ADE: ACCURATE, INFERENCE-EFFICIENT, AND TUNABLE DELTA COMPRESSION FOR TASK-SPECIFIC FINE-TUNED FOUNDATION MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Given multiple task-specific fine-tuned models with the same base, how can we deploy them compactly while preserving each model's performance? Delta compression stores only the difference between a fine-tuned model and its base, enabling compact deployment of many models from a shared base. However, existing methods largely treat the delta as an isolated artifact: they ignore the base model, disregard inference computation, and produce a static result that cannot be further adapted. We propose ADE (Accurate Delta compression with Efficient inference), an accurate, inference-efficient, and tunable delta compression framework with (1) *base-awareness*, allocating parameters by the sensitivity of each module's output, including the base weights, for accuracy, (2) *computation-awareness*, aligning quantization with the direction of matrix multiplication for fast low-bit inference, and (3) *task-awareness*, supporting near-zero-overhead tuning on the compressed delta for low-cost adaptation. Under the same memory budget, ADE is up to 3.9% relatively more accurate than the best prior method, and has up to 18.2% lower inference latency than the fastest prior method.

1 INTRODUCTION

Given multiple task-specific fine-tuned models with the same base, how can we deploy them compactly while preserving each model's performance? Fine-tuning foundation models for specific tasks is standard practice, producing many models from a single base. Serving them independently duplicates the base in memory, scaling cost linearly with the number of models. Delta compression, which stores the differences between fine-tuned and base models in a compact form, has emerged as an active line of research to address this challenge (Liu et al., 2024b; Ping et al., 2024).

However, existing delta compression methods treat the delta as an isolated artifact, decoupled from the base model, the serving system, and downstream adaptation. They focus on reducing each delta's size, treating it in isolation from the final model. Their objective is storage reduction, with inference speed treated as a secondary concern. Once compressed, the delta becomes static and cannot adapt to new tasks.

In this paper, we propose ADE (Accurate Delta compression with Efficient inference), an accurate, inference-efficient, and tunable delta compression framework. As illustrated in Figure 1, while previous methods compress the delta in isolation, ADE takes a broader view guided by three principles. First (base-awareness), ADE maintains accuracy at high compression ratios by decomposing each delta via SVD and allocating the parameter budget across the resulting factors by its sensitivity to the combined base-delta output, rather than delta-only properties. Second (computation-awareness), ADE achieves fast low-bit inference by aligning quantization groups with the direction of matrix multiplication, reducing quantization error via near-zero-overhead rotation. Third (task-awareness), ADE enables near-zero-overhead task adaptation of compressed models through lightweight tuning of vectors trained directly on the compressed delta, avoiding the cost of full fine-tuning.

Our contributions are summarized as follows:

- **Method.** We propose ADE, a delta compression framework that jointly addresses accuracy, inference efficiency, and downstream tunability.

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.

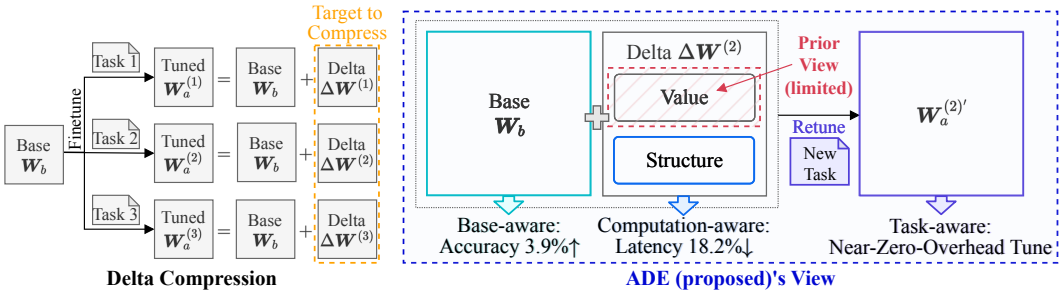


Figure 1: A base-, computation-, and task-aware view of delta compression in ADE (proposed). Prior view (red) considers only delta value. ADE view (blue) achieves high accuracy by exploiting base, fast inference via delta structure, and near-zero-overhead tunability from task specificity.

- **Sustainability.** ADE enables model sustainability. Further adaptation requires only a small number of trainable parameters and incurs no memory overhead after training, minimizing the cost of repeated updates for model enhancement.
- **Experiments.** Across diverse models and tasks, ADE is up to 3.9% relatively more accurate than the best prior delta compression method under the same memory budget, and has up to 18.2% lower inference latency than the fastest one. The codes are available at `https://github.com/ADEforDelta/ADE`.

2 Preliminaries and Related Works

2.1 Delta Compression

Delta compression aims to reduce the storage cost of serving multiple aligned models derived from a common backbone while preserving their accuracy. It compresses the weight differences between the aligned models and the backbone, termed deltas.

2.1.1 Problem Definition

Let $\{{\theta_a}^{(1)}, \ldots, {\theta_a}^{(N)}\}$ denote the weights of N aligned models, where ${\theta_a}^{(n)}$ denotes the weights of the n-th aligned model, and let θ_b denote the weights of their common backbone. The delta of each aligned model is defined as

$$\Delta\theta^{(n)} = {\theta_a}^{(n)} - \theta_b, \qquad n \in \{1, \ldots, N\}, \tag{1}$$

and $\widehat{\Delta\theta}^{(n)}$ denotes its compressed approximation, from which the aligned weights are approximated as $\hat{\theta}_a{}^{(n)} = \theta_b + \widehat{\Delta\theta}^{(n)}$. Compressing each delta to a ratio $\alpha \in (0, 1)$ of the full model size F reduces the storage for the N models from NF to $(1 + N\alpha)F$. We formalize the problem as follows.

Problem 1 (Delta compression)**.** *Given a shared backbone θ_b, N deltas $\{\Delta\theta^{(1)}, \ldots, \Delta\theta^{(N)}\}$, a target compression ratio α, and a calibration dataset $\mathcal{C}$, find compressed approximations $\{\widehat{\Delta\theta}^{(1)}, \ldots, \widehat{\Delta\theta}^{(N)}\}$ at the target ratio that maximize the accuracy of the corresponding aligned models with minimal inference overhead.*

2.1.2 Related Works

Existing methods for delta compression build compact representations of deltas using techniques such as quantization and low-rank approximation. Liu et al. (2024b) binarize fine-tuning deltas, Yu et al. (2024a) randomly drop delta entries and rescale the rest, Ping et al. (2024) apply SVD and mixed-precision quantization, and Yang et al. (2025b) sparsify singular vectors according to their importance. Existing methods use the properties of delta weights to guide compression, but these properties alone do not directly account for the output error induced when the compressed delta is combined with the base model. In contrast, our ADE accounts for how the delta integrates with the base, minimizes the delta's computation cost to accelerate inference, and enables flexible adaptation, which is the intrinsic role of a delta.

2.2 QUANTIZATION

Among model compression techniques such as pruning (Chen et al., 2025; Yang et al., 2025a) and low-rank factorization (Liu et al., 2024c; Dettmers et al., 2023), quantization reduces memory and computational costs through low-bit representations. We review group-wise uniform quantization, a standard scheme adopted in recent LLM compression works (Lin et al., 2024; Xiao et al., 2023). It represents each group of weights with low-bit integer codes, and a shared scale and zero-point, from which approximate real-valued weights are reconstructed.

Let $\boldsymbol{w}^{(g)} \in \mathbb{R}^G$ collect G consecutive entries of a weight matrix $\boldsymbol{W}$ along a row or column, yielding *row-wise* or *column-wise* group quantization. For a target bit-width b, the stored representation is $(\breve{\boldsymbol{w}}^{(g)}, s_g, z_g)$, where $\breve{\boldsymbol{w}}^{(g)} \in \{0, \ldots, 2^b - 1\}^G$ denotes the integer codes, and s_g and z_g are the group's scale and zero-point. The integer codes are computed as

$$\breve{\boldsymbol{w}}^{(g)} = \mathrm{clip}(\lfloor \boldsymbol{w}^{(g)}/s_g \rceil + z_g,\, 0,\, 2^b - 1), \tag{2}$$

where $s_g = (w_{\max} - w_{\min})/(2^b - 1)$ and $z_g = -\lfloor w_{\min}/s_g \rceil$, with $w_{\min} = \min(\boldsymbol{w}^{(g)})$ and $w_{\max} = \max(\boldsymbol{w}^{(g)})$ denoting the group's minimum and maximum values. The operator $\lfloor \cdot \rceil$ rounds to the nearest integer; rounding and clipping are applied elementwise. Dequantization yields the real-valued approximation $\hat{\boldsymbol{w}}^{(g)} = s_g\,(\breve{\boldsymbol{w}}^{(g)} - z_g) \approx \boldsymbol{w}^{(g)}$.

3 PROPOSED METHOD

We propose ADE (Accurate Delta compression with Efficient inference), a unified delta compression framework that is accurate, inference-efficient, and tunable.

3.1 OVERVIEW

ADE combines truncated SVD with quantization to compress deltas along two complementary dimensions, parameter count and numerical precision.

We address the following three challenges in delta compression:

- **C1. Limits of delta-internal signals.** The properties of delta weights alone do not fully capture the importance of delta components to model accuracy. How can we identify which components are essential to preserve?
- **C2. Inflated inference overhead.** Reducing storage does not necessarily accelerate inference, as reconstructing the compressed delta introduces additional computation. How can we minimize the inference overhead of a compressed delta?
- **C3. Loss of adaptability under compression.** Full-model fine-tuning followed by recompression is costly, while additional adapters increase deployment memory. How can we adapt a compressed delta at low training cost without exceeding its memory budget?

The main ideas are summarized as follows:

- **I1. Output-aware capacity allocation (Section 3.2).** We assign each module of delta a retained SVD rank based on its output sensitivity to compression, accounting for both base weights and input activations.
- **I2. Inference-aligned quantization (Section 3.3).** We align quantization groups with the factor-contraction axis to enable low-bit delta reconstruction, while reducing quantization error through a near-zero-overhead rotation.
- **I3. Light fold-in adaptation on compressed delta (Section 3.4).** We train only two vectors per linear module and fold them exactly into the quantization scales of its compressed delta, enabling adaptation without additional deployment memory or computation.

Figure 2 provides an overview of ADE. For the weight matrix of each linear module in delta, we apply truncated SVD with the rank determined by output-aware capacity allocation (I1), then quantize the resulting factors using inference-aligned quantization (I2). For further adaptation, we apply light fold-in adaptation (I3) and fold the learned vectors into quantization scales for inference. We provide the detailed algorithm in Appendix B.

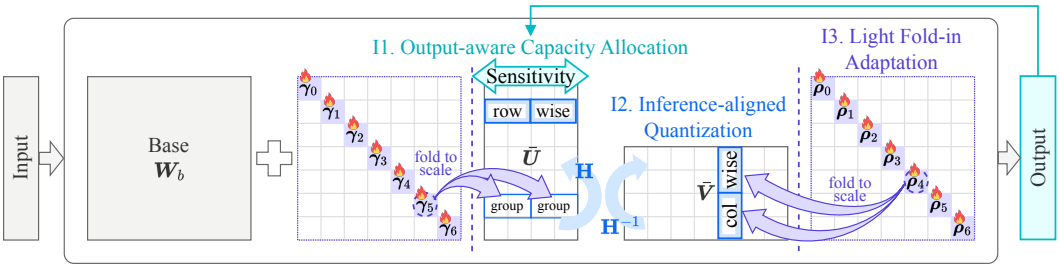


Figure 2: Overall architecture of ADE. ADE allocates parameter capacity to each delta in proportion to its sensitivity to the combined base–delta output, quantizes the delta with a layout aligned to the matrix multiplication direction, and applies lightweight tuning vectors for adaptation.

3.2 Output-aware Capacity Allocation

How can we identify which components of a delta are essential to preserve? We propose output-aware capacity allocation, which applies truncated SVD to the linear-module weight matrices in the delta and determines their retained ranks according to module-output sensitivity, accounting for both base weights and input activations. Existing methods use the properties of delta weights to guide compression, but these properties alone do not capture how approximation errors affect the combined base–delta output. Figure 3 illustrates that the output sensitivity is highly heterogeneous across both modules and models. This variation motivates assigning ranks according to each module's output sensitivity rather than using a uniform or model-independent allocation.

Parameter allocation. Let $\boldsymbol{W}_a \in \mathbb{R}^{d_o \times d_i}$ denote the weight matrix of a linear module in the aligned model, and let $\boldsymbol{W}_b \in \mathbb{R}^{d_o \times d_i}$ denote the corresponding weight matrix in the backbone. d_i and d_o denote the input and output dimensions of both matrices, respectively. The corresponding weight matrix of delta $\Delta\theta$ is $\Delta\boldsymbol{W} = \boldsymbol{W}_a - \boldsymbol{W}_b$. We construct a truncated-SVD approximation $\Delta\boldsymbol{W}_r$ of $\Delta\boldsymbol{W}$ allocating a retained rank r according to the module's sensitivity $S(\boldsymbol{W}_a)$ as

$$\Delta\boldsymbol{W}_r := \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top \approx \Delta\boldsymbol{W}, \qquad (r - R_{\min}) \propto \frac{S(\boldsymbol{W}_a)}{\sum_{\boldsymbol{W}\in\mathcal{W}_a} S(\boldsymbol{W})}, \tag{3}$$

where $R_{\min}$ is the minimum rank reserved for each module and $\mathcal{W}_a$ is the set of linear-module weight matrices in the aligned model. The matrices $\boldsymbol{U} \in \mathbb{R}^{d_o \times r}$ and $\boldsymbol{V} \in \mathbb{R}^{d_i \times r}$ contain the leading r left and right singular vectors, respectively, and $\boldsymbol{\Sigma} = \mathrm{diag}(\sigma_1, \ldots, \sigma_r) \in \mathbb{R}^{r \times r}$ contains the corresponding singular values. The proportionality constant is the remaining rank budget derived from the target compression ratio α, with the detailed allocation rule provided in Appendix C.

Sensitivity score. The sensitivity $S(\boldsymbol{W}_a)$ measures the normalized output drift caused by compressing $\Delta\boldsymbol{W}$ to the uniform-allocation rank $\tilde{r}$ at the target compression ratio α (see Appendix D):

$$S(\boldsymbol{W}_a) = \frac{\left\|\boldsymbol{W}_a\boldsymbol{X} - (\boldsymbol{W}_b + \Delta\boldsymbol{W}_{\tilde{r}})\boldsymbol{X}\right\|_F^2}{\left\|\boldsymbol{W}_a\boldsymbol{X}\right\|_F^2}, \tag{4}$$

where $\boldsymbol{X} \in \mathbb{R}^{d_i \times T}$ denotes calibration activations with sequence length T, and $\Delta\boldsymbol{W}_{\tilde{r}}$ is the rank-$\tilde{r}$ truncated-SVD approximation of $\Delta\boldsymbol{W}$. Since $\boldsymbol{W}_a = \boldsymbol{W}_b + \Delta\boldsymbol{W}$, the numerator is the squared Frobenius norm of $(\Delta\boldsymbol{W} - \Delta\boldsymbol{W}_{\tilde{r}})\boldsymbol{X}$, which measures the delta approximation error, whereas the denominator normalizes it by the output energy of the combined base and delta. Thus, modules with similar approximation errors receive different sensitivities according to their relative effect on the module output, making the metric account for the base model rather than relying solely on the delta.

3.3 Inference-aligned Quantization

How can we minimize the inference-time overhead of applying a compressed delta? We propose inference-aligned quantization, which (1) absorbs the singular values into the two SVD factors and quantizes them along their multiplication direction, enabling inexpensive dequantization, and

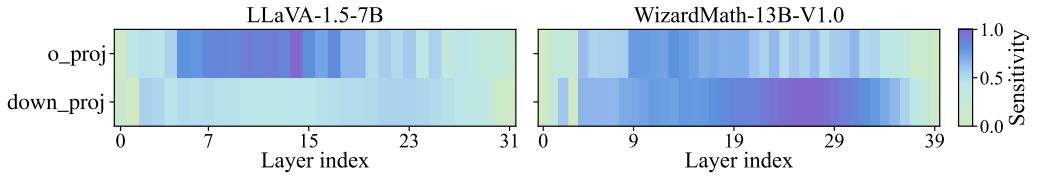


Figure 3: Heterogeneity of module sensitivity across models measured by equation 4. LLaVA concentrates on *o_proj* in middle layers, while WizardMath concentrates on *down_proj* in later layers.

(2) applies a near-zero-overhead orthogonal rotation to reduce quantization error. Existing SVD-based methods quantize SVD factors only for storage and ignore the cost of dequantization, so their low-bit format saves memory but no computation cost.

For a delta with the truncated SVD $\Delta \boldsymbol{W}_r = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ of equation 3, the quantized factors are

$$\big(\check{\boldsymbol{U}},\ \{u_{o,g}\},\ \{\mu_{o,g}\}\big) := Q_r\big(\boldsymbol{U}\boldsymbol{\Sigma}^{1/2}\boldsymbol{H}\big), \qquad \big(\check{\boldsymbol{V}}^\top,\ \{v_{g,j}\},\ \{\nu_{g,j}\}\big) := Q_c\big(\boldsymbol{H}^\top\boldsymbol{\Sigma}^{1/2}\boldsymbol{V}^\top\big), \quad (5)$$

where $\boldsymbol{H} \in \mathbb{R}^{r\times r}$ is an orthogonal rotation matrix with $\boldsymbol{H}\boldsymbol{H}^\top = \boldsymbol{I}$ (details in Section 3.3.2). $Q_r(\cdot)$ and $Q_c(\cdot)$ respectively quantize their inputs row-wise and column-wise in groups of G, following equation 2, and return the corresponding integer codes $(\check{\boldsymbol{U}}, \check{\boldsymbol{V}}^\top)$, scales $(\{u_{o,g}\}, \{v_{g,j}\})$, and zero-points $(\{\mu_{o,g}\}, \{\nu_{g,j}\})$ (details in Section 3.3.1). The real-valued factors are recovered elementwise as

$$\hat{\boldsymbol{U}}_{o,k} := u_{o,g}\big(\check{\boldsymbol{U}}_{o,k} - \mu_{o,g}\big), \qquad \hat{\boldsymbol{V}}^\top_{k,j} := v_{g,j}\big(\check{\boldsymbol{V}}^\top_{k,j} - \nu_{g,j}\big), \quad (6)$$

for every rank index k in the g-th group, so that $\Delta\boldsymbol{W}_r \approx \hat{\boldsymbol{U}}\hat{\boldsymbol{V}}^\top$.

3.3.1 Computation-efficient layout

We lay out the quantized factors so that the delta is reconstructed in low-bit arithmetic during inference without adding cost to other computations. A compressed delta is applied to the input activations $\boldsymbol{X} \in \mathbb{R}^{d_i\times T}$ at inference in one of two ways depending on the number of tokens T: (1) *reconstructed application*, which reconstructs the delta and adds it to the base weight, $(\boldsymbol{W}_b + \hat{\boldsymbol{U}}\hat{\boldsymbol{V}}^\top)\boldsymbol{X}$; or (2) *factored application*, which applies the two factors successively, $\boldsymbol{W}_b\boldsymbol{X} + \hat{\boldsymbol{U}}(\hat{\boldsymbol{V}}^\top\boldsymbol{X})$. Reconstructed application is used for long inputs and factored application for short ones, since reconstruction is performed once per forward regardless of T while the cost of factored application grows with T (see Appendix E.1). Factored application multiplies each quantized factor only with activations, whereas reconstruction multiplies the two quantized factors with each other, tying its cost to the quantization layout of each factor. We therefore quantize the SVD factors in groups along the rank axis with $\boldsymbol{\Sigma}$ absorbed evenly into both, $\boldsymbol{U}\boldsymbol{\Sigma}^{1/2}\boldsymbol{H}$ row-wise and $\boldsymbol{H}^\top\boldsymbol{\Sigma}^{1/2}\boldsymbol{V}^\top$ column-wise, so that their product is computed in low-bit arithmetic.

In *reconstructed application*, the delta is reconstructed in low-bit arithmetic. The integer codes are contracted directly, and floating-point rescaling occurs only once per group after accumulation:

$$(\hat{\boldsymbol{U}}\hat{\boldsymbol{V}}^\top)_{o,j} = \sum_{g=0}^{r/G-1} \underbrace{u_{o,g}\, v_{g,j}}_{\text{full-precision}} \underbrace{\sum_{k=gG}^{(g+1)G-1} \big(\check{\boldsymbol{U}}_{o,k} - \mu_{o,g}\big)\big(\check{\boldsymbol{V}}^\top_{k,j} - \nu_{g,j}\big)}_{\text{low-bit}}. \quad (7)$$

This holds because row o of $\hat{\boldsymbol{U}}$ and column j of $\hat{\boldsymbol{V}}^\top$ are split at the same group boundaries, so the scale and zero-point of a group are constant over its G terms and factor out of the sum. An entry thus requires r/G rescalings, whereas a layout whose groups run along the input or output axis leaves a rank-dependent scale inside every contraction and requires r rescalings, one per product. We analyze the full-precision FLOPs of the two layouts in Appendix E.2.

Factored application is computed most efficiently by multiplying each factor with activations on its integer codes and applying the scale of each group once after its products are accumulated (Frantar et al., 2023; Lin et al., 2024). Under the rank-axis layout, however, this holds only for $\hat{\boldsymbol{U}}$, since the scale of $\hat{\boldsymbol{V}}^\top$ varies along the input dimension and cannot be applied after accumulation. To keep factored application efficient, we instead apply the scale to the input once per group before the product, which suffices because all rank indices of a group share it (see Appendix E.3).

3.3.2 NEAR-ZERO-OVERHEAD ERROR REDUCTION

A shared orthogonal matrix $\boldsymbol{H}$ rotates $\boldsymbol{U}\boldsymbol{\Sigma}^{1/2}$ on the right and $\boldsymbol{\Sigma}^{1/2}\boldsymbol{V}^\top$ on the left before quantization. This placement flattens the magnitude spread of both factors along the rank axis and, at the same time, makes the two rotations cancel inside the product, which removes the need for an inverse rotation at inference. The rotation is necessary because absorbing the singular values concentrates the quantization error: with a long-tail spectrum, both factors $\boldsymbol{U}\boldsymbol{\Sigma}^{1/2}$ and $\boldsymbol{\Sigma}^{1/2}\boldsymbol{V}^\top$ carry a wide magnitude spread along the rank axis, where the quantization groups are formed. Rotation before quantization is a common remedy (Chee et al., 2023; Ashkboos et al., 2024), but existing methods must apply the inverse rotation at every inference call.

The long-tail spread lies across the columns of $\boldsymbol{U}\boldsymbol{\Sigma}^{1/2}$ and the rows of $\boldsymbol{\Sigma}^{1/2}\boldsymbol{V}^\top$. Right and left multiplication by an orthogonal rotation matrix $\boldsymbol{H}$ equalize the scales of the columns of $\boldsymbol{U}\boldsymbol{\Sigma}^{1/2}$ and the rows of $\boldsymbol{\Sigma}^{1/2}\boldsymbol{V}^\top$, respectively (see Appendix F for the properties of the rotation matrix). These are exactly the two sides that meet in the product, and the rotations cancel since $\boldsymbol{H}\boldsymbol{H}^\top = \boldsymbol{I}$:

$$\Delta \boldsymbol{W}_r = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top = \big(\boldsymbol{U}\boldsymbol{\Sigma}^{1/2}\boldsymbol{H}\big)\big(\boldsymbol{H}^\top\boldsymbol{\Sigma}^{1/2}\boldsymbol{V}^\top\big) \;\approx\; \hat{\boldsymbol{U}}\,\hat{\boldsymbol{V}}^\top, \tag{8}$$

where $\hat{\boldsymbol{U}}$ and $\hat{\boldsymbol{V}}^\top$ are the dequantized factors of equation 6. Moreover, no online computation is added since $\boldsymbol{H}$ is absorbed into the stored factors at compression time.

3.4 LIGHT FOLD-IN ADAPTATION ON COMPRESSED DELTA

How can we adapt a compressed delta at low training cost without exceeding its memory budget? We propose light fold-in adaptation, which trains only two vectors that rescale the rows of $\hat{\boldsymbol{U}}$ and the columns of $\hat{\boldsymbol{V}}^\top$, and folds the trained values into the per-group scaling factors of equation 7 so that adaptation leaves no parameter at inference. Existing delta compression methods (Ping et al., 2024; Yang et al., 2025b) treat a compressed delta as fixed: adapting it requires retraining and recompressing the uncompressed model, and an adapter persists at deployment, exceeding the ratio α.

Formally, we train the two vectors on data for a new task a', yielding the adapted weight

$$\boldsymbol{W}_{a'} \;:=\; \boldsymbol{W}_b + \mathrm{diag}(\boldsymbol{\gamma})\,\hat{\boldsymbol{U}}\,\hat{\boldsymbol{V}}^\top\,\mathrm{diag}(\boldsymbol{\rho}), \tag{9}$$

where $\hat{\boldsymbol{U}}$ and $\hat{\boldsymbol{V}}^\top$ are frozen, and $\boldsymbol{\gamma} \in \mathbb{R}^{d_o}$ and $\boldsymbol{\rho} \in \mathbb{R}^{d_i}$ are the trainable vectors, initialized to the all-ones vector so that the adapted delta starts identical to the compressed one. The o-th entry γ_o scales every quantization group in the o-th row of $\hat{\boldsymbol{U}}$, modulating how that row interacts with all columns of $\hat{\boldsymbol{V}}^\top$; symmetrically, ρ_j scales every group in the j-th column of $\hat{\boldsymbol{V}}^\top$. ADE trains $(d_o + d_i)$ parameters per linear module, $1/r_{\text{lora}}$ as many as LoRA (Hu et al., 2022) at rank r_{lora} (Appendix G). The training forward pass follows the same low-bit execution path as inference without materializing the factors in full precision, since $\boldsymbol{\gamma}$ and $\boldsymbol{\rho}$ rescale only the output of $\hat{\boldsymbol{U}}$ and input of $\hat{\boldsymbol{V}}$, respectively.

Merging into scaling factors. After training, the two tuning vectors can be absorbed exactly into the per-group scaling factors of $\hat{\boldsymbol{U}}$ and $\hat{\boldsymbol{V}}^\top$, so that no parameter beyond the compressed delta remains at inference, with no extra computation or memory. Each γ_o is shared across every quantization group in the o-th row of $\hat{\boldsymbol{U}}$ and therefore folds into their scaling factors by the update

$$u_{o,g} \leftarrow \gamma_o\, u_{o,g} \quad \forall g \in \{0, \ldots, r/G - 1\}, \tag{10}$$

where $u_{o,g}$ is the scale of the g-th group in row o of $\hat{\boldsymbol{U}}$, as defined in equation 6. Analogously, ρ_j is folded into the scaling factors $v_{g,j}$ of the j-th column of $\hat{\boldsymbol{V}}^\top$. The integer codes and zero-points are unchanged, $\boldsymbol{\gamma}$ and $\boldsymbol{\rho}$ are discarded, and both the reconstruction of equation 7 and factored application are preserved without modification.

Training objective. We supervise the tuning scalars with the fine-tuned model as the teacher, combining three standard objectives: a language-modeling cross-entropy $\mathcal{L}_{\text{LM}}$, a layer-wise reconstruction loss $\mathcal{L}_{\text{RE}}$ that matches each adapted module's output to its fine-tuned counterpart, and a logit-level distillation $\mathcal{L}_{\text{KD}}$ against the fine-tuned model. The full objective and the precise definitions of $\mathcal{L}_{\text{LM}}$, $\mathcal{L}_{\text{KD}}$, and $\mathcal{L}_{\text{RE}}$ are deferred to Appendix H.

Table 1: Performance of ADE and baselines. ADE shows the best performance.

Method	Train	Math		Code		Multi-Modal		Avg.
		WIZARD-7B	WIZARD-13B	MAGIC-7B	WIZARD-13B	LLAVA-7B	LLAVA-13B	
Aligned	—	55.3	59.4	70.1	69.8	59.6	63.1	62.9
BitDelta	✓	51.1	<u>58.4</u>	68.5	66.9	56.9	60.3	60.4
Delta-CoMe	✗	51.4	57.3	68.5	69.3	53.2	61.2	60.2
DARE	✗	47.9	58.2	68.3	68.3	58.4	62.3	60.6
ImPart	✗	46.9	54.6	66.7	69.0	58.8	62.4	59.7
ADE$_{-\mathcal{L}}$	✗	<u>51.5</u>	<u>58.4</u>	<u>69.3</u>	<u>70.6</u>	<u>59.7</u>	<u>62.9</u>	<u>62.1</u>
ADE (proposed)	✓	**53.4**	**58.8**	**70.4**	**71.2**	**59.9**	**63.0**	**62.8**

4 EXPERIMENTS

Our experiments address the following questions:

Q1. Performance (Section 4.2). How accurate is ADE compared to existing methods?
Q2. Ablation study (Section 4.3). How effective is each component of ADE for the performance?
Q3. Effectiveness of light fold-in adaptation (Section 4.4). How accurately can light fold-in adaptation adapt compressed deltas to new tasks?
Q4. Inference speed (Section 4.5). How fast is ADE compared to baselines?

4.1 EXPERIMENTAL SETTINGS

We briefly introduce the experimental setup. Further setups are detailed in Appendix I.

Tasks. We evaluate on three domains of aligned models, each with a representative benchmark: GSM8K (Cobbe et al., 2021) for mathematical reasoning, MBPP (Austin et al., 2021) from EvalPlus (Liu et al., 2023) for code generation, and TextVQA (Singh et al., 2019) for multi-modal chat. For the adaptation experiment (Section 4.4), we additionally evaluate on PubMedQA (Jin et al., 2019) and FinQA (Chen et al., 2021) as domain QA benchmarks for language models, and COCO Captioning (Lin et al., 2014), ScienceQA-IMG (Lu et al., 2022), and PMC-VQA (Zhang et al., 2023) for multi-modal models. We report accuracy for all benchmarks (pass@1 for MBPP), except BLEU-4 for COCO Captioning, and improvements as relative gains.

Models and baselines. We evaluate six aligned models: WizardMath-v1.0 (7B and 13B) (Luo et al., 2025) for math, MagicoderS-CL-7B (Wei et al., 2024) and WizardCoder-Py-V1.0-13B (Luo et al., 2024) for code, and LLaVA-v1.5 (7B and 13B) (Liu et al., 2024a) for multi-modal chat. Table 6 in Appendix I lists the backbone model of each. We compare ADE against four delta compression methods: 1) BitDelta (Liu et al., 2024b), 1-bit delta weights with a single per-tensor scale distilled on a calibration set; 2) Delta-CoMe (Ping et al., 2024), SVD with magnitude-based mixed-precision quantization of the singular components; 3) DARE (Yu et al., 2024a), random dropping of delta entries with rescaling of the survivors; and 4) ImPart (Yang et al., 2025b), SVD-based delta sparsification that sets the sparsity ratio of each singular vector by its importance. ADE quantizes the delta to a uniform 4 bits with group size $G = 128$, meeting each experiment's target compression ratio through its allocated rank budget, with the per-group scales and zero-points counted in the budget. Only BitDelta calibrates its parameters by gradient descent, distilling its per-tensor scales against the uncompressed model on a general corpus; the other three are training-free. ADE's fold-in vectors are likewise trained against the uncompressed model, on public instruction data in each aligned model's domain rather than on any new task data. We report ADE both without and with this adaptation, as ADE$_{-\mathcal{L}}$ and ADE. In every table, the best and second-best results among the compared methods are in bold and underlined, respectively.

4.2 PERFORMANCE (Q1)

We compare ADE with prior delta compression methods to evaluate its accuracy. Table 1 reports the results at $\alpha = 1/16$, with the uncompressed aligned models as references. We have two main observations. First, ADE$_{-\mathcal{L}}$ outperforms every baseline, improving over the strongest one by up to

Table 2: Ablation study of ADE. Each variant changes one design choice of the untuned ADE$_{-\mathcal{L}}$ (I1, I2) or the tuned ADE (I3). Column labels omit the ADE prefix.

Model	ADE$_{-\mathcal{L}}$	I1 (untuned)			I2 (untuned)				ADE	I3 (tuned)	
		$-\boldsymbol{W}_b$	$-\boldsymbol{X}$	$uniform$	$-\boldsymbol{H}$	$Q_r \times Q_r$	$Q_c \times Q_c$	$Q_c \times Q_r$		$-\boldsymbol{\gamma}$	$-\boldsymbol{\rho}$
WizardMath-13B	58.4	56.5	58.2	57.5	58.1	58.7	58.2	58.6	58.8	58.0	58.6
WizardCoder-13B	70.6	69.6	70.4	70.1	70.4	69.3	69.6	69.8	71.2	69.6	69.8
Avg.	64.5	63.1	64.3	63.8	64.3	64.0	63.9	64.2	65.0	63.8	64.2

Table 3: Performance of ADE and different parameter-efficient fine-tuning baselines on new tasks. ADE outperforms other methods without additional deployment parameters.

Method	TinyLlama-1.1B-Chat				Qwen2.5-14B-Instruct				LLaVA-1.5-7B				Avg.
	Code	Math	Med.	Fin.	Code	Math	Med.	Fin.	Cap.	Chat	SQA	PMC	
Aligned model	16.1	2.1	39.0	1.1	82.0	94.2	61.6	67.6	29.9	59.3	65.6	35.8	46.2
ADE$_{-\mathcal{L}}$	20.9	2.7	25.0	1.9	82.3	94.8	66.6	67.0	29.8	56.1	63.0	23.1	44.4
+ LoRA (rank=1)	25.4	6.0	69.6	11.8	84.9	94.7	79.8	70.9	36.7	56.4	76.2	49.9	55.2
+ DoRA (rank=1)	26.7	5.1	69.6	12.6	85.4	94.3	80.4	70.8	36.8	56.4	77.4	53.5	55.8
ADE (proposed)	**28.6**	**6.4**	**70.6**	**13.9**	**86.2**	**95.2**	**82.0**	**72.1**	**37.1**	**59.1**	**79.9**	**54.4**	**57.1**

1.9% under the same memory budget. This margin is obtained before any tuning, and it includes BitDelta, which distills its per-tensor scales against the uncompressed model. Second, the fold-in adaptation raises accuracy by up to a further 3.7%, making ADE the most accurate configuration overall. It learns per-row and per-column vectors that fold into the existing per-group scales, giving finer control than a single per-tensor scale at an identical deployment cost. Appendix J reports model-wise results at the tighter compression ratio $\alpha = 1/32$.

4.3 Ablation Study (Q2)

We ablate ADE's components to assess their individual contributions to accuracy. Table 2 compares variants that change one design choice at a time on WizardMath-13B and WizardCoder-13B at $\alpha = 1/16$. The I1 and I2 variants ablate the untuned model ADE$_{-\mathcal{L}}$, since these two ideas determine the compressed delta before adaptation, whereas the I3 variants ablate the tuned model ADE. For (I1) output-aware capacity allocation, ADE$_{-\boldsymbol{W}_b}$ and ADE$_{-\boldsymbol{X}}$ remove the base weight $\boldsymbol{W}_b$ and activations $\boldsymbol{X}$, respectively, from the sensitivity metric in equation 4 (Appendix I); ADE$_{uniform}$ uses a uniform SVD rank across modules. For (I2) inference-aligned quantization, ADE$_{-\boldsymbol{H}}$ removes the rotation of equation 8 and quantizes the unrotated factors directly. The remaining three variants examine how the group layout for low-bit reconstruction affects accuracy. Q_r and Q_c denote row-wise and column-wise grouping, and $Q_\cdot \times Q_\cdot$ lists the quantizers of $\boldsymbol{U}\boldsymbol{\Sigma}^{1/2}\boldsymbol{H}$ and $\boldsymbol{H}^\top\boldsymbol{\Sigma}^{1/2}\boldsymbol{V}^\top$ in order. ADE corresponds to $Q_r \times Q_c$, the only layout that admits the low-bit reconstruction of equation 7; the other three layouts do not. For (I3) light fold-in adaptation on compressed delta, ADE$_{-\boldsymbol{\gamma}}$ and ADE$_{-\boldsymbol{\rho}}$ remove one of the two tuning vectors in equation 9 and train only the other, to assess the importance of each vector.

We make three observations. First, all I1 variants underperform ADE$_{-\mathcal{L}}$, showing the importance of output-aware allocation that accounts for both base weights and activations. Second, ADE$_{-\boldsymbol{H}}$ underperforms ADE$_{-\mathcal{L}}$, confirming that the rotation improves accuracy, while ADE$_{-\mathcal{L}}$ achieves the highest average accuracy among the four layouts, showing that the aligned layout enables low-bit reconstruction without sacrificing accuracy. Third, both I3 variants underperform ADE, showing that the per-row and per-column tuning vectors are complementary and both contribute to accuracy.

4.4 Light Fold-in Adaptation (Q3)

We evaluate light fold-in adaptation on new tasks to assess its accuracy and deployment cost. Table 3 reports the accuracy of ADE's lightweight tuning vectors against LoRA (Hu et al., 2022) and DoRA (Liu et al., 2024c) baselines applied on top of ADE$_{-\mathcal{L}}$, on two language models and one multi-modal model. To isolate the effect of the tuning structure, we compress the deltas at $\alpha = 1/32$

Table 4: Forward latency (ms) of ADE and baselines. ADE offers balanced inference speed across sequence lengths.

	Per linear layer					End-to-end				
Method	T=1	T=32	T=128	T=2048	T=8192	T=1	T=32	T=128	T=2048	T=8192
BitDelta	0.122	0.122	<u>0.128</u>	**0.358**	**1.254**	<u>41.1</u>	42.2	<u>46.2</u>	**207.4**	**1059.3**
DARE	<u>0.100</u>	0.126	0.134	<u>0.365</u>	<u>1.268</u>	45.4	49.2	57.2	<u>218.2</u>	<u>1064.7</u>
Delta-CoMe	0.636	0.637	0.649	0.909	1.660	298.2	300.6	304.1	460.4	1306.5
ImPart	0.113	<u>0.120</u>	0.130	0.835	1.834	<u>41.1</u>	<u>41.8</u>	48.3	357.5	1279.9
ADE (proposed)	**0.067**	**0.076**	**0.098**	0.411	1.328	**35.8**	**36.2**	**37.8**	239.5	1091.5

before adaptation and train ADE and all baselines on the same task data for the same number of epochs, tuning the learning rate separately for each method and model. All methods use only the language-modeling cross-entropy $\mathcal{L}_{\text{LM}}$ in equation 20, since the aligned model has not been trained on the new task and therefore cannot serve as a distillation teacher. We make two observations. First, with only light fold-in adaptation of its tuning vectors, ADE achieves substantially higher accuracy on the new task than the uncompressed aligned model while preserving the compression ratio. Second, ADE is more effective than attaching a rank-1 LoRA adapter to the compressed delta, although both rank-1 LoRA and ADE have $(d_o + d_i)$ trainable parameters per linear layer. Moreover, ADE's advantage stands out in post-training memory efficiency: ADE's trainable parameters are absorbed into the compressed delta's scaling factors after training without adding parameters at deployment, unlike a plain LoRA attachment that increases the model size by the number of trainable parameters.

4.5 Inference Speed (Q4)

We compare ADE with prior delta compression methods to evaluate its inference speed in Table 4. We compress WizardMath-7B using each method at a compression ratio of $1/16$ and measure forward latency on an NVIDIA A100 GPU. We measure both the latency per linear layer and the end-to-end latency of the full model for sequence lengths $T \in \{1, 32, 128, 2048, 8192\}$. We evaluate every method including ours using both delta reconstruction and factored application (Section 3.3.1) and report the lower latency at each sequence length. Appendix K details the implementations.

ADE offers balanced inference performance across all sequence lengths. We make three observations. First, ADE is the fastest compression method for short sequences, reducing latency per linear layer and end-to-end by up to 36.7% and 18.2%, respectively, relative to the fastest baseline at $T \in \{1, 32, 128\}$. Second, low-bit reconstruction of the proposed quantization layout makes ADE substantially faster for long sequences than Delta-CoMe and ImPart, reducing latency per linear layer and end-to-end by up to 54.8% and 48.0% at $T \in \{2048, 8192\}$, respectively. This speedup arises because the proposed layout multiplies the two factors in low-bit arithmetic, whereas Delta-CoMe and ImPart multiply FP16 factors, consistent with the analysis in Appendix E.2. Third, for long sequences, ADE approaches the speed of BitDelta and DARE as T grows, reaching within 5.9% per linear layer and 3.0% end-to-end at $T = 8192$. While the remaining gap comes from reconstructing the delta by multiplying two low-rank factors rather than restoring a single stored matrix elementwise as BitDelta and DARE do, the gap shrinks as T increases. This is because the factor multiplication of ADE, which is fixed regardless of T, accounts for a smaller fraction of the total computation for longer sequences (Appendix E.1). Overall, ADE is the fastest for short sequences, which include every decoding step of generation, and approaches the fastest methods as T grows, while achieving the highest accuracy.

5 Conclusion

We propose ADE, a delta compression framework that jointly addresses accuracy, inference efficiency, and adaptability for fine-tuned models. ADE combines output-aware capacity allocation to preserve accuracy by considering base weights and activations, inference-aligned quantization to enable efficient low-bit computation while reducing quantization error, and light fold-in adaptation on compressed delta to support low-cost training without deployment overhead. Experiments show that ADE is up to 3.9% relatively more accurate than the best prior method under the same memory budget, and serves compressed deltas with up to 18.2% lower latency than the fastest prior method.

AI USE STATEMENT

In this work, we use generative AI tools solely to improve the clarity and readability of the manuscript and to assist with writing and editing code. All AI-generated or AI-edited text and code are manually reviewed by the authors. We take full responsibility for the final content of this work, including all text, code, and scientific claims.

REPRODUCIBILITY STATEMENT

We describe the experimental settings, including models, datasets, and baselines, in Section 4.1 and Appendix I. Appendix K details the hardware, inference implementations, and timing protocol used to measure latency. Our code is publicly available through the link in Section 1.

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. In *NeurIPS*, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

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

Xinrui Chen, Haoli Bai, Tao Yuan, Ruikang Liu, Kang Zhao, Xianzhi Yu, Lu Hou, Tian Guan, Yonghong He, and Chun Yuan. A simple linear patch revives layer-pruned large language models. In *NeurIPS*, 2025.

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan Routledge, and William Yang Wang. FinQA: A dataset of numerical reasoning over financial data. In *EMNLP*, 2021.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing GPT-4 with 90%* ChatGPT quality, 2023. URL `https://lmsys.org/blog/2023-03-30-vicuna/`.

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.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *NeurIPS*, 2023.

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

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *ICLR*, 2022.

Qiao Jin, Bhuwan Dhingra, Zhengping Liu, William W. Cohen, and Xinghua Lu. PubMedQA: A dataset for biomedical research question answering. In *EMNLP-IJCNLP*, 2019.

Solomon Kullback and Richard A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

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.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common objects in context. In *ECCV*, 2014.

Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning. In *CVPR*, 2024a.

James Liu, Guangxuan Xiao, Kai Li, Jason D. Lee, Song Han, Tri Dao, and Tianle Cai. BitDelta: Your fine-tune may only be worth one bit. In *NeurIPS*, 2024b.

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.

Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. DoRA: Weight-decomposed low-rank adaptation. In *ICML*, 2024c.

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*, 2025.

Pan Lu, Swaroop Mishra, Tony Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. Learn to explain: Multimodal reasoning via thought chains for science question answering. In *NeurIPS*, 2022.

Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Jian-Guang Lou, Chongyang Tao, Xiubo Geng, Qingwei Lin, Shifeng Chen, Yansong Tang, and Dongmei Zhang. WizardMath: Empowering mathematical reasoning for large language models via reinforced Evol-Instruct. In *ICLR*, 2025.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. WizardCoder: Empowering code large language models with Evol-Instruct. In *ICLR*, 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.

Bowen Ping, Shuo Wang, Hanqing Wang, Xu Han, Yuzhuang Xu, Yukun Yan, Yun Chen, Baobao Chang, Zhiyuan Liu, and Maosong Sun. Delta-CoMe: Training-free delta-compression with mixed-precision for large language models. In *NeurIPS*, 2024.

Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Amanpreet Singh, Vivek Natarajan, Meet Shah, Yu Jiang, Xinlei Chen, Dhruv Batra, Devi Parikh, and Marcus Rohrbach. Towards VQA models that can read. In *CVPR*, 2019.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford Alpaca: An instruction-following LLaMA model, 2023. URL `https://github.com/tatsu-lab/stanford_alpaca`.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with OSS-Instruct. In *ICML*, 2024.

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.

Mingzhe Yang, Sihao Lin, Changlin Li, and Xiaojun Chang. Let LLM tell what to prune and how much to prune. In *ICML*, 2025a.

Yan Yang, Yixia Li, Hongru Wang, Xuetao Wei, James Jianqiao Yu, Yun Chen, and Guanhua Chen. ImPart: Importance-aware delta-sparsification for improved model compression and merging in LLMs. In *ACL*, 2025b.

Le Yu, Bowen Yu, Haiyang Yu, Fei Huang, and Yongbin Li. Language models are Super Mario: Absorbing abilities from homologous models as a free lunch. In *ICML*, 2024a.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T. Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. MetaMath: Bootstrap your own mathematical questions for large language models. In *ICLR*, 2024b.

Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. TinyLlama: An open-source small language model. *arXiv preprint arXiv:2401.02385*, 2024.

Xiaoman Zhang, Chaoyi Wu, Ziheng Zhao, Weixiong Lin, Ya Zhang, Yanfeng Wang, and Weidi Xie. PMC-VQA: Visual instruction tuning for medical visual question answering. *arXiv preprint arXiv:2305.10415*, 2023.

A Notation

We summarize the frequently used notations in Table 5. Note that "resp." stands for "respectively."

B Overall Algorithm of ADE

Algorithm 1 summarizes ADE, which processes each aligned model independently. It comprises three stages: output-aware capacity allocation (lines 1–7), inference-aligned quantization (lines 8–11), and light fold-in adaptation (lines 13–20).

The first stage allocates rank according to each module's estimated sensitivity to compression. We collect calibration activations that characterize how each module is used (line 1), then compute the SVD of each delta and evaluate the normalized output drift induced by truncation at its uniform probe rank (lines 2–4). After reserving $R_{\min}$ ranks for every module, we distribute the remaining bit budget required to meet the target compression ratio α in proportion to these sensitivities (lines 6–7).

The second stage transforms the allocated low-rank approximations into a low-bit representation designed for direct inference-time computation. For each module, we truncate its SVD to the allocated rank to satisfy the target compression ratio α (lines 8–9), apply paired rotations to reduce quantization error without inference-time overhead, and quantize the factors in groups along their rank (contraction) axis to enable direct low-bit contractions (lines 10–11). Here, $u^{(l)} = \{u_{o,g}\}$ and $\mu^{(l)} = \{\mu_{o,g}\}$ collect the group scales and zero-points of $\check{\boldsymbol{U}}_l$ in equation 7, and $v^{(l)} = \{v_{g,j}\}$ and $\nu^{(l)} = \{\nu_{g,j}\}$ collect the group scales and zero-points of $\check{\boldsymbol{V}}_l{}^{\top}$.

The final stage adapts the compressed delta without increasing deployment memory or computation. We initialize the row- and column-scaling vectors to one, so that optimization begins from the compressed delta (line 13), attach them to every module (line 14), and optimize only these vectors on $\mathcal{D}$ (lines 16–17). We then fold the learned vectors into the corresponding quantization scales; the returned delta therefore preserves both the original storage budget and low-bit inference path (lines 19–20).

C Sensitivity-aware Rank Allocation

We allocate the rank of each module in two parts: a minimum rank $R_{\min}$ that every module receives, and an additional rank ${R_{\text{sen}}}^{(l)}$ granted in proportion to the module's sensitivity. The total is fixed in

Table 5: Frequently used notations.

Symbol	Description
$\theta_a{}^{(n)},\ \theta_b$	Weights of the n-th aligned and shared backbone models, resp.
$\Delta\theta^{(n)},\ \widehat{\Delta\theta}^{(n)}$	Original and reconstructed delta weights of $\theta_a{}^{(n)}$, resp.
$\mathrm{Mem}(\widehat{\Delta\theta}^{(n)})$	Storage of the compressed representation, including metadata
$\hat{\theta}_a{}^{(n)}$	Restored aligned weights ($\hat{\theta}_a{}^{(n)} = \theta_b + \widehat{\Delta\theta}^{(n)}$)
$N,\ F$	Number of aligned models and full-model memory size, resp.
α	Ratio of compressed delta storage to full-model storage
$\boldsymbol{W}_b,\ \boldsymbol{W}_a \in \mathbb{R}^{d_o \times d_i}$	Base and aligned weight matrices, resp.
$\Delta\boldsymbol{W}$	Delta weight matrix ($\Delta\boldsymbol{W} = \boldsymbol{W}_a - \boldsymbol{W}_b$)
$d_o,\ d_i$	Output and input dimensions, resp.
$\boldsymbol{X} \in \mathbb{R}^{d_i \times T}$	Calibration activation matrix with sequence length T
$S(\boldsymbol{W}_a)$	Sensitivity score of $\boldsymbol{W}_a$
$\tilde{r},\ \Delta\boldsymbol{W}_{\tilde{r}}$	Probe rank and rank-$\tilde{r}$ SVD approximation of $\Delta\boldsymbol{W}$, resp.
$R(\cdot),\ r$	Allocated rank and corresponding realized rank
$\boldsymbol{U},\ \boldsymbol{\Sigma},\ \boldsymbol{V}$	Truncated SVD factors ($\Delta\boldsymbol{W} \approx \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$)
$\sigma_i,\ \boldsymbol{u}_i,\ \boldsymbol{v}_i$	i-th singular value and corresponding left/right singular vectors
$L,\ l$	Number of linear modules and module index, resp.
$R_{\min},\ R_{\text{sen}}{}^{(l)},\ B$	Minimum rank, sensitivity-allocated rank, and bit budget
$b_{\text{orig}},\ b_{\text{quant}}$	Bit-widths of the original and quantized factors, resp.
$b_s,\ b_z$	Bit-widths of the per-group scale and zero-point, resp.
$\boldsymbol{w}^{(g)} \in \mathbb{R}^G$	Vector of G consecutive entries from a row or column of $\boldsymbol{W}$
$b,\ s_g,\ z_g$	Bit-width, group-g scale, and zero-point, resp.
$w_{\min},\ w_{\max}$	Minimum and maximum values within the current group
$\check{\boldsymbol{w}}^{(g)},\ \hat{\boldsymbol{w}}^{(g)}$	b-bit integer codes and their real-valued reconstruction, resp.
$G,\ g$	Number of entries per quantization group and group index, resp.
$(\cdot)^{(g)}$	Entries of quantization group g
$\boldsymbol{H}$	Orthogonal rotation matrix
$u_{o,g},\ \mu_{o,g}$	Scale and zero-point of the g-th group in row o of $\check{\boldsymbol{U}}$, resp.
$v_{g,j},\ \nu_{g,j}$	Scale and zero-point of the g-th group in column j of $\check{\boldsymbol{V}}^\top$, resp.
$Q_r(\cdot),\ Q_c(\cdot)$	Row-wise and column-wise group quantization operators, resp.
$\hat{\boldsymbol{U}},\ \hat{\boldsymbol{V}}^\top$	Dequantized factors, equation 6
$\boldsymbol{\gamma},\ \boldsymbol{\rho}$	Tuning vectors for row- and column-wise scaling, resp.
$\mathcal{L}_{\text{LM}},\ \mathcal{L}_{\text{KD}},\ \mathcal{L}_{\text{RE}}$	Language-modeling, KD, and reconstruction losses, resp.
$\lambda_{\text{LM}},\ \lambda_{\text{KD}},\ \lambda_{\text{RE}}$	Balancing hyperparameters
p, h	Output distributions and hidden representations, resp.

bits, so that the compressed deltas meet the target compression ratio α exactly. Let $\theta_a = \{\boldsymbol{W}_a{}^{(l)}\}_{l=1}^{L}$ denote the L linear modules of the fine-tuned model, $\boldsymbol{W}_a{}^{(l)} \in \mathbb{R}^{d_o{}^{(l)} \times d_i{}^{(l)}}$, and $\Delta\boldsymbol{W}_l$ the corresponding delta. The rank assigned to $\Delta\boldsymbol{W}_l$ is

$$
\begin{aligned}
r_l \;&=\; R_{\min} \;+\; R_{\text{sen}}{}^{(l)}, \\
R_{\text{sen}}{}^{(l)} \;&=\; \frac{S(\boldsymbol{W}_a{}^{(l)})}{\sum_{l'=1}^{L} S(\boldsymbol{W}_a{}^{(l')})} \cdot \frac{B}{\bar{c}_S},
\end{aligned}
\tag{11}
$$

where $S(\boldsymbol{W}_a{}^{(l)})$ is the sensitivity of module l from equation 4, B is the bit budget that remains after the $R_{\min}$ ranks of every module are reserved, and $\bar{c}_S$ is the average number of bits that one rank costs under this split; $B/\bar{c}_S$ is therefore the number of ranks distributed according to sensitivity, and $R_{\text{sen}}{}^{(l)}$ is module l's share of it.

The budget is expressed in bits because a rank costs a different number of bits in modules of different shapes. A rank-r_l truncated SVD of $\Delta\boldsymbol{W}_l$ is stored as the quantized factors $\check{\boldsymbol{U}} \in \mathbb{Z}^{d_o{}^{(l)} \times r_l}$ and

Algorithm 1: Overall process of ADE.

Input: Base weights $\theta_b = \{\boldsymbol{W}_b{}^{(l)}\}_{l=1}^{L}$ and aligned weights $\theta_a = \{\boldsymbol{W}_a{}^{(l)}\}_{l=1}^{L}$ of L linear modules, calibration data $\mathcal{C}$, target ratio α, bit-width b, group size G, minimum rank $R_{\min}$, task-specific data $\mathcal{D}$, and number of epochs E

Output: Compressed delta $\widehat{\Delta\theta}$ stored as low-bit integer codes and zero-points with full-precision scales

/ Output-aware capacity allocation (Section 3.2) */*

Collect the input activations $\boldsymbol{X}_l$ of each module $\boldsymbol{W}_a{}^{(l)}$ by forwarding $\mathcal{C}$ through the aligned model;
for $l = 1, \ldots, L$ **do**
  Compute $\mathrm{SVD}(\Delta\boldsymbol{W}_l)$, where $\Delta\boldsymbol{W}_l = \boldsymbol{W}_a{}^{(l)} - \boldsymbol{W}_b{}^{(l)}$;
  Compute the sensitivity $S(\boldsymbol{W}_a{}^{(l)})$ under $\boldsymbol{X}_l$ at probe rank $\tilde{r}_l$; *// Eqs. 4, 13*
end
$B \leftarrow$ bit budget at ratio α after reserving $R_{\min}$ per module; *// Eq. 12*
Allocate rank r_l to each $\Delta\boldsymbol{W}_l$ by apportioning B in proportion to the sensitivity $S(\boldsymbol{W}_a{}^{(l)})$; *// Eq. 11*

/ Inference-aligned quantization (Section 3.3) */*

for $l = 1, \ldots, L$ **do**
  Truncate the SVD of $\Delta\boldsymbol{W}_l$ to rank r_l: $\Delta\boldsymbol{W}_l \approx \boldsymbol{U}_l\boldsymbol{\Sigma}_l\boldsymbol{V}_l{}^{\top}$;
  $(\check{\boldsymbol{U}}_l, u^{(l)}, \mu^{(l)}) \leftarrow Q_r(\boldsymbol{U}_l{\boldsymbol{\Sigma}_l}^{1/2}\boldsymbol{H}_l)$; *// rotate by $\boldsymbol{H}_l$ and quantize along the rank axis; Eqs. 5, 8*
  $(\check{\boldsymbol{V}}_l{}^{\top}, v^{(l)}, \nu^{(l)}) \leftarrow Q_c(\boldsymbol{H}_l{}^{\top}{\boldsymbol{\Sigma}_l}^{1/2}\boldsymbol{V}_l{}^{\top})$; *// likewise for $\boldsymbol{V}_l$*
end

/ Light fold-in adaptation (Section 3.4) */*

$\boldsymbol{\gamma}_l \leftarrow \mathbf{1}_{d_o^{(l)}}, \boldsymbol{\rho}_l \leftarrow \mathbf{1}_{d_i^{(l)}}, \forall l$; *// initialize the trainable tuning vectors*
Attach the tuning vectors to each module as $\boldsymbol{W}_b{}^{(l)} + \mathrm{diag}(\boldsymbol{\gamma}_l)\hat{\boldsymbol{U}}_l\hat{\boldsymbol{V}}_l{}^{\top}\mathrm{diag}(\boldsymbol{\rho}_l), \forall l$; *// Eq. 9*
repeat
  Compute $\mathcal{L}$ of the adapted model on $\mathcal{D}$; *// Eq. 20*
  Update $\{\boldsymbol{\gamma}_l, \boldsymbol{\rho}_l\}_{l=1}^{L}$ by a gradient step on $\mathcal{L}$;
until E *epochs*;
Fold $\boldsymbol{\gamma}_l$ into the row scales $u^{(l)}$ and $\boldsymbol{\rho}_l$ into the column scales $v^{(l)}, \forall l$; *// Eq. 10*
return $\widehat{\Delta\theta} = \{(\check{\boldsymbol{U}}_l, u^{(l)}, \mu^{(l)}, \check{\boldsymbol{V}}_l{}^{\top}, v^{(l)}, \nu^{(l)})\}_{l=1}^{L}$;

$\check{\boldsymbol{V}}^{\top} \in \mathbb{Z}^{r_l \times {d_i}^{(l)}}$ (Section 3.3), so each rank adds one column of $\check{\boldsymbol{U}}$ and one row of $\check{\boldsymbol{V}}^{\top}$, costing $c_l = \big(b_{\mathrm{quant}} + (b_s + b_z)/G\big)\,(d_o{}^{(l)} + d_i{}^{(l)})$ bits, where b_{quant} is the bit-width of the factors and b_s and b_z are the bit-widths of the scale and zero-point stored for each group of G entries. The budget and the average cost are

$$B \;=\; \alpha\, b_{\mathrm{orig}} \sum_{l=1}^{L} d_o{}^{(l)}\, d_i{}^{(l)} \;-\; R_{\min} \sum_{l=1}^{L} c_l, \qquad \bar{c}_S \;=\; \sum_{l=1}^{L} \frac{S(\boldsymbol{W}_a{}^{(l)})}{\sum_{l'=1}^{L} S(\boldsymbol{W}_a{}^{(l')})}\, c_l, \tag{12}$$

where b_{orig} is the bit-width of the original weights. With this choice the allocated ranks cost exactly the budget, $\sum_l c_l\, r_l \;=\; \alpha\, b_{\mathrm{orig}} \sum_l d_o{}^{(l)} d_i{}^{(l)}$, while the ranks beyond $R_{\min}$ remain proportional to sensitivity.

In practice, every rank must be a multiple of the group size G, since the factors are quantized in groups of G along the rank axis (Section 3.3). We therefore choose $R_{\min}$ as a multiple of G and round each r_l to the nearest multiple of G; if the rounded ranks exceed the budget, G ranks are removed from the least sensitive modules, and if they fall short, G ranks are added to the most sensitive modules, so that the total storage never exceeds the budget.

D Probe Rank for Sensitivity

The probe rank $\tilde{r}$ in equation 4 is the per-module rank that would exactly meet the target compression ratio α under uniform allocation, providing a budget-aware reference point at which the sensitivity score evaluates each module.

Storing a rank-$\tilde{r}$ truncated-SVD delta takes $\big(b_{\mathrm{quant}} + (b_s + b_z)/G\big) \cdot \tilde{r} \cdot (d_o + d_i)$ bits, including the per-group scales and zero-points (Appendix C), compared to $b_{\mathrm{orig}} \cdot d_o \cdot d_i$ bits for the original

module. Setting the former to α times the latter and rounding the resulting $\tilde{r}$ down to an integer clipped to $[1, \min(d_o, d_i)]$ yields

$$\tilde{r} \;=\; \mathrm{clip}\left(\left\lfloor \frac{\alpha\, b_{\mathrm{orig}}\, d_o\, d_i}{\big(b_{\mathrm{quant}} + (b_s + b_z)/G\big)\,(d_o + d_i)} \right\rfloor,\; 1,\; \min(d_o, d_i)\right), \tag{13}$$

where b_{orig} and b_{quant} denote the bit-widths of the original weights and the low-rank factors, b_s and b_z those of each group's scale and zero-point, G is the group size, and d_o, d_i are the output and input dimensions of the module, respectively.

E Cost Analysis of the Inference Paths

E.1 Reconstructed versus factored application

We compare the arithmetic of the two paths, counting every product as a full-precision matrix multiplication and ignoring quantization for simplicity. For a module with $\boldsymbol{W}_b \in \mathbb{R}^{d_o \times d_i}$, a delta of rank r with factors $\hat{\boldsymbol{U}} \in \mathbb{R}^{d_o \times r}$ and $\hat{\boldsymbol{V}}^\top \in \mathbb{R}^{r \times d_i}$, and an input $\boldsymbol{X} \in \mathbb{R}^{d_i \times T}$ of T tokens, the two paths compute

$$\textit{Reconstructed:}\quad \underbrace{\big(\boldsymbol{W}_b \underbrace{+}_{d_o d_i} \underbrace{\hat{\boldsymbol{U}}\hat{\boldsymbol{V}}^\top}_{2r d_o d_i}\big)\boldsymbol{X}}_{2 d_o d_i T}, \tag{14}$$

$$\textit{Factored:}\quad \underbrace{\boldsymbol{W}_b\boldsymbol{X}}_{2 d_o d_i T} \underbrace{+}_{d_o T} \underbrace{\hat{\boldsymbol{U}}\ \underbrace{(\hat{\boldsymbol{V}}^\top \boldsymbol{X})}_{2 r d_i T}}_{2 r d_o T}, \tag{15}$$

where the number below each matrix multiplication and addition is its FLOPs. Reconstructed application thus takes $2d_o d_i T + (2r+1) d_o d_i$ FLOPs and factored application $2d_o d_i T + \big(2r(d_o + d_i) + d_o\big)T$. Both share the base term $2d_o d_i T$, and the delta term is fixed at $(2r+1)d_o d_i$ for reconstruction but grows linearly with T for factored application. Factored application is therefore cheaper for $T < (2r+1)d_o d_i / \big(2r(d_o + d_i) + d_o\big)$, and reconstructed application for longer inputs. In practice, the quantized factors are applied by low-bit kernels whose dequantization and rescaling costs, arithmetic precision, and memory traffic differ between the two paths, so the actual crossover differs from this bound.

E.2 Reconstruction cost and the quantization layout

Equation 7 shows that the proposed layout reduces the number of floating-point rescalings per output entry from r to r/G. We make the difference explicit using the controlled case $G = r$, where ADE quantizes the first factor of equation 5 row-wise and the second column-wise, while direction-unaware quantization applies row-wise quantization to both factors:

$$\textit{ADE:}\quad (\Delta \boldsymbol{W}_r)_{o,j} \approx u_o\, v_j \sum_{k=0}^{r-1} \big(\check{\boldsymbol{U}}_{o,k} - \mu_o\big)\big(\check{\boldsymbol{V}}^\top_{k,j} - \nu_j\big), \tag{16}$$

$$\textit{Direction-unaware:}\quad (\Delta \boldsymbol{W}_r)_{o,j} \approx u_o \sum_{k=0}^{r-1} v_k \,\big(\check{\boldsymbol{U}}_{o,k} - \mu_o\big)\big(\check{\boldsymbol{V}}^\top_{k,j} - \nu_k\big). \tag{17}$$

(u_o, μ_o) is the row-wise scale–zero-point pair of $\check{\boldsymbol{U}}$, and (v_j, ν_j) and (v_k, ν_k) are the column-wise and row-wise pairs of $\check{\boldsymbol{V}}^\top$, respectively. ADE uses 2 full-precision FLOPs per entry, whereas row-wise quantization of both factors uses $2r$: r multiplications by v_k, $r-1$ additions, and one multiplication by u_o. Both layouts multiply the same low-bit codes. Under the protocol of Appendix K, on the 4096×4096 query projection with $r = 512$ and $G = 128$, reconstruction in the layout of equation 16 reduces the latency per linear layer by 36.1% at sequence length $T = 2048$ and by 12.8% at $T = 8192$ relative to the layout of equation 17.

E.3 FACTORED APPLICATION AND THE QUANTIZATION LAYOUT

We show that the layout of equation 5 for low-bit reconstruction is applied efficiently in factored form $\hat{\boldsymbol{U}}(\hat{\boldsymbol{V}}^\top \boldsymbol{X})$ by ordering the operations to scale the input before the contraction. We compare its FLOPs with the direction-unaware layout of equation 17, which quantizes both $\hat{\boldsymbol{U}}$ and $\hat{\boldsymbol{V}}^\top$ row-wise. We count the full-precision FLOPs of $\boldsymbol{Z} = \hat{\boldsymbol{V}}^\top \boldsymbol{X}$ for a simple comparison, since both layouts quantize the first factor $\hat{\boldsymbol{U}}$ row-wise identically. The two layouts compute the (k, t)-th entry of $\boldsymbol{Z}$ as follows:

$$\textit{ADE:} \quad \boldsymbol{Z}_{k,t} = \sum_{j=0}^{d_i-1} \left(\check{\boldsymbol{V}}^\top_{k,j} - \nu_{g,j}\right) \underbrace{\left(v_{g,j}\, \boldsymbol{X}_{j,t}\right)}_{\text{scaled once per group}}, \tag{18}$$

$$\textit{Direction-unaware:} \quad \boldsymbol{Z}_{k,t} = \sum_{g'=0}^{d_i/G-1} v_{k,g'} \sum_{j=g'G}^{(g'+1)G-1} \left(\check{\boldsymbol{V}}^\top_{k,j} - \nu_{k,g'}\right) \boldsymbol{X}_{j,t}, \tag{19}$$

where $(v_{g,j}, \nu_{g,j})$ is the scale–zero-point pair of the g-th group in column j of $\check{\boldsymbol{V}}^\top$, with $g = \lfloor k/G \rfloor$ the group to which row k belongs, $(v_{k,g'}, \nu_{k,g'})$ is the pair of the g'-th group in row k, and G is the group size. Both layouts use $(2d_i - 1 + d_i/G)$ full-precision FLOPs per entry of $\boldsymbol{Z}$. In equation 18, ADE spends d_i/G multiplications on scaling the input by $v_{g,j}$, once per group, with the scaled input reused across the G rows of the group, followed by d_i multiplications with the codes and $d_i - 1$ additions. In equation 19, the direction-unaware layout spends d_i multiplications with the codes, $d_i - 1$ additions, and d_i/G multiplications by $v_{k,g'}$. Standard quantized GEMM applies each scale to a partial sum, as in equation 19, which is not possible in equation 18 because $v_{g,j}$ changes along the summation index j. Applying the scale to the input instead keeps the full-precision FLOPs of equation 18 equal to those of standard quantized GEMM on the row-wise layout.

F INCOHERENCE PROCESSING WITH ORTHOGONAL ROTATIONS

Incoherence processing is a technique for transforming a matrix distribution into a more flat distribution when there exist large scale disparities between rows or columns of the matrix (Chee et al., 2023). In model compression, it is used to reshape the distribution into a form that is easier to compress (Chee et al., 2023; Ashkboos et al., 2024; Liu et al., 2025). The most common approach for incoherence processing is to rotate the matrix using an orthogonal rotation matrix, which has been shown to reduce the scale disparities within the matrix (Chee et al., 2023). When the scale disparity is large across rows, an orthogonal rotation matrix is multiplied on the left side of the matrix to normalize row-wise scales. Conversely, when the scale disparity is large across columns, an orthogonal rotation matrix is multiplied on the right side of the matrix to normalize column-wise scales.

The rotation of equation 8 runs along the rank axis, so $\boldsymbol{H}$ is an $r \times r$ matrix, where the rank r of each module is a multiple of G (Appendix C). We take the normalized Sylvester Hadamard matrix when r is a power of two and a random orthogonal matrix from the QR decomposition of a Gaussian random matrix (Chee et al., 2023) otherwise; neither costs anything at inference, since $\boldsymbol{H}$ is absorbed into the stored factors.

G PARAMETER COUNT OF LIGHT FOLD-IN ADAPTATION

Our tuning scheme is both cheaper to train and more inference-efficient than standard PEFT (Parameter-Efficient Fine-Tuning) alternatives. A common recipe for adapting a quantized model is to attach LoRA modules (Hu et al., 2022); however, LoRA introduces two low-rank matrices per weight matrix, requiring $r_{\text{lora}}(d_o + d_i)$ trainable parameters in total, and their structure is not aligned with the quantization layout, so the trainable parameters cannot be folded into the scaling factors. As a result, LoRA enlarges the inference-time memory footprint and adds extra matrix multiplications at every forward pass. In contrast, our scheme uses only $(d_o + d_i)$ trainable parameters per linear module ($\boldsymbol{\gamma} \in \mathbb{R}^{d_o}$ and $\boldsymbol{\rho} \in \mathbb{R}^{d_i}$)—a factor of $1/r_{\text{lora}}$ relative to LoRA—and the alignment of $\boldsymbol{\gamma}$ and $\boldsymbol{\rho}$ with the per-row and per-column quantization groups allows them to be absorbed exactly into the existing scaling factors after training, incurring zero overhead at inference.

Table 6: Backbone and aligned models for each task.

Task	Base model	Aligned model
Math	Llama-2-7B (Touvron et al., 2023)	WizardMath-7B-V1.0 (Luo et al., 2025)
	Llama-2-13B (Touvron et al., 2023)	WizardMath-13B-V1.0 (Luo et al., 2025)
Code	CodeLlama-Py-7B (Rozière et al., 2023)	MagicoderS-CL-7B (Wei et al., 2024)
	CodeLlama-Py-13B (Rozière et al., 2023)	WizardCoder-Py-V1.0-13B (Luo et al., 2024)
Multi-Modal	Vicuna-v1.5-7B (Chiang et al., 2023)	LLaVA-v1.5-7B (Liu et al., 2024a)
	Vicuna-v1.5-13B (Chiang et al., 2023)	LLaVA-v1.5-13B (Liu et al., 2024a)
Chat	TinyLlama-1.1B-intermediate-step-1431k-3T (Zhang et al., 2024)	TinyLlama-1.1B-Chat-v1.0 (Zhang et al., 2024)
Instruction Following	Qwen2.5-14B (Qwen Team, 2024)	Qwen2.5-14B-Instruct (Qwen Team, 2024)

H Training Objective for Light Fold-in Adaptation

We supervise the tuning scalars introduced in Section 3.4 with the fine-tuned model as the teacher. The full training objective combines a language-modeling cross-entropy $\mathcal{L}_{\text{LM}}$, a logit-level distillation $\mathcal{L}_{\text{KD}}$ against the fine-tuned model, and a layer-wise reconstruction loss $\mathcal{L}_{\text{RE}}$ that matches each adapted module's output to its fine-tuned counterpart:

$$\begin{aligned}
\mathcal{L} \;&=\; \lambda_{\text{LM}}\,\mathcal{L}_{\text{LM}} + \lambda_{\text{KD}}\,\mathcal{L}_{\text{KD}} + \lambda_{\text{RE}}\,\mathcal{L}_{\text{RE}},\\
\mathcal{L}_{\text{LM}} &= -\,\mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\big[\log p_{\theta^{\text{ad}}}(y \mid \boldsymbol{x})\big],\\
\mathcal{L}_{\text{KD}} &= \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\big[D_{\text{KL}}\big(p_{\theta_a}(\cdot \mid \boldsymbol{x}) \,\|\, p_{\theta^{\text{ad}}}(\cdot \mid \boldsymbol{x})\big)\big],\\
\mathcal{L}_{\text{RE}} &= \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\big[\,\|h_a(\boldsymbol{x}) - h_{\text{ad}}(\boldsymbol{x})\|_2^2\,\big],
\end{aligned} \tag{20}$$

where $\lambda_{\text{LM}}, \lambda_{\text{KD}}, \lambda_{\text{RE}}$ are balancing hyperparameters, $\mathcal{D}$ is the task-specific data distribution from which $(\boldsymbol{x}, y)$ is sampled, p_{θ_a} and $p_{\theta^{\text{ad}}}$ are the output distributions of the fine-tuned and adapted models, and $D_{\text{KL}}(\cdot \,\|\, \cdot)$ denotes KL divergence between two probability distributions (Kullback & Leibler, 1951). The reconstruction term $\mathcal{L}_{\text{RE}}$ aligns the hidden representations of the teacher and the compressed model across all linear layers, where $h_a(\boldsymbol{x})$ and $h_{\text{ad}}(\boldsymbol{x})$ denote those representations from the fine-tuned model and the adapted model, respectively. However, when adapting to a new task on which the aligned model has not been trained, the aligned model cannot serve as a teacher; we therefore optimize only the language-modeling loss $\mathcal{L}_{\text{LM}}$.

I Details of the Experimental Settings

Table 6 lists the backbone and aligned model pairs for each task. All experiments are conducted on a single NVIDIA A100 GPU, and all models are implemented in PyTorch (Paszke et al., 2019). For ADE, we train the tuning vectors with the AdamW optimizer, searching the number of epochs and the learning rate for each model. The maximum sequence length is fixed to 2048 tokens. The fold-in vectors are trained on MetaMathQA (Yu et al., 2024b) for the math models, the Magicoder instruction data (Wei et al., 2024) for the code models, and Alpaca (Taori et al., 2023) for the multi-modal models; the number of samples for each model is given in our code. The calibration activations for the sensitivity in equation 4 are drawn from the same datasets. The adaptation baselines (LoRA, DoRA) are attached to ADE$_{-\mathcal{L}}$ on the same linear modules as ADE's tuning vectors, and trained with the same objective, data, epochs, and optimizer as ADE, with the learning rate searched independently for each method and model. For each delta compression baseline, we use the official public implementation with its default configuration. All results are reported from a single run under a fixed random seed for reproducibility. We provide all model-specific hyperparameters in our public code repository.

The ablations ADE$_{-\boldsymbol{W}_b}$ and ADE$_{-\boldsymbol{X}}$ in Table 2 replace the sensitivity of equation 4 with

$$S_{-\boldsymbol{W}_b}(\boldsymbol{W}_a) = \frac{\big\|(\Delta\boldsymbol{W} - \Delta\boldsymbol{W}_{\tilde{r}})\boldsymbol{X}\big\|_F^2}{\big\|\Delta\boldsymbol{W}\,\boldsymbol{X}\big\|_F^2}, \qquad S_{-\boldsymbol{X}}(\boldsymbol{W}_a) = \frac{\big\|\Delta\boldsymbol{W} - \Delta\boldsymbol{W}_{\tilde{r}}\big\|_F^2}{\big\|\boldsymbol{W}_a\big\|_F^2}, \tag{21}$$

which normalize the approximation error by the delta output instead of the combined output, and measure it on the weights instead of the outputs, respectively.

Table 7: Performance of delta compression methods at $\alpha = 1/32$.

Method	Math		Code		Multi-Modal	
	WIZARD-7B	WIZARD-13B	MAGIC-7B	WIZARD-13B	LLAVA-7B	LLAVA-13B
Aligned	55.3	59.4	70.1	69.8	59.6	63.1
BitDelta	35.0	50.6	68.3	64.8	56.4	60.0
Delta-CoMe	48.4	56.3	67.5	68.3	52.8	60.8
DARE	43.9	55.4	66.4	67.5	56.2	61.6
ImPart	42.4	52.0	66.1	68.3	56.8	60.1
ADE$-\mathcal{L}$	48.6	57.2	69.0	70.1	59.0	62.8
ADE (proposed)	**50.6**	**58.0**	**69.3**	**70.4**	**59.2**	**62.9**

J PERFORMANCE UNDER STRONGER COMPRESSION

We compare delta compression methods under a tighter memory budget to evaluate accuracy under stronger compression. Table 7 reports results for each of the six aligned models in Table 1 at $\alpha = 1/32$, with ADE using 4-bit quantization and $G = 128$. Uncompressed aligned models are included as references. For BitDelta, we binarize only the deltas of the matrices with the largest $\|\Delta \boldsymbol{W}\|_F / \|\boldsymbol{W}_b\|_F$ that fit within the budget, as spending one bit per parameter takes twice the budget at $\alpha = 1/32$.

K INFERENCE SPEED MEASUREMENT PROTOCOL

We detail the measurement protocol for the inference speed results reported in Section 4.5.

Hardware and latency measurement. All latencies are measured on one NVIDIA A100-SXM4-40GB GPU with PyTorch in eager mode and no CUDA graphs. We measure latency per linear layer on a 4096×4096 query projection using `cudaEvent` for each iteration, discarding 30 warmup iterations and reporting the median of 200 measured iterations. We measure end-to-end latency for a single forward pass of WizardMath-7B over T input tokens at batch size 1, with 3 warmup passes and the median of 50 measured passes (10 at $T \geq 2048$).

Inference implementation for each method. For a target compression ratio of $1/16$, each method stores its delta weights as follows: our proposed ADE uses INT4 codes with per-group FP16 scales and INT4 zero-points, with a group size of $G = 128$; BitDelta (Liu et al., 2024b) uses a 1-bit sign matrix with a per-tensor scale; DARE (Yu et al., 2024a) uses a sparse matrix of the retained entries after pruning; Delta-CoMe (Ping et al., 2024) uses its mixed-precision matrix, which the released code dequantizes to 16-bit precision; and ImPart (Yang et al., 2025b) uses sparsified FP16 SVD factors, which are full rank.

Each method uses either reconstructed application $(\boldsymbol{W}_b + \widehat{\Delta \boldsymbol{W}})\boldsymbol{X}$ or factored application $\boldsymbol{W}_b\boldsymbol{X} + \widehat{\Delta \boldsymbol{W}}\boldsymbol{X}$ depending on the sequence length T (Section 3.3.1); reconstruction is faster for long sequences and factored application for short ones. We measure both applications for every method and report the faster one at each sequence length. BitDelta uses reconstructed application at every T. DARE applies its sparse matrix directly at $T = 1$ per linear layer and for $T \leq 128$ end to end, and uses reconstructed application otherwise. Delta-CoMe uses reconstructed application at every T, as its released forward implementation materializes the delta. ImPart uses factored application for $T \leq 2048$ and reconstructs only at $T = 8192$; all of its computation is in FP16. For its factored application, we densify the sparsified factors into FP16 matrices, which exceed the $1/16$ memory budget and thus favor ImPart. ADE uses factored application for $T \leq 128$ and reconstructed application for larger T. We implement both applications of ADE with custom CUDA kernels that read the same packed 4-bit codes, whereas the baselines run their released implementations or standard PyTorch and cuSPARSE operations. Reconstruction uses the INT4 Tensor Core contraction in equation 7, with INT32 accumulation followed by per-group rescaling, and factored application multiplies the codes with activations in FP16, applying the scales as described in Appendix E.3.