantization time. For example, our framework can quantize the Llama 3.1-70B model using 2 RTX 4090 GPU (48GB VRAM) and only 64GB of CPU RAM, significantly reducing both GPU and CPU memory requirements.

Although weight quantization reduces memory footprint, traditional methods require dequantization to higher precision (8-bit/FP16) for computation, introducing significant overhead that can paradoxically slow down inference at low bit-widths. Recent advances (Wei et al., 2025; Park et al., 2025b; 2022) address this by replacing standard GEMM with lookup table (LUT)-based operations that implement generalized FP-INT multiplication, eliminating dequantization while achieving both linear latency reduction with bit-width and improved energy efficiency. FLGLUT (Park et al., 2025b) accelerates these operations on GPUs and T-MAC (Wei et al., 2025) optimizes for CPUs via SIMD instructions. We enhance this paradigm through a pure C++ kernel redesign that specifically supports our novel Hierarchical Linear Quantization format, maintaining cross-platform compatibility while optimizing for our method's unique requirements.

Figure 1 shows the ELUTQ design. Our contributions are as follows.

- **We propose Hierarchical Linear Quantization (HLQ)**, a novel non-uniform quantization format that provides greater flexibility in weight representation compared to uniform quantization, while enabling efficient matrix computation through LUT-based GEMM.
- **We integrate HLQ into existing efficient quantization methods**, including post-training quantization and efficient fine-tuning techniques, demonstrating that HLQ is orthogonal to most existing methods and can significantly enhance model performance under low-bit settings, without introducing noticeable memory or time overhead to the quantization pipeline.
- **We design an efficient CPU kernel tailored for the HLQ format**, which enables high-performance matrix operations on edge devices.
- **We introduce ELUTQ**, a unified quantization frame-

work built upon HLQ. ELUTQ incorporates common quantization optimization techniques and includes a fully C++-implemented inference runtime, which supports end-to-end deployment of quantized models on edge devices.

2. Related works

2.1. Model Quantization

2.1.1. Uniform Quantization.

The uniform quantization formula is as follows:

Quantization:

$$\mathrm{W}_{int} = \mathrm{clamp}(\lfloor \frac{\mathrm{W}}{s} \rceil + z, 0, 2^q - 1). \tag{1}$$

Here, $\lfloor \cdot \rceil$ denotes the rounding operation, q is the quantization bit width, s is the scale factor, z is the zero-point, W represents the original weights, and W_{int} denotes the quantized integer weights.

Dequantization:

$$\hat{\mathrm{W}} = (\mathrm{W}_{int} - z) \cdot s, \tag{2}$$

where $\hat{\mathrm{W}}$ represents the dequantized weights. Uniform quantization is hardware-friendly, as dequantization can be implemented with simple multiplication. However, its representational capability is limited because it maps weights into a uniformly spaced range. Moreover, in modern computing systems, the smallest storage and computation unit is typically 8 bits. Therefore, for low-bit settings (e.g., 2-bit or 3-bit), the quantized integer weights $\hat{\mathrm{W}}$ must first be dequantized into 8-bit or 16-bit formats before computation, which often introduces non-negligible computational overhead.

2.1.2. Efficient Quantization.

Efficient quantization aims to complete model quantization with minimal time and memory consumption. It is generally categorized into two types: post-training quantization (PTQ) and efficient finetuning. PTQ requires only a small calibration dataset and can be completed within a relatively short time. Several research streams have advanced PTQ for Large Language Models. Some works, such as GPTQ (Frantar et al., 2022) and AWQ (Lin et al., 2024), focus on weight-only quantization, demonstrating minimal performance degradation (e.g., less than 3% perplexity increase) with 4-bit uniform quantization. Other approaches (Dettmers et al., 2023b; Kim et al., 2023) differentiate weights by their significance, storing a subset of important weights in higher precision. Furthermore, studies including SqueezeLLM (Kim et al., 2023) and GANQ (Zhao & Yuan, 2025) observe that weights in LLMs often follow a bell-shaped distribution and subsequently propose non-uniform quantization methods based on k-means clustering. Beyond weight quantization, activation quantization has also been explored. Methods like SmoothQuant (Xiao et al., 2023) and LLM.int8() (Dettmers et al., 2022) have achieved notable results in the challenging W8A8 weight-activation joint quantization setting. Compared with post-training quantization, efficient finetuning typically requires a little more data and time to search and adjust model parameters. OmniQuant (Shao et al., 2023) searches quantization parameters block by block and is the first to achieve promising results under 2-bit settings. PB-LLM (Shang et al., 2023) employs a feature segmentation strategy to achieve competitive performance under 2-bit weight quantization. Meanwhile, DB-LLM (Chen et al., 2024b) decomposes 2-bit quantized weights into two independent sets of binary weights and further utilizes distillation to enhance model performance, albeit at the cost of introducing substantial fine-tuning overhead. BiLLM (Huang et al., 2024) pushes the boundary further by leveraging weight distribution characteristics to achieve an average bit-width of approximately 1.11 bits.

2.2. LUT-Based GEMM

There are two primary computational paradigms for LUT-based GEMM, as illustrated in the Figure 2.

Figure 2(b) depicts the first implementation scheme of LUT-based GEMM. Its core idea is to directly store the high-precision weight values corresponding to low-bit integer weights in the LUT. Consequently, during dequantization, high-precision weights can be reconstructed via direct LUT accesses instead of computationally expensive arithmetic operations, significantly reducing the overhead of dequantization. It should be noted that, in this paradigm, although weights are reconstructed into a high-precision format via the LUT, they maintain the same numerical precision (e.g., both FP16) as the activations during the matrix multiplication.

Figure 2(c) illustrates another GEMM paradigm termed **Bit-serial LUT-based GEMM**. Unlike the scheme in Figure 2(b), this method utilizes the LUT to store all possible dot products between an activation vector and single-bit weights. The detail computational procedure is shown in Figure 3. First, a q-bit quantized weight matrix W_{int} is decomposed into q single-bit matrices $\{W_0, W_1, ..., W_{q-1}\}$ offline, where each element is either 0 or 1, representing the respective bit planes of the original weights. For example, for integer values (9, 7, 6, 3) with binary representations (1001, 0111, 0110, 0011), the matrix for the lowest bit is (1, 1, 0, 1), and the matrix for the highest bit is (1, 0, 0, 0). This decomposition is performed offline, incurring no

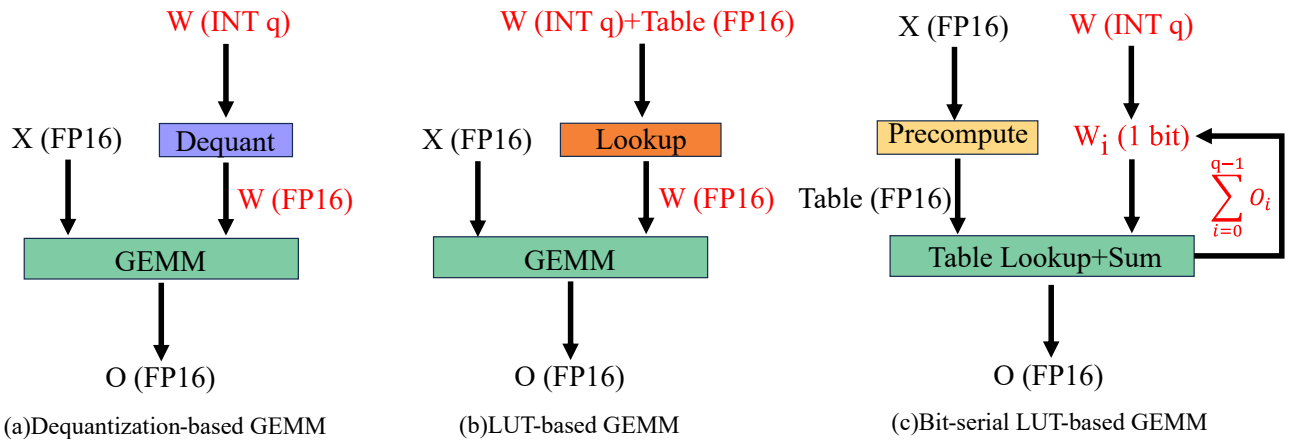


Figure 2: Illustration of three computation paradigms for weight-only quantized matrix multiplication.

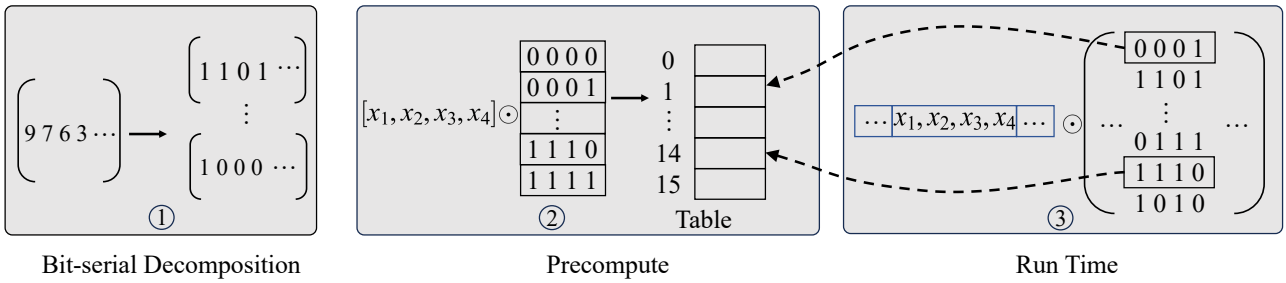


Figure 3: Detail pipeline of Bit-serial Lut-based GEMM.

runtime overhead. During inference, for an activation vector of the group size g, the system precomputes the dot products between this activation vector and all 2^g possible combinations of single-bit weights, storing the results in the LUT. Thus, the original matrix computation requiring high-precision multiply-accumulate operations is simplified into highly efficient table lookups followed by summation. This paradigm has been demonstrated to offer high computational efficiency and energy efficiency (Park et al., 2025b; Wei et al., 2025). For example, FIGLUT (Park et al., 2025b) optimized the table structure for GPU architectures to avoid bank conflicts, while T-MAC leveraged CPU vectorized lookup instructions (AVX2/NEON) to enable efficient LUT operations on CPUs.

The HLQ method is built upon the Bit-serial LUT-based GEMM computational paradigm. In contrast to prior works like FIGLUT and T-MAC, which primarily focused on designing efficient computational kernels for this paradigm, our work emphasizes optimization at the quantization algorithm level. The objective is to enhance the accuracy of low-bit quantized models, thereby achieving a synergistic improvement in both accuracy and efficiency within this highly efficient computational paradigm.

2.3. Inference Framework

Many frameworks aim to enable efficient inference of quantized models across a wide range of hardware platforms. For GPUs, vLLM (Kwon et al., 2023) employs the PageAttention technique to optimize key-value (KV) memory management and supports quantization formats such as GPTQ and AWQ. SGlang (Zheng et al., 2024) reduces response latency through shared prefix requests and efficient caching strategies. In addition, frameworks like TensorRT-LLM (NVIDIA, 2023) and MLC-LLM (Chen, 2023) have also been developed for GPU-optimized inference. For CPUs and other edge processors, llama.cpp (Gerganov, 2023) is a lightweight framework implemented entirely in C++. Although its inference speed is slower compared to GPU-accelerated solutions and thus not suitable for large-scale online services, it is well-suited for edge computing, IoT, and low-throughput scenarios, providing a practical solution for basic inference in GPU-free environments.

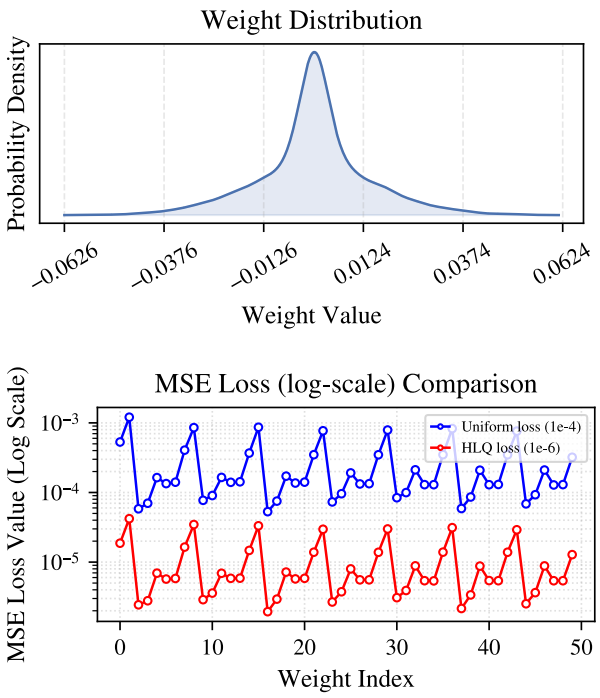


Figure 4: (Top) The weight distribution of one output channel in a up_proj of LLaMA3.1-8B. (Bottom) MSE loss for weight comparsion between Uniform and HLQ.

3. Motivation

3.1. Bell-Shaped Distribution of Weights

As noted in prior studies (Kim et al., 2023; Dettmers et al., 2023a; Zhao & Yuan, 2025), weights often exhibit a bell-shaped distribution. Uniform quantization, which maps weights to a uniformly spaced grid, struggles to accurately capture such distributions, especially under low-bit settings. Figure 4 shows that the weight distribution of an out_channel in the up_projection layer of the LLaMA3.1-8B model closely follows a Gaussian pattern, with values heavily concentrated near zero. This characteristic, commonly observed across various models, highlights the inherent limitation of uniform quantization in effectively representing natural weight distributions. Consequently, non-uniform quantization strategies are often necessary to achieve higher fidelity in low-bit compression.

3.2. Optimization for Bit-Serial LUT-Based GEMM

For ordinary LUT-based GEMM algorithms shown in figure 2(b), previous efforts such as SqueezeLLM (Kim et al., 2023) and GANQ (Zhao & Yuan, 2025) have achieved significant improvements in algorithmic performance for 3-bit and 4-bit quantization, along with the development of highly efficient CUDA kernels for accelerated inference. However, these implementations still lack support for CPU devices. Regarding Bit-serial LUT-Based GEMM, existing works including T-MAC (Wei et al., 2025), LUT-GEMM (Park et al., 2022), and FIGULT (Park et al., 2025b) have primarily focused on optimizing kernel design to support efficient uniform quantization schemes, yet they do not address algorithmic enhancements for improving quantization accuracy. Contemporary work like AnyBCQ (Park et al., 2025a) has made certain explorations, focusing on designing multi-precision models, with its 2-bit models achieving limited performance and primarily targeting model inference on GPUs. This limitations restricts the broader application of such kernels in cpu-based edge devices. To bridge this gap, our work proposes using HLQ to boost quantization accuracy at the algorithm level, while leveraging Bit-Serial LUT-Based GEMM to enable efficient inference on CPU devices.

3.3. Why Choose Bit-Serial LUT-Based GEMM for Edge Devices

Edge devices are predominantly based on CPU architectures, which offer significantly lower programmibility compared to GPU architectures. This limitation prevents developers from customizing lookup tables to implement LUT-based GEMM shown in figure 2(b). However, CPUs with x86_64 or ARM architectures provide a set of vectorized table-lookup instructions, such as _mm256_shuffle_epi8 and vqtbl1q_u8, which naturally align with the computational paradigm of Bit-serial LUT-based GEMM. Prior work such as T-MAC (Wei et al., 2025) has demonstrated that these instructions can be leveraged to enable efficient execution of Bit-serial LUT-based GEMM on edge devices. This observation motivates our work to bridge the algorithm-kernel co-design gap by combining high-accuracy non-uniform quantization at the algorithm level with a Bit-serial LUT-based inference framework tailored for common CPU instruction sets.

4. Methodology

In this section, we first present the proposed Hierarchical Linear Quantization method. Then, we discuss its integration with existing efficient quantization techniques. Finally, we describe the memory organization strategy and LUT design that enable efficient inference on edge devices.

4.1. Hierarchical Linear Quantization

As discussed previously, most PTQ methods, such as GPTQ and AWQ, adopt uniform quantization to facilitate hardware acceleration. However, uniform quantization cannot effectively represent the distribution of weights. Motivated by the empirical observation that weight distributions in LLMs tend to follow a bell-shaped curve (Kim et al., 2023), inspired by Binary Coding (Xu et al., 2018), we propose

Hierarchical Linear Quantization as an alternative approach. Specifically, for an n-dimensional weight vector W, its q-bit quantized representation is denoted as $\hat{\mathrm{W}}$:

$$\hat{\mathrm{W}} = \sum_{j=0}^{q-1} s_j \cdot b_j + z. \tag{3}$$

Here, b_j is a binary vector $\in \{0,1\}^n$,$s_j \in R$ is the quantization scale, and $z \in R$ is the zero-point. Similar to uniform quantization, given s and z, we present the quantization process of HLQ. For a q-bit quantization, we first generate all possible binary combinations, which form a codebook denoted by $C \in \{0,1\}^{2^q \times q}$. Based on this codebook, we construct a candidate set:

$$V = s \times C^T + z \tag{4}$$

Then, we define **Quantization** and **Dequantization** as follows:

Quantization:

$$k^* = \arg\min_k ||\mathrm{W} - V_k||, \quad B = C_{k^*} \tag{5}$$

Dequantization:

$$\hat{\mathrm{W}} = s \times B^T + z \tag{6}$$

It is worth noting that HLQ neither requires weight reconstruction nor introduces any additional factors, making it highly generalizable. It can be seamlessly integrated with various quantization methods. As a form of non-uniform quantization, it not only offers significant advantages in reducing weight representation error (see Section 4.2), but also introduces no additional computational overhead to Bit-serial LUT-based GEMM (see Section 5.4.4).

4.2. Weight Error Reduction via Hierarchical Linear Quantization

In this subsection, we introduce how to use Hierarchical Linear Quantization to reduce weight quantization error. Let $\hat{\mathrm{W}} = \mathrm{HLQ}(\mathrm{W}; s, z)$, The objective is to find the optimal set of q scales and a zero-point z to represent W. The optimization objective can be formulated as:

$$\arg\min_{s,z} ||\mathrm{W} - \hat{\mathrm{W}}||_2^2. \tag{7}$$

For this optimization problem, there are two possible approaches: a heuristic alternating optimization method and a gradient-based search method. The alternating optimization method features a simple computation flow and can efficiently obtain a locally convergent solution. In contrast, the gradient-based approach relies on backpropagation, which is computationally more expensive and is sensitive to the learning rates. Here, we just introduce Alternating optimization method. Gradient-based search method can be seen in Appendix A

Alternating optimization is a heuristic algorithm. At initialization, we set $z = \min(\mathrm{W})$, and the initial value of s is set to the scale factor used in uniform quantization. Specifically, let $\Delta = \frac{\max(\mathrm{W}) - \min(\mathrm{W})}{2^q - 1}$, For a q-bit quantization, the initial value of s is set to $s = [\Delta, 2\Delta, ...2^{q-1}\Delta]$. After initialization, we solve the minimization problem in formulation 7 through an alternating optimization procedure between **Bit-Pattern Selection** and **Linear Reconstruction**.

Bit-Pattern Selection. At t-th step, given the current scale parameters $s^{(t-1)}$ and zero-points $z^{(t-1)}$, we determine the optimal bit pattern $B^{(t)}$ that minimizes the reconstruction error under the fixed quantization parameters:

$$B^{(t)} = \arg\min_B \|\mathrm{W} - \mathrm{Dequant}(B, s^{(t-1)}, z^{(t-1)})\|^2, \tag{8}$$

where W denotes original pretrained weight, Dequant denotes dequantization method. This process is essentially equivalent to the one described in Equation 5, i.e., for each w, selecting the optimal bit combination from the candidate codebook.

Linear Reconstruction. Once the optimal bit pattern B_t is obtained, we fix B_t and update the continuous quantization parameters by solving a least-squares regression problem:

$$(s^{(t)}, z^{(t)}) = \arg\min_{s,z} \|\mathrm{W} - (B^{(t)} s + \mathbf{1} \cdot z)\|^2. \tag{9}$$

Here, adding a constant column of ones allows the zero-point to be incorporated directly into the linear system. This formulation enables joint optimization of scales and zero-point given a fixed discrete structure and reduces to a standard linear least-squares problem with a closed-form solution. Therefore, the optimization objective in Equation 9 can be equivalently expressed as follows:

$$(s^{(t)}, z^{(t)}) = \mathrm{LSE}(\mathrm{W}, B^{(t)}), \tag{10}$$

where LSE denotes the standard least squares estimation.

We summarize alternating optimization in Algorithm 1.

Figure 4 shows that HLQ significantly reduces the quantization error compared to uniform quantization, with the MSE of uniform quantization at the 1e-4 level, while HLQ decreases it to the 1e-6 level. In Table 4, we present an ablation study showing that simply replacing RTN with HLQ, without any calibration-based fine-tuning, can substantially improve model accuracy in both the 3-bit and 2-bit settings.

Algorithm 1 Alternating Optimization for Hierarchical Linear Quantization

Require: Weight matrix $\mathrm{W} \in R^{n\times k}$, bit width q, Group size g, Max iterations T_{max}
Ensure: scales $s \in R^{n\times \frac{k}{g}\times q}$, zero points $z \in R^{n\times \frac{k}{g}}$

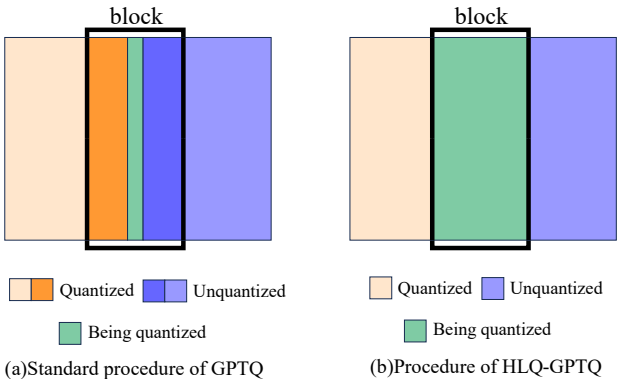
Reshape W into groups $\widetilde{\mathrm{W}} \in R^{n\times \frac{k}{g}\times g}$
Calculate uniform quantization scale : $\Delta \leftarrow \frac{\max(\widetilde{\mathrm{W}})-\min(\widetilde{\mathrm{W}})}{2^q-1}$
Initialize $s^{(0)} \leftarrow [\Delta, 2\Delta, ...2^{q-1}\Delta]$, $z^{(0)} \leftarrow \min(\widetilde{W})$
Generate binary combinations C
for $t \leftarrow 1$ to T_{max} **do**
  $\hat{\mathrm{W}}^{(\mathrm{t})} \leftarrow \widetilde{\mathrm{W}} - z^{(t-1)}$
  $V^{(t)} \leftarrow s^{(t-1)} \times C^T$ # Matrix product
  $k^* \leftarrow \arg\min_k \|\hat{\mathrm{W}}^{(\mathrm{t})} - V_k^{(t)}\|$
  $B^{(t)} \leftarrow C_{k^*}$ # Choose best binary combination
  $s^{(t)},\ z^{(t)} \leftarrow LSE(\widetilde{\mathrm{W}}, B^{(t)})$ # Least Squares Estimation
end for

4.3. Post-Training Quantization for HLQ

Here, using GPTQ as an example, we illustrate how HLQ can be employed to enhance existing post-training quantization methods. GPTQ, a widely adopted PTQ approach, offers strong hardware efficiency due to its use of group-wise uniform quantization and can quantize models within a very short time. However, it suffers from significant accuracy degradation under 2-bit quantization. By integrating HLQ into the GPTQ quantization pipeline, we effectively improve its performance in both 2-bit and 3-bit settings.

Figure 5 illustrates the integration of HLQ into the GPTQ pipeline. As shown in Figure 5(a), the standard GPTQ process partitions the weights into multiple column blocks and recursively quantizes each block column by column, using unquantized columns to compensate for quantization errors. After a block is fully quantized, subsequent unquantized blocks are leveraged to further correct accumulated errors. In contrast, Figure 5(b) presents our HLQ-GPTQ procedure. Instead of recursive column-wise quantization, our method directly applies HLQ to each column block as a whole, while retaining the error compensation mechanism through subsequent blocks. In this PTQ pipeline, we only replace the block-wise quantization step with HLQ, keeping all other components identical to GPTQ. Despite this straightforward modification, it substantially enhances GPTQ's performance in low-bit quantization scenarios (see Section 5.1.4).

Although integrating HLQ introduces additional parameter search overhead to GPTQ, this cost remains within a controllable range. Specifically, HLQ-GPTQ can still quantize the Llama2-7B model within half an hour using only 8 GB of GPU memory, thereby retaining the time and memory



(a)Standard procedure of GPTQ (b)Procedure of HLQ-GPTQ

Figure 5: Standard GPTQ VS HLQ-GPTQ.

efficiency characteristic of PTQ methods (see Section 5.3). It is worth noting that under 2-bit quantization, the performance of HLQ-GPTQ still lags behind state-of-the-art PTQ methods such as DB-LLM (Chen et al., 2024b). However, this performance gap primarily arises from fundamental differences in methodological complexity. Methods like DB-LLM typically incorporate additional procedures such as knowledge distillation, which require significantly more computational and memory resources and often take tens of times longer to complete quantization. In contrast, HLQ-GPTQ preserves the simplicity and hardware friendliness of the PTQ paradigm. More importantly, HLQ serves as a fundamental quantization format that is largely orthogonal to most post-training optimization techniques. This implies that by integrating existing PTQ refinements, the performance of HLQ under low-bit settings can be further improved.

4.4. Efficient Finetuning for HLQ

Previous quantization methods (Shao et al., 2023; Chen et al., 2024b; Li et al., 2021; Chen et al., 2024c) typically improve quantization performance by performing block-wise search for optimal quantization parameters or end-to-end finetuning. However, most of these methods are designed for uniform quantization and are not directly applicable to HLQ. As shown in figure 6, We propose an efficient finetuning scheme tailored for HLQ, which consists of two parts: **block-wise reconstruction** and **end-to-end tuning**.

The block-wise reconstruction aims to minimize the block output error. We adopt the same objective but replace uniform quantization with HLQ :

$$\arg\min_{s,z} ||\mathcal{F}(\mathbf{W}, \mathbf{X}) - \mathcal{F}\left(\mathrm{HLQ}(\mathbf{W}; s, z), \mathbf{X}\right)||, \quad (11)$$

where $\mathcal{F}$ denotes the mapping function of a Transformer block, $\mathbf{W}$ and $\mathbf{X}$ are the full-precision weights and activa-

tions, and HLQ represents the Hierarchical Linear Quantization function. To jointly consider local and block-level errors, we divide the block-wise reconstruction into two stages. In the first stage, we search for the optimal $\mathbf{W}_{int}, s, z$ for each linear layer within the block using the technique described in Algorithm 1. In the second stage, we fix $\mathbf{W}_{int}$ and only optimize the scale s and zero-point z. For each linear layer, this two-strategy preserves some local information while leveraging block-level context to further refine the parameters. Additionally, this two-stage scheme is also computationally efficient. In the first stage, parameter initialization for each linear layer is independent, allowing for batch-wise optimization. In the second stage, with $\mathbf{W}_{int}$ fixed, there is no need to access the original full-precision weights, further reducing computation and memory footprint. The experiments in Section 5.3 demonstrate the efficiency of this scheme.

After completing the block-wise reconstruction, we introduce end-to-end tuning, a training-based refinement approach. In this stage, we perform end-to-end optimization on the model using a calibration dataset. This technique is also adopted in EfficientQAT (Chen et al., 2024c), and following their practice, we update only the quantization scale s during training. To avoid overfitting, we limit the tuning process to only 1–2 epochs.

To reduce memory consumption on both the GPU and CPU, we implemented strategies including lazy loading, CPU offloading, and disk offloading (see Appendix F for details). As a result, ELUTQ can quantize the LLaMA 3.1-70B model on a system configured with 64 GB of CPU RAM and two NVIDIA RTX 4090 GPUs (48 GB VRAM).

4.5. Deploy with Edge Framework

In this subsection, we present the core design of the C++ inference framework tailored for Hierarchical Linear Quantization.

4.5.1. Weight-Rearrange.

Conventional dequantization-based inference frameworks typically store weights in either row-major or column-major order. Some works, such as AWQ, employ interleaved weight storage for 4-bit quantization to accelerate runtime decoding, but the approach fundamentally remains column-major.

Bit-serial LUT-based GEMM operates by loading weights corresponding to activation combinations and performing table lookups to compute results. This requires careful consideration of the weight organization in memory. Taking activation groups of size g=4 as an example, Figure 7 illustrates our design for rearranging the one-bit weight matrics to fit the 128-bit ARM NEON registers and efficiently unpacking them at runtime.

Given ARM's 128-bit register width, a 16×8 one-bit matrix block stored in conventional row-major or column-major order would reside in non-contiguous memory. To maximize memory bandwidth utilization, we reorganize such blocks along the out-channel dimension, ensuring contiguous memory access. Since weights remain static during inference, this rearrangement can be performed offline, introducing no runtime overhead. During execution, weight decoding only requires simple bitwise AND and shift operations, maintaining computational efficiency.

4.5.2. Mirror Storage.

Given the formulation:

$$\hat{\mathrm{W}} = \sum_{j=0}^{q-1} s_j \cdot b_j + z, b_j \in \{0,1\}^n, \tag{12}$$

we apply a simple linear transformation by setting $\hat{s}_j = \frac{1}{2} s_j$, $\hat{b}_j = 2b_j - 1$, $\hat{z} = z + \frac{1}{2}\sum_{j=0}^{q-1} s_j$, under this transformation, the quantized weights can be rewritten as:

$$\hat{\mathrm{W}} = \sum_{j=0}^{q-1} \hat{s}_j \cdot \hat{b}_j + \hat{z}, \hat{b}_j \in \{-1,1\}^n. \tag{13}$$

For an input activation combinations of size 4, such as $[x_1, x_2, x_3, x_4]$, the output of the dot product with the weight has 16 possible outcomes, ranging from $(-x_1 - x_2 - x_3 - x_4,\ ...,\ x_1 + x_2 + x_3 + x_4)$. When storing the lookup table, we only need to store half of the possible results, as the remaining half can be obtained by negating the stored values. This table compression method is lossless, fully preserving model inference accuracy while also reducing memory usage by half and accelerating table access.

4.5.3. Table Quantization.

For ARM NEON, activations are typically stored in FP16 precision, and accordingly, each entry in the lookup table also uses FP16 precision. To optimize table storage and access, each FP16 value can be split into two int8 values and interleaved in memory. During table loading, efficient dual-channel load operations such as $vld2q\_u8$ can be employed to construct the table, after which the retrieved values are reconstructed back into FP16.

An alternative approach is to quantize the table itself, mapping FP16 table values to int8. While this may introduce some degradation in model accuracy, it eliminates the need for interleaved storage and FP16 reconstruction, significantly improving runtime efficiency.

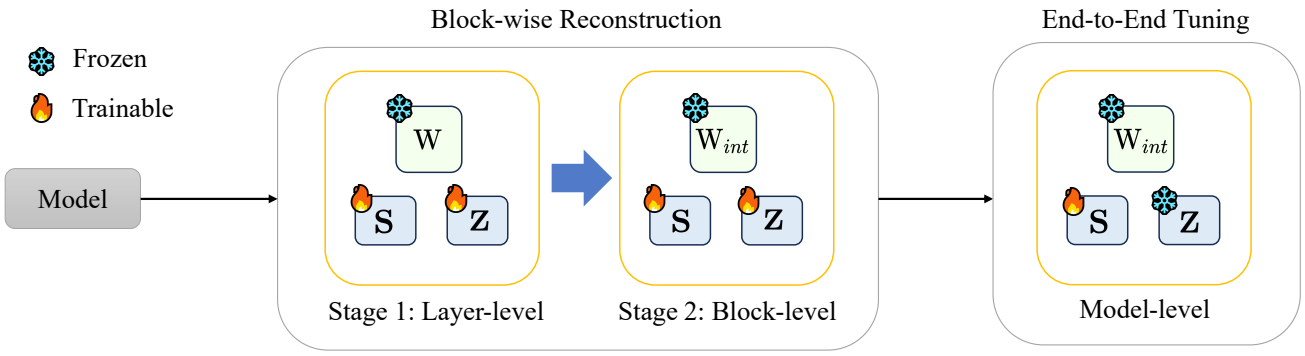


Figure 6: The pipeline of efficient finetuning for HLQ.

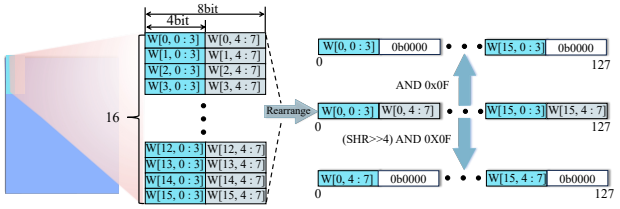


Figure 7: Rearrange weight for ARM NEON's 128-bit registers to ensure memory access continuity and improve decoding speed during runtime.

5. Experiments

Our experiments consist of three parts. First, we evaluate the performance of the proposed hierarchical linear quantization method under both PTQ and efficient finetuning settings. Second, we assess the time and memory efficiency of our quantization approach. Finally, we evaluate our CPU kernel and the end-to-end inference performance of the quantized model on edge hardware.

5.1. Main Results

5.1.1. Models, Datasets and Baseline.

We conduct experiments on several widely used models, including LLaMA2 (Touvron et al., 2023), LLaMA3.1 (Dubey et al., 2024), Qwen3 (Yang et al., 2025). We use the C4 (Raffel et al., 2020) dataset as the calibration set. For 3-bit quantization, Our results are compared with weight-only methods employing uniform quantization schemes, such as GPTQ (Frantar et al., 2022), AWQ (Lin et al., 2024), and OmniQuant (Shao et al., 2023), ShiftAddLLM (You et al., 2024), AnyBCQ (Park et al., 2025a), GANQ (Zhao & Yuan, 2025) and SqueezeLLM (Kim et al., 2023). For 2-bit quantization, we additionally compare our method with several efficient mixed-precision quantization approaches, including PB-LLM (Shang et al., 2023), DB-LLM (Chen et al., 2024b) and ApiQ (Liao et al., 2024). In Appendix C, we also compare our method with codebook-based non-uniform quantization approaches such as Quip# (Tseng et al., 2024) and AQLM (Egiazarian et al., 2024), as well as QAT method.

5.1.2. Evaluation.

We evaluate all models using perplexity on the Wikitext2 (Merity et al., 2016) and C4 dataset. Additionally, we assess their zero-shot capabilities using the LM Harness framework (Gao et al., 2021). The evaluation tasks include ARC Easy (Clark et al., 2018), ARC Challenge (Clark et al., 2018), HellaSwag (Zellers et al., 2019), WinoGrande (Sakaguchi et al., 2021), and PIQA (Bisk et al., 2020).

5.1.3. Configuration.

We evaluate weight-only quantization at two precision levels: INT3 and INT2, while keeping activations in full precision. For HLQ-GPTQ, following previous PTQ settings, we select 128 calibration samples from the C4 dataset. we set the maximum number of alternating optimization step T_{max}=10. For efficient finetuning, we randomly select 1K calibration samples from the C4 dataset. In the block-wise stage, we set the learning rate to 1×10^{-4} and train for 2 epochs, while in the end-to-end stage, the learning rate is set to 2×10^{-5} with 1 training epochs.

5.1.4. Perplexity Results.

Table 1 presents the perplexity results on the C4. The results on Wikitext2 can be found in Appendix G.

For HLQ-GPTQ, our method enhances the performance of GPTQ and surpasses other PTQ approaches with uniform quantization. Under the 3-bit configuration, compared to GPTQ, our approach achieves a perplexity reduction of 0.15 for the Llama2 models and 0.8 for Llama3.1-8B. In the 2-bit setting, our method substantially improves upon GPTQ, particularly for the Llama3.1-8B model, where perplexity on C4 is reduced from 181.82 to 25.44. Although 2-bit results do not surpass those of DB-LLM, we consider

Table 1: A comparison of c4 perplexity (↓) between weight-only quantization methods, with a context length of 2048. **BPW** represents the average number of bits per weight. Scale and zero-point are assumed to be stored in fp16 format. PB-LLM* denotes the result of PB-LLM with GPTQ and 20% salient weight. ”–” indicates that the framework does not support this model. The results on Wikitext2 can be found in Appendix G.

Method	# W	# G	BPW	L2-7	L2-13	L3.1-8	L3.1-70	Q3-8	Q3-14	Q3-32
Baseline	16	-	16	6.97	6.47	8.89	6.92	13.30	12.02	10.81
GPTQ	2	128	2.25	33.70	20.97	181.82	22.76	35.57	26.73	24.75
AWQ	2	128	2.25	1.72e5	9.41e4	2.14e6	1.57e5	2.52e6	1.28e7	5.47e7
OmniQuant	2	128	2.25	15.02	11.05	31.76	21.79	-	-	-
ApiQ	2	128	2.25	12.04	9.72	-	-	-	-	-
PB-LLM*	-	-	2.2	20.60	15.32	57.33	34.52	-	-	-
ShiftAddLLM	2	128	2.37	14.60	9.85	21.97	18.16	-	-	-
AnyBCQ	2	128	2.37	-	-	21.64	-	-	-	-
HLQ-GPTQ	2	128	2.37	14.89	9.75	25.44	19.92	24.06	18.30	14.25
HLQ-Finetuning	2	128	2.37	**11.21**	**9.43**	**15.87**	**11.24**	**19.46**	**15.94**	**12.98**
DB-LLM	2	64	-	9.62	8.38	19.20	-	-	-	-
HLQ-GPTQ	2	64	2.75	13.27	9.24	20.52	17.31	20.82	16.08	13.48
HLQ-Finetuning	2	64	2.75	**9.12**	**8.02**	**14.43**	**10.21**	**17.68**	**15.05**	**12.76**
GPTQ	3	128	3.25	7.89	7.00	11.66	8.18	14.39	12.93	11.78
AWQ	3	128	3.25	7.84	6.94	11.58	8.32	18.51	12.91	11.92
OmniQ	3	128	3.25	7.75	6.98	11.66	8.12	-	-	-
SqueezeLLM(0.45%)	3	-	3.25	7.51	6.82	-	-	-	-	-
GANQ	3	-	3.25	7.51	-	10.95	-	-	-	-
ShiftAddLLM	3	128	3.50	7.70	6.92	11.14	7.78	-	-	-
AnyBCQ	3	128	3.50	-	-	12.23	-	-	-	-
HLQ-GPTQ	3	128	3.50	7.74	6.90	10.80	7.96	14.15	12.79	11.49
HLQ-Finetuning	3	128	3.50	**7.45**	**6.77**	**10.47**	**7.42**	**13.54**	**12.36**	**11.24**

this reasonable. DB-LLM incorporates a complex knowledge distillation process, which demands substantial computational resources and time for finetuning. Moreover, its quantization format is less amenable to practical hardware deployment. In contrast, HLQ-GPTQ preserves the simplicity of PTQ, requiring minimal computational resources and time to complete model quantization.

For HLQ-Finetuning, our method further improves model performance under low-bit quantization, particularly in the 2-bit setting. For the W2g128 configuration of Llama3.1-8B, the perplexity on C4 is further reduced to 15.87. Moreover, under the W2g64 configuration, our method surpasses DB-LLM, while requiring only 1K calibration samples compared to 20K used by DB-LLM.

5.1.5. Zero-Shot Tasks.

Table 2 presents the experimental results on five zero-shot tasks. Under 3-bit quantization, HLQ-GPTQ and HLQ-finetuning all improve the average accuracy by approximately 1% compared to GPTQ on Llama3.1-8B. For 2-bit quantization, HLQ-GPTQ achieves an improvement of roughly 9% over GPTQ on Llama3.1-8B, also outperforming AWQ and OmniQuant, while HLQ-Finetuning achieves an improvement of 20%.

5.1.6. Compression Rates.

It can be observed that under the same weight precision setting, HLQ incurs a higher average bit-width compared to uniform quantization. For example, at 2-bit, the average bit-width is 2.25 for uniform quantization versus 2.37 for HLQ; at 3-bit, it is 3.25 for uniform quantization versus 3.5 for HLQ. This is because HLQ requires storing a separate scale for each bit plane. Table 3 presents the model compression rates. For LLaMA3.1-8B, HLQ at 2-bit requires an additional 0.1 GB of storage space compared to uniform quantization—an increase of approximately 5%. However, this storage overhead does not introduce additional computational cost in Bit-Serial LUT-based GEMM, enabling end-to-end inference performance on par with uniform quantization. We will provide a detailed analysis of this aspect in Section 5.4.4.

Table 2: Average accuracy (%) on zero-shot tasks by lm_eval v0.4.9. PB-LLM* denotes the result of PB-LLM with GPTQ and 20% salient weight. **Bold**: best result; underlined: second-best. "–" indicates that the framework does not support this model. Full results can be found in Appendix G. Acc is reported , not acc norm.

Method	#W	#G	L2-7	L2-13	L3.1-8	L3.1-70	Q3-8	Q3-14	Q3-32B
FP16	16	-	64.13	67.81	69.25	75.33	68.10	71.29	72.15
GPTQ	3	128	61.47	66.18	64.21	71.28	63.04	67.28	66.44
AWQ	3	128	62.23	66.14	64.98	72.13	59.65	62.31	67.10
OmniQ	3	128	61.66	66.18	-	-	-	-	-
HLQ-GPTQ	3	128	62.32	66.44	65.02	72.48	65.64	69.05	67.45
HLQ-Finetuning	3	128	**62.52**	**66.62**	**65.22**	**73.13**	**66.41**	**70.03**	**68.21**
GPTQ	2	128	40.66	48.29	35.98	45.57	39.45	48.96	50.34
OminQ	2	128	46.98	53.56	-	-	-	-	-
PB-LLM*	2	128	37.60	38.39	-	-	-	-	-
HLQ-GPTQ	2	128	49.04	58.28	44.35	56.47	53.04	60.03	65.81
HLQ-Finetuning	2	128	**55.07**	**59.25**	**57.79**	**66.60**	**59.44**	**64.99**	**67.57**

Table 3: Comparison of Model Compression Rates between Uniform Quantization and Hierarchical Linear Quantization on Llama3.1-8B. Embedding and LM-head layers are excluded from quantization for all models. The symbol '-' denotes per-channel quantization. "Rates(↑)" denote the compression ratio, and "Mem(↓)" indicates the size of the compressed model. A comprehensive presentation of the results is provided in Appendix B.

Model	Wbit	G	Uniform		HLQ	
			Rates	Mem(GB)	Rates	Mem(GB)
LLaMA3.1-8B	2	128	3.95	3.785	3.84	3.887
	3	128	3.25	4.598	3.11	4.801

Table 4: Comparison between HLQ and RTN quantization across multiple model under w3g128 and w2g128 configurations. Results are reported in terms of C4 perplexity, highlighting the standalone contribution of HLQ over standard RTN.

Models	Configuration	RTN	HLQ
LLaMA2-7B	w3g128	28.24	**9.04**
	w2g128	1.10e5	**450.06**
LLaMA2-13B	w3g128	13.37	**7.52**
	w2g128	5.75e4	**765.76**
Qwen3-8B	w3g128	541.95	**17.60**
	w2g128	6.52e6	**829.91**
Qwen3-14B	w3g128	579.17	**15.43**
	w2g128	2.72e6	**3432.17**

5.2. Ablation Study.

5.2.1. Ablation Study on HLQ vs. RTN Quantization

To better understand the isolated contribution of HLQ beyond standard RTN quantization, we conduct a comprehensive ablation study across multiple model scales and architectures. Specifically, we evaluate LLaMA2-7B, LLaMA2-13B, Qwen3-8B, and Qwen3-14B under both W3G128 and W2G128 settings, and only apply HLQ without any tuning technicals. As shown in Table 4, using HLQ alone yields substantial perplexity improvements under both 3-bit and 2-bit settings, demonstrating its effectiveness as a drop-in replacement for RTN.

5.2.2. Hyparameter Study for HLQ-GPTQ.

We conduct a parameter search experiment on the step T_{max} of the alternating optimization in HLQ-GPTQ and investigate its impact on the final model performance. The experiments are carried out on LLaMA3.1-8B and Qwen3-8B, covering both w3g128 and w2g128 settings. The results are shown in the figure 8. We observe that increasing T_{max} does not necessarily lead to continuous improvement. For both 3-bit and 2-bit quantization, setting T_{max} = 10 yields a relatively good perplexity across both models, suggesting that a moderate optimization depth strikes a balance between convergence and stability. Specifically, for 3-bit quantization, the perplexity decreases steadily as T_{max} increases up to 10 and then stabilizes, indicating that the alternating optimization approaches a convergence point. In contrast, the 2-bit setting exhibits a more irregular pattern, with notable oscillations in perplexity as T_{max} increases. Perhaps incorporating certain regularization techniques could help the results converge further, which we leave for future work.

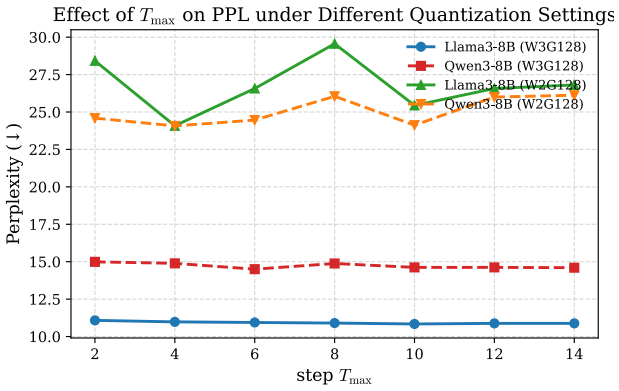


Figure 8: Effect of T_{max} on perplexity under different quantization configurations. The figure shows the variation of perplexity with respect to T_{max} for Llama3-8B and Qwen3-8B models under two quantization settings (W3G128 and W2G128).

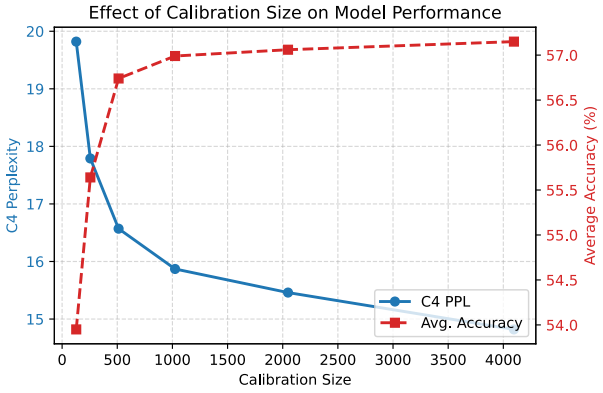


Figure 9: Model performance of w2g128 Llama3.1-8b with different calibration size when finetuning.

5.2.3. Calibration Data Size for Efficient Finetuning.

We investigated the effect of the size of the calibration data on the final performance. As illustrated in Figure 9, our method yields substantial improvements as the calibration set size increases when it is below 1k. However, once the size exceeds 1k, the improvement becomes marginal. Since enlarging the calibration set introduces considerable memory overhead, especially during the block-wise stage, we set the calibration set size to 1k, thereby striking a balance between performance and efficiency.

5.2.4. Impact of Each Components in Efficient Finetuning.

Finetuning consists of two main stages: block-wise quantization and end-to-end tuning. We investigate the impact of each stage on the final model performance. The results are presented in Table 5.

Table 5: Impact of each component in Finetuning on Llama3.1-8B w2g128 quantization.

Block-Wise	E2E-Finetuning	c4 PPL(↓)	avg.acc(↑)
✗	✗	1937.08	30.11
✓	✗	17.49	53.78
✗	✓	583.75	35.42
✓	✓	**15.87**	**57.79**

5.3. Efficiency Evaluation.

5.3.1. Time and Memory Efficiency.

Table 6 reports the quantization time and peak GPU memory usage of HLQ-GPTQ and HLQ-Finetuning. HLQ-GPTQ is highly efficient, requiring only 0.15–0.40 hours for Llama2-7B and Llama2-13B, and 0.20 hours for Llama3.1-8B, with GPU memory usage between 7 and 10 GB. For Llama3.1-70B, it completes in 4 hours using 18 GB of memory, which fits easily within a single RTX 4090. HLQ-Finetuning also remains efficient. It uses 7–17 GB of memory for smaller models and 40 GB for Llama3.1-70B, and finishes in 1.5–4.5 hours for Llama2-7B and Llama2-13B, and 20 hours for Llama3.1-70B. Overall, both methods scale to large models while keeping memory usage low, making 2-bit and 3-bit quantization feasible on commodity GPUs.

Table 7 summarizes the quantization time and memory consumption of several 2-bit quantization pipelines on Llama3.1-70B during block reconstruction. Methods such as GPTQ and AWQ run relatively quickly but demand very large amounts of CPU memory. More advanced approaches, including OmniQ, Quip#, AQLM, DB-LLM, and EfficientQAT, require long processing times ranging from tens to hundreds of hours and extremely high CPU memory. Several of these methods also exceed the VRAM capacity of four RTX 4090 GPUs. In comparison, HLQ-GPTQ completes in 4 hours with 64 GB of CPU memory and 18 GB of GPU memory, offering a significantly more practical and resource-efficient solution. HLQ-Finetuning shows similar advantages: it keeps the same low memory requirements and completes within 20 hours, outperforming existing finetuning-based quantizers. These results demonstrate that HLQ is an efficient and scalable framework for large-scale 2-bit quantization under realistic hardware constraints.

Table 6: Quantization time and peak GPU memory usage of HLQ-GPTQ and HLQ-Finetuning.

Model	HLQ-GPTQ		HLQ-Finetuning	
	T	Mem(3/2 bit)	T	Mem(3/2 bit)
llama2-7	0.15h	8/7GB	1.5h	9/7GB
llama2-13	0.40h	10/9GB	4.5h	17/13GB
llama3.1-8	0.20h	9/8GB	2h	10/8GB
llama3.1-70	4.00h	24/18GB	20h	40/32GB

Table 7: Comparison of quantization time and peak memory usage on W2g128 Llama3.1-70B across different quantization methods during block reconstruction. The results of Quip and Aqlm are inferred from LLaMA2-70B using 4k calibration data. >96 indicates that the experiment could not be conducted on four RTX 4090 GPUs. The detailed experimental configurations for all methods can be found in Appendix D.

Method	Time(h)	CPU Mem(GB)	GPU Mem(GB)
GPTQ	2.50	180	28
AWQ	5	180	24
OmniQ	40	240	40
HLQ-GPTQ	4	64	18
Quip#	260	1200	>96
Aqlm	300	950	>96
DB-LLM	72	1200	>96
EfficientQAT	50	300	30
HLQ-Finetuning	20	64	18

5.3.2. Data Efficiency.

Compared with previous approaches, our method demonstrates substantially higher data efficiency. For HLQ-GPTQ, we follow prior works and use only 128 calibration samples. For HLQ-Finetuning, although our final experiments adopt a calibration set size of 1k, even with only 128 samples our method still outperforms previous uniform quantization methods. Methods such as DB-LLM require up to 20k calibration samples, and Quip# and AQLM require 4k samples, whereas our approach achieves competitive performance with only 1k calibration samples, substantially reducing the need for calibration data.

5.4. Hardware Deployment on Edge Devices

5.4.1. Hardware Devices.

In this section, we primarily evaluate the inference performance of our framework on edge devices built on the ARM architecture. ARM-based processors are widely adopted in edge computing due to their highly integrated System-on-Chip (SoC) design and cache-friendly characteristics, which are advantageous for table-lookup–based operations. Moreover, ARM processors are well known for their low power consumption, making them a common choice in resource-constrained edge scenarios. For evaluation, we select two representative ARM-based chips: the Apple M2 from the Apple Silicon series and the RK3588 based on the ARM Cortex architecture. However, since HLQ can be converted into the BCQ format used in ShiftAddLLM (You et al., 2024) and AnyBCQ (Park et al., 2025a) through simple linear transformations, our quantization format is also compatible with GPU inference. We additionally provide experimental results on GPUs (see Appendix E.2 for details).

5.4.2. Evaluation Setup.

In terms of hardware evaluation, our experiments focus on the following aspects:

- Numerical precision evaluation: Since our optimization algorithm is executed on the GPU, deploying the optimized model on edge devices may introduce numerical precision errors. Therefore, we evaluate the numerical accuracy of our framework. The experimental results demonstrate that the output of our framework achieves a cosine similarity greater than 99% compared to the GPU output, meeting the requirements for practical application.
- Overhead of HLQ: we follow T-MAC's design and adapt it to support HLQ. Experimental results show that the introduction of HLQ incurs no additional overhead to the Bit-serial LUT-based GEMM kernel.
- End-to-End Speedup Comparison with AWQ: We compare our inference framework with AWQ for end-to-end speedup.
- Compared with llama.cpp: We further compare our framework with the state-of-the-art edge-side inference framework, llama.cpp, to demonstrate the superiority of our method.

5.4.3. Numerical Precision Evaluation.

As mentioned in TensorRT (NVIDIA, 2023) and the MLPerf (Reddi et al., 2020), the cosine similarity between GPU and CPU outputs needs to be greater than 99% to meet the application standards. For the LLaMA2-7B model with 3-bit weight quantization, we select the output hidden states from the 9th, 18th, and 31st (final) layers on the M2 chip and compare them with the corresponding outputs on the GPU. The hidden states of the GPU denote $\mathbf{a}$, and the hidden states of the M2 denote $\mathbf{b}$. The cosine similarity is computed as

$$\cos\_\text{sim}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

Results in Table 8 demonstrate that the cosine similarity between our framework and the GPU simulation exceeds 99.9%.

Table 8: Cosine similarity(%) between simulated quantization on GPU and actual quantization on the M2 chip at the 9th, 18th, and 31st(final) layers of the LLaMA2-7B model with 3-bit weight quantization.

Model	Prompt	L9	L18	L31
LLaMA2-7B-3Bit	128	99.9994	99.9990	99.9982
	256	99.9984	99.9986	99.9970
	512	99.9988	99.9978	99.9960

5.4.4. Overhead of HLQ.

We modified the T-MAC operators to support the HLQ format and evaluated their performance at the kernel level. Table 9 presents the latency results of two types of matrix operations in the 2-bit quantized LLaMA2-7B model, evaluated on both the RK3588 and Apple M2 chips. For each operation, the reported latency is the average of 10 runs. The results demonstrate that the introduction of HLQ does not introduce additional overhead to the Bit-serial LUT-based GEMM. The increase in average latency is negligible compared to the inherent fluctuations and overall runtime. Therefore, although the HLQ quantization format occupies slightly more memory than uniform quantization under the same bit-width and group configuration, it does not introduce any additional computational overhead during inference. Moreover, the extra memory required by HLQ is small relative to the overall size of the compressed model, typically remaining below 5%, which is considered acceptable in practice.

Table 9: Comparison of GEMM latency (ms) between uniform quantization and HLQ formats using the T-MAC kernel.

Chip	Kernel	11008×4096	4096×32000
RK3588	T-MAC-Uniform	$5.70\ (\pm 0.10)$	$22.11\ (\pm 0.17)$
	T-MAC-HLQ	$5.72\ (\pm 0.11)$	$22.23\ (\pm 0.16)$
M2	T-MAC-Uniform	$1.84\ (\pm 0.07)$	$4.46\ (\pm 0.05)$
	T-MAC-HLQ	$1.86\ (\pm 0.04)$	$4.49\ (\pm 0.05)$

5.4.5. End-to-End Speedup Comparison with AWQ

Here, we compare ELUTQ with the uniform-quantization-based inference framework AWQ in terms of end-to-end speedup on the LLaMA3.1-8B model. All experiments are conducted on the Apple M2 chip. As shown in Table 10, under the 3-bit setting, AWQ achieves a 2.0× speedup over FP16, while ELUTQ achieves 2.5×. Under the 2-bit setting, AWQ achieves 1.6×, whereas ELUTQ reaches 3.4×. These results indicate that AWQ suffers from significant dequantization overhead at lower bit widths, resulting in reduced speedup at 2 bits, whereas our framework continues to improve efficiency as the bit width decreases.

Table 10: End-to-end performance comparison of ELUTQ and AWQ on LLaMA3.1-8B running on Apple M2 (batch size = 1, prompt length = 512).

Method	W	G	Mem(GB)	Speedup
Baseline	16	-	15.0	1.0×
AWQ	3	128	4.7	2.0×
ELUTQ	3	128	4.8	**2.5**×
AWQ	2	128	3.9	1.6×
ELUTQ	2	128	3.9	**3.4**×

5.4.6. Compared with llama.cpp.

Directly comparing inference speed with llama.cpp is not entirely fair, since llama.cpp adopts its own quantization format, GGUF, which is incompatible with our quantization scheme. To ensure a fair comparison, we instead use bits per weight (BPW) to measure model size, focusing on how inference speed changes as the model size varies.

Table 11 presents the results on LLaMA2-7B and Qwen2-1.5B. We observe that under lower-bit settings, llama.cpp becomes slower as the model size decreases. For example, Q3_K_S is slower than Q3_K_M and Q2_K_S is slower than Q3_K_S. This phenomenon mainly arises from the substantial decoding overhead associated with low-bit quantization. In contrast, our framework benefits from the Bit-serial LUT-based GEMM paradigm, achieving nearly linear improvements in inference speed as the average bit-width decreases. Specifically, when BPW = 3.5, ELUTQ outperforms llama.cpp by approximately 20% in the prefill stage and 10% in the decode stage. When BPW $\approx$ 2.5, the speedup increases to about 45% and 25%, respectively.

Moreover, the improvement is more pronounced in the prefill stage than in the decode stage. This is because the prefill stage is computation-intensive and primarily involves GEMM operations, while the decode stage is memory-intensive and mainly consists of GEMV operations. In the prefill stage, our method replaces multiplication in matrix multiplication with efficient lookup and sum operations, significantly accelerating computation. In contrast, the decode stage is memory-intensive, where runtime is dominated by memory access. Since the Bit-serial LUT-based GEMM requires additional table lookups, its memory access cost is slightly higher than that of traditional GEMV, leading to smaller speed gains in the decode stage.

Table 11: Comparison of prefill and decode throughput (tokens/s) between ELUTQ and llama.cpp on the Apple M2 chip. All experiments are conducted with 4 threads and a batch size of 1. More results can be found in Appendix E.1

Model	Framework	Wbits	BPW	Input,Output	Prefill	Decode
llama2-7B	llama.cpp	Q3_K_M	3.91	128,128	19.87	15.65
				256,128	19.75	15.53
		Q3_K_S	3.50	128,128	17.60	15.44
				256,128	17.43	15.00
		Q2_K_S	2.50	128,128	14.82	13.35
				256,128	14.54	13.16
	ELUTQ	W3(g128)	3.50	128,128	20.60	16.63
				256,128	21.54	16.21
		W2(g128)	2.37	128,128	25.06	20.48
				256,128	25.94	20.25

6. Limitations and Discussion

Here, we discuss the limitations of our work and outline several promising directions for future research.

- **Weight-only quantization.** The current study focuses exclusively on weight only quantization. Extending our framework to weight activation quantization could further enhance inference efficiency and reduce memory consumption. This remains an open and valuable direction for future research.
- **Integration with other quantization frameworks.** At present, HLQ has been integrated only with a limited set of efficient quantization methods. Combining HLQ with Quantization-aware Training or LoRA-based adaptation may lead to additional gains in quantization accuracy, which we plan to explore in subsequent work.

7. Conclusion

In this paper, we propose **ELUTQ**, an efficient quantization framework designed for deploying large language models on edge devices. The framework is carefully aligned with the computation pipeline of Bit-serial LUT-based GEMM and introduces a novel *Hierarchical Linear Quantization* method that better captures the weight distribution compared to traditional uniform quantization. Moreover, HLQ can be seamlessly integrated with existing quantization techniques, including post-training and fine-tuning, to further enhance model accuracy. To reduce both GPU and CPU memory consumption, we further optimize the model and data loading pipeline in the quantization process, enabling quantization algorithms to operate efficiently under limited computational resources. Finally, we develop a pure C++ inference framework that enables accurate and efficient on-device inference, facilitating the practical deployment of quantized models in edge scenarios.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 92473208), the Tianjin Science and Technology Planning Program (Grant No. 24ZYCGYS00680), and in part by the Tianjin Key Laboratory of Optical-electronic Sensor and Sensor Network Technology.

References

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.

Chee, J., Cai, Y., Kuleshov, V., and De Sa, C. M. Quip: 2-bit quantization of large language models with guarantees. *Advances in Neural Information Processing Systems*, 36: 4396–4429, 2023.

Chen, H., Lv, C., Ding, L., Qin, H., Zhou, X., Ding, Y., Liu, X., Zhang, M., Guo, J., Liu, X., and Tao, D. DB-LLM: Accurate dual-binarization for efficient LLMs. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 8719–8730, 2024a.

Chen, H., Lv, C., Ding, L., Qin, H., Zhou, X., Ding, Y., Liu, X., Zhang, M., Guo, J., Liu, X., et al. Db-llm: Accurate dual-binarization for efficient llms. *arXiv preprint arXiv:2402.11960*, 2024b.

Chen, M., Shao, W., Xu, P., Wang, J., Gao, P., Zhang, K., and Luo, P. Efficientqat: Efficient quantization-aware training for large language models. *arXiv preprint arXiv:2407.11062*, 2024c.

Chen, T. mlc-llm. https://github.com/mlc-ai/mlc-llm, 2023. Accessed: July 31, 2025.

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 preprint arXiv:1803.05457*, 2018.

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. GPT3.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=dXiGWqBoxaD.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023a.

Dettmers, T., Svirschevski, R., Egiazarian, V., Kuznedelev, D., Frantar, E., Ashkboos, S., Borzunov, A., Hoefler, T., and Alistarh, D. Spqr: A sparse-quantized representation for near-lossless llm weight compression. *arXiv preprint arXiv:2306.03078*, 2023b.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 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.

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., Biderman, S., Black, S., DiPofi, A., Foster, C., Golding, L., Hsu, J., McDonell, K., Muennighoff, N., et al. A framework for few-shot language model evaluation. *Version v0. 0.1. Sept*, 10:8–9, 2021.

Gerganov, G. llama.cpp. https://github.com/ggml-org/llama.cpp, 2023. Accessed: July 31, 2025.

Huang, W., Liu, Y., Qin, H., Li, Y., Zhang, S., Liu, X., Magno, M., and Qi, X. Billm: Pushing the limit of post-training quantization for llms. In *ICML*, 2024. URL https://openreview.net/forum?id=qOl2WWOqFg.

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.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Li, Y., Gong, R., Tan, X., Yang, Y., Hu, P., Zhang, Q., Yu, F., Wang, W., and Gu, S. Brecq: Pushing the limit of post-training quantization by block reconstruction. *arXiv preprint arXiv:2102.05426*, 2021.

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. *arXiv preprint arXiv:2310.08659*, 2023.

Liao, B., Herold, C., Khadivi, S., and Monz, C. Apiq: Finetuning of 2-bit quantized large language model. *arXiv preprint arXiv:2402.05147*, 2024.

Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.-M., Wang, W.-C., Xiao, G., Dang, X., Gan, C., and Han, S. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of machine learning and systems*, 6:87–100, 2024.

Liu, Z., Cheng, K.-T., Huang, D., Xing, E. P., and Shen, Z. Nonuniform-to-uniform quantization: Towards accurate quantization via generalized straight-through estimation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 4942–4952, 2022.

Liu, Z., Zhao, C., Fedorov, I., Soran, B., Choudhary, D., Krishnamoorthi, R., Chandra, V., Tian, Y., and Blankevoort, T. Spinquant: Llm quantization with learned rotations. *arXiv preprint arXiv:2405.16406*, 2024.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

NVIDIA. Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, 2023. Accessed: July 31, 2025.

NVIDIA. Deeplearningexamples: Deep learning reference scripts and models. https://github.com/NVIDIA/DeepLearningExamples, 2023. Accessed: July 30, 2025.

Park, G., Park, B., Kim, M., Lee, S., Kim, J., Kwon, B., Kwon, S. J., Kim, B., Lee, Y., and Lee, D. Lut-gemm: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022.

Park, G., Bae, J., Kwon, B., Kim, B., Kwon, S. J., and Lee, D. Anybcq: Hardware efficient flexible binary-coded quantization for multi-precision llms. *arXiv preprint arXiv:2510.10467*, 2025a.

Park, G., Kwon, H., Kim, J., Bae, J., Park, B., Lee, D., and Lee, Y. Figlut: An energy-efficient accelerator design for fp-int gemm using look-up tables. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1098–1111, 2025b.

Park, Y., Hyun, J., Cho, S., Sim, B., and Lee, J. W. Any-precision llm: low-cost deployment of multiple, different-sized llms. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24, 2024.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21 (140):1–67, 2020.

Reddi, V. J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., Chukka, R., Coleman, C., Davis, S., Deng, P., Diamos, G., Duke, J., Fick, D., Gardner, J. S., Hubara, I., Idgunji, S., Jablin, T. B., Jiao, J., John, T. S., Kanwar, P., Lee, D., Liao, J., Lokhmotov, A., Massa, F., Meng, P., Micikevicius, P., Osborne, C., Pekhimenko, G., Rajan, A. T. R., Sequeira, D., Sirasao, A., Sun, F., Tang, H., Thomson, M., Wei, F., Wu, E., Xu, L., Yamada, K., Yu, B., Yuan, G., Zhong, A., Zhang, P., and Zhou, Y. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, pp. 446–459, 2020.

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.

Shang, Y., Yuan, Z., Wu, Q., and Dong, Z. Pb-llm: Partially binarized large language models, 2023. URL https://arxiv.org/abs/2310.00034.

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.

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*, 2023.

Tseng, A., Chee, J., Sun, Q., Kuleshov, V., and De Sa, C. Quip#: Even better llm quantization with hadamard incoherence and lattice codebooks. *arXiv preprint arXiv:2402.04396*, 2024.

Wei, J., Cao, S., Cao, T., Ma, L., Wang, L., Zhang, Y., and Yang, M. T-mac: Cpu renaissance via table lookup for low-bit llm deployment on edge. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 278–292, 2025.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International conference on machine learning*, pp. 38087–38099, 2023.

Xu, C., Yao, J., Lin, Z., Ou, W., Cao, Y., Wang, Z., and Zha, H. Alternating multi-bit quantization for recurrent neural networks. *arXiv preprint arXiv:1802.00150*, 2018.

Xu, Y., Xie, L., Gu, X., Chen, X., Chang, H., Zhang, H., Chen, Z., Zhang, X., and Tian, Q. Qa-lora: Quantization-aware low-rank adaptation of large language models. *arXiv preprint arXiv:2309.14717*, 2023.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yao, Z., Yazdani Aminabadi, R., Zhang, M., Wu, X., Li, C., and He, Y. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in neural information processing systems*, 35: 27168–27183, 2022.

You, H., Guo, Y., Fu, Y., Zhou, W., Shi, H., Zhang, X., Kundu, S., Yazdanbakhsh, A., and Lin, Y. C. Shiftaddllm: Accelerating pretrained llms via post-training multiplication-less reparameterization. *Advances in Neural Information Processing Systems*, 37:24822–24848, 2024.

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., and Choi, Y. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

Zhao, P. and Yuan, X. Ganq: Gpu-adaptive non-uniform quantization for large language models. *arXiv preprint arXiv:2501.12956*, 2025.

Zheng, L., Yin, L., Xie, Z., Sun, C. L., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

A. Gradient-Based Search for Hierarchical Linear Quantization

The gradient-based search updates the quantization parameters by directly computing the MSE and propagating the gradients. At initialization, we set $z = \min(\mathrm{W})$, and the initial value of s is set to the scale factor used in uniform quantization. Specifically, let $\Delta = \frac{\max(\mathrm{W})-\min(\mathrm{W})}{2^q-1}$, For a q-bit quantization, the initial value of s is set to $s = [\Delta, 2\Delta, ...2^{q-1}\Delta]$. The workflow of the search process is illustrated in Algorithm 2. Here, we adopt an approximately differentiable optimization strategy to jointly update the scales and zero-points. The core idea is to decouple the quantization process into two stages: discrete selection and continuous reconstruction, thereby circumventing the gradient issues inherent in discrete operations.

- **Forward Pass**: the optimal combination of discrete codewords is determined via nearest-neighbor search. Subsequently, this combination is used together with learnable scaling factors and zero-points to reconstruct the quantized weights through a continuous weighted summation.
- **Backward Pass**: During backpropagation, gradients cannot flow through the argmin-based discrete selection step. However, they can naturally propagate back through the continuous reconstruction step to update the scaling factors and zero-points. This enables the continuous parameters to be effectively optimized via standard gradient descent.

To ensure training stability, we enforce a non-negativity constraint on the scaling factors and apply dynamic range clipping to the zero-point values, which effectively prevents training divergence. From an optimization perspective, this approach is conceptually aligned with the straight-through estimator (STE) (Liu et al., 2022) philosophy. However, a key distinction lies in the fact that instead of relying on handcrafted gradient approximation rules, we carefully design the quantization expression itself to allow gradients to flow naturally to the continuous parameters, resulting in a more elegant approximation of gradient flow.

Algorithm 2 Gradient-Based Optimization for Hierarchical Linear Quantization

Require: Weight matrix $\mathrm{W} \in R^{n\times k}$, bit width q,
Group size g, Max iterations T_{max}, Tolerance ε
Ensure: scales $s \in R^{n\times \frac{k}{g}\times q}$, zero points $z \in R^{n\times \frac{k}{g}}$

Reshape W into groups $\widetilde{\mathrm{W}} \in R^{n\times \frac{k}{g}\times g}$
Calculate uniform quantization scale :
$\Delta \leftarrow \frac{\max(\widetilde{\mathrm{W}})-\min(\widetilde{\mathrm{W}})}{2^q-1}$
Initialize $s^{(0)} \leftarrow [\Delta, 2\Delta, ...2^{q-1}\Delta]$, $z^{(0)} \leftarrow \min(\widetilde{W})$, $preL \leftarrow INF$
Generate binary combinations C
for $t \leftarrow 1$ to T_{max} **do**
  $\hat{\mathrm{W}}^{(\mathrm{t})} \leftarrow \widetilde{\mathrm{W}} - z^{(t-1)}$
  $V^{(t)} \leftarrow s^{(t-1)} \times C^T$ # Matrix product
  $k^* \leftarrow \arg\min_k \|\hat{\mathrm{W}}^{(\mathrm{t})} - V_k^{(t)}\|$
  $B^{(t)} \leftarrow C_{k^*}$ # Choose best binary combination
  $\hat{\mathrm{W}}^{(\mathrm{t})} \leftarrow s^{(t-1)} \times {B^{(t)}}^T$
  $\hat{\mathrm{W}}^{(\mathrm{t})} \leftarrow \hat{\mathrm{W}}^{(\mathrm{t})} + z^{(t-1)}$
  $L \leftarrow mean((\widetilde{\mathrm{W}} - \hat{\mathrm{W}}^{(\mathrm{t})})^2)$
  $s^{(t)},\ z^{(t)} \leftarrow Backpropagate(L)$
  if $L - preL < \varepsilon$ **then**
    break
  end if
  $perL \leftarrow L$
end for

We conduct an experiment where HLQ-GPTQ is implemented using both alternating optimization and gradient-based search methods. As shown in Table 12, both methods have similar results. Therefore, considering algorithmic robustness, we recommend adopting the alternating optimization method in practical applications. Besides, the alternating optimization approach can complete model quantization within a very short time (Table 13).

Table 12: Performance comparison of HLQ-GPTQ using alternating optimization and gradient-based optimization across different models and bit configurations. perplexity(↓) is reported.

Method	# W	# G	BPW	LLaMA2-7		LLaMA2-13		LLaMA3.1-8		Qwen3-8	
				wikitext2	c4	wikitext2	c4	wikitext2	c4	wikitext2	c4
Baseline	16	-	16	5.47	6.97	4.88	6.47	6.15	8.89	9.72	13.30
Alternating Optimization	2	128	2.37	9.95	15.02	7.12	9.92	16.93	25.44	18.15	24.63
Gradient-based Method	2	128	2.37	9.90	14.89	7.02	9.75	17.32	27.14	18.13	24.60
Alternating Optimization	3	128	3.5	5.97	7.74	5.14	6.91	7.24	10.80	10.45	14.15
Gradient-based Method	3	128	3.5	5.97	7.74	5.14	6.90	7.41	10.85	10.53	14.22

Table 13: Runtime (h) comparison of GPTQ, HLQ-GPTQ (Alternate), and HLQ-GPTQ (Gradient).

Model	GPTQ	HLQ-GPTQ(Alternate)	HLQ-GPTQ(Gradient)
LLaMA2-7B	0.20	0.15	0.20
LLaMA2-13B	0.40	0.35	0.40
LLaMA3.1-8B	0.25	0.20	0.25

B. Compression Ratios and Actual Memory Footprint of Quantized Models

Here,we provide a comparison of the compression ratios and actual model sizes between hierarchical linear quantization and uniform quantization. Specially, table 14 show the results of compression radios, while table 15 show the results of actual model size. It can be observed that, under the same weight bit-width, HLQ incurs a slightly higher memory footprint compared to uniform quantization.

Table 14: Complete comparison of Model Compression Rates (↑) between Uniform Quantization and Hierarchical Linear Quantization. Embedding and LM-head layers are excluded from quantization for all models.

Wbit	G	Method	L2-7	L2-13	L3.1-8	L3.1-70	Q3-8	Q3-14	Q3-32
2	128	RTN	5.74	6.16	3.95	6.00	3.68	4.32	5.51
		HLQ	5.50	5.88	3.84	5.75	3.59	4.19	5.29
3	128	RTN	4.27	4.48	3.25	4.40	3.08	3.48	4.14
		HLQ	4.01	4.19	3.11	4.13	2.96	3.32	3.90

Table 15: Complete comparison of Model size (GB) between Uniform Quantization and Hierarchical Linear Quantization. Embedding and LM-head layers are excluded from quantization for all models.

Wbit	G	Method	L2-7	L2-13	L3.1-8	L3.1-70	Q3-8	Q3-14	Q3-32
2	128	RTN	2.18	3.93	3.78	21.84	4.13	6.35	11.07
		HLQ	2.27	4.11	3.88	22.84	4.23	6.55	11.52
3	128	RTN	2.93	5.41	4.59	29.81	4.95	7.89	14.70
		HLQ	3.12	5.78	4.80	31.80	5.15	8.28	15.61

C. Compared with Non-Uniform and Quantization-Aware Training Methods

In this section, we further compare our results with several non-uniform quantization that adopt codebooks and QAT methods. As shown in the table 16, HLQ-Finetuning exhibit a certain gap in model accuracy compared to these non-uniform approaches, which is expected. Methods such as Quip# (Tseng et al., 2024) and AQLM (Egiazarian et al., 2024) require a large calibration set (e.g., 4K training samples with a context length of 4096) and involve complex distillation procedures, leading to substantial computational and time costs during quantization. Meanwhile, when using the same training data, our method achieves comparable results to EfficientQAT (Chen et al., 2024c), however, EfficientQAT necessitates retraining the model weights, whereas our approach operates without any weight training, significantly reducing memory consumption, accelerating quantization, and mitigating the risk of overfitting.

Table 16: Evaluation of quantized LLaMA3.1 models for 2 bits. The table reports perplexity on WikiText2 and c4, as well as accuracy for zero-shot tasks. The Average column is the mean of 5 zero-shot task accuracies. EfficientQAT* indicates the version trained using the same dataset as HLQ-Finetuning.

Model	Method	wikitext2	c4	WinoGrande	HellaSwag	ArcC	ArcE	PiQA	Average(↑)
LlaMA3.1-8B	Baseline	6.15	8.89	73.32	60.04	51.28	81.57	80.03	69.25
	Quip#	9.92	13.45	69.78	54.18	40.12	74.56	78.75	63.48
	AQLM	9.56	12.98	70.92	54.92	40.84	74.48	78.12	63.85
	EfficientQAT	10.12	14.95	64.58	49.89	36.13	68.34	74.56	58.71
	EfficientQAT*	11.56	16.34	62.58	49.76	33.13	68.34	74.56	57.68
	HLQ-Finetuning	11.24	15.87	62.72	49.52	32.91	69.02	74.78	57.79
LlaMA3.1-70B	Baseline	2.93	6.92	80.51	66.36	60.41	86.99	82.37	75.33
	Quip#	6.89	10.47	77.11	61.94	50.45	77.05	78.32	68.98
	AQLM	6.38	9.73	77.56	62.87	50.21	78.26	79.15	69.61
	EfficientQAT	6.92	10.56	69.27	59.95	49.04	78.84	79.22	67.26
	EfficientQAT *	7.66	11.48	71.57	57.66	46.29	76.80	77.72	66.00
	HLQ-Finetuning	7.48	11.24	72.38	58.86	46.52	77.16	77.54	66.50

D. Experimental Configurations for Memory and Time Comparison.

In this section, we provide the detailed experimental configurations used to measure the computational memory consumption (CPU and GPU) and quantization time of different quantization methods on LLaMA3.1–70B under 2-bit quantization. All the experiments is conducted on RTX 4090.

- GPTQ: Following the official implementation, we use 128 calibration samples with a context length of 2048 and perform layer-wise quantization.
- AWQ: According to the official codebase, we use 512 calibration samples from the default AWQ dataset, each with a context length not exceeding 512.
- OmniQuant: We use 128 calibration samples with a context length of 2048 and follow the official recommendation to iterate 20 epochs for each transformer block.
- HLQ-GPTQ: We adopt the same configuration as GPTQ, using 128 calibration samples with a context length of 2048 for layer-wise quantization.
- Quip#: We use 4K calibration samples with a context length of 4096 and conduct both layer-wise quantization.
- AQLM: We follow the same configuration as Quip#, using 4K calibration samples with a context length of 4096, and perform block-wise quantization.
- DB-LLM: We use a self-collected dataset of 20K samples for calibration.
- EfficientQAT: We use 1K calibration samples with a context length of 2048,iterate 2 epochs for each transformer block.
- HLQ-Finetuning: We use 1K calibration samples with a context length of 2048,iterate 2 epochs for each transformer block.

E. More results on Devices

E.1. More Results on ARM-Based Devices

Table 17: Comparison of prefill and decode throughput (tokens/s) between ELUTQ and llama.cpp on the Apple M2 chip and RK3588. All experiments are conducted with 4 threads and a batch size of 1.

Model	Framework	Wbits	BPW	Input,Output	Apple M2		RK3588	
					Prefill	Decode	Prefill	Decode
Qwen2-1.5B	llama.cpp	Q3_K_M	3.91	128,128	103.11	57.69	21.53	15.36
				256,128	101.37	56.11	21.28	15.13
		Q3_K_S	3.50	128,128	86.67	54.64	19.79	14.57
				256,128	85.54	53.49	19.58	14.37
		Q2_K_S	2.50	128,128	62.73	48.55	16.12	12.95
				256,128	60.17	47.87	15.98	12.78
	ELUTQ	W3(g128)	3.50	128,128	88.72	55.03	27.77	14.84
				256,128	93.06	54.90	27.16	14.54
		W2(g128)	2.37	128,128	120.65	65.19	33.61	17.42
				256,128	124.36	64.81	35.39	17.15

E.2. Results on GPUs

Here, we provide the experimental results on GPUs. Our evaluation covers a range of hardware platforms, including high-performance server-grade chips (A100 and H100), a consumer-grade GPU (RTX 4090), and an edge-computing–oriented accelerator (Jetson AGX Orin). As baselines, we consider AWQ, which uses a uniform quantization format, and SqueezeLLM, which adopts a non-uniform quantization format. We convert our HLQ weights into the BCQ format via a simple linear transformation and perform inference using the AnyBCQ framework. The experiments evaluate the speedup of throughput for generating a sequence of length 1024 compared with fp16 using cu, and the results are presented in the table 18.

Table 18: End to End Evaluation (tokens/s) of generating a sequence lengeth of 1024 on gpus. OOM indicates that the model cannot be deployed on the corresponding GPU.

Model	Wbits	Method	A100	H100	RTX3090	RTX4090
LLaMA3.1-8B	16	Baseline	115	100	55	60
	2	AWQ	175	217	119	138
		SqueezeLLM	160	189	99	109
		BCQ	**256**	**311**	**184**	**210**
	3	AWQ	198	233	143	164
		SqueezeLLM	185	228	130	141
		BCQ	**220**	**282**	**155**	**174**
LLaMA2-13B	16	Baseline	60	105	OOM	OOM
	2	AWQ	104	146	79	91
		SqueezeLLM	92	107	75	80
		BCQ	**185**	**246**	**135**	**152**
	3	AWQ	123	178	88	104
		SqueezeLLM	10	155	80	87
		BCQ	**152**	**208**	**106**	**118**
LlaMA3.1-70B	16	Baseline	OOM	OOM	OOM	OOM
	2	AWQ	34	51	OOM	OOM
		SqueezeLLM	28	41	OOM	OOM
		BCQ	**55**	**98**	OOM	OOM
	3	AWQ	34	54	OOM	OOM
		SqueezeLLM	28	45	OOM	OOM
		BCQ	**43**	**79**	OOM	OOM

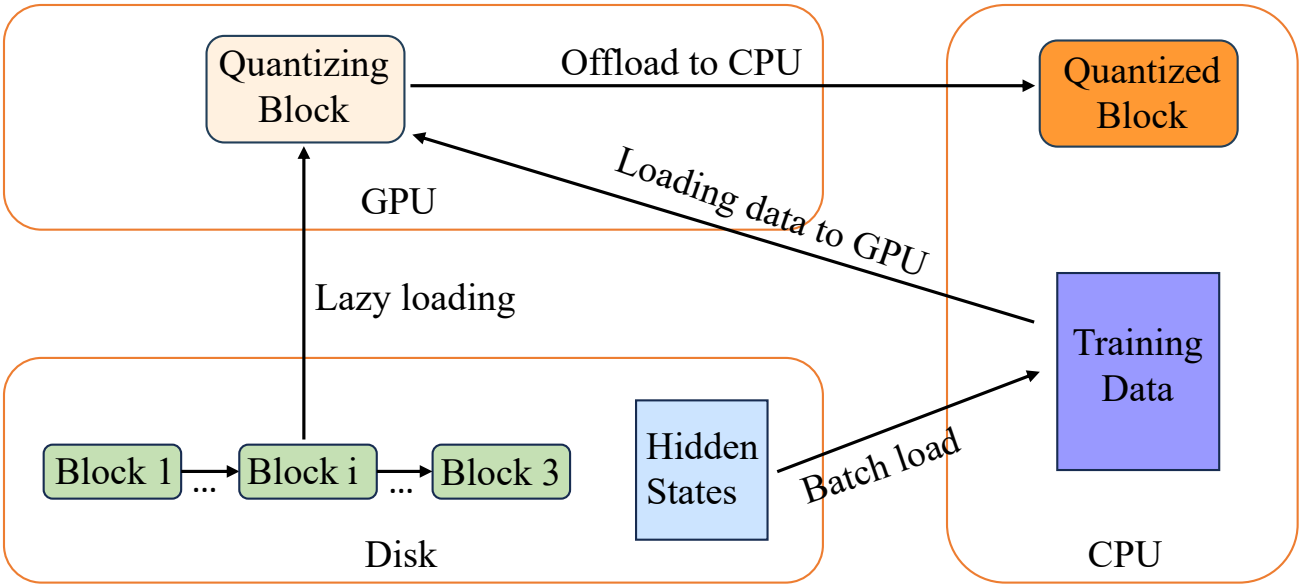


Figure 10: overview of system-level optimizations for the quantization pipeline.

F. System-level Optimizations for the Quantization Pipeline

As illustrated in the figure 10 , we present how to perform model quantization for large-scale models with limited CPU and GPU memory.

F.1. Lazy Loading

In both GPTQ's Hessian-based quantization and Efficient Finetuning's block reconstruction, the quantization unit is a Transformer block. Quantizing a single block does not depend on other blocks, neither the already quantized nor the unquantized ones. Therefore, instead of loading the entire model into memory, we can load only one transformer block at a time from disk. Thanks to the sharded weight storage mechanism in the Transformers (Wolf et al., 2020) framework, we can leverage the weight mapping file (e.g., `model.safetensors.index.json`) to selectively load a specific block from disk for quantization. Once the quantization of a block on the GPU is completed, it is immediately offloaded to the CPU to reduce GPU memory usage.

F.2. Chunked Computation of HLQ Parameters

In Algorithm 1, when computing the initial quantization parameters of HLQ for a weight matrix, the optimization within each weight group is independent. For extremely large weight matrices, we can split the matrix into smaller chunks and perform parameter search independently for each chunk. As shown in the figure, for the largest matrix in Llama3.1-70B, the down_proj of shape [28672, 8192] performing HLQ parameter search without batching leads to a peak memory usage of approximately 30 GB for 2-bit quantization and 35 GB for 3-bit quantization. In contrast, when the search is executed in two chunks, the peak memory drops to about 22 GB for 2-bit and 24 GB for 3-bit quantization. This chunked computation strategy significantly reduces GPU memory consumption and achieves approximately linear memory scaling with respect to partition granularity.

F.3. Offloading Hidden States to Disk

During block reconstruction, a large amount of intermediate data (hidden states) must be stored. For example, with a 1K calibration set and a context length of 2048, the hidden size of LLaMA 3.1-8B (hiddensize = 4096) requires about 16 GB of CPU memory in FP16 format, while LLaMA 3.1-70B requires around 64 GB. This imposes heavy memory pressure on the CPU.

To mitigate this, we offload hidden states to disk. However, fully offloading them leads to excessive disk I/O latency, as

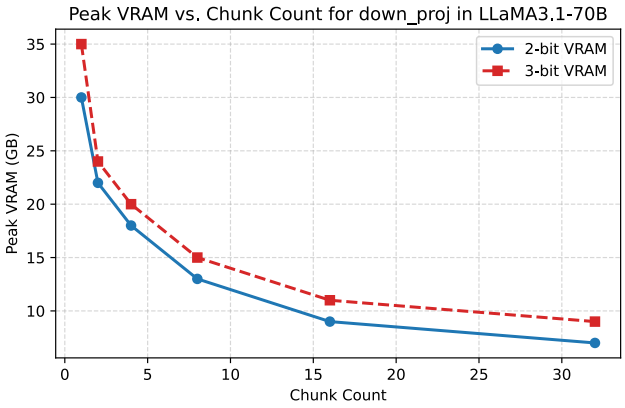


Figure 11: Peak VRAM usage of the Llama-3.1-70B down_proj matrix under 2-bit and 3-bit quantization with different chunk counts. Increasing the number of chunks significantly reduces the peak memory footprint.

disk-to-CPU bandwidth is much lower than CPU-to-GPU bandwidth. To address this, we employ a batched loading strategy: the hidden states are divided into several batches, and only one batch is loaded from disk at a time. For instance, in LLaMA 3.1-8B, we load 4 GB of data ($\frac{1}{4}$ of the total) per iteration. Once a batch is consumed, the next one is fetched from disk. This approach substantially reduces CPU memory usage while minimizing disk access frequency and context switching overhead, thereby ensuring efficient and continuous quantization throughput.

F.4. Overhead of Disk Access

Using lazy loading and loading training data directly from disk requires disk access during the execution of quantization algorithms. This inevitably introduces additional memory access overhead. Even with NVMe-based communication protocols, disk bandwidth remains far below that of the CPU. As shown in Table 19, we evaluate HLQ-GPTQ and HLQ-Finetuning on LLaMA3.1-8B, Qwen3-32B, and LLaMA3.1-70B. For HLQ-GPTQ, enabling disk-based loading substantially increases the quantization time, by approximately 30%–40% compared to in-memory loading. However, in the case of HLQ-Finetuning, the impact of disk access is relatively minor compared to the overall tuning time. Specifically, the tuning time increases by 25% for LLaMA3.1-8B, 12% for Qwen3-32B, and only about 5% for LLaMA3.1-70B. Therefore, when performing HLQ-Finetuning on very large models, the overhead caused by disk access can be considered negligible relative to the total tuning time.

Table 19: Comparison of tuning times with and without disk usage. All models are tuned under the same quantization configuration w2g128.

Method	**Model**	**disk(✗)**	**disk(✓)**	Overhead (%)
HLQ-GPTQ	llama3.1-8B	0.25h	0.35h	40%
	Qwen3-32B	1.35h	1.85h	37%
	llama3.1-70B	3.00h	4.00h	33%
HLQ-Finetuning	llama3.1-8B	2.00h	2.50h	25%
	Qwen3-32B	8.25h	9.30h	12%
	llama3.1-70B	19.00	20.00h	5%

G. Full Results on Model Accuracy

Here, we present the full results of our method across various models. Specifically, Table 20 reports the perplexity on wikitext2, Table 21 reports the accuracy on the LLaMA series, while Table 22 shows the accuracy on the Qwen series models.

Table 20: A comparison of wikitext2 perplexity (↓) between weight-only quantization methods, with a context length of 2048. **BPW** represents the average number of bits per weight. Scale and zero-point are assumed to be stored in fp16 format. PB-LLM* denotes the result of PB-LLM with GPTQ and 20% salient weight. "–" indicates that the framework does not support this model.

Method	# W	# G	BPW	L2-7	L2-13	L3.1-8	L3.1-70	Q3-8	Q3-14	Q3-32
Baseline	16	-	16	5.47	4.88	6.15	2.81	9.72	8.64	7.61
GPTQ	2	128	2.25	16.01	10.33	109.30	22.54	29.19	26.15	23.18
AWQ	2	128	2.25	2.21e5	1.23e5	1.74e6	1.32e6	1.21e5	1.42e5	1.43e4
OmniQuant	2	128	2.25	11.06	8.26	19.18	16.21	-	-	-
ApiQ	2	128	2.25	8.25	6.84	-	-	-	-	-
PB-LLM*	-	-	2.2	17.19	12.47	21.84	18.72	-	-	-
ShiftAddLLM	2	128	2.37	9.58	12.57	34.41	12.46	-	-	-
AnyBCQ	2	128	2.37	-	-	19.01	-	-	-	-
HLQ-GPTQ	2	128	2.37	9.90	7.02	17.32	14.42	18.15	17.54	11.37
HLQ-Finetuning	2	128	2.37	**8.05**	**6.75**	**11.24**	**8.43**	**17.87**	**14.92**	**10.31**
DB-LLM	2	64	-	7.23	6.19	13.60	-	-	-	-
HLQ-GPTQ	2	64	2.75	8.72	6.47	13.22	11.77	17.13	13.11	10.47
HLQ-Finetuning	2	64	2.75	**7.07**	**6.07**	**11.22**	**7.06**	**15.09**	**12.24**	**9.72**
GPTQ	3	128	3.25	6.29	5.42	9.58	5.67	10.76	9.69	9.15
AWQ	3	128	3.25	6.24	5.32	8.22	5.51	14.90	10.41	8.89
OmniQ	3	128	3.25	6.03	5.28	8.27	5.66	-	-	-
SqueezeLLM(0.45%)	3	-	3.25	5.96	5.23	-	-	-	-	-
GANQ	3	-	3.25	5.96	-	7.46	-	-	-	-
ShiftAddLLM	3	128	3.50	6.04	5.33	7.71	4.66	-	-	-
AnyBCQ	3	128	3.50	-	-	7.68	-	-	-	-
HLQ-GPTQ	3	128	3.50	5.97	5.14	7.24	5.44	10.54	9.46	8.28
HLQ-Finetuning	3	128	3.50	**5.93**	**5.12**	**7.01**	**4.22**	**9.95**	**9.21**	**7.92**

Table 21: Accuracy (%) on zero-shot tasks on LlaMA models by lm_eval v0.4.9. PB-LLM* denotes the result of PB-LLM with GPTQ and 20% salient weight. **Bold**: best result; underlined: second-best. ”–” indicates that the framework does not support this model. Acc is reported, not acc norm.

Model	Method	Bits	Group	WinoGrande	HellaSwag	ArcC	ArcE	PiQA	Average(↑)
LLaMA2-7B	Baseline	16	-	70.24	56.86	41.04	74.66	77.86	64.13
	GPTQ	3	128	66.93	54.55	38.31	70.54	77.04	61.47
	AWQ	3	128	69.77	54.73	38.82	71.59	76.22	62.23
	OmniQ	3	128	66.19	54.12	38.72	72.15	77.12	61.66
	HLQ-GPTQ	3	128	67.11	54.29	40.75	73.25	76.18	62.32
	HLQ-Finetuning	3	128	69.30	54.22	39.42	72.98	76.66	**62.52**
	GPTQ	2	128	53.12	34.17	21.33	35.40	59.30	40.66
	AWQ	2	128	49.72	25.38	22.53	25.46	52.67	35.15
	OmniQ	2	128	55.88	40.28	23.46	50.13	65.13	46.98
	PB-LLM*	-	-	50.36	30.49	22.01	29.88	55.22	37.60
	HLQ-GPTQ	2	128	59.67	38.61	25.26	54.34	67.30	49.04
	HLQ-Finetuning	2	128	62.75	46.61	30.72	63.01	72.25	**55.07**
LLaMA2-13B	Baseline	16	-	72.22	60.07	48.29	79.42	79.05	67.81
	GPTQ	3	128	70.88	57.83	45.65	77.99	78.56	66.18
	AWQ	3	128	71.82	58.58	44.62	77.95	77.75	66.14
	OmniQ	3	128	70.01	58.46	46.16	77.86	78.40	66.18
	HLQ-GPTQ	3	128	70.95	58.10	45.79	78.76	78.60	66.44
	HLQ-Finetuning	3	128	71.94	58.54	45.32	78.70	78.56	**66.62**
	GPTQ	2	128	55.80	41.06	21.93	55.60	67.08	48.29
	OmniQ	2	128	57.93	46.23	30.29	63.22	70.13	53.56
	PB-LLM*	-	-	52.33	30.23	23.12	31.27	55.01	38.39
	HLQ-GPTQ	2	128	67.43	47.83	35.14	66.92	74.10	58.28
	HLQ-Finetuning	2	128	66.38	50.78	36.18	67.72	75.19	**59.25**
LLaMA3.1-8B	Baseline	16	-	73.32	60.04	51.28	81.57	80.03	69.25
	GPTQ	3	128	70.72	55.58	42.75	74.54	77.48	64.21
	AWQ	3	128	70.96	55.41	44.45	76.53	77.58	64.98
	HLQ-GPTQ	3	128	69.69	56.25	45.05	75.80	78.29	65.02
	HLQ-Finetuning	3	128	69.25	55.75	45.65	77.31	78.13	**65.22**
	GPTQ	2	128	48.78	27.43	19.20	28.96	55.55	35.98
	HLQ-GPTQ	2	128	59.27	38.96	22.18	40.70	60.66	44.35
	HLQ-Finetuning	2	128	62.72	49.52	32.91	69.02	74.78	**57.79**
LLaMA3.1-70B	Baseline	16	-	80.51	66.36	60.41	86.99	82.37	75.33
	GPTQ	3	128	78.14	62.58	52.99	82.07	80.63	71.28
	AWQ	3	128	78.13	63.51	56.01	82.71	80.28	72.13
	HLQ-GPTQ	3	128	78.54	64.08	55.47	83.54	80.79	72.48
	HLQ-Finetuning	3	128	79.48	64.92	56.91	83.25	81.10	**73.13**
	GPTQ	2	128	59.92	41.07	24.65	38.95	62.78	45.57
	HLQ-GPTQ	2	128	61.25	48.58	38.77	68.58	65.16	56.47
	HLQ-Finetuning	2	128	72.38	58.86	46.52	77.16	77.54	**66.50**

Table 22: Accuracy (%) on zero-shot tasks on Qwen models by lm_eval v0.4.9. **Bold**: best result; underlined: second-best. "–" indicates that the framework does not support this model. Acc is reported, not acc norm.

Model	Method	Bits	Group	WinoGrande	HellaSwag	ArcC	ArcE	PiQA	Average(↑)
Qwen3-8B	Baseline	16	-	67.80	57.11	55.55	83.42	76.61	68.10
	GPTQ	3	128	65.75	53.83	44.80	75.55	75.24	63.04
	AWQ	3	128	62.25	54.71	41.03	68.27	71.98	59.65
	HLQ-GPTQ	3	128	68.35	54.03	49.83	79.84	76.17	65.64
	HLQ-Finetuning	3	128	68.27	53.88	52.13	80.98	76.77	**66.41**
	GPTQ	2	128	51.48	30.45	21.56	37.82	55.92	39.45
	AWQ	2	128	51.72	25.32	23.81	25.76	52.47	35.82
	HLQ-GPTQ	2	128	61.17	40.48	32.25	61.83	69.48	53.04
	HLQ-Finetuning	2	128	64.56	46.07	41.21	72.18	73.18	**59.44**
Qwen3-14B	Baseline	16	-	72.93	60.93	58.53	84.13	79.92	71.29
	GPTQ	3	128	70.22	57.33	55.10	77.49	76.26	67.28
	AWQ	3	128	57.06	55.64	48.81	75.70	74.32	62.31
	HLQ-GPTQ	3	128	71.58	58.25	56.05	81.35	78.02	69.05
	HLQ-Finetuning	3	128	72.56	59.50	56.31	82.74	79.05	**70.03**
	GPTQ	2	128	58.74	35.12	31.87	51.26	67.79	48.96
	AWQ	2	128	53.87	28.16	27.31	29.16	58.78	39.46
	HLQ-GPTQ	2	128	66.14	48.92	40.87	70.49	73.72	60.03
	HLQ-Finetuning	2	128	70.96	50.54	48.46	79.59	75.41	**64.99**
Qwen3-32B	Baseline	16	-	73.65	63.92	57.85	84.42	80.93	72.15
	GPTQ	3	128	72.21	56.17	50.92	78.60	74.31	66.44
	AWQ	3	128	72.45	57.64	50.87	79.31	75.25	67.10
	HLQ-GPTQ	3	128	71.67	58.16	51.56	79.76	76.14	67.45
	HLQ-Finetuning	3	128	72.76	58.57	52.45	80.23	77.03	**68.21**
	GPTQ	2	128	56.62	40.67	35.92	61.30	57.19	50.34
	AWQ	2	128	53.56	35.79	32.43	58.27	53.65	46.76
	HLQ-GPTQ	2	128	73.01	55.15	48.81	76.43	75.63	65.81
	HLQ-Finetuning	2	128	72.98	57.02	50.45	80.48	76.92	**67.57**