COMMUNICATE THE CHANGE: EFFICIENT EXPERT PARALLELISM VIA ACTIVATION COMPRESSION

Anonymous authors
Paper under double-blind review

ABSTRACT

Scaling has driven the success of large language models (LLMs), with sparse mixture-of-experts (MoE) architectures enabling greater parameter capacity without a proportional increase in computation. At large scales, MoE models rely on expert parallelism to distribute experts across devices. Each MoE layer requires two synchronous collective communication steps during inference: dispatching hidden states to the activated experts and combining the resulting expert outputs. As compute throughput has outpaced interconnect bandwidth, these synchronous collectives have become a major bottleneck to MoE scaling and efficiency. We address this challenge with *EchoMoE*, a lightweight architecture modification that retrofits existing MoE models, requiring ~ 8500 times fewer training tokens than full pretraining. Motivated by the observation that transformer blocks update the residual stream incrementally, we use a hidden state from an earlier layer as a reference and communicate only the compressed difference between it and the current state, rather than communicating each hidden state in full. EchoMoE periodically refreshes these reference states and transmits only the compressed differences synchronously, reducing the traffic on latency-critical paths without significant computational overhead. EchoMoE reduces inter-node communication volume by 46% and improves the throughput by 17.2%, while preserving average performance within 2 percentage points across 18 benchmarks. EchoMoE also tolerates quantization of the communicated representations without measurable quality loss. We release an efficient implementation for applying EchoMoE to pretrained MoE models, together with an inference integration for vLLM [1].

1 INTRODUCTION

Scaling has become a central paradigm in Large Language Models (LLM) development, with empirical scaling laws driving a continued push toward larger models (Kaplan et al., 2020; Hoffmann et al., 2022). Mixture-of-Experts (MoE) model (Shazeer et al., 2017) enables substantial growth in parameter count while keeping per-token computation relatively modest through sparse expert activation. Serving frontier MoE language models such as Kimi K3 (Kimi Team et al., 2026) and DeepSeek-V4 (DeepSeek-AI et al., 2026) requires distributing experts across many accelerators (e.g., GPUs) using Expert Parallelism (EP) (Rajbhandari et al., 2022). In standard EP (Figure 1 top left), at every MoE layer, each token's activation is sent to the devices hosting its selected experts (*dispatch*), and the expert outputs are sent back (*combine*). When that model is sharded across multiple nodes, both transfers carry the hidden state across the network (RDMA). For DeepSeek-V3 (DeepSeek-AI et al., 2025) served with 128 GPUs (EP128), a single 128K-token prefill moves up to 812 GiB of activations between nodes (Zhao et al., 2025a) and these collectives account for 32–60% of transformer-layer latency even after overlapping communication with computation (Research, 2025). As accelerator compute grows faster than interconnect bandwidth (Gholami et al., 2024), EP communication overhead increasingly limits MoE serving throughput and latency.

Prior work reduces this cost along two axes. The first redesigns the model architecture to reduce communication volume, either by communicating lower-dimensional activations or by organizing experts to localize more traffic within a node. (Elango et al., 2026; Jin et al., 2025; Abdurrahman et al., 2026; Cai et al., 2025). These designs are effective, but they require architectural changes

[1] https://anonymous.4open.science/r/EchoMoE-751C

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

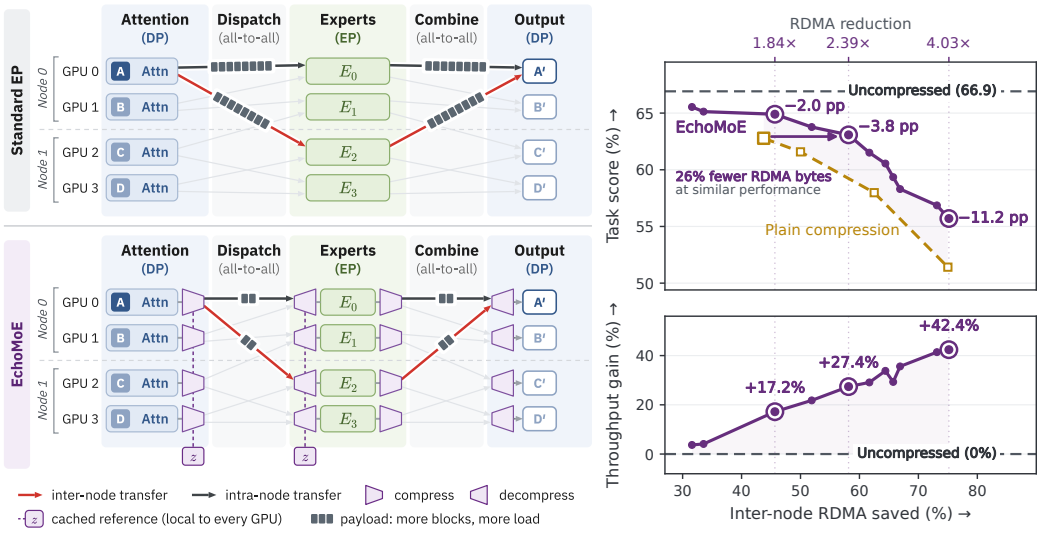


Figure 1: **EchoMoE reduces EP communication and improves inference throughput. Left:** Standard EP dispatches full activations to selected experts and returns full expert outputs to each token's owner rank. EchoMoE instead compresses dispatch activations using cached cross-layer references and compresses expert outputs before combine. Token A's activation communication path is highlighted. **Right:** EchoMoE saves 46% of RDMA bytes and increases throughput by 17.2% at a 2.0 pp task score drop. At similar task performance, EchoMoE sends 26% less RDMA traffic than plain compression, which directly compresses dispatch activations. RDMA savings include reference transfers at EP32 over 4 nodes, with EP communication in BF16. Throughput is measured with EP32, 128 requests per GPU, 4096 input and 256 output tokens.

before pretraining and cannot be directly applied to existing pretrained MoE models. The second leaves the model unchanged and reduces the cost of each communication through faster collective kernels or reduced-precision activations (Hwang et al., 2023; Zhao et al., 2025b). These methods apply to off-the-shelf pretrained MoE models and are orthogonal to EchoMoE, as they reduce the cost per communicated element rather than the number of elements communicated. Regardless of precision, every MoE layer still transfers all d activation dimensions on both collective operations.

Transformer blocks update the residual stream incrementally, causing nearby MoE layers to receive highly correlated inputs (Fang et al., 2025; Eliseev & Mazur, 2023; Lee et al., 2024; Yu et al., 2026). We therefore hypothesize that successive full-width activations communicated across nodes during EP are highly redundant. Given a cached activation from an earlier layer, much of the current activation is predictable. Standard EP nevertheless transmits the full d-dimensional activation over RDMA at every layer. In this paper, we investigate whether this redundancy can be exploited to reduce EP communication in existing MoE models without retraining from scratch.

We introduce EchoMoE, which periodically caches a full-width activation from an earlier layer and reuses it as a reference for subsequent MoE layers. Rather than transmitting the full current activation, EchoMoE communicates only a compressed change relative to the cached activation, substantially reducing inter-node RDMA traffic (Figure 1 bottom left, Figure 2).

We evaluate EchoMoE on Qwen3-30B-A3B (Yang et al., 2025) across 97 trained variants and 18 benchmarks spanning knowledge & reasoning, mathematics, coding and instruction following (Figure 1). One of our frontier configurations ($s = 4$, $r_d = 512$, $r_c = 1280$) reduces inter-node RDMA traffic by 46% at EP32 across four nodes, while mean task performance drops by 2.0 percentage points (pp). When served with vLLM at 128 requests per GPU, EchoMoE lowers prefill latency by 18% and decode latency by 2%, yielding a 17.2% throughput improvement. At 512 requests per GPU, the decode gain rises to 7%. EchoMoE also composes cleanly with FP8 quantization, with no additional loss in task performance. We empirically validate the effectiveness of cross-layer references by showing that EchoMoE substantially outperforms plain compression at comparable RDMA. We also develop an effective procedure for searching the frontier configuration space.

	Architecture		Inter-node RDMA Communication		
Model	d	L	MiB/token	GiB/128K	Transfer (ms)
Qwen3-30B-A3B	2048	48	1.125	144	97
Qwen3-235B-A22B	4096	94	4.406	564	379
DeepSeek-V3	7168	58	4.758	609	409

Table 1: Inter-node activation traffic and transfer time for a 131,072-token prefill on an EP32 group of 4 nodes at 400 Gb/s per GPU, computed from Equation (1). All three models (Yang et al., 2025; DeepSeek-AI et al., 2025) activate eight experts per token. We consider MoE layers only.

2 Expert-Parallel Communication

2.1 Mixture of Experts and Expert Parallelism

An MoE layer replaces the dense feed-forward network with N expert networks and a router. For each token activation $x \in \mathbb{R}^d$, the router scores all N experts and selects the K highest where typically $K \ll N$. Let S index the selected experts and E_i denote expert i. The MoE output is $y = \sum_{i \in S} w_i \, E_i(x)$, where the router weights w_i are normalized over the selected experts. Each expert is a gated MLP with intermediate width f, and the resulting output y is added to the residual stream. Increasing N increases model capacity while keeping computation proportional to K.

At frontier scale, the experts of a layer are distributed across accelerators using EP. Throughout this paper, we use EP to refer to expert parallelism combined with data parallelism (DP), i.e., the EP+DP setting. Experts are partitioned across ranks, while non-expert parameters are replicated. At EP degree P, each rank stores N/P experts from every MoE layer. Each sequence is associated with an owner rank which holds its residual state and executes the non-expert computation locally.

Because the experts selected for a token may reside on remote ranks, every MoE layer requires two collective communications. During dispatch, the owner sends the token activation x to the ranks hosting the selected experts, which compute $E_i(x)$. During combine, the expert outputs are returned to the owner which aggregates them to form y. Both transfers are typically implemented with all-to-all collectives (Lepikhin et al., 2020; Hwang et al., 2023; Zhao et al., 2025b), which route each token only between its owner and the ranks hosting its selected experts rather than replicating it across all ranks. Thus, a model with L MoE layers performs $2L$ communication rounds per forward pass, with each routed activation still transmitted at the full hidden width d.

2.2 Expert-Parallel Communication Overheads

Dispatch and combine introduce communication dependencies and latency. For DeepSeek-V3 (DeepSeek-AI et al., 2025) at EP128 on H100 nodes, they account for roughly 60% of transformer-layer latency in isolation, and still 32% once overlapped with computation through microbatching (Research, 2025). Because inter-node links are much slower than intra-node interconnects, inter-node transfers dominate communication cost. Let T denote the number of tokens processed across the EP group, b the bytes per activation dimension, and $\bar{m}$ the average number of remote nodes reached by each token per collective communication.[2] Each round sends one width d activation to each of those nodes, so the total inter-node activation traffic across L MoE layers is:

$$C_{\text{base}}(T) = \underbrace{b\,d\,L\,\bar{m}\,T}_{\text{dispatch}} + \underbrace{b\,d\,L\,\bar{m}\,T}_{\text{combine}} = 2\,b\,d\,L\,\bar{m}\,T \tag{1}$$

Lower precision reduces b and expert placement reduces $\bar{m}$, but d is fixed by the architecture and every transfer carries all of it. Since a token's K selected experts can span at most K nodes, $\bar{m}$ eventually saturates, after which increasing the EP group size no longer reduces per-token traffic.

Communication time depends on both payload and number of collective calls. Each call incurs a fixed startup cost and the time required to transmit its payload at the available bandwidth. Table 1

[2] $\bar{m}$ counts distinct remote nodes, not experts. Hierarchical forwarding sends one activation per node in each direction, so several selected experts on one node cost one transfer. DeepEP V1's low-latency kernels instead send one payload per expert (Zhao et al., 2025b;a). We exclude intra-node transfers and protocol overhead.

reports the traffic volume and the lower-bound communication time during prefill for two Qwen3 models and DeepSeek-V3.[3] During prefill, all prompt tokens are processed in batches, yielding large payloads for which bandwidth-dependent transfer time dominates. During decoding, each pass processes one token per sequence, producing much smaller payloads, memory bottleneck and startup overhead are comparatively more significant (Zhao et al., 2025a;b).

3 Similarity and Predictability in Dispatched Activations

Each transformer block adds its output to the residual stream rather than overwriting it, so a token's hidden state changes gradually with depth and nearby MoE layers read highly similar inputs. Yet conventional expert parallelism still communicates the full d-dimensional activation at every layer, even when much of it is predictable from an earlier state.

Table 2 quantifies the similarity and predictability. Across all four models, a token's inputs to adjacent MoE layers have a mean cosine similarity above 90%. We also fit a per-dimension affine predictor from an earlier-layer input to a later one and evaluate it on held-out tokens. The predictor recovers over 80% of the later input at $k = 1$ and over 50% at $k = 4$. The high similarity and high predictability hold consistently across models. See Appendix A for metric details.

Model	L	Cosine similarity (%)				Predictability (%)			
		$k=1$	4	8	16	$k=1$	4	8	16
Qwen3-30B-A3B-Instruct-2507	48	91.6	74.6	61.5	46.3	83.1	55.7	39.7	24.6
Qwen3-235B-A22B-Instruct-2507	94	95.1	83.3	71.6	57.9	90.4	70.0	53.0	36.5
DeepSeek-V3	58	93.7	77.5	61.5	39.7	88.1	62.2	40.3	17.5
GLM-5.3	75	93.4	76.7	60.0	38.0	87.3	60.1	38.2	16.6

Table 2: Similarity and predictability of MoE inputs across layer separations k. Values are averaged over all pairs of MoE layers separated by k. Predictability is the fraction of the later-layer input recovered from the earlier-layer input by a per-dimension affine predictor.

4 EchoMoE

EchoMoE exploits the cross-layer similarity and predictability described in Section 3 by caching an earlier activation for each token on every rank and reusing it as a reference across several subsequent MoE layers. Between reference layers, only the change relative to its cached state is compressed and dispatched, and combine returns a compressed representation instead of full expert outputs.

Let $x_\ell \in \mathbb{R}^d$ denote a token's input to MoE layer ℓ. EchoMoE partitions the L MoE layers into groups of s consecutive layers, where s is the *stride*. The first layer of each group is a reference layer. It broadcasts x_ℓ at the full width d to every other rank, where every rank caches it as $z \in \mathbb{R}^d$. The $s - 1$ layers that follow reuse z, and only need to communicate the compressed change. Full width hidden states are transmitted once per layer group rather than at every MoE layer.

4.1 Compressing Dispatch

For each non-reference layers, let $C_d \in \mathbb{R}^{r_d \times d}$ and $D_d \in \mathbb{R}^{d \times r_d}$ denote the linear compressor and decompressor for dispatch. Since every rank already holds the z from the latest reference layer, the owner rank compresses and dispatches only the component of x_ℓ not predicted from z. Specifically, a learned gain $a \in \mathbb{R}^d$ rescales z, and the compressor encodes the difference $x_\ell - a \odot z$ as:

$$c_d = C_d \left(x_\ell - a \odot z \right) \tag{2}$$

The owner dispatches $c_d \in \mathbb{R}^{r_d}$ to the ranks hosting the selected experts. At reference layers, dispatch is uncompressed. The owner first sends x_ℓ to the selected expert ranks, then asynchronously

[3] We use bfloat16 communication ($b = 2$) and maximum fan-out so every token reaches all three remote nodes ($\bar{m} = 3$). Fewer destinations reduce traffic proportionally. Only DeepSeek-V3 has a shared expert and we assume it is replicated on every node so it adds no inter-node traffic. These estimations are lower bounds since they assume peak bandwidth on a balanced network and exclude startup overhead.

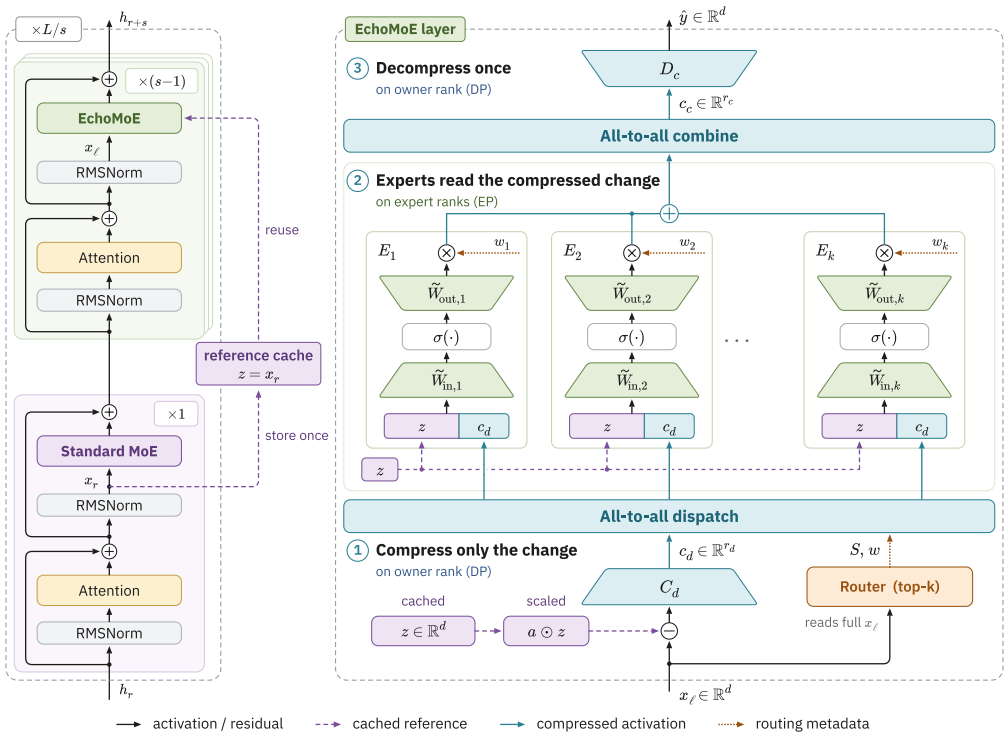


Figure 2: **EchoMoE. Left:** The L MoE layers are partitioned into L/s groups of s consecutive layers. The first layer in each group serves as a reference layer, broadcasting its full-width state x_ℓ, which every rank caches as z. The following $s-1$ layers in each group reus this reference z. **Right:** At each non-reference layer, the owner rank compresses and dispatches only the difference $x_\ell - a \odot z$ as $c_d \in \mathbb{R}^{r_d}$. Each expert applies $\widetilde{W}_{\text{in},i}$ to $\left[\begin{smallmatrix} z \\ c_d \end{smallmatrix}\right]$ and $\widetilde{W}_{\text{out},i}$, without explicit full-width input reconstruction or output materialization. Each node returns a single compressed state of width r_c, which is combined before the owner rank decompresses with D_c. The router reads the full x_ℓ.

broadcasts x_ℓ to all ranks to cache as the reference state. Although this broadcast incurs additional traffic, it does not block the forward pass or introduce additional synchronization latency.

Expert rank reconstructs the input $\hat{x}_\ell \in \mathbb{R}^d$ from the cached reference z and dispatched code c_d:

$$\hat{x}_\ell = D_d c_d + G z \tag{3}$$

where $G_z \in \mathbb{R}^{d \times d}$ is a learned gain matrix. We learn G_z seperately from a, allowing the reconstruction to compensate for information lost when the difference $x_\ell - a \odot z$ is compressed. The rank then executes its selected experts on $\hat{x}_\ell$. For plain compression baseline without the cached reference, dispatch compresses x_ℓ directly as $c_d = C_d x_\ell$ and a receiving rank recovers it as $\hat{x}_\ell = D_d c_d$.

Folding The decompressor D_d, the reconstruction gain G, and the gate and up projections $W_{\text{gate},i}$ and $W_{\text{up},i}$ of expert i are all linear, therefore can be folded into a expert specific single projection:

$$\widetilde{W}_{\text{in},i} = \begin{bmatrix} W_{\text{gate},i} \\ W_{\text{up},i} \end{bmatrix} [G \quad D_d] \in \mathbb{R}^{2f \times (d + r_d)} \tag{4}$$

The expert applies $\widetilde{W}_{\text{in},i}$ directly to $\left[\begin{smallmatrix} z \\ c_d \end{smallmatrix}\right]$, avoiding explicit reconstruction of $\hat{x}_\ell$. Folding absorbs both D_d and G, increases the width of the expert's input projection from d to $d + r_d$. A dense G therefore incurs no additional serving time operation. This expert specific folding also makes D_d and G expert specific, where E_d and a remain shared across experts within each layer.

4.2 COMPRESSING COMBINE

Recent MoE models use fine-grained experts whose expert intermediate dimension is smaller than the model hidden dimension, while activating more experts per token. This suggests that expert outputs, which are updates to the residual stream, might be represented in a lower-dimensional space.

Let $C_c \in \mathbb{R}^{r_c \times d}$ and $D_c \in \mathbb{R}^{d \times r_c}$ denote the linear compressor and decompressor for combine, and let S_j index the selected experts hosted by node j. Both combine and compression are linear, we thus apply both router weight w_i and compressor C_c locally, so node j forms $c_c^{(j)} \in \mathbb{R}^{r_c}$:

$$c_c^{(j)} = \sum_{i \in S_j} w_i \, C_c E_i(\hat{x}_\ell) = C_c \sum_{i \in S_j} w_i \, E_i(\hat{x}_\ell) \tag{5}$$

$c_c^{(j)}$ are combined at width r_c. The owner rank decompresses the combine sum as:

$$\hat{y} = D_c \sum_j c_c^{(j)} \in \mathbb{R}^d \tag{6}$$

Combine compression is applied to every MoE layer. At a reference layer, the experts run on x_ℓ.

Folding Similar to Equation (4), the compressor C_c and the down projection $W_{\text{down},i}$ of expert i are both linear so they compose into one expert specific matrix:

$$\widetilde{W}_{\text{out},i} = C_c W_{\text{down},i} \in \mathbb{R}^{r_c \times f} \tag{7}$$

The expert applies $\widetilde{W}_{\text{out},i}$ to its hidden state without materializing an output at width d. This allows the implicit C_c to be expert specific, while D_c is shared across experts per layer.

4.3 COMMUNICATION COST

Let $m_{\max}$ denote the number of remote nodes in EP. A reference reaches all $m_{\max}$ nodes while a dispatch or combine reaches only $\bar{m}$ that host selected experts, so $\bar{m} \le m_{\max}$. A group of s layers sends one reference, $s-1$ dispatch and s combine. Following Equation (1), the traffic volume is

$$C_{\text{Echo}}(T) = \underbrace{b\,d\,\tfrac{L}{s}\,m_{\max}\,T}_{\text{reference states}} + \underbrace{b\,r_d\,\tfrac{s-1}{s} L\,\bar{m}\,T}_{\text{dispatch compressed}} + \underbrace{b\,r_c\,L\,\bar{m}\,T}_{\text{combine compressed}} \tag{8}$$

Additionally, the broadcast of reference states can be done asynchronously, further reducing communication time through compute-communication overlap. Reduced precision lowers b in every term so it can be composed with EchoMoE. The memory overhead for caching z is negligible comparing with KV-cache.

4.4 RETROFITTING AN EXISTING MODEL

To retrofit an existing model without training from scratch, EchoMoE aims to preserve its original behavior by first minimizing activation reconstruction error as initialization, then train end-to-end.

Initialization We run the uncompressed model over a set of documents and record the input activation x_ℓ and the combined expert output y at every MoE layer. For combine, we reconstruct y. For dispatch, reconstructing x_ℓ itself is sub-optimal because each expert utilize $\hat{x_\ell}$ only through its gate and up projections. We therefore minimize reconstruction error after these projections. We define:

$$R = \sum_i \rho_i \left(W_{\text{gate},i}^\top W_{\text{gate},i} + W_{\text{up},i}^\top W_{\text{up},i} \right) \tag{9}$$

where ρ_i denote the fraction of tokens routed to expert i. Then $\|R^{1/2} v\|^2$ is the projection error induced by an activation reconstruction error v, averaged over router assignments. We then initialize a, C_d, D_d, C_c, and D_c with closed form solutions that minimize reconstruction error:

$$\min_{C_d,\, D_d} \mathbb{E} \left\| R^{1/2} \left(I - D_d C_d \right) \left(x_\ell - a \odot z \right) \right\|^2 \qquad \text{and} \qquad \min_{C_c,\, D_c} \mathbb{E} \left\| \left(I - D_c C_c \right) y \right\|^2 \tag{10}$$

Configuration				Task Performance (%) ↑				
Stride s	Dispatch r_d	Combine r_c	RDMA Reduction	Knowledge & Reasoning	Math	Code	Instruction Following	Overall (Δ)
Uncompressed ($d = 2048$)			1.00×	77.6	50.7	59.2	55.6	66.9
4	1280	1280	1.46×	76.2	49.1	58.1	54.5	65.5 (−1.4)
4	512	1280	1.84×	75.3	49.3	56.8	53.7	64.9 (−2.0)
4	512	1024	2.08×	74.3	47.8	55.2	53.5	63.8 (−3.1)
4	512	768	2.39×	73.6	47.2	54.4	53.3	63.1 (−3.8)
6	256	768	3.02×	68.3	42.5	50.4	50.1	58.3 (−8.6)
8	256	512	4.03×	64.2	40.9	50.7	51.0	55.7 (−11.2)

Table 3: **Task performance and RDMA reduction after adaptation.** Six EchoMoE configurations with the backbone adapted. RDMA reduction compares estimated BF16 bytes at EP32 over four nodes. Δ shows the change in overall score from the uncompressed model (66.92%) in percentage points. See Table 21 for frozen-backbone results and Table 14 for FP8 results.

End-to-End Training We perform the folding as described in Equation (4) and Equation (7) to obtain the expert specific initialization for $\widetilde{W}_{\text{in},i}$ and $\widetilde{W}_{\text{out},i}$. Because this initialization minimizes reconstruction error independently at each layer, the resulting errors can compound across the L layers. We therefore adapt the compressed model end-to-end using only a tiny amount of data, minimizing the KL divergence from the uncompressed model's output distribution and a small next-token cross-entropy loss. For faster, lower-cost experimentation, we also use a development setting (dev-mode) that skips folding and updates only the compression parameters (backbone frozen).

5 Experiments

5.1 Experimental Setup

Model We fine-tune Qwen3-30B-A3B-Base (Yang et al., 2025) on filtered Dolci-Instruct (Olmo et al., 2026) with tool-call samples excluded. The resulting checkpoint is the uncompressed model, the starting point for retrofitting and the distillation teacher.

Configurations and baselines A configuration (s, r_d, r_c) specifies the reference stride and the dispatch and combine code widths. We consider $r_d, r_c \in \{256, 512, 768, 1024, 1280\}$ and $s \in \{4, 6, 8, 12\}$. For analysis, we consider plain compression ($s = 0$; Section 4.1) and dispatch-only and combine-only variants, which compress only dispatch or combine, respectively.

Retrofitting We follow Section 4.4 using a mixture of Dolmino (Olmo et al. (2026)) and Dolci-Instruct (Olmo et al. (2026)). We use 4.32B tokens for adaptation and 1.04B in dev mode, approximately 8,500× and 34,000× fewer than Qwen3 pretraining respectively (Yang et al., 2025). More details about the training can be found in Appendix B.

Evaluation We evaluate task performance across 18 benchmarks spanning knowledge and reasoning, mathematics, coding and instruction following. We use MMLU (Hendrycks et al., 2021), HellaSwag Zellers et al. (2019) and GSM8K (Cobbe et al., 2021) in zero-shot chat-CoT format as the development suite. Section B.3 describes the full evaluation details.

Hardware and communication. We benchmark throughput on 4 DGX H100 nodes, each with 8 H100 SXM5 80GB GPUs and 8 ConnectX-7 400 Gb/s NICs. More details in Appendix F.

5.2 Task Performance and RDMA Reduction

EchoMoE reduces RDMA traffic by 46% with only 2.0 pp drop in task performance. The retrofitted $(4, 512, 1280)$ configuration achieves a 1.84× RDMA reduction (Table 3). More aggressive compression reaches 2.39× and 4.03× reductions with performance drops of 3.8 and 11.2 pp, respectively. Per-configuration results are provided in Appendix H.

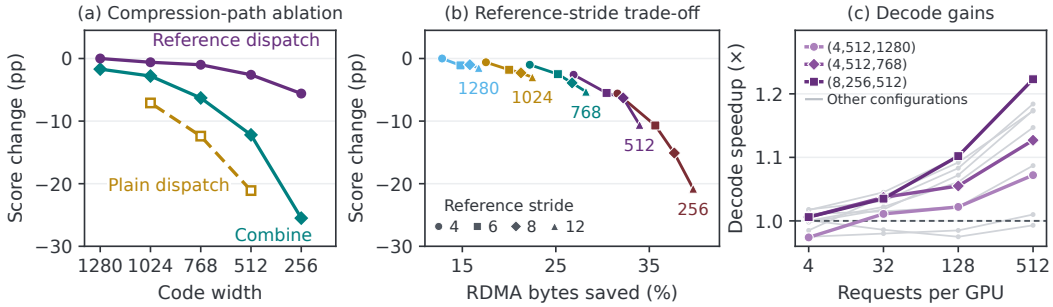


Figure 3: **(a)** Reference states substantially improve dispatch compression, consistently outperforming both plain dispatch and combine at matched code widths. Combine still outperforms plain dispatch, consistent with expert outputs being lower-dimensional, but requires larger code width. All results are single-path ablations, with the other communication path left uncompressed. **(b)** At a fixed RDMA-byte reduction, smaller reference strides with narrower code widths preserve higher task performance. **(c)** Decode gains increase across workloads with more requests per GPU. Input lengths are $128K$, $16K$, $4K$ and $1K$ tokens, respectively. Speedup uses time per output token for the 10% of requests prefilled last. The dashed line marks the uncompressed model.

EchoMoE outperforms plain compression at comparable RDMA reductions. At $2.08\times$ reduction, EchoMoE outperforms plain compression at $2.00\times$ by 5.0 pp with the backbone frozen and 2.2 pp after adaptation. At roughly $4\times$ RDMA reduction, adapted EchoMoE outperforms plain compression by 4.3 pp. Across all matched comparisons, the frontier EchoMoE configuration achieves better performance with the same or less RDMA traffic (Section C.2). Full-model adaptation consistently outperforms freezing the backbone, while FP8 transfer of the compressed representations incurs no measurable performance degradation (Section C.1, Table 14).

5.3 Ablation Studies of Each Compression Path

Reference states substantially improve dispatch compression. At width 512, plain dispatch loses 21.1 pp whereas reference dispatch at stride 4 loses only 2.6 pp (Figure 3a). This consistent advantage translates into a better RDMA–performance frontier for EchoMoE as shown in Figure 1. Tables 10 and 12 provide the ablations details.

Expert outputs are more compressible than dispatch activations. At the same code width, combine incurs substantially less degradation than plain dispatch, consistent with expert outputs having a lower-dimensional structure. Combine compression does require larger widths than reference-based dispatch to achieve comparable performance. We also explored augmenting combine with additional side information analogous to the dispatch reference state but observed similar performance (Section D.2); we therefore use direct combine compression throughout.

Smaller strides with narrower codes outperform larger strides with wider codes. The benefit of frequent reference updates grows as the dispatch code narrows (Figure 3b). At comparable RDMA savings, $(s, r_d) = (4, 512)$ consistently outperforms $(8, 768)$ across combine widths (Table 11), showing that communication is better spent on more frequent references than wider dispatch codes.

Efficient search for high-performing configurations. The isolated dispatch and combine performance drops are approximately additive (Figure 5), allowing us to estimate joint performance by summing their dev-mode drops (Appendix E). We retrospectively evaluate the estimation by selecting its highest-scoring configuration at each RDMA target, and comparing the selection with the empirical optimum. The selected configuration is within 0.5 pp of the empirical optimum for 94% of RDMA targets, with a maximum gap of 0.7 pp. This enables compute-efficient search for high-performing joint configurations without exhaustively training in the full Cartesian product.

5.4 Throughput, Latency and Inference Speedup

Prefill savings translate into higher throughput. With 4096 input tokens, 256 output tokens and 128 requests per GPU, EchoMoE improves throughput by 17.2% at $1.84\times$ RDMA reduction and by

42.4% at $4.03\times$ (Figure 1 bottom right). Time to first token falls by 17.7% and 34.9%, respectively. Profiling attributes these gains to lower communication and expert computation costs (Section G.3).

Decode gains depend on the workload. Decode speedups reach $1.22\times$ at 512 requests per GPU, while lower-concurrency workloads remain close to the uncompressed model (Figure 3c). As the number of tokens per step increases, communication constitutes a larger fraction of decode time, allowing reduced RDMA traffic to translate more directly into latency savings. Sections G.1 and G.2 detail the protocol and workload comparisons.

The implementation and throughput gain extends to larger MoE architectures. With $s = 4$, $r_d = d/4$ and $r_c = d/2$, throughput improves by $1.15\times$ on Qwen3-235B-A22B, $1.18\times$ on DeepSeek-V3 and $1.05\times$ on Kimi-K2.6 (Section G.4). Due to limited compute for retrofitting, these tests use full-sized models with random weights and report throughput improvements only.

6 Related Work

Communication-efficient MoE architectures. Several works reduce MoE communication by modifying the model architecture. LatentMoE (Elango et al., 2026) and BigMac (Jin et al., 2025) project routed activations into a lower-dimensional expert space, reducing dispatch and combine width. Multi-Head LatentMoE (Cui et al., 2026) and Federation of Experts (Abdurrahman et al., 2026) instead restructure expert partitioning and communication, while ScMoE (Cai et al., 2025) uses earlier activations to overlap expert communication with computation. These methods are designed for training new architectures from scratch. In contrast, EchoMoE retrofits existing MoEs, using earlier activations as references to compress communication while reconstructing current expert inputs.

Efficient expert-parallelism. DeepSpeed-MoE (Rajbhandari et al., 2022) uses hierarchical all-to-all communication, while Tutel (Hwang et al., 2023) combines adaptive parallelism with communication pipelining to improve interconnect efficiency. At the transport level, DeepEP (Zhao et al., 2025b) provides specialized dispatch and combine kernels with low-precision communication. These approaches optimize the movement of expert-parallel traffic. EchoMoE is complementary to these systems: it reduces the communication traffic and is integrated with DeepEP, while transport precision further reduces the cost per value. Fang et al. (2025); Eliseev & Mazur (2023); Yu et al. (2026) exploits cross-layer similarity in MoE models to accelerate expert fetching or offloading.

Activation and residual compression. Learned bottlenecks exploit low-dimensional activation structure; ResBM applies this idea to pipeline communication while preserving a low-rank identity path (Aboudib et al., 2026). Other methods exploit shared information: DIANA compresses gradients relative to estimates maintained across optimization steps (Mishchenko et al., 2019), while LSH-MoE communicates centroids of similar tokens within a layer (Nie et al., 2024). EchoMoE instead combines a learned bottleneck with an earlier-layer reference for the same token, exploiting cross-layer predictability within a forward pass. Unlike LSH-MoE, which adds token–centroid residuals to expert outputs, EchoMoE uses compressed residuals to reconstruct current expert inputs.

7 Conclusion and Future Work

We introduced EchoMoE, a lightweight architectural retrofit for reducing EP communication in existing MoE models. EchoMoE exploits the cross-layer predictability of MoE input by periodically caching a full-width activation from an earlier layer as a reference, communicating only a compressed change relative to this reference on the dispatch path, and compressing expert outputs on the combine path. On Qwen3-30B-A3B, EchoMoE reduces inter-node RDMA traffic by 46% with a 2.0 pp drop in task performance and improves inference throughput by 17.2%, while requiring approximately $8{,}500\times$ fewer training tokens than pretraining. Its communication-performance frontier can also be searched efficiently from isolated dispatch and combine evaluations. Overall, EchoMoE establishes cross-layer predictability as a practical mechanism for substantially reducing EP communication in existing MoE models.

While this work focuses on retrofitting pretrained models, EchoMoE could also be incorporated as a novel architecture for pre-training, where reducing representational redundancy may enable stronger performance. Its communication savings should also extend naturally to training.

AI USE STATEMENT

In this work, we used generative AI tools to modify scientific figures, create and edit software code (implement methods), edit the paper to improve readability. We have not used generative AI tools to design or provide feedback on research methodology or experiments, support qualitative and thematic data analysis, and interpret results. And the use of generative AI to generate synthetic data sets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, assist with translation, and clean and reformat dataset are not applicable to this work. We have reviewed all AI-assisted work. LLM-generated code was verified and tested for correctness by 2 authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The codebase for reproducing our results are available through an anonymous repository and will be made publicly available upon acceptance. Assumptions, experiment details, configurations, communication accounting, and complete results are available in the main text and appendix.

ACKNOWLEDGMENTS

Acknowledgments will be added upon acceptance.

REFERENCES

Muhammad Shahir Abdurrahman, Chun Deng, Azalia Mirhoseini, and Philip Levis. Federation of experts: Communication efficient distributed inference for large language models, 2026. URL https://arxiv.org/abs/2605.06206.

Alan Aboudib, Rodrigo Lopez Portillo A., Kalei Brady, and Steffen Cruz. ResBM: Residual bottleneck models for low-bandwidth pipeline parallelism, 2026. URL https://arxiv.org/abs/2604.11947.

Weilin Cai, Juyong Jiang, Le Qin, Junwei Cui, Sunghun Kim, and Jiayi Huang. Shortcut-connected expert parallelism for accelerating mixture-of-experts, 2025. URL https://arxiv.org/abs/2404.05019.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try ARC, the AI2 reasoning challenge, 2018. URL https://arxiv.org/abs/1803.05457.

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

Chenwei Cui, Rockwell Jackson, Benjamin Joseph Herrera, Ana M. Tárano, and Hannah Kerner. Multi-head latentmoe and head parallel: Communication-efficient and deterministic moe parallelism. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=M6h4vUgvl4.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J.

Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou,

Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang, Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Venmugil Elango, Nidhi Bhatia, Roger Waleffe, Rasoul Shafipour, Tomer Asida, Abhinav Khattar, Nave Assaf, Maximilian Golub, Joey Guman, Tiyasa Mitra, Ritchie Zhao, Ritika Borkar, Ran Zilberstein, Mostofa Patwary, Mohammad Shoeybi, and Bita Rouhani. Latentmoe: Toward optimal accuracy per flop and parameter in mixture of experts, 2026. URL https://arxiv.org/abs/2601.18089.

Artyom Eliseev and Denis Mazur. Fast inference of mixture-of-experts language models with offloading, 2023. URL https://arxiv.org/abs/2312.17238.

Zhiyuan Fang, Zicong Hong, Yuegui Huang, Yufeng Lyu, Wuhui Chen, Yue Yu, Fan Yu, and Zibin Zheng. Fate: Fast edge inference of mixture-of-experts models via cross-layer gate, 2025. URL https://arxiv.org/abs/2502.12224.

Aryo Pradipta Gema, Joshua Ong Jun Leang, Giwon Hong, Alessio Devoto, Alberto Carlo Maria Mancino, Rohit Saxena, Xuanli He, Yu Zhao, Xiaotang Du, Mohammad Reza Ghasemi Madani, Claire Barale, Robert McHardy, Joshua Harris, Jean Kaddour, Emile van Krieken, and Pasquale Minervini. Are we done with MMLU?, 2024. URL https://arxiv.org/abs/2406.04127.

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024. doi: 10.1109/MM.2024.3373763.

Xiangming Gu, Tianyu Pang, Chao Du, Qian Liu, Fengzhuo Zhang, Cunxiao Du, Ye Wang, and Min Lin. When attention sink emerges in language models: An empirical view. In *International Conference on Learning Representations*, 2025.

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

HMMT. Harvard–mit mathematics tournament, february 2025, 2025. URL https://www.hmmt.org/www/archive/282.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022. URL https://arxiv.org/abs/2203.15556.

Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, Joe Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. Tutel: Adaptive mixture-of-experts at scale, 2023. URL https://arxiv.org/abs/2206.03382.

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

Zewen Jin, Shengnan Wang, Jiaan Zhu, Hongrui Zhan, Youhui Bai, Lin Zhang, Zhenyu Ming, and Cheng Li. Bigmac: A communication-efficient mixture-of-experts model structure for fast training and inference, 2025. URL https://arxiv.org/abs/2502.16927.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, M. C., Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y. Charles, H. S. Che, Guanduo Chen, Guangyu Chen, Guanzheng Chen, Huarong Chen, Jia Chen, Jianlong Chen, Jun Chen, Kexin Chen, Peng Chen, Ruijue Chen, Wentao Chen, Xin Chen, Yang Chen, Yanru Chen, Yifei Chen, Yingjiang Chen, Yuankun Chen, Yujie Chen, Yutian Chen, Zhirong Chen, Dazhi Cheng, Yean Cheng, Jialei Cui, Jingbing Cui, Anqi Dai, Jiaqi Deng, Hao Ding, Rui Ding, Shaofeng Ding, Mengfan Dong, Mengnan Dong, Yuhao Dong, Yuxin Dong, Angang Du, Chenzhuang Du, Dikang Du, Jusen Du, Yulun Du, Yu Fan, Jing Feng, Qiulin Feng, Yichen Feng, Kelin Fu, Qiang Fu, Fuxuan Gao, Hongcheng Gao, Jingyue Gao, Tong Gao, Weijia Gao, Shangyi Geng, Jie Gong, Linhu Gong, Shengao Gong, Xiaochen Gong, Qizheng Gu, Yicheng Gu, Shuhao Guan, Haiqing Guo, Shiqi Guo, Xiang Guo, Zhengyan Guo, Beixi Hao, Wenxin Hao, Xiaoru Hao, Dailan He, Haotian He, Lehan He, Qi He, Weiran He, Xinran He, Xinyi He, Yibo He, Yunjia He, Chao Hong, Tiange Hong, Hao Hu, Jiaxi Hu, Ruikun Hu, Weiming Hu, Yangyang Hu, Zhenxing Hu, Liang Hua, Jinbin Huang, Ke Huang, Ruiyuan Huang, Siying Huang, Weixiao Huang, Yan Huang, Zhengjie Huang, Zhiqi Huang, Yulong Hui, Chaobo Jia, Yutong Jiang, Zhejun Jiang, Zuoyou Jiang, Wenyi Jin, Xinyi Jin, Yu Jing, Huanjun Kong, Guokun Lai, Aidi Li, Cheng Li, Chengyuan Li, Cong Li, Fang Li, Guanyu Li, Haoyang Li, Jia Li, Junxiong Li, Lei Li, Letian Li, Lincan Li, Weihong Li, Wentao Li, Xintong Li, Yang Li, Yishen Li, Yiwei Li, Yuxiao Li, Zhaowei Li, Zhaoxi Li, Zheming Li, Zhengxiao Li, Zhiyuan Li, Jiawei Lin, Xiaohan Lin, Yibo Lin, Zichao Lin, Ziyan Lin, Bill Liu, Boxiao Liu, Chuan Liu, Liang Liu, Shaowei Liu, Shudong Liu, Shuran Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yanming Liu, Yibo Liu, Yipeng Liu, Zhengying Liu, Zhiheng Liu, Enzhe Lu, Haoyu Lu, Linqiang Lu, Tingzhan Lu, Zhiyuan Lu, Aotian Luo, G. Luo, Junyu Luo, Yifan Luo, B. Lyu, Wenzhou Lyu, Shaoguang Mao, Yuan Mei, Xin Men, Minqing Ni, Yixuan Niu, Siyuan Pan, Shujun Peng, Zhangyang Qi, Ruoyu Qin, ZeChao Qin, Zeyu Qin, Haiquan Qiu, Jianxin Qiu, Jiezhong Qiu, Bowen Qu, Yuhao Qu, Zeyu Shang, Youbo Shao, Han Shen, Jincheng Shi, Juanfeng Shi, Lidong Shi, Shengyuan Shi, Wingchun Siu, Pengwei Song, Xiaoxi Song, Jianlin Su, Yunfeng Su, Zhaochen Su, Lin Sui, Jingsong Sun, Junyao Sun, Shaoning Sun, Shuzhe Sun, Tongyu Sun, Yujun Sun, Yunpeng Tai, Chuning Tang, Heyi Tang, Sirui Tang, Zecheng Tang, Chaoran Tian, Rongpeng Tian, Yu Tian, Wei Tu, Chensi Wang, Chuang Wang, Chunjie Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Hao Wang, Hao Wang, Huaqing Wang, Hui Wang, Jiayi Wang, Jinglong Wang, Jinhong Wang, Jiuzheng Wang, Linian Wang, Shaobo Wang, Shenzhi Wang, Shuyi Wang, Si Wang, Siyuan Wang, Tianfu Wang, Wenjue Wang, Xingran Wang, Xinmei Wang, Xinyuan Wang, Xusheng Wang, Yalin Wang, Yangkun Wang, Yao Wang, Yaoyu Wang, Yejie Wang, Yiqin Wang, Yucheng Wang, Yuzhi Wang, Zhaoji Wang, Zhaowei Wang, Zhengtao Wang, Zhenhao Wang, Zhongsheng Wang, Zifan Wang, Chu Wei, Ming Wei, Shouxin Wei, Zichen Wen, Fan Wu, Haoning Wu, Rucong Wu, Wenhao Wu, Xiaoxue Wu, Yingcong Wu, Yongqi Wu, Yuxin Wu, Zijian Wu, Xinglang Xian, Chenxuan Xiang, Yuye Xiang, Bocheng Xiao, Chenjun Xiao, Xin Xiao, Jin Xie, Xiaotong Xie, Yifeng Xie, Zhe Xie, Bowei Xing, Yiming Xiong, Baosheng Xu, Boyu Xu, Jiale Xu, Jianfan Xu, Jing Xu, Jinjing Xu, L. H. Xu, Qingtao Xu, Shuyao Xu, Suting Xu, Tiantian Xu, Tianxiang Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ye Xu, Yueni Xu, Ziyao Xu, Haonan Xue, Junjie Yan, Yaoyao Yan, Fan Yang, Guangyao Yang, Hao Yang, Junwei Yang, Ruoyu Yang, Wenjie Yang, Xiaofei Yang, Xinyu Yang, Yi Yang, Yiling Yang, Ying Yang, Yuchen Yang, Zhen Yang, Zhilin Yang, Zian Yang, Zuhao Yang, Haotian Yao, Dan Ye, Haoran Ye, Wenjie Ye, Zhanbo Ye, Bohong Yin, Haoxiang Yin, Xietong Yin, Chengzhen Yu, Haozhen Yu, Longhui Yu, Shengnan Yu, Shuying Yu, Tianxiang Yu, Enming Yuan, Mengjie Yuan, Tongtian Yue, Wei Yue, Yang Yue, Dunyuan Zha, Haobing Zhan, B. H. Zhang, Dehao Zhang, Fei Zhang, Hao Zhang, Haoyuan Zhang, Huanyu Zhang, Jiapei Zhang, Jiaxuan Zhang, Jin Zhang, Kaiyi Zhang, Miaozhen Zhang, Puqi Zhang, Qinglei Zhang, Rong Zhang, Rui Zhang, Shaoshuai Zhang, Shiyi Zhang, Xiaobin Zhang, Xiaoyun Zhang, Y. Zhang, Yangkun Zhang, Ye Zhang, Yichi Zhang, Yikun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Zijing Zhang, Bin Zhao, Chenguang Zhao, Feifan Zhao, Jinglun Zhao, Jinxiang Zhao, Shuai Zhao, Wenshuo Zhao, Xiangyu Zhao, Xuanle Zhao, Yikai Zhao, Zijia Zhao, Haozhi Zheng, Huabin Zheng, Ruihan Zheng, Shaojie Zheng, Tengyang Zheng, Haofeng Zhong, Lei Zhong, Longguang Zhong, M. Zhou, Qiankang Zhou, Runjie Zhou, Ruozhang Zhou, Xinyu Zhou, Yiqiao Zhou, Zaida Zhou, Jinguo Zhu, Liya Zhu, Xinhao Zhu,

Yangjunfeng Zhu, Yuxuan Zhu, Zhen Zhu, Chen Zhuang, Weiyu Zhuang, and Xinxing Zu. Kimi k3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. Infinigen: Efficient generative inference of large language models with dynamic kv cache management, 2024. URL https://arxiv.org/abs/2406.19707.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding, 2020. URL https://arxiv.org/abs/2006.16668.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023. URL https://arxiv.org/abs/2305.20050.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

MAA. American invitational mathematics examination (AIME), 2024. Mathematical Association of America, 2024. URL https://maa.org/maa-invitational-competitions/. AIME I and AIME II.

MAA. American invitational mathematics examination (AIME), 2025. Mathematical Association of America, 2025. URL https://maa.org/maa-invitational-competitions/. AIME I and AIME II.

Konstantin Mishchenko, Eduard Gorbunov, Martin Takáč, and Peter Richtárik. Distributed learning with compressed gradient differences, 2019. URL https://arxiv.org/abs/1901.09269.

Xiaonan Nie, Qibin Liu, Fangcheng Fu, Shenhan Zhu, Xupeng Miao, Xiaoyang Li, Yang Zhang, Shouda Liu, and Bin Cui. Lsh-moe: Communication-efficient moe training via locality-sensitive hashing, 2024. URL https://arxiv.org/abs/2411.08446.

Team Olmo, :, Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heineman, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, Jacob Morrison, Jake Poznanski, Kyle Lo, Luca Soldaini, Matt Jordan, Mayee Chen, Michael Noukhovitch, Nathan Lambert, Pete Walsh, Pradeep Dasigi, Robert Berry, Saumya Malik, Saurabh Shah, Scott Geng, Shane Arora, Shashank Gupta, Taira Anderson, Teng Xiao, Tyler Murray, Tyler Romero, Victoria Graf, Akari Asai, Akshita Bhagia, Alexander Wettig, Alisa Liu, Aman Rangapur, Chloe Anastasiades, Costa Huang, Dustin Schwenk, Harsh Trivedi, Ian Magnusson, Jaron Lochner, Jiacheng Liu, Lester James V. Miranda, Maarten Sap, Malia Morgan, Michael Schmitz, Michal Guerquin, Michael Wilson, Regan Huff, Ronan Le Bras, Rui Xin, Rulin Shao, Sam Skjonsberg, Shannon Zejiang Shen, Shuyue Stella Li, Tucker Wilde, Valentina Pyatkin, Will Merrill, Yapei Chang, Yuling Gu, Zhiyuan Zeng, Ashish Sabharwal, Luke Zettlemoyer, Pang Wei Koh, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. Olmo 3, 2026. URL https://arxiv.org/abs/2512.13961.

Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hannaneh Hajishirzi. Generalizing verifiable instruction following, 2025. URL https://arxiv.org/abs/2507.02833.

Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale, 2022. URL https://arxiv.org/abs/2201.05596.

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

Perplexity Research. Lower latency and higher throughput with multi-node deepseek deployment, Apr 2025. URL https://www.perplexity.ai/hub/blog/lower-latency-and-higher-throughput-with-multi-node-deepseek-deployment.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. URL https://arxiv.org/abs/1701.06538.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. Challenging BIG-Bench tasks and whether chain-of-thought can solve them, 2022. URL https://arxiv.org/abs/2210.09261.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark, 2024. URL https://arxiv.org/abs/2406.01574.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, 2024.

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

Enda Yu, Dezun Dong, Zhaoning Zhang, Zhe Bai, Weiling Yang, Haojie Wang, Dongsheng Li, Yongwei Wu, and Xiangke Liao. Layerscope: Predictive cross-layer scheduling for efficient multi-batch moe inference on legacy servers. In *Proceedings of the 40th ACM International Conference on Supercomputing*, ICS '26, pp. 881–892. ACM, 2026. doi: 10.1145/3797905.3807834. URL http://dx.doi.org/10.1145/3797905.3807834.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In Anna Korhonen, David Traum, and Lluís Màrquez (eds.), *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4791–4800, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1472. URL https://aclanthology.org/P19-1472/.

Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Huazuo Gao, Jiashi Li, Liyue Zhang, Panpan Huang, Shangyan Zhou, Shirong Ma, Wenfeng Liang, Ying He, Yuqing Wang, Yuxuan Liu, and Y.X. Wei. Insights into deepseek-v3: Scaling challenges and reflections on hardware for ai architectures. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, SIGARCH '25, pp. 1731–1745. ACM, 2025a. doi: 10.1145/3695053.3731412. URL http://dx.doi.org/10.1145/3695053.3731412.

Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025b.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models, 2023. URL https://arxiv.org/abs/2311.07911.

A Cross-Layer Similarity of MoE-Layer Inputs

Table 2 compares a token's input to one MoE layer with its input to an earlier MoE layer. The activation we compare is the normalized input to the MoE layer, which is what dispatch sends at the full width d. Each model runs one forward pass over 256 documents from the Dolci Instruct SFT Mixture (Olmo et al., 2026), tokenized by its own tokenizer and truncated to 2048 positions. The documents are short so each model contributes about 137K token positions, 74K on the half used to fit the predictor below and 63K on the half used to score it. The first position of every document is excluded because it is the attention sink (Xiao et al., 2024; Gu et al., 2025) whose norm exceeds every other position's by orders of magnitude.

Let $x_\ell \in \mathbb{R}^d$ denote a token's input to MoE layer ℓ and let $\ell - k$ be an earlier MoE layer. Table 2 reports a cosine similarity and a predictability for every such pair, both computed on the held-out half and averaged over all pairs of MoE layers at separation k. The cosine similarity between the two inputs is

$$\cos\left(x_{\ell-k}, x_\ell\right) = \frac{\langle x_{\ell-k}, x_\ell \rangle}{\|x_{\ell-k}\| \, \|x_\ell\|} \tag{11}$$

A receiver holding $x_{\ell-k}$ needs more than direction to reconstruct x_ℓ. Predictability is the share of x_ℓ that a predictor fixed in advance recovers from $x_{\ell-k}$. We fit one scale a_i and one shift b_i per dimension i on one half of the documents by least squares and score them on the held-out half as

$$R^2 = 1 - \frac{\sum_{t,i} \left(x_{\ell,i}^{(t)} - a_i x_{\ell-k,i}^{(t)} - b_i \right)^2}{\sum_{t,i} \left(x_{\ell,i}^{(t)} - \bar{x}_{\ell,i} \right)^2} \tag{12}$$

where t indexes the scored tokens and $\bar{x}_{\ell,i}$ is the mean of dimension i over the documents used for fitting. The two halves hold different documents so the score is what a predictor fitted in advance achieves on tokens it has not seen. The denominator measures spread around $\bar{x}_{\ell,i}$ so $x_{\ell-k}$ earns credit only for what it explains beyond a per-dimension constant that a receiver could hold without any transfer.

B Training and Evaluation Details

We use Megatron-LM (Shoeybi et al., 2020) for fine-tuning and compression training and vLLM (Kwon et al., 2023) for task evaluation. All compressed checkpoints are evaluated after folding the learned projections into the experts as described in Section 4.4.

B.1 Model and Compression Configurations

Uncompressed model. We fine-tune Qwen3-30B-A3B-Base (Yang et al., 2025) on Dolci-Instruct (Olmo et al., 2026) with tool-call samples excluded, using the Qwen3 Instruct chat template and tokenizer. The filtered dataset contains 1.9M conversations and 1.16B tokens. The resulting checkpoint provides the initial model weights and distillation targets for compression. Table 4 lists the fine-tuning hyperparameters.

Compression configurations. We consider reference strides $s \in \{0, 4, 6, 8, 12\}$ and code widths $r_d, r_c \in \{256, 512, 768, 1024, 1280\}$, resulting in 125 candidate joint configurations. Stride $s = 0$ denotes plain compression, which encodes dispatch activations directly. We train 40 joint configurations in dev-mode and adapt the full model for 19 of these configurations (Table 5). Dispatch-only and combine-only ablations leave the other transfer at full width. The combine ablations cover 5 widths and 3 decoder variants (Section D.2). All joint configurations use direct combine decoding.

Setting	Value
Loss function	Cross-entropy
Training epochs	2
Training steps	2230
Global batch size	128 packed sequences
Sequence length	8192 tokens
Training tokens	2.32B
Supervised tokens	1.45B
Learning rate (peak $\rightarrow$ final)	$2 \times 10^{-5} \rightarrow 2 \times 10^{-6}$
Warmup steps	200
Weight decay	0.1
Gradient clipping	1.0
Precision	BF16
Random seed	42

Table 4: **Fine-tuning hyperparameters for the uncompressed model.** Only response tokens are supervised. Token counts cover both epochs.

Compression setting	Dev-mode	Adaptation
Dispatch-only	23	–
Combine-only	15	–
Joint	40	19

Table 5: **Number of trained configurations.** The 19 adapted configurations are a subset of the 40 joint configurations trained in dev-mode. Dashes indicate settings not trained.

B.2 Initialization and Adaptation

Initialization. We fit the compression parameters on 512K token positions from 64K Dolci-Instruct documents (Section 4.4). We evaluate initialization KL on disjoint Dolci-Instruct documents using the training distillation loss.

Training settings. Table 6 lists the hyperparameters and training data budgets for dev-mode and full-model adaptation. The larger training budget for full-model adaptation makes it difficult to isolate the effect of updating the backbone.

Setting	Dev-mode	Adaptation
Trainable parameters	Compression parameters	All model parameters
Folding	After training	Before training
Training steps	1,000	4,152
Warmup steps	50	208
Training tokens	1.04B	4.32B
Dolmino	481M	2.00B
Dolci-Instruct	560M	2.32B
Initial Dolci-Instruct share	46.50%	46.45%
Learning rate (peak $\rightarrow$ final)		
Dense backbone	–	$2 \times 10^{-5} \rightarrow 2 \times 10^{-6}$
Router	–	$1 \times 10^{-5} \rightarrow 1 \times 10^{-6}$
Experts and folded factors	–	$3 \times 10^{-5} \rightarrow 3 \times 10^{-6}$

Table 6: **Training settings for dev-mode and adaptation.** Token counts are per configuration. Dolci-Instruct share is the fraction of packed sequences drawn from that dataset. Dashes indicate frozen parameter groups.

Training data and objective. We mix Dolmino with Dolci-Instruct (Olmo et al., 2026). The Dolci-Instruct share is constant for the first $2/3$ of training then increases linearly to 90%. To avoid keeping

a separate copy of the model in memory as a teacher during training, we precompute and cache its top-1024 logits and the remaining probability mass. Distillation compares teacher and student probabilities for these tokens and a tail bucket containing the remaining mass. The objective combines this KL loss with cross-entropy on Dolci-Instruct responses. Table 7 lists the hyperparameters common to dev-mode and full-model adaptation.

Setting	Value
Global batch size	128 packed sequences
Sequence length	8192 tokens
Distillation temperature	1
KL loss weight	1.0
Cross-entropy loss weight	0.01
Optimizer	Adam
Adam (β_1, β_2)	$(0.9, 0.999)$
Adam ϵ	10^{-8}
Learning rate (compression)	$5 \times 10^{-5} \rightarrow 5 \times 10^{-6}$
Learning-rate schedule	Linear warmup and cosine decay
Warmup fraction	5%
Weight decay	0
Gradient clipping	1.0
Precision	BF16
Random seed	42

Table 7: **Hyperparameters common to dev-mode and full-model adaptation.** The compression learning rate (peak $\rightarrow$ final) applies to all compression parameters in both the dispatch and combine paths.

B.3 Task Evaluation

Evaluation setup. We evaluate models with `lm-eval` 0.4.12 and EvalPlus 0.3.1, using vLLM 0.29.0 (Kwon et al., 2023) as the inference backend. All evaluations run on 8 GPUs with EP$=8$, TP$=1$ and DP$=8$, a context limit of 32768 tokens and seed 42. Each configuration is trained and evaluated with a single seed.

Evaluation suite. We evaluate all trained configurations on 18 benchmarks using zero-shot chat and few-shot non-chat prompts, with 9 benchmarks evaluated in both formats. Table 8 lists all benchmarks grouped by domain including their prompting formats.

Development suite. We use zero-shot chat-CoT evaluations of MMLU, HellaSwag and GSM8K for development. Their mean score for dispatch-only and combine-only configurations guides joint configuration selection (Appendix E).

Chat evaluation. We use the chat template with thinking mode disabled and decode with temperature 0.7, top-$p = 0.8$, top-$k = 20$ and presence penalty 1.5. We generate 1 sample per question with a maximum of 8192 new tokens. For AIME and HMMT, we increase the sample count to 64 and the token limit to 24576. For GPQA-Diamond, we increase the sample count to 10. For LiveCodeBench, we generate 10 samples per problem, lower the temperature to 0.2, set top-$p = 0.95$, disable top-k filtering and increase the token limit to 16384. For IFEval and IFBench, we use greedy decoding with token limits of 1280 and 2048, respectively.

Few-shot evaluation. We use prompts with in-context examples and score answer-option log-likelihoods for MMLU, ARC-Easy, ARC-Challenge and HellaSwag. For the remaining few-shot benchmarks, we use greedy decoding with a maximum of 2048 new tokens for MMLU-Pro, GSM8K and MATH-500, 1024 for BBH and 32 for MMLU-Redux. We select in-context examples using seed 42.

Scoring. For tasks with multiple samples, we average correctness over samples and then questions. AIME and HMMT use `math_verify` on boxed answers. Code benchmarks report pass@1, with

Benchmark	Non-chat prompt	Chat prompt
Knowledge and reasoning		
MMLU (Hendrycks et al., 2021)	5-shot	0-shot, CoT
MMLU-Pro (Wang et al., 2024)	5-shot, CoT	0-shot, CoT
MMLU-Redux (Gema et al., 2024)	5-shot	0-shot, CoT
ARC-Easy (Clark et al., 2018)	5-shot	0-shot, CoT
ARC-Challenge (Clark et al., 2018)	5-shot	0-shot, CoT
HellaSwag (Zellers et al., 2019)	5-shot	0-shot, CoT
BBH (Suzgun et al., 2022)	3-shot, CoT	0-shot, CoT
GPQA-Diamond (Rein et al., 2023)	–	0-shot, CoT
Mathematics		
GSM8K (Cobbe et al., 2021)	5-shot, CoT	0-shot, CoT
MATH-500 (Lightman et al., 2023)	4-shot, CoT	0-shot, CoT
AIME 2024 (MAA, 2024)	–	0-shot, CoT
AIME 2025 (MAA, 2025)	–	0-shot, CoT
HMMT Feb. 2025 (HMMT, 2025)	–	0-shot, CoT
Coding		
HumanEval+ (Liu et al., 2023)	–	0-shot
MBPP+ (Liu et al., 2023)	–	0-shot
LiveCodeBench (Jain et al., 2024)	–	0-shot
Instruction following		
IFEval (Zhou et al., 2023)	–	0-shot
IFBench (Pyatkin et al., 2025)	–	0-shot

Table 8: **Benchmarks and prompt formats used for evaluation grouped by domain.** The non-chat and chat columns specify the number of in-context examples and whether chain-of-thought (CoT) prompting is used.

164 HumanEval+ and 378 MBPP+ problems. IFEval reports strict prompt-level accuracy. The remaining benchmarks report accuracy.

Scoring. For tasks with multiple samples per question, we score each question by the fraction of correct samples and report the average across questions. We verify boxed answers for AIME and HMMT using `math_verify`. We report pass@1 for code benchmarks, with 164 HumanEval+ and 378 MBPP+ problems. We use strict prompt-level accuracy for IFEval and accuracy for the remaining benchmarks.

Uncompressed reference. We report score changes relative to the uncompressed model's score of 66.92% with the standard vLLM inference engine. For the BF16–FP8 comparison, we evaluate all models with our vLLM integration, which supports a custom DeepEP-based FP8 combine kernel. The uncompressed model retains full-width activations and scores 66.63% in BF16 (Table 14).

C Adaptation and Baseline Comparisons

C.1 Dev-Mode and Full-Model Adaptation

Full-model adaptation achieves similar task performance with less RDMA bytes than dev-mode. It outperforms dev-mode in 18 of the 19 joint configurations, with a 10.4 pp improvement at $4.03\times$ RDMA reduction (Figure 4). For example, adapted $(4, 512, 768)$ achieves 63.08% overall task performance at $2.39\times$ RDMA reduction, compared with 63.12% at $1.89\times$ for dev-mode $(4, 768, 1024)$. The adapted configuration therefore sends 21% less RDMA bytes at comparable task performance. Tables 21 and 22 report the results for both settings.

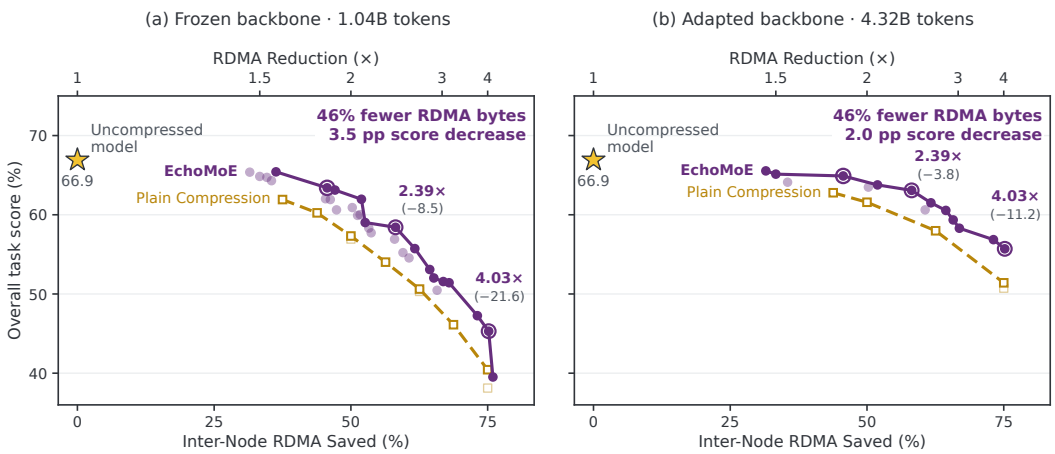


Figure 4: **Full-model adaptation improves task performance at comparable RDMA savings.** EchoMoE outperforms plain compression in both dev-mode **(a)** and full-model adaptation **(b)**. Lines show the empirical Pareto frontiers over 40 dev-mode and 19 adapted joint configurations, with selected configurations from Figure 1 highlighted. BF16 RDMA savings are calculated over 4 nodes EP32 group, including reference transfers.

C.2 Comparison with Plain Compression

EchoMoE improves task performance within the same RDMA budget. EchoMoE outperforms plain compression in all 10 dev-mode and 5 adapted comparisons, with improvements of up to 7.2 and 5.0 pp, respectively as shown in Table 9. Each comparison allows the EchoMoE stride and both code widths to vary while restricting RDMA bytes to the plain-compression budget.

Plain compression				EchoMoE					
Configuration				Configuration					
Dispatch r_d	Combine r_c	RDMA Reduction	Overall (%)	Stride s	Dispatch r_d	Combine r_c	RDMA Reduction	Overall (%)	Δ (pp)
Dev-mode									
1280	1280	1.60×	61.9	4	512	1280	1.84×	63.4	+1.5
1280	1024	1.78×	60.2	4	512	1280	1.84×	63.4	+3.2
1024	1024	2.00×	56.9	4	512	1024	2.08×	62.0	+5.0
1280	768	2.00×	57.3	4	512	1024	2.08×	62.0	+4.6
1024	768	2.29×	54.0	4	512	768	2.39×	58.4	+4.4
1024	512	2.67×	50.6	4	512	512	2.81×	53.1	+2.5
768	768	2.67×	50.3	4	512	512	2.81×	53.1	+2.8
768	512	3.20×	46.1	6	256	512	3.72×	47.3	+1.1
512	512	4.00×	40.4	8	256	512	4.03×	45.3	+4.9
768	256	4.00×	38.1	8	256	512	4.03×	45.3	+7.2
Full-model adaptation									
1280	1024	1.78×	62.8	4	512	1280	1.84×	64.9	+2.1
1024	1024	2.00×	61.6	4	512	1024	2.08×	63.8	+2.2
768	768	2.67×	58.0	4	512	512	2.81×	60.6	+2.6
512	512	4.00×	51.4	8	256	512	4.03×	55.7	+4.3
768	256	4.00×	50.7	8	256	512	4.03×	55.7	+5.0

Table 9: **EchoMoE and plain compression at matched RDMA budgets.** Each plain configuration is paired with the highest-scoring tested EchoMoE configuration that sends no more BF16 RDMA bytes at EP= 32 over 4 nodes, counting reference transfers. Scores report overall task performance. Δ shows the EchoMoE score minus the plain-compression score, in percentage points.

D COMPRESSION ABLATIONS

D.1 REFERENCE STRIDE AND DISPATCH WIDTH

Smaller dispatch widths benefit more from frequent reference updates. Increasing the reference stride from 4 to 12 reduces task performance by 1.5 pp at width 1280, compared with 15.2 pp at width 256 (see Table 10). Dispatch compression with a reference consistently outperforms plain dispatch at all 3 compression widths evaluated with both methods. Across all 23 configurations, we keep combine at full width and follow the dev-mode training recipe.

Configuration	Overall drop (pp)				
Dispatch r_d	Reference stride s				Plain dispatch
	4	6	8	12	
1280	0.0	1.1	1.0	1.5	–
1024	0.6	1.8	2.3	3.0	7.1
768	1.0	2.5	3.9	5.3	12.4
512	2.6	5.5	6.3	10.6	21.1
256	5.6	10.7	15.1	20.8	–

Table 10: **Dispatch compression across code widths and reference strides.** Overall score drops are relative to the uncompressed model (66.92%), in percentage points. Dispatch compression with a reference is compared with plain dispatch at matched code widths. All configurations follow the dev-mode training recipe with combine at full width.

More frequent reference updates compensate for smaller dispatch codes. At fixed combine width, $(s, r_d) = (4, 512)$ outperforms $(8, 768)$ by up to 1.9 pp across all 3 comparisons while sending slightly less RDMA bytes (see Table 11). These results suggest that, at similar RDMA budgets, updating the reference more frequently can preserve task performance better than increasing dispatch width.

Configuration			RDMA Reduction	Overall (%)	Δ (pp)
Stride s	Dispatch r_d	Combine r_c			
8	768	1280	1.83×	62.0	–
4	512	1280	1.84×	63.4	+1.4
8	768	1024	2.07×	60.0	–
4	512	1024	2.08×	62.0	+1.9
8	768	768	2.38×	56.9	–
4	512	768	2.39×	58.4	+1.5

Table 11: **Reference stride and dispatch width at comparable RDMA reductions.** Both configurations in each comparison use the same combine width and follow the dev-mode training recipe. RDMA reductions are calculated for BF16 dispatch, combine and reference transfers at EP32 group across 4 nodes. Δ shows the improvement in overall task performance for the stride-4 configuration relative to its stride-8 counterpart, in percentage points.

Less frequent reference updates leave larger residuals. At initialization, the residual is $\delta_\ell = x_\ell - a \odot z$. Averaged over compressed layers, its relative energy $\mathbb{E}\|\delta_\ell\|^2/\mathbb{E}\|x_\ell\|^2$ increases from 26.6% at stride 4 to 52.3% at stride 12. At each code width, the fraction of residual energy lost during compression, $\mathbb{E}\|\delta_\ell - D_d C_d \delta_\ell\|^2/\mathbb{E}\|\delta_\ell\|^2$, varies by less than 2 pp across strides. This suggests that the higher reconstruction error primarily reflects the larger residual being compressed.

D.2 COMBINE DECODER VARIANTS

Bias corrections provide no consistent improvement over direct combine decoding. Across 5 combine widths, per-expert bias and gated-bias corrections change overall task performance by at most 0.4 pp (see Table 12). Both variants add parameters and computation without reducing RDMA bytes. We therefore use direct combine decoding in all joint configurations.

Configuration	Direct decoding	Change from direct decoding	
Combine r_c	Overall (%)	Bias Δ (pp)	Gated bias Δ (pp)
Uncompressed	66.9	–	–
1280	65.2	+0.2	+0.2
1024	64.1	−0.2	−0.3
768	60.6	+0.2	+0.4
512	54.7	+0.2	+0.1
256	41.4	+0.1	0.0

Table 12: **Combine decoder variants at fixed code width.** All configurations use dev-mode with dispatch at full width. Overall reports task performance with direct decoding. Δ shows the change in task performance relative to direct decoding at the same combine width, in percentage points.

Decoder variants. Let $c = \sum_j c_c^{(j)}$ be the combined code, $w \in \mathbb{R}^N$ the routing weights with zeros for unselected experts and x_ℓ the input retained by the owner rank. Direct decoding returns $D_c c$ (Equation (6)). The bias and gated-bias variants add trainable per-expert offsets $o_i \in \mathbb{R}^d$:

$$\hat{y}_{\text{bias}} = D_c c + \sum_i w_i o_i \tag{13}$$

$$\hat{y}_{\text{gate}} = D_c c + \sum_i w_i (1 + \alpha_i) o_i \tag{14}$$

$$\alpha = U \,\text{SiLU}\big(W_x \tilde{x}_\ell + W_c c/\sigma_c + E_{\text{exp}}^\top w\big)$$

where $\tilde{x}_\ell$ is the RMS-normalized input and σ_c is the code normalization scale. The gate uses $W_x \in \mathbb{R}^{q \times d}$, $W_c \in \mathbb{R}^{q \times r_c}$, expert embeddings $E_{\text{exp}} \in \mathbb{R}^{N \times q}$ and an output projection $U \in \mathbb{R}^{N \times q}$, producing $\alpha \in \mathbb{R}^N$, with $q = N = 128$.

Initialization. We jointly fit $\bar{Y} \in \mathbb{R}^{N \times d}$ so that $\bar{Y}^\top w$ predicts the combined expert output y, then initialize each offset from the reconstruction residual of its fitted vector,

$$\bar{Y} = \big(\mathbb{E}[ww^\top] + \epsilon I\big)^{-1} \mathbb{E}[wy^\top], \qquad o_i^{(0)} = (I - D_c C_c)\bar{y}_i \tag{15}$$

where $\bar{y}_i^\top$ is row i of $\bar{Y}$ and $\epsilon = 10^{-6}$. We initialize the code normalization scale as

$$\sigma_c^2 = \text{tr}(C_c \Sigma_y C_c^\top)/r_c, \qquad \Sigma_y = \mathbb{E}[yy^\top] \tag{16}$$

We initialize W_x, W_c and E_{exp} randomly with seed 42 and set $U = 0$, so the bias and gated-bias variants produce identical outputs at initialization.

E CONFIGURATION SELECTION

Selection rule. Let Q_{dev} denote mean task performance on development task suite. We estimate joint performance by summing the drops from separate dispatch and combine ablations:

$$\widehat{Q}_{\text{dev}}(s, r_d, r_c) = Q^{\text{d}}_{\text{dev}}(s, r_d) + Q^{\text{c}}_{\text{dev}}(r_c) - Q^{0}_{\text{dev}} \tag{17}$$

where $Q^{\text{d}}_{\text{dev}}$ and $Q^{\text{c}}_{\text{dev}}$ denote performance with only dispatch or combine compressed, respectively, and Q^{0}_{dev} is the uncompressed model's performance. Combine uses direct decoding and $s = 0$ denotes plain dispatch.

Development scores identify competitive joint configurations. We evaluate selection retrospectively on 37 joint configurations trained in dev-mode with corresponding dispatch and combine ablations. Their 35 distinct RDMA reductions define the targets. For each target, we select the configuration with the highest predicted score among those that meet or exceed that reduction target, then compare its full-suite performance with the best tested configuration under the same constraint. Selected configurations are within 0.5 pp at 33 of 35 targets (94%), with a maximum gap of 0.70 pp.

Excluding the 3 development evaluations from the full-suite average preserves the 33 of 35 targets within 0.5 pp, with a maximum gap of 0.75 pp. This check uses rounded per-task results and retains the few-shot evaluations of these benchmarks. These results assess selection within the trained subset, which was itself chosen using the separate ablations, and do not establish optimality over the full configuration grid.

Performance drops are approximately additive at moderate compression. We also assess additivity using dispatch, combine and joint performance drops on the full evaluation suite (Figure 5). Across the 25 configurations with predicted drops of at most 15 pp, the mean absolute error is 0.50 pp and the maximum is 1.13 pp. The approximation becomes less accurate at higher compression, where the sum increasingly overestimates the joint drop.

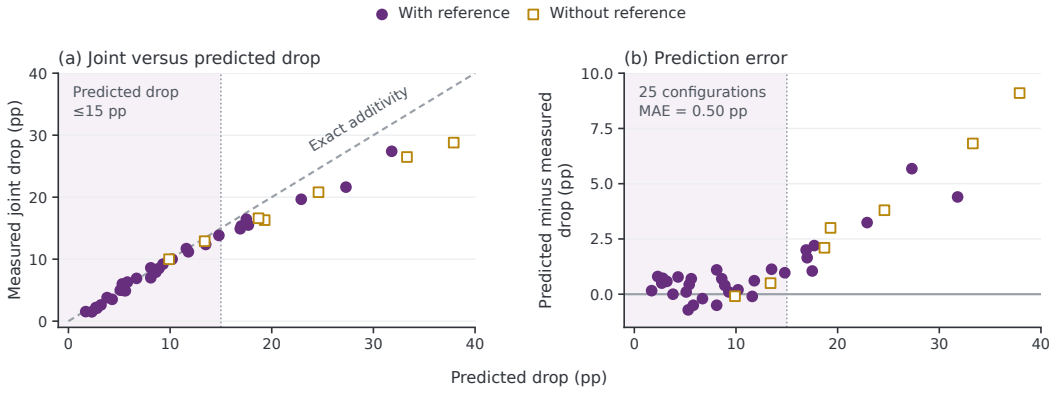


Figure 5: **Dispatch and combine performance drops are approximately additive at moderate compression.** Predictions sum the separate drops on the full evaluation suite, with all configurations trained in dev-mode. **(a)** Predicted and joint performance drops. **(b)** Prediction error, computed as predicted minus joint drop. Mean absolute error is 0.50 pp for the 25 configurations with predicted drops of at most 15 pp.

F COMMUNICATION AND MODEL COSTS

F.1 RDMA REDUCTION ACROSS EP TOPOLOGIES

RDMA calculation. BF16 dispatch and combine transfer one row per token per destination node, with expert outputs combined within each node before transfer. Reference transfers are also counted per destination node. At EP32 group across 4 nodes, routed transfers reach 2.73 remote nodes per token on average. These estimates exclude protocol overhead and collective startup latency and remain fixed for a given configuration, topology and routing trace.

Reference transfers reduce savings at larger EP sizes. Reference updates must reach more nodes as the EP group grows, while routed transfers reach only nodes hosting selected experts. For $(4, 512, 768)$, RDMA reduction falls from $2.39\times$ at EP32 to $2.15\times$ at EP64 (Table 13). Plain compression retains the same reduction factor at both topologies. At EP64, EchoMoE outperforms plain compression in all 8 available matched dev-mode comparisons by up to 2.1 pp, but the trained configurations provide no counterpart for the 2 plain configurations at $4\times$ reduction.

Configuration			BF16 RDMA Reduction	
Stride s	Dispatch r_d	Combine r_c	EP$=32$ 4 nodes	EP$=64$ 8 nodes
4	1280	1280	$1.46\times$	$1.37\times$
4	512	1280	$1.84\times$	$1.69\times$
4	512	1024	$2.08\times$	$1.89\times$
4	512	768	$2.39\times$	$2.15\times$
8	256	512	$4.03\times$	$3.66\times$

Table 13: **RDMA reduction across EP topologies.** Each configuration is compared with uncompressed BF16 transfers at the same topology.

At stride 4, recorded reference destinations within each layer group average 3.0 remote nodes at EP32 and 6.906 at EP64. Broadcasting references to all remote nodes, as assumed in Equation (8) would increase reference bytes at EP64 by 1.4%, with a smaller relative increase in total RDMA bytes.

F.2 FP8 COMMUNICATION

FP8 and BF16 achieve similar task performance. Across the 19 adapted checkpoints, using FP8 for dispatch, reference and combine transfers changes overall task performance by at most 0.29 pp (Table 14). Both evaluations use the same vLLM backend and the uncompressed model retains full-width activations.

Configuration			Overall (%)		
Stride s	Dispatch r_d	Combine r_c	BF16	FP8	Δ (pp)
Uncompressed			66.63	66.83	+0.20
EchoMoE					
4	1280	1280	65.54	65.41	−0.13
6	1280	1280	65.13	65.17	+0.04
12	1280	1280	64.11	63.92	−0.19
4	512	1280	64.90	64.86	−0.04
6	768	1024	63.49	63.42	−0.07
4	512	1024	63.77	63.65	−0.13
4	512	768	63.08	62.99	−0.09
6	256	1024	60.62	60.71	+0.09
6	512	768	61.51	61.45	−0.06
4	512	512	60.55	60.76	+0.21
12	768	512	59.35	59.32	−0.03
6	256	768	58.30	58.48	+0.18
6	256	512	56.86	56.70	−0.16
8	256	512	55.70	55.56	−0.15
Plain compression					
–	1280	1024	62.78	63.03	+0.25
–	1024	1024	61.58	61.28	−0.29
–	768	768	57.98	57.91	−0.07
–	512	512	51.41	51.42	+0.01
–	768	256	50.73	50.63	−0.11

Table 14: **Task performance with BF16 and FP8 communication.** Overall task performance for the same 19 adapted configurations using BF16 or FP8 for dispatch, reference and combine transfers. Δ shows the change from BF16 to FP8 in percentage points, computed before rounding.

FP8 reduces bytes per activation row. With one FP32 scale per 128 elements, FP8 values and scales occupy $33/64$ of the corresponding BF16 row, saving 48.4%. This ratio excludes routing metadata. FP8 dispatch sends one row per destination node but the custom combine kernel sends rows per destination rank and can add padding under CUDA graphs. The per-row saving therefore does not directly determine the total RDMA reduction.

F.3 Parameter, Memory and Computation Costs

Folding can reduce the total parameter count. Expert input width increases from d to $d + r_d$ at non-reference layers, while output width decreases from d to r_c at every layer. Shared projections on the owner rank contribute additional parameters. The $(4, 512, 1280)$ configuration contains 30.70B parameters, only 0.5% more than the uncompressed model. Reducing the code widths brings the total down to 25.47B for $(8, 256, 512)$ (Table 15).

Reference-buffer memory grows with the EP degree. Each rank reuses a single full-width reference buffer across layer groups. At $d = 2048$, storing a BF16 reference requires 4096 bytes per token. The buffer capacity is the engine's per-rank batched-token limit multiplied by the EP degree. With a limit of 8192 tokens per rank, the buffer occupies 1.00 GiB at EP32 and 2.00 GiB at EP64 on each rank, with staging buffers requiring additional memory.

Reducing combine width can lower total matrix computation. With expert intermediate width $f = 768$ and $K = 8$ active experts, the uncompressed model requires $3Kdf$ multiply-accumulates (MACs) per token and MoE layer for expert computation. For reference-based dispatch, the expert and owner matrix costs, averaged over layers and normalized by this uncompressed cost, are

$$\eta_{\text{expert}} = \frac{2d + 2r_d(1 - 1/s) + r_c}{3d}, \qquad \eta_{\text{owner}} = \frac{r_d(1 - 1/s) + r_c}{3Kf} \tag{18}$$

η_{expert} represents the relative cost of the folded input and output projections and η_{owner} that of dispatch encoding and combine decoding. At $(4, 512, 1280)$, their combined cost is 9.0% higher than the uncompressed expert computation. Reducing r_c to 768 lowers this cost to 2.1% below the uncompressed level. These counts exclude attention, routing, nonlinearities, elementwise reference operations and communication. Appendix G reports end-to-end inference performance.

Configuration					Relative Matrix MACs		
Stride s	Dispatch r_d	Combine r_c	BF16 RDMA Reduction	Parameters (B)	Expert	Owner	Total
Uncompressed			1.00×	30.53	1.000	–	1.000
4	512	1280	1.84×	30.70	1.000	0.090	1.090
4	512	768	2.39×	28.23	0.917	0.062	0.979
6	256	768	3.02×	26.60	0.861	0.053	0.914
8	256	512	4.03×	25.47	0.823	0.040	0.863

Table 15: **Parameter counts and matrix computation after folding.** Parameter counts cover the full model with shared owner projections. Expert and Owner report MACs relative to the uncompressed expert computation and Total reports their sum. RDMA reductions cover BF16 dispatch, combine and reference transfers at EP32 group across 4 nodes.

G Inference Benchmarks

G.1 Benchmark Setup

Hardware and runtime. We use 4 DGX H100 nodes, each with 8 H100 SXM5 80GB GPUs and 8 ConnectX-7 400 Gb/s NICs. Inference uses vLLM and DeepEP with EP32, TP1 and DP32. Each GPU owns its requests and stores 4 experts per layer for Qwen3-30B-A3B. We benchmark the same adapted checkpoints used for task evaluation. EchoMoE and the uncompressed model share the hardware allocation, engine settings (Table 16), inactive-request handling and routing compaction.

Setting	Value
Prefill scheduling	Chunked prefill
Token budget per step	4096
Activation transfers	BF16
Collectives	DeepEP V2, hybrid mode
DP synchronization	NCCL
CUDA graphs	Decode steps
`torch.compile`	Disabled
End-of-sequence stopping	Disabled
Warmup repetitions	1
Timed repetitions	2

Table 16: **Inference settings for Qwen3-30B-A3B.** EchoMoE and the uncompressed model use the same settings across the 5 workloads in Table 17. Settings for larger architectures are listed in Section G.4.

Throughput and latency. Both models receive identical random-token prompts, seeded by rank and request. We submit all requests together and generate a fixed number of output tokens. Throughput is the output-token count divided by batch completion time. Time to first token (TTFT) measures the interval from submission to the first generated token, including queueing. We average timings over 2 repetitions after 1 warmup. The reported repetitions agree within 0.5%, with settings affected by network contention repeated. These timings describe batch completion rather than steady-state request arrivals.

Decode latency. Time per output token (TPOT) divides the interval between the first and last generated tokens by the number of remaining output tokens. To reduce overlap with prefill, we report mean TPOT for the 10% of requests with the latest first token, selected across all 32 ranks within each repetition. We then average over repetitions. These requests generate mostly after prefill, although concurrency decreases as requests finish. This statistic retains host overhead and differs from the GPU-step timings in Section G.3.

Reference transfers. At reference layers, the Qwen3-30B-A3B implementation uses NCCL all-gather to publish full-width states and reduce-scatter to return expert outputs. Other layers use DeepEP at the configured code widths. RDMA reductions follow the tensor sizes and routing traces in Appendix F, with reference publication once per destination node. They do not measure the actual network bytes of these collectives.

G.2 Throughput and Latency Across Workloads

Table 17 reports all 11 configurations across 5 workloads. For the 128000-token workload, both models use YaRN scaling with factor 4 over 32768 positions.

Longer generation reduces the throughput benefit of faster prefill. For $(4, 512, 768)$ at 512 requests per GPU, TTFT improves by $1.34\times$ with 1024 input and 128 output tokens, and by $1.35\times$ with 128 input and 1024 output tokens. Despite similar TTFT improvements, throughput improves by $1.30\times$ and $1.14\times$, respectively. The longer decode phase reduces the contribution of faster prompt processing to total completion time.

Decode improvements depend on the workload. The $(4, 512, 1280)$ configuration has a decode speedup of $0.97\times$ with 128000 input tokens and 4 requests per GPU, compared with $1.07\times$ with

1024 input tokens and 512 requests. Stronger compression reaches $1.22\times$ in the latter workload, while dispatch width 1280 provides little decode benefit. Context length and submitted request count change together, so these comparisons do not isolate the effect of concurrency.

Configuration			Input / output tokens (requests per GPU)				
Stride s	Dispatch r_d	Combine r_c	1024/128 (512)	4096/256 (128)	16384/512 (32)	128000/512 (4)	128/1024 (512)
Throughput speedup (×)							
4	1280	1280	1.043	1.037	1.032	1.025	1.005
6	1280	1280	1.048	1.041	1.033	1.023	1.024
4	512	1280	1.190	1.172	1.149	1.093	1.078
4	512	1024	1.241	1.218	1.188	1.120	1.098
4	512	768	1.301	1.274	1.235	1.151	1.143
6	512	768	1.327	1.291	1.241	1.155	1.142
4	512	512	1.378	1.338	1.279	1.180	1.185
12	768	512	1.330	1.293	1.241	1.156	1.165
6	256	768	1.388	1.356	1.297	1.184	1.166
6	256	512	1.464	1.414	1.342	1.218	1.206
8	256	512	1.474	1.424	1.341	1.215	1.213
TTFT speedup (×)							
4	1280	1280	1.054	1.053	1.049	1.030	1.049
6	1280	1280	1.056	1.057	1.052	1.033	1.056
4	512	1280	1.214	1.215	1.208	1.121	1.206
4	512	1024	1.276	1.278	1.265	1.145	1.279
4	512	768	1.341	1.343	1.326	1.185	1.348
6	512	768	1.369	1.364	1.344	1.193	1.360
4	512	512	1.423	1.423	1.400	1.220	1.405
12	768	512	1.364	1.364	1.346	1.194	1.353
6	256	768	1.439	1.443	1.424	1.234	1.431
6	256	512	1.522	1.522	1.495	1.269	1.515
8	256	512	1.536	1.536	1.498	1.268	1.534
Decode TPOT speedup (×)							
4	1280	1280	0.993	0.975	0.986	1.001	1.002
6	1280	1280	1.010	0.985	0.980	0.975	1.021
4	512	1280	1.072	1.022	1.011	0.974	1.068
4	512	1024	1.087	1.022	1.015	1.011	1.085
4	512	768	1.127	1.055	1.037	1.006	1.129
6	512	768	1.147	1.061	1.022	1.000	1.127
4	512	512	1.184	1.083	1.031	1.018	1.169
12	768	512	1.174	1.072	1.018	0.998	1.151
6	256	768	1.173	1.092	1.039	0.985	1.147
6	256	512	1.224	1.101	1.045	1.017	1.185
8	256	512	1.223	1.102	1.035	1.006	1.192

Table 17: **Throughput and latency across workloads.** Each configuration uses full-model adaptation and is evaluated at EP32 over 4 nodes. Column headings give input/output lengths, with requests per GPU below. Throughput speedup is the EchoMoE-to-uncompressed throughput ratio. Latency speedups use the inverse ratio, so values above 1 indicate improvement throughout. Decode TPOT uses the 10% of requests with the latest first token (Section G.1).

G.3 Communication and Computation Costs

We profile $(4, 512, 768)$, a configuration highlighted in Figure 1 and Table 3, using 4096 input tokens, 256 output tokens and 128 requests per GPU. Table 18 reports median times across ranks for 1 prefill step and 8 early decode steps, first taking the median over decode steps within each rank. Profiling captures instrumentation overhead and synchronization waits. Operation medians are not additive and should not be used to reconstruct end-to-end speedup.

	Prefill (ms)		Decode (ms)	
Operation	Uncompressed	EchoMoE	Uncompressed	EchoMoE
Dispatch	85.8	21.9	9.9	7.2
Combine	205.1	124.5	12.8	9.7
Reference transfers	0.0	48.9	0.0	5.4
Compression kernels	0.0	20.1	0.0	0.9
Expert matrix products	83.2	59.1	4.0	3.6
Expert output sum	41.2	15.5	1.3	0.3
Attention	12.3	12.3	17.6	17.6
Other matrix products	10.1	11.7	1.6	2.0
Full GPU step	522.9	395.3	56.6	54.4

Table 18: **Operation times for the adapted** $(4, 512, 768)$ **configuration.** Columns report time per prefill or decode step at EP32. Reference transfers cover all-gather and reduce-scatter. Compression kernels cover encoding, decoding and assembly, while dense compression projections are counted under other matrix products. The full step also contains routing, synchronization, normalization and idle time.

Reference transfers offset more of the communication savings during decode. In prefill, dispatch and combine fall from 290.9 to 146.4 ms, with 48.9 ms added for reference transfers. During decode, their 5.8 ms reduction is almost offset by 5.4 ms of reference transfers before compression overhead. This explains the smaller decode improvement despite substantial RDMA savings.

Smaller expert outputs also reduce computation. Folding narrows the expert down projection and output sum. Expert matrix products and output summation together take 49.8 ms less during prefill, while compression kernels add 20.1 ms and other matrix products take slightly longer. Throughput improvements therefore reflect changes in both communication and computation.

Attention limits the relative benefit at long context. For $(4, 512, 1024)$ with 128000 input tokens, an early 4096-token chunk falls from 515.6 to 405.8 ms with EchoMoE. At approximately 106000 tokens of context, a late chunk falls from 1017.8 to 922.1 ms. Dispatch and combine savings remain similar at 138 and 135 ms, but attention takes a larger share of the late step. Faster collectives consequently have less effect on total step time.

G.4 Inference on Larger MoE Architectures

We use $s = 4$, $r_d = d/4$ and $r_c = d/2$ for Qwen3-235B-A22B, DeepSeek-V3 and Kimi-K2.6 (Table 19). These full-sized architectures use random weights, with the same backbone tensors used to construct the compressed model. The comparisons establish execution cost and memory feasibility, without assessing task performance after adaptation.

	Code width		Precision		
Model	Dispatch r_d	Combine r_c	Expert weights	Dispatch	KV cache per GPU
Qwen3-235B-A22B	1024	2048	BF16	BF16	28 GiB
DeepSeek-V3	1792	3584	FP8	FP8	12 GiB
Kimi-K2.6	1792	3584	INT4	BF16	9 GiB

Table 19: **Inference settings for larger MoE architectures.** All configurations use stride 4. Precisions and KV cache capacity are matched between EchoMoE and the uncompressed model. INT4 applies to Kimi's routed expert weights, with BF16 activations and compression projections.

Execution settings. Each comparison uses EP32, TP1 and DP32 over 4 nodes, with 4096 input tokens, 256 output tokens, 128 requests per GPU and a 2048-token step budget. Expert computation uses Triton BF16 for Qwen3-235B, Triton block-scaled FP8 for DeepSeek-V3 and Marlin INT4 for Kimi-K2.6. Both models use the same backend, inactive-request handling and CUDA-graph

settings. Padded routing rows are masked for DeepSeek-V3 and Kimi-K2.6. KV cache capacity holds approximately 36, 42 and 32 full requests per GPU, respectively, so later prompts wait for earlier requests to finish. Reference layers use DeepEP dispatch with asynchronous NCCL reference publication on a separate stream.

Throughput improves across all 3 architectures. Qwen3-235B, DeepSeek-V3 and Kimi-K2.6 achieve $1.15\times$, $1.18\times$ and $1.05\times$ throughput speedups, respectively (Table 20). Qwen3-235B and Kimi-K2.6 improve overall throughput despite slightly slower decode steps, indicating that faster prompt processing accounts for their improvements. Here, decode latency is the median duration of rank-0 engine steps without prefill across the 2 timed repetitions, distinct from request-level TPOT in Table 17.

	Batch time (s) ↓		Speedup (×) ↑		
Model	Uncompressed	EchoMoE	Throughput	TTFT	Decode step
Qwen3-235B-A22B	268.5	232.9	1.152	1.174	0.988
DeepSeek-V3	357.0	301.9	1.182	1.192	1.046
Kimi-K2.6	281.9	268.3	1.051	1.057	0.950

Table 20: **Throughput and latency for larger MoE architectures.** Full-sized models with random weights are benchmarked at EP32 over 4 nodes. Batch times average 2 timed repetitions. Speedups compare EchoMoE with the uncompressed model, with values above 1 indicating improvement. Decode uses engine-step latency rather than request-level TPOT.

H EXTENDED RESULTS

Tables 21 and 22 report task performance for all 40 joint configurations in dev-mode and 19 with full-model adaptation. Separate dispatch and combine ablations are reported in Tables 10 and 12. Domain results average the rounded per-task values, while Overall uses the reported aggregate.

Table 21: **Task performance for all dev-mode joint configurations.** Δ shows the change in overall task performance from the uncompressed model (66.92%), in percentage points. BF16 RDMA reductions use EP= 32 over 4 nodes and count reference transfers.

Configuration				Task Performance (%) ↑				
Stride s	Dispatch r_d	Combine r_c	RDMA Reduction	Knowledge & Reasoning	Math	Code	Instruction Following	Overall (Δ)
Uncompressed ($d = 2048$)			1.00×	77.6	50.7	59.2	55.6	66.9
EchoMoE								
4	1280	1280	1.46×	75.8	50.0	56.9	53.8	65.4 (−1.5)
6	1280	1280	1.50×	75.5	49.3	55.6	53.2	64.8 (−2.1)
8	1280	1280	1.53×	74.8	49.6	56.0	54.6	64.7 (−2.2)
12	1280	1280	1.55×	74.7	48.9	55.5	53.4	64.3 (−2.6)
4	1024	1280	1.57×	75.7	50.5	55.9	54.9	65.4 (−1.5)
8	768	1280	1.83×	71.2	48.3	53.7	54.6	62.0 (−4.9)
4	512	1280	1.84×	73.8	47.9	55.1	52.0	63.4 (−3.5)
8	1024	1024	1.86×	71.5	46.6	54.9	54.6	61.9 (−5.0)
4	768	1024	1.89×	73.1	48.5	54.2	53.0	63.1 (−3.8)
12	1024	1024	1.90×	69.7	46.7	53.2	51.9	60.6 (−6.3)
6	768	1024	2.01×	71.0	45.4	51.1	54.0	60.9 (−6.0)
6	1024	768	2.05×	70.0	45.2	49.8	51.8	59.9 (−7.0)
8	768	1024	2.07×	69.0	45.8	53.4	51.9	60.0 (−6.9)
4	512	1024	2.08×	71.9	46.6	53.3	53.9	62.0 (−5.0)
8	1024	768	2.11×	68.2	44.5	51.0	53.4	59.0 (−7.9)
12	768	1024	2.14×	67.6	43.7	51.3	50.8	58.3 (−8.6)
12	1024	768	2.16×	66.7	44.0	48.9	51.0	57.7 (−9.2)
8	768	768	2.38×	65.4	43.4	49.4	52.8	56.9 (−10.0)
4	512	768	2.39×	68.2	43.7	48.3	52.0	58.4 (−8.5)
12	768	768	2.47×	63.7	41.9	48.0	49.5	55.2 (−11.7)
6	256	1024	2.54×	64.5	38.5	44.7	51.0	54.6 (−12.4)
6	512	768	2.61×	65.6	40.2	45.5	51.2	55.7 (−11.2)
4	512	512	2.81×	62.8	37.8	42.6	49.4	53.1 (−13.8)
12	512	768	2.87×	60.8	36.9	43.7	50.8	52.0 (−14.9)
12	768	512	2.92×	58.9	36.8	42.9	46.5	50.5 (−16.4)
6	256	768	3.02×	61.6	35.7	40.3	49.1	51.6 (−15.3)
6	512	512	3.12×	60.5	36.4	43.3	48.3	51.4 (−15.5)
6	256	512	3.72×	56.4	32.2	37.5	46.2	47.3 (−19.7)
8	256	512	4.03×	54.0	29.4	38.2	46.6	45.3 (−21.6)
8	512	256	4.16×	47.0	26.7	32.2	39.5	39.5 (−27.4)
Plain compression								
–	1280	1280	1.60×	71.6	47.5	53.6	52.7	61.9 (−5.0)
–	1280	1024	1.78×	69.5	45.7	53.4	51.9	60.2 (−6.7)
–	1024	1024	2.00×	65.5	43.2	50.8	49.8	56.9 (−10.0)
–	1280	768	2.00×	65.8	44.2	50.6	50.3	57.3 (−9.6)
–	1024	768	2.29×	62.2	40.8	45.3	51.3	54.0 (−12.9)
–	1024	512	2.67×	58.8	36.8	43.7	47.4	50.6 (−16.3)
–	768	768	2.67×	59.1	36.1	41.7	47.3	50.3 (−16.6)
–	768	512	3.20×	53.9	33.1	38.9	44.1	46.1 (−20.8)
–	512	512	4.00×	47.9	26.9	33.8	42.3	40.4 (−26.5)
–	768	256	4.00×	45.7	25.3	28.8	39.9	38.1 (−28.8)

Table 22: **Task performance for all adapted joint configurations.** Columns follow Table 21. Δ_{adapt} shows the change in overall task performance from dev-mode for the same configuration, in percentage points.

Configuration				Task Performance (%) ↑					
Stride s	Dispatch r_d	Combine r_c	RDMA Reduction	Knowledge & Reasoning	Math	Code	Instruction Following	Overall (Δ)	Δ_{adapt} (pp)
Uncompressed ($d = 2048$)			1.00×	77.6	50.7	59.2	55.6	66.9	–
EchoMoE									
4	1280	1280	1.46×	76.2	49.1	58.1	54.5	65.5 (−1.4)	+0.2
6	1280	1280	1.50×	75.6	49.1	57.4	53.8	65.1 (−1.8)	+0.3
12	1280	1280	1.55×	73.7	49.6	57.4	53.1	64.1 (−2.8)	−0.2
4	512	1280	1.84×	75.3	49.3	56.8	53.7	64.9 (−2.0)	+1.5
6	768	1024	2.01×	74.0	47.8	55.2	52.3	63.5 (−3.4)	+2.6
4	512	1024	2.08×	74.3	47.8	55.2	53.5	63.8 (−3.1)	+1.8
4	512	768	2.39×	73.6	47.2	54.4	53.3	63.1 (−3.8)	+4.7
6	256	1024	2.54×	70.4	45.8	51.8	52.6	60.6 (−6.3)	+6.1
6	512	768	2.61×	71.7	46.4	52.4	51.9	61.5 (−5.4)	+5.8
4	512	512	2.81×	70.0	45.6	53.4	52.5	60.6 (−6.4)	+7.5
12	768	512	2.92×	68.3	45.5	52.1	51.8	59.4 (−7.6)	+8.9
6	256	768	3.02×	68.3	42.5	50.4	50.1	58.3 (−8.6)	+6.7
6	256	512	3.72×	65.8	41.6	51.6	51.2	56.9 (−10.1)	+9.6
8	256	512	4.03×	64.2	40.9	50.7	51.0	55.7 (−11.2)	+10.4
Plain compression									
–	1280	1024	1.78×	72.6	47.5	55.7	53.5	62.8 (−4.1)	+2.5
–	1024	1024	2.00×	70.4	47.4	55.8	53.5	61.6 (−5.3)	+4.6
–	768	768	2.67×	66.2	44.3	52.8	51.8	58.0 (−8.9)	+7.7
–	512	512	4.00×	59.0	37.6	47.1	49.6	51.4 (−15.5)	+11.0
–	768	256	4.00×	58.0	38.1	45.1	48.6	50.7 (−16.2)	+12.6