CoBIT: Unified Optimization of Codebooks, Assignments, and Activation Composition for Extreme Low-Bit LLM Quantization

Anonymous authors
Paper under double-blind review

Abstract

Post-training quantization is essential for efficient large model inference, yet preserving accuracy at ultra-low budget below 4 bits remains challenging. Clustering-based quantization provides adaptive non-uniform representations, but existing methods use limited information to construct codebooks and assign weights, while activation quantization is typically handled separately. We present *CoBIT*, a post-training quantization framework that optimizes weight clustering and activation quantization under a common layer-output reconstruction objective. For weights, CoBIT alternates Hessian-aware codebook construction with error-compensated assignment. For activations, CoBIT selects a per-token exponent–mantissa composition based on layer-output error. We introduce *PhaseLUT* to replace runtime composition search with a lookup, and a hardware-aware kernel that integrates composition selection, activation quantization, and low-precision computation. Across all four general-purpose models, CoBIT improves average accuracy over the strongest baseline in each setting by 4.8% and 35.9% points with 4-bit activations and 3- and 2-bit weight clustering, respectively. With 4-bit weight indices, it also achieves up to $1.53\times$ end-to-end decoding speedup over dense FP16.

1 Introduction

Large language models (LLMs) have achieved remarkable performance across a wide range of tasks, but their growing model sizes impose substantial memory and computational costs during inference. Quantization offers a promising solution by reducing model precision, thereby lowering memory footprint and enabling efficient low-precision computation (Ashkboos et al., 2024; Zandieh et al., 2025; Liu et al., 2025; Liang et al., 2026). Recent GPUs further support low-precision formats such as FP8 and FP4 (NVIDIA, 2022; 2024). However, jointly quantizing activations to 4 bits and weights below 4 bits with post-training quantization (PTQ) remains challenging (Song et al., 2025; Yin et al., 2026) due to substantial quantization error.

The general quantization process can be divided into two stages: *grid construction* and *value assignment*. Grid construction defines the set of representable quantized values, while value assignment maps each full-precision weight to one of these values. For instance, conventional linear quantization uses a fixed uniform grid with round-to-nearest (RTN) assignment, which is simple to implement but offers limited flexibility in adapting to diverse input data distributions. More flexible approaches employ non-uniform or distribution-aware grids (Zhang et al., 2023; Dettmers et al., 2023), while clustering-based quantization directly learns the codebook from the model weights. Although clustering-based quantization is promising (Kim et al., 2024; Egiazarian et al., 2024; Kim et al., 2025; Yin et al., 2026), limitations in both stages hinder accuracy at lower bit widths.

Beyond weights, activations can also be quantized to improve hardware efficiency. Due to their dynamic nature, existing methods often use simple schemes, such as FP or linear quantization, to limit runtime overhead. These methods typically minimize the ℓ_2 reconstruction error between full-precision and quantized activations. However, activation quantization is often optimized independently of quantized weights, so minimizing activation error does not necessarily minimize the resulting error on the activation–weight product (Li et al., 2023). To address these limitations, we present *CoBIT*, which jointly considers weight clustering and activation quantization under a unified

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

layer-output objective. CoBIT leverages Hessian information to optimize both codebook construction and weight assignment. For activations, CoBIT adopts FP quantization and adaptively selects the exponent–mantissa composition to minimize the resulting layer-output error. To enable efficient inference, we introduce *PhaseLUT*, which replaces runtime composition search with a simple lookup, and co-design a GPU kernel that integrates composition selection, activation quantization, and low-precision computation. Our main contributions are summarized as follows:

- We develop a Hessian-aware method that alternates between codebook construction and error-compensated weight assignment, improving low-bit quantization accuracy while substantially reducing quantization time.
- We decompose the layer-output objective to isolate the activation-induced output error from weight clustering, providing a direct criterion for selecting the per-token exponent–mantissa composition under a fixed bit budget.
- We co-design PhaseLUT with a hardware-aware inference kernel that integrates per-token composition selection, activation quantization, and low-precision computation, minimizing runtime overhead and enabling end-to-end inference speedups.

2 Related Work

2.1 LLM Post-Training Quantization

Activation outliers introduce a major challenge for low-bit quantization by expanding the dynamic range and increasing quantization error (Dettmers et al., 2022; Sun et al., 2024). A common strategy is to jointly transform weights and activations while preserving the layer output. SmoothQuant (Xiao et al., 2024) shifts quantization difficulty from activations to weights through channel-wise scaling, while QuaRot (Ashkboos et al., 2024), SpinQuant (Liu et al., 2025), and DuQuant (Lin et al., 2024) redistribute activation outliers across channels through rotations. ParoQuant (Liang et al., 2026) applies a similar transformation strategy to improve weight-only quantization. The transformed weights can then be quantized using RTN, GPTQ, or AWQ (Frantar et al., 2023; Lin et al., 2026).

Another line of work learns non-uniform codebooks for weight quantization. SqueezeLLM (Kim et al., 2024) uses Fisher-diagonal weighted clustering, while GuidedQuant (Kim et al., 2025) introduces LNQ, which alternates closed-form codebook updates with coordinate-descent assignments. CodeQuant (Yin et al., 2026) fine-tunes its codebooks and uses a lookup-table kernel to compute directly on clustered weights. CoBIT instead uses error-compensated assignment and derives its weight-clustering and activation-quantization objectives from a common layer-output objective.

2.2 Mixed Precision and Composition Quantization

Mixed-precision quantization assigns different bit widths to model components according to their quantization sensitivity or importance (Dong et al., 2019; Yao et al., 2021). Beyond bit-width allocation, floating-point formats introduce another design dimension: how bits are allocated between the exponent and mantissa (Kuzmin et al., 2024). Under a fixed bit budget, exponent–mantissa compositions trade dynamic range for numerical precision, and their effectiveness depends on the data distribution (Liu et al., 2021; 2023). This motivates mixed-composition quantization, which adaptively selects exponent–mantissa configurations (Dotzel et al., 2024; Park et al., 2026). CoBIT introduces an activation-statistics-based lookup table for token-wise composition selection.

2.3 Hardware-Efficient Quantization

Unlike weights, activations are generated during inference and must be quantized at runtime. This additional overhead motivates system-level optimizations such as operator fusion, optimized data layouts, and specialized kernels (Zhao et al., 2024). TQT (Jain et al., 2020) uses power-of-two (PoT) scales, allowing scaling operations to be implemented with efficient bit shifts rather than explicit division. Similarly, MX formats (Rouhani et al., 2023) adopt shared PoT scales, while NVFP4 (NVIDIA, 2025) uses more flexible FP8 block-level scales. Building on PoT scaling, CoBIT integrates mixed-composition activation quantization and clustered computation into an efficient kernel, achieving both kernel-level and end-to-end speedups.

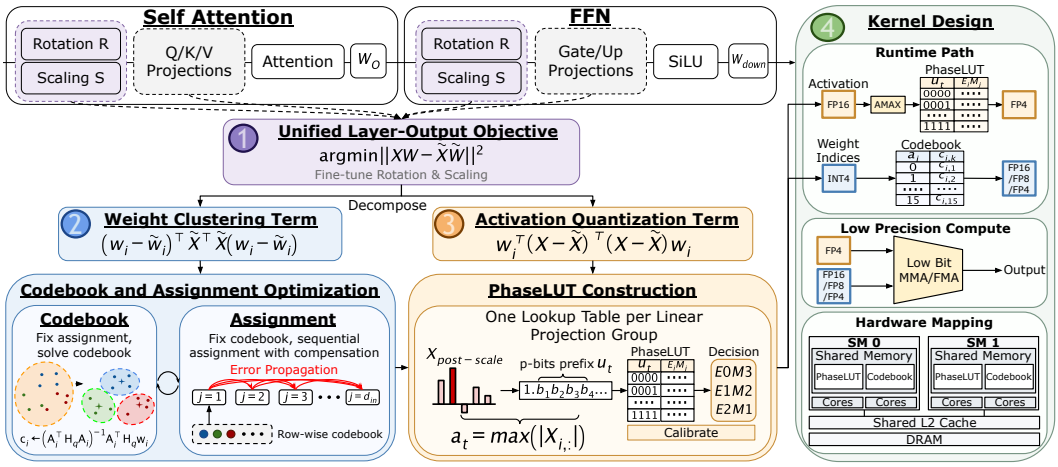

Figure 1: Overview of CoBIT framework.

3 METHODOLOGY

Figure 1 overviews CoBIT. First, outlier smoothing (Section 3.1) transforms weights and activations for low-bit quantization. CoBIT then alternates codebook construction and weight assignment (Section 3.3). Next, PhaseLUT (Section 3.4) selects token-wise FP compositions calibrated to minimize activation-induced output error. Finally, the inference kernel (Section 3.5) integrates PhaseLUT lookup, activation quantization, and low-precision computation.

3.1 OUTLIER SMOOTHING

As demonstrated by prior work (Dettmers et al., 2022; Sun et al., 2024), activation outliers expand the range required by the activation quantizer and increase activation-quantization error. Consider a linear layer with activation $\mathbf{X} \in \mathbb{R}^{T\times d_{\text{in}}}$ over T tokens and weight $\mathbf{W} \in \mathbb{R}^{d_{\text{out}}\times d_{\text{in}}}$. Following QuaRot and SmoothQuant (Ashkboos et al., 2024; Xiao et al., 2024), we combine scaling and rotation to smooth the activation outliers while preserving the activation–weight product. Mathematically, this transformation can be expressed as $\mathbf{X}\mathbf{W}^\top = (\mathbf{X}\mathbf{S}\mathbf{R})\left(\mathbf{R}^\top\mathbf{S}^{-1}\mathbf{W}^\top\right)$. $\mathbf{S} \in \mathbb{R}^{d_{\text{in}}\times d_{\text{in}}}$ is an invertible diagonal scaling matrix, and $\mathbf{R} \in \mathbb{R}^{d_{\text{in}}\times d_{\text{in}}}$ is an orthogonal rotation matrix. For simplicity, we define $\mathbf{X}_{R,S} = \mathbf{X}\mathbf{S}\mathbf{R}$ as the transformed activation and $\mathbf{W}_{R,S}^\top = \mathbf{R}^\top\mathbf{S}^{-1}\mathbf{W}^\top$ as the transformed weight. Since $\mathbf{S}\mathbf{S}^{-1} = \mathbf{I}_{d_{\text{in}}}$ and $\mathbf{R}\mathbf{R}^\top = \mathbf{I}_{d_{\text{in}}}$, the scaling and rotation will not impact the matrix product. $\mathbf{X}_{R,S}$ and $\mathbf{W}_{R,S}$ are then represented at low precision for efficient hardware execution, yielding the representations $\widetilde{\mathbf{X}}_{R,S}$ and $\widetilde{\mathbf{W}}_{R,S}$, respectively. To minimize the error in the resulting activation–weight product, we optimize $\mathbf{R}$ and $\mathbf{S}$ using the following objective.

$$\mathcal{L}(\mathbf{R},\mathbf{S}) = \left\|\mathbf{X}_{R,S}\mathbf{W}_{R,S}^\top - \widetilde{\mathbf{X}}_{R,S}\widetilde{\mathbf{W}}_{R,S}^\top\right\|_F^2, \tag{1}$$

where $\widetilde{\mathbf{X}}_{R,S}$ denotes the FP-quantized representation of $\mathbf{X}_{R,S}$, and $\widetilde{\mathbf{W}}_{R,S}$ denotes the clustered representation of $\mathbf{W}_{R,S}$. Specifically, during the optimization of $\mathbf{R}$ and $\mathbf{S}$, we construct $\widetilde{\mathbf{W}}_{R,S}$ at each step using row-wise K-means clustering. Since the clustering operation is non-differentiable, we utilize a straight-through estimator (STE) to propagate gradients through it. We further parameterize $\mathbf{R}$ using the Cayley transform, which preserves orthogonality by construction (Liu et al., 2025).

3.2 LAYER-OUTPUT OBJECTIVE DECOMPOSITION

Given the optimized $\mathbf{R}$ and $\mathbf{S}$, we investigate Eq. 1 to derive objectives for the weight-clustering and activation-quantization stages. For notational simplicity, we use $\mathbf{X} = \mathbf{X}_{R,S}$ and $\mathbf{W} = \mathbf{W}_{R,S}$ to denote the transformed activations and weights, respectively. Equation 1 can be reformulated as:

$$\mathcal{L} = \sum_{i=1}^{d_{\text{out}}} \left[\underbrace{\Delta\mathbf{W}_{i,:}\mathbf{H}_q\Delta\mathbf{W}_{i,:}^\top}_{(1)} + \underbrace{2\Delta\mathbf{W}_{i,:}\widetilde{\mathbf{X}}^\top\Delta\mathbf{X}\mathbf{W}_{i,:}^\top}_{(2)} + \underbrace{\mathbf{W}_{i,:}\Delta\mathbf{X}^\top\Delta\mathbf{X}\mathbf{W}_{i,:}^\top}_{(3)} \right], \tag{2}$$

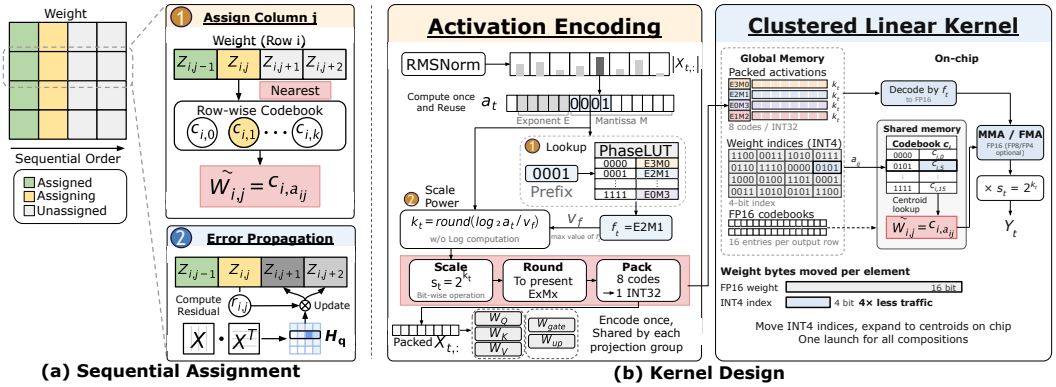

Figure 2: Two key mechanisms in CoBIT. (a) Sequential error-compensated assignment for a fixed row-wise codebook. (b) PhaseLUT indexing and the subsequent low-precision kernel path.

where $\widetilde{\mathbf{X}} = \widetilde{\mathbf{X}}_{R,S}$ and $\widetilde{\mathbf{W}} = \widetilde{\mathbf{W}}_{R,S}$ denote the FP-quantized representation of $\mathbf{X}_{R,S}$ and the clustered representation of $\mathbf{W}_{R,S}$, respectively. We define $\Delta\mathbf{W} = \mathbf{W} - \widetilde{\mathbf{W}}$ and $\Delta\mathbf{X} = \mathbf{X} - \widetilde{\mathbf{X}}$ as the weight-clustering and activation-quantization errors. $\mathbf{H}_q = \widetilde{\mathbf{X}}^\top \widetilde{\mathbf{X}} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}$ denotes the Hessian associated with the objective in Equation 1, and $\mathbf{W}_{i,:}$ denotes the i-th row of $\mathbf{W}$.

The first component, $\Delta\mathbf{W}_{i,:}\mathbf{H}_q\Delta\mathbf{W}_{i,:}^\top$, depends only on the weight-clustering error $\Delta\mathbf{W}_{i,:}$ and is independent of the activation-quantization error. In contrast, the last component, $\mathbf{W}_{i,:}\Delta\mathbf{X}^\top\Delta\mathbf{X}\mathbf{W}_{i,:}^\top$, depends only on the activation-quantization error $\Delta\mathbf{X}$ and is therefore determined by the activation quantization strategy. The second component couples weight-clustering and activation-quantization errors and therefore cannot be optimized independently within either stage. Incorporating it throughout the pipeline would require jointly coordinating codebook updates, weight assignment, and token-wise composition selection. We found that including this component in the codebook update provides no consistent improvement; Appendix A.1.3 provides further analysis. We therefore use the first and third components to optimize weight clustering and activation quantization, respectively, while retaining the full objective for $\mathbf{R}$ and $\mathbf{S}$ optimization.

3.3 Codebook and Assignment Optimization for Weight Clustering

In this section, we design a weight-clustering algorithm to minimize the first component, $\Delta\mathbf{W}_{i,:}\mathbf{H}_q\Delta\mathbf{W}_{i,:}^\top$, in Equation 2. Specifically, we cluster each row of the weight matrix independently. The clustered weight row $\widetilde{\mathbf{W}}_{i,:}$ can be represented as $\widetilde{\mathbf{W}}_{i,:}^\top = \mathbf{A}_i\mathbf{c}_i$, where $\mathbf{c}_i \in \mathbb{R}^K$ contains the K codebook centroids and $\mathbf{A}_i \in \{0,1\}^{d_{\text{in}} \times K}$ is the corresponding binary assignment matrix. The j-th row of $\mathbf{A}_i$ is one-hot and selects the centroid assigned to $\mathbf{W}_{ij}$. Under this parameterization, the row-wise weight-clustering error becomes $\Delta\mathbf{W}_{i,:}^\top = \mathbf{W}_{i,:}^\top - \mathbf{A}_i\mathbf{c}_i$. Substituting this representation into $\Delta\mathbf{W}_{i,:}\mathbf{H}_q\Delta\mathbf{W}_{i,:}^\top$ yields the following row-wise optimization problem:

$$(\mathbf{A}_i^\star, \mathbf{c}_i^\star) = \underset{\mathbf{A}_i, \mathbf{c}_i}{\operatorname{argmin}} \left(\mathbf{W}_{i,:}^\top - \mathbf{A}_i\mathbf{c}_i\right)^\top \mathbf{H}_q \left(\mathbf{W}_{i,:}^\top - \mathbf{A}_i\mathbf{c}_i\right). \tag{3}$$

To optimize Eq. 3, we alternate between codebook and assignment updates, as described below.

Assignment optimization. With the codebooks fixed, we optimize the assignment matrix $\mathbf{A}_i$ using GPTQ-style sequential error compensation (Frantar et al., 2023). Figure 2(a) illustrates the assignment and error propagation process. Unlike GPTQ, which projects weights onto a fixed quantization grid, CoBIT assigns each weight to a centroid in its learned row-wise codebook and performs error compensation using $\mathbf{H}_q = \widetilde{\mathbf{X}}^\top\widetilde{\mathbf{X}}$, computed from the quantized transformed activations. At the start of each assignment update, we initialize $\mathbf{Z} = \mathbf{W}$ as a working copy of $\mathbf{W}$ and process its columns sequentially. Each assignment introduces a clustering error, which is compensated through updates to the remaining unassigned columns. Specifically, after assigning column j, we update $Z_{ij'}$ for each unassigned column $j' > j$ to account for this error. For column j, we first map the current compensated value Z_{ij} to its nearest centroid in the codebook of row i:

$$a_{ij} = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmin}} |Z_{ij} - c_{ik}|, \qquad \widetilde{\mathbf{W}}_{ij} = c_{i,a_{ij}} \tag{4}$$

where c_{ik} denotes the k-th centroid in $\mathbf{c}_i$, and a_{ij} denotes the centroid index assigned to Z_{ij}, such that $(\mathbf{A}_i)_{j,a_{ij}} = 1$. The resulting clustering error, $Z_{ij} - \widetilde{\mathbf{W}}_{ij}$, is then propagated to the remaining unassigned columns $Z_{ij'}$ using GPTQ-style error compensation based on $\mathbf{H}_q$.

Codebook optimization. With the assignments $\mathbf{A}_i$ fixed, expanding Equation 3 yields the terms involving the codebook $\mathbf{c}_i$: $\mathbf{c}_i^\top \mathbf{A}_i^\top \mathbf{H}_q \mathbf{A}_i \mathbf{c}_i - 2\mathbf{c}_i^\top \mathbf{A}_i^\top \mathbf{H}_q \mathbf{W}_{i,:}^\top$. This objective is quadratic in $\mathbf{c}_i$, and setting its gradient to zero gives the linear system $\mathbf{A}_i^\top \mathbf{H}_q \mathbf{A}_i \mathbf{c}_i = \mathbf{A}_i^\top \mathbf{H}_q \mathbf{W}_{i,:}^\top$. To improve numerical stability, we introduce ridge regularization, reducing the sensitivity of the optimized centroids to small variations in the estimated $\mathbf{H}_q$:

$$\mathbf{c}_i \leftarrow \left(\mathbf{A}_i^\top \mathbf{H}_q \mathbf{A}_i + \lambda \bar{h}_q \mathbf{I}_K\right)^{-1} \mathbf{A}_i^\top \mathbf{H}_q \mathbf{W}_{i,:}^\top, \qquad \bar{h}_q = \frac{\mathrm{tr}(\mathbf{H}_q)}{d_{\mathrm{in}}}, \tag{5}$$

where $\bar{h}_q$ denotes the average diagonal entry of $\mathbf{H}_q$, λ controls the ridge regularization strength, and $\mathbf{I}_K$ is the $K \times K$ identity matrix. For fixed assignments, this update minimizes the row-wise objective in Equation 3 with the additional penalty $\lambda \bar{h}_q |\mathbf{c}_i|_2^2$. By retaining the off-diagonal entries of $\mathbf{H}_q$, the update captures interactions among clustering errors across different weight columns. We alternate between assignment and codebook updates.

3.4 Mixed Composition and PhaseLUT

In this section, we optimize low-precision FP activation quantization by minimizing the third component of Equation 2, $\mathbf{W}_{i,:} \Delta\mathbf{X}^\top \Delta\mathbf{X} \mathbf{W}_{i,:}^\top$, whose sum over the weight rows equals $\|\Delta\mathbf{X}\mathbf{W}^\top\|_F^2$. Specifically, we optimize this objective by dynamically selecting the FP composition, i.e., the allocation of exponent and mantissa bits. Different compositions provide different trade-offs between dynamic range and numerical precision, making the optimal choice dependent on the activation distribution. Specifically, let $\mathcal{W}$ denote either $\{\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v\}$ or $\{\mathbf{W}_{\mathrm{gate}}, \mathbf{W}_{\mathrm{up}}\}$. For each token, we select the optimal allocation of exponent and mantissa bits, denoted by e and m, under a fixed FP bit width $b = 1 + e + m$ (one extra bit for sign), by minimizing the total output error:

$$(e_t^\star, m_t^\star) = \underset{(e,m)\in\mathcal{C}_b}{\mathrm{argmin}} \, \mathbb{E}_{\mathbf{W}\in\mathcal{W}} \left[\left\| \Delta\mathbf{X}_t^{(e,m)} \mathbf{W}^\top \right\|_2^2 \right], \tag{6}$$

where $\mathbf{X}_{t,:}$ denotes the row corresponding to t, $\Delta\mathbf{X}_t^{(e,m)}$ denotes the activation quantization error under composition (e, m), and $\mathcal{C}_b$ contains all candidate compositions under the b-bit budget. The expectation uniformly averages the output error across the weights in $\mathcal{W}$.

A naive approach to solving Equation 6 is to evaluate all candidate compositions for each token during inference and select the one with the lowest output error. However, this exhaustive search introduces substantial runtime overhead. We therefore seek a simple statistic that can be efficiently computed and used to guide composition selection. After rotation and scaling, we observe that the transformed activations entering the linear projections approximately follow a standard normal distribution, with token-wise minimum and maximum values of similar magnitude (Appendix A.2). These observations suggest a simple approximation to each token's activation distribution. Let μ_t and σ_t^2 denote the mean and variance of the activation vector for token t, respectively. We approximate each token's distribution using a tuple $\mathbf{s}_t$ comprising its minimum, maximum, mean, and variance, as follows:

$$\mathbf{s}_t = \left(\min(\mathbf{X}_{t,:}), \max(\mathbf{X}_{t,:}), \mu_t, \sigma_t^2\right) \approx (-a_t, a_t, 0, 1), \qquad a_t = \max|\mathbf{X}_{t,:}|. \tag{7}$$

Under this approximation, the token-wise absmax a_t becomes the key statistic for composition selection. In particular, a_t characterizes the effective dynamic range of the token, which directly affects the trade-off between exponent range and mantissa precision: tokens with different a_t may therefore favor different (e, m) compositions. We therefore map a_t into a small number of bins and assign one composition to each bin. Importantly, a_t is measured before the per-token scaling used for activation quantization, so the binning scheme should remain consistent after scaling. A general real-valued scale can change the mantissa of a_t, whereas a power-of-two (PoT) scale changes only its exponent while preserving its mantissa. This motivates using the mantissa of a_t as a scale-invariant feature for binning. Specifically, we introduce *PhaseLUT*, a lookup table indexed by the first p mantissa bits of a_t. Each entry stores the profiled composition that minimizes the average error in Equation 6 over calibration tokens mapped to the corresponding index:

$$\mathrm{PhaseLUT}_\varphi = \underset{(e,m)\in\mathcal{C}_b}{\mathrm{argmin}} \, \mathbb{E}_{\mathbf{X},\mathbf{W}} \left[\left\| \Delta\mathbf{X}^{(e,m)} \mathbf{W}^\top \right\|_2^2 \right], \tag{8}$$

where $\mathbf{X}$ ranges over the calibration tokens assigned to entry φ. During calibration, we compute φ_t for each token and group tokens sharing the same index φ to evaluate Equation 8. At inference, we extract φ_t in the same manner and directly retrieve the corresponding FP composition from PhaseLUT, eliminating the need for online search over candidate compositions.

3.5 Quantization Kernel Design

While PhaseLUT eliminates online composition search, a straightforward implementation would still require separate passes for lookup and mixed-composition activation encoding. Moreover, PoT scales enable efficient bit-wise operations and eliminate expensive arithmetic operations, but these benefits are lost with a conventional implementation. To address both issues, we design a **custom CUDA kernel** that integrates PhaseLUT lookup, PoT scale computation, activation encoding, and clustered linear computation into a unified execution path. Figure 2(b) summarizes the runtime path from PhaseLUT lookup to activation encoding and clustered linear computation. We set $s = 2^k$, where $k = \mathrm{round}(\log_2(a_t/v_{\max}))$ and $v_{\max}$ is the maximum magnitude representable by the selected FP composition. First, $\log_2(a_t/v_{\max}) = \log_2 a_t - \log_2 v_{\max}$ removes the runtime division. Writing $a_t = m_{a_t} 2^{e_{a_t}}$ gives $\log_2 a_t = e_{a_t} + \log_2 m_{a_t}$, where e_{a_t} is read directly from the FP representation. Since k only needs the rounded result and $v_{\max}$ is fixed for each composition, we precompute the threshold on m_{a_t} that separates the two possible values of k. One comparison with this threshold then replaces the remaining logarithm. Second, PoT scales also simplify activation scaling. Dividing by a power of two is equivalent to shifting the activation's floating-point exponent. Thus we utilize the bit-wise exponent shift to avoid floating-point division. The kernel also reuses a_t for PhaseLUT lookup and scale selection, avoiding an additional scan of the activation row.

Token-wise mixed compositions allow different token rows to use different FP4 formats, but a naive implementation would require regrouping rows by format and launching separate kernels. We avoid this by using the same 4-bit packing layout for all compositions, with eight FP4 activation values packed into one INT32 value. For token t, PhaseLUT selects $f_t = (e_t, m_t)$. The encoder uses f_t to round and clip the activations before packing them. The clustered linear kernel receives the packed activations together with f_t and the scale exponent k_t, and uses f_t to decode each 4-bit value. As a result, the kernel can process different compositions without regrouping the rows.

At inference, we fuse PhaseLUT-guided activation encoding into the RMSNorm preceding each projection group. The packed activations are reused across projections, reducing the number of encodings from five to two per layer. The clustered linear kernel then consumes packed FP4 activations and INT4 weight indices, with one FP16 codebook per weight row. The kernel retrieves centroids on chip for multiplication without materializing the dense weight matrix. Our implementation supports FP16, FP8, and FP4 arithmetic depending on hardware, while the FP8 and FP4 paths require quantizing the FP16 codebook values. We report FP16 latency to align with our accuracy evaluation.

4 Experiments

We evaluate CoBIT on four general-purpose Qwen3 models with 4B, 8B, 14B, and 32B parameters (Yang et al., 2025). Evaluation metrics include perplexity on WikiText-2 and C4 (Merity et al., 2016; Raffel et al., 2023), forward KL divergence on WikiText-2, and accuracy on ARC, PIQA, HellaSwag, and MMLU (Clark et al., 2018; Bisk et al., 2019; Zellers et al., 2019; Hendrycks et al., 2021). Baselines include matched-setting adaptations of QuaRot with GPTQ, ParoQuant, LNQ, SqueezeLLM, and CodeQuant (Frantar et al., 2023; Ashkboos et al., 2024; Liang et al., 2026; Kim et al., 2025; Yin et al., 2026). Further details are provided in Appendices A.4.1, A.4.3, and A.5.

We additionally evaluate CoBIT on reasoning tasks using three models: Qwen3-4B-Thinking-2507, Llama-3.1-Nemotron-Nano-4B, and DeepSeek-R1-Distill-Llama-8B (Yang et al., 2025; Sreenivas et al., 2024; Guo et al., 2025). This evaluation covers mathematical and scientific reasoning on AIME25, AIME26, and GPQA (Zhang & Math-AI, 2025; 2026; Rein et al., 2023), as well as code generation on HumanEval and LiveCodeBench (Chen et al., 2021; Jain et al., 2024).

For hyperparameters, we optimize the rotation and scaling parameters (Section 3.1) using AdamW (Loshchilov & Hutter, 2019) with a learning rate of 10^{-4}, zero weight decay, and a cosine learning-rate schedule. For clustering, we set $\lambda = 10^{-4}$, as described in Section 3.3. For PhaseLUT, we use $p = 4$ mantissa-prefix bits, yielding 16 entries, based on the parameter sweep in

Table 1: A4W3 and A4W2 performance on Qwen3 models. FKL and Hella denote forward KL divergence and HellaSwag. The best quantized result within each model and bit width is bold.

Models	W Bits	Methods	FKL ↓	Wiki2 ↓	C4 ↓	ARC ↑	PIQA ↑	Hella ↑	MMLU ↑	Avg ↑
Qwen3-4B	–	BF16	–	14.39	19.86	0.899	0.815	0.733	0.750	0.799
	3	$\text{QuaRot}_{\text{GPTQ}}$	0.703	22.75	31.09	0.714	0.549	0.474	0.568	0.576
		SqueezeLLM	1.060	29.31	39.16	0.598	0.520	0.392	0.467	0.494
		CodeQuant	0.814	26.04	35.53	0.775	0.661	0.556	0.535	0.632
		CoBIT	**0.351**	**17.21**	**22.84**	**0.850**	**0.738**	**0.665**	**0.654**	**0.727**
	2	$\text{QuaRot}_{\text{GPTQ}}$	4.573	328.03	400.35	0.103	0.163	0.090	0.082	0.109
		SqueezeLLM	3.733	195.76	265.43	0.074	0.155	0.071	0.040	0.085
		CodeQuant	3.086	197.48	247.02	0.020	0.041	0.027	0.020	0.027
		CoBIT	**0.738**	**21.35**	**29.22**	**0.675**	**0.669**	**0.492**	**0.421**	**0.564**
Qwen3-8B	–	BF16	–	10.05	15.43	0.917	0.860	0.798	0.773	0.837
	3	$\text{QuaRot}_{\text{GPTQ}}$	0.417	12.61	18.32	0.841	0.711	0.667	0.660	0.720
		SqueezeLLM	0.720	16.01	23.64	0.806	0.701	0.518	0.372	0.599
		CodeQuant	0.524	14.18	20.82	0.845	0.684	0.628	0.658	0.704
		CoBIT	**0.266**	**11.58**	**17.31**	**0.884**	**0.816**	**0.736**	**0.729**	**0.791**
	2	$\text{QuaRot}_{\text{GPTQ}}$	3.005	71.07	79.38	0.052	0.107	0.030	0.031	0.055
		SqueezeLLM	3.225	183.98	257.19	0.036	0.064	0.017	0.021	0.034
		CodeQuant	2.069	52.78	68.05	0.072	0.056	0.039	0.037	0.051
		CoBIT	**0.545**	**13.90**	**20.75**	**0.819**	**0.563**	**0.595**	**0.595**	**0.643**
Qwen3-32B	–	BF16	–	7.86	12.45	0.941	0.910	0.859	0.835	0.886
	3	$\text{QuaRot}_{\text{GPTQ}}$	0.549	9.71	14.69	0.912	0.845	0.758	0.738	0.813
		SqueezeLLM	0.830	12.10	18.32	0.897	0.805	0.653	0.396	0.688
		CodeQuant	0.532	9.84	15.07	0.924	0.876	0.793	0.788	0.845
		CoBIT	**0.351**	**9.10**	**14.03**	**0.933**	**0.885**	**0.819**	**0.803**	**0.860**
	2	$\text{QuaRot}_{\text{GPTQ}}$	3.977	98.18	97.38	0.088	0.119	0.068	0.090	0.091
		SqueezeLLM	3.343	84.58	106.06	0.085	0.179	0.078	0.066	0.102
		CodeQuant	1.209	16.09	24.30	0.795	0.693	0.480	0.471	0.610
		CoBIT	**0.601**	**10.42**	**15.93**	**0.912**	**0.855**	**0.801**	**0.739**	**0.827**

Table 2: A8W4 performance on reasoning models. A25, A26, HE, and LCB denote AIME25, AIME26, HumanEval, and LiveCodeBench. The best quantized result within each model is bold.

Method	Qwen3-4B-Thinking-2507						Llama-3.1-Nemotron-Nano-4B-v1.1						DeepSeek-R1-Distill-Llama-8B					
	A25	A26	GPQA	HE	LCB	Avg	A25	A26	GPQA	HE	LCB	Avg	A25	A26	GPQA	HE	LCB	Avg
BF16	0.692	0.725	0.702	0.970	0.739	0.766	0.546	0.621	0.505	0.860	0.604	0.627	0.350	0.375	0.510	0.866	0.534	0.527
$\text{QuaRot}_{\text{GPTQ}}$	0.617	0.633	0.626	0.951	0.679	0.701	0.521	0.525	0.424	0.878	0.542	0.578	0.321	0.338	0.465	0.835	0.475	0.487
SqueezeLLM	0.550	0.638	0.616	0.939	0.648	0.678	0.442	0.463	0.475	0.854	0.507	0.548	0.300	0.283	0.389	0.768	0.408	0.430
CodeQuant	0.583	0.613	0.591	0.945	0.627	0.672	0.433	0.454	0.419	0.848	0.486	0.528	0.308	0.279	0.384	0.842	0.413	0.445
CoBIT	**0.671**	**0.717**	**0.641**	**0.970**	**0.701**	**0.740**	**0.558**	**0.583**	**0.505**	**0.896**	**0.574**	**0.624**	**0.375**	**0.367**	**0.470**	**0.878**	**0.483**	**0.515**

Figure 8. For calibration, we use 512 sequences of up to 512 tokens for general-purpose models. For reasoning models, we add 512 math and code sequences, yielding $1{,}024$ sequences in total. CoBIT's complete offline pipeline takes 0.30–2.40 hours on two NVIDIA H200 GPUs. At inference, we quantize the Q/K/V and gate/up projections while retaining the remaining modules in BF16.

4.1 Main Results

Table 1 reports CoBIT performance across different codebook sizes. For simplicity, we denote the codebook size by its corresponding bit width. For example, W3 indicates a codebook with $2^3 = 8$ centroids. CoBIT consistently outperforms all baselines at 3 bits. The advantage is largest on Qwen3-4B, where CoBIT raises average downstream accuracy from 63.2% with CodeQuant to 72.7%, while reducing WikiText-2 perplexity from 26.04 to 17.21 and C4 perplexity from 35.53 to 22.84. At extremely low precision (2 bits), all baselines suffer substantial performance degradation, while CoBIT remains stable. On Qwen3-32B, its average accuracy is only 5.9% below BF16 and exceeds that of 3-bit QuaRot. Finally, under the A4W4 setting, CoBIT achieves the best or second-best result among the quantized methods in 30 of the 32 model–metric comparisons. A4W4 and Qwen3-14B results are provided in Appendix A.6.1 and Table 17 due to the page limit.

Table 3 isolates the impact on weight quantization by keeping activations in BF16. At 2 bits, CoBIT exceeds LNQ by 3.9% and 3.5% in average accuracy on Qwen3-8B and Qwen3-14B, respectively. CoBIT and LNQ use per-channel quantization, while ParoQuant uses a finer group size of 128. CoBIT still leads in both 2-bit settings and trails ParoQuant by only 0.2% in the 3-bit setting.

Although CoBIT is not specifically designed for reasoning tasks, it achieves the highest average accuracy among the quantized methods across all three models. In Table 2, CoBIT outperforms the

Table 3: Weight-only results with BF16 activations. Best quantized results are bold.

	W Bits	Method	FKL↓	Wiki2↓	C4↓	Avg↑
Qwen3-8B	3	$\text{QuaRot}_{\text{GPTQ}}$	0.265	11.50	17.05	0.762
		SqueezeLLM	0.526	13.53	20.00	0.705
		ParoQuant	0.110	10.70	16.13	0.812
		LNQ	0.119	10.72	16.20	0.823
		CodeQuant	0.344	12.37	18.37	0.751
		CoBIT	**0.106**	**10.55**	**16.05**	**0.823**
	2	$\text{QuaRot}_{\text{GPTQ}}$	2.479	46.15	54.68	0.041
		SqueezeLLM	2.868	129.84	162.73	0.013
		ParoQuant	0.618	16.71	22.28	0.571
		LNQ	0.550	16.34	19.77	0.686
		CodeQuant	1.723	40.62	50.92	0.131
		CoBIT	**0.405**	**12.74**	**19.23**	**0.725**
Qwen3-14B	3	$\text{QuaRot}_{\text{GPTQ}}$	0.208	9.98	14.88	0.836
		SqueezeLLM	0.429	11.36	17.25	0.808
		ParoQuant	0.088	9.45	14.41	**0.861**
		LNQ	0.092	9.49	14.34	0.858
		CodeQuant	0.213	10.25	15.57	0.846
		CoBIT	**0.085**	**9.43**	**14.31**	0.859
	2	$\text{QuaRot}_{\text{GPTQ}}$	1.635	23.71	29.67	0.044
		SqueezeLLM	2.257	55.17	69.47	0.028
		ParoQuant	0.500	13.61	18.19	0.759
		LNQ	0.427	12.33	16.88	0.786
		CodeQuant	0.800	15.99	22.76	0.666
		CoBIT	**0.347**	**11.18**	**16.56**	**0.821**

Table 4: Codebook (CB) and assignment with BF16 activations. Diag./Full denote Hessian matrix used for codebook construction. Near./Seq. denote assignment methods.

	K	CB	Assign.	FKL↓	Wiki2↓	C4↓	Avg↑
Qwen3-8B	8	Diag.	Near.	0.557	15.87	22.11	0.686
		Full	Near.	0.323	11.64	17.55	0.778
		Diag.	Seq.	0.114	10.60	16.22	0.819
		Full	**Seq.**	**0.106**	**10.55**	**16.05**	**0.823**
	4	Diag.	Near.	2.828	104.81	135.28	0.150
		Full	Near.	1.701	32.16	44.67	0.336
		Diag.	Seq.	0.515	13.55	20.60	0.704
		Full	**Seq.**	**0.405**	**12.74**	**19.23**	**0.725**
Qwen3-14B	8	Diag.	Near.	0.494	12.93	18.65	0.774
		Full	Near.	0.175	9.51	14.70	0.845
		Diag.	Seq.	0.088	**9.43**	14.47	**0.863**
		Full	**Seq.**	**0.085**	**9.43**	**14.31**	0.859
	4	Diag.	Near.	2.203	46.38	64.48	0.094
		Full	Near.	1.107	18.84	26.44	0.489
		Diag.	Seq.	0.411	11.73	17.72	0.814
		Full	**Seq.**	**0.347**	**11.18**	**16.56**	**0.821**

strongest quantized baseline by 3.9%, 4.6%, and 2.8% on Qwen3-4B-Thinking-2507, Llama-3.1-Nemotron-Nano-4B-v1.1, and DeepSeek-R1-Distill-Llama-8B, respectively.

4.2 Understanding Hessian-Aware Clustering

In this section, we study the impact of Hessian-aware clustering described in Section 3.3. Table 4 compares codebooks constructed using only the diagonal of the Hessian (*Diag.*) or the full Hessian (*Full*), under independent nearest-centroid (*Near.*) and sequential error-compensated (*Seq.*) assignment. Across both models and both K values, *Seq.* improves average accuracy more than *Full* alone. For example, on Qwen3-8B with $K = 4$, *Seq.* raises average accuracy from 0.150 to 0.704 with *Diag.*, whereas *Full* raises it to 0.336 with *Near.*. Combining *Full* and *Seq.* generally yields the lowest FKL and C4 perplexity, showing the effectiveness of both designs. To further examine the role of off-diagonal Hessian information, Figure 3 varies the ratio of retained off-diagonal Hessian mass. Its effect is marginal at $K = 16$ but becomes more pronounced at smaller K, indicating that off-diagonal interactions matter more when fewer centroids are available.

4.3 Component Analysis

Table 5: Ablation studies with $K = 8$. The "forward" setting sequentially adds components from left to right, while the "leave-one-out" setting removes each component from full CoBIT. Stage I uses BF16 activations, while Stage II introduces FP4 activation quantization.

		Stage I: Weight Quantization (BF16 Activations)				Stage II: Activation Quantization (FP4)		
Model	Analysis	Plain K-means	Rotation & scaling	Hessian-aware codebook	Sequential assignment	FP4 E2M1 Quantization	PhaseLUT	Full CoBIT
Qwen3-8B	Forward →	0.648	0.670 (+0.021)	0.778 (+0.109)	0.823 (+0.045)	0.790 (-0.032)	0.791 (+0.001)	**0.791**
	Leave-one-out ←	—	0.391 (-0.400)	0.786 (-0.005)	0.727 (-0.064)	0.823 (+0.032)	0.790 (-0.001)	**0.791**
Qwen3-14B	Forward →	0.805	0.798 (-0.007)	0.845 (+0.047)	0.859 (+0.014)	0.833 (-0.026)	0.835 (+0.002)	**0.835**
	Leave-one-out ←	—	0.798 (-0.037)	0.839 (+0.004)	0.819 (-0.016)	0.859 (+0.024)	0.833 (-0.002)	**0.835**

Table 5 isolates weight quantization with BF16 activations before adding FP4 activation quantization. Hessian-aware codebook construction and sequential assignment raise average accuracy from 0.670 to 0.823 on Qwen3-8B and from 0.798 to 0.859 on Qwen3-14B. Codebook construction gives the larger forward gain, but sequential assignment has the larger leave-one-out effect. Removing rotation and scaling also substantially reduces accuracy on Qwen3-8B, from 0.791 to 0.391.

With FP4 activations, PhaseLUT maintains similar average accuracy to E2M1, with gains of 0.1–0.2% points. The mantissa-bit sweep shows larger perplexity gains on smaller models: with a 16-

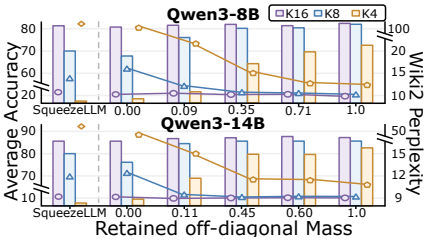

Figure 3: Effect of retained off-diagonal Hessian mass. Bars show average accuracy, and curves show Wiki2 perplexity.

Table 6: Aggregate kernel speedup of CoBIT. For each method, latency is averaged across the four tested Qwen3 models. Entries are $T_{\text{baseline}}/T_{\text{CoBIT}}$.

GPU	Baseline	Batch size			
		1	2	4	8
RTX 5060 Ti	FP16 cuBLAS	3.66×	3.36×	3.00×	2.33×
	SqueezeLLM	1.05×	1.52×	2.33×	3.30×
RTX A6000	FP16 cuBLAS	2.73×	2.50×	2.15×	1.64×
	SqueezeLLM	1.03×	1.49×	2.18×	3.01×

entry LUT, WikiText-2/C4 perplexity decreases from $16.87/22.58$ to $16.76/22.52$ on Qwen3-4B and from $10.90/16.65$ to $10.86/16.62$ on Qwen3-8B, showing an additional activation-side benefit without sacrificing downstream accuracy. Appendix A.2.4 reports the complete sweep results.

4.4 System Efficiency of CoBIT

We evaluate CoBIT's A4W4 representation with FP16 computation at both the kernel and end-to-end levels. Because the codebook centroids are stored in FP16 and native FP4×FP16 matrix multiplication is unavailable on the evaluated GPUs, the kernel decodes packed FP4 activations to FP16 for multiplication without quantizing the centroids (Section 3.5). All reported CoBIT latencies include the complete execution path, including PhaseLUT lookup, activation encoding, format conversion, and clustered linear computation. Table 6 reports kernel-level speedups across batch sizes 1, 2, 4, and 8. CoBIT achieves 1.64–3.66× speedup over FP16 cuBLAS. Compared with SqueezeLLM, CoBIT achieves 1.03–1.05× speedup at batch 1, increasing to 3.01–3.30× at batch 8.

Table 7: Decoding speedup of A4W4 CoBIT on Qwen3-4B and Qwen3-8B with a batch size of 1. Each entry reports $T_{\text{baseline}}/T_{\text{CoBIT}}$, where higher is better. OOM indicates out of GPU memory.

GPU	Model	FP16 cuBLAS A16W16	QuaRot$_{\text{GPTQ}}$ CUTLASS A4W4	SqueezeLLM Official CUDA A16W4	SqueezeLLM Official CUDA A4W4	ParoQuant AutoAWQ CUDA A16W4	ParoQuant AutoAWQ CUDA A4W4	ParoQuant AutoAWQ Triton A16W4	ParoQuant AutoAWQ Triton A4W4	ParoQuant Marlin A16W4
RTX 5060 Ti	Qwen3-4B	1.53×	5.56×	1.08×	1.28×	1.08×	1.28×	1.27×	1.47×	1.06×
	Qwen3-8B	OOM	5.73×	1.06×	1.18×	1.09×	1.19×	1.31×	1.43×	1.01×
RTX A6000	Qwen3-4B	1.44×	1.34×	1.13×	1.42×	1.12×	1.43×	1.37×	1.68×	1.07×
	Qwen3-8B	1.51×	1.25×	1.11×	1.27×	1.12×	1.31×	1.38×	1.58×	1.03×

At batch 1, CoBIT is 1.44–1.53× faster than dense FP16, 1.25–5.73× faster than QuaRot, and 1.18–1.42× faster than SqueezeLLM. QuaRot uses INT8 MMA on the RTX 5060 Ti, where native INT4 MMA is unavailable. Compared with ParoQuant's AutoAWQ CUDA and Triton backends, CoBIT achieves 1.19–1.43× and 1.43–1.68× speedups at batch 1, respectively, and 1.07–1.17× and 1.22–1.32× at batch 8. CoBIT achieves comparable end-to-end latency to vLLM's optimized Marlin A16W4 implementation across the evaluated batch sizes. Analysis of the A6000 measurements reveals two main sources of speedup. First, CoBIT's packed 4-bit weight indices reduce memory traffic relative to dense FP16, while achieving higher effective bandwidth than SqueezeLLM and ParoQuant's AWQ kernel. Second, fusing activation encoding into RMSNorm and reusing the encoded activations provides an additional gain. Appendix A.6.3 provides the complete results.

5 Conclusion

In summary, CoBIT improves extreme low-bit weight quantization through Hessian-aware codebook construction and error-compensated sequential assignment. For activations, PhaseLUT enables mixed-composition quantization with lower FKL and perplexity while eliminating runtime composition search. An integrated kernel combines activation quantization and clustered linear computation without materializing the dense weights for efficient inference. A current limitation is MoE calibration, where uneven token routing can bias Hessian estimates for under-sampled experts. Initial results on MoE models are promising, while a systematic study remains future work.

AI USE STATEMENT

The research ideas and core method are entirely the authors' own, and no AI was used to generate research ideas. Generative AI was used for three purposes: (1) to implement approximately 60% of the code under the authors' direction, (2) to polish the paper's language, and (3) to check mathematical derivations developed by the authors and improve their presentation. All AI-assisted content was reviewed by all authors, and AI-generated code was additionally validated using author-written test cases. The authors take responsibility for the final content of this work, including any text, claims, code, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies post-training quantization of publicly available large language models using publicly available calibration and evaluation data. It involves no human subjects, no private or sensitive data, and no new data collection. Quantization does not change the intended use of the underlying models, and we are not aware of any ethical concerns specific to this work beyond those already associated with the base models.

REPRODUCIBILITY STATEMENT

All models and datasets used in this work are publicly available and can be used directly. Calibration data sources and sampling are described in Appendix A.4.1, evaluation methods in Appendix A.4.3, and baseline configurations in Appendix A.5. Upon acceptance, we will release the complete codebase, including the quantization pipeline and the kernels. We will also upload the quantized checkpoints to a public model hub so that the reported evaluation results can be verified directly.

REFERENCES

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms, 2024. URL https://arxiv.org/abs/2404.00456.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language, 2019. URL https://arxiv.org/abs/1911.11641.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge, 2018. URL https://arxiv.org/abs/1803.05457.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023. URL https://arxiv.org/abs/2307.08691.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022. URL https://arxiv.org/abs/2208.07339.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023. URL https://arxiv.org/abs/2305.14314.

Zhen Dong, Zhewei Yao, Yaohui Cai, Daiyaan Arfeen, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. Hawq-v2: Hessian aware trace-weighted quantization of neural networks, 2019. URL https://arxiv.org/abs/1911.03852.

Jordan Dotzel, Gang Wu, Andrew Li, Muhammad Umar, Yun Ni, Mohamed S. Abdelfattah, Zhiru Zhang, Liqun Cheng, Martin G. Dixon, Norman P. Jouppi, Quoc V. Le, and Sheng Li. Fliqs: One-shot mixed-precision floating-point and integer quantization search, 2024. URL https://arxiv.org/abs/2308.03290.

Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. Extreme compression of large language models via additive quantization, 2024. URL https://arxiv.org/abs/2401.06118.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023. URL https://arxiv.org/abs/2210.17323.

Elias Frantar, Roberto L. Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. Marlin: Mixed-precision auto-regressive parallel inference on large language models, 2024. URL https://arxiv.org/abs/2408.11743.

Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvarna, Benjamin Feuer, Liangyu Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muennighoff, Shiye Su, Wanjia Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan Deng, Sarah Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak, Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill, Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy, Alexandros G. Dimakis, and Ludwig Schmidt. Openthoughts: Data recipes for reasoning models, 2025. URL https://arxiv.org/abs/2506.04178.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021. URL https://arxiv.org/abs/2009.03300.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Sambhav R. Jain, Albert Gural, Michael Wu, and Chris H. Dick. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks, 2020. URL https://arxiv.org/abs/1903.08066.

Jinuk Kim, Marwa El Halabi, Wonpyo Park, Clemens JS Schaefer, Deokjae Lee, Yeonhong Park, Jae W. Lee, and Hyun Oh Song. Guidedquant: Large language model quantization via exploiting end loss guidance, 2025. URL https://arxiv.org/abs/2505.07004.

Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization, 2024. URL https://arxiv.org/abs/2306.07629.

Andrey Kuzmin, Mart Van Baalen, Yuwei Ren, Markus Nagel, Jorn Peters, and Tijmen Blankevoort. Fp8 quantization: The power of the exponent, 2024. URL https://arxiv.org/abs/2208.09225.

Zhengyi Li, Cong Guo, Zhanda Zhu, Yangjie Zhou, Yuxian Qiu, Xiaotian Gao, Jingwen Leng, and Minyi Guo. Efficient adaptive activation rounding for post-training quantization, 2023. URL https://arxiv.org/abs/2208.11945.

Yesheng Liang, Haisheng Chen, Zihan Zhang, Song Han, and Zhijian Liu. Paroquant: Pairwise rotation quantization for efficient reasoning llm inference, 2026. URL https://arxiv.org/abs/2511.10645.

Haokun Lin, Haobo Xu, Yichen Wu, Jingzhi Cui, Yingtao Zhang, Linzhan Mou, Linqi Song, Zhenan Sun, and Ying Wei. Duquant: Distributing outliers via dual transformation makes stronger quantized llms, 2024. URL https://arxiv.org/abs/2406.01721.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration, 2026. URL https://arxiv.org/abs/2306.00978.

Fangxin Liu, Wenbo Zhao, Zhezhi He, Yanzhi Wang, Zongwu Wang, Changzhi Dai, Xiaoyao Liang, and Li Jiang. Improving neural network efficiency via post-training quantization with adaptive floating-point. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 5261–5270. IEEE, 2021.

Shih-yang Liu, Zechun Liu, Xijie Huang, Pingcheng Dong, and Kwang-Ting Cheng. Llm-fp4: 4-bit floating-point quantized transformers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 592–605. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.39. URL http://dx.doi.org/10.18653/v1/2023.emnlp-main.39.

Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant: Llm quantization with learned rotations, 2025. URL https://arxiv.org/abs/2405.16406.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016. URL https://arxiv.org/abs/1609.07843.

NVIDIA. NVIDIA announces Hopper architecture, the next generation of accelerated computing, March 2022. URL https://nvidianews.nvidia.com/news/nvidia-announces-hopper-architecture-the-next-generation-of-accelerated-computing.

NVIDIA. NVIDIA Blackwell platform arrives to power a new era of computing, March 2024. URL https://nvidianews.nvidia.com/news/nvidia-blackwell-platform-arrives-to-power-a-new-era-of-computing.

NVIDIA. NVFP4: 4-bit floating-point quantization. https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/features/low_precision_training/nvfp4/nvfp4.html, 2025. NVIDIA Transformer Engine Documentation.

Dahoon Park, Jahyun Koo, Sangwoo Hwang, and Jaeha Kung. Mx-safe: Versatile inference- and training-proof microscaling format with on-the-fly exponent and mantissa bit allocation, 2026. URL https://arxiv.org/abs/2605.24391.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023. URL https://arxiv.org/abs/1910.10683.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. Gpqa: A graduate-level google-proof q&a benchmark, 2023. URL https://arxiv.org/abs/2311.12022.

Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, Stosic Dusan, Venmugil Elango, Maximilian Golub, Alexander Heinecke, Phil James-Roxby, Dharmesh Jani, Gaurav Kolhe, Martin Langhammer, Ada Li, Levi Melnick, Maral Mesmakhosroshahi, Andres Rodriguez, Michael Schulte, Rasoul Shafipour, Lei Shao, Michael Siu, Pradeep Dubey, Paulius Micikevicius, Maxim Naumov, Colin Verrilli, Ralph Wittig, Doug Burger, and Eric Chung. Microscaling data formats for deep learning, 2023. URL https://arxiv.org/abs/2310.10537.

Siqing Song, Chuang Wang, Ruiqi Wang, Yi Yang, and Xu-Yao Zhang. Achieving binary weight and activation for llms using post-training quantization, 2025. URL https://arxiv.org/abs/2504.05352.

Sharath Turuvekere Sreenivas, Saurav Muralidharan, Raviraj Joshi, Marcin Chochowski, Ameya Sunil Mahabaleshwarkar, Gerald Shen, Jiaqi Zeng, Zijia Chen, Yoshi Suhara, Shizhe Diao, Chenhan Yu, Wei-Chun Chen, Hayley Ross, Oluwatobi Olabiyi, Ashwath Aithal, Oleksii Kuchaiev, Daniel Korzekwa, Pavlo Molchanov, Mostofa Patwary, Mohammad Shoeybi, Jan Kautz, and Bryan Catanzaro. Llm pruning and distillation in practice: The minitron approach, 2024. URL https://arxiv.org/abs/2408.11796.

Mingjie Sun, Xinlei Chen, J. Zico Kolter, and Zhuang Liu. Massive activations in large language models, 2024. URL https://arxiv.org/abs/2402.17762.

ModelScope Team. EvalScope: Evaluation framework for large models, 2024. URL https://github.com/modelscope/evalscope.

Maurice Weber, Daniel Fu, Quentin Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, Ben Athiwaratkun, Rahul Chalamala, Kezhen Chen, Max Ryabinin, Tri Dao, Percy Liang, Christopher Ré, Irina Rish, and Ce Zhang. Redpajama: an open dataset for training large language models, 2024. URL https://arxiv.org/abs/2411.12372.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models, 2024. URL https://arxiv.org/abs/2211.10438.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael W. Mahoney, and Kurt Keutzer. Hawqv3: Dyadic neural network quantization, 2021. URL https://arxiv.org/abs/2011.10680.

Xiangyang Yin, Xingyu Liu, Tianhua Xia, Bo Bao, Vithursan Thangarasa, Valavan Manohararajah, Eric Sather, and Sai Qian Zhang. Codequant: Unified clustering and quantization for enhanced outlier smoothing in low-precision mixture-of-experts, 2026. URL https://arxiv.org/abs/2604.10496.

Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. Turboquant: Online vector quantization with near-optimal distortion rate, 2025. URL https://arxiv.org/abs/2504.19874.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019. URL https://arxiv.org/abs/1905.07830.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025, 2025.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2026, 2026.

Yijia Zhang, Lingran Zhao, Shijie Cao, Wenqiang Wang, Ting Cao, Fan Yang, Mao Yang, Shanghang Zhang, and Ningyi Xu. Integer or floating point? new outlooks for low-bit quantization on large language models, 2023. URL https://arxiv.org/abs/2305.12356.

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving, 2024. URL https://arxiv.org/abs/2310.19102.

A Appendix

A.1 Weight Clustering and Optimization

A.1.1 Objective Derivation

Throughout this derivation, $\mathbf{X} = \mathbf{X}_{R,S}$ and $\mathbf{W} = \mathbf{W}_{R,S}$ denote the transformed activation and weight after $\mathbf{R}$ and $\mathbf{S}$ are fixed, following Section 3.2. We use $\widetilde{\mathbf{W}}$ for the clustered transformed weight and $\Delta\mathbf{W} = \mathbf{W} - \widetilde{\mathbf{W}}$ for its clustering error. With unquantized transformed activations $\mathbf{X} \in \mathbb{R}^{T \times d_{\text{in}}}$, the symmetric local output difference is

$$\mathcal{L}_{\text{sym}}(\widetilde{\mathbf{W}}) = \left\| \mathbf{X}\mathbf{W}^\top - \mathbf{X}\widetilde{\mathbf{W}}^\top \right\|_F^2 \tag{9}$$

$$= \sum_{i=1}^{d_{\text{out}}} \Delta\mathbf{W}_{i,:} \mathbf{H}_{\text{fp}} \Delta\mathbf{W}_{i,:}^\top, \tag{10}$$

where

$$\mathbf{H}_{\text{fp}} = \mathbf{X}^\top \mathbf{X}. \tag{11}$$

We call $\mathbf{H}_{\text{fp}}$ the Hessian. Its off-diagonal entries capture interactions between errors at different input dimensions.

During low-bit inference, however, the current module does not receive the clean activation $\mathbf{X}$. First, its input activation is quantized. Second, this activation is produced by upstream modules that already use clustered weights and quantized activations. As in Section 3.2, $\widetilde{\mathbf{X}}$ denotes the quantized transformed activation and $\Delta\mathbf{X} = \mathbf{X} - \widetilde{\mathbf{X}}$ its quantization error. The asymmetric output difference is

$$\mathcal{L}_{\text{asym}}(\widetilde{\mathbf{W}}) = \left\| \mathbf{X}\mathbf{W}^\top - \widetilde{\mathbf{X}}\widetilde{\mathbf{W}}^\top \right\|_F^2. \tag{12}$$

For output row i, the residual can be written directly as

$$\mathbf{X}\mathbf{W}_{i,:}^\top - \widetilde{\mathbf{X}}\widetilde{\mathbf{W}}_{i,:}^\top = \widetilde{\mathbf{X}}\Delta\mathbf{W}_{i,:}^\top + \Delta\mathbf{X}\mathbf{W}_{i,:}^\top. \tag{13}$$

Substituting this residual into the layer-wise objective gives

$$\mathcal{L}_{\text{asym}}(\widetilde{\mathbf{W}}) = \sum_{i=1}^{d_{\text{out}}} \left\| \widetilde{\mathbf{X}}\Delta\mathbf{W}_{i,:}^\top + \Delta\mathbf{X}\mathbf{W}_{i,:}^\top \right\|_2^2. \tag{14}$$

Using $\|\mathbf{a}+\mathbf{b}\|_2^2 = \mathbf{a}^\top\mathbf{a} + 2\mathbf{a}^\top\mathbf{b} + \mathbf{b}^\top\mathbf{b}$, we obtain

$$\begin{aligned} \mathcal{L}_{\text{asym}}(\widetilde{\mathbf{W}}) = \sum_{i=1}^{d_{\text{out}}} \Bigg[& \Delta\mathbf{W}_{i,:}\widetilde{\mathbf{X}}^\top\widetilde{\mathbf{X}}\Delta\mathbf{W}_{i,:}^\top \\ & + 2\Delta\mathbf{W}_{i,:}\widetilde{\mathbf{X}}^\top\Delta\mathbf{X}\mathbf{W}_{i,:}^\top \\ & + \mathbf{W}_{i,:}\Delta\mathbf{X}^\top\Delta\mathbf{X}\mathbf{W}_{i,:}^\top \Bigg]. \end{aligned} \tag{15}$$

We denote the Hessian computed from the inference-path activations by

$$\mathbf{H}_q = \widetilde{\mathbf{X}}^\top\widetilde{\mathbf{X}}. \tag{16}$$

Using this definition, the objective becomes

$$\begin{aligned} \mathcal{L}_{\text{asym}}(\widetilde{\mathbf{W}}) = \sum_{i=1}^{d_{\text{out}}} \Bigg[& \underbrace{\Delta\mathbf{W}_{i,:}\mathbf{H}_q\Delta\mathbf{W}_{i,:}^\top}_{(1)} \\ & + \underbrace{2\Delta\mathbf{W}_{i,:}\widetilde{\mathbf{X}}^\top\Delta\mathbf{X}\mathbf{W}_{i,:}^\top}_{(2)} \\ & + \underbrace{\mathbf{W}_{i,:}\Delta\mathbf{X}^\top\Delta\mathbf{X}\mathbf{W}_{i,:}^\top}_{(3)} \Bigg]. \end{aligned} \tag{17}$$

Term (1) measures the output error from weight clustering under the activations encountered during low-bit inference. With $\widetilde{\mathbf{X}}$ fixed, it guides both codebook and assignment updates. Term (3) measures the output error from activation quantization and guides composition selection. Term (2) couples the two errors and is not optimized separately. PhaseLUT uses Term (3) as a local current-layer objective rather than accounting for error propagation through later layers.

A.1.2 Optimization Algorithm

For output row i, let $\mathbf{c}_i \in \mathbb{R}^K$ be its codebook and let $\mathbf{A}_i \in \{0,1\}^{d_{\text{in}} \times K}$ be the one-hot assignment matrix defined by $\{a_{ij}\}_{j=1}^{d_{\text{in}}}$. Then $\widetilde{\mathbf{W}}_{i,:}^\top = \mathbf{A}_i\mathbf{c}_i$. With $\widetilde{\mathbf{X}}$ fixed, we alternate two updates to reduce Term (1). The codebook update solves a regularized row-wise objective for fixed assignments. The assignment update greedily processes input dimensions and propagates each residual. Because assignment is greedy and uses different damping from the codebook update, this is not exact block coordinate descent; the original asymmetric objective need not decrease at every iteration.

Codebook update. For fixed assignments, the row-wise problem is

$$\min_{\mathbf{c}_i} \left(\mathbf{W}_{i,:}^\top - \mathbf{A}_i\mathbf{c}_i\right)^\top \mathbf{H}_q \left(\mathbf{W}_{i,:}^\top - \mathbf{A}_i\mathbf{c}_i\right). \tag{18}$$

Setting its derivative to zero gives the normal equations

$$\mathbf{A}_i^\top \mathbf{H}_q \mathbf{A}_i \mathbf{c}_i = \mathbf{A}_i^\top \mathbf{H}_q \mathbf{W}_{i,:}^\top. \tag{19}$$

To stabilize the solve, we add $\lambda \bar{h}_q \mathbf{I}_K$ to the normal matrix, where $\bar{h}_q = \mathrm{tr}(\mathbf{H}_q)/d_{\mathrm{in}}$. This scale-normalized damping is also used to stabilize Hessian inversion in GPTQ (Frantar et al., 2023). The resulting update is

$$\mathbf{c}_i \leftarrow \left(\mathbf{A}_i^\top \mathbf{H}_q \mathbf{A}_i + \lambda \bar{h}_q \mathbf{I}_K\right)^{-1} \mathbf{A}_i^\top \mathbf{H}_q \mathbf{W}_{i,:}^\top. \tag{20}$$

When $\mathbf{H}_q$ is diagonal and $\lambda = 0$, $\mathbf{A}_i^\top \mathbf{H}_q \mathbf{A}_i$ is also diagonal, and each centroid becomes

$$c_{ik} = \frac{\sum_{j:a_{ij}=k} H_{q,jj} W_{ij}}{\sum_{j:a_{ij}=k} H_{q,jj}}. \tag{21}$$

This has the same algebraic form as the sensitivity-weighted centroid update in SqueezeLLM (Kim et al., 2024). The weighting signals differ: SqueezeLLM uses per-weight diagonal Fisher scores derived from the final loss, whereas we use $\mathrm{diag}(\mathbf{H}_q)$, the local curvature computed from the layer-output objective.

Algorithm 1 Hessian-weighted row-wise codebook update

Require: weights $\mathbf{W}$, Hessian $\mathbf{H}_q$, assignments $\{a_{ij}\}$, codebook size K, regularization λ
Ensure: row-wise codebooks $\{\mathbf{c}_i\}$

$\bar{h}_q \leftarrow \mathrm{tr}(\mathbf{H}_q)/d_{\mathrm{in}}$
for $i = 1, \ldots, d_{\mathrm{out}}$ **do**
  $\mathbf{A}_i \leftarrow \textsc{OneHot}(\{a_{ij}\}_{j=1}^{d_{\mathrm{in}}}, K)$
  $\mathbf{G}_i \leftarrow \mathbf{A}_i^\top \mathbf{H}_q \mathbf{A}_i + \lambda \bar{h}_q \mathbf{I}_K$
  $\mathbf{b}_i \leftarrow \mathbf{A}_i^\top \mathbf{H}_q \mathbf{W}_{i,:}^\top$
  $\mathbf{c}_i \leftarrow \mathrm{solve}(\mathbf{G}_i, \mathbf{b}_i)$
end for
return $\{\mathbf{c}_i\}$

Assignment update. For fixed codebooks, we assign the weights one input dimension at a time and compensate for each assignment over the dimensions that remain unassigned. We derive this step from the same second-order compensation principle used in GPTQ (Frantar et al., 2023), with each row projected onto its learned codebook.

The derivation requires an invertible Hessian. As discussed above, $\mathbf{H}_q$ may be singular or ill-conditioned because it is estimated from finite calibration data. We therefore apply the same stabilization idea used for the codebook solution:

$$\mathbf{H}_\rho = \mathbf{H}_q + \rho \bar{h}_q \mathbf{I}_{d_{\mathrm{in}}}, \qquad \bar{h}_q = \frac{\mathrm{tr}(\mathbf{H}_q)}{d_{\mathrm{in}}}. \tag{22}$$

Here, ρ controls the damping applied before Hessian inversion, whereas λ in Eq. 5 regularizes the codebook solution. In practice, we set $\rho = 10^{-3}$.

Consider output row i when dimension j is to be assigned, and let

$$\mathcal{F}_j = \{j, j+1, \ldots, d_{\mathrm{in}}\}$$

denote the dimensions that remain unassigned. We write $\mathbf{H}_{\rho,\mathcal{F}_j}$ for the corresponding principal submatrix and define

$$\mathbf{M}_j = \mathbf{H}_{\rho,\mathcal{F}_j}^{-1}. \tag{23}$$

Let Z_{ij} be the current value at dimension j, after compensation from earlier assignments. Assigning it to a candidate centroid c_{ik} introduces the candidate error

$$\epsilon_{ij}(k) = Z_{ij} - c_{ik}. \tag{24}$$

Equivalently, the change imposed at dimension j is $c_{ik} - Z_{ij} = -\epsilon_{ij}(k)$. Among all changes to the remaining dimensions that preserve this assignment, we seek the one that minimizes the quadratic error:

$$\begin{aligned} \min_{\boldsymbol{\delta}_{i,\mathcal{F}_j}} \quad & \boldsymbol{\delta}_{i,\mathcal{F}_j}^\top \mathbf{H}_{\rho,\mathcal{F}_j} \boldsymbol{\delta}_{i,\mathcal{F}_j} \\ \text{s.t.} \quad & \mathbf{e}_j^\top \boldsymbol{\delta}_{i,\mathcal{F}_j} = -\epsilon_{ij}(k), \end{aligned} \tag{25}$$

where the restricted vectors and matrices retain their original coordinate labels, and $\mathbf{e}_j$ selects coordinate j in $\mathcal{F}_j$.

Introducing a Lagrange multiplier ν gives

$$\mathcal{J} = \boldsymbol{\delta}_{i,\mathcal{F}_j}^\top \mathbf{H}_{\rho,\mathcal{F}_j} \boldsymbol{\delta}_{i,\mathcal{F}_j} + 2\nu \left(\mathbf{e}_j^\top \boldsymbol{\delta}_{i,\mathcal{F}_j} + \epsilon_{ij}(k) \right). \tag{26}$$

Setting its gradient with respect to $\boldsymbol{\delta}_{i,\mathcal{F}_j}$ to zero yields

$$\mathbf{H}_{\rho,\mathcal{F}_j} \boldsymbol{\delta}_{i,\mathcal{F}_j} + \nu \mathbf{e}_j = \mathbf{0}. \tag{27}$$

Using the constraint in Eq. 25, the minimizing change is

$$\boldsymbol{\delta}_{i,\mathcal{F}_j}^\star = -\frac{\epsilon_{ij}(k)}{[\mathbf{M}_j]_{jj}} \mathbf{M}_j \mathbf{e}_j. \tag{28}$$

The inverse Hessian gives the minimum-error adjustment to the remaining dimensions after the value at dimension j has been fixed.

Substituting Eq. 28 back into the quadratic objective gives the error incurred by choosing centroid c_{ik}:

$$\frac{\epsilon_{ij}(k)^2}{[\mathbf{M}_j]_{jj}}. \tag{29}$$

The denominator is independent of k at a fixed dimension j. Consequently, the best centroid is still the nearest one to the compensated value:

$$a_{ij} = \operatorname*{argmin}_{k \in \{1,\dots,K\}} |Z_{ij} - c_{ik}|, \qquad \widetilde{\mathbf{W}}_{ij} = c_{i,a_{ij}}. \tag{30}$$

For an unassigned dimension $j' > j$, Eq. 28 gives the corresponding compensation:

$$Z_{ij'} \leftarrow Z_{ij'} - \left(Z_{ij} - \widetilde{\mathbf{W}}_{ij} \right) \frac{[\mathbf{M}_j]_{jj'}}{[\mathbf{M}_j]_{jj}}. \tag{31}$$

After fixing dimension j, the inverse Hessian for the remaining dimensions is obtained by eliminating its row and column:

$$\mathbf{M}_{j+1} = \left(\mathbf{M}_j - \frac{[\mathbf{M}_j]_{:,j} [\mathbf{M}_j]_{j,:}}{[\mathbf{M}_j]_{jj}} \right)_{\mathcal{F}_{j+1}, \mathcal{F}_{j+1}}. \tag{32}$$

Applying this elimination recursively is equivalent to an upper-triangular Cholesky factorization of the initial inverse Hessian:

$$\mathbf{H}_\rho^{-1} = \mathbf{U}^\top \mathbf{U}, \qquad \mathbf{U} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}. \tag{33}$$

At step j, the relevant entries of the Cholesky factor satisfy

$$U_{jj} = \sqrt{[\mathbf{M}_j]_{jj}}, \qquad \frac{U_{jj'}}{U_{jj}} = \frac{[\mathbf{M}_j]_{jj'}}{[\mathbf{M}_j]_{jj}}, \quad j' > j. \tag{34}$$

The factorization therefore provides all conditional compensation coefficients without explicitly recomputing Eq. 32 after every assignment.

Define the scaled residual

$$r_{ij} = \frac{Z_{ij} - \widetilde{\mathbf{W}}_{ij}}{U_{jj}}, \tag{35}$$

For use across all output rows, let $\mathbf{r}_j = (r_{1j}, \dots, r_{d_{\text{out}}j})^\top$. Equation 31 becomes

$$Z_{ij'} \leftarrow Z_{ij'} - r_{ij} U_{jj'}, \qquad j' > j. \tag{36}$$

This gives the sequential assignment procedure summarized in Algorithm 2. Unlike GPTQ, which projects onto a fixed quantization grid, our feasible values are given by the learned row-wise codebooks. The resulting assignments are subsequently used to refine these codebooks in the next alternating iteration.

Algorithm 2 Hessian-aware row-wise assignment update

Require: weights $\mathbf{W}$, Hessian $\mathbf{H}_q$, codebooks $\{\mathbf{c}_i\}$, damping ρ
Ensure: assignments $\{a_{ij}\}$

$\bar{h}_q \leftarrow \mathrm{tr}(\mathbf{H}_q)/d_{\mathrm{in}}$
$\mathbf{H}_\rho \leftarrow \mathbf{H}_q + \rho \bar{h}_q \mathbf{I}_{d_{\mathrm{in}}}$
$\mathbf{U} \leftarrow \mathrm{chol}(\mathbf{H}_\rho^{-1})$ ▷ upper triangular
$\mathbf{Z} \leftarrow \mathbf{W}$
for $j = 1, \ldots, d_{\mathrm{in}}$ **do**
  for $i = 1, \ldots, d_{\mathrm{out}}$ **do**
    $a_{ij} \leftarrow \arg\min_k |Z_{ij} - c_{ik}|$
    $\widetilde{\mathbf{W}}_{ij} \leftarrow c_{i,a_{ij}}$
    $r_{ij} \leftarrow (Z_{ij} - \widetilde{\mathbf{W}}_{ij})/U_{jj}$
  end for
  $r_{ij} = [r_{1j}, r_{2j}, \ldots, r_{d_{\mathrm{out}}j}]^\top \in \mathbb{R}^{d_{\mathrm{out}} \times 1}$
  $\mathbf{Z}_{:,j+1:d_{\mathrm{in}}} \leftarrow \mathbf{Z}_{:,j+1:d_{\mathrm{in}}} - \mathbf{r}_j \mathbf{U}_{j,j+1:d_{\mathrm{in}}}$
end for
return $\{a_{ij}\}$

Iterative refinement. Starting from the row-wise K-means codebooks retained from transformation optimization, we alternate the assignment and codebook updates for at most N_{iter} iterations and stop early when the codebooks stabilize.

Algorithm 3 Iterative Hessian-weighted clustering

Require: weights $\mathbf{W}$, Hessian $\mathbf{H}_q$, initial codebooks $\{\mathbf{c}_i^{(0)}\}$, codebook size K, maximum iterations N_{iter}, damping ratios λ, ρ, tolerance τ
Ensure: codebooks $\{\mathbf{c}_i\}$ and assignments $\{a_{ij}\}$

$\{\mathbf{c}_i\} \leftarrow \{\mathbf{c}_i^{(0)}\}$
for $\ell = 1, \ldots, N_{\mathrm{iter}}$ **do**
  $\{\mathbf{c}_i^{\mathrm{prev}}\} \leftarrow \{\mathbf{c}_i\}$
  $\{a_{ij}\} \leftarrow \textsc{AssignmentUpdate}(\mathbf{W}, \mathbf{H}_q, \{\mathbf{c}_i\}, \rho)$
  $\{\mathbf{c}_i\} \leftarrow \textsc{CodebookUpdate}(\mathbf{W}, \mathbf{H}_q, \{a_{ij}\}, K, \lambda)$
  if $\max_{i,k} |c_{ik} - c_{ik}^{\mathrm{prev}}| < \tau$ **then**
    break
  end if
end for
return $\{\mathbf{c}_i\}, \{a_{ij}\}$

A.1.3 Interaction-Term Transfer

Term (2) in the asymmetric objective is

$$\mathcal{L}_{\mathrm{int}} = 2 \sum_{i=1}^{d_{\mathrm{out}}} \Delta \mathbf{W}_{i,:} \widetilde{\mathbf{X}}^\top \Delta \mathbf{X} \mathbf{W}_{i,:}^\top. \tag{37}$$

Term (2) depends on the codebooks and assignments through $\Delta \mathbf{W}$ and on activation quantization through $\widetilde{\mathbf{X}}^\top \Delta \mathbf{X}$. We compare this activation-dependent statistic between calibration and held-out data using Qwen3-4B (Yang et al., 2025).

We use the activation-quantization error and its weight-projected cross-statistic:

$$\Delta \mathbf{X} = \mathbf{X} - \widetilde{\mathbf{X}}, \qquad \mathbf{G} = \frac{1}{T} \mathbf{W} \Delta \mathbf{X}^\top \widetilde{\mathbf{X}}. \tag{38}$$

Table 8: Calibration–holdout cosine similarity of the vectorized statistics on Qwen3-4B. Higher values indicate greater similarity between the two sets.

K	$\mathbf{H}_q \uparrow$	$\Delta\mathbf{X}^\top\Delta\mathbf{X} \uparrow$	$\Delta\mathbf{X}^\top\widetilde{\mathbf{X}} \uparrow$	$\mathbf{G} \uparrow$
4	0.914	0.255	0.264	0.317
8	0.912	0.296	0.124	0.172
16	0.910	0.436	0.054	0.072

Table 8 shows substantial calibration-to-held-out changes in $\Delta\mathbf{X}^\top\widetilde{\mathbf{X}}$, the transpose of $\widetilde{\mathbf{X}}^\top\Delta\mathbf{X}$ used above. The weight-projected $\mathbf{G}$ also changes markedly.

A.1.4 Optimization Schedule and Interleaving Analysis

Our pipeline follows a single pass. We first use fixed E2M1 activation quantization to optimize rotation and scaling. We then cluster the weights module by module in execution order. Immediately before clustering each module, we collect $\mathbf{H}_q$ once from activations propagated through all preceding clustered modules. We retain the row-wise K-means codebooks from transformation optimization and keep this $\mathbf{H}_q$ fixed for five assignment and codebook refinement rounds. After the transformations and clustered weights are fixed, we calibrate PhaseLUT and do not revisit the earlier stages.

Because the earlier stages use fixed E2M1 rather than the learned mixed composition, we measure how much their outputs would change under interleaving. Pass 1 follows the standard schedule above. In pass 2, we repeat rotation-and-scaling optimization and weight clustering using the mixed composition learned in pass 1. We compare corresponding intermediate outputs on Qwen3-0.6B (Yang et al., 2025). The main experiments use FP4, and we also include FP8 to check whether the same stability pattern appears at another floating-point precision.

Table 9: Stability of intermediate outputs across the two passes on Qwen3-0.6B. Each entry compares pass 1 with pass 2. For a vectorized intermediate output $\mathbf{y}$, relative L_2 is $\|\mathbf{y}^{(2)} - \mathbf{y}^{(1)}\|_2/\|\mathbf{y}^{(1)}\|_2$. Lower relative L_2 and higher cosine or agreement indicate greater similarity.

		Pass-1/pass-2 similarity	
Intermediate output	**Metric**	**FP8**	**FP4**
Rotation, scaling, and weight clustering			
Rotation	rel. $L_2 \downarrow$	0.00830	0.00890
	cosine $\uparrow$	0.99997	0.99996
Scaling	rel. $L_2 \downarrow$	0.00021	0.00025
	cosine $\uparrow$	1.00000	1.00000
Codebooks	rel. $L_2 \downarrow$	0.05800	0.06050
Clustered weights	rel. $L_2 \downarrow$	0.17400	0.18700
	cosine $\uparrow$	0.98480	0.98250
Mixed-composition calibration			
Activation LUT	cell agreement $\uparrow$	98.9%	98.2%
	identical LUTs $\uparrow$	52/56	52/56

We use Qwen3-0.6B to keep the two-pass comparison tractable. Table 9 shows that rotation and scaling change by at most 0.89% and 0.025% in relative L_2, respectively. The clustered weights retain cosine similarity above 0.98, although their relative L_2 difference reaches 18.7%. At least 98.2% of PhaseLUT cells agree between the two passes, and 52 of 56 LUTs are identical. These results support the single-pass schedule used in the main experiments.

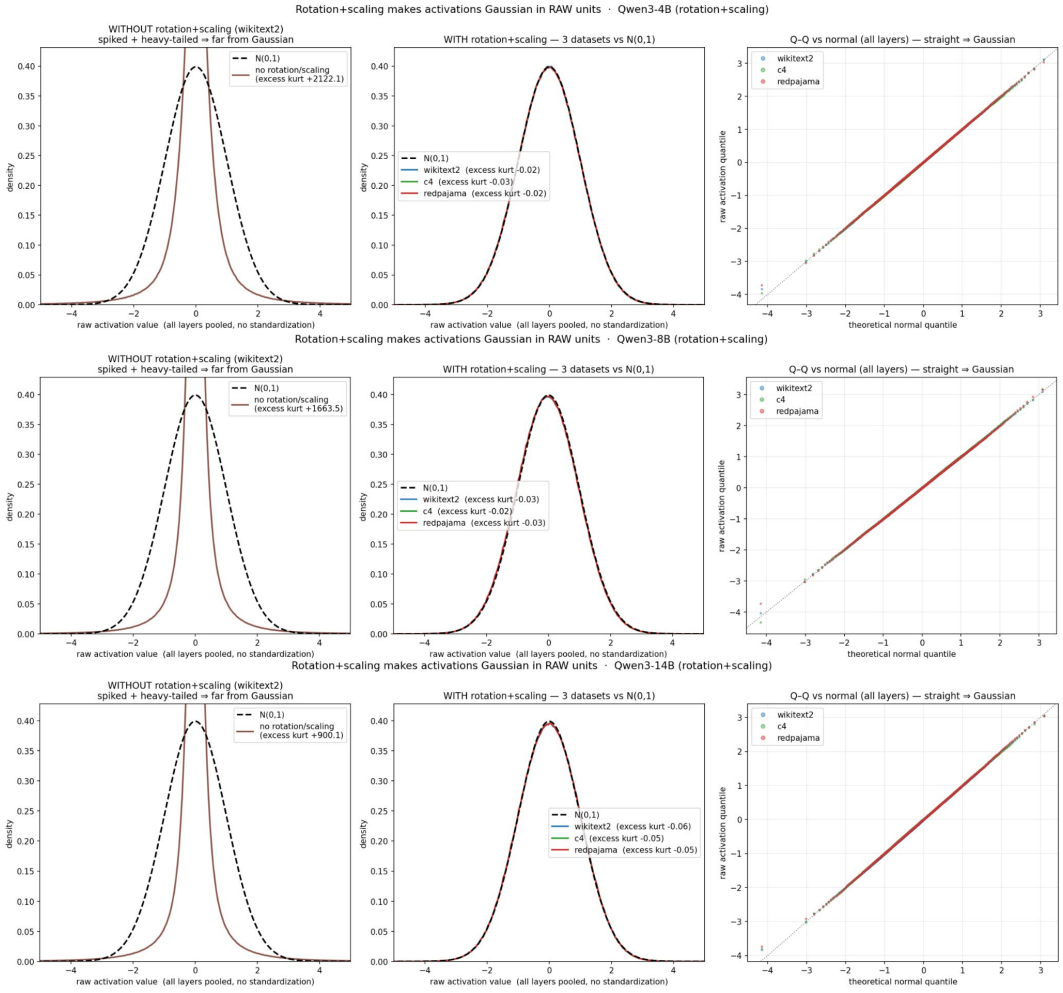

Figure 4: Activation distributions before and after rotation, scaling, and RMSNorm. Rows correspond to Qwen3-4B, Qwen3-8B, and Qwen3-14B. The rightmost column compares the post-transformation activations with a standard normal distribution using Q–Q plots.

A.2 Mixed-Composition Activation Quantization

A.2.1 Amax Approximation

PhaseLUT needs a compact token statistic that predicts which composition minimizes output error. We use the token maximum magnitude, motivated by the following empirical approximation for the activation row $\mathbf{X}_{t,:} \in \mathbb{R}^{1\times d_{\text{in}}}$ after RMSNorm, rotation, and scaling:

$$X_{tj} \overset{\text{approx.}}{\sim} \mathcal{N}(0,1), \qquad j = 1, \ldots, d_{\text{in}}. \tag{39}$$

This is consistent with the observation in SpinQuant (Liu et al., 2025) that random rotations reduce activation kurtosis to approximately three across LLM layers, indicating a more Gaussian-shaped distribution. Figure 4 shows this behavior across the three model sizes.

Even under this approximation, token rows can have different minima and maxima (Figure 5). We summarize each row using

$$\begin{aligned} \mathbf{s}_t &= \left(\min(\mathbf{X}_{t,:}), \max(\mathbf{X}_{t,:}), \mu_t, \sigma_t^2\right) \\ &\approx (-a_t, a_t, 0, 1)\,, \qquad a_t = \operatorname{amax}(\mathbf{X}_{t,:}) = \max|\mathbf{X}_{t,:}|. \end{aligned} \tag{40}$$

The standard-normal approximation gives $\mu_t \approx 0$ and $\sigma_t^2 \approx 1$. Its symmetry, together with our empirical observation after rotation and scaling, gives $-\min(\mathbf{X}_{t,:}) \approx \max(\mathbf{X}_{t,:})$. Figure 6 shows

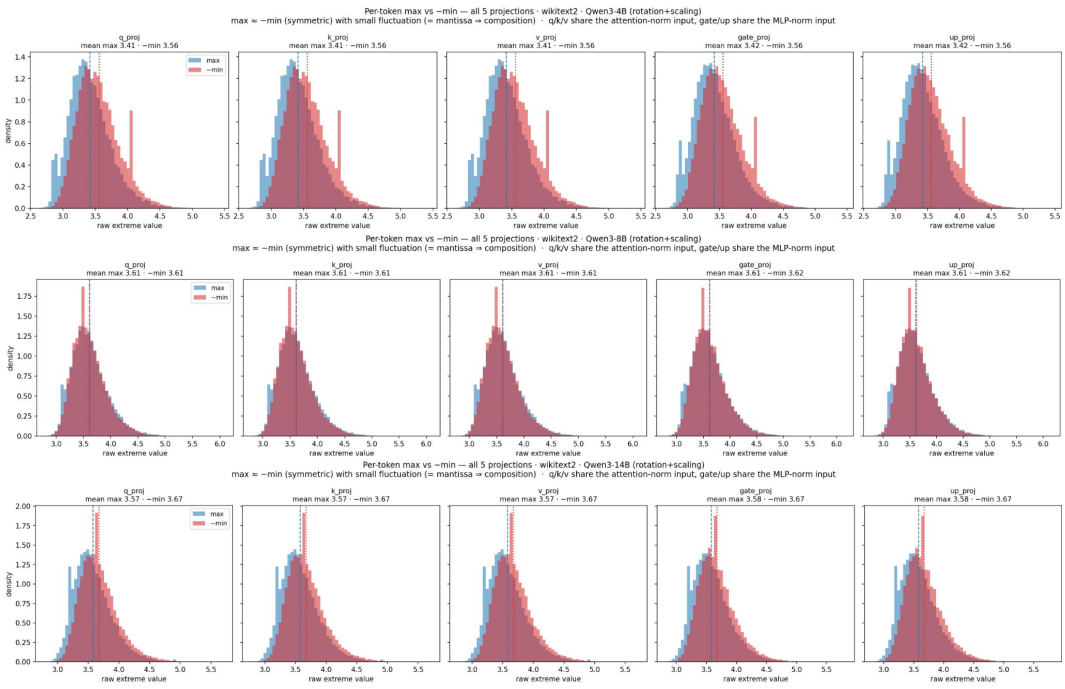

Figure 5: Tokenwise minimum and maximum values after rotation, scaling, and RMSNorm. Rows correspond to Qwen3-4B, Qwen3-8B, and Qwen3-14B. Columns correspond to `q_proj`, `k_proj`, `v_proj`, `gate_proj`, and `up_proj`.

the corresponding magnitude differences across datasets and layers. Figure 7 further shows that the tokenwise standard deviation remains close to one despite variation in the sampled extrema.

We next consider how to retain the information in a_t under per-token scaling. Because the scale is restricted to a power of two, scaling changes only the exponent of a_t and leaves its normalized mantissa unchanged. The same PhaseLUT index can therefore be extracted before or after scaling, independently of the selected composition. We use the first p fractional mantissa bits as the index:

$$\varphi_t = \left\lfloor 2^p \big(\mathrm{mantissa}(a_t) - 1\big) \right\rfloor, \qquad \varphi_t \in \{0, \dots, 2^p - 1\}, \tag{41}$$

where $\mathrm{mantissa}(a_t) \in [1, 2)$. This partitions $[1, 2)$ into 2^p uniform bins.

A.2.2 LUT Calibration Algorithm

Let $\mathcal{X}$ denote the transformed activations received by a layer before its current activation quantization. These activations already contain upstream quantization effects. Let $\mathcal{W}$ be the projections that share this activation and $\mathcal{C}_b$ the candidate compositions. For token $\mathbf{X}_{t,:}$ and composition (e, m), let $\Delta\mathbf{X}_t^{(e,m)} = \mathbf{X}_{t,:} - Q_{e,m}(\mathbf{X}_{t,:})$. The corresponding group-averaged objective from Eq. 6 is

$$\mathcal{L}_{\mathrm{comp}}(\mathbf{X}_{t,:}, e, m; \mathcal{W}) = \mathbb{E}_{\mathbf{W} \in \mathcal{W}} \left\| \Delta\mathbf{X}_t^{(e,m)} \mathbf{W}^\top \right\|_2^2. \tag{42}$$

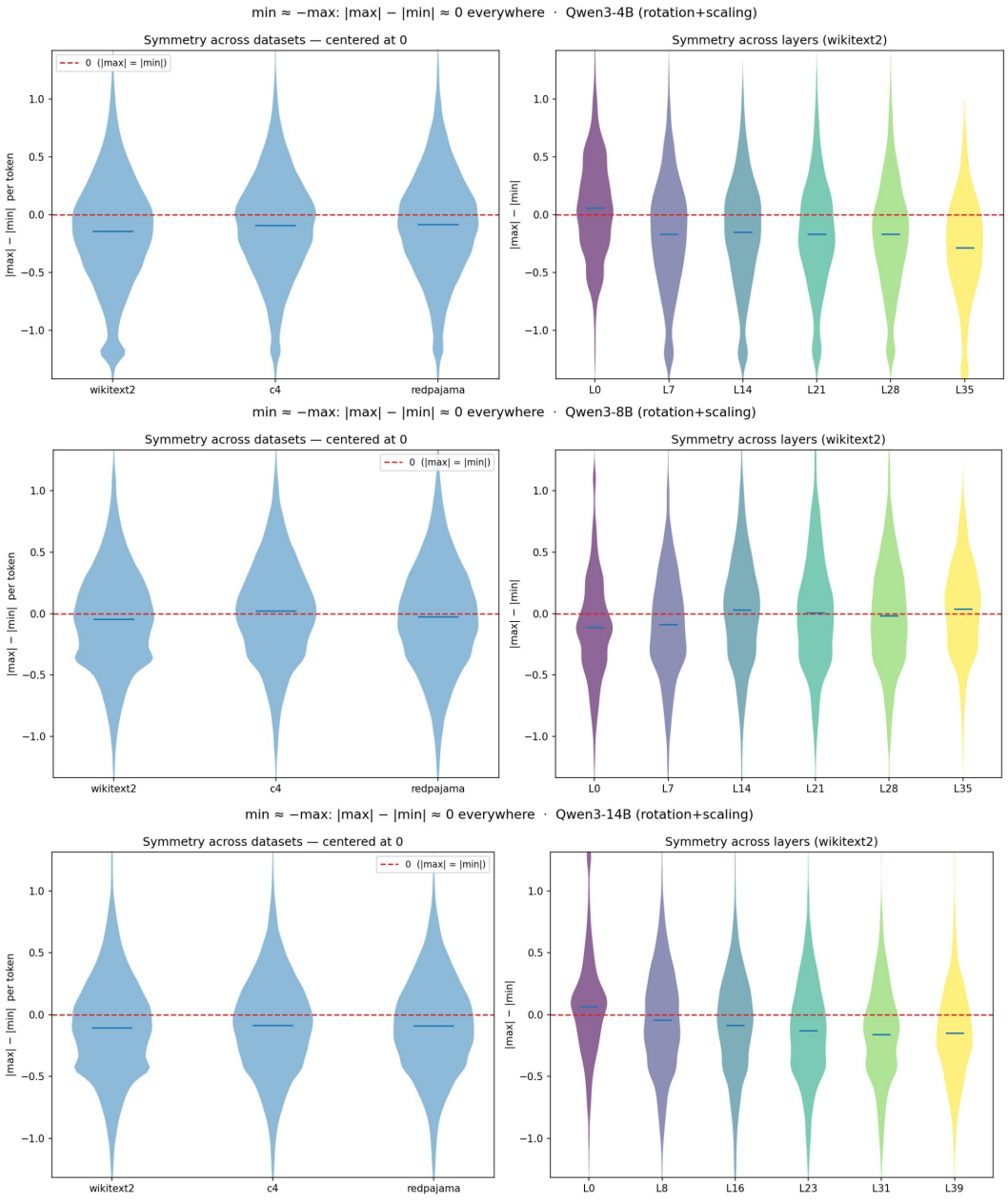

Figure 6: Distribution of $|\max(\mathbf{X}_{t,:})| - |\min(\mathbf{X}_{t,:})|$ after rotation, scaling, and RMSNorm. The left panel compares datasets. The right panel shows the distribution across layers on WikiText-2.

Algorithm 4 PhaseLUT calibration for one shared activation

Require: activation collection $\mathcal{X}$, weight group $\mathcal{W}$, candidate compositions $\mathcal{C}_b$, prefix length p
Ensure: PhaseLUT

for each token activation $\mathbf{X}_{t,:} \in \mathcal{X}$ **do**
  $\varphi_t \leftarrow \textsc{MantissaPrefix}(\mathrm{amax}(\mathbf{X}_{t,:}), p)$
  for each composition $(e, m) \in \mathcal{C}_b$ **do**
    evaluate $\mathcal{L}_{\mathrm{comp}}(\mathbf{X}_{t,:}, e, m; \mathcal{W})$
  end for
end for
for each LUT index $\varphi \in \{0, \ldots, 2^p - 1\}$ **do**
  $\mathrm{PhaseLUT}[\varphi] \leftarrow \arg\min_{(e,m)\in\mathcal{C}_b} \sum_{\mathbf{X}_{t,:}\in\mathcal{X}:\varphi_t=\varphi} \mathcal{L}_{\mathrm{comp}}(\mathbf{X}_{t,:}, e, m; \mathcal{W})$
end for
return PhaseLUT

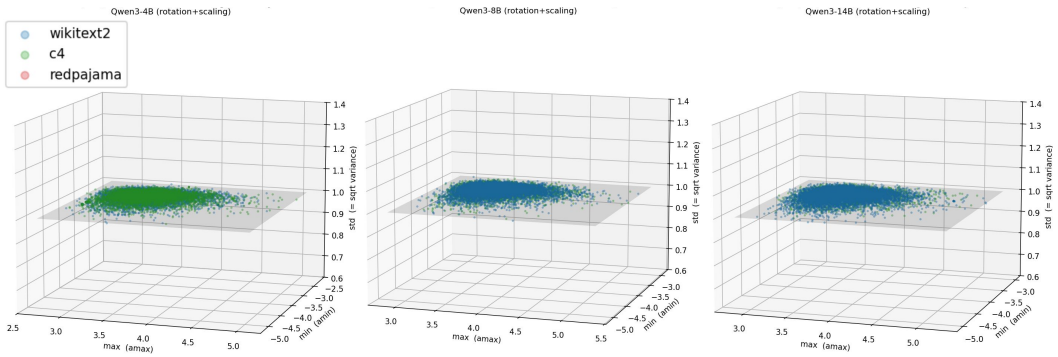

Figure 7: Token-level samples after rotation, scaling, and RMSNorm for Qwen3-4B, Qwen3-8B, and Qwen3-14B. Across different calibration samples, the token-wise standard deviation remains close to one.

Table 10: Relative loss reduction and oracle recovery (%) under FP4. PhaseLUT uses 16 Amax-phase entries.

Model	Group	Oracle gain ↑	LUT gain ↑	Recovery ↑
Qwen3-4B	QKV	0.40	0.36	91.21
Qwen3-4B	Gate/Up	1.80	0.64	35.72
Qwen3-8B	QKV	0.74	0.69	93.32
Qwen3-8B	Gate/Up	3.64	1.86	50.93
Qwen3-14B	QKV	0.47	0.44	93.22
Qwen3-14B	Gate/Up	2.02	1.00	49.56

A.2.3 LUT Information Sufficiency

PhaseLUT uses the phase of a_t to approximate per-token composition selection. We evaluate how much of the per-token oracle improvement it recovers and whether other activation statistics provide additional information. All LUTs are fitted on the calibration set, while the losses reported below are evaluated on held-out activations.

Recovery from the per-token oracle. For each projection group, let L_{best} be the holdout loss of the best single composition shared by all tokens. Let L_{oracle} be the loss obtained by selecting the best composition independently for each token. The per-token oracle gives the largest possible gain among the candidate compositions but requires an online search. For FP4, E2M1 is the best single composition in all evaluated settings. Given the holdout loss L of a composition-selection method, we report

$$\text{Gain}(L) = \frac{L_{\text{best}} - L}{L_{\text{best}}} \times 100\%, \qquad \text{Recovery}(L) = \frac{L_{\text{best}} - L}{L_{\text{best}} - L_{\text{oracle}}} \times 100\%. \tag{43}$$

The gain measures the loss reduction relative to the best single composition. Recovery measures how much of the available oracle improvement is obtained.

Table 10 shows different behaviors for the two projection groups. For QKV, the best single composition is already close to the per-token oracle, leaving only 0.40–0.74% improvement. PhaseLUT recovers more than 91% of this gap. Gate/Up has a larger oracle gain of 1.80–3.64%. Although PhaseLUT recovers a smaller fraction of this gap, it produces larger absolute loss reductions of 0.64–1.86%. Mixed composition therefore provides more practical benefit for Gate/Up, while only a small correction is available for QKV.

Information carried by Amax phase. We next test whether the improvement comes from the Amax phase or merely from introducing a calibrated lookup table. Table 11 compares the 16-entry Amax-phase key with several controls. The shape key discretizes $\log_2(a_t / \text{RMS}(\mathbf{X}_{t,:}))$. The

Table 11: Oracle recovery (%) using different LUT keys under FP4. Joint-$P\times S$ uses P Amax-phase entries and S shape entries.

Model	Group	Amax Phase-16 ↑	Shape-16 ↑	Joint-4×4 ↑	Joint-16×4 ↑	Random-16 ↑
Qwen3-4B	QKV	91.21	0.00	0.00	91.21	0.00
Qwen3-4B	Gate/Up	35.72	1.27	3.74	35.69	0.00
Qwen3-8B	QKV	93.32	0.30	0.00	93.32	0.00
Qwen3-8B	Gate/Up	50.93	2.07	2.01	50.93	0.00
Qwen3-14B	QKV	93.22	0.02	0.00	93.22	0.00
Qwen3-14B	Gate/Up	49.56	2.50	2.13	49.56	0.00

joint keys combine the Amax phase and shape using the indicated number of entries along each dimension. Random-16 uses an activation-independent random partition with the same number of entries as Amax Phase-16. Each entry is calibrated using the same objective, and all results report the recovery defined in Eq. 43.

The shape key alone recovers at most 2.50% of the oracle improvement. Joint-4×4 has the same total number of entries as Amax Phase-16 but performs substantially worse, showing that distributing the available entries across phase and shape loses important phase resolution. Increasing the joint table to 16×4 restores this resolution but provides no improvement over Amax Phase-16. Random-16 recovers none of the oracle gap, showing that table size alone does not explain the gain. Among the activation summaries considered here, the Amax phase captures the useful information without requiring an additional token statistic.

A.2.4 Mantissa Bit Tradeoff

We sweep the mantissa-prefix length p from 1 to 8, corresponding to PhaseLUTs with 2^p entries. Figure 8 reports the resulting FP4 perplexity. To make this sweep faster, we evaluate perplexity on a fixed subset of each evaluation set, using the same subset for every value of p and the E2M1 baseline. The effect is not monotonic across models and datasets. Four bits gives the most consistent result: it achieves the lowest PPL on WikiText2 for all three models and on C4 for Qwen3-8B. Five bits is best on C4 for Qwen3-4B and Qwen3-14B, although the differences are small. Larger tables provide no consistent benefit and can degrade WikiText2. We therefore use $p = 4$, which limits PhaseLUT to 16 entries.

A.2.5 Learned LUT Distribution

Figure 9 shows the composition distribution learned by the 4-bit PhaseLUT. Most layers prefer E2M1, while several early layers favor compositions with more mantissa bits, prioritizing precision over dynamic range. For quantization after scaling, E1M2 and E0M3 produce the same quantization results. Although they represent different values, both use uniformly spaced grids, so they behave identically after scaling. In our implementation, we keep them as separate candidates to avoid potential numerical errors from floating-point conversion. The candidate list therefore contains all four compositions in the order E3M0, E2M1, E1M2, and E0M3. When candidates tie, the code returns the first one in this list. This is why E0M3 does not appear in the distribution: ties between E1M2 and E0M3 are assigned to E1M2.

A.3 Kernel Design

We retain the clustered weight layout of SqueezeLLM (Kim et al., 2024), where each weight is represented by a 4-bit index into a 16-entry FP16 codebook for its output channel. Our kernel adds a mixed-composition FP4 activation path and integrates composition selection, activation encoding, and clustered linear computation in one pipeline.

A.3.1 Mixed-Composition Activation Encoding

For each token t, the kernel computes the maximum magnitude $a_t = \max|\mathbf{X}_{t,:}|$ and extracts its mantissa-prefix index φ_t. PhaseLUT selects the composition $f_t = (e_t, m_t)$. The kernel then com-

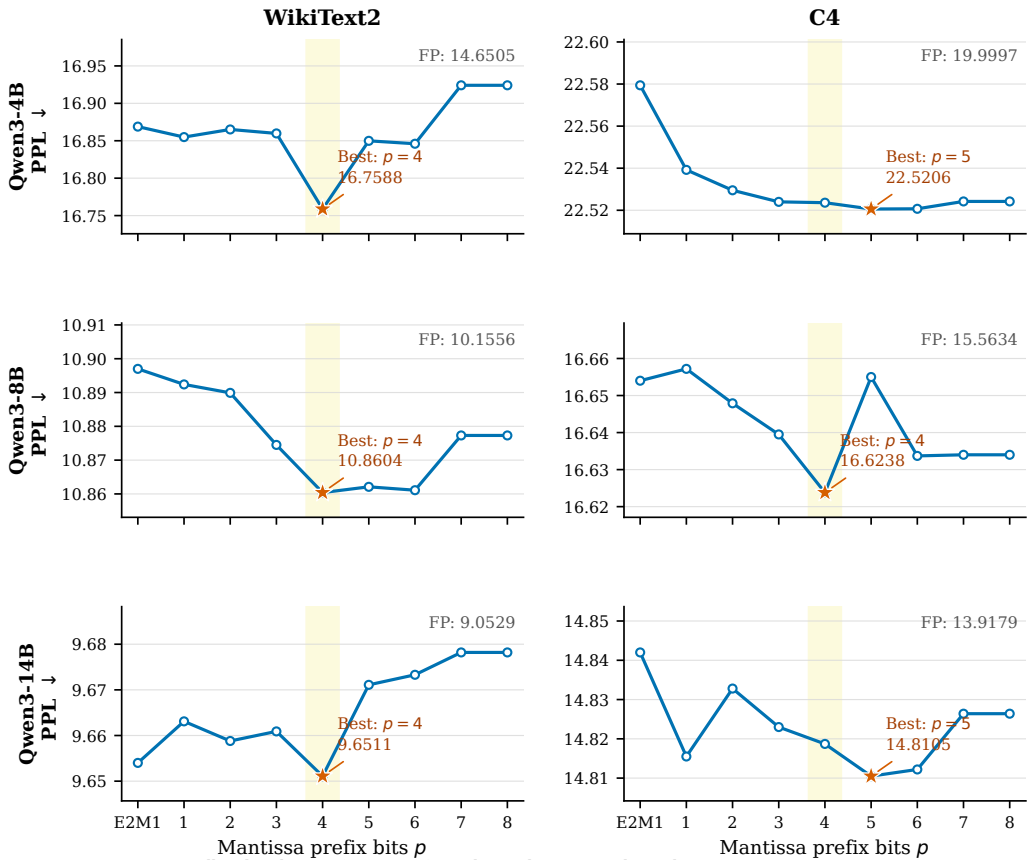

Figure 8: FP4 perplexity across mantissa-prefix lengths. E2M1 denotes fixed E2M1 without a LUT. Results are computed on a fixed subset of each evaluation set for faster comparison, with the same subset used for all settings. Each panel uses an independent raw-PPL vertical range, and lower is better. The shaded region marks the $p = 4$ setting used in our experiments.

putes the scale s_t and encodes the activation row as a packed code vector $\mathbf{q}_t$:

$$a_t \longrightarrow \varphi_t \longrightarrow f_t = \mathrm{PhaseLUT}[\varphi_t] \longrightarrow s_t \longrightarrow \mathbf{q}_t. \tag{44}$$

The composition must be selected before the scale because E3M0, E2M1, E1M2, and E0M3 have different maximum magnitudes: 16, 6, 3.5, and 0.875, respectively.

We use a per-token PoT scale $s_t = 2^{k_t}$, where

$$k_t = \mathrm{round}\left(\log_2 \frac{a_t}{v_{\max}(f_t)}\right). \tag{45}$$

As shown in Algorithm 6, the kernel multiplies $\mathbf{X}_{t,:}$ by $s_t^{-1} = 2^{-k_t}$, rounds and clips each value to the selected 4-bit format, and packs eight codes into each INT32 word. Algorithm 5 computes k_t from the floating-point exponent and one mantissa comparison, avoiding a general logarithm or scale search. Both PhaseLUT indexing and scale selection reuse a_t, so no additional scan of the activation row is needed.

A.3.2 Clustered Linear Computation

The linear kernel consumes the packed activation codes, 4-bit weight indices, one FP16 codebook for each output row, and the selected composition and scale exponent for each token. It decodes each activation value, retrieves the corresponding weight centroid, and accumulates their product using either half2 FMA or FP16 MMA. The PoT activation scale is restored once after accumulation.

All compositions use the same 4-bit code width and packing order. Composition selection therefore changes only the activation decoder and does not alter the memory-access pattern or require a

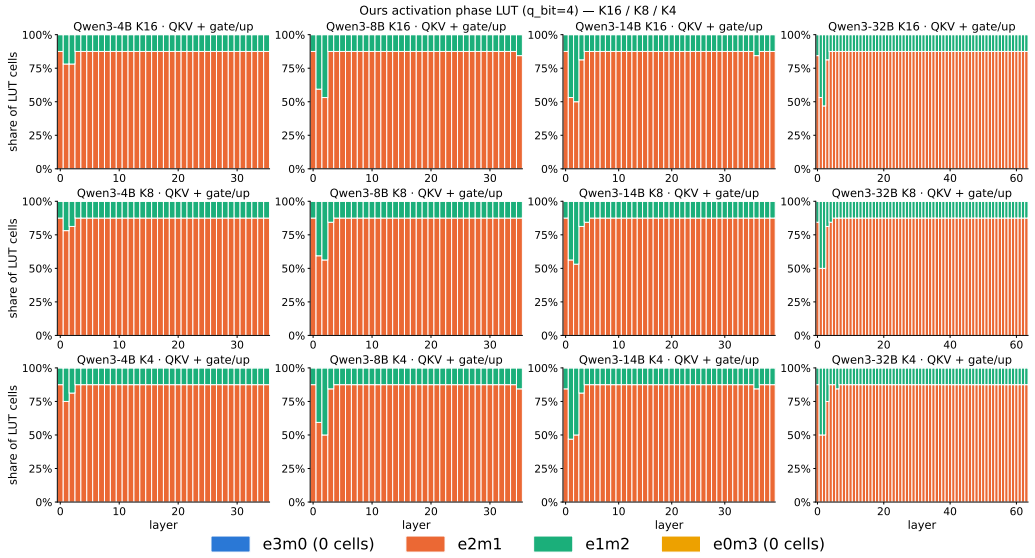

Figure 9: Layer-wise FP4 composition distribution learned by the 4-bit PhaseLUT.

Algorithm 5 Computing the FP4 PoT scale exponent from an FP16 maximum

Require: positive normal FP16 absmax a_t, selected composition f_t
Ensure: $k_t = \mathrm{round}(\log_2(a_t / v_{\max}(f_t)))$
FP16 constants: $p_m = 10$, $p_e = 5$, $\beta = 2^{p_e-1} - 1$

$(E_f, d_f, T_f) \leftarrow \mathrm{ScaleParams}[f_t]$
$B \leftarrow \textsc{BitCastUInt16}(a_t)$
$E_a \leftarrow ((B \gg p_m) \,\&\, (2^{p_e} - 1)) - \beta$
$M_a \leftarrow B \,\&\, (2^{p_m} - 1)$
$k_t \leftarrow E_a - E_f$
if $d_f = +1$ **then**
  if $M_a \geq T_f$ **then**
    $k_t \leftarrow k_t + 1$
  end if
else
  if $M_a < T_f$ **then**
    $k_t \leftarrow k_t - 1$
  end if
end if
return k_t

Algorithm 6 PhaseLUT-guided FP4 activation encoding

Require: activation row $\mathbf{x}_t = (x_{t1}, \ldots, x_{td})$, PhaseLUT, prefix length p
Ensure: packed FP4 codes $\mathbf{q}_t$, composition f_t, scale exponent k_t

$a_t \leftarrow \max_j |x_{tj}|$
$\varphi_t \leftarrow \textsc{MantissaPrefix}(a_t, p)$
$f_t \leftarrow \mathrm{PhaseLUT}[\varphi_t]$
$k_t \leftarrow \textsc{FP4PoTExponent}(a_t, f_t)$
for $j = 1, \ldots, d$ in parallel **do**
  $(\mathrm{sign}_j, E_j, M_j) \leftarrow \textsc{Unpack}(x_{tj})$
  $E'_j \leftarrow E_j - k_t$
  $(\widehat{E}_j, \widehat{M}_j) \leftarrow \textsc{RoundClipFP4}(E'_j, M_j, f_t)$
  $c_j \leftarrow \textsc{PackFP4}(\mathrm{sign}_j, \widehat{E}_j, \widehat{M}_j, f_t)$
end for
$\mathbf{q}_t \leftarrow \textsc{Pack8ToINT32}(c_1, \ldots, c_d)$
return $\mathbf{q}_t, f_t, k_t$

separate kernel launch. PhaseLUT adds one lookup after the token-maximum reduction, while the remaining clustered linear computation is shared across compositions.

A.3.3 Additional Kernel Analysis

Algorithm 5 introduces conditional branches, which could cause branch divergence and reduce GPU utilization. However, in our implementation, these branches do not cause branch divergence. Each block encodes one token row: a block-wide reduction gives all threads the same a_t, and they use the same PhaseLUT-selected f_t. Every branch condition depends only on these shared values and fixed format parameters, so all 32 threads in a warp take the same branch.

Table 12: End-to-end quantization cost of our pipeline for **non-reasoning models** on 2× NVIDIA H200 GPUs. Total wall-clock time is divided into rotation and scaling processing (P), clustering (C), and activation LUT calibration (A). Peak memory is the maximum aggregate GPU memory footprint of the two-device pipeline. It is dominated by the processing stage and is therefore largely insensitive to K.

Models	K	Total (hr) ↓	P (hr) ↓	C (hr) ↓	A (hr) ↓	Peak Mem. (GB) ↓
Qwen3-4B	4	0.41	0.21	0.16	0.04	29.7
	8	0.48	0.23	0.16	0.09	29.7
	16	0.48	0.20	0.21	0.07	29.7
Qwen3-8B	4	0.62	0.37	0.19	0.06	39.3
	8	0.74	0.41	0.24	0.09	39.3
	16	0.70	0.40	0.25	0.05	39.7
Qwen3-14B	4	1.01	0.59	0.34	0.08	59.4
	8	1.16	0.72	0.37	0.07	59.4
	16	1.27	0.72	0.47	0.08	59.4
Qwen3-32B	4	1.90	1.00	0.79	0.11	101.1
	8	2.00	1.25	0.65	0.10	101.1
	16	2.40	1.29	0.98	0.13	101.1

A.4 Reproducibility Details

A.4.1 Calibration Dataset

Sources. For general-purpose models, we calibrate on a mixture of the same three public datasets used by ParoQuant (Liang et al., 2026): WikiText-2 (Merity et al., 2016),[1] C4 (Raffel et al., 2023),[2] and RedPajama (Weber et al., 2024).[3] For reasoning models, we additionally add samples from the math and code subsets of OpenThoughts3 (Guha et al., 2025).[4]

Sampling. We remove empty documents from each data pool and shuffle once with seed 42. Each sample is one document truncated to at most $L = 512$ tokens and padded within its batch. For general-purpose models, we use 512 sequences from the three-source mixture. For reasoning models, we use 1,024 sequences to preserve both general and reasoning capabilities, combining the exact same 512 general-purpose calibration sequences with 512 additional reasoning sequences. Each model uses the same calibration set for rotation and scaling training, weight clustering, and activation LUT calibration. For each model, all baselines requiring calibration data use exactly the same sequences as CoBIT.

A.4.2 Optimization Efficiency

We report the end-to-end quantization cost for non-reasoning models in Table 12 and for reasoning models in Table 13. We profile the reasoning models separately because they use a distinct set of model checkpoints and the reasoning-augmented calibration set described in Section A.4.1. Specifically, the non-reasoning profile uses the 512-sequence general-purpose calibration set, whereas the reasoning profile uses 1,024 sequences comprising the same 512 general-purpose sequences and 512 additional reasoning sequences. Table 14 shows that CoBIT takes 25–76 minutes for these models, less than CodeQuant and ParoQuant at each reported bit width, while SqueezeLLM and QuaRot take less time.

[1] https://huggingface.co/datasets/Salesforce/wikitext/viewer/wikitext-2-raw-v1

[2] https://huggingface.co/datasets/allenai/c4

[3] https://huggingface.co/datasets/liang2kl/RedPajama-Data-1T-Sample-Backup

[4] https://huggingface.co/datasets/open-thoughts/OpenThoughts3-1.2M

Table 13: End-to-end quantization cost of our pipeline for **reasoning models** on 2× NVIDIA H200 GPUs, profiled using the reasoning-augmented calibration set described in Section A.4.1. Total wall-clock time is divided into rotation and scaling processing (P), clustering (C), and activation LUT calibration (A). Peak memory is the maximum aggregate GPU memory footprint of the two-device pipeline. It is dominated by the processing stage and is therefore largely insensitive to K.

Models	K	Total (hr) ↓	P (hr) ↓	C (hr) ↓	A (hr) ↓	Peak Mem. (GB) ↓
Qwen3-4B-Thinking-2507	4	0.30	0.14	0.11	0.05	40.3
	8	0.32	0.11	0.11	0.10	40.3
	16	0.38	0.19	0.14	0.05	40.3
Llama-3.1-Nemotron-Nano-4B	4	0.50	0.23	0.17	0.09	40.0
	8	0.42	0.18	0.15	0.09	40.0
	16	0.48	0.26	0.17	0.05	40.0
DeepSeek-R1-Distill-Llama-8B	4	0.53	0.32	0.15	0.06	59.8
	8	0.57	0.26	0.16	0.15	59.8
	16	0.62	0.32	0.19	0.11	59.8

Table 14: Observed offline quantization time (minutes) across Qwen3 models. Bits 2/3/4 correspond to $K = 4/8/16$ for clustering-based methods and W2/W3/W4 for ParoQuant. QuaRot entries are available runtime ranges rather than bit-specific measurements.

Model	Bits	CoBIT	CodeQuant	ParoQuant	SqueezeLLM	QuaRot$_{\text{GPTQ}}$
Qwen3-4B	2	25	68	76	2.2	2–5
	3	29	64	65	2.2	2–5
	4	29	75	125	2.7	2–5
Qwen3-8B	2	37	118	90	3.1	2–5
	3	44	150	89	3.1	2–5
	4	42	117	79	4.0	2–5
Qwen3-14B	2	61	229	137	4.6	4–5
	3	70	224	122	4.9	4–5
	4	76	231	133	5.9	4–5

For the non-reasoning models, the total quantization time ranges from 0.41 to 2.40 hours and the peak memory ranges from 29.7 to 101.1 GB as the model size increases from 4B to 32B parameters. Under the reasoning-specific model and calibration configuration, the total quantization time remains between 0.30 and 0.62 hours, with a peak memory footprint between 40.0 and 59.8 GB. Across these reasoning runs, activation LUT calibration requires at most 0.15 hours, while rotation and scaling processing and weight clustering account for most of the end-to-end cost.

All profiles use PyTorch's native scaled-dot-product attention, a fixed calibration set, and the same processing recipe for every codebook size without model- or K-specific hyperparameter tuning. The reported end-to-end wall-clock times were collected on a shared node and represent the complete optimization pipeline rather than peak-throughput benchmarks. The implementation also supports FlashAttention2 (Dao, 2023), which can further reduce processing time.

A.4.3 Evaluation Protocol

We use EvalScope (Team, 2024) for all downstream evaluations. All evaluations are zero-shot. We use one generation per example for all benchmarks except AIME25 and AIME26, for which we report pass@1 averaged over eight independent runs. The maximum model sequence length is $32{,}768$ tokens. For the general benchmarks, we set `max_tokens` to 512 (except for MMLU where we set `max_tokens` to $2{,}048$). For the reasoning benchmarks, we set `max_tokens` to $30{,}720$ for both AIME25 and AIME26, and to $28{,}672$ for GPQA, HumanEval, and LiveCodeBench. The remaining task-specific decoding configurations are summarized in Table 15.

For distributional evaluation, we use held-out data disjoint from all data used for rotation and scaling optimization, weight clustering, and PhaseLUT calibration. Perplexity is computed on the WikiText-2 test split (Merity et al., 2016) and a shard of the C4 validation split (Raffel et al., 2023), using

Table 15: Decoding configurations for downstream evaluation.

Setting	Benchmarks	Decoding configuration
General	ARC, PIQA, HellaSwag, MMLU	no thinking, T=0
Reasoning	AIME25/26, GPQA, HumanEval, LiveCodeBench	thinking, T=0.6, p=0.95, k=20

2,048-token sequences and up to 524,288 evaluated tokens. Forward KL divergence is reported in nats on the held-out WikiText-2 test split, using 512-token sequences and 65,536 evaluated tokens.

A.5 Baseline Configuration

The baseline names below refer to matched-setting adaptations used for controlled comparison, not to each method's complete original configuration. We document every change from the released methods below.

A.5.1 Downstream Setup

Common setup. All baselines quantize the same five projections as our method: `q_proj`, `k_proj`, `v_proj`, `gate_proj`, and `up_proj`. Each output channel uses one quantizer spanning the full input dimension, with no column grouping. We match the 2/3/4-bit index widths (K=4/8/16). All methods use the shared calibration stream described above and apply their published preprocessing where applicable. For baselines that originally use integer activation quantization, we replace only the activation quantizer with a floating-point counterpart at the same bit width. Their weight quantizers remain unchanged.

GPTQ. GPTQ (Frantar et al., 2023) is not evaluated as a standalone baseline. We use it as the weight quantizer for methods that do not specify their own weight quantizer, such as QuaRot (Ashkboos et al., 2024). Our GPTQ implementation follows the authors' original codebase,[5] without additional optimizations. To match our quantization granularity, we set the group size to the full input dimension, yielding one quantizer per output channel.

QuaRot. QuaRot (Ashkboos et al., 2024) is a rotation-based quantization method. We follow the original paper and apply its randomized Hadamard rotation. For the matched projection scope, we omit rotation and quantization of `o_proj` and disable QuaRot's online rotations, leaving `down_proj` unquantized. GPTQ serves as QuaRot's weight quantizer.

SqueezeLLM. SqueezeLLM (Kim et al., 2024) uses Sensitivity-Based K-means Clustering with quantization granularity matching ours. For the weight-and-activation setting, we first fuse the same offline rotation and scaling preprocessing used by CoBIT, then apply SqueezeLLM to the transformed weights. The Dense-and-Sparse component is excluded because the comparison fixes a uniform weight bit width.

ParoQuant. We report the weight-only configuration obtained with the authors' released code and original procedure.[6] Under the matched A4W4 setup, the weight-and-activation procedure described in Appendix A.3 of the original paper did not yield a stable checkpoint, so that configuration is omitted.

LNQ. LNQ (Kim et al., 2025) is a layer-wise non-uniform quantization method that alternates between a closed-form update of the codebook and a coordinate-descent update of the assignments. We use the authors' released code[7] with its default settings to build the checkpoints. To match our quantization granularity, we set the number of groups to one, yielding one codebook per output channel. Because LNQ is a weight-only method, we report it in the weight-only setting only.

[5] https://github.com/ist-daslab/gptq
[6] https://github.com/z-lab/paroquant
[7] https://github.com/snu-mllab/GuidedQuant

CodeQuant. CodeQuant (Yin et al., 2026) uses the same quantization granularity as CoBIT. We retain its original outlier-processing procedure and use the shared calibration budget described above for both methods. The reported results therefore represent the matched-budget configuration rather than the larger calibration budget used in the original experiments.

A.5.2 System Setup

A4W4 denotes the quantized activation and weight values rather than 4-bit arithmetic. CoBIT, SqueezeLLM, and ParoQuant use FP16 multiplication. CoBIT decodes packed activations and weight indices into FP16 values within its linear kernel. For SqueezeLLM and ParoQuant, a separate kernel produces FP4-quantized activation values in FP16 format for their native A16W4 weight kernels. The A16W4 timings use those same weight kernels without activation quantization.

cuBLAS. cuBLAS is NVIDIA's optimized library for dense GPU linear algebra. We use its FP16 GEMM implementation as the A16W16 reference.

AutoAWQ. AutoAWQ (Lin et al., 2026) provides CUDA and Triton kernels for 4-bit weight inference. We use both backends with the ParoQuant weight-only checkpoint. Each backend is evaluated with native A16W4 inputs and with the A4W4 numerical path described above.

Marlin. Marlin (Frantar et al., 2024) introduced a highly optimized FP16×INT4 kernel for autoregressive inference. Our measurements use the optimized vLLM implementation to execute the ParoQuant weights as a strong A16W4 systems baseline. Because Marlin takes FP16 activations as input, it appears only in the A16W4 comparison.

A.6 Additional Evaluation Results

This section reports supplementary main results and the complete results behind the ablations in the main text. FKL denotes forward KL divergence to the BF16 model in nats. Avg is the mean accuracy over ARC, PIQA, HellaSwag, and MMLU. Unless stated otherwise, activations and the KV cache remain in BF16 in the weight-only experiments.

A.6.1 Performance Evaluation

A4W4 results. Table 16 reports the complete A4W4 results omitted from the main table. The clustering-based methods use $K = 16$ codebook entries per output channel.

Weight-only quantization. Tables 19 and 20 isolate weight quantization from activation quantization. The ParoQuant rows use group size 128 and therefore have finer scaling granularity and a slightly larger effective bit width than our per-row codebooks.

Codebook and assignment isolation. Tables 21 and 22 report the full factorial study. The highlighted Full/Sequential rows correspond to our method. In the codebook column, Full uses the complete $\mathbf{H}_q$ in the closed-form update, while Diag retains only its diagonal in a weighted Lloyd update. In the assignment column, Sequential denotes error-compensated assignment using $\mathbf{H}_q$, while Nearest assigns each weight independently. The transplanted SqueezeLLM and CodeQuant-style codebooks are frozen before one assignment pass.

At $K = 4$, applying *Sequential* rather than *Nearest* assignment to the frozen SqueezeLLM codebook raises average accuracy from 0.013 to 0.729 on Qwen3-8B and from 0.028 to 0.821 on Qwen3-14B. With *Sequential* assignment, these results are close to *Full* (0.725 and 0.821). Together with the *Full–Diag* comparison in Table 4, this shows that codebook construction contributes, but the assignment rule has a larger effect at low K and can make different learned codebooks perform similarly.

Use of off-diagonal Hessian information. We retain only contiguous diagonal blocks of $\mathbf{H}_q$ and vary the block size. Let $\mathcal{B}_b(\mathbf{H}_q)$ retain contiguous diagonal blocks of size b, and let $\mathrm{Diag}(\mathbf{H}_q)$

Table 16: A4W4 performance on general-purpose Qwen3 models. Lower is better for FKL and perplexity, while higher is better for task accuracy. The best quantized result within each model is bold.

Models	Methods	FKL (↓)	Wiki2 (↓)	C4 (↓)	ARC (↑)	PIQA (↑)	Hella (↑)	MMLU (↑)	Avg (↑)
Qwen3-4B	BF16	–	14.3903	19.8578	0.8988	0.8150	0.7334	0.7497	0.7992
	$\text{QuaRot}_{\text{GPTQ}}$	0.3878	19.9350	26.6690	0.8605	0.7720	**0.6866**	0.6807	**0.7500**
	SqueezeLLM	0.4251	17.7426	24.5019	0.8385	0.7209	0.6410	0.6616	0.7155
	CodeQuant	0.3636	17.0097	23.0250	0.8557	0.7541	0.6603	**0.6818**	0.7380
	CoBIT	**0.2710**	**16.7588**	**22.5461**	**0.8698**	**0.7786**	0.6791	0.6688	0.7491
Qwen3-8B	BF16	–	10.0499	15.4294	0.9171	0.8596	0.7981	0.7733	0.8370
	$\text{QuaRot}_{\text{GPTQ}}$	**0.1706**	10.8640	**16.5550**	**0.9019**	0.8036	0.7624	0.7446	0.8031
	SqueezeLLM	0.4317	13.1877	21.5051	0.8703	0.7884	0.6929	0.7116	0.7658
	CodeQuant	0.2650	11.5000	17.4040	0.8915	0.7840	0.7291	0.7297	0.7836
	CoBIT	0.1928	**10.8626**	16.6395	0.9010	**0.8243**	**0.7714**	**0.7447**	**0.8104**
Qwen3-14B	BF16	–	8.9276	13.8218	0.9298	0.8939	0.8439	0.8206	0.8721
	$\text{QuaRot}_{\text{GPTQ}}$	0.1767	9.7283	14.9158	0.9169	0.8596	**0.8283**	0.7968	**0.8504**
	SqueezeLLM	0.2560	10.5357	15.8355	0.9194	0.8553	0.8122	0.7797	0.8417
	CodeQuant	0.1949	10.0186	15.1851	0.9168	**0.8645**	0.8123	0.7962	0.8475
	CoBIT	**0.1676**	**9.6608**	**14.8187**	**0.9225**	0.8526	0.8195	**0.7976**	0.8481
Qwen3-32B	BF16	–	7.8634	12.4475	0.9408	0.9097	0.8585	0.8353	0.8861
	$\text{QuaRot}_{\text{GPTQ}}$	0.2961	**8.5128**	**13.3246**	0.9284	0.8825	0.8199	0.8165	0.8618
	SqueezeLLM	0.3994	9.0159	14.3045	0.9301	0.8738	0.8131	0.7897	0.8517
	CodeQuant	0.3402	8.8139	13.7465	0.9312	**0.8874**	0.8264	0.8143	0.8648
	CoBIT	**0.2829**	8.5968	13.5415	**0.9366**	0.8841	**0.8339**	**0.8193**	**0.8685**

Table 17: A4W3 and A4W2 performance on Qwen3-14B. FKL and Hella denote forward KL divergence and HellaSwag, respectively. The best quantized result within each bit width is bold.

Models	W Bits	Methods	FKL ↓	Wiki2 ↓	C4 ↓	ARC ↑	PIQA ↑	Hella ↑	MMLU ↑	Avg ↑
Qwen3-14B	–	BF16	–	8.93	13.82	0.930	0.894	0.844	0.821	0.872
	3	$\text{QuaRot}_{\text{GPTQ}}$	0.364	10.68	15.84	0.892	0.813	0.784	0.743	0.808
		SqueezeLLM	0.634	13.46	19.81	0.889	0.794	0.714	0.608	0.751
		CodeQuant	0.367	11.50	17.05	0.903	0.851	0.777	0.761	0.823
		CoBIT	**0.218**	**10.04**	**15.21**	**0.917**	**0.853**	**0.787**	**0.783**	**0.835**
	2	$\text{QuaRot}_{\text{GPTQ}}$	2.404	39.93	46.18	0.060	0.092	0.045	0.049	0.061
		SqueezeLLM	2.383	55.93	73.18	0.051	0.095	0.034	0.049	0.057
		CodeQuant	1.024	19.29	26.87	0.793	0.721	0.458	0.477	0.613
		CoBIT	**0.473**	**12.24**	**17.66**	**0.886**	**0.815**	**0.763**	**0.691**	**0.789**

denote the matrix obtained by zeroing all off-diagonal entries. The retained off-diagonal mass is

$$\eta_b = \frac{\|\mathcal{B}_b(\mathbf{H}_q) - \mathrm{Diag}(\mathbf{H}_q)\|_F}{\|\mathbf{H}_q - \mathrm{Diag}(\mathbf{H}_q)\|_F}. \tag{46}$$

The metric excludes the diagonal, so diagonal and full Hessians correspond to $\eta_b = 0$ and 1. Table 23 shows that contiguous blocks retain little off-diagonal mass after rotation. Tables 24 and 25 give the complete accuracy results. The Full rows correspond to our method and are highlighted.

Across both models and all codebook sizes, retaining more off-diagonal information consistently lowers the optimization objective and FKL. The downstream effect is largest at $K = 4$. Moving from a diagonal to the complete Hessian raises average accuracy from 0.1501 to 0.7246 on Qwen3-8B and from 0.0943 to 0.8214 on Qwen3-14B. This supports the main-text observation that error interactions become more important as the number of centroids decreases.

Assignment displacement. Table 26 compares nearest-centroid assignment with our sequential assignment while holding the final codebook fixed. Across models and codebook sizes, sequential compensation changes about 13% of assignments. More than 99.9% of these changes move to an adjacent centroid. The flip rate increases with column position, consistent with accumulated error propagation.

Table 18: A4W4 performance on reasoning models. A25, A26, HE, and LCB denote AIME25, AIME26, HumanEval, and LiveCodeBench. The best quantized result within each model is bold.

Method	Qwen3-4B-Thinking-2507						Llama-3.1-Nemotron-Nano-4B						DeepSeek-R1-Distill-Llama-8B					
	A25	A26	GPQA	HE	LCB	Avg	A25	A26	GPQA	HE	LCB	Avg	A25	A26	GPQA	HE	LCB	Avg
BF16	0.692	0.725	0.702	0.970	0.739	0.766	0.546	0.621	0.505	0.860	0.604	0.627	0.350	0.375	0.510	0.866	0.534	0.527
$\text{QuaRot}_{\text{GPTQ}}$	0.479	0.488	0.546	**0.939**	0.549	0.600	**0.371**	0.321	0.349	0.805	0.409	0.451	0.246	**0.246**	0.333	0.762	**0.381**	**0.394**
SqueezeLLM	0.379	0.383	0.439	0.915	0.474	0.518	0.338	0.283	0.338	0.829	0.385	0.435	0.217	0.179	**0.354**	0.720	0.311	0.356
CodeQuant	0.483	0.467	0.530	0.921	0.516	0.583	0.308	0.263	0.338	0.744	0.333	0.397	0.171	0.129	0.242	0.738	0.303	0.317
CoBIT	**0.563**	**0.600**	**0.561**	0.921	**0.612**	**0.651**	**0.371**	**0.333**	**0.399**	**0.848**	**0.455**	**0.481**	**0.267**	0.213	**0.354**	**0.768**	0.371	**0.394**

Table 19: Complete weight-only results on Qwen3-8B. Lower is better for FKL and perplexity. Higher is better for all accuracy columns.

Method	Weight	Bits/W	FKL ↓	Wiki2 ↓	C4 ↓	ARC ↑	PIQA ↑	Hella ↑	MMLU ↑	Avg ↑
BF16	BF16	16.00	0	10.0499	15.4294	0.9171	0.8596	0.7981	0.7733	0.8370
RTN	INT4	4.00	0.3159	12.0790	18.3348	0.8864	0.8052	0.7284	0.7178	0.7844
RTN	INT3	3.00	2.1095	49.6453	70.5940	0.1049	0.2187	0.0650	0.0464	0.1087
RTN	INT2	2.00	11.9035	1076269.3750	1577357.1250	0.0781	0.0925	0.0719	0.0606	0.0758
GPTQ ($\text{QuaRot}_{\text{GPTQ}}$)	INT4, per-channel	4.00	0.0617	10.2689	15.7487	0.9129	0.8297	0.7909	0.7661	0.8249
GPTQ ($\text{QuaRot}_{\text{GPTQ}}$)	INT3, per-channel	3.00	0.2647	11.4981	17.0473	0.8588	0.7835	0.7224	0.6849	0.7624
GPTQ ($\text{QuaRot}_{\text{GPTQ}}$)	INT2, per-channel	2.00	2.4792	46.1509	54.6752	0.0305	0.0550	0.0280	0.0502	0.0409
SqueezeLLM	$K = 16$	4.06	0.1410	10.8059	16.7716	0.9112	0.8466	0.7797	0.7522	0.8224
SqueezeLLM	$K = 8$	3.03	0.5264	13.5282	19.9950	0.8534	0.7612	0.6473	0.5567	0.7046
SqueezeLLM	$K = 4$	2.02	2.8683	129.8409	162.7274	0.0130	0.0256	0.0061	0.0080	0.0132
ParoQuant	INT4, g128	4.16	0.0248	10.1523	15.5649	0.9168	0.8547	0.7909	0.7663	0.8322
ParoQuant	INT3, g128	3.15	0.1099	10.7046	16.1327	0.9061	0.8341	0.7632	0.7461	0.8124
ParoQuant	INT2, g128	2.14	0.6180	16.7078	22.2812	0.7754	0.5609	0.3831	0.5630	0.5706
LNQ	$K = 16$	4.06	0.0297	10.1984	15.6398	0.9160	0.8591	0.7926	0.7802	0.8370
LNQ	$K = 8$	3.03	0.1191	10.7150	16.2020	0.9076	0.8373	0.7834	0.7620	0.8226
LNQ	$K = 4$	2.02	0.5499	16.3375	19.7724	0.8492	0.6338	0.7150	0.5452	0.6858
CodeQuant	$K = 16$	4.06	0.1049	10.4282	15.8784	0.9096	0.8368	0.7705	0.7616	0.8196
CodeQuant	$K = 8$	3.03	0.3437	12.3651	18.3729	0.8833	0.6953	0.7107	0.7148	0.7510
CodeQuant	$K = 4$	2.02	1.7230	40.6213	50.9218	0.1801	0.1431	0.1515	0.0493	0.1310
CoBIT	$K = 16$	4.06	0.0326	10.1556	15.5634	0.9140	0.8580	0.7941	0.7791	0.8363
CoBIT	$K = 8$	3.03	0.1064	10.5482	16.0467	0.9042	0.8477	0.7756	0.7635	0.8227
CoBIT	$K = 4$	2.02	0.4047	12.7396	19.2323	0.8628	0.6643	0.7137	0.6576	0.7246

Native FP4 formats. Table 27 compares CoBIT with native MXFP4 and NVFP4 activation formats using the same 4-bit ($K = 16$) weight checkpoint. The native formats use their standard scale granularity, with 32 values per MXFP4 scale and 16 per NVFP4 scale. The per-token controls change only the activation quantizer, separating format choice from scale granularity and weight optimization.

A.6.2 Understanding Generation Failures

We study whether extremely low generative accuracies arise from answer-extraction errors or genuine model degradation, and whether collapsed checkpoints exhibit abnormal truncation. We analyze 40 Qwen3-8B checkpoints and 138 checkpoint–task pairs across BF16, the evaluated baselines, CoBIT at $K \in \{4, 8, 16\}$ with BF16 or FP4 activations, and internal ablations. Weight-only and FP4-activation checkpoints are treated separately, and a checkpoint is classified as collapsed when it has extremely low original accuracy on every evaluated task. To determine whether the low scores are caused by answer-extraction errors, we re-evaluate every generated response with a stricter extractor. It returns an answer only when the response explicitly states a choice or uniquely matches one candidate option, avoiding the unreliable fallback that treats the final capital letter in the response as the prediction. We also measure the truncation rate, defined as the fraction of responses that reach the generation-length limit, to identify responses that continue generating until forced termination.

Table 28 reports results for the collapsed and non-collapsed checkpoint groups. **Checkpoints** and **Checkpoint–task pairs** give the number of checkpoints and individual task evaluations in each group. **Original accuracy** is obtained using the original evaluator, whereas **Re-extracted accuracy** uses the stricter extractor. **Accuracy change** is the re-extracted accuracy minus the original accuracy, with the 95% confidence interval obtained by bootstrapping checkpoints. **Median truncation rate** is the median fraction of responses reaching the generation-length limit across checkpoint–task

Table 20: Complete weight-only results on Qwen3-14B.

Method	Weight	Bits/W	FKL ↓	Wiki2 ↓	C4 ↓	ARC ↑	PIQA ↑	Hella ↑	MMLU ↑	Avg ↑
BF16	BF16	16.00	0	8.9276	13.8218	0.9298	0.8939	0.8439	0.8206	0.8721
RTN	INT4	4.00	0.2787	10.6630	16.0418	0.9151	0.8444	0.8065	0.7682	0.8336
RTN	INT3	3.00	1.8484	32.3490	45.1090	0.0584	0.0582	0.0326	0.0446	0.0485
RTN	INT2	2.00	10.6423	262319.8125	302615.5000	0.0316	0.0245	0.0265	0.0160	0.0247
GPTQ ($\text{QuaRot}_{\text{GPTQ}}$)	INT4, per-channel	4.00	0.0471	9.1370	14.0581	0.9278	0.8890	0.8413	0.8122	0.8676
GPTQ ($\text{QuaRot}_{\text{GPTQ}}$)	INT3, per-channel	3.00	0.2080	9.9788	14.8802	0.9129	0.8455	0.8001	0.7848	0.8358
GPTQ ($\text{QuaRot}_{\text{GPTQ}}$)	INT2, per-channel	2.00	1.6353	23.7070	29.6680	0.0330	0.0724	0.0348	0.0345	0.0437
SqueezeLLM	$K = 16$	4.05	0.1213	9.7420	14.8381	0.9227	0.8776	0.8276	0.8039	0.8579
SqueezeLLM	$K = 8$	3.03	0.4292	11.3554	17.2480	0.9045	0.8422	0.7644	0.7195	0.8076
SqueezeLLM	$K = 4$	2.01	2.2574	55.1697	69.4736	0.0270	0.0664	0.0106	0.0097	0.0284
ParoQuant	INT4, g128	4.16	0.0198	9.0018	13.9424	0.9287	0.8896	0.8419	0.8167	0.8692
ParoQuant	INT3, g128	3.15	0.0882	9.4450	14.4054	0.9242	0.8808	0.8335	0.8034	0.8605
ParoQuant	INT2, g128	2.14	0.5003	13.6130	18.1877	0.8816	0.7584	0.7171	0.6783	0.7589
LNQ	$K = 16$	4.05	0.0240	9.0136	13.9602	0.9307	0.8890	0.8416	0.8168	0.8695
LNQ	$K = 8$	3.03	0.0921	9.4931	14.3419	0.9236	0.8754	0.8276	0.8060	0.8582
LNQ	$K = 4$	2.01	0.4272	12.3260	16.8755	0.8810	0.7922	0.7576	0.7122	0.7858
CodeQuant	$K = 16$	4.05	0.0574	9.2037	14.1546	0.9276	0.8847	0.8316	0.8152	0.8648
CodeQuant	$K = 8$	3.03	0.2126	10.2470	15.5718	0.9208	0.8645	0.8125	0.7877	0.8464
CodeQuant	$K = 4$	2.01	0.8000	15.9932	22.7561	0.8472	0.7427	0.5096	0.5646	0.6660
CoBIT	$K = 16$	4.05	0.0238	9.0529	13.9179	0.9295	0.8836	0.8448	0.8154	0.8683
CoBIT	$K = 8$	3.03	0.0854	9.4319	14.3100	0.9236	0.8727	0.8274	0.8103	0.8585
CoBIT	$K = 4$	2.01	0.3465	11.1836	16.5605	0.9053	0.8406	0.8029	0.7369	0.8214

pairs. Accuracies and accuracy changes are averaged first across tasks within each checkpoint and then across checkpoints.

For the 10 collapsed checkpoints, re-extraction changes the average accuracy by only -1.54% points, with a 95% confidence interval of $[-4.05, 1.23]$. The largest single-task increase occurs for CodeQuant K=4 on PIQA, where accuracy rises from 0.143 to 0.309 after re-extraction. Thus, answer extraction does not explain the extremely low accuracies. These checkpoints instead have a median truncation rate of 99.5%, compared with 0% for the 30 non-collapsed checkpoints, showing that their responses almost always reach the generation-length limit. Together with their degraded WikiText-2 perplexity and FKL, these results indicate genuine model degradation rather than an evaluation artifact.

A.6.3 System Evaluation

We evaluate system performance at both the kernel and end-to-end levels. For kernel-level measurements, we use synthetic activations with the exact linear-layer shapes of Qwen3-4B, Qwen3-8B, Qwen3-14B, and Qwen3-32B (Yang et al., 2025). We select one decoder layer from each model and sum the latency of its seven linear projections. The CoBIT measurements include both FP4 activation encoding and clustered linear computation.

For end-to-end measurements, we evaluate Qwen3-4B and Qwen3-8B using 128-token prompts followed by 128 tokens generated with greedy decoding. We test batch sizes of 1, 2, 4, and 8 in a decoder with CUDA graph enabled. The evaluated kernels replace the Q/K/V and gate/up projections. All remaining projections and operators use the same implementation across methods. Each result is the median of five runs. Lower latency is better. OOM indicates that the model does not fit in GPU memory. The system baseline configurations are described in the preceding section.

Kernel-level latency. Table 29 summarizes the latency of all seven linear projections in one decoder layer for each of Qwen3-4B, Qwen3-8B, Qwen3-14B, and Qwen3-32B. At $M = 1$, these operations require $6{,}169\,\mu$s with dense FP16 cuBLAS, $1{,}775\,\mu$s with SqueezeLLM (Kim et al., 2024), and $1{,}687\,\mu$s with CoBIT. The last two columns report the baseline latency divided by the CoBIT latency to indicate speedup.

At $M = 1$, CoBIT and SqueezeLLM have similar aggregate latency, while both substantially reduce latency relative to dense FP16. As M increases, SqueezeLLM latency grows more quickly. The CoBIT speedup over SqueezeLLM consequently increases to 3.01–$3.30\times$ at $M = 8$.

End-to-end decode latency. Table 31 reports milliseconds per generated token and excludes the prefilling stage. The three header rows identify the method, kernel backend, and activation/weight

Table 21: Complete codebook and assignment ablation on Qwen3-8B.

K	Codebook	Assignment	FKL ↓	Wiki2 ↓	C4 ↓	ARC ↑	PIQA ↑	Hella ↑	MMLU ↑	Avg ↑
16	Full	Sequential	0.0326	10.1556	15.5634	0.9140	0.8580	0.7941	0.7791	0.8363
16	Full	Nearest	0.0878	10.5976	16.0501	0.9081	0.8536	0.7892	0.7710	0.8305
16	Diag	Sequential	0.0336	10.1751	15.6879	0.9140	0.8526	0.7948	0.7778	0.8348
16	Diag	Nearest	0.1352	10.4746	16.1202	0.9050	0.8275	0.7793	0.7561	0.8170
16	SqueezeLLM	Nearest	0.1410	10.8059	16.7716	0.9112	0.8466	0.7797	0.7522	0.8224
16	SqueezeLLM	Sequential	0.0324	10.2072	15.6272	0.9191	0.8564	0.7849	0.7560	0.8291
16	CodeQuant-style	Nearest	0.0950	10.3315	15.8952	0.9095	0.8406	0.7903	0.7681	0.8271
16	CodeQuant-style	Sequential	0.0347	10.2306	15.6213	0.9154	0.8498	0.7958	0.7726	0.8334
8	Full	Sequential	0.1064	10.5482	16.0467	0.9042	0.8477	0.7756	0.7635	0.8227
8	Full	Nearest	0.3234	11.6373	17.5501	0.8791	0.8030	0.7284	0.7025	0.7782
8	Diag	Sequential	0.1137	10.5957	16.2164	0.9028	0.8303	0.7864	0.7567	0.8191
8	Diag	Nearest	0.5567	15.8697	22.1068	0.8436	0.7530	0.6007	0.5471	0.6861
8	SqueezeLLM	Nearest	0.5264	13.5282	19.9950	0.8534	0.7612	0.6473	0.5567	0.7046
8	SqueezeLLM	Sequential	0.1065	10.4530	16.0753	0.9044	0.8362	0.7680	0.7580	0.8166
8	CodeQuant-style	Nearest	0.3662	12.6715	18.8016	0.8757	0.7737	0.7055	0.6952	0.7625
8	CodeQuant-style	Sequential	0.1172	10.5633	16.1972	0.9036	0.8281	0.7639	0.7597	0.8138
4	Full	Sequential	0.4047	12.7396	19.2323	0.8628	0.6643	0.7137	0.6576	0.7246
4	Full	Nearest	1.7007	32.1599	44.6669	0.3678	0.4614	0.2510	0.2639	0.3360
4	Diag	Sequential	0.5148	13.5494	20.6015	0.8588	0.6110	0.6981	0.6480	0.7040
4	Diag	Nearest	2.8280	104.8059	135.2803	0.1440	0.2057	0.1259	0.1247	0.1501
4	SqueezeLLM	Nearest	2.8683	129.8409	162.7274	0.0130	0.0256	0.0061	0.0080	0.0132
4	SqueezeLLM	Sequential	0.4230	12.5713	18.9398	0.8729	0.6513	0.7287	0.6636	0.7291
4	CodeQuant-style	Nearest	2.0125	54.2493	65.2342	0.2264	0.2682	0.1272	0.1731	0.1987
4	CodeQuant-style	Sequential	0.4455	13.0824	19.4312	0.8594	0.6197	0.6934	0.6452	0.7044

Table 22: Complete codebook and assignment ablation on Qwen3-14B.

K	Codebook	Assignment	FKL ↓	Wiki2 ↓	C4 ↓	ARC ↑	PIQA ↑	Hella ↑	MMLU ↑	Avg ↑
16	Full	Sequential	0.0238	9.0529	13.9179	0.9295	0.8836	0.8448	0.8154	0.8683
16	Full	Nearest	0.0468	9.1595	14.0988	0.9264	0.8857	0.8367	0.8132	0.8655
16	Diag	Sequential	0.0243	9.0747	13.9720	0.9284	0.8945	0.8407	0.8135	0.8693
16	Diag	Nearest	0.1134	9.5794	14.7456	0.9225	0.8776	0.8219	0.8066	0.8571
16	SqueezeLLM	Nearest	0.1213	9.7420	14.8381	0.9227	0.8776	0.8276	0.8039	0.8579
16	SqueezeLLM	Sequential	0.0238	9.0873	13.9553	0.9261	0.8896	0.8418	0.8108	0.8671
16	CodeQuant-style	Nearest	0.0532	9.0604	14.0714	0.9262	0.8819	0.8352	0.8144	0.8644
16	CodeQuant-style	Sequential	0.0256	9.0339	13.9247	0.9258	0.8836	0.8383	0.8147	0.8656
8	Full	Sequential	0.0854	9.4319	14.3100	0.9236	0.8727	0.8274	0.8103	0.8585
8	Full	Nearest	0.1746	9.5112	14.7046	0.9185	0.8591	0.8097	0.7920	0.8448
8	Diag	Sequential	0.0879	9.4344	14.4745	0.9267	0.8760	0.8381	0.8097	0.8626
8	Diag	Nearest	0.4944	12.9290	18.6522	0.8108	0.8346	0.7304	0.7206	0.7741
8	SqueezeLLM	Nearest	0.4292	11.3554	17.2480	0.9045	0.8422	0.7644	0.7195	0.8076
8	SqueezeLLM	Sequential	0.0839	9.4557	14.4376	0.9236	0.8819	0.8357	0.8040	0.8613
8	CodeQuant-style	Nearest	0.2407	10.4925	15.7242	0.9177	0.8640	0.8145	0.7820	0.8446
8	CodeQuant-style	Sequential	0.0877	9.3695	14.3784	0.9251	0.8770	0.8339	0.8072	0.8608
4	Full	Sequential	0.3465	11.1836	16.5605	0.9053	0.8406	0.8029	0.7369	0.8214
4	Full	Nearest	1.1070	18.8401	26.4439	0.7334	0.6915	0.3323	0.1968	0.4885
4	Diag	Sequential	0.4113	11.7295	17.7157	0.9045	0.8493	0.7785	0.7250	0.8143
4	Diag	Nearest	2.2029	46.3832	64.4826	0.0973	0.1572	0.0345	0.0880	0.0943
4	SqueezeLLM	Nearest	2.2574	55.1697	69.4736	0.0270	0.0664	0.0106	0.0097	0.0284
4	SqueezeLLM	Sequential	0.3499	11.1716	16.5946	0.9050	0.8422	0.7963	0.7411	0.8212
4	CodeQuant-style	Nearest	1.3536	23.2303	31.2374	0.4025	0.5892	0.1496	0.0996	0.3102
4	CodeQuant-style	Sequential	0.3615	11.3591	16.7972	0.9011	0.8373	0.7864	0.7336	0.8146

Table 23: Retained off-diagonal Hessian mass for each contiguous block size. Q/K/V and gate/up share the same input activation and agree to the shown precision.

Model	$b = 16$	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
Qwen3-8B	0.0604	0.0869	0.1241	0.1760	0.2494	0.3532	0.4998	0.7069
Qwen3-14B	0.0538	0.0777	0.1110	0.1574	0.2232	0.3159	0.4471	0.5972

precision. For SqueezeLLM and ParoQuant, paired columns use the same weight kernel, with the A4W4 path additionally including activation quantization.

The kernel-level gains remain visible in the full decoder. At batch 1, CoBIT is 1.44–$1.53\times$ faster than dense FP16 where the FP16 model fits and 1.01–$1.07\times$ faster than the highly optimized Marlin path. Across both GPUs, both models, and all batch sizes, CoBIT is also faster than the A4W4 numerical paths of SqueezeLLM (Kim et al., 2024) and ParoQuant (Liang et al., 2026). CoBIT's latencies include PhaseLUT lookup and activation encoding.

For Qwen3-8B on RTX A6000 at batch 1, the five evaluated projections summed over 36 layers take 13.03 ms/token with FP16 cuBLAS and 4.68 ms/token with CoBIT. In a separate activation-path ablation, fusing encoding with RMSNorm and sharing the result across Q/K/V and gate/up reduces decoding latency from 17.49 to 16.98 ms/token.

End-to-end generation latency. The previous measurements isolate autoregressive decoding. Table 32 instead measures the complete request, including prefill of the 128-token prompt and generation of 128 tokens. The reported value is total wall-clock time in milliseconds. For any baseline, its end-to-end speedup relative to CoBIT is obtained by dividing the corresponding table entry by the CoBIT entry in the same row.

Including prefill, CoBIT remains 1.23–$1.52\times$ faster than FP16 on the RTX 5060 Ti and 1.13–$1.48\times$ faster on the RTX A6000 where FP16 fits. Relative to the matched A4W4 paths, CoBIT achieves 1.34–$2.50\times$ over SqueezeLLM, 1.08–$1.42\times$ over AutoAWQ (Lin et al., 2026) CUDA, and 1.21–$1.67\times$ over AutoAWQ Triton across the two GPUs. Against the weight-only Marlin (Frantar et al., 2024) path, CoBIT ranges from 0.85 to $1.07\times$: it is competitive at batch 1, but Marlin gains an advantage in some larger-batch settings.

Table 24: Complete off-diagonal Hessian information curve on Qwen3-8B. The SqueezeLLM rows report the original method and serve as a diagonal-information reference.

K	Mode	b	η_b	Obj. ↓	FKL ↓	Wiki2 ↓	C4 ↓	ARC ↑	PIQA ↑	Hella ↑	MMLU ↑	Avg ↑
16	Diag	1	0	2.573e-3	0.1352	10.4746	16.1202	0.9050	0.8275	0.7793	0.7561	0.8170
16	Block	32	0.0869	1.345e-3	0.0893	10.5718	16.1240	0.9090	0.8444	0.7781	0.7709	0.8256
16	Block	512	0.3532	9.972e-4	0.0507	10.2147	15.7679	0.9123	0.8411	0.7908	0.7648	0.8273
16	Block	2048	0.7069	8.011e-4	0.0376	10.2807	15.6779	0.9143	0.8471	0.7894	0.7784	0.8323
16	Full	4096	1	6.899e-4	0.0326	10.1556	15.5634	0.9140	0.8580	0.7941	0.7791	0.8363
16	SqueezeLLM	1	0	–	0.1410	10.8059	16.7716	0.9112	0.8466	0.7797	0.7522	0.8224
8	Diag	1	0	9.654e-3	0.5567	15.8697	22.1068	0.8436	0.7530	0.6007	0.5471	0.6861
8	Block	32	0.0869	5.005e-3	0.3394	12.3194	18.4771	0.8836	0.7508	0.7096	0.7160	0.7650
8	Block	512	0.3532	3.643e-3	0.1788	10.9499	16.6766	0.8991	0.8090	0.7619	0.7491	0.8048
8	Block	2048	0.7069	2.913e-3	0.1261	10.7057	16.1834	0.9036	0.8177	0.7628	0.7531	0.8093
8	Full	4096	1	2.468e-3	0.1064	10.5482	16.0467	0.9042	0.8477	0.7756	0.7635	0.8227
8	SqueezeLLM	1	0	–	0.5264	13.5282	19.9950	0.8534	0.7612	0.6473	0.5567	0.7046
4	Diag	1	0	3.735e-2	2.8280	104.8059	135.2803	0.1440	0.2057	0.1259	0.1247	0.1501
4	Block	32	0.0869	1.906e-2	1.5048	27.5916	37.7754	0.3337	0.5283	0.2529	0.2246	0.3349
4	Block	512	0.3532	1.339e-2	0.7676	15.4766	22.7775	0.7900	0.6240	0.5890	0.5371	0.6350
4	Block	2048	0.7069	1.066e-2	0.4992	12.9207	19.4848	0.8374	0.6850	0.6584	0.6184	0.6998
4	Full	4096	1	9.187e-3	0.4047	12.7396	19.2323	0.8628	0.6643	0.7137	0.6576	0.7246
4	SqueezeLLM	1	0	–	2.8683	129.8409	162.7274	0.0130	0.0256	0.0061	0.0080	0.0132

Table 25: Complete off-diagonal Hessian information curve on Qwen3-14B.

K	Mode	b	η_b	Obj. ↓	FKL ↓	Wiki2 ↓	C4 ↓	ARC ↑	PIQA ↑	Hella ↑	MMLU ↑	Avg ↑
16	Diag	1	0	2.636e-3	0.1134	9.5794	14.7456	0.9225	0.8776	0.8219	0.8066	0.8571
16	Block	64	0.1110	1.341e-3	0.0454	9.0037	14.0019	0.9273	0.8857	0.8375	0.8126	0.8658
16	Block	1024	0.4471	1.004e-3	0.0312	9.0861	14.0064	0.9278	0.8879	0.8414	0.8136	0.8677
16	Block	2048	0.5972	9.198e-4	0.0283	9.1104	14.0266	0.9287	0.8961	0.8401	0.8147	0.8699
16	Full	5120	1	7.434e-4	0.0238	9.0529	13.9179	0.9295	0.8836	0.8448	0.8154	0.8683
16	SqueezeLLM	1	0	–	0.1213	9.7420	14.8381	0.9227	0.8776	0.8276	0.8039	0.8579
8	Diag	1	0	9.907e-3	0.4944	12.9290	18.6522	0.8108	0.8346	0.7304	0.7206	0.7741
8	Block	64	0.1110	4.873e-3	0.1648	9.5316	14.7858	0.9222	0.8585	0.8118	0.7941	0.8467
8	Block	1024	0.4471	3.626e-3	0.1112	9.3869	14.4254	0.9253	0.8770	0.8304	0.8037	0.8591
8	Block	2048	0.5972	3.323e-3	0.1034	9.4583	14.4534	0.9216	0.8700	0.8315	0.8054	0.8571
8	Full	5120	1	2.690e-3	0.0854	9.4319	14.3100	0.9236	0.8727	0.8274	0.8103	0.8585
8	SqueezeLLM	1	0	–	0.4292	11.3554	17.2480	0.9045	0.8422	0.7644	0.7195	0.8076
4	Diag	1	0	3.923e-2	2.2029	46.3832	64.4826	0.0973	0.1572	0.0345	0.0880	0.0943
4	Block	64	0.1110	1.819e-2	0.7742	15.0883	21.6445	0.8419	0.7073	0.5787	0.5977	0.6814
4	Block	1024	0.4471	1.312e-2	0.4797	11.9040	17.8952	0.8966	0.8188	0.7568	0.7057	0.7945
4	Block	2048	0.5972	1.201e-2	0.4333	11.8749	17.4489	0.8963	0.7873	0.7599	0.7237	0.7918
4	Full	5120	1	9.752e-3	0.3465	11.1836	16.5605	0.9053	0.8406	0.8029	0.7369	0.8214
4	SqueezeLLM	1	0	–	2.2574	55.1697	69.4736	0.0270	0.0664	0.0106	0.0097	0.0284

Table 26: Assignment displacement relative to nearest-centroid assignment. The slope is measured against the normalized input-column index.

Model	K	Rule	Flipped ↑	$\lvert\Delta k\rvert = 1$	$\lvert\Delta k\rvert \geq 2$	Slope	FKL ↓	Wiki2 ↓	C4 ↓
Qwen3-8B	16	Nearest	0%	–	–	0	0.1330	10.5979	16.3141
	16	Sequential	13.01%	99.97%	0.03%	0.199	0.0326	10.1556	15.5634
	8	Nearest	0%	–	–	0	0.5525	15.6236	22.4152
	8	Sequential	13.04%	99.97%	0.03%	0.201	0.1064	10.5482	16.0467
	4	Nearest	0%	–	–	0	2.8989	122.2285	148.3698
	4	Sequential	12.67%	99.95%	0.05%	0.199	0.4047	12.7396	19.2323
Qwen3-14B	16	Nearest	0%	–	–	0	0.1139	9.4311	14.6596
	16	Sequential	13.02%	99.96%	0.04%	0.199	0.0238	9.0529	13.9179
	8	Nearest	0%	–	–	0	0.4627	12.0533	18.0396
	8	Sequential	13.06%	99.96%	0.04%	0.201	0.0854	9.4319	14.3100
	4	Nearest	0%	–	–	0	2.2951	53.5844	73.4757
	4	Sequential	12.62%	99.94%	0.06%	0.198	0.3465	11.1836	16.5605

Table 27: Complete FP4 activation-format comparison on Qwen3-8B and Qwen3-14B with 4-bit weights ($K = 16$).

Model	Scale setting	Act. bits	Composition	FKL ↓	Wiki2 ↓	C4 ↓	ARC ↑	PIQA ↑	Hella ↑	MMLU ↑	Avg ↑
Qwen3-8B	**Per-token E8M0 (ours)**	4.00	PhaseLUT	0.1928	10.8626	16.6395	0.9010	0.8243	0.7714	0.7447	0.8104
	Per-token E8M0 (MX rule)	4.00	E2M1	0.2020	10.9293	16.7174	0.8980	0.8123	0.7670	0.7445	0.8054
	Per-token E4M3+FP32 (NV rule)	4.01	E2M1	0.1915	10.8918	16.6728	0.8991	0.8243	0.7642	0.7511	0.8097
	MXFP4, block 32	4.25	E2M1	0.1901	11.1885	16.8815	0.9005	0.8063	0.7568	0.7539	0.8044
	NVFP4, block 16	4.50+	E2M1	0.1468	10.8691	16.3683	0.9028	0.8357	0.7696	0.7383	0.8116
Qwen3-14B	**Per-token E8M0 (ours)**	4.00	PhaseLUT	0.1676	9.6608	14.8187	0.9225	0.8526	0.8195	0.7976	0.8481
	Per-token E8M0 (MX rule)	4.00	E2M1	0.1756	9.6838	14.8579	0.9205	0.8596	0.8246	0.7967	0.8504
	Per-token E4M3+FP32 (NV rule)	4.01	E2M1	0.1655	10.0107	14.9968	0.9211	0.8651	0.8336	0.7935	0.8533
	MXFP4, block 32	4.25	E2M1	0.1649	9.8750	14.9104	0.9219	0.8624	0.8169	0.7925	0.8484
	NVFP4, block 16	4.50+	E2M1	0.1202	9.7315	14.6772	0.9205	0.8830	0.8355	0.8018	0.8602

Table 28: Generation rescoring of 40 Qwen3-8B checkpoints covering 138 checkpoint–task pairs. A checkpoint is classified as collapsed when its original accuracy is below chance on every evaluated task. Original and re-extracted accuracies are averaged first across tasks within each checkpoint and then across checkpoints. Accuracy change is the re-extracted accuracy minus the original accuracy, with confidence intervals obtained by bootstrapping checkpoints. The truncation rate measures the fraction of responses that reach the generation-length limit.

Checkpoint group	Checkpoints	Checkpoint–task pairs	Original accuracy	Re-extracted accuracy	Accuracy change [95% CI]	Median truncation rate
Collapsed: extremely low accuracy on all tasks	10	40	0.065	0.050	−0.015 [−0.041, 0.012]	99.5%
Not collapsed: otherwise	30	98	0.714	0.719	+0.005 [0.003, 0.008]	0.0%

Table 29: Aggregate kernel latency over one decoder layer from each of Qwen3-4B, Qwen3-8B, Qwen3-14B, and Qwen3-32B. M is the number of token rows processed by the linear operator. Latency sums all seven projections from the four layers.

GPU	M	cuBLAS (μs) ↓	SqueezeLLM (μs) ↓	CoBIT (μs) ↓	vs. cuBLAS ↑	vs. Squeeze ↑
RTX 5060 Ti	1	6169	1775	1687	3.66×	1.05×
	2	6015	2725	1790	3.36×	1.52×
	4	6063	4714	2022	3.00×	2.33×
	8	6245	8822	2676	2.33×	3.30×
RTX A6000	1	3176	1194	1164	2.73×	1.03×
	2	3167	1885	1266	2.50×	1.49×
	4	3176	3226	1477	2.15×	2.18×
	8	3174	5828	1935	1.64×	3.01×

Table 30: Effective bandwidth (GB/s) on RTX A6000 at $M = 1$. CoBIT includes activation encoding and clustered linear computation; both ParoQuant variants include rotation.

Kernel	**GB/s**
FP16 cuBLAS	722
INT4 matvec	632
Marlin	617
SqueezeLLM	491
CoBIT (encode + GEMM)	509
ParoQuant (AWQ kernel)	482
ParoQuant (AWQ–Marlin)	610

Table 31: End-to-end decode latency in milliseconds per generated token (↓). All measurements use a 128-token prompt and generate 128 tokens.

GPU	**Model**	**Batch**	**FP16**	**CoBIT**	**QuaRot**$_{\text{GPTQ}}$	**SqueezeLLM**		**ParoQuant**				**ParoQuant**
			cuBLAS	Ours	CUTLASS	Official CUDA		AutoAWQ CUDA		AutoAWQ Triton		Marlin
			A16W16	A4W4	A4W4	A16W4	A4W4	A16W4	A4W4	A16W4	A4W4	A16W4
RTX 5060 Ti	Qwen3-4B	1	22.07	14.41	80.15	15.56	18.37	15.62	18.47	18.36	21.12	15.28
		2	23.72	15.96	82.02	18.38	21.16	17.11	19.62	19.70	22.38	16.30
		4	26.49	18.38	84.09	24.64	27.45	19.63	22.18	23.24	24.87	18.97
		8	30.93	24.48	89.02	37.46	40.41	24.87	27.21	27.01	29.79	23.69
	Qwen3-8B	1	OOM	22.80	130.70	24.18	26.97	24.74	27.14	29.87	32.62	23.14
		2	OOM	24.57	130.36	29.14	32.04	26.33	28.89	31.61	35.04	24.72
		4	OOM	27.41	131.29	38.77	41.79	29.28	31.62	34.13	37.53	27.54
		8	OOM	34.57	140.42	58.59	61.77	34.22	36.98	39.29	42.09	32.03
RTX A6000	Qwen3-4B	1	16.34	11.38	15.29	12.91	16.11	12.73	16.23	15.61	19.09	12.23
		2	16.90	12.16	15.91	15.10	18.12	13.44	16.74	16.40	19.56	12.85
		4	18.60	14.20	17.26	19.42	22.39	14.89	18.15	17.89	21.04	14.17
		8	21.09	18.07	19.77	27.95	31.00	17.79	21.18	20.57	23.78	16.70
	Qwen3-8B	1	26.29	17.43	21.75	19.29	22.20	19.55	22.86	24.13	27.49	17.91
		2	27.37	18.44	22.61	22.89	25.81	20.39	23.69	25.19	28.34	18.62
		4	28.86	20.87	24.07	29.85	32.85	21.91	25.29	26.78	29.92	20.01
		8	31.53	25.80	27.15	43.47	46.66	24.93	28.50	29.56	32.86	22.60

Table 32: Complete generation latency in milliseconds (↓), including prefill of 128 prompt tokens and generation of 128 tokens.

GPU	**Model**	**Batch**	**FP16**	**CoBIT**	**QuaRot**$_{\text{GPTQ}}$	**SqueezeLLM**		**ParoQuant**				**ParoQuant**
			cuBLAS	Ours	CUTLASS	Official CUDA		AutoAWQ CUDA		AutoAWQ Triton		Marlin
			A16W16	A4W4	A4W4	A16W4	A4W4	A16W4	A4W4	A16W4	A4W4	A16W4
RTX 5060 Ti	Qwen3-4B	1	2866.9	1888.3	10357.9	2277.9	2658.5	2054.1	2460.9	2406.4	2801.4	2008.2
		2	3091.8	2111.2	10638.5	2900.9	3281.4	2267.9	2615.1	2611.2	2977.3	2144.3
		4	3484.7	2471.4	10961.8	4233.6	4620.9	2654.4	3011.8	3146.5	3373.5	2530.9
		8	4137.1	3372.2	11736.1	6947.4	7381.0	3400.3	3757.6	3657.6	4069.9	3223.5
	Qwen3-8B	1	OOM	2986.7	16877.6	3616.0	3995.7	3237.7	3571.6	3908.7	4283.8	3021.1
		2	OOM	3259.7	16880.8	4748.2	5147.7	3494.5	3850.5	4194.7	4668.9	3258.9
		4	OOM	3717.0	17146.1	6984.6	7411.9	3984.7	4324.0	4652.1	5131.5	3697.2
		8	OOM	4822.2	18574.5	11532.8	12053.8	4769.0	5229.6	5394.1	5854.6	4429.6
RTX A6000	Qwen3-4B	1	2113.7	1489.2	1978.4	1856.2	2271.5	1662.7	2119.1	2037.3	2490.9	1587.2
		2	2194.2	1609.9	2065.0	2331.2	2726.2	1777.0	2211.8	2171.7	2590.0	1678.0
		4	2436.4	1910.7	2255.4	3279.6	3681.3	2013.9	2459.6	2433.0	2861.8	1873.4
		8	2814.1	2493.0	2620.9	5177.8	5607.8	2414.5	2911.1	2764.2	3238.0	2255.0
	Qwen3-8B	1	3399.9	2291.0	2813.1	2847.9	3227.5	2553.1	2986.5	3155.7	3595.2	2324.6
		2	3567.7	2457.3	2942.6	3684.5	4070.9	2707.4	3149.6	3354.5	3776.1	2442.1
		4	3793.2	2839.5	3155.3	5320.2	5739.1	2981.8	3461.5	3673.4	4120.9	2661.4
		8	4223.7	3625.8	3614.1	8573.9	9058.4	3422.9	3975.1	4001.5	4520.6	3091.2