PARAMOLE: EFFICIENT TENSOR-PARALLEL FINE-TUNING OF MIXTURE OF LORA EXPERTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Mixtures of LoRA experts (MoLE) expand the adaptation capacity of large language models through sparse routing, but fine-tuning retains training states for the entire expert pool. Distributing these states can reduce memory use, yet memory feasibility alone does not ensure efficient training. Joint expert and router updates require cross-GPU coordination, while uneven expert batches can limit the efficiency of sparse computation. To address these challenges, we propose **ParaMoLE**, a framework for memory-efficient MoLE fine-tuning on tensor-parallel (TP) backbones under a fixed GPU budget. ParaMoLE adopts partial factor replication to reduce expert-state storage relative to full replication while jointly evaluating each selected expert within the backbone's TP group. During backward, ParaMoLE reuses reduced low-rank activations or activation gradients for both expert and router updates, avoiding separate synchronization of replicated-factor parameter gradients. Grouped matrix multiplication processes variable-sized expert batches in both forward and backward without padding them to a common length. We establish equivalence to the original MoLE outputs and gradients in exact arithmetic. Experiments across five NLP datasets show lower memory, training-step time, and communication on 3B and 14B LLMs, with comparable downstream quality.

1 INTRODUCTION

Low-rank adaptation (LoRA) reduces the cost of fine-tuning large language models (LLMs) by learning low-rank updates to a frozen backbone Hu et al. (2022). Mixtures of LoRA experts (MoLE) extend this formulation by combining multiple LoRA through learned routing Dou et al. (2024); Gao et al. (2025); Liao et al. (2025). Sparse token-level routing limits the number of active experts per token, allowing total adaptation capacity to increase without evaluating the entire pool. The memory requirements of fine-tuning, however, extend beyond the active experts: parameters, gradients, and optimizer states for the full pool coexist with backbone weights and intermediate activations. This combined footprint can restrict the scale of MoLE fine-tuning even when expert computation remains sparse. We study this problem for backbones deployed with tensor parallelism (TP), seeking to support larger backbones and expert pools under a limited GPU count and per-GPU memory budget while maintaining efficient end-to-end training.

Existing approaches distribute expert parameters at different granularities, with distinct consequences for memory and execution. Full replication keeps the entire expert pool on every GPU, duplicating its training states. Expert parallelism (EP) reduces this replication by assigning experts to different GPUs, but makes each GPU's workload depend on the tokens routed to its resident experts. Skewed routing can therefore lead to load imbalance and synchronization delays Zhang et al. (2025). A finer-grained alternative is to partition each expert's LoRA factors across the backbone's TP group. Existing LoRA systems illustrate different factor layouts: S-LoRA Sheng et al. (2023) shards both factors to reduce per-GPU adapter storage, at the cost of communication between the two LoRA multiplications, and BD-LoRA Wang et al. (2026) instead constrains selected factors to be block-diagonal, removing additional adapter communication by changing the adapter parameterization. The partially replicated layout in vLLM Kwon et al. (2023) keeps both multiplications local by replicating one factor, trading additional storage and computation for fewer intermediate exchanges. For MoLE fine-tuning, however, lower expert-state storage does not necessarily translate into efficient training, because experts and routers must still be trained jointly under uneven, token-dependent workloads.

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.

The central problem is therefore not simply to fit a larger expert pool into memory, but to train the resulting mixture efficiently under the same GPU budget.

Achieving both memory feasibility and training efficiency presents two coupled challenges. *First, joint expert and router training must remain communication-efficient after expert states are distributed.* Expert updates and routing-weight derivatives depend on shared intermediate computations, but recovering an expert's parameter gradient does not generally recover the token-level information needed to train the router. Synchronizing the two update paths independently can therefore introduce redundant exchanges. Efficient training requires a communication scheme that supports both updates while preserving the original gradients. *Second, sparse expert activation must translate into efficient GPU computation.* A fixed top-k limits the number of experts selected by each token, but does not balance the number of tokens assigned to each expert. Low LoRA ranks and TP feature partitioning further produce narrow matrix products with different batch sizes in both forward and backward. Executing these products separately incurs repeated launch overhead, while padding batches to a common length adds unnecessary work. Efficient execution must accommodate these unequal batches without changing expert parameters or token assignments.

To address these challenges, we propose **ParaMoLE**, a framework for memory-efficient MoLE fine-tuning on TP backbones. Building on the partially replicated TP-LoRA layout, ParaMoLE jointly evaluates each selected expert across the backbone's GPU group while reducing expert-state duplication relative to full replication. All GPUs select experts using the same complete routing scores. For joint expert and router updates, ParaMoLE synchronizes low-rank activations or their gradients instead of parameter-gradient matrices. Specifically, we derive whether each adapted layer requires the activation between its two LoRA factors or the gradient of that activation. The corresponding contributions from all selected token-expert pairs are aggregated in one backward all-reduce per layer. Each GPU reuses the result to compute both expert-factor gradients and routing-weight derivatives. For efficient expert computation, ParaMoLE processes variable-sized expert batches with grouped GEMM. We group tokens by their selected experts and retain each expert's actual batch size, avoiding the redundant computation of padding all batches to a common length. Forward and backward reuse the same grouping, while recorded token indices guide the weighted accumulation of expert outputs. We establish that this execution preserves the outputs and gradients of the original MoLE formulation in exact arithmetic. Our contributions are summarized as follows:

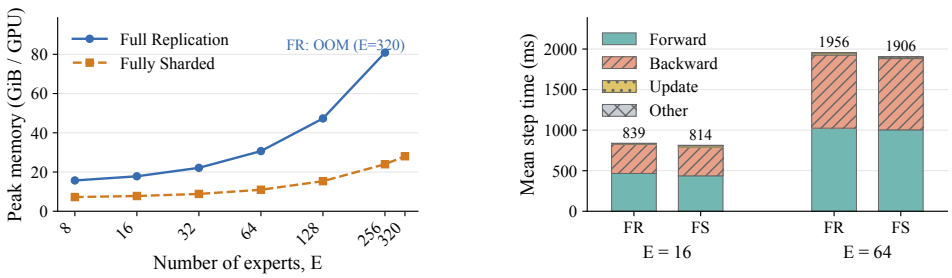
- We propose **ParaMoLE**, a framework for memory-efficient fine-tuning of token-routed MoLE on TP backbones. ParaMoLE jointly trains experts and routers within the backbone's GPU group while reducing expert-state storage relative to full replication.
- We derive a backward synchronization scheme that reuses reduced low-rank activations or activation gradients for both expert and router updates. We integrate this scheme with grouped forward and backward computation over variable-sized expert batches, and establish equivalence to the original MoLE outputs and gradients in exact arithmetic.
- Experiments on 3B and 14B backbones across five NLP datasets show lower peak memory and competitive training speed.

2 MOTIVATION

We first examine whether reducing expert-state replication alone is sufficient for efficient MoLE fine-tuning. We then examine two remaining choices: which tensors to exchange across GPUs and how to execute the unequal batches of tokens routed to different experts.

State sharding alone does not ensure efficient training. We compare Full Replication (FR) and Fully Sharded (FS) on a frozen Qwen2.5-3B backbone using four H100 GPUs with four-way tensor parallelism (TP). FR replicates expert parameters, gradients, and optimizer states on every GPU, whereas FS distributes these states across the GPU group. Both implementations attach experts to the FC1 (`up`) and FC2 (`Down`) projections of the feed-forward network (FFN) and use the same per-expert loop backend. We vary the number of experts E per adapted projection, fixing the LoRA rank at $r = 16$, top-k at $k = 2$, sequence length at 1024, and both micro-batch and global batch size at 1. All runs use a fixed set of XSum examples.

At $E = 64$, FR and FS reach peak allocated memory of 30.7 and 11.0 GiB per GPU, respectively, but take similar mean step times of 1956 and 1906 ms (Fig. 1). At $E = 320$, FR runs out of



(a) Training memory

(b) Training-step breakdown

Figure 1: Resource costs of FR and FS with the same per-expert loop backend. (a) Peak allocated memory over GPUs and measured steps as E increases; FR encounters OOM at $E = 320$. (b) Mean complete-step time at $E = 16$ and $E = 64$, including forward, backward, and optimizer updates. Backward includes final gradient synchronization; other is the time outside the three measured phase intervals. OOM is reported without an imputed memory value.

memory, whereas FS completes the measured steps with a peak of 28.0 GiB per GPU. Increasing E from 16 to 64 raises step time by approximately $2.3\times$ in both implementations, despite fixed top-2 routing. Thus, state sharding improves memory feasibility without eliminating the observed growth in training time in these loop-based implementations. The measured phases include both computation and communication and do not isolate the cause of the increase. These results motivate examining training execution beyond expert-state storage. We focus on two choices: which tensors are synchronized across GPUs and how the tokens routed to each expert are batched for computation.

The same gradient can be recovered with different message sizes. We illustrate this choice using FC2, the FFN down projection. Under the partially replicated layout, each expert's first LoRA factor A_e is partitioned across GPUs, while its second factor $B_e \in \mathbb{R}^{r\times D}$ is replicated, where D is the backbone hidden width. For the T_e tokens selecting expert e, let $Z_e \in \mathbb{R}^{T_e\times r}$ collect the intermediate vectors between its two LoRA multiplications. GPU p computes a contribution $Z_e^{(p)}$ to these vectors, with $Z_e = \sum_p Z_e^{(p)}$. FC2 sums the GPUs' output contributions, so each contribution receives the same output gradient. With identical expert selections and routing weights across GPUs, let $H_e \in \mathbb{R}^{T_e\times D}$ contain the corresponding gradient rows multiplied by their routing weights and the LoRA scale. For the training loss $\mathcal{L}$,

$$\nabla_{B_e}\mathcal{L} = \underbrace{\sum_p (Z_e^{(p)})^\top H_e}_{\text{reduce parameter gradients}} = \underbrace{(\sum_p Z_e^{(p)})^\top H_e}_{\text{reduce low-rank activations}} . \tag{1}$$

Each sum can be performed by an all-reduce that returns the result to every GPU. The first path forms local parameter gradients before reducing them, exchanging an $r \times D$ matrix per GPU. The second reduces the $T_e \times r$ activations first and then computes the complete parameter gradient locally. Both paths are equivalent in exact arithmetic, but their reduction inputs differ in size: at equal precision, the second is smaller when $T_e \ll D$.

Router learning introduces an additional requirement. Reducing parameter gradients accumulates token contributions into a matrix, but does not in general recover the derivative of each token's mixing weight. In contrast, Z_e retains an r-dimensional vector for each token selecting expert e. Together with the replicated B_e and the FC2 output gradients, these vectors also provide the mixing-weight derivatives needed to train the router. The reduction tensor should therefore be chosen not only for its size, but also for whether the same exchange supports both expert and router updates. This example compares the messages for one factor update; Appendix D accounts for the remaining training communication.

Fixed top-k does not guarantee uniform expert batches. Top-k routing fixes the number of experts selected by each token, but not the number of tokens received by each expert. Each expert therefore applies its own factors to an input matrix with a potentially different number of token rows. Executing these narrow matrix products separately requires repeated calls, whereas padding all nonempty batches to a common length adds redundant computation and storage. The unequal token counts

also determine how many contributions are accumulated in each parameter-gradient product during backward. Efficient execution must therefore accommodate the actual batch sizes in both passes while preserving the correspondence between tokens, selected experts, and routing weights. Appendix C.4 illustrates the resulting batch shapes, padding overhead, and output accumulation.

These two execution choices guide ParaMoLE's synchronization and grouped computation, developed in Section 3.

3 METHODOLOGY

3.1 MOLE FORMULATION AND TP SETTING

Consider an adapted linear layer with input $X \in \mathbb{R}^{T\times d_{\text{in}}}$ and frozen weight $W \in \mathbb{R}^{d_{\text{in}}\times d_{\text{out}}}$. Each row x_i is one token's representation at this layer; T counts the token rows passed to an adapted projection in one microbatch, whereas b_μ counts input examples. Expert e has two trainable factors, $A_e \in \mathbb{R}^{d_{\text{in}}\times r}$ and $B_e \in \mathbb{R}^{r\times d_{\text{out}}}$, where r is the LoRA rank. The router $R \in \mathbb{R}^{d_{\text{in}}\times E}$ computes scores $L{=}XR$. Token i selects k distinct experts and normalizes their scores:

$$\mathcal{T}_i = \text{TopK}(L_i, k), \qquad g_{i,e} = \frac{\exp(L_{i,e})}{\sum_{v\in\mathcal{T}_i}\exp(L_{i,v})}, \quad e \in \mathcal{T}_i. \tag{2}$$

The original MoLE output is

$$y_i = x_i W + s\sum_{e\in\mathcal{T}_i} g_{i,e}(x_i A_e)B_e, \qquad s = \alpha/r. \tag{3}$$

We use this complete formulation, including the backbone, experts, and router, as the reference for distributed execution. Each adapted projection has an independent LoRA expert pool and router. We update these parameters using the task loss, retaining all selected pairs without adapter dropout or an auxiliary routing loss. The frozen base still transmits input gradients to earlier layers.

We consider P TP GPUs processing the same T token rows. The adapted sites are the up and down projections of a gated feed-forward network (FFN), with widths $D \to F \to D$. We refer to these sites as up (FC1) and down (FC2); only the up branch is adapted when gate and up are fused. Up computes different output coordinates on different GPUs, whereas down sums contributions computed from different input coordinates. We assume equal partitions, $P \mid F$, and no sequence or context partitioning within the TP group. Superscript (p) identifies GPU $p \in \{0,\dots,P-1\}$; r is reserved for the LoRA rank. ParaMoLE changes the execution of Eq. 3, not its model or objective.

3.2 EXPERT PLACEMENT AND FORWARD COMPUTATION

We adopt the partially replicated TP-LoRA layout: one expert factor follows the base weight partition, and the other is replicated. Every selected expert is consequently evaluated jointly by the backbone's TP group. Fig. 2 shows this forward computation. Subscripts u and d distinguish the two sites and their independent expert selections.

Up: compute the assigned output coordinates. Each GPU holds the full FFN input $X \in \mathbb{R}^{T\times D}$. The router $R_u \in \mathbb{R}^{D\times E}$ and expert factors $A_{u,e} \in \mathbb{R}^{D\times r}$ are replicated. GPU p holds $W_{\text{up}}^{(p)} \in \mathbb{R}^{D\times(F/P)}$ and $B_{u,e}^{(p)} \in \mathbb{R}^{r\times(F/P)}$ for matching output columns. It computes $L_u = XR_u$, applies Eq. 2, and evaluates

$$z_{u,i,e} = x_i A_{u,e}, \qquad U_i^{(p)} = x_i W_{\text{up}}^{(p)} + s\sum_{e\in\mathcal{T}_{u,i}} g_{u,i,e} z_{u,i,e} B_{u,e}^{(p)}. \tag{4}$$

Both terms contain complete values for the same output coordinates and are added locally. The corrected output enters the unchanged SwiGLU operation:

$$G^{(p)} = XW_{\text{gate}}^{(p)}, \qquad M^{(p)} = \text{SiLU}(G^{(p)}) \odot U^{(p)}. \tag{5}$$

Here $G^{(p)}, U^{(p)}, M^{(p)} \in \mathbb{R}^{T\times(F/P)}$. The notation $U_i^{(p)}$ denotes the i-th row of $U^{(p)}$. The complete low-rank activations $z_{u,i,e}$ are saved for backward, while $M^{(p)}$ directly supplies the down input.

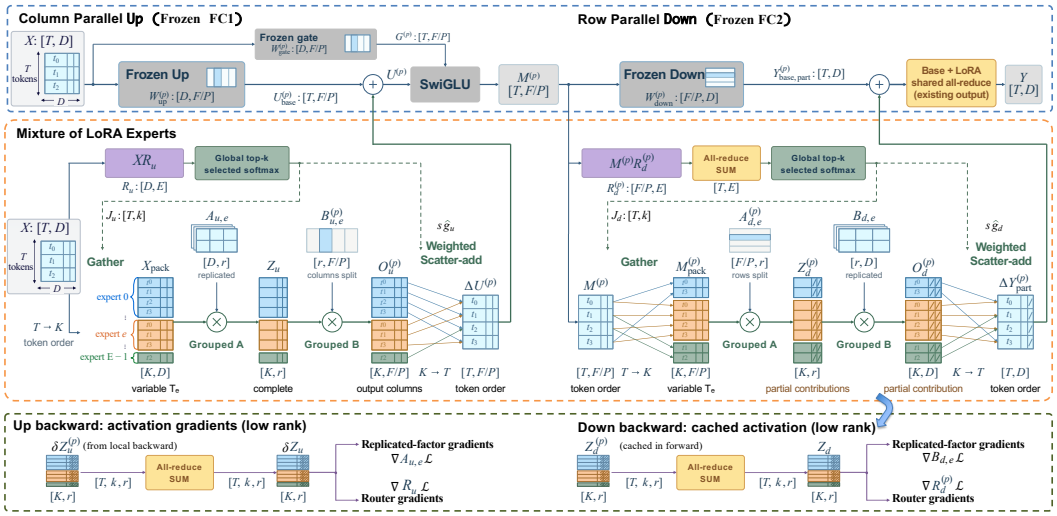


Figure 2: ParaMoLE execution on GPU p. Up computes output coordinates assigned to that GPU; down computes additive contributions to all output coordinates. Down routing uses summed scores, and expert corrections share the existing base output all-reduce. Grouped products retain each expert's actual token count. Weighted accumulation restores token order. Shapes are per GPU; $K = Tk$ counts selected token-expert pairs.

Down: sum contributions to the same output. GPU p holds $M^{(p)}$, $W^{(p)}_{\text{down}} \in \mathbb{R}^{(F/P)\times D}$, and $A^{(p)}_{d,e} \in \mathbb{R}^{(F/P)\times r}$, while $B_{d,e} \in \mathbb{R}^{r\times D}$ is replicated. The router follows the input partition, with $R^{(p)}_d \in \mathbb{R}^{(F/P)\times E}$. Expert selection first requires complete scores:

$$L_d = \mathrm{AR}\Big(M^{(p)} R^{(p)}_d\Big) = M R_d, \qquad M = [M^{(0)}, \ldots, M^{(P-1)}]. \tag{6}$$

The complete R_d vertically stacks the row partitions $R^{(p)}_d$. AR denotes SUM all-reduce: it sums corresponding GPU contributions and returns the sum to every GPU. Both sites use the same expert-index tie rule and selected-slot ordering, so each pair refers to the same expert across GPUs. After routing, down computes

$$z^{(p)}_{d,i,e} = m^{(p)}_i A^{(p)}_{d,e}, \qquad \widetilde{y}^{(p)}_i = m^{(p)}_i W^{(p)}_{\text{down}} + s \sum_{e\in\mathcal{T}_{d,i}} g_{d,i,e} z^{(p)}_{d,i,e} B_{d,e}, \qquad y_i = \mathrm{AR}\Big(\widetilde{y}^{(p)}_i\Big). \tag{7}$$

Each expert result has all D output coordinates but contributes only part of their values. Since $\sum_p z^{(p)}_{d,i,e} B_{d,e} = (\sum_p z^{(p)}_{d,i,e}) B_{d,e}$, these results join the base partial output before its existing all-reduce, as in prior TP-LoRA execution Kwon et al. (2023); Sheng et al. (2023). The two expert products therefore remain local. We cache $z^{(p)}_{d,i,e}$ rather than assembling complete $z_{d,i,e}$ in forward; the latter is recovered when needed for training.

3.3 Low-Rank Synchronization for Joint Training

We select the tensors to reduce from the derivatives shared by expert and router updates. For either adapted linear layer, let x_i, y_i be its input and output and $q_i = \partial\mathcal{L}/\partial y_i$. At down, this input is m_i. Let $\mathcal{I}_e = \{i : e \in \mathcal{T}_i\}$ and define the intermediate activation and its gradient:

$$z_{i,e} = x_i A_e, \qquad \delta z_{i,e} = s g_{i,e} q_i B_e^\top. \tag{8}$$

Both are r-dimensional vectors for one selected token-expert pair. Differentiating the original MoLE formulation gives

$$\nabla_{A_e}\mathcal{L} = \sum_{i\in\mathcal{I}_e} x_i^\top \delta z_{i,e}, \qquad \nabla_{B_e}\mathcal{L} = s \sum_{i\in\mathcal{I}_e} g_{i,e} z_{i,e}^\top q_i, \qquad h_{i,e} \equiv \frac{\partial\mathcal{L}}{\partial g_{i,e}} = s\langle z_{i,e}, q_i B_e^\top\rangle. \tag{9}$$

The first factor needs complete δz, the second needs complete z, and the mixing-weight derivative uses the same low-rank vectors. This allows one reduced tensor to support multiple updates instead of synchronizing the corresponding parameter-gradient matrices.

Up: reduce activation gradients. The complete $z_{i,e}$ is cached, but GPU p receives only the output gradient coordinates $q_i^{(p)}$ matching $B_e^{(p)}$. We compute

$$\delta z_{i,e}^{(p)} = s g_{i,e} q_i^{(p)} (B_e^{(p)})^\top, \qquad \delta z_{i,e} = \mathrm{AR}(\delta z_{i,e}^{(p)}). \tag{10}$$

The sum supplies the complete gradient of replicated A_e and the expert contribution to the input gradient. The local $B_e^{(p)}$ gradient uses cached $z_{i,e}$ and $q_i^{(p)}$. For positive selected weights, the same reduction gives $h_{i,e} = \langle z_{i,e}, \delta z_{i,e} \rangle / g_{i,e}$. Thus, router differentiation requires no separate exchange of expert outputs or mixing-weight derivatives. Appendix B.5 discusses finite-precision recovery.

Down: reduce cached activations. Each partial output receives the same q_i. Because B_e is replicated, $q_i B_e^\top$ and $\delta z_{i,e}$ are already complete locally. The replicated B_e update instead requires

$$z_{i,e} = \mathrm{AR}(z_{i,e}^{(p)}). \tag{11}$$

The summed activation supplies both $\nabla_{B_e}\mathcal{L}$ and $h_{i,e}$ in Eq. 9. The partitioned $A_e^{(p)}$ gradient uses local $x_i^{(p)}$ and complete $\delta z_{i,e}$. Both sites reuse cached activations without replaying expert forward products inside the cached adapter backward.

For $P > 1$, one backward all-reduce batches the $K = Tk$ selected pairs at each adapted projection, with Kr input elements per GPU. The resulting mixing-weight derivatives pass through the selected-score softmax and linear router. Input gradients retain the base, expert, and router paths: up adds its already complete expert and router contributions after the base input-gradient reduction, while down keeps all contributions in their input-coordinate partitions. Replicated factors consequently receive complete gradients without another TP parameter-gradient reduction. Appendix B derives the full chain rule and the conditions for consistent optimizer updates.

3.4 Grouped Forward and Backward Computation

Form expert batches with their actual token counts. For each invocation, expert e receives $T_e = |\mathcal{I}_e|$ token rows, with $\sum_e T_e = K$. We gather these rows into a contiguous expert block and retain their original token indices and routing weights. A token selected by several experts appears once in each corresponding block. Offsets $o_0 = 0$, $o_{e+1} = o_e + T_e$ locate the blocks in a packed buffer. This organization preserves unequal T_e rather than padding all expert batches to a common length. Up and down construct independent groupings.

Reuse the grouping across both forward products. Using existing grouped-matmul support, we evaluate each nonempty expert block with its own factors:

$$\begin{aligned} \text{up:} \quad & Z_{u,e} = X_e A_{u,e}, \qquad O_{u,e}^{(p)} = Z_{u,e} B_{u,e}^{(p)}, \\ \text{down:} \quad & Z_{d,e}^{(p)} = M_e^{(p)} A_{d,e}^{(p)}, \quad O_{d,e}^{(p)} = Z_{d,e}^{(p)} B_{d,e}. \end{aligned} \tag{12}$$

The first product changes the representation width, not the token rows, so the second reuses the same block boundaries. If packed row ℓ belongs to token t_ℓ with weight $\widehat{g}_\ell$, a weighted scatter-add forms

$$\Delta Y_{\mathrm{loc}}[i,:] = s \sum_{\ell : t_\ell = i} \widehat{g}_\ell O_{\mathrm{pack}}^{(p)}[\ell,:]. \tag{13}$$

This produces the up correction $\Delta U^{(p)}$ or down correction $\Delta Y_{\mathrm{part}}^{(p)}$ in the token order of the base output. The grouped products and scatter remain separate operations.

Reuse the same token selections in backward. Saved token indices gather the corresponding output-gradient rows. For an expert block, let Q_e contain these rows and $H_e = s\,\mathrm{diag}(g_e) Q_e$ include the routing weights. The complete-block derivatives are

$$\begin{aligned} \nabla_{B_e}\mathcal{L} &= Z_e^\top H_e, \qquad \delta Z_e = H_e B_e^\top, \\ \nabla_{A_e}\mathcal{L} &= X_e^\top \delta Z_e, \quad \delta X_e = \delta Z_e A_e^\top. \end{aligned} \tag{14}$$

Here δX_e is the expert branch's input-gradient contribution. We group these products using the same expert boundaries, substituting local factor partitions and the sums from Section 3.3. In parameter-gradient products, T_e is the contraction dimension even when gradient output shapes agree. Input gradients return to their original token rows without a second routing-weight multiplication. Low-rank all-reduces align rows belonging to the same selected token-expert pair across GPUs. Appendix C gives the exact indexing, and Appendix C.4 provides a worked example.

3.5 Training Equivalence and Resource Costs

Proposition 3.1 (Equivalence to the original MoLE formulation)**.** *Under exact arithmetic, identical model inputs and parameters, consistent ordered expert selections and weights, and the TP loss conventions in Appendix B.1, ParaMoLE preserves the original MoLE outputs and expert, router, and input gradients. At a top-k boundary, the statement concerns the selected computation with expert indices held fixed.*

Appendix B.3 proves the result, including the connection between the two adapted projections. Finite-precision agreement is treated separately. For one up/down pair with the same E and r, the per-GPU expert parameter count and its fraction of full replication are

$$N_{\text{expert}} = 2Er(D + F/P), \qquad \rho_P = \frac{D + F/P}{D + F}. \tag{15}$$

For $P > 1$, $1/P < \rho_P < 1$: partial replication retains more expert state than equal full sharding but less than full replication. Gradients and optimizer states follow this ownership under matched storage conventions. Router states, cached activations, and workspaces contribute separately to peak memory.

For $P > 1$, one up/down forward/backward invocation without replay adds a TE-element down-router all-reduce and two Tkr-element backward all-reduces beyond the base operations. At equal precision, the total additional collective input is $T(E+2kr)$ elements per GPU. At $P = 1$, these cross-GPU collectives are unnecessary. Useful expert forward work per GPU is $4Tkr(D + F/P)$ FLOPs. The same expert batch sizes and equal feature partitions give equal nominal expert work across GPUs, but batch shapes still affect kernel efficiency. Retained factor copies also repeat computation, and caching consumes memory. Appendix D derives these trade-offs rather than predicting complete-step time or peak memory from one resource count.

4 Experiment

4.1 Setup

Models and tasks. We evaluate training efficiency on Qwen2.5-3B, Llama-3.2-3B, and Qwen2.5-14B, and downstream quality on the two 3B Base models. The five tasks are sentiment classification on SST-2, question answering on SQuAD, mathematical reasoning on GSM8K, and summarization on XSum and CNN/DailyMail. We report accuracy for SST-2, exact match (EM) and token-level F1 for SQuAD, answer EM for GSM8K, and ROUGE-1/2/L for summarization.

Training configuration. The main experiments use one node with four NVLink-connected H100 GPUs (94 GiB), BF16 computation, and AdamW with a learning rate of 10^{-4}. We freeze the backbone and train an independent expert pool and router at every FFN up and down projection. Unless varied, each pool uses $E = 64$ experts, top-$k = 2$ routing, and $r = \alpha = 16$. The global batch size is $b_{\text{g}} = 64$. The sequence budgets L_{seq} are 128, 512, 512, 1024, and 2048 for the five datasets above, with corresponding micro-batch sizes b_μ of 16, 8, 8, 4, and 2.

Execution strategies and metrics. We compare Full Replication (FR), Fully Sharded (FS), Expert Parallelism (EP), and ParaMoLE. FR replicates expert training states on every GPU. FS shards these states across GPUs. EP assigns complete experts to GPUs and dispatches tokens to the selected experts. ParaMoLE uses the partially replicated factor layout, low-rank synchronization, and grouped computation in Section 3. Efficiency is measured by complete training-step time and peak allocated GPU memory. We also report communication payloads in the ablation and sensitivity experiments.

4.2 Memory Feasibility and Training Efficiency

Table 1 compares memory feasibility and complete-step efficiency under the main batch settings. ParaMoLE completes the measured Qwen2.5-14B training steps on all five datasets, using 30.84-41.64 GiB per GPU. The evaluated EP configuration encounters OOM on every dataset, while FR encounters OOM on four datasets. FS also completes all five configurations, but ParaMoLE uses less peak memory and is 5.39-8.13× faster. On XSum, for example, ParaMoLE takes 9.258 s per step at 41.64 GiB per GPU, compared with 54.193 s and 52.87 GiB for FS. These results demonstrate memory feasibility for the tested 14B workloads without the long steps observed in the FS implementation.

Table 1: Complete-step time and peak allocated memory on four H100 GPUs, using BF16, E=64, k=2, r=16, and global batch size $b_{\mathrm{g}} = 64$. Sequence budgets are shown in parentheses; micro-batches are 16, 8, 8, 4, and 2 in dataset order.

Model	Method	SST-2 (128)		SQuAD (512)		GSM8K (512)		XSum (1024)		CNN/DM (2048)	
		Step (s)	Mem. (GiB)	Step (s)	Mem. (GiB)	Step (s)	Mem. (GiB)	Step (s)	Mem. (GiB)	Step (s)	Mem. (GiB)
Qwen2.5-3B	FR	8.228	40.58	25.521	61.87	24.891	61.87	51.319	62.49	101.306	61.86
	FS	8.141	14.48	19.404	22.44	20.219	22.43	39.226	23.04	83.756	22.40
	EP	**0.989**	42.77	3.078	76.35	3.076	75.14	5.989	78.29	13.015	73.65
	ParaMoLE	1.001	**11.61**	**2.215**	**15.61**	**2.205**	**15.61**	**4.637**	**16.23**	**9.993**	**15.57**
Llama-3.2-3B	FR	6.710	27.86	19.084	42.33	18.402	42.33	37.913	43.03	77.903	41.78
	FS	6.164	10.54	15.821	15.91	16.031	15.91	31.459	16.60	63.375	15.34
	EP	**0.699**	29.33	2.257	51.82	2.228	51.39	4.500	54.18	10.058	49.66
	ParaMoLE	0.756	**9.48**	**1.675**	**12.28**	**1.683**	**12.28**	**3.517**	**12.95**	**8.065**	**11.71**
Qwen2.5-14B	FR	13.796	85.14	/	OOM	/	OOM	/	OOM	/	OOM
	FS	11.294	34.96	28.722	50.91	27.141	50.90	54.193	52.87	116.994	49.06
	EP	/	OOM	/	OOM	/	OOM	/	OOM	/	OOM
	ParaMoLE	**1.390**	**30.84**	**4.435**	**39.73**	**4.361**	**39.73**	**9.258**	**41.64**	**21.705**	**37.87**

Table 2: The downstream quality on two 3B models and five datasets.

Model	Method	SST-2	SQuAD		GSM8K	XSum			CNN/DM		
		Acc.	EM	F1	EM	R-1	R-2	R-L	R-1	R-2	R-L
Qwen2.5-3B	FR	95.60	89.00	93.54	68.52	34.20	13.88	27.26	41.10	18.77	27.68
	FS	95.80	89.40	93.72	69.51	33.99	13.80	27.05	40.78	18.33	27.39
	EP	95.60	88.80	93.57	70.01	33.54	13.65	26.78	40.74	18.36	27.41
	ParaMoLE	95.80	89.80	94.03	69.89	33.95	13.88	27.02	40.78	18.68	27.49
Llama-3.2-3B	FR	93.80	86.00	92.28	37.41	33.31	12.22	24.90	39.03	16.74	25.46
	FS	93.80	86.20	92.62	37.68	33.45	12.50	24.84	38.92	16.77	25.52
	EP	93.00	86.20	92.16	37.22	33.19	12.27	24.96	38.74	16.46	25.39
	ParaMoLE	93.60	86.40	92.48	37.46	33.27	12.20	25.00	38.99	16.55	25.45

On the two 3B backbones, ParaMoLE reduces peak memory by 66.0-74.8% relative to FR and 67.7-79.6% relative to EP. ParaMoLE is also 1.25-1.40× faster than EP on SQuAD, GSM8K, XSum, and CNN/DailyMail, while EP remains slightly faster on SST-2. Thus, the main configurations show consistent memory savings over FR and EP, but the speed comparison with EP depends on the task.

4.3 Downstream Quality

Table 2 evaluates whether the execution changes preserve downstream quality after fine-tuning. On Qwen2.5-3B, ParaMoLE matches the highest SST-2 accuracy of 95.80% and obtains 89.80/94.03 SQuAD EM/F1. ParaMoLE reaches 69.89% GSM8K EM, within the baseline range of 68.52-70.01%. On Llama-3.2-3B, ParaMoLE achieves 37.46% GSM8K EM, compared with 37.22-37.68% for the baselines, and remains close to the baseline results on SST-2 and SQuAD. Across both backbones and both summarization datasets, ParaMoLE's ROUGE scores differ from EP's by at most 0.41 points. The reported results show no systematic quality penalty across the evaluated tasks. This empirical comparison is separate from the exact-arithmetic equivalence established in Proposition 3.1.

4.4 Sensitivity Analysis

Figure 3 varies expert count E, top-k, LoRA rank r, and sequence budget L_{seq} on Qwen2.5-3B with XSum. Increasing E or r raises the memory cost of the expert pool. ParaMoLE retains short training steps, whereas the FR and FS implementations remain substantially slower. FR additionally encounters OOM at the largest expert count. EP provides competitive step times, but uses more peak memory and communication payload in these Qwen2.5-3B configurations.

The top-k sweep shows a pronounced increase in EP's memory and payload, while ParaMoLE's measured step time and peak memory change little. Longer sequences increase ParaMoLE's memory and payload, but ParaMoLE retains the lowest values for these two metrics. The small changes in some curves describe this measurement regime, not independence from k, r, or L_{seq}. Appendix F.2 reports the corresponding Llama-3.2-3B results. Appendices F.3 and F.4 further examine how micro-batch size and GPU count affect throughput and memory feasibility.

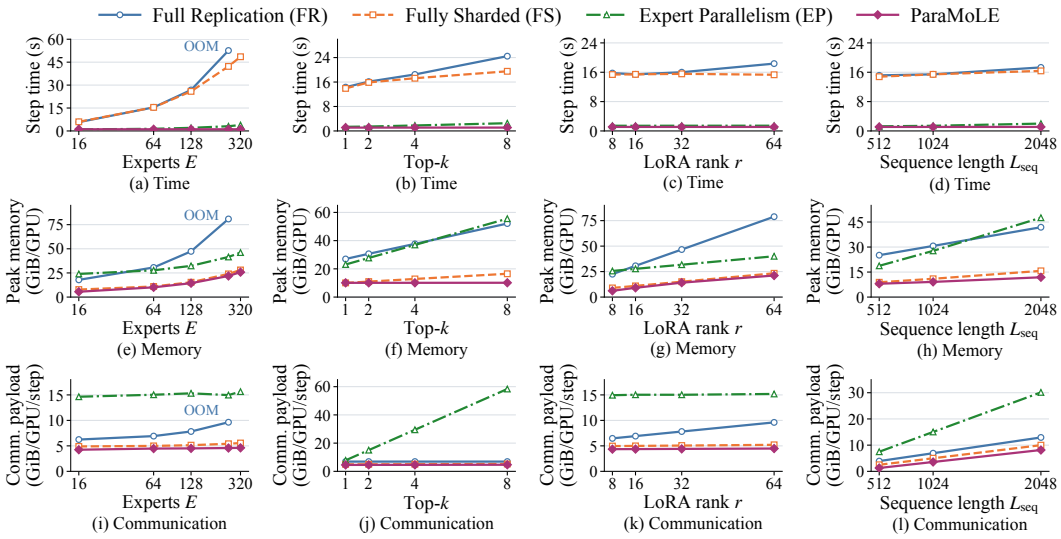


Figure 3: Sensitivity analysis on Qwen2.5-3B and XSum. (a,e,i) vary expert count E, (b,f,j) vary routing top-k, (c,g,k) vary LoRA rank r, and (d,h,l) vary sequence length L_{seq}. Each group reports complete training-step time, peak allocated memory per GPU, and communication payload per GPU per step. E=64, k=2, r=16, L_{seq}=1024, micro-batch size b_μ=1, and global batch size b_{g}=8.

Table 3: Ablation of ParaMoLE on Qwen2.5-3B with XSum. We use TP=4, $E = 64$, $r = 16$, top-$k = 2$, sequence length $L_{\text{seq}} = 1024$, micro-batch size $b_\mu = 4$, and global batch size $b_{\text{g}} = 64$.

Method	**Step time**	**Peak memory**	**Commu. (GiB/GPU/step)**	
	(s)↓	**(GiB/GPU)↓**	**MoLE↓**	**+ Base↓**
Full Replication	51.319	62.49	12.999	52.70
Fully Sharded	39.226	23.04	7.43	43.86
ParaMoLE-Loop	21.988	16.23	0.563	32.84
ParaMoLE-GradSync	5.229	16.24	9.844	42.12
ParaMoLE	**4.637**	**16.23**	**0.563**	**32.84**

4.5 Ablation Study

Table 3 evaluates grouped computation and low-rank backward synchronization on Qwen2.5-3B with XSum. **ParaMoLE-Loop** replaces grouped GEMM with per-expert forward and backward products, retaining low-rank synchronization. **ParaMoLE-GradSync** retains grouped GEMM but synchronizes replicated-factor parameter gradients instead of low-rank tensors, with complete router and input gradients (Appendix E.1). Both variants preserve ParaMoLE's factor layout and forward output aggregation.

Compared with Loop, ParaMoLE reduces step time from 21.988 to 4.637 s, a 4.74× speedup, without changing peak memory or communication payload. Compared with GradSync, ParaMoLE reduces step time by 11.3% and MoLE payload from 9.844 to 0.563 GiB per GPU per step, a 94.3% reduction, with nearly unchanged peak memory. Grouped computation yields the larger speedup in this configuration, while low-rank synchronization further reduces communication and runtime.

5 Conclusion

We presented **ParaMoLE** for memory-efficient MoLE fine-tuning on tensor-parallel backbones. ParaMoLE combines low-rank synchronization for joint expert and router updates with grouped forward and backward computation over variable-sized expert batches. We established equivalence to the original MoLE outputs and gradients in exact arithmetic. Experiments demonstrate a favorable balance among training time, memory, and communication, with comparable downstream quality on 3B models and improved memory feasibility for the tested 14B configurations.

AI USE STATEMENT

We used large language models to polish the writing, including grammar correction, style harmonization, and sentence-level revisions. All suggested edits were reviewed by the authors, who take full responsibility for the final content.

REPRODUCIBILITY STATEMENT

The main paper describes the datasets, base models, and evaluation metrics used in our experiments. All datasets are publicly available. We describe the method's execution and the experimental settings reported in the main paper and appendices. We will release the code upon acceptance to facilitate reproduction of our results.

REFERENCES

Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems*, 6:1–13, 2024.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

Shihan Dou, Enyu Zhou, Yan Liu, Songyang Gao, Wei Shen, Limao Xiong, Yuhao Zhou, Xiao Wang, Zhiheng Xi, Xiaoran Fan, et al. Loramoe: Alleviating world knowledge forgetting in large language models via moe-style plugin. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1932–1945, 2024.

Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. Megablocks: Efficient sparse training with mixture-of-experts. *Proceedings of Machine Learning and Systems*, 5:288–304, 2023.

Chongyang Gao, Kezhen Chen, Jinmeng Rao, Ruibo Liu, Baochen Sun, Yawen Zhang, Daiyi Peng, Xiaoyuan Guo, and VS Subrahmanian. Mola: Moe lora with layer-wise expert allocation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 5112–5127, 2025.

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 *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Dawid Kopiczko, Tijmen Blankevoort, and Yuki Asano. Vera: Vector-based random matrix adaptation. In *International Conference on Learning Representations*, volume 2024, pp. 6815–6835, 2024.

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 *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Dengchun Li, Yingzi Ma, Naizheng Wang, Zhengmao Ye, Zhiyuan Cheng, Yinghao Tang, Yan Zhang, Lei Duan, Jie Zuo, Cal Yang, et al. Mixlora: Enhancing large language models fine-tuning with lora-based mixture of experts. *arXiv preprint arXiv:2404.15159*, 2024.

Kevin Li, Dibyadeep Saha, Avni Kanodia, and Fan Lai. tlora: Efficient multi-lora training with elastic shared super-models. *arXiv preprint arXiv:2602.07263*, 2026.

Mengqi Liao, Wei Chen, Junfeng Shen, Shengnan Guo, and Huaiyu Wan. Hmora: Making llms more effective with hierarchical mixture of lora experts. In *International Conference on Learning Representations*, volume 2025, pp. 91309–91330, 2025.

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. *arXiv preprint arXiv:2402.09353*, 2024.

Zefang Liu and Jiahua Luo. Adamole: Fine-tuning large language models with adaptive mixture of low-rank adaptation experts. *arXiv preprint arXiv:2405.00361*, 2024.

Fanxu Meng, Zhaohui Wang, and Muhan Zhang. Pissa: Principal singular values and singular vectors adaptation of large language models. *Advances in Neural Information Processing Systems*, 37: 121038–121072, 2024.

Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Xinyu Wang, Jonas Kübler, Kailash Budhathoki, Yida Wang, and Matthäus Kleindessner. Block-diagonal lora for eliminating communication overhead in tensor parallel lora serving. *Advances in Neural Information Processing Systems*, 38:256–327, 2026.

Xun Wu, Shaohan Huang, and Furu Wei. Mixture of lora experts. *arXiv preprint arXiv:2404.13628*, 2024.

Minghao Yan, Zhuang Wang, Zhen Jia, Shivaram Venkataraman, and Yida Wang. PLoRA: Efficient concurrent loRA training for large language models. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=khYw62jVZ2.

Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. Lora-fa: Memory-efficient low-rank adaptation for large language models fine-tuning. *arXiv e-prints*, pp. arXiv–2308, 2023a.

Qingru Zhang, Minshuo Chen, Alexander Bukharin, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adaptive budget allocation for parameter-efficient fine-tuning. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=lq62uWRJjiY.

Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, et al. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. *Proceedings of Machine Learning and Systems*, 7, 2025.

Yun Zhu, Nevan Wichers, Chu-Cheng Lin, Xinyi Wang, Tianlong Chen, Lei Shu, Han Lu, Canoee Liu, Liangchen Luo, Jindong Chen, et al. Sira: Sparse mixture of low rank adaptation. *arXiv preprint arXiv:2311.09179*, 2023.

Zhanda Zhu, Qidong Su, Yaoyao Ding, Kevin Song, Shang Wang, and Gennady Pekhimenko. Lorafusion: Efficient lora fine-tuning for llms. In *Proceedings of the 21st European Conference on Computer Systems*, pp. 586–604, 2026.

Jingwei Zuo, Xinze Feng, Zien Liu, Kaijian Wang, Fanjiang Ye, Ye Cao, Zhuang Wang, and Yuke Wang. Alto: Adaptive lora tuning and orchestration for heterogeneous lora training workloads. *arXiv preprint arXiv:2604.05426*, 2026.

A Related Work

LoRA and mixtures of LoRA experts. LoRA Hu et al. (2022) adapts pretrained models through low-rank weight updates while keeping the backbone frozen. Subsequent methods improve the allocation, representation, and training cost of the low-rank parameters. AdaLoRA Zhang et al. (2023b) allocates the parameter budget across weight matrices, while DoRA Liu et al. (2024) separates weight magnitude from direction. PiSSA Meng et al. (2024) initializes the trainable factors from principal singular components of pretrained weights. QLoRA Dettmers et al. (2023) reduces backbone memory through quantization, VeRA Kopiczko et al. (2024) learns scaling vectors over shared frozen random matrices, and LoRA-FA Zhang et al. (2023a) reduces activation storage by

freezing the input projection. The improvements address different sources of cost within individual adapters.

Mixtures of LoRA experts extend adaptation through conditional combinations of low-rank modules Zhu et al. (2023); Wu et al. (2024). LoRAMoE Dou et al. (2024) encourages expert specialization to mitigate knowledge forgetting. MixLoRA Li et al. (2024) constructs sparse LoRA-based experts within frozen FFN blocks and optimizes shared-backbone execution during training and inference. Further designs refine how experts are organized and selected. MoLA Gao et al. (2025) allocates experts across Transformer layers, HMoRA Liao et al. (2025) combines token-level and task-level routing, and AdaMoLE Liu & Luo (2024) adjusts expert activation through adaptive routing thresholds. ParaMoLE studies the distributed execution of a specified token-routed LoRA mixture without changing the expert parameterization or routing rule.

Efficient distributed training and inference. Concurrent training systems share a backbone across independent adapter jobs. PLoRA Yan et al. (2026) combines automatic job orchestration with optimized adapter kernels. LoRAFusion Zhu et al. (2026) reduces activation-memory accesses through kernel fusion and uses dependency-aware batching for concurrent jobs. tLoRA Li et al. (2026) groups adapters into elastic shared super-models and supports distributed model execution, including tensor parallelism. ALTO Zuo et al. (2026) combines grouped forward and backward computation with early termination and scheduling of heterogeneous LoRA jobs. In contrast to independent-job batching, ParaMoLE coordinates experts that contribute to the same token output. Joint expert and router training consequently requires recovering both expert gradients and derivatives of the learned mixing weights across the TP group.

Parallel model execution and LoRA inference provide complementary building blocks. Megatron-LM Shoeybi et al. (2019) partitions Transformer layers across GPUs, MegaBlocks Gale et al. (2023) uses block-sparse kernels for unequal expert batches without token dropping, and COMET Zhang et al. (2025) overlaps expert computation with communication. For LoRA inference, Punica Chen et al. (2024) batches computations across adapters. S-LoRA Sheng et al. (2023) combines heterogeneous adapter batching with TP layouts that partition both factors and exchange intermediate activations. The partially replicated implementation in vLLM Kwon et al. (2023) instead retains one complete factor on each GPU, allowing both LoRA products to remain local. BD-LoRA Wang et al. (2026) removes additional adapter communication by imposing block-diagonal constraints on selected factors. ParaMoLE adopts partial factor replication and existing grouped matrix multiplication, then reuses low-rank reductions for joint expert and router updates while preserving standard LoRA factors and the specified routing rule.

B Training Derivations and Equivalence

B.1 Notation and Training Conventions

Table 4 summarizes the notation. Generic layer formulas use X, Y, W, A_e, B_e, R for one adapted linear layer. For up, its input is the FFN input X; for down, it is M. Subscripts u, d are restored when both layers appear together. Token representations and their gradients are row vectors, $Q_V = \partial\mathcal{L}/\partial V$, and token, expert, and GPU indices are zero-based. The symbol r always denotes the LoRA rank. Experimental quantities use b_{g} and b_μ for global and micro-batch sizes, respectively, L_{seq} for the sequence budget, τ for step time, M_{peak} for peak memory, C_{step} for communication payload, and V_{tok} for token throughput.

Model and routing. We analyze Eq. 3 with fixed s, finite inputs and parameters, selected-score softmax, and no auxiliary routing objective or adapter dropout. Each token selects k distinct experts. Parameter partitions are disjoint and reconstruct the original matrices; parameter copies agree. All TP GPUs process the same token rows and use the same ordered expert IDs and weights. An empty expert batch contributes zero task-loss derivatives. At a top-k membership boundary, derivatives refer to the chosen computation with selected indices fixed. Other branch choices and stochastic realizations are matched when executions are compared.

One model loss across the TP group. Replicated tensors represent one model evaluation, rather than P independent losses. For a column-partitioned output, $Y = [Y^{(0)}, \ldots, Y^{(P-1)}]$, its gradient

Table 4: Notation used in the method and appendices.

Symbol	Meaning
$T, E, k, K = Tk$	Token rows per microbatch, experts per adapted projection, selected experts per token, and selected token-expert pairs.
r, α, s	LoRA rank, scaling parameter, and $s = \alpha/r$.
P, p	TP GPU count and GPU index $p \in \{0, \dots, P-1\}$.
D, F	Backbone hidden width and FFN intermediate width.
$d_{\text{in}}, d_{\text{out}}$	Input and output widths of one adapted linear layer.
X, Y, W	Layer input, layer output, and frozen base weight.
A_e, B_e	Expert factors of shapes $d_{\text{in}} \times r$ and $r \times d_{\text{out}}$.
$R, L = XR$	Router parameter matrix and complete expert scores.
$\mathcal{T}_i, g_{i,e}$	Experts selected for token i and their mixing weights.
$G^{(p)}, U^{(p)}, M^{(p)}$	FFN gate output, corrected up output, and down input; each has shape $T \times (F/P)$.
$\mathcal{I}_e, T_e$	Tokens selecting expert e and their number, with $\sum_e T_e = K$.
Z_e, O_e	Intermediate activation $X_e A_e$ and unweighted expert output $Z_e B_e$.
Q, q_i	Gradient of the complete layer output and its token-i row.
D_e, H_e	$\text{diag}(g_e)$ and weighted output-gradient rows sD_eQ_e.
$\delta Z_e, \delta X_e$	Gradient of Z_e and the expert branch's input-gradient contribution.
$h_{i,e}, \Gamma$	Derivatives with respect to $g_{i,e}$ and complete scores L.
$J_{i,j}, \nu = ik + j$	Selected expert in slot j and the corresponding canonical pair index.
$t_\ell, e_\ell, \widehat{g}_\ell$	Token ID, expert ID, and mixing weight of packed row ℓ.
$o_e, \pi(\ell)$	Expert block offset and canonical index of packed row ℓ.
$\mathcal{P}_\pi, P_\pi$	Canonical-to-expert permutation operator and its matrix.
S_e, C	Expert row-gather matrix and token-expansion matrix.
$\text{AR}, \overline{V}$	TP SUM all-reduce and its summed result.
$E_{\text{active}}, T_{\max}$	Number of nonempty expert batches and their largest row count.

has matching columns $Q = [Q^{(0)}, \dots, Q^{(P-1)}]$. For a row-parallel sum $Y = \sum_p \widetilde{Y}^{(p)}$,

$$d\mathcal{L} = \left\langle Q, \sum_p d\widetilde{Y}^{(p)} \right\rangle_F = \sum_p \langle Q, d\widetilde{Y}^{(p)} \rangle_F, \qquad \frac{\partial \mathcal{L}}{\partial \widetilde{Y}^{(p)}} = Q. \tag{16}$$

Each additive contribution therefore receives one copy of the same complete gradient. The router-score sum uses the same convention. Forward values and low-rank contributions are SUM-reduced; the loss normalization matches the original model. This specifies the TP backward mappings assumed below.

B.2 Complete Derivatives of One MoLE Layer

Let $S_e \in \{0,1\}^{T_e \times T}$ gather the token rows selecting expert e in a fixed order. Then $X_e = S_e X$, $Q_e = S_e Q$, and $S_e^\top$ adds expert rows back to their original token positions. With $D_e = \text{diag}(g_e)$, the layer is

$$Y = XW + s \sum_e S_e^\top D_e O_e, \qquad O_e = Z_e B_e, \quad Z_e = X_e A_e. \tag{17}$$

S_e and D_e express indexing and row weighting; they need not be allocated as matrices. Selected indices are held fixed while their weights remain differentiable.

Expert factors and activations. The differential through expert e is

$$d\mathcal{L}\big|_e = s\langle Q_e, D_e\, dO_e + (dD_e)O_e \rangle_F.$$

The first term gives $H_e = sD_eQ_e = \partial\mathcal{L}/\partial O_e$. Since $dO_e = (dZ_e)B_e + Z_e(dB_e)$,

$$\langle H_e, dO_e \rangle_F = \langle H_e B_e^\top, dZ_e \rangle_F + \langle Z_e^\top H_e, dB_e \rangle_F.$$

Applying the same rule to $Z_e = X_e A_e$ yields

$$\begin{aligned} \nabla_{B_e}\mathcal{L} &= Z_e^\top H_e, \qquad \delta Z_e = H_e B_e^\top, \\ \nabla_{A_e}\mathcal{L} &= X_e^\top \delta Z_e, \quad \delta X_e = \delta Z_e A_e^\top. \end{aligned} \tag{18}$$

The expert input derivative already contains the mixing weight and LoRA scale through H_e.

Router parameters and activations. The term containing dD_e gives, for every selected pair,

$$h_{i,e} = s\langle q_i, z_{i,e}B_e\rangle = s\langle z_{i,e}, q_iB_e^\top\rangle. \tag{19}$$

On the selected set, the softmax Jacobian is $\partial g_{i,e}/\partial L_{i,v} = g_{i,e}(\mathbf{1}\{e=v\} - g_{i,v})$. Thus

$$\Gamma_{i,e} \equiv \frac{\partial \mathcal{L}}{\partial L_{i,e}} = \begin{cases} g_{i,e}\left(h_{i,e} - \sum_{v\in\mathcal{T}_i} g_{i,v}h_{i,v}\right), & e \in \mathcal{T}_i, \\ 0, & e \notin \mathcal{T}_i. \end{cases} \tag{20}$$

Differentiating $L = XR$ gives

$$\nabla_R \mathcal{L} = X^\top \Gamma, \qquad \nabla_X \mathcal{L}\big|_{\text{router}} = \Gamma R^\top. \tag{21}$$

The full input gradient combines three paths:

$$\nabla_X \mathcal{L} = QW^\top + \sum_e S_e^\top \delta Z_e A_e^\top + \Gamma R^\top. \tag{22}$$

Freezing W removes its parameter update but not the first term. The zero outside $\mathcal{T}_i$ in Eq. 20 is specific to the stated loss and normalization. In particular, $k = 1$ gives selected weight one and $\Gamma = 0$ on that branch; this is a property of the original router, not of TP execution.

B.3 Distributed Derivatives and FFN Composition

Forward values. At a column-parallel layer, concatenating $W^{(p)}$ and $B_e^{(p)}$ by columns recovers W and B_e. Replicated X, A_e, R provide the same scores, weights, and intermediate activations. The outputs therefore recover Eq. 3 by concatenation. At a row-parallel layer, $X^{(p)}$ contains input columns and $W^{(p)}, A_e^{(p)}, R^{(p)}$ contain the matching rows. Hence

$$\begin{aligned} \sum_p X^{(p)}W^{(p)} = XW, \qquad \sum_p X^{(p)}R^{(p)} = XR, \\ \sum_p (X_e^{(p)}A_e^{(p)})B_e = (X_eA_e)B_e. \end{aligned} \tag{23}$$

Summing scores before selection and adding expert contributions before the base output SUM preserve both routing and layer outputs.

Column-parallel derivatives. Write $Q = [Q^{(0)}, \ldots, Q^{(P-1)}]$ and $B_e = [B_e^{(0)}, \ldots, B_e^{(P-1)}]$, with $H_e^{(p)} = sD_eQ_e^{(p)}$. The reduced tensor is

$$\sum_p H_e^{(p)}(B_e^{(p)})^\top = sD_eQ_eB_e^\top = \delta Z_e. \tag{24}$$

Substitution into the original derivatives gives

$$\begin{aligned} \nabla_{B_e^{(p)}}\mathcal{L} = Z_e^\top H_e^{(p)}, \\ \nabla_{A_e}\mathcal{L} = X_e^\top \delta Z_e, \quad \nabla_R \mathcal{L} = X^\top \Gamma. \end{aligned} \tag{25}$$

The first is exactly the local column partition of the original B_e gradient. The other two are complete gradients for copied parameters. Because $\langle z_{i,e}, \delta z_{i,e}\rangle = g_{i,e}h_{i,e}$, positive selected weights also recover the original h and Γ. The input gradient is

$$\begin{aligned} \nabla_X \mathcal{L} = \ &\mathrm{AR}\Big(Q^{(p)}(W^{(p)})^\top\Big) \\ &+ \sum_e S_e^\top \delta Z_e A_e^\top + \Gamma R^\top. \end{aligned} \tag{26}$$

Only the base term still needs summation. Putting the complete expert or router term inside that SUM would multiply it by P.

Row-parallel derivatives. By Eq. 16, each partial output receives Q. The replicated B_e therefore gives complete $\delta Z_e = sD_eQ_eB_e^\top$ locally. Recovering $Z_e = \sum_p Z_e^{(p)} = X_eA_e$ supplies

$$\begin{aligned} \nabla_{B_e}\mathcal{L} &= Z_e^\top(sD_eQ_e), \\ \nabla_{A_e^{(p)}}\mathcal{L} &= (X_e^{(p)})^\top \delta Z_e, \\ \nabla_{R^{(p)}}\mathcal{L} &= (X^{(p)})^\top \Gamma. \end{aligned} \tag{27}$$

Equation 19 uses complete Z_e and local $Q_eB_e^\top$, recovering the same mixing-weight derivatives. Every input-gradient term occupies the local input-coordinate block:

$$\begin{aligned} \nabla_{X^{(p)}}\mathcal{L} = Q(W^{(p)})^\top \\ + \sum_e S_e^\top \delta Z_e (A_e^{(p)})^\top + \Gamma(R^{(p)})^\top. \end{aligned} \tag{28}$$

No additional input-gradient reduction is required at this site.

Connection through the gated FFN. Down backward produces $Q_M^{(p)}$, including its base, expert, and router paths. From Eq. 5,

$$\begin{aligned} Q_U^{(p)} &= Q_M^{(p)} \odot \mathrm{SiLU}(G^{(p)}), \\ Q_G^{(p)} &= Q_M^{(p)} \odot U^{(p)} \odot \mathrm{SiLU}'(G^{(p)}). \end{aligned} \tag{29}$$

With the up-specific selections $S_{u,e}$,

$$\begin{aligned} Q_X = \mathrm{AR}\Big(Q_U^{(p)}(W_{\mathrm{up}}^{(p)})^\top + Q_G^{(p)}(W_{\mathrm{gate}}^{(p)})^\top\Big) \\ + \sum_e S_{u,e}^\top \delta Z_{u,e} A_{u,e}^\top + \Gamma_u R_u^\top. \end{aligned} \tag{30}$$

The unadapted gate branch still transmits gradients, and its derivative uses the corrected up output $U^{(p)}$.

Proof of Proposition 3.1. Equation 23 and column concatenation recover the original scores and forward outputs. Under the common selection convention, both executions select the same experts and weights. Eq. 24-28 then recover the original expert, router, and input derivatives. The grouping identities in Appendix C preserve every selected pair and apply the corresponding gather and scatter adjoints, so they change only the organization of these products. Eq. 29 and 30 connect the adapted projections through the same gated FFN. Composing these identities through the model proves equality of outputs and gradients under the stated branch and TP loss conventions. □

B.4 Optimizer Updates and Replicated Parameters

If parameters and optimizer states initially match, equal gradients give equal subsequent parameters under the same deterministic update, minibatch sequence, learning-rate schedule, and loss normalization. Induction extends this result over successive steps in exact arithmetic. Identically normalized gradient accumulation preserves the argument. Copied parameters receive complete gradients and are updated identically; their TP copies do not require a second gradient average. Empty experts must be handled consistently, because skipping an update can differ from applying a zero task gradient under momentum or weight decay.

Global gradient clipping must count each model parameter coordinate once. Let $\mathcal{S}$ and $\mathcal{R}$ denote the sets of partitioned and replicated trainable tensors, and let $d_\theta^{(p)}$ be their stored parameter gradients. The complete model gradient norm is

$$\|d_{\mathrm{model}}\|_2^2 = \sum_p \sum_{\theta\in\mathcal{S}} \|d_\theta^{(p)}\|_F^2 + \frac{1}{P}\sum_p \sum_{\theta\in\mathcal{R}} \|d_\theta^{(p)}\|_F^2. \tag{31}$$

All GPUs apply the same clipping coefficient. These global operations and synchronization between data-parallel groups, when present, are separate from the layer-local TP reductions. The argument establishes execution equivalence, not a new convergence guarantee.

B.5 Finite-Precision Routing and Gradient Recovery

The described BF16 execution computes local router scores in FP32, communicates down-router scores in the activation dtype, and performs selection and normalization in FP32 after reduction. Converting the reduced scores to FP32 does not reverse earlier rounding. Common expert-index tie-breaking resolves equal scores but does not guarantee agreement between computations with different rounded scores. Cross-GPU route consistency and numerical agreement with the original model must therefore be checked separately from exact-arithmetic identities.

Top-k membership. For $k < E$, order exact scores in descending order using zero-based positions and define $\Delta_i = L_{i,(k-1)} - L_{i,(k)}$. If $\|\widetilde{L}_i - L_i\|_\infty \le \epsilon_i$ and $\Delta_i > 2\epsilon_i$, the selected expert set is unchanged. Indeed, every selected e and unselected v satisfy

$$\widetilde{L}_{i,e} - \widetilde{L}_{i,v} \ge L_{i,e} - L_{i,v} - 2\epsilon_i > 0.$$

For $k = E$, membership is fixed. Equal membership alone does not fix the order of selected slots, which must also agree for pair-aligned collectives.

Mixing weights and local output sensitivity. With an unchanged selected set, align scores by expert ID. If the selected-score perturbation has infinity norm at most ϵ, exact softmax satisfies

$$\|\widetilde{g} - g\|_1 \le \epsilon. \tag{32}$$

To see this, write $\varphi(t) = \mathrm{softmax}(a + tv)$, $\|v\|_\infty \le \epsilon$, and $\mu(t) = \sum_j \varphi_j(t) v_j$. Then $\varphi_j'(t) = \varphi_j(t)(v_j - \mu(t))$ and

$$\|\varphi'(t)\|_1 \le \sqrt{\sum_j \varphi_j(t)(v_j - \mu(t))^2} \le \epsilon.$$

Integrating over $t \in [0, 1]$ gives Eq. 32. If the base and expert outputs are unchanged, then

$$\|\widetilde{y}_i - y_i\|_2 \le |s| C_v \epsilon, \qquad C_v = \max_{e \in \mathcal{T}_i} \|x_i A_e B_e\|_2. \tag{33}$$

These are fixed-input, fixed-selection bounds. They exclude rounding inside softmax itself and do not bound training-trajectory errors.

Weighted activation gradients. For finite logits, selected weights are positive in exact arithmetic. The up path therefore recovers $h_{i,e} = \langle z_{i,e}, \delta z_{i,e} \rangle / g_{i,e}$. Small weights can make this recovery numerically sensitive. The algebraically equivalent expression

$$c_{i,e} = \langle z_{i,e}, \delta z_{i,e} \rangle, \qquad \Gamma_{i,e} = c_{i,e} - g_{i,e} \sum_{v \in \mathcal{T}_i} c_{i,v} \tag{34}$$

avoids explicit division. This identity is an analytical alternative, not a change to the described implementation, and cannot recover contributions already lost through underflow. Down directly contracts complete Z_e with $sQ_eB_e^\top$ to obtain mixing-weight derivatives.

C Grouped Execution and a Worked Example

This appendix specifies how token rows are gathered for expert products, how results return to token order, and how low-rank rows are aligned across GPUs. These are three distinct operations with different adjoints and communication requirements.

C.1 From Token Rows to Expert Blocks

Let $J_{i,j}$ denote the expert in token i's selected slot j, with canonical pair index $\nu = ik + j$. Sort pairs by $(J_{i,j}, \nu)$. If $\pi(\ell)$ is the canonical index of expert-packed row ℓ, then

$$\begin{aligned} (\mathcal{P}_\pi V)[\ell, :] &= V[\pi(\ell), :], \\ t_\ell &= \lfloor \pi(\ell)/k \rfloor, \qquad e_\ell = J_{t_\ell,\, \pi(\ell) \bmod k}, \\ X_{\mathrm{pack}}[\ell, :] &= X_{\mathrm{loc}}[t_\ell, :], \qquad \widehat{g}_\ell = g_{t_\ell, e_\ell}. \end{aligned} \tag{35}$$

X_{loc} denotes the layer input available on a GPU: X at up and $M^{(p)}$ at down. Expert e occupies $[o_e, o_{e+1})$, with $o_0 = 0$ and $o_{e+1} = o_e + T_e$. The same grouping is used for that invocation's backward pass; other layers and later microbatches construct their own selections.

For analysis, let $C \in \{0,1\}^{K \times T}$ expand token rows, with $C_{\nu,i} = \mathbf{1}\{i = \lfloor \nu/k \rfloor\}$, and let P_π be the matrix of $\mathcal{P}_\pi$. With $\widehat{D} = \mathrm{diag}(\widehat{g})$,

$$\begin{aligned} X_{\mathrm{pack}} &= P_\pi C X_{\mathrm{loc}}, \\ \Delta Y_{\mathrm{loc}} &= s C^\top P_\pi^\top \widehat{D} O_{\mathrm{pack}}. \end{aligned} \tag{36}$$

These matrices express indexing, not additional allocated operators. C repeats each token for its selected experts, P_π changes the order of those selected uses, and $C^\top$ accumulates them into the original tokens. Consequently, $P_\pi^\top = P_\pi^{-1}$ but $C^\top C = kI_T$: weighted output accumulation is not simply an inverse permutation.

C.2 Grouped Products and Their Backward Operations

A grouped task records each expert's operands and matrix dimensions. For local input and output widths a, b, expert e has forward products with dimensions $(T_e \times a)(a \times r)$ and $(T_e \times r)(r \times b)$. The output widths change between stages, but the row identities and expert boundaries remain fixed. Up uses $(a, b) = (D, F/P)$; down uses $(F/P, D)$. Each block uses its own factors. Grouping does not imply a common factor or a single fused device kernel for both stages.

For output gradient Q_{loc}, the scatter adjoint gives

$$H_{\mathrm{pack}} = s \widehat{D} P_\pi C Q_{\mathrm{loc}}. \tag{37}$$

These rows drive the four products in Eq. 14, with the local factor partitions and complete low-rank tensors prescribed in Section 3.3. Up reduces its partial δZ_e before computing the copied A_e gradient and expert input gradient. Down reduces cached $Z_e^{(p)}$ before computing the copied B_e gradient and the mixing-weight derivative. The described down path also evaluates the unweighted contraction $sQ_e B_e^\top$ for that derivative. Its arithmetic is included in Appendix D.

The gather adjoint returns expert input gradients by

$$\delta X_{\mathrm{expert}} = C^\top P_\pi^\top \delta X_{\mathrm{pack}}. \tag{38}$$

No second routing-weight multiplication is applied here because δX_{pack} already includes those weights. Cached inputs, low-rank activations, weights, and indices support these products without replaying the two expert forward projections. This statement concerns the cached adapter backward, not arbitrary outer activation-checkpointing policies.

C.3 Aligning Low-Rank Rows Across GPUs

The implementation converts expert-packed rows to canonical (i, j) pair order before the low-rank SUM and restores expert order afterward:

$$\begin{aligned} V_{\mathrm{pair}}^{(p)} &= \mathrm{reshape}_{T,k,r}(\mathcal{P}_\pi^{-1} V_{\mathrm{pack}}^{(p)}), \\ \overline{V}_{\mathrm{pair}} &= \mathrm{AR}(V_{\mathrm{pair}}^{(p)}), \\ \overline{V}_{\mathrm{pack}} &= \mathcal{P}_\pi \, \mathrm{reshape}_{K,r}(\overline{V}_{\mathrm{pair}}). \end{aligned} \tag{39}$$

Here $V = \delta Z$ at up and $V = Z$ at down. Only the middle operation communicates. It sums the same $(i, j, :)$ entry across GPUs; the k selected experts of a token remain separate. Dependent products wait for the reduction result.

Lemma C.1 (Grouping and pair alignment preserve the computation)**.** *For fixed consistent selections, the gather, expert products, and weighted scatter preserve the original selected expert mixture and its derivatives. If all GPUs use the same permutation, then*

$$P_\pi \left(\sum_p P_\pi^\top V_{\mathrm{pack}}^{(p)} \right) = \sum_p V_{\mathrm{pack}}^{(p)}. \tag{40}$$

Proof. Every selected pair appears once after token expansion and once after permutation, uses the same factors and mixing weight, and returns to the same token under $C^\top P_\pi^\top$. Eq. 37 and 38 are the corresponding adjoints. The reduction identity follows from linearity and $P_\pi P_\pi^\top = I_K$. □

Thus, a common expert-packed order also permits direct reduction. Explicit pair alignment is the described implementation choice, not an additional mathematical requirement when row identities already agree. Its local movement cost is retained. Reordering cannot correct disagreement about which expert a pair represents.

C.4 Worked Example

Fixed top-k, unequal expert batches. Consider four tokens, four experts, and top-2 routing. The following selections are illustrative, not measurements. Slots are displayed in expert-ID order.

Table 5: An illustrative top-2 selection with unequal expert batches.

Token	Selected slot 0	Selected slot 1
t_0	e_0	e_1
t_1	e_0	e_1
t_2	e_0	e_2
t_3	e_0	e_1

Expert token sets are $\mathcal{I}_0 = \{0,1,2,3\}$, $\mathcal{I}_1 = \{0,1,3\}$, $\mathcal{I}_2 = \{2\}$, and $\mathcal{I}_3 = \varnothing$. Their sizes are $(4,3,1,0)$, with $K = Tk = 8$. Every token selects two experts; it is the expert batch size, not the number of experts per token, that varies.

Gathering into expert order. Let $X_{\mathrm{loc}} = [x_0; x_1; x_2; x_3]$. The canonical pair list is

$$[(t_0,e_0),(t_0,e_1),(t_1,e_0),(t_1,e_1),\\ (t_2,e_0),(t_2,e_2),(t_3,e_0),(t_3,e_1)].$$

Sorting by expert and canonical pair index gives

$$\pi = (0,2,4,6,1,3,7,5), \qquad (t_\ell)_{\ell=0}^{7} = (0,1,2,3,0,1,3,2). \tag{41}$$

The nonempty expert inputs are

$$X_0 = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}, \qquad X_1 = \begin{bmatrix} x_0 \\ x_1 \\ x_3 \end{bmatrix}, \qquad X_2 = [x_2]\,. \tag{42}$$

Their offsets are $(o_0,o_1,o_2,o_3,o_4) = (0,4,7,8,8)$. Weights follow the same order:

$$\widehat{g} = (g_{0,0}, g_{1,0}, g_{2,0}, g_{3,0}, g_{0,1}, g_{1,1}, g_{3,1}, g_{2,2})^\top.$$

The expert ID determines the factor, the token ID determines the output destination, and the weight determines the mixture contribution. The empty expert has no products, but its parameter and optimizer states follow Appendix B.4.

Variable-sized products. For local factors of shapes $a \times r$ and $r \times b$, the first stage produces blocks of shapes $4 \times r$, $3 \times r$, and $1 \times r$; the second produces $4 \times b$, $3 \times b$, and $1 \times b$. Both stages use the same offsets. Padding the three active batches to four rows would allocate twelve row positions for eight real pairs. This is logical batch padding; kernel tile padding or boundary masks can still exist. The ratio $12/8$ is not a measured speedup.

Numerical projection and output accumulation. Treat the displayed routes as supplied to the expert operator; the example does not specify a router producing them. Set $a = b = 2$, $r = s = 1$, and choose

$$X_{\mathrm{loc}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 2 & 1 \end{bmatrix}, \tag{43}$$

$$A_0 = \begin{bmatrix}1\\1\end{bmatrix}, \quad A_1 = \begin{bmatrix}1\\-1\end{bmatrix}, \quad A_2 = \begin{bmatrix}2\\0\end{bmatrix}, \\ B_0 = [1 \quad 2], \quad B_1 = [-1 \quad 1], \quad B_2 = [3 \quad 0]. \tag{44}$$

The two grouped stages give

$$Z_0 = \begin{bmatrix}1\\1\\2\\3\end{bmatrix}, \quad Z_1 = \begin{bmatrix}1\\-1\\1\end{bmatrix}, \quad Z_2 = [2], \tag{45}$$

$$O_0 = \begin{bmatrix}1 & 2\\1 & 2\\2 & 4\\3 & 6\end{bmatrix}, \quad O_1 = \begin{bmatrix}-1 & 1\\1 & -1\\-1 & 1\end{bmatrix}, \quad O_2 = [6 \quad 0]. \tag{46}$$

Token t_2 appears in row 2 of expert e_0 and row 0 of expert e_2, which are packed rows 2 and 7. With supplied weights $g_{2,0} = 0.25$ and $g_{2,2} = 0.75$,

$$\Delta y_2 = 0.25[2, 4] + 0.75[6, 0] = [5, 1]. \tag{47}$$

This is the direct weighted mixture for t_2. Eight pair outputs become four token corrections through addition, not merely a permutation. The factors above are the local operands for this projection example. Up uses the same organization for complete low-rank activations and output-coordinate partitions; down uses it for additive low-rank and output contributions.

Backward gathers the gradient rows belonging to the same recorded pairs. For example, $Z_0^\top H_0$, $Z_1^\top H_1$, and $Z_2^\top H_2$ contract over 4, 3, and 1 rows, respectively. Pair alignment before a low-rank reduction preserves all eight vectors in $[T, k, r]$ order; it does not sum the experts selected by a token.

D Resource Analysis

We distinguish persistent training states, activation and workspace storage, collective input payloads, and arithmetic. Counts below refer to one up/down pair with equal E, r, k, unless stated otherwise. They do not predict peak memory or complete-step time independently.

D.1 Expert and Router States

A generic column-parallel adapted layer stores $Er(d_{\text{in}} + d_{\text{out}}/P)$ expert parameters per GPU. A row-parallel layer stores $Er(d_{\text{in}}/P + d_{\text{out}})$. For the up/down pair,

$$\begin{aligned} N_{\text{FR}} &= 2Er(D + F), \\ N_{\text{FS}} &= 2Er(D + F)/P, \\ N_{\text{PM}} &= 2Er(D + F/P). \end{aligned} \tag{48}$$

N_{FS} is the ideal equal full-sharding count, assuming valid unpadded partitions. The additional retained state relative to that count is

$$N_{\text{PM}} - N_{\text{FS}} = 2ErD(1 - 1/P), \qquad \rho_P = \frac{D}{D + F} + \frac{F}{D + F}\frac{1}{P}. \tag{49}$$

Thus, $1/P < \rho_P < 1$ for $P > 1$. Gradients and optimizer states follow the same ownership under matched storage conventions; actual bytes depend on parameter precision, gradient precision, master copies, and optimizer state. The up and down routers add DE and FE/P parameters per GPU. These are separate from the expert counts.

D.2 Cached Activations and Peak Memory

Each saved adapted-layer invocation retains $Kr = Tkr$ low-rank activation elements per GPU, giving $2Kr$ for an up/down pair. The corresponding input references have shapes $[T, D]$ and $[T, F/P]$; their incremental storage depends on reuse of base caches. Token indices, weights, and permutations use $O(K)$ entries, expert offsets use $O(E)$, and materialized router scores or their gradients use TE

entries. For local widths a, b, packed inputs and expert outputs occupy Ka and Kb elements. In particular, down can materialize a KD full-width expert-output buffer. Pair alignment uses $O(Kr)$ workspace.

The measured peak depends on simultaneous lifetimes:

$$M_{\text{peak}} = \max_t \big[M_{\text{base}}(t) + M_{\text{train states}}(t) + M_{\text{activations}}(t) + M_{\text{workspace}}(t)\big]. \tag{50}$$

The expert-state ratio therefore does not imply the same ratio in peak memory. Partial replication is a fixed execution choice, not a minimum-memory guarantee. Outer activation checkpointing may replay an invocation; the cached backward described here avoids replaying expert products within that backward itself.

D.3 Collective Payloads and Network Traffic

For $P > 1$, beyond base communication, an up/down invocation without replay uses a TE-element down-score all-reduce and two Kr-element backward all-reduces. Let b_L, b_δ, b_Z be the bytes per communicated score, activation gradient, and activation. The total additional collective input payload per GPU is

$$V_{\text{input}} = b_L TE + (b_\delta + b_Z)Kr. \tag{51}$$

For equal element sizes, the router accounts for the fraction $E/(E + 2kr)$. Communication thus grows with both token count and expert count, despite fixed top-k. Counts over an optimizer step must sum actual layer and microbatch invocations, including any forward replay. Initialization, base collectives, global optimizer operations, and data-parallel synchronization are outside Eq. 51.

For an illustrative ring all-reduce with $P > 1$, reduce-scatter and all-gather each send $P - 1$ chunks of size $1/P$ of the input. Per-GPU sent bytes for the extra collectives are consequently

$$V_{\text{send}} = \frac{2(P-1)}{P} V_{\text{input}}. \tag{52}$$

Received bytes are equal, and cluster-wide egress is PV_{send}. At $P = 1$, no network reduction is needed. Other collective algorithms can have different traffic patterns. With per-round startup ℓ_{comm} and effective bandwidth $\mathcal{B}$, a serial ring service-time model for an n-byte input is

$$t_{\text{AR}}(n) = 2(P-1)\ell_{\text{comm}} + \frac{2(P-1)}{P}\frac{n}{\mathcal{B}}. \tag{53}$$

This model excludes scheduling, contention, overlap, and local reordering. Neither payload nor this service time equals exposed training latency.

For one invocation, reducing all nonempty replicated-factor gradients would involve $E_{\text{active}} rD$ elements, compared with Kr for the selected low-rank vectors. At equal precision, the latter input is smaller when $K < E_{\text{active}} D$. This comparison concerns candidate messages, not complete implementations; the direct gradient path must also supply router and input derivatives. Appendix E defines that comparison and its gradient-accumulation scope.

D.4 Useful Work, Repeated Work, and Batch Shapes

With local widths a, b, expert e requires $2T_e r(a + b)$ forward FLOPs, counting a multiply-add as two FLOPs. Summing over experts gives

$$\mathcal{F}_{\text{site,fwd}} = 2Tkr(a + b), \qquad \mathcal{F}_{\text{PM,fwd}} = 4Tkr(D + F/P). \tag{54}$$

All TP GPUs process the same T_e with equal local feature widths, so their nominal expert work is equal. Different routing distributions can still change batch shapes, occupancy, and launch efficiency.

Ideal uniform partitioning of the original expert products would require $\mathcal{F}_{\text{ideal}} = 4Tkr(D + F)/P$ FLOPs per GPU. Partial replication gives

$$\begin{aligned} \frac{\mathcal{F}_{\text{PM,fwd}}}{\mathcal{F}_{\text{ideal}}} &= P\rho_P, \\ \Delta\mathcal{F}_{\text{fwd}} &= 4TkrD(1 - 1/P), \\ \frac{\Delta\mathcal{F}_{\text{fwd}}}{6TDF/P} &= \frac{2kr(P-1)}{3F}. \end{aligned} \tag{55}$$

The last denominator counts the gate, up, and down base FFN forward matrix products. This ratio is not the overhead of complete training. Retained expert state grows with E, while repeated useful expert forward work depends on the selected Tk pairs.

The four core backward products in Eq. 14 cost $4Tkr(a+b)$ FLOPs per site. Forward plus these products costs $12Tkr(D+F/P)$ for the pair. The described separate down contraction $sQ_eB_e^\top$ for mixing-weight derivatives adds $2TkrD$ FLOPs. Algebraic reuse could avoid repeating that contraction, but it is not assumed here. Dense router scoring and its two backward products add $6TE(D+F/P)$ FLOPs, excluding selection and normalization. Optimizer work grows with resident trainable parameters. These operations, together with gathering, scattering, and synchronization, are included when evaluating complete-step efficiency.

E Definitions of Controlled Execution Variants

The controlled variants isolate the synchronization and batching choices in Sections 3.3 and 3.4. ParaMoLE-GradSync and ParaMoLE-Loop correspond to the measured ablations in Table 3. Both retain the same MoLE parameters, TP factor placement, forward routing, loss, and base output aggregation. Padded and forward recovery are analytical comparisons for which no experimental results are reported. The definitions specify complete computational paths without assuming a performance ordering.

E.1 ParaMoLE-GradSync: Synchronizing Parameter Gradients

GradSync retains grouped expert products but replaces low-rank backward reconstruction with reductions of copied-factor gradients and per-pair mixing-weight derivatives. All communicated expert gradients are packed in a common expert-ID order; empty experts need not occupy message space. Complete input and router gradients remain required.

Up. Each GPU forms its local activation-gradient contribution and local copied-factor gradient:

$$\begin{aligned} \delta Z_e^{(p)} &= H_e^{(p)}(B_e^{(p)})^\top, \\ G_{A,e}^{(p)} &= X_e^\top \delta Z_e^{(p)}, \qquad \nabla_{A_e}\mathcal{L} = \mathrm{AR}(G_{A,e}^{(p)}). \end{aligned} \tag{56}$$

It also computes local mixing-weight contributions

$$h_{i,e}^{(p)} = s\langle z_{i,e}, q_i^{(p)}(B_e^{(p)})^\top\rangle, \qquad h_{i,e} = \sum_p h_{i,e}^{(p)}. \tag{57}$$

One reduction batches these scalar contributions for the real selected pairs. Complete h gives Γ through Eq. 20; the replicated router gradient is then $X^\top\Gamma$. The partitioned $B_e^{(p)}$ gradient remains $Z_e^\top H_e^{(p)}$.

Unlike ParaMoLE's reconstructed input contribution, GradSync's expert input contribution is still partial. It can join the base input-gradient SUM:

$$\nabla_X\mathcal{L} = \mathrm{AR}\left(Q^{(p)}(W^{(p)})^\top + \sum_e S_e^\top \delta Z_e^{(p)} A_e^\top\right) + \Gamma R^\top. \tag{58}$$

This avoids adding an independent full-width input-gradient reduction. The router contribution is already complete and is added afterward. For the gated FFN, the unchanged gate-branch contribution is included in the base backward computation as in Eq. 30.

Down. Each GPU retains its cached $Z_e^{(p)}$ and forms

$$\begin{aligned} G_{B,e}^{(p)} &= (Z_e^{(p)})^\top H_e, \qquad \nabla_{B_e}\mathcal{L} = \mathrm{AR}(G_{B,e}^{(p)}), \\ h_{i,e}^{(p)} &= s\langle z_{i,e}^{(p)}, q_i B_e^\top\rangle, \qquad h_{i,e} = \sum_p h_{i,e}^{(p)}. \end{aligned} \tag{59}$$

Mixing-weight contributions are reduced over the real pairs. Because B_e and Q_e are complete, $\delta Z_e = H_e B_e^\top$ is local. The partitioned A_e and router gradients and the input gradient use Eqs. 27 and 28, with complete Γ obtained from the reduced h.

Equivalence and synchronization timing. Linearity gives $\sum_p X_e^\top \delta Z_e^{(p)} = X_e^\top \delta Z_e$ at up and $\sum_p (Z_e^{(p)})^\top H_e = Z_e^\top H_e$ at down. The same sums recover h, and Eq. 58 recovers the original input gradient. Thus all required gradients, not only the copied-factor gradients, are included.

Copied-factor gradients may accumulate locally across microbatches and be reduced before the optimizer update, whereas mixing-weight and input gradients must be completed for the current backward pass. Under this delayed synchronization, let $\mathcal{E}_u^\cup$ and $\mathcal{E}_d^\cup$ be the unions of nonempty experts over an optimizer step. With $K_m = T_m k$ real pairs in microbatch m, its additional backward reduction inputs contain

$$V_{\text{GradSync,bwd}} = b_\theta r D\big(|\mathcal{E}_u^\cup| + |\mathcal{E}_d^\cup|\big) + 2b_h \sum_m K_m, \tag{60}$$

where b_θ and b_h are the transmitted element sizes. ParaMoLE's corresponding low-rank inputs contain $(b_\delta + b_Z) r \sum_m K_m$. Common router-score and base communication is excluded from both expressions. A different synchronization schedule must be counted according to its actual calls. Skipping empty messages does not change the optimizer treatment of empty experts, including momentum and weight decay.

E.2 ParaMoLE-Padded: Common-Length Batched Products

Padded retains ParaMoLE's low-rank synchronization and changes only the local expert products. Let E_{active} count nonempty expert batches and $T_{\max} = \max_e T_e$. Pack these experts in the same ID order and pad their token rows to $T_{\max}$:

$$\begin{aligned} X_{\text{pad}} &\in \mathbb{R}^{E_{\text{active}} \times T_{\max} \times a}, \\ A_{\text{stack}} &\in \mathbb{R}^{E_{\text{active}} \times a \times r}, \qquad B_{\text{stack}} \in \mathbb{R}^{E_{\text{active}} \times r \times b}. \end{aligned} \tag{61}$$

Forward uses ordinary batched matrix multiplication:

$$Z_{\text{pad}} = \text{bmm}(X_{\text{pad}}, A_{\text{stack}}), \qquad O_{\text{pad}} = \text{bmm}(Z_{\text{pad}}, B_{\text{stack}}). \tag{62}$$

Backward evaluates the same four products as Eq. 14 using the padded tensors. Padding inputs and upstream-gradient rows are zero; artificial rows have no route, output destination, or mixing-weight derivative. Consequently they contribute neither outputs nor parameter gradients.

Communication still uses only the K real pairs. Before a low-rank all-reduce, the real rows are extracted and aligned; afterward, complete real rows are restored to the padded buffers needed by subsequent products. These local conversions and buffer lifetimes are part of Padded's execution cost. Sending padded rows would also change communication and would not isolate local batching. The row-inflation factor is

$$\rho_{\text{pad}} = \frac{E_{\text{active}} T_{\max}}{K}. \tag{63}$$

It describes logical row counts rather than a runtime ratio. When expert batches are nearly equal, common-length batched products can remain competitive; no strict performance ordering is assumed.

E.3 ParaMoLE-Loop: Separate Expert Products

The loop comparison retains the same selected experts, caches, low-rank reductions, and weighted output accumulation. It evaluates each nonempty expert block's forward and backward matrix products separately rather than through grouped products. It does not pad expert batches or add per-expert communication calls. Any host-side offset retrieval, synchronization, or buffer-management costs are part of the implementation and must be distinguished from device kernel-launch overhead when interpreting profiles.

E.4 Recovering Down Activations in Forward

This comparison changes the timing of activation recovery, not the factor placement or the tensor being reduced. For fixed selections and weights, write $\mathcal{S}_g(\{O_e\}) = s \sum_e S_e^\top D_e O_e$ and $Y_{\text{base}}^{(p)} = X^{(p)} W^{(p)}$. ParaMoLE computes

$$Y = \text{AR}\Big(Y_{\text{base}}^{(p)} + \mathcal{S}_g(\{Z_e^{(p)} B_e\})\Big) \tag{64}$$

and recovers Z_e during backward. The alternative first reduces all real low-rank activation rows in one batch and then computes

$$Z_e = \sum_p Z_e^{(p)}, \qquad Y = \mathrm{AR}(Y_{\mathrm{base}}^{(p)}) + \mathcal{S}_g(\{Z_e B_e\}). \tag{65}$$

Its complete expert correction is added after the base SUM, so it is not counted P times. Linearity gives the same output. Caching complete Z_e supplies the B_e and mixing-weight derivatives in backward; $\delta Z_e = H_e B_e^\top$ remains complete locally and is propagated to each partial activation under the single-loss convention.

Without replay and at matched precision, both schedules reduce Kr activation elements once. They differ in dependencies and cache lifetimes, not this payload. Separately reducing a full-width expert correction is a different comparison: it adds a TD-element output reduction while the copied B_e update still requires completion. Checkpointing and overlap benefits must follow the implemented schedule and measurements, not the algebraic identity alone.

F Additional Experimental Results

F.1 Experimental Protocol

Table 6 summarizes the additional experiments on XSum. Each configuration uses BF16, seed 42. Unless varied, the default settings are $E = 64$, $k = 2$, $r = \alpha = 16$, sequence length $L_{\mathrm{seq}} = 1024$, $P = 4$ GPUs, micro-batch size $b_\mu = 1$, and global batch size $b_{\mathrm{g}} = 8$. FR, FS, and EP use the Loop backend, whereas ParaMoLE uses Grouped GEMM. FR, FS, and ParaMoLE use $\mathrm{TP} = P$ and $\mathrm{DP} = 1$. EP uses $\mathrm{TP} = 1$ with P data streams and an expert-parallel group of size P. The gradient accumulation count is b_{g}/b_μ for FR, FS, and ParaMoLE, and $b_{\mathrm{g}}/(Pb_\mu)$ for EP. A valid configuration requires a positive integer accumulation count. The comparisons include these backend and parallelism choices rather than isolating factor placement alone.

Here P denotes the allocated GPU count. It equals the TP degree for FR, FS, and ParaMoLE, whereas EP uses $\mathrm{TP} = 1$. The batch sizes count input examples and satisfy

$$b_{\mathrm{g}} = b_\mu n_{\mathrm{data}} n_{\mathrm{acc}},$$

where n_{data} is the number of concurrent streams of distinct examples and n_{acc} is the number of accumulated micro-batches per stream. Under the stated layouts, $n_{\mathrm{data}} = 1$ for FR, FS, and ParaMoLE, and $n_{\mathrm{data}} = P$ for EP. In contrast, T counts token rows passed to an adapted projection in one micro-batch, and $K = Tk$ counts selected token-expert pairs. For fixed-length padded batches, $T = b_\mu L_{\mathrm{seq}}$ when every example is padded to L_{seq} and all padded rows are processed by that projection.

Within the experimental tables, τ denotes mean complete-step time in seconds, including forward, backward, and the optimizer update. The duration of each step is the maximum across GPUs. M_{peak} denotes peak allocated memory in GiB per GPU over the measured window, excluding warm-up. C_{step} denotes captured public-API cross-GPU input payload in GiB per GPU per step, reported as the maximum across GPUs. Captured payload measures the input sizes of recorded communication calls rather than physical-link traffic or communication duration.

Non-padding token throughput V_{tok} is defined over the same timing window. For n_{step} measured optimizer steps, let τ_j be the duration of step j and N_j the number of non-padding tokens in its global batch. The reporting convention is

$$\tau = \frac{1}{n_{\mathrm{step}}} \sum_{j=1}^{n_{\mathrm{step}}} \tau_j, \qquad V_{\mathrm{tok}} = \frac{\sum_{j=1}^{n_{\mathrm{step}}} N_j}{\sum_{j=1}^{n_{\mathrm{step}}} \tau_j}. \tag{66}$$

Token counts exclude duplicate TP copies and repeated expert assignments. OOM denotes an observed out-of-memory failure, and N/A denotes an incompatible batch configuration. Results from different experimental campaigns are reported separately.

Table 6: Configurations of the additional experiments on XSum. Unspecified settings follow the defaults in the text. Both models refers to Qwen2.5-3B and Llama-3.2-3B.

Experiment	Model	Scan values	Batch setting	GPUs
Expert count	Llama-3.2-3B	$E \in \{8, 16, 32, 64, 128, 256, 320\}$	$b_\mu = 1,\ b_\mathrm{g} = 8$	4
Top-k	Llama-3.2-3B	$k \in \{1, 2, 4, 8\}$	$b_\mu = 1,\ b_\mathrm{g} = 8$	4
LoRA rank	Llama-3.2-3B	$r \in \{8, 16, 32, 64\}$	$b_\mu = 1,\ b_\mathrm{g} = 8$	4
Sequence length	Llama-3.2-3B	$L_\mathrm{seq} \in \{256, 512, 1024, 2048\}$	$b_\mu = 1,\ b_\mathrm{g} = 8$	4
Micro-batch ($b_\mathrm{g} = 8$)	Both models	$b_\mu \in \{1, 2, 4, 8\}$	$b_\mathrm{g} = 8$	4
Micro-batch ($b_\mathrm{g} = 32$)	Both models	$b_\mu \in \{1, 2, 4, 8\}$	$b_\mathrm{g} = 32$	4
Strong scaling	Both models	$P \in \{1, 2, 4\}$	$b_\mu = 1,\ b_\mathrm{g} = 8$	1, 2, 4

Table 7: Sensitivity results on Llama-3.2-3B and XSum. τ is step time in seconds, M_peak is peak memory in GiB/GPU, and C_step is captured communication payload in GiB/GPU/step.

Value	FR			FS			EP			ParaMoLE		
	τ	M_peak	C_step	τ	M_peak	C_step	τ	M_peak	C_step	τ	M_peak	C_step
Expert count E												
8	3.092	11.384	6.202	3.245	5.743	5.647	0.724	17.740	10.035	1.460	4.053	5.281
16	4.746	12.882	6.278	4.654	6.097	5.660	0.791	18.337	9.921	1.467	4.713	5.284
32	7.218	15.590	6.431	7.740	6.861	5.688	0.884	19.036	10.267	1.464	5.981	5.291
64	12.778	21.183	6.737	12.910	9.354	5.742	2.119	20.639	10.030	1.474	8.519	5.305
128	22.120	32.513	7.349	21.993	13.159	5.852	1.565	23.514	10.124	1.482	13.662	5.332
256	40.547	55.113	8.572	37.283	26.843	6.070	2.610	29.591	10.002	1.496	23.879	5.387
320	52.117	66.379	9.184	40.764	29.791	6.180	2.921	32.857	10.357	1.506	28.974	5.414
Top-k												
1	11.914	18.747	6.732	11.881	8.716	5.715	1.023	17.411	5.080	1.473	8.517	5.291
2	12.778	21.183	6.732	12.910	9.354	5.742	2.119	20.639	10.030	1.474	8.519	5.305
4	14.809	26.053	6.732	13.911	10.563	5.797	1.335	26.822	20.074	1.491	8.525	5.332
8	18.871	35.756	6.734	15.581	12.012	5.906	1.829	38.830	39.437	1.503	8.533	5.387
LoRA rank r												
8	13.176	15.766	6.436	13.013	6.942	5.714	1.121	19.233	10.078	1.456	6.070	5.290
16	12.778	21.183	6.737	12.910	9.354	5.740	2.119	20.639	10.030	1.474	8.519	5.306
32	13.913	32.014	7.338	12.444	14.014	5.794	1.132	23.272	10.027	1.478	13.444	5.331
64	14.453	53.684	8.542	12.971	26.407	5.907	1.131	28.713	9.999	1.492	23.286	5.386
Sequence length L_seq												
256	11.789	15.643	2.143	11.652	8.102	1.436	1.046	11.760	2.507	1.454	7.318	1.326
512	12.283	17.494	3.674	12.747	8.843	2.871	1.054	14.532	4.968	1.461	7.705	2.652
1024	12.778	21.183	6.737	12.910	9.354	5.742	2.119	20.639	10.030	1.474	8.519	5.305
2048	15.455	29.201	12.862	13.188	12.856	11.485	1.589	35.039	20.746	1.472	10.783	10.610

F.2 Sensitivity to the MoLE Configuration

Table 7 examines the effects of expert count, routing sparsity, LoRA rank, and sequence length on Llama-3.2-3B. Increasing E from 8 to 320 changes ParaMoLE's step time from 1.460 to 1.506 s, an increase of 3.2%, while peak memory rises from 4.053 to 28.974 GiB per GPU. At $E = 320$, ParaMoLE takes 1.506 s per step, compared with 52.117 s for FR, 40.764 s for FS, and 2.921 s for EP. ParaMoLE also uses less peak memory than FS and EP at this setting, although FS has a lower peak at some intermediate expert counts. Increasing the LoRA rank likewise raises ParaMoLE's memory requirement while changing its measured step time only slightly. EP is faster at $E \in \{8, 16, 32\}$ in the expert-count sweep and $r \in \{8, 32, 64\}$ in the rank sweep, whereas ParaMoLE is faster at the default $E = 64, r = 16$ configuration. EP requires more memory and communication payload at these settings.

Increasing top-k has a larger effect on EP's communication. From $k = 1$ to 8, EP's captured payload rises from 5.080 to 39.437 GiB per GPU per step, while ParaMoLE's increases from 5.291 to 5.387 GiB. EP has the smaller payload at $k = 1$, but ParaMoLE achieves the lowest step time, peak memory, and payload at $k = 8$. At $k = 1$, selected-score softmax gives zero task-loss router gradients (Eq. 20). This configuration tests execution cost rather than learned top-1 routing. As L_seq grows from 256 to 2048, ParaMoLE's peak memory rises from 7.318 to 10.783 GiB and its payload from 1.326 to 10.610 GiB per GPU per step. ParaMoLE nevertheless records the lowest values of all

Table 8: Micro-batch sensitivity on XSum at fixed global batch sizes. All runs use four GPUs and the default settings in Section F.1. τ is complete-step time (s), M_{peak} is peak allocated memory (GiB/GPU), and V_{tok} is non-padding token throughput (tokens/s). OOM denotes an observed memory failure, and N/A denotes an incompatible batch configuration. † marks reconstructed throughput estimates described below the table.

Model	b_μ	FR			FS			EP			ParaMoLE		
		τ	M_{peak}	V_{tok}	τ	M_{peak}	V_{tok}	τ	M_{peak}	V_{tok}	τ	M_{peak}	V_{tok}
Qwen2.5-3B	*Global batch* $b_{\text{g}} = 8$												
	1	15.429	30.666	223.6	15.428	10.969	223.7	2.410	27.708	1431.6†	1.960	10.101	1760.3
	2	8.933	41.175	386.3	8.553	15.028	403.5	1.963	44.385	1758.5†	0.987	12.161	3495.6
	4	6.186	62.475	557.8	4.881	23.055	706.9	N/A	N/A	N/A	0.579	16.224	5962.5
	8	OOM	OOM	OOM	2.792	38.848	1235.7	N/A	N/A	N/A	0.457	24.218	7543.0
	Global batch $b_{\text{g}} = 32$												
	1	62.914	30.671	224.6	64.711	10.969	218.3	7.689	27.719	1837.6†	7.948	10.101	1777.6
	2	37.298	41.183	378.8	34.974	15.026	404.0	5.726	44.419	2467.1†	4.001	12.161	3531.2
	4	25.423	62.478	555.7	19.248	23.057	734.0	3.007	78.270	4698.3	2.259	16.226	6255.3
	8	OOM	OOM	OOM	11.202	38.839	1261.2	OOM	OOM	OOM	1.768	24.225	7989.8
Llama-3.2-3B	*Global batch* $b_{\text{g}} = 8$												
	1	12.778	21.183	266.5	12.910	9.354	263.8	2.119	20.639	1607.8†	1.474	8.519	2311.0
	2	7.438	28.449	457.9	7.340	11.108	464.0	1.707	31.841	1995.7†	0.753	10.048	4524.7
	4	4.730	43.019	720.0	3.928	16.592	867.1	N/A	N/A	N/A	0.450	12.953	7575.9
	8	3.535	72.065	963.4	2.205	27.591	1544.3	N/A	N/A	N/A	0.372	18.907	9156.4
	Global batch $b_{\text{g}} = 32$												
	1	54.787	21.183	254.5	49.621	8.356	281.0	6.395	20.633	2180.6†	5.808	8.519	2401.0
	2	30.881	28.450	451.5	29.004	11.102	480.8	6.840	31.843	2038.6†	2.985	10.048	4672.0
	4	19.175	43.015	727.2	15.342	16.591	908.9	2.215	54.194	6295.8	1.765	12.953	7900.3
	8	14.442	72.070	965.5	8.495	27.584	1641.4	OOM	OOM	OOM	1.440	18.907	9680.6

† The eight EP throughput entries at $b_\mu = 1$ and 2 are approximated as $V_{\text{tok,old}}\tau_{\text{old}}/\tau$ using the earlier reported pair for the same configuration and the current step time. This reconstruction assumes an unchanged token workload and the window-level reporting convention in Eq. 66. These values are derived from rounded summaries, not recomputed from raw token logs or measured in new runs. The remaining throughput entries are unchanged.

three metrics at $L_{\text{seq}} = 2048$. These results show limited growth in complete-step time within the tested regime, while larger expert pools and longer sequences still increase resource requirements.

F.3 Micro-batch Size at a Fixed Global Batch

Table 8 examines whether memory savings support larger micro-batches and higher throughput without increasing the global batch size. At $b_{\text{g}} = 32$, EP is slightly faster than ParaMoLE on Qwen2.5-3B at $b_\mu = 1$, but ParaMoLE is faster at $b_\mu = 2$ and 4. On Llama-3.2-3B, ParaMoLE is faster at all three micro-batch sizes feasible for EP. At the matched $b_\mu = 4$, ParaMoLE is 1.33× faster on Qwen2.5-3B and 1.25× faster on Llama-3.2-3B. ParaMoLE also completes $b_\mu = 8$, where EP encounters OOM on both backbones, reaching 7989.8 and 9680.6 tokens/s at 24.225 and 18.907 GiB per GPU, respectively. These throughputs are 1.70× and 1.54× the highest reported feasible EP throughputs in the same $b_{\text{g}} = 32$ sweep. The latter comparison uses each method's best reported feasible micro-batch, rather than an identical micro-batch size. FS also completes $b_\mu = 8$, but requires more memory and longer steps than ParaMoLE.

At $b_{\text{g}} = 8$, increasing b_μ from 1 to 8 raises ParaMoLE's throughput from 1760.3 to 7543.0 tokens/s on Qwen2.5-3B and from 2311.0 to 9156.4 tokens/s on Llama-3.2-3B. The corresponding peak memory increases from 10.101 to 24.218 GiB and from 8.519 to 18.907 GiB per GPU. Under the specified four-GPU EP layout, $b_\mu = 4$ and 8 would require an accumulation count $b_{\text{g}}/(Pb_\mu)$ below one. These configurations are therefore marked N/A rather than OOM. The results distinguish batch-compatibility constraints from observed memory limits and show how larger micro-batches can improve throughput within ParaMoLE's available memory.

F.4 Strong Scaling at a Fixed Global Batch

Table 9 increases GPU count from one to four while fixing $b_{\text{g}} = 8$ and $b_\mu = 1$. ParaMoLE reduces peak memory from 31.045 to 10.101 GiB per GPU on Qwen2.5-3B and from 22.919 to 8.519 GiB

Table 9: Strong scaling on XSum with $b_g = 8$ and $b_\mu = 1$. τ is step time in seconds and M_{peak} is peak memory in GiB/GPU. Scaling efficiency is $\eta(P) = 100\tau(1)/(P\tau(P))$ in percent, computed within each method from the displayed one-GPU and P-GPU step times.

Model	P	FR			FS			EP			ParaMoLE		
		τ	M_{peak}	η	τ	M_{peak}	η	τ	M_{peak}	η	τ	M_{peak}	η
Qwen2.5-3B	1	14.301	42.486	100.0	14.283	42.485	100.0	11.425	39.561	100.0	2.001	31.045	100.0
	2	14.930	34.683	47.9	15.378	21.588	46.4	3.782	31.582	151.0	1.843	17.158	54.3
	4	15.429	30.666	23.2	15.428	10.969	23.1	2.410	27.708	118.5	1.960	10.101	25.5
Llama-3.2-3B	1	12.061	31.155	100.0	12.655	31.161	100.0	9.107	28.233	100.0	1.507	22.919	100.0
	2	13.550	24.560	44.5	12.955	15.980	48.8	3.023	22.958	150.6	1.468	13.372	51.3
	4	12.778	21.183	23.6	12.910	9.354	24.5	2.119	20.639	107.4	1.474	8.519	25.6

on Llama-3.2-3B, corresponding to reductions of 67.5% and 62.8%. Step time changes only from 2.001 to 1.960 s and from 1.507 to 1.474 s, respectively. The four-GPU scaling efficiencies are 25.5% and 25.6%. Both backbones have slightly shorter measured steps at two GPUs than at four GPUs. Additional TP GPUs therefore primarily reduce the per-GPU memory requirement in this small-micro-batch regime rather than provide near-linear acceleration.

At four GPUs, ParaMoLE is faster than EP on both backbones, with step times of 1.960 versus 2.410 s on Qwen2.5-3B and 1.474 versus 2.119 s on Llama-3.2-3B. ParaMoLE also uses less peak memory than EP. FS reaches a similar memory range, but requires substantially longer steps. EP's four-GPU scaling efficiencies are 118.5% and 107.4%, computed relative to its own one-GPU step times. Values above 100% describe superlinear runtime ratios, not GPU utilization. These ratios do not imply that EP has a shorter absolute step time than ParaMoLE, and the aggregate timings do not identify the cause of the superlinear scaling.