SAGE: PUSHING 2-BIT VECTOR-QUANTIZED LLMS TOWARD REAL-WORLD USABILITY

Anonymous authors
Paper under double-blind review

ABSTRACT

Extreme low-bit weight-only quantization is an effective approach to reducing the deployment costs of large language models. Below 3 bits, scalar quantization often suffers severe accuracy loss, while vector quantization (VQ) better preserves accuracy through more expressive joint encoding. However, existing VQ methods still exhibit substantial degradation on real-world tasks, and two design challenges remain underexplored: (1) protecting sensitive weight channels within jointly encoded vectors; and (2) allocating a fixed bit budget among representation components, such as direction and radius, to suit different weight matrices. To address these challenges, we propose **SAGE**: **S**plit **A**daptation and **G**rouping by **E**nergy for weight-only vector quantization. First, Energy-Stratified Grouping (ESG) couples channel grouping with activation-weighted fitting, grouping channels from different energy tiers within each subvector to prioritize sensitive channels. Second, Adaptive Bit Split (ABS) balances direction-codebook capacity and radius precision per matrix, selecting the split with the lowest activation-weighted reconstruction error under a fixed budget. Across benchmarks covering knowledge, mathematics, code generation, instruction following, and tool use, SAGE achieves state-of-the-art performance in the 2-bit quantization setting. On Qwen3-30B-A3B, it retains 95.3% of the BF16 model's average benchmark score. With our custom-designed CUDA kernel, SAGE delivers up to a $2.55\times$ non-streaming end-to-end speedup over FP16. We will release our code, models, and kernels.

1 INTRODUCTION

Recent large language models (LLMs) (Kimi Team, 2026; GLM-5 Team, 2026; Yang et al., 2025; DeepSeek Team, 2026; Liu et al., 2026) have made remarkable progress in reasoning, coding, and generation. Those gains have come with steadily growing parameter counts. A larger model occupies more memory and spends more compute on every token it generates, which makes it hard to run on edge devices and other resource-constrained platforms. Local deployment prioritizes low-latency inference with small batches. In this setting, decoding offers little weight reuse across requests, making memory bandwidth rather than compute the primary bottleneck (Song et al., 2024).

Weight-only post-training quantization (PTQ) (Frantar et al., 2023; Lin et al., 2024; Tseng et al., 2024b; Liang et al., 2026) targets exactly this bottleneck. Lower-bit weights shrink the memory footprint and can accelerate memory-bound decoding by reducing weight traffic (Kim et al., 2024). Scalar methods such as GPTQ (Frantar et al., 2023) and AWQ (Lin et al., 2024) have made 3–4-bit weights practical with little accuracy loss. Below 3 bits the picture changes: each weight has only a handful of reconstruction values left, and accuracy degrades sharply (Egiazarian et al., 2024). Vector quantization (VQ) (Egiazarian et al., 2024; Baalen et al., 2024; van der Ouderaa et al., 2026) instead encodes groups of weights jointly against a shared codebook, so at the same bit budget its reconstruction points follow the weight distribution far more closely. This is what motivates the sub-3-bit line of work built on VQ and related structured coding (Tseng et al., 2024a;b).

Existing VQ methods differ in how the codebook is structured and how a weight is recovered from it. QuIP# (Tseng et al., 2024a) and QTIP (Tseng et al., 2024b) use incoherence processing, requiring online Hadamard transforms on activations during inference. QuIP# combines table lookups with sign and shift operations, QTIP supports lookup-only, computed, and hybrid trellis codes, and LLVQ (van der Ouderaa et al., 2026) uses structured Leech lattice decoding without an explicit codebook. PCDVQ (Yue et al., 2025) does use an explicit direction codebook, but a substantially

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.

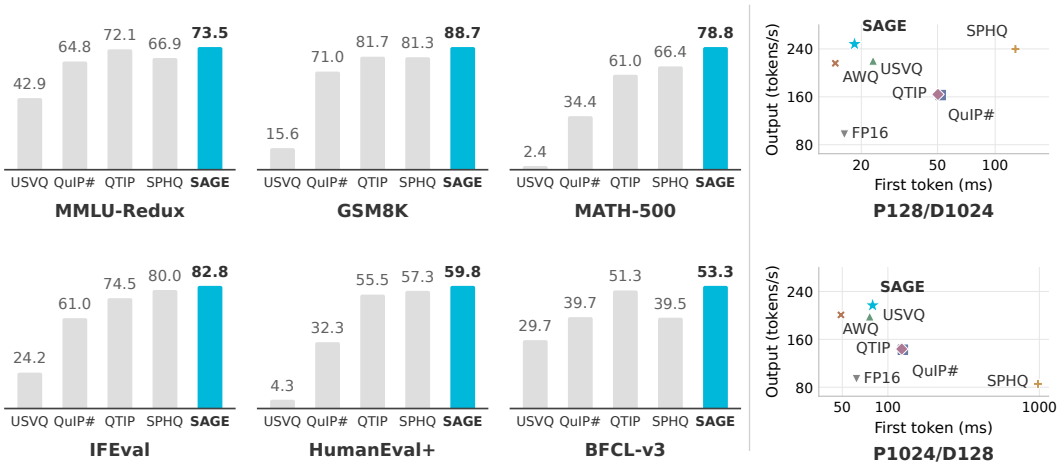


Figure 1: **Accuracy and inference efficiency on Qwen3-4B-Instruct-2507.** Left: generative scores, with independent zero-based axes. Right: batch-one non-streaming first-token latency versus output throughput on an RTX 4090; upper left is better. Throughput includes prompt processing. P/D denote input/output token counts. Quantized weights are reconstructed on demand.

larger one. SPHQuant (Zhang et al., 2026) reaches comparable accuracy and decodes considerably faster. It stores each group of eight weights, a subvector, in sign-separated form: the signs explicitly, the magnitude through a quantized radius, and the direction as an index into a learned codebook over the non-negative orthant. Folding all 2^8 orthants into one keeps that codebook at 64 entries, or 1 KiB, so it stays in shared memory and decoding is a lookup, a sign flip, and a multiply.

However, two challenges remain insufficiently addressed in vector quantization. **(1) How to protect important channels.** Protecting the channels that matter most is standard in scalar quantization, by rescaling them (Lin et al., 2024) or by permuting them so that a group's channels are alike and share a scale well (Yan et al., 2026). Neither strategy transfers directly to vector quantization, where eight channels share one codeword and one radius: both rely on properties of a per-weight scalar grid that a jointly coded subvector does not have. **(2) How to divide the fixed budget.** A group of eight weights costs 18 bits: eight signs, six for the direction index, four for the radius, a nominal 2.25 bits per weight. How the remaining ten are split between the index and the radius is itself a choice, since moving one bit from the radius to the index doubles the codebook at almost no extra cost. Previous work adopts a fixed split (Zhang et al., 2026).

Beyond quantization algorithm design, evaluating the practical capabilities of quantized models is equally important. Common metrics such as perplexity and multiple-choice accuracy on benchmarks like HellaSwag (Zellers et al., 2019) do not fully characterize the generative capabilities of post-trained LLMs. In particular, a small increase in perplexity does not necessarily imply that a model can still reason correctly, follow instructions, or produce functional code. We therefore evaluate quantized models on generative benchmarks covering knowledge, mathematical reasoning, code generation, instruction following, and tool use. Our goal is to preserve the ability of extremely low-bit models to complete real-world tasks, which requires assessing these capabilities directly.

In this paper, we propose **SAGE**: **S**plit **A**daptation and **G**rouping by **E**nergy for weight-only vector quantization. To address Challenge (1), we introduce Energy-Stratified Grouping (ESG), which couples channel grouping with activation-weighted reconstruction. ESG distributes energy-ranked channels across subvectors, placing channels from different energy tiers within each subvector so that weighted fitting can prioritize sensitive channels. To address Challenge (2), we introduce Adaptive Bit Split (ABS), which evaluates candidate allocations between the direction index and radius using activation-weighted reconstruction error and selects a split for each matrix. Together, ESG and ABS adapt the representation to channel importance and matrix-specific demands while preserving the encoding budget. Fig. 1 illustrates both accuracy and inference efficiency on Qwen3-4B-Instruct-2507. SAGE achieves the highest quantized scores across all six generative benchmarks and combines low first-token latency with high output throughput under both input/output length settings. Our custom-designed CUDA kernels further enable up to a $2.55\times$ non-streaming end-to-end speedup over FP16. We hope SAGE advances low-bit models toward real-world deployment.

Our key contributions can be summarized as follows:

- We propose **SAGE**: **S**plit **A**daptation and **G**rouping by **E**nergy, a novel weight-only VQ framework that advances 2-bit quantization toward real-world usability.
- We design **Energy-Stratified Grouping (ESG)**, which couples channel grouping with activation-weighted fitting. ESG groups channels from different energy tiers within each subvector, enabling weighted fitting to prioritize sensitive channels.
- We develop **Adaptive Bit Split (ABS)**, which derives the division between the direction index and the radius from activation-weighted reconstruction error and adapts it per matrix, rather than applying one fixed split across the whole model, at an unchanged bit budget.
- Experiments on five LLMs and six generative benchmarks establish **SAGE** as a new SOTA for 2-bit VQ, retaining 95.3% of the BF16 average score on Qwen3-30B-A3B. Our CUDA kernel achieves up to a $2.55\times$ non-streaming end-to-end speedup over FP16.

2 Related Works

Quantization can be categorized by optimization and representation. Post-training quantization (PTQ) uses small calibration sets for calibration or lightweight blockwise reconstruction, while quantization-aware training (QAT) typically involves large-scale end-to-end training. Scalar quantization (SQ) encodes weights individually, whereas vector quantization (VQ) encodes them jointly. Our method is weight-only vector PTQ, and we organize this section by representation.

Scalar Quantization. Among SQ approaches, PTQ is the most widely adopted, for its low calibration cost. At 8 bits, LLM.int8() (Dettmers et al., 2022) and SmoothQuant (Xiao et al., 2023) preserve accuracy via outlier handling and channel-wise scaling, while at 4 bits, GPTQ (Frantar et al., 2023) and AWQ (Lin et al., 2024) reduce quantization error through second-order compensation and activation-aware scaling. Beyond direct quantization, rotation-based methods redistribute outliers with fixed or learned orthogonal transforms (Ashkboos et al., 2024; Liu et al., 2025a; Chee et al., 2023; Sun et al., 2025; Hu et al., 2025; Liang et al., 2026), while permutation-based methods leave channels intact and merely reorder them, gathering channels of similar range so that a shared scale fits every member of a group (Yan et al., 2026). To limit error accumulation, GPTAQ (Li et al., 2025a) and QEP (Arai & Ichikawa, 2025) account for errors from preceding quantized layers when reconstructing layer outputs. Others add auxiliary representations, a sparse full-precision component (Kim et al., 2024) or shared low-rank corrections (Gu et al., 2026), or optimize the quantizer itself through clipping and equivalent transformations (Shao et al., 2024) and loss-sensitive non-uniform grids (Zhang & Shrivastava, 2025). At even lower precision, ARB-LLM (Li et al., 2025b) and PT^2-LLM (Yan et al., 2026) extend parameter refinement to binary and ternary quantization by alternating updates to grid parameters and discrete assignments. However, scalar PTQ can suffer substantial accuracy loss at 3 bits and below (Liu et al., 2025b). QAT addresses this through distillation (Liu et al., 2024b; Du et al., 2024) or tailored optimization (Chen et al., 2025; Liu et al., 2025b), while limited scalar-grid capacity motivates more expressive VQ representations.

Vector Quantization. VQ methods differ mainly in how they group weights, how they construct codebooks, and how they optimize the resulting assignments. GPTVQ (Baalen et al., 2024) extends GPTQ's second-order compensation to weight vectors spanning columns; VPTQ (Liu et al., 2024a) instead groups weights within columns for column-wise compensation. Additive and residual codebooks further refine reconstruction (Egiazarian et al., 2024; Liu et al., 2024a). QuIP# (Tseng et al., 2024a) combines incoherence processing with lattice codebooks. QTIP (Tseng et al., 2024b) retains incoherence processing but replaces these codebooks with trellis coding, enabling higher-dimensional quantization without exponential codebook storage. NestQuant (Savkin et al., 2025) develops the lattice approach using self-similar nested lattices. A further branch applies the classical shape–gain decomposition (Gersho & Gray, 1991), coding magnitude and direction separately: PCDVQ (Yue et al., 2025) pairs a scalar magnitude code with an E_8 direction codebook, LLVQ (van der Ouderaa et al., 2026) uses the Leech lattice, and SPHQuant (Zhang et al., 2026) additionally separates the sign pattern so that one small codebook entry serves every sign pattern. Optimization advances include joint refinement of parameters and assignments (Malinovskii et al., 2024) and mixed precision guided by gradients and Hessians for VLMs (Wang et al., 2026d). UniSVQ (Wang et al., 2026b) and LC-QAT (Wang et al., 2026a) use affine lattice codebooks for scalar kernels and lookup-free QAT, respectively.

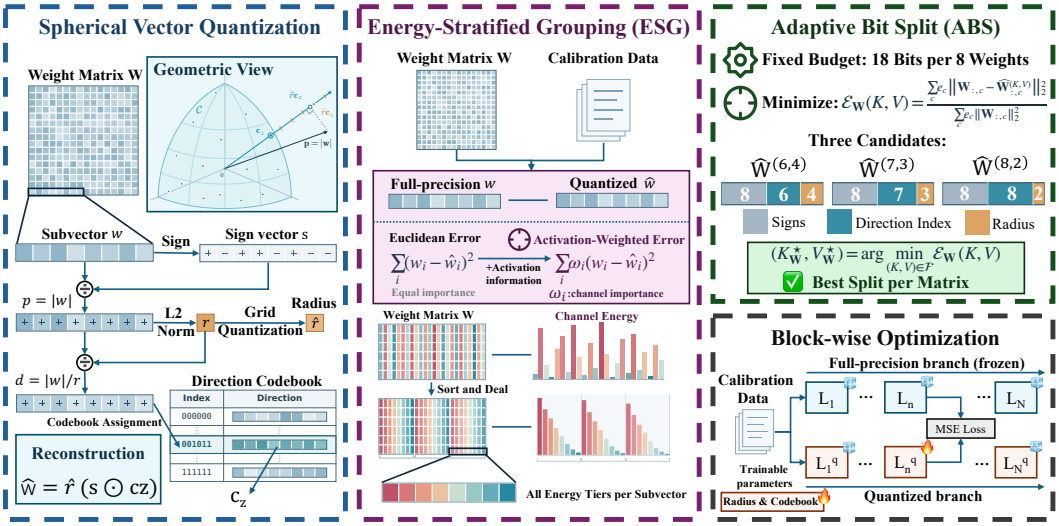


Figure 2: Overview of SAGE. **Spherical Vector Quantization** encodes each weight subvector using signs, a direction index, and a quantized radius. **Energy-Stratified Grouping (ESG)** uses calibration activations to group channels across energy tiers within subvectors and protect sensitive channels through weighted fitting. **Adaptive Bit Split (ABS)** compares activation-weighted reconstruction errors to select a direction-index/radius bit split per matrix, using 18 bits per eight weights. **Blockwise Optimization** updates radius and codebook angles to match full-precision block outputs.

3 METHOD

Overview. As illustrated in Fig. 2, SAGE combines activation-guided channel grouping with adaptive bit allocation for spherical vector quantization. We first introduce the spherical representation and its bit budget in Sec. 3.1. Building on this formulation, Sec. 3.2 presents ESG to protect sensitive channels through energy-stratified grouping and weighted fitting. Sec. 3.3 then introduces ABS to adapt the direction-index and radius bit allocation to each weight matrix. Finally, Sec. 3.4 covers calibration and blockwise optimization, and Sec. 3.5 presents our fused inference kernel.

3.1 BACKGROUND: SPHERICAL VECTOR QUANTIZATION

Spherical Representation. Following SPHQuant (Zhang et al., 2026), we partition each row of a weight matrix $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ into subvectors of dimension $v = 8$ along the input axis. Each subvector $\mathbf{w}$ is decomposed into signs $\mathbf{s} = \text{sign}(\mathbf{w})$, radius $r = \|\mathbf{w}\|_2$, and nonnegative unit direction $\mathbf{d} = |\mathbf{w}|/r$. Signs are stored exactly, while the direction is represented by an entry $\mathbf{c}_z$ from a matrix-specific codebook $\mathcal{C}$ of K nonnegative unit vectors. The quantized reconstruction is:

$$\hat{\mathbf{w}} = \hat{r}\,(\mathbf{s} \odot \mathbf{c}_z), \tag{1}$$

where $\hat{r}$ is the quantized radius and $\odot$ denotes elementwise multiplication.

Direction and Radius Quantization. In this formulation, the codebook is initialized using spherical k-means, and directions are selected by cosine similarity, $z = \arg\max_k \mathbf{d}^\top \mathbf{c}_k$. Given the selected direction, the continuous radius $\tilde{r} = |\mathbf{w}|^\top \mathbf{c}_z$ minimizes Euclidean reconstruction error. Every $G = 128$ weights share a uniform radius grid with offset μ_g and step δ_g:

$$\hat{r} = \mu_g + q\delta_g, \qquad q = \text{clip}\left(\left\lfloor \frac{\tilde{r} - \mu_g}{\delta_g} \right\rceil, 0, 2^V - 1\right), \tag{2}$$

where V is the radius bit width, and the grid spans each group's fitted radius range.

Bit Budget. Each subvector stores v sign bits, a $\log_2 K$-bit direction index, and a V-bit radius code. The total encoding budget is:

$$B = v + \log_2 K + V. \tag{3}$$

The baseline configuration $(K, V) = (64, 4)$ uses 18 bits per subvector, or 2.25 bits per weight excluding shared metadata. At this fixed budget, direction capacity and radius precision can be traded by varying K and V. Further details on spherical quantization are provided in Appendix A.6. We next introduce Energy-Stratified Grouping (ESG) and Adaptive Bit Split (ABS).

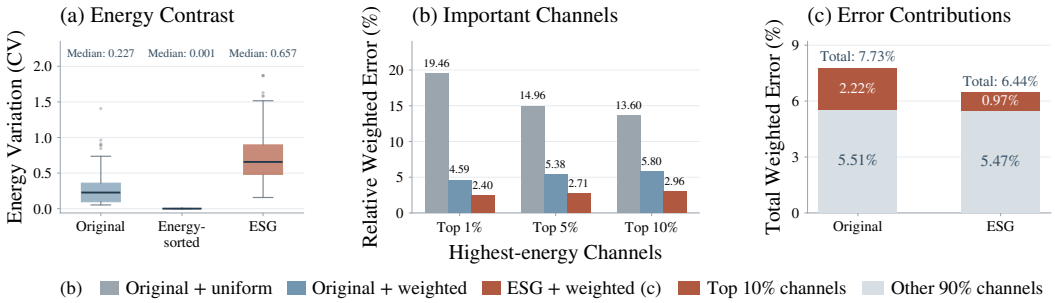


Figure 3: ESG analysis on Qwen3-4B-Instruct-2507: (a) within-subvector energy variation, (b) high-energy channel errors, and (c) total error contributions.

3.2 Energy-Stratified Grouping

Activation-Weighted Reconstruction. The Euclidean reconstruction criterion in Sec. 3.1 treats all positions within a subvector equally. Motivated by activation-aware channel protection (Lin et al., 2024) and data-aware codebook fitting (Baalen et al., 2024), we incorporate input activation statistics into the reconstruction objective to guide weight quantization. For the linear layer input $\mathbf{x}$, we estimate channel energy as $e_c = \mathbb{E}[x_c^2]$ over calibration tokens, where x_c is the activation of input channel c. For channels $a_1, \ldots, a_v$ in a subvector, we define the weights and objective as:

$$\omega_i = \frac{e_{a_i}^{\gamma}}{\frac{1}{v}\sum_{j=1}^{v} e_{a_j}^{\gamma}}, \qquad \mathcal{L}_{\boldsymbol{\omega}}(\mathbf{p}, \tilde{r}, \mathbf{c}_k) = \sum_{i=1}^{v} \omega_i (p_i - \tilde{r} c_{k,i})^2, \tag{4}$$

where $\mathbf{p} = |\mathbf{w}|$, $\tilde{r}$ is the continuous radius before quantization, and $c_{k,i}$ is the i-th component of codeword $\mathbf{c}_k$. Weights are normalized to unit mean per subvector. We use $\gamma = 0.75$ to temper energy differences while prioritizing high-energy channels, with ablations in Fig. 4(a).

Energy-Stratified Grouping. The effect of weighting depends on which channels share a subvector. When their energies are equal, $\omega_i = 1$ and the objective reduces to Euclidean reconstruction. This motivates grouping channels from different energy levels, allowing the weighted fit to prioritize important positions within each jointly encoded vector. Let σ sort input channels by decreasing energy, and divide the sorted list into v tiers of $m = d_{\text{in}}/v$ channels. ESG constructs each subvector by taking one channel from each tier, giving the permutation:

$$\pi\big((j-1)v + t\big) = \sigma\big((t-1)m + j\big), \qquad j = 1, \ldots, m, \quad t = 1, \ldots, v. \tag{5}$$

Each subvector therefore contains channels spanning the energy ranking, with positions consistently ordered from higher to lower energy. Fig. 3(a) shows that ESG increases within-subvector energy variation compared with both the original order and grouping adjacent energy-ranked channels. We apply π to the columns of $\mathbf{W}$ and reorder the activation statistics accordingly before fitting the quantizer. This preserves each channel's identity while changing which weights share a direction and radius. At inference, the corresponding permutation is applied to the input activations.

Weighted Quantizer Fitting. We fit the codebook using weighted spherical k-means in the permuted channel order. For a candidate codeword $\mathbf{c}_k$, minimizing Eq. (4) over the continuous radius gives a closed-form solution. Substituting it into the objective yields the direction assignment:

$$\tilde{r}_k^{\star} = \frac{\sum_i \omega_i p_i c_{k,i}}{\sum_i \omega_i c_{k,i}^2}, \qquad z = \arg\max_k \frac{\left(\sum_i \omega_i p_i c_{k,i}\right)^2}{\sum_i \omega_i c_{k,i}^2}. \tag{6}$$

The selected radius $\tilde{r}_z^{\star}$ is quantized using the shared grid in Eq. (2). With uniform weights, the assignment reduces to the cosine rule in Sec. 3.1. At initialization with $(K, V) = (64, 4)$, ESG reduces high-energy channel errors compared with weighted fitting without reordering (Fig. 3(b)). Most of the overall weighted-error reduction comes from the highest-energy 10% of channels (Fig. 3(c)), supporting the role of grouping in protecting sensitive channels.

3.3 Adaptive Bit Split

ESG changes the structure of weight subvectors and their reconstruction priorities, potentially altering the balance between direction and radius errors. Under a fixed encoding budget, increasing direction-codebook capacity requires reducing radius precision. Since the benefit of this trade-off depends on each matrix's weight distribution and activation statistics, Adaptive Bit Split (ABS) selects a matrix-specific split by comparing activation-weighted reconstruction errors at initialization.

Candidate Bit Splits. With $v = 8$ and $B = 18$, we consider three splits of the remaining ten bits:

$$\mathcal{F} = \{(64, 4), (128, 3), (256, 2)\}, \qquad \log_2 K + V = 10. \tag{7}$$

All candidates use the same 2.25-bit encoding budget per weight, with a fixed radius group size and grid-parameter precision. For each candidate, we fit the codebook and assign directions and continuous radii in the ESG-permuted order, following Sec. 3.2. Radii are quantized using Eq. (2), accounting for both direction approximation and radius discretization.

Activation-Aware Selection. Let $\widehat{\mathbf{W}}^{(K,V)}$ denote the reconstruction in original channel order, and $\mathbf{W}_{:,c}$ the weight column for input channel c. Using calibration energies e_c, we compute:

$$\mathcal{E}_{\mathbf{W}}(K, V) = \frac{\sum_c e_c \left\| \mathbf{W}_{:,c} - \widehat{\mathbf{W}}^{(K,V)}_{:,c} \right\|_2^2}{\sum_c e_c \left\| \mathbf{W}_{:,c} \right\|_2^2}. \tag{8}$$

The numerator approximates the expected squared output error by ignoring correlations between input channels. Fitting uses tempered, normalized weights $\boldsymbol{\omega}$, while scoring uses raw energies e_c to preserve relative importance across subvectors. The selected split is:

$$(K^\star_{\mathbf{W}}, V^\star_{\mathbf{W}}) = \arg \min_{(K,V)\in\mathcal{F}} \mathcal{E}_{\mathbf{W}}(K, V). \tag{9}$$

Equal candidate payloads allow independent selection for each matrix while preserving the total encoding budget. The split is fixed after initialization throughout blockwise optimization.

3.4 Calibration Data and Blockwise Optimization

Calibration Data Construction. Since ESG and ABS rely on calibration activations, the calibration inputs should reflect the contexts encountered during deployment, including both user prompts and model-generated responses. Following the use of model-generated data in LLM-QAT (Liu et al., 2024b) and target-model reasoning traces in AYOT (Wang et al., 2026c), we construct calibration sequences from the full-precision model's own responses. We sample prompts spanning mathematics, code, reasoning, instruction following, tool use, and general conversation, apply the target model's chat template, and generate responses using greedy decoding. Each prompt is concatenated with its response, and the resulting token streams are packed into fixed-length sequences for activation statistics collection and blockwise optimization. Further details are provided in Appendix A.5.

Blockwise Optimization. Following SPHQuant, we optimize blocks sequentially against full-precision outputs, with student inputs from preceding quantized blocks. For block ℓ, we minimize:

$$\mathcal{L}_{\text{block}} = \mathbb{E}_{\mathcal{D}_{\text{cal}}} \left[\left\| \widehat{F}_\ell(\mathbf{x}^{\text{q}}_\ell) - F_\ell(\mathbf{x}^{\text{fp}}_\ell) \right\|_2^2 \right], \tag{10}$$

where F_ℓ and $\widehat{F}_\ell$ denote the full-precision and quantized blocks, respectively. We jointly update the continuous radius parameters and codebook angles while keeping channel permutations, bit splits, and direction indices fixed. Radius quantization uses a straight-through estimator, and angular parameterization preserves nonnegative unit codewords. This procedure allows each block to compensate for errors introduced by preceding quantized blocks.

3.5 CUDA Kernel Design for Efficient Inference

Channel Permutation and Fusion. ESG stores each weight matrix in permuted input-channel order; each projection applies its corresponding permutation to the input activations. In compiled decoding, we keep permutation visible to the graph compiler for fusion with preceding normalization or activation operations. This avoids repeating the gather in every output thread block. Eager decoding fuses gather into matrix–vector multiplication for short inputs, avoiding a separate launch. Prefill reorders activations once per projection for reuse across output tiles.

Quantized Matrix Multiplication with Adaptive Bit Splits. All ABS splits share a 16-bit main word plus two auxiliary bits per subvector, with compile-time specialization of bit extraction. During decoding, contiguous shared-memory codewords enable 128-bit loads for register-local packed FP16 dot products, followed by FP32 radius weighting and reduction. Shape-dependent output tiling balances activation reuse against register pressure and thread-block parallelism. During prefill, Tensor Cores reuse weight tiles decoded into shared memory across 32 or 64 activation rows. The larger tile amortizes decoding, while staging only 64 activation channels at a time limits shared-memory usage. For small grids, we expose more parallelism by splitting the reduction dimension across thread blocks, then combine partial sums in FP32. Further details are provided in Appendix A.1.

Table 1: Weight-only vector quantization results on six benchmarks (higher is better). **Bold** indicates the best quantized result for each model and metric. BF16 denotes the full-precision model.

Method	BPW	MMLU-R*	GSM8K	MATH-500	IFEval	HumanE+*	BFCL-v3	Average
Qwen3-4B-Instruct-2507								
BF16	16.00	81.28	94.01	93.80	84.66	82.32	59.94	82.67
QuIP#	2.00	64.82	70.96	34.40	61.00	32.32	39.74	50.54
QTIP	2.00	72.11	81.73	61.00	74.49	55.49	51.33	66.03
UniSVQ	2.00	42.89	15.62	2.40	24.21	4.27	29.73	19.85
SPHQuant	2.25	66.93	81.35	66.40	80.04	57.32	39.48	65.25
SAGE	2.25	**73.47**	**88.70**	**78.80**	**82.81**	**59.76**	**53.33**	**72.81**
Qwen3-30B-A3B-Instruct-2507								
BF16	16.00	88.81	96.59	96.80	83.92	89.63	63.31	86.51
QuIP#	2.00	85.35	93.18	90.80	77.63	81.10	51.99	80.01
QTIP	2.00	**86.65**	94.16	91.40	80.22	84.76	**56.65**	82.31
UniSVQ	2.00	77.42	83.85	66.00	60.63	62.80	36.18	64.48
SPHQuant	2.25	83.18	93.18	88.40	80.41	75.61	49.79	78.43
SAGE	2.25	85.86	**94.31**	**93.60**	**80.96**	**85.37**	54.77	**82.48**
Qwen3.5-4B-Thinking								
BF16	16.00	89.19	96.89	93.20	87.62	83.54	64.19	85.77
QuIP#	2.00	36.98	40.18	15.80	27.17	13.41	24.13	26.28
QTIP	2.00	68.60	76.80	25.80	49.17	28.66	41.93	48.49
UniSVQ	2.00	7.53	0.68	0.00	1.48	0.00	28.85	6.42
SPHQuant	2.25	68.04	70.66	32.40	59.89	39.02	38.60	51.44
SAGE	2.25	**78.04**	**88.93**	**53.60**	**71.53**	**65.85**	**48.12**	**67.68**
Ministral-3-8B-Instruct-2512								
BF16	16.00	82.14	94.47	87.40	62.29	76.22	51.97	75.75
QuIP#	2.00	67.88	73.16	40.40	42.51	43.29	30.19	49.57
QTIP	2.00	**72.05**	78.32	50.00	47.50	56.71	29.81	55.73
UniSVQ	2.00	50.68	28.05	6.00	29.39	10.37	29.73	25.70
SPHQuant	2.25	63.65	67.55	44.80	48.43	50.00	35.01	51.57
SAGE	2.25	**72.05**	**84.23**	**63.40**	**53.42**	**57.32**	**48.85**	**63.21**
Ministral-3-14B-Instruct-2512								
BF16	16.00	84.82	95.75	90.20	64.88	81.10	56.50	78.88
QuIP#	2.00	73.65	85.97	55.60	52.68	57.32	39.00	60.70
QTIP	2.00	70.74	83.09	46.80	50.28	30.49	29.71	51.85
UniSVQ	2.00	63.30	64.37	27.20	46.40	26.83	28.91	42.84
SPHQuant	2.25	72.33	82.41	62.00	**63.03**	65.85	47.81	65.57
SAGE	2.25	**78.81**	**90.14**	**77.40**	62.48	**68.90**	**54.03**	**71.96**

*MMLU-R denotes MMLU-Redux, and HumanE+ denotes HumanEval+.

4 Experiments

4.1 Experimental Settings

Implementation Details. Accuracy experiments use PyTorch (Paszke et al., 2019) and HuggingFace Transformers (Wolf et al., 2020) on NVIDIA RTX PRO 6000 Blackwell GPUs; the inference-efficiency experiments in Sec. 4.4 use an RTX 4090. We perform blockwise optimization by minimizing the mean squared error between quantized and full-precision block outputs. The radius and codebook parameters are jointly optimized using Adam for 20 epochs, with a batch size of 16 and a constant learning rate of 10^{-3}. The calibration corpus contains 1920 training sequences and 128 validation sequences, each with 2048 tokens. We use a subvector dimension of $v = 8$ and a radius group size of 128 weights. We initialize codebooks with 20 iterations of activation-weighted spherical k-means on 65,536 sampled subvectors. By default, ESG uses $\gamma = 0.75$, and ABS selects $(K, V) \in \{(64, 4), (128, 3), (256, 2)\}$ per matrix, unless otherwise stated.

Table 2: ESG and ABS ablation on Qwen3-4B-Instruct-2507. Best results are in **bold**.

ESG	ABS	MMLU-R*	GSM8K	MATH-500	IFEval	HumanE+*	BFCL-v3	Average
✗	✗	66.93	81.35	66.40	80.04	57.32	39.48	65.25
✓	✗	69.05	82.18	68.40	80.04	54.88	45.17	66.62
✗	✓	71.33	84.84	74.40	79.85	58.54	44.02	68.83
✓	✓	**73.47**	**88.70**	**78.80**	**82.81**	**59.76**	**53.33**	**72.81**

*MMLU-R denotes MMLU-Redux, and HumanE+ denotes HumanEval+.

Models and Evaluation. We evaluate five models covering dense, mixture-of-experts (MoE), and hybrid attention architectures, as well as instruct and thinking modes. These include Qwen3-4B-Instruct-2507 (Yang et al., 2025), Qwen3-30B-A3B-Instruct-2507 (MoE) (Yang et al., 2025), Ministral-3-8B-Instruct-2512 (Liu et al., 2026), and Ministral-3-14B-Instruct-2512 (Liu et al., 2026) in instruct mode, and Qwen3.5-4B (Qwen Team, 2026), which combines linear and full attention, in thinking mode. For multimodal checkpoints, we quantize the language backbone. We assess practical capabilities on six benchmarks: MMLU-Redux (Gema et al., 2025) for general knowledge; GSM8K (Cobbe et al., 2021) and MATH-500 (Lightman et al., 2024) for mathematical reasoning; IFEval (Zhou et al., 2023) for instruction following; HumanEval+ (Liu et al., 2023) for code generation; and BFCL-v3 (Patil et al., 2025) for tool use. We report accuracy for MMLU-Redux, GSM8K, and MATH-500, prompt-level strict accuracy for IFEval, pass@1 for HumanEval+, and the overall score for BFCL-v3. See Appendix for quantization and evaluation details.

Baselines. We compare SAGE with four 2-bit weight-only VQ baselines: SPHQuant (Zhang et al., 2026), using spherical quantization; QTIP (Tseng et al., 2024b), using trellis coding with incoherence processing; QuIP# (Tseng et al., 2024a), using lattice codebooks with Hadamard incoherence processing; and UniSVQ (Wang et al., 2026b), unifying scalar and vector quantization through affine lattice codebooks. Speed benchmarks additionally include AWQ (Lin et al., 2024) at W4G128.

4.2 Main Results

Comparisons with State-of-the-Art Methods. As shown in Table 1, SAGE achieves the highest average score on all five models, retaining 78.9–95.3% of their BF16 average scores. At the same 2.25-bit encoding budget, it improves over SPHQuant by 4.05–16.24 percentage points. The largest gain occurs on Qwen3.5-4B-Thinking, where SAGE raises the average score from 51.44 to 67.68 and leads all quantized baselines on every benchmark. On the three dense instruct models, SAGE surpasses the strongest baseline for each model by 6.39–7.48 points. On Qwen3-30B-A3B, it achieves an average score of 82.48, retaining 95.3% of BF16 performance. These results demonstrate gains across dense, MoE, and hybrid attention models in instruct and thinking modes.

Task Sensitivity to Quantization. Quantization affects tasks unevenly, with code generation and challenging mathematical reasoning showing substantial losses. For example, SAGE trails BF16 by 22.56 points on HumanEval+ but only 1.85 points on IFEval for Qwen3-4B, indicating that strong instruction following can coexist with substantial degradation in code generation. Sensitivity also varies within mathematical reasoning. On Qwen3.5-4B-Thinking, the gap to BF16 reaches 39.60 points on MATH-500, compared with 7.96 points on GSM8K. This contrast highlights the difficulty of preserving demanding reasoning in a thinking model, even when performance on simpler mathematical tasks remains relatively strong. Across five models, SAGE gains 2.44–26.83 points on HumanEval+ and 5.20–21.20 on MATH-500 over SPHQuant. Remaining gaps to BF16 motivate task-level evaluation beyond average scores.

4.3 Ablation Studies

Contributions of ESG and ABS. Table 2 evaluates ESG and ABS on Qwen3-4B-Instruct-2507. ESG alone improves the average score by 1.37 points, from 65.25 to 66.62, while ABS alone yields a 3.58-point gain, reaching 68.83. Full SAGE achieves 72.81, improving over the baseline by 7.56 points and outperforming the individual-component configurations by 6.19 and 3.98 points, respectively. It also achieves the best score across all six benchmarks, indicating broad improvements in capability preservation. The larger gain of the full configuration suggests that grouping and allocation work complementarily. ESG changes the structure and reconstruction priorities of subvectors, while ABS adapts each matrix's direction–radius split to this structure. By providing a more suitable allocation of representational capacity at initialization, ABS can help realize the benefits of ESG.

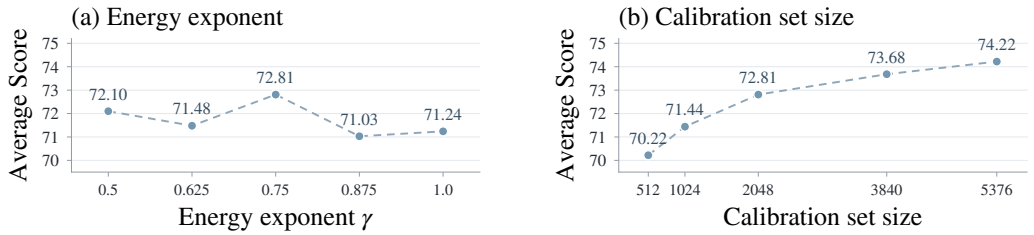
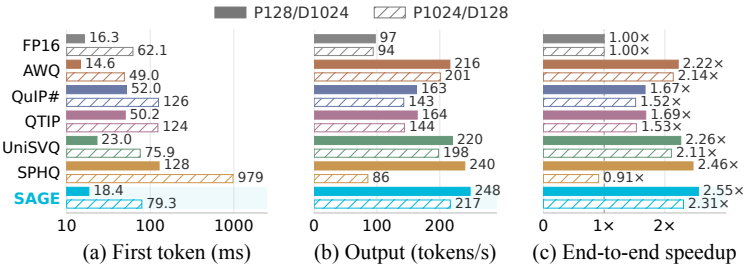


Figure 4: Sensitivity to (a) the energy exponent γ and (b) calibration set size on Qwen3-4B-Instruct-2507. Scores are averaged over six benchmarks covering diverse generative capabilities.

Sensitivity to the Energy Exponent. Fig. 4(a) evaluates γ from 0.5 to 1.0 on Qwen3-4B-Instruct-2507. Average scores range from 71.03 to 72.81, with $\gamma = 0.75$ achieving the highest score. All tested settings outperform the baseline score of 65.25, suggesting that the benefits are maintained across a range of weighting strengths. The exponent controls how strongly activation-energy differences influence reconstruction: smaller values produce more uniform weights, while larger values emphasize high-energy channels. We default to $\gamma = 0.75$, which achieves the highest average score.

Effect of Calibration Set Size. As shown in Fig. 4(b), increasing the calibration set size from 512 to 5376 sequences improves SAGE's average score from 70.22 to 74.22. The largest setting outperforms our default by 1.41 percentage points, suggesting that additional calibration data can further improve capability preservation. Given the computational cost of larger calibration sets, we use the same calibration set size as SPHQuant in main experiments to match the data budget.



Method	4B	30B-A3B
FP16	7.49	56.87
QuIP#	$1.58_{4.76\times\downarrow}$	$8.24_{6.90\times\downarrow}$
QTIP	$1.58_{4.75\times\downarrow}$	$8.32_{6.84\times\downarrow}$
UniSVQ	$1.57_{4.76\times\downarrow}$	$8.24_{6.90\times\downarrow}$
SPHQ	$1.78_{4.20\times\downarrow}$	$9.90_{5.74\times\downarrow}$
SAGE	$1.79_{4.19\times\downarrow}$	$10.02_{5.68\times\downarrow}$

(d) Packed model size (GiB)

Figure 5: **Inference efficiency.** Qwen3-4B-Instruct-2507, RTX 4090 (non-streaming, batch 1). P/D: input/output tokens. Output throughput includes prefill; weights are reconstructed on demand. Packed model sizes include unquantized weights; AWQ uses W4G128 for speed benchmarks.

4.4 Inference Efficiency

End-to-End Inference Speed. We benchmark non-streaming, batch-one greedy generation with packed weights and FP16 activations and KV cache. As shown in Fig. 5, SAGE achieves a 2.31–2.55× end-to-end speedup over dense FP16, the highest throughput in both workloads, and 85.5–91.9% lower first-token latency than SPHQuant. Detailed protocols and the tradeoff between latency and memory usage when retaining reconstructed weights are provided in Appendix A.2.

Memory and Storage Costs. Following prior work (Zhang et al., 2026; Tseng et al., 2024b; Wang et al., 2026b), we quantize all linear layers except the LM head, leaving embeddings and LayerNorm parameters unquantized. SAGE peaks at 2.19 GiB versus 7.94 GiB for FP16 (a 3.63× reduction). The encoding payload uses 2.25 bits per weight, with codebook and permutation metadata stored separately. Fig. 5(d) reports packed model sizes, including unquantized parameters.

5 Conclusion

In this work, we presented SAGE, a weight-only vector quantization method that uses activation information to guide channel grouping and bit allocation. These two decisions are closely connected: grouping determines which weights share a representation, while bit allocation determines its capacity to encode direction and magnitude. By addressing them together, SAGE makes more effective use of a limited encoding budget. Experiments across diverse generative benchmarks demonstrate improved capability preservation, while our custom CUDA kernel translates the compact representation into practical inference acceleration. We hope this work helps move extreme low-bit quantization beyond evaluations centered on perplexity toward models that reliably solve real-world tasks.

AI USE STATEMENT

We used generative AI tools to assist with code development and improve the manuscript's language. We also used full-precision target models to generate responses for calibration data. The authors prepared the key tables, figures, and conclusions independently. All AI-supported text and code were reviewed and verified by the authors, who take full responsibility for the final manuscript and associated artifacts.

ETHICS STATEMENT

This work studies weight-only quantization for efficient deployment of large language models. Quantized models may retain the biases and potential for misuse of their pretrained counterparts, while lower deployment costs may broaden their accessibility. Our evaluations focus on capability preservation and inference efficiency and do not establish the safety of the quantized models.

REPRODUCIBILITY STATEMENT

To support independent replication, we will make our code and model weights publicly available. The appendix documents the experimental settings and implementation details underlying our results.

REFERENCES

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. Opencodeinstruct: A large-scale instruction tuning dataset for code llms. *arXiv preprint arXiv:2504.04030*, 2025.

Yamato Arai and Yuma Ichikawa. Quantization error propagation: Revisiting layer-wise post-training quantization. In *NeurIPS*, 2025.

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. In *NeurIPS*, 2024.

Mart Van Baalen, Andrey Kuzmin, Markus Nagel, Peter Couperus, Artem Bolshakov, Cedric Bastoul, Eric Mahurin, Tijmen Blankevoort, and Paul Whatmough. GPTVQ: The blessing of dimensionality for LLM quantization. In *Workshop on Efficient Systems for Foundation Models II @ ICML2024*, 2024.

Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. Quip: 2-bit quantization of large language models with guarantees. In *NeurIPS*, 2023.

Mengzhao Chen, Wenqi Shao, Peng Xu, Jiahao Wang, Peng Gao, Kaipeng Zhang, and Ping Luo. Efficientqat: Efficient quantization-aware training for large language models. In *ACL (1)*, 2025.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

DeepSeek Team. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. In *NeurIPS*, 2022.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations. In *EMNLP*, 2023.

Dayou Du, Yijia Zhang, Shijie Cao, Jiaqi Guo, Ting Cao, Xiaowen Chu, and Ningyi Xu. Bitdistiller: Unleashing the potential of sub-4-bit llms via self-distillation. In *ACL (1)*, 2024.

Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. Extreme compression of large language models via additive quantization. In *ICML*, 2024.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: Accurate quantization for generative pre-trained transformers. In *ICLR*, 2023.

Aryo Pradipta Gema, Joshua Ong Jun Leang, Giwon Hong, Alessio Devoto, Alberto Carlo Maria Mancino, Rohit Saxena, Xuanli He, Yu Zhao, Xiaotang Du, Mohammad Reza Ghasemi Madani, Claire Barale, Robert McHardy, Joshua Harris, Jean Kaddour, Emile van Krieken, and Pasquale Minervini. Are we done with mmlu? In *NAACL (Long Papers)*, 2025.

Allen Gersho and Robert M Gray. *Vector quantization and signal compression*. Springer New York, NY, 1991.

GLM-5 Team. GLM-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Hongyaoxing Gu, Xinzhe Chen, Lijuan Hu, and Liu fangfang. Tileq: Efficient low-rank quantization of mixture-of-experts with 2d tiling. In *ICML*, 2026.

Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. Deepmath-103k: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. In *ICLR*, 2026.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. *Thirty-fifth Conference on NeurIPS Datasets and Benchmarks Track (Round 2)*, 2021.

Xing Hu, Yuan Cheng, Dawei Yang, Zhixuan Chen, Zukang Xu, Jiangyong Yu, Chen Xu, Zhihang Yuan, Zhe Jiang, and Sifan Zhou. Ostquant: Refining large language model quantization with orthogonal and scaling transformations for better distribution fitting. In *ICLR*, 2025.

Sehoon Kim, Coleman Richard Charles Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. SqueezeLLM: Dense-and-sparse quantization. In *ICML*, 2024.

Kimi Team. Kimi K3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *SOSP*, 2023.

Yuhang Li, Ruokai Yin, Donghyun Lee, Shiting Xiao, and Priyadarshini Panda. GPTAQ: Efficient finetuning-free quantization for asymmetric calibration. In *ICML*, 2025a.

Zhiteng Li, Xianglong Yan, Tianao Zhang, Haotong Qin, Dong Xie, Jiang Tian, zhongchao shi, Linghe Kong, Yulun Zhang, and Xiaokang Yang. ARB-LLM: Alternating refined binarizations for large language models. In *ICLR*, 2025b.

Yesheng Liang, Haisheng Chen, Song Han, and Zhijian Liu. Paroquant: Pairwise rotation quantization for efficient reasoning LLM inference. In *ICLR*, 2026.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *ICLR*, 2024.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. In *MLSys*, 2024.

Alexander H Liu, Kartik Khandelwal, Sandeep Subramanian, Victor Jouault, Abhinav Rastogi, Adrien Sadé, Alan Jeffares, Albert Jiang, Alexandre Cahill, Alexandre Gavaudan, et al. Ministral 3. *arXiv preprint arXiv:2601.08584*, 2026.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *NeurIPS*, 2023.

Yifei Liu, Jicheng Wen, Yang Wang, Shengyu Ye, Li Lyna Zhang, Ting Cao, Cheng Li, and Mao Yang. Vptq: Extreme low-bit vector post-training quantization for large language models. In *EMNLP*, 2024a.

Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. LLM-QAT: Data-free quantization aware training for large language models. In *ACL (Findings)*, 2024b.

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. In *ICLR*, 2025a.

Zechun Liu, Changsheng Zhao, Hanxian Huang, Sijia Chen, Jing Zhang, Jiawei Zhao, Scott Roy, Lisa Jin, Yunyang Xiong, Yangyang Shi, Lin Xiao, Yuandong Tian, Bilge Soran, Raghuraman Krishnamoorthi, Tijmen Blankevoort, and Vikas Chandra. Paretoq: Improving scaling laws in extremely low-bit LLM quantization. In *NeurIPS*, 2025b.

Vladimir Malinovskii, Denis Mazur, Ivan Ilin, Denis Kuznedelev, Konstantin Pavlovich Burlachenko, Kai Yi, Dan Alistarh, and Peter Richtárik. PV-tuning: Beyond straight-through estimation for extreme LLM compression. In *NeurIPS*, 2024.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *ICML*, 2025.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

Hayley Ross, Ameya Sunil Mahabaleshwarkar, and Yoshi Suhara. When2call: When (not) to call tools. In *NAACL*, 2025.

Semyon Savkin, Eitan Porat, Or Ordentlich, and Yury Polyanskiy. Nestquant: nested lattice quantization for matrix products and LLMs. In *ICML*, 2025.

Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. In *ICLR*, 2024.

Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *SOSP*, 2024.

Yuxuan Sun, Ruikang Liu, Haoli Bai, Han Bao, Kang Zhao, Yuening Li, JiaxinHu, Xianzhi Yu, Lu Hou, Chun Yuan, Xin Jiang, Wulong Liu, and Jun Yao. Flatquant: Flatness matters for LLM quantization. In *ICML*, 2025.

Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. QuIP$\#$: Even better LLM quantization with hadamard incoherence and lattice codebooks. *ICML*, 2024a.

Albert Tseng, Qingyao Sun, David Hou, and Christopher De Sa. QTIP: Quantization with trellises and incoherence processing. In *NeurIPS*, 2024b.

Tycho F. A. van der Ouderaa, Mart Van Baalen, Paul N. Whatmough, and Markus Nagel. Leech lattice vector quantization for efficient LLM compression. In *AdaptFM: Resource-Adaptive Foundation Model Inference*, 2026.

Haoyu Wang, Xingyu Yu, Haiyan Zhao, Fengxiang Wang, and Xu Han. LC-QAT: Data-efficient 2-bit QAT for LLMs via linear-constrained vector quantization. In *ICML*, 2026a.

Haoyu Wang, Haiyan Zhao, Xingyu Yu, Zhangyang Yao, Xu Han, Zhiyuan Liu, and Maosong Sun. UniSVQ: 2-bit unified scalar-vector quantization. In *ICML*, 2026b.

Shigeng Wang, Chao Li, Yangyuxuan Kang, Jiawei Fan, and Anbang Yao. Attend to your own thoughts: Breaking the barrier for post-training quantization of reasoning llms through the lens of 1.58-bit quantization. *arXiv preprint arXiv:2608.01078*, 2026c.

Zhong Wang, Zukang Xu, Xing Hu, and Dawei Yang. Mgvq: Synergizing multi-dimensional sensitivity-aware and gradient-hessian fusion for vector quantization. *arXiv preprint arXiv:2605.24019*, 2026d.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *EMNLP (Demos)*, 2020.

Guangxuan Xiao, Ji Lin, Mickaël Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *ICML*, 2023.

Xianglong Yan, ChengZhu Bao, Zhiteng Li, Tianao Zhang, Kaicheng Yang, Haotong Qin, Ruobing Xie, Xingwu Sun, and Yulun Zhang. PT$ˆ2$-LLM: Post-training ternarization for large language models. In *ICLR*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yuxuan Yue, Zukang Xu, Zhihang Yuan, Dawei Yang, Jianlong Wu, and Liqiang Nie. Pcdvq: Enhancing vector quantization for large language models via polar coordinate decoupling. *arXiv preprint arXiv:2506.05432*, 2025.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *ACL (1)*, 2019.

Kewei Zhang, Zheng Chen, Haotong Qin, and Yulun Zhang. SPHQuant: Efficient extreme low bit weight quantization for vision-language models. *NeurIPS*, 2026.

Tianyi Zhang and Anshumali Shrivastava. Leanquant: Accurate and scalable large language model quantization with loss-error-aware grid. In *ICLR*, 2025.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*, 2023.

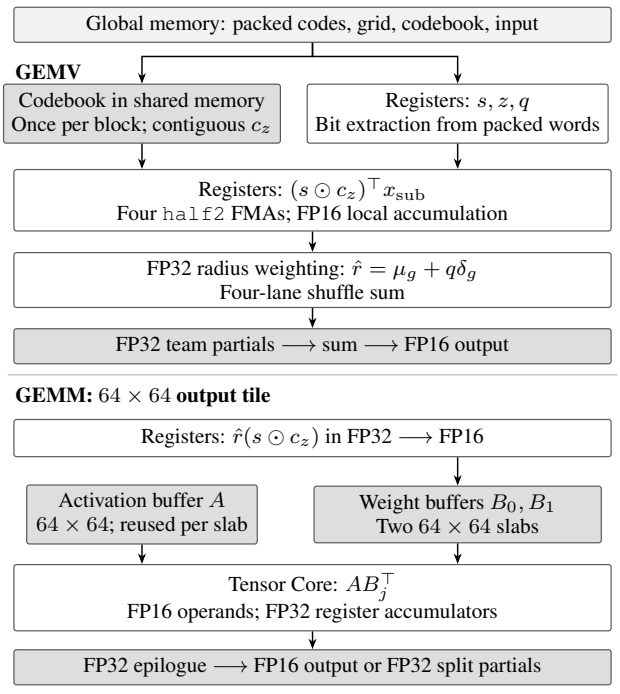
A Appendix

A.1 CUDA Kernel Implementation Details

GEMV. During decoding, each eight-weight subvector contributes a scalar to one output. Its radius is shared by all eight weights, so we apply that radius after taking the dot product with the signed direction (Fig. 6a). This replaces eight radius multiplications with one. In the shape-specialized kernels, we place each codeword's eight components consecutively in shared memory, allowing a thread to fetch the whole direction in one 128-bit load. We divide each 128-channel group among four neighboring threads. Each thread processes four subvectors, while other four-thread teams cover the remaining groups. Assigning multiple outputs to a thread lets it reuse the same activations; using fewer outputs per block instead increases the number of independent blocks.

GEMM. During prefill, many token rows use the same weights, so we reconstruct each weight once per output tile and reuse it across those rows. The schedule in Fig. 6b processes each 128-channel group in two halves. We decode both weight halves together from the same packed group and retain them until every warp finishes the group. Only the activation half currently being multiplied occupies shared memory, leaving room for other resident thread blocks. We fetch the next half's activations into registers before the current matrix multiply. A barrier prevents us from overwriting the shared activation buffer until every warp has finished reading it; another makes the replacement visible before the next multiply. Likewise, we prefetch the next group's packed weights while the current group is still in use, but wait to decode them into shared memory until both current weight halves have been consumed.

(a) Data paths within a thread block



(b) GEMM slab schedule

g starts at this block's first group. The FP32 accumulator `acc` starts at zero; the codebook is already in shared memory.

```
stage_A(load_A(g,0))
stage_B(decode(load_Q(g)))
sync()
for g in assigned_groups:
  a1 = load_A(g,1)
  mma(A,B[0],acc)
  sync(); stage_A(a1)
  if has_next(g):
    a0 = load_A(g+1,0)
    q1 = load_Q(g+1)
  sync(); mma(A,B[1],acc)
  sync()
  if has_next(g):
    stage_A(a0)
    stage_B(decode(q1))
  sync()
```

`load_A/Q` reads activations or codes and grid into registers. `stage_A/B` writes shared memory. `decode` reconstructs both weight halves; `sync` waits for all threads.

Figure 6: Kernel data paths and the 64-row GEMM loop. x_{sub} contains eight reordered activations. In (b), $j \in \{0, 1\}$ selects a half-group, and `mma` adds $AB_j^\top$ to `acc`.

A.2 Inference Measurement Details

Hardware and preparation. We benchmark Qwen3-4B-Instruct-2507 on one RTX 4090 with 14 AMD EPYC 9354 CPU cores and 60.1 GB RAM, using Python 3.11, PyTorch 2.10.0 (cu128), CUDA 12.8, cuDNN 9.10.2, and Transformers 5.15.0. Unlike accuracy checkpoints, timing check-

points use synthetic packed weights, SAGE permutations, and bit splits; only decoder linear layers are replaced. Batch size is one; activations and the static KV cache are FP16, with cache capacity $P + D - 1$ for P input and D output tokens. UniSVQ uses BF16 internally, including boundary casts in timing. SDPA priority is cuDNN, FlashAttention, memory-efficient attention, then math, during both compilation and execution; an untimed profiler checks actual kernel selection. Prefill and decode use `torch.compile` with `fullgraph=True`, `dynamic=False`, and `reduce-overhead`: two compilation/capture passes, profiling, one recovery pass, five warmups, then ten measurements with recompilation forbidden. Loading, input preparation, compilation, profiling, and per-window setup (cache reset, pinned buffers, events, copy stream, observer, and device synchronization) are untimed.

End-to-end timing. Figure 7 defines the timing intervals for fixed-length greedy generation without early stopping. Argmax and token feedback stay on the main CUDA stream, with no per-token host waits. TTFT uses GPU events; total latency and TPOT use the CPU wall clock, so TPOT includes observer scheduling effects and final delivery rather than measuring isolated decode kernels. Each latency metric reports its own median over ten runs; speedup is the ratio of FP16's median total latency to the method's. Output throughput is total generated tokens divided by total wall time across runs, including prompt processing.

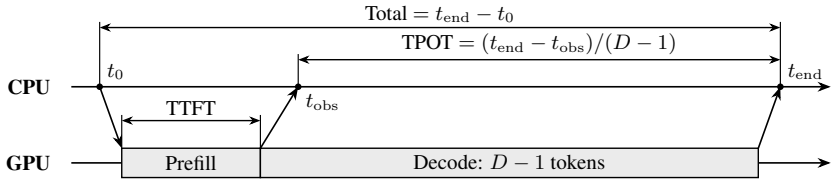


Figure 7: Inference timing for $D > 1$ output tokens. t_{obs} marks CPU observation of the first token.

Memory accounting. All plots reconstruct quantized weights on demand, without retaining additional full-precision copies between requests; online activation transforms remain timed. For FP16, AWQ, SPHQuant, and SAGE, the weight-retention switch has no effect; the figures consistently use runs with that switch disabled. Peak memory is `torch.cuda.max_memory_allocated`, reset before model initialization and read after measurement, so compilation and warmup peaks are included. It excludes CUDA-context and other allocations outside PyTorch's allocator: exceeding a device's capacity precludes deployment, but falling below it does not guarantee a fit. Main-text peaks are maxima over the two main workloads; Fig. 5(d) instead reports packed model storage, including unquantized parameters.

Retaining reconstructed weights. QuIP#, QTIP, and UniSVQ reconstruct each layer's weights before prompt processing. Discarding each reconstructed matrix after use saves memory; retaining all such matrices across requests avoids repeated reconstruction but adds a 16-bit copy of the quantized layers alongside the compressed weights. With retention, first-token latencies for P128/D1024 and P1024/D128 are QuIP# 21.7/87.3 ms, QTIP 21.9/89.7 ms, and UniSVQ 18.9/68.3 ms, respectively. Peak allocations rise to 8.80–8.86 GiB.

Table 3: Short non-streaming workload: 128 input and 128 output tokens, without retained weight copies. First token is GPU TTFT; next token is CPU-observation-based TPOT, each reported as a median over runs. SAGE achieves a $2.63\times$ end-to-end speedup over FP16.

Method	First token (ms)	Next token (ms)	Total (s)	Output (tokens/s)	Peak (GiB)
FP16	14.28	9.869	1.268	101.0	7.665
AWQ	13.18	4.115	0.536	238.7	2.611
QuIP#	51.96	5.719	0.779	164.4	1.714
QTIP	48.35	5.638	0.765	167.3	1.898
UniSVQ	21.53	4.159	0.550	232.7	1.714
SPHQ	125.13	3.699	0.595	215.1	1.921
SAGE	17.14	3.661	0.483	265.2	1.857

A.3 End-to-End Numerical Consistency

Table 4 compares each quantized method with its own dense quantize–dequantize (QDQ) reference, obtained by independently decoding its packed weights. The 4-token references use eager execution; the 32- and 128-token references use compiled execution. Decode relative L2 is the maximum over the three steps following prefill; maximum absolute error covers all four steps. The Dense row instead compares compiled and eager execution with the original pretrained weights.

Table 4: End-to-end numerical consistency. Lower relative L2 is better.

Method	Prefill tokens	Prefill relative L2 (%)	Decode maximum relative L2 (%)	Maximum absolute error
Dense	4	0.1192	0.2266	0.01660
SPHQuant	4	0.8494	1.0412	0.16016
QuIP#	4	0.8565	1.2647	0.21484
QTIP	4	1.0206	1.3674	0.19336
UniSVQ	4	5.7379	7.5981	1.02002
SAGE	4	**0.6233**	**0.7746**	**0.11914**
SAGE	32	0.8459	0.8986	0.14453
QuIP#	128	4.4006	6.2508	0.89844
QTIP	128	3.0801	3.5425	0.50781
UniSVQ	128	20.0390	20.1103	3.22803
SAGE	128	**1.0376**	**1.0906**	**0.16406**

A.4 Experimental Setting Details

Serving. For the accuracy results in Table 1, we serve each model with vLLM 0.19.0 (Kwon et al., 2023) on a single RTX Pro 6000, with prefix caching, chunked prefill, and asynchronous scheduling enabled. Context limits are 262,144 tokens for Qwen3.5-4B, 65,536 for Qwen3-30B-A3B, and 131,072 for the remaining models. The Ministral models use the Transformers backend and Mistral tokenizer. Automatic tool calling uses the `hermes` parser for Qwen3, `qwen3_coder` for Qwen3.5, and `mistral` for Ministral; Qwen3.5 additionally uses the `qwen3` reasoning parser.

Evaluation. We use EvalScope 1.10.0 through the OpenAI-compatible API, following each model's official model card for sampling hyperparameters. Qwen3.5 runs with thinking enabled and a maximum of 16,384 generated tokens, including reasoning; all other models allow 8,192 generated tokens. Evaluation concurrency is 16 for Qwen3-30B-A3B and 32 otherwise. We fix the generation and dataset-shuffling seeds to 42 and evaluate BFCL-v3 in function-calling mode. Code execution uses network-isolated Docker sandboxes, each limited to one CPU and 512 MiB of memory.

A.5 Calibration Data Construction

Prompt Collection and Filtering. Our calibration prompts are drawn from publicly available datasets, including OpenCodeInstruct (Ahmad et al., 2025), the training splits of GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021), DeepMath-103K (He et al., 2026), OpenHermes-2.5[1], UltraChat (Ding et al., 2023), ShareGPT[2], and When2Call (Ross et al., 2025). The collection covers code, mathematics, reasoning, instruction following, tool use, and general conversation. We rigorously screen calibration prompts against all evaluation sets and remove any detected overlaps. Before response generation, we also exclude prompts that cannot be rendered by the target model's chat template or leave insufficient context for response generation. We sample the six domains with a target prompt ratio of $13 : 13 : 13 : 10 : 9 : 6$ using a fixed random seed and interleave the sampled prompts across domains. These proportions apply to prompt counts rather than token counts.

Model-Specific Response Generation. We retain the source prompts and regenerate responses using each target model. For each model being quantized, we apply its own chat template and

[1] `https://huggingface.co/datasets/teknium/OpenHermes-2.5`

[2] `https://huggingface.co/datasets/Aeala/ShareGPT_Vicuna_unfiltered`

generate responses with its full-precision checkpoint using greedy decoding. For thinking models, we enable thinking mode and retain the generated reasoning traces alongside the final answers. We preserve the template's role markers and concatenate each prompt with its response. These sequences provide inputs for activation statistics collection and blockwise reconstruction without requiring ground-truth answer labels.

Sequence Packing. We concatenate the tokenized conversations, inserting an end-of-sequence token where needed, and partition the resulting streams into contiguous blocks of 2048 tokens. A block may contain multiple conversations or a segment of a longer conversation. Training and held-out streams are packed separately, yielding 1920 training blocks and 128 validation blocks. This fixed token budget is used across models, while the tokenizer, chat template, and generated responses remain specific to each target model.

A.6 Spherical Vector Quantization Preliminaries

We supplement Sec. 3.1 with the reconstruction derivation, continuous parameterization, radius-grid construction, and storage accounting.

Direction and Radius Reconstruction. Since signs are preserved, the reconstruction error in Eq. (1) reduces to approximating $\mathbf{p} = |\mathbf{w}|$ by a scaled nonnegative codeword. For a unit codeword $\mathbf{c}_k$, the error decomposes as:

$$\|\mathbf{p} - \hat{r}\mathbf{c}_k\|_2^2 = \|\mathbf{p}\|_2^2 - (\mathbf{p}^\top \mathbf{c}_k)^2 + (\hat{r} - \mathbf{p}^\top \mathbf{c}_k)^2. \tag{11}$$

The first two terms measure the residual with an unrestricted continuous radius, whose optimum is $\tilde{r}_k^\star = \mathbf{p}^\top \mathbf{c}_k$. Because both $\mathbf{p}$ and $\mathbf{c}_k$ are nonnegative, selecting the smallest residual is equivalent to maximizing cosine similarity. The last term captures the additional error when the fitted radius is quantized. Thus, direction assignment with a continuous radius and selection after radius quantization are distinct problems. The activation-weighted extension yields Eq. (6).

Continuous Parameterization. Following SPHQuant (Zhang et al., 2026), each codeword is parameterized by $v-1$ hyperspherical angles:

$$c_1 = \cos\theta_1, \qquad c_j = \left(\prod_{i=1}^{j-1} \sin\theta_i\right)\cos\theta_j \quad (2 \le j < v), \qquad c_v = \prod_{i=1}^{v-1} \sin\theta_i. \tag{12}$$

Unconstrained logits α_i determine the angles through:

$$\theta_i = \epsilon + \left(\frac{\pi}{2} - 2\epsilon\right)\mathrm{sigmoid}(\alpha_i), \tag{13}$$

where $\epsilon = 10^{-4}$ keeps angles away from the boundaries. This mapping produces nonnegative unit codewords by construction. Continuous radii are parameterized as $\tilde{r} = \mathrm{softplus}(\rho)$, ensuring positivity. Blockwise optimization updates ρ and the codebook logits while retaining the discrete assignments.

Shared Radius Grid. Each group contains $G/v = 16$ consecutive subvectors within a weight row. The grid offset and step in Eq. (2) are computed from their continuous fitted radii:

$$\mu_g = \min_{i \in g} \tilde{r}_i, \qquad \delta_g = \frac{\max_{i \in g} \tilde{r}_i - \mu_g}{2^V - 1}. \tag{14}$$

These parameters are recomputed from the current radii during optimization. If all radii in a group coincide, the reconstruction is their common value. The forward pass uses quantized radii, while a straight-through estimator passes gradients to the continuous radii. Because the grid is shared, its resolution depends on the fitted radius range of the entire group.

Encoding and Storage Costs. The 2.25 BPW reported in the main text refers to the sign, direction-index, and radius-code payload in Eq. (3). For a matrix with $N = d_{\mathrm{out}} d_{\mathrm{in}}$ weights, storing each grid parameter in b_g bits and each codeword component in b_c bits gives:

$$\mathrm{BPW}_{\mathrm{matrix}} = \frac{B}{v} + \frac{2b_g}{G} + \frac{Kvb_c}{N}, \tag{15}$$

excluding permutation and format metadata. FP16 grid parameters contribute an additional 0.25 BPW at $G = 128$, bringing the subtotal to 2.50 BPW before codebook and other metadata. The

codebook overhead depends on both K and matrix size. Accordingly, ABS preserves the per-subvector encoding budget, while total storage also reflects the selected codebooks and deployment metadata.

A.7 SAGE Quantization Algorithm

Algorithm 1: SAGE quantization

Input: Full-precision model and non-evaluation calibration prompts.
Pack prompts and greedy full-precision responses into calibration sequences $\mathcal{D}_{\text{cal}}$.
Run full-precision calibration; estimate each matrix's input-channel energies $e_c := \mathbb{E}[x_c^2]$.
for each weight matrix W to quantize **do**
  Sort channels by decreasing e_c; interleave eight equal-sized tiers to obtain π (ESG).
  Permute columns by π; split rows into 8-vectors, storing signs s and magnitudes p.
  Set position weights $\omega_i \propto e_{a_i}^{0.75}$, normalized to unit mean within each subvector.
  for each $(K, V) \in \{(64, 4), (128, 3), (256, 2)\}$ **do**
    Fit K nonnegative unit codewords by activation-weighted spherical k-means.
    For each p, choose $z, \tilde{r}$ minimizing $\sum_i \omega_i (p_i - \tilde{r} c_{z,i})^2$.
    Round radii to 2^V uniform levels spanning their range in each 128-weight group.
    Score $\hat{w} := \hat{r}(s \odot c_z)$ in original channel order by $\sum_c e_c \|W_{:,c} - \widehat{W}_{:,c}\|_2^2$.
  end for
  Retain the lowest-score candidate to initialize the quantized matrix (ABS).
end for
Initialize $x_1^{\text{q}} := x_1^{\text{fp}}$ from the calibration embeddings.
for $\ell = 1, \ldots, L$ **do**
  Cache full-precision targets $F_\ell(x_\ell^{\text{fp}})$.
  Fix signs, direction indices, permutations, and bit splits in the quantized block $\widehat{F}_\ell$.
  for each minibatch over all optimization epochs **do**
    Recompute radius grids and reconstruct quantized weights.
    Use Adam to minimize mean squared error between $\widehat{F}_\ell(x_\ell^{\text{q}})$ and cached targets.
    Update only radii and codebook angles; pass gradients through radius rounding.
  end for
  Propagate $x_{\ell+1}^{\text{q}} := \widehat{F}_\ell(x_\ell^{\text{q}})$ and $x_{\ell+1}^{\text{fp}} := F_\ell(x_\ell^{\text{fp}})$.
end for
Output: Signs, direction and radius codes, grids, codebooks, and permutations;
  retain unquantized model parameters unchanged.

A.8 Dialog Examples

Table 5: Qwen3.5-4B dialogues with native BF16 or quantized–dequantized (QDQ) weights. Both prompts use seed 42, temperature 1.0, and thinking mode. Generation limits, including reasoning, are 512 tokens for Prompt 1 and 256 for Prompt 2. Gray text denotes reasoning.

Prompt 1 Five numbers have an average of 18. If the number 22 is removed, what is the average of the remaining four numbers? Choices: A. 16 B. 17 C. 18 D. 19. Please show your choice in the `answer` field with only the choice letter, e.g., {"answer": "C"}.

BF16: The user wants me to solve a math problem and provide the answer in a specific format. **1. Understand the Problem:** There are 5 numbers. The average (arithmetic mean) of these 5 numbers is 18. One number, which is 22, is removed from the set. I need to find the new average of the remaining 4 numbers. **2. Formulate Equations:** Let the five numbers be x_1, x_2, x_3, x_4, x_5. The sum of the five numbers is S_5. Average formula: Average $= \frac{\text{Sum}}{\text{Count}}$ Given: $\frac{S_5}{5} = 18$. Therefore, $S_5 = 18 \times 5$. Calculate S_5: $18 \times 5 = 90$. So, the sum of the original 5 numbers is 90. Now, one number is removed. This number is $x_k = 22$. The sum of the remaining 4 numbers is $S_4 = S_5 - x_k$. Calculate S_4: $S_4 = 90 - 22 = 68$. The count of the remaining numbers is 4. The new average is $\text{Average}_{new} = \frac{S_4}{4}$. Calculate Average_{new}: $\frac{68}{4}$. $68/4 = 17$. **3. Check the Choices:** A. 16 B. 17 C. 18 D. 19 My calculated answer is 17, which corresponds to Choice B. **4. Structure the Output:** State the steps clearly. Conclude with the final choice. Format as requested: {"answer": "B"}. **Double Check:** Original Sum = $5 \times 18 = 90$. Remove 22 *(TRUNCATED)*

QuIP#: The user wants me to solve a math problem and provide the answer in a specific format. 1. **Understand the problem**: There are five numbers. The average of these five numbers is 18. One number, specifically 22, is removed. The remaining four numbers must have an average value. I need to determine what that average is from the given choices (A=16, B=17, C=18, D=19). 2. **Analyze the data**: Let n be the number of items in a set. Let $x_1, x_2, \ldots, x_n$ be the elements. The formula for the average is: average $= \frac{\text{sum}}{n}$. Here, we have 5 numbers whose sum equals $5 \times 18 = 90$. When 22 is removed, there are 4 numbers left. The new sum is 90 - 22 = 68. The new average would be $68/4 = 17$. 3. **Wait, let me check my reasoning.** If the original average is 18, with 5 numbers, their total sum is 90. If one number, 22, is removed, the remaining 4 numbers sum to $90 - 22 = 68$. Average of the remaining 4 numbers = $68/4 = 17$. This seems like a straightforward calculation based on averages. However, does this logic work? The problem doesn't say "one of the numbers was 22". It says "the number 22 is removed". Since there were 5 numbers with an average of 18, the sum is indeed 90. Is it possible for the average of the remaining 4 numbers to still be 18? The question asks which choice is correct. If the sum is 90, removing 22 leaves 68. 68 divided by 4 is 17. So the answer should be 17. 4 *(TRUNCATED)*

Continued on next page

Table 5 (continued)

QTIP: Thinking Process: 1. **Calculate total sum of the initial five numbers:** Since the average is 18 and there are 5 numbers, the total sum is $18 \times 5 = 90$. 2. **Remove the number 22 from the set.** The remaining four numbers have a sum of $90 - 22 = 68$. 3. **Calculate the new average:** Divide the remaining sum by the count of remaining numbers (4). New average = $68/4 = 17$. 4. **Match with choices:** Option B corresponds to 17. 5. **Format output:** Provide the choice letter in the answer field as requested. **Step-by-Step Calculation:** Sum of original numbers = Average $\times$ Count = $18 \times 5 = 90$. After removing 22, Sum = $90 - 22 = 68$. Remaining count = 4. New Average = $68/4 = 17$. Choice B matches 17. Answer should be "B". {"answer": "B"}

UniSVQ: The user wants me to know the average of the remaining 4 numbers after removing the number 22. The five original numbers were used to get an average of 18. What are the implications? A. It's about a math problem where I can't remember what it was when I was at school or work? B. It's about statistics: What is the mean of a series? C. It's a question that needs to be answered without knowing the answer. D. It could be related to the average value being something else entirely. E. The question might have nothing to do with statistics, other than the fact that the average is calculated differently in some contexts. *(NO FINAL)*

SPHQuant: The user wants me to solve a math problem about averages. 1. **Analyze the given information:** There are five numbers. Their average is 18. One of the numbers (22) is removed. I need to find the average of the remaining four numbers. 2. **Formulate the relationship between average and sum:** Average = Sum / Count Sum = Average * Count 3. **Calculate the total sum of the original five numbers:** Count = 5 Average = 18 Sum = 5 * 18 4. **Perform the multiplication:** $5 \times 18 = 90$ 5. **Determine the sum of the remaining four numbers:** Original Sum - Removed Number Sum = 90 - 22 $90 - 22 = 68$ 6. **Calculate the new average:** New Average = Sum of remaining numbers / Count of remaining numbers New Count = 4 New Average = 68 / 4 7. **Perform the division:** $68/4 = 17$ 8. **Verify with the choices:** A. 16 B. 17 C. 18 D. 19 The calculated result matches choice B. 9. **Format the output:** The user requested the answer in the answer field with only the choice letter. Format: {"answer": "B"} 10. **Final check:** Original sum: $5 \times 18 = 90$. Remove 22: $90 - 22 = 68$. New count: $5 - 1 = 4$. New average: $68/4 = 17$. Choice is B. Constraint: "Please show your choice in the answer field with only the choice *(TRUNCATED)*

SAGE: The user wants me to solve a math problem and provide the answer in a specific format. **Problem:** Five numbers have an average of 18. One number (22) is removed. What is the average of the remaining four numbers? **Step 1: Calculate the sum of the original five numbers.** Average = Sum / Count $18 = \text{Sum}/5$ Sum $= 18 \times 5$ Sum $= 90$ **Step 2: Calculate the sum of the remaining four numbers.** The number 22 is removed. New Sum = Original Sum - 22 New Sum = $90 - 22$ New Sum = 68 **Step 3: Calculate the new average.** New Average = New Sum / New Count New Average = $68/4$ New Average = 17 **Step 4: Format the output.** The user requested the choice letter in the answer field. Choices: A. 16, B. 17, C. 18, D. 19. My calculated answer is 17, which corresponds to Choice B. The format requires {"answer": "B"}. **Double Check:** Original sum = 90. Remove 22 -> Remainder sum = 68. Count = 4. Average = 68 / 4 = 17. Matches Choice B. **Final Output Generation:** Construct the JSON response. The sum of the five numbers is $18 \times 5 = 90$. After removing 22, the new sum is $90 - 22 = 68$. The average of the remaining four numbers is $68/4 = 17$. This corresponds to Choice B. {"answer": "B"}

Prompt 2 Do not enumerate steps or analyze options. Answer immediately. Which day follows Tuesday? Output the lowercase name of the day, followed by an exclamation mark, with no other text.

BF16: Thinking Process: 1. **Analyze the Request:** Task: Identify the day that follows Tuesday. Constraint 1: Do not enumerate steps. Constraint 2: Do not analyze options. Constraint 3: Answer immediately. Format: Lowercase name of the day, followed by an exclamation mark, with no other text. 2. **Determine the Answer:** Day sequence: Monday -> Tuesday -> Wednesday... Day following Tuesday is Wednesday. 3. **Format the Output:** Name: wednesday Punctuation: ! Combined: wednesday! Check constraints: No extra text, lowercase, exclamation mark at the end. 4. **Final Verification:** Is it lowercase? Yes (wednesday). Is there an exclamation mark? Yes (!). Is there any other text? No. Did I enumerate or analyze? No. 5. **Construct Output:** wednesday! 6. **Final Decision:** Output "wed *(TRUNCATED)*

QuIP#: Wednesday!

QTIP: Yes! yes!

UniSVQ: The user wants me to know what day follows Tuesday. The answer is Wednesday. It's a straightforward question that requires an immediate response without any further analysis or steps. *(NO FINAL)*

SPHQuant: Thinking Process: 1. **Analyze the Request:** Task: Identify the day that follows Tuesday. Constraint 1: Do not enumerate steps or analyze options. (No list, no explanation). Constraint 2: Answer immediately. Constraint 3: Output the lowercase name of the day. Constraint 4: Follow with an exclamation mark (!). Constraint 5: No other text. 2. **Determine the Day:** Current day is Tuesday. The day following Tuesday is Wednesday. 3. **Format the Output:** Lowercase name: "Wednesday" Exclamation mark: "!" Combined: "Wednesday!" No extra text. 4. **Review Constraints:** Did I enumerate? No. Did I analyze options? No. Is it immediate? Yes. Is it lowercase? Yes ("Wednesday"). Is there an exclamation mark? Yes ("!"). Is there any other text *(TRUNCATED)*

SAGE: Thinking Process: 1. **Analyze the Request:** Task: Determine which day follows Tuesday. Constraint 1: Do not enumerate steps or analyze options. Constraint 2: Answer immediately. Constraint 3: Output the lowercase name of the day, followed by an exclamation mark. Constraint 4: No other text. 2. **Determine the Answer:** The day following Tuesday is Wednesday. Format required: lowercase "wednesday!" 3. **Check Constraints:** Is it a single output? Yes. Is it lowercase? Yes ("wednesday"). Is there an exclamation mark? Yes ("!"). Is there any analysis or enumeration? No. 4. **Final Output Formulation:** wednesday! 5. **Final Review:** Does this meet all criteria? Yes.cw wednesday!