FP8-FLOW-MOE: A CASTING-FREE FP8 RECIPE WITHOUT DOUBLE QUANTIZATION ERROR

Fengjuan Wang [1] **Zhiyi Su** [1] **Xingzhu Hu** [1] **Cheng Wang** [1] **Mou Sun** [1]

ABSTRACT

Training large Mixture-of-Experts (MoE) models remains computationally prohibitive due to their extreme compute and memory demands. Although low-precision training promises to accelerate computation and reduce memory footprint, existing implementations still rely on BF16-dominated dataflows with frequent quantize-dequantize (Q/DQ) conversions. These redundant casts erode much of FP8's theoretical efficiency. However, naively removing these casts by keeping dataflows entirely in FP8 introduces double quantization error: tensors quantized along different dimensions accumulate inconsistent scaling factors, degrading numerical stability.

We propose FP8-Flow-MoE, an FP8 training recipe featuring a quantization-consistent FP8-centric dataflow with a scaling-aware transpose and fused FP8 operators that streamline computation and eliminate explicit cast operations from 12 to 2. Evaluations on a 671B-parameter MoE model demonstrate up to 21% higher throughput and 16.5 GB lower memory usage per GPU compared to BF16 and naïve FP8 baselines, while maintaining stable convergence. We provide a plug-and-play FP8 recipe compatible with TransformerEngine and Megatron-LM, with the reference implementation available at our GitHub repository.

1 INTRODUCTION

The development of large language models (LLMs) has witnessed remarkable progress in recent years, with model scales growing to hundreds of billions and even trillions of parameters (Brown et al., 2020; Zhang et al., 2022; Meta AI Team, 2023). These massive models have demonstrated unprecedented capabilities across natural language understanding, reasoning, and knowledge-intensive tasks (OpenAI, 2023; Bubeck et al., 2023). However, training such large-scale models poses significant computational challenges, and the enormous cost has become a de facto barrier for many research institutions and even large industrial labs.

To mitigate this, the Mixture-of-Experts (MoE) architecture (Lepikhin et al., 2021; Fedus et al., 2022; Rajbhandari et al., 2022) has been introduced to increase model capacity without proportionally increasing computation, activating only a subset of experts per token. While MoE significantly reduces the overall training cost compared to dense transformers, its large communication footprint and memory demand still make training at scale highly expensive—motivating the exploration of low-precision formats such as FP8 to further improve efficiency.

Low-precision computation has emerged as a promising direction to address this challenge. Compared to the mainstream BF16-based Automatic Mixed Precision (AMP) scheme, adopting the FP8 format can theoretically double computation throughput and halve communication latency (Micikevicius et al., 2022). Since the introduction of NVIDIA's Hopper architecture with native FP8 Tensor Core support (NVIDIA Corporation, 2022), FP8 training has drawn increasing attention. DeepSeek first demonstrated the feasibility of FP8 precision at massive scale in its V3 Mixture-of-Experts (MoE) model (DeepSeek-AI Team, 2024), and subsequently open-sourced *DeepGEMM* (DeepSeek-AI Team, 2025b) and *DeepEP* (DeepSeek-AI Team, 2025a)—two critical kernels for FP8 MoE efficiency—enabling high-throughput grouped GEMM computation and low-latency cross-node all-to-all communication.

Despite these advances, a complete training recipe designed around an FP8 dataflow has been missing. Naively replacing BF16 kernels with their FP8 counterparts introduces **excessive casting overheads**—each GEMM boundary typically involves multiple quantize–dequantize (Q/DQ) conversions (DeepSeek-AI Team, 2024; NVIDIA Corporation, 2023; Zhao et al., 2025). As a result, the realized throughput of current FP8 systems often falls short of their theoretical potential, sometimes even lagging behind optimized BF16 baselines. Moreover, without a properly designed recipe and careful memory management, FP8 kernels may incur

[1]Zhejiang Lab, Hangzhou, China. Correspondence to: Mou Sun <123sssmmm@gmail.com>.

Proceedings of the 9th MLSys Conference, Bellevue, WA, USA, 2026.

extra activation copies, undermining the expected memory savings.

However, simply removing these casts is not a viable solution. A straightforward strategy is to keep operator inputs and outputs directly in FP8—bypassing BF16 storage and boundary casting—to avoid Q/DQ overhead. While this approach reduces conversions, it introduces **double quantization error**: tensors quantized along different dimensions (e.g., row-wise before GEMM and column-wise afterward) accumulate inconsistent scaling factors, leading to directional precision loss and potential instability (Dettmers et al., 2022).

To address these challenges, we propose **FP8-Flow-MoE**, a systematic and practical FP8 training recipe centered on an end-to-end FP8 dataflow. Our key contributions are summarized as follows:

1. **Avoidance of double quantization error in FP8 dataflows.** We eliminate quantization inconsistency by introducing a **scaling-aware transpose** operator that directly converts FP8 tensors between row- and column-wise layouts while preserving scaling consistency. This design removes the need for dequantize–requantize cycles across operators, effectively avoiding double quantization error in FP8 dataflows.

2. **Casting-free FP8 recipe with a native FP8 kernel suite.** We present the FP8-Flow-MoE recipe, where the entire MoE stage operates in FP8 with no explicit casting except at the entry point. Compared to a drop-in kernel replacement approach, the number of cast operations is reduced from 12 to 2, marking a paradigm shift from BF16-centric to FP8-centric computation. We further provide a suite of native FP8 operators that make the recipe feasible and efficient in practice. Several of these kernels have already been merged into upstream repositories (e.g., NVIDIA/TransformerEngine (NVIDIA Corporation, 2023)), while others will be released as part of our open-source implementation.

3. **Comprehensive analysis of FP8 training at system scale.** We conduct an in-depth study comparing FP8 and BF16 across MoE components, analyzing both kernel-level and quantization-related effects. FP8-Flow-MoE achieves up to 21% throughput improvement and 8 GB memory reduction per GPU compared to the BF16 baseline, while naive FP8 kernel replacement yields only a 3% gain with negligible or even negative memory savings.

The remainder of this paper is organized as follows. Section 2 reviews related work on low-precision training techniques for LLMs and the architectural design of Mixture-of-Experts models. Section 3 details how the scaling-aware transpose operator eliminates double-quantization error and enables a casting-free FP8 dataflow, supported by a suite of high-performance FP8 kernels integrated into FP8-Flow-MoE. Section 4 compares FP8-Flow-MoE, BF16, and naïve FP8 on a 671B DeepSeek-V3 model for compute, memory, and communication efficiency, and validates convergence on a 16B DeepSeek-V2-lite model trained with 200B tokens. Finally, Section 5 concludes the paper and discusses potential directions for extending FP8-Flow-MoE to other low-precision formats and representations.

2 Related Work

Low-precision computation has become a key enabler for efficient large-scale model training. Early FP16 and BF16 mixed-precision schemes significantly reduced training cost while maintaining convergence (Micikevicius et al., 2018; Wang et al., 2018). Recently, the FP8 format has gained attention for its ability to double arithmetic throughput on modern accelerators (NVIDIA Corporation, 2022; Micikevicius et al., 2022; DeepSeek-AI Team, 2024). FP8 generally follows IEEE 754 conventions but defines multiple variants, most notably E4M3 and E5M2, which balance precision and dynamic range differently (Wang et al., 2018). E4M3 offers finer precision, while E5M2 provides a wider numeric range. Another variant, UE8M0, encodes powers of two and is typically used for scaling factors rather than activations.

Converting higher-precision tensors (e.g., FP32 or BF16) into FP8 requires quantization. The scaling strategy defines the quantization recipe and directly impacts stability and accuracy. Early works relied on per-tensor scaling (Micikevicius et al., 2018), which often underutilizes FP8's range when tensor values span several magnitudes. Later studies proposed finer-grained scaling, such as per-block or per-channel quantization (Peng et al., 2023), which preserve accuracy for small-magnitude values while maintaining efficiency. These techniques have become standard in recent FP8 training systems.

The adoption of FP8 was accelerated by both hardware and software advances. NVIDIA's Hopper architecture introduced native FP8 Tensor Core support (NVIDIA Corporation, 2022), while TransformerEngine provided open-source operator-level support for FP8 quantization (NVIDIA Corporation, 2023). Early large-scale studies, including simulated FP8 GPT-3 training (Micikevicius et al., 2022) and Inflection-2 (Inflection AI, 2023), demonstrated that FP8 can maintain convergence comparable to BF16. DeepSeek-V3 (DeepSeek-AI Team, 2024) further validated FP8 training at trillion-parameter scale and released two key libraries—*DeepGEMM* (DeepSeek-AI Team, 2025b) for grouped-GEMM and *DeepEP* (DeepSeek-AI Team, 2025a) for cross-node communication—establishing a strong FP8

baseline for Mixture-of-Experts (MoE) models.

However, these implementations still follow BF16 dominated dataflows, requiring frequent Q/DQ conversions across GEMM boundaries (Zhao et al., 2025). Such redundant casting limits realized FP8 performance and increases memory traffic. Conversely, removing these casts introduces double quantization error, as tensors quantized along different dimensions (row-wise vs. column-wise) accumulate inconsistent scaling factors (Dettmers et al., 2022). This trade-off between computational efficiency and numerical stability remains a major challenge in current FP8 systems.

The rapid scaling of large language models (LLMs) (Brown et al., 2020; Meta AI Team, 2023) has pushed training infrastructure to its limits. Parallelization strategies such as tensor, pipeline, and data parallelism (Rajbhandari et al., 2022), together with the Mixture-of-Experts (MoE) architecture (Lepikhin et al., 2021; Fedus et al., 2022), have become essential for trillion-parameter training. MoE reduces active computation by routing each token to a small subset of experts, achieving high model capacity without proportional cost. However, the sparse and irregular computation pattern introduces new bottlenecks: all-to-all expert communication, activation padding, and dynamic routing overheads (Rajbhandari et al., 2022; DeepSeek-AI Team, 2024). Integrating FP8 precision into such workflows is particularly challenging, as each communication or expert boundary typically requires additional quantization and data reformatting. These factors fragment computation into many small kernels and diminish arithmetic intensity, preventing FP8's theoretical benefits from being fully realized in end-to-end MoE training.

Motivation Within each MoE layer, a router assigns tokens to a subset of experts. From a dataflow perspective, one MoE layer consists of routing, (EP) dispatch, token reordering (permute/unpermute), expert computation (two grouped linear projections with an activation in between), and final combination. Figure 1(a) illustrates this BF16 baseline, where communication and data reordering sit on the critical path around the expert GEMMs.

When extending this flow to FP8 computation, existing frameworks primarily adopt conservative integration strategies. For instance, NVIDIA's Transformer Engine implements a blockwise FP8 recipe (Figure 1(b)), where FP8 is applied only to GEMM kernels and quantization conversions confined strictly within the grouped linear modules. As a consequence, each MoE block still incurs frequent quantization conversions around the two grouped linear layers: per layer, Fprop and Dgrad each introduce 1 row-wise quantization, and Wgrad introduces 2 column-wise quantizations, resulting in 4 conversions per grouped linear and 8 in total.This design allows FP8 acceleration in grouped GEMMs, achieving higher arithmetic throughput compared to BF16, but leaves all EP communication and data movement stages in higher precision formats. As a result, while the GEMM kernels benefit from FP8 tensor cores, the overall MoE layer performance improvement is largely limited to GEMM and remains bounded by high-precision communication, data movement, and the conversion overhead around grouped linear modules.

Another representative implementation is the DeepSeek-V3 training system (Figure 1(c)), which demonstrated successful large-scale FP8 MoE training with state-of-the-art performance. Although the complete training framework was not fully open-sourced, the available descriptions and accompanying releases—specifically DeepGEMM and DeepEP—suggest that both expert computation and inter-expert communication are accelerated via customized, high-performance FP8 kernels. However, the overall dataflow still incurs substantial (de)quantization overhead. In particular, DeepEP extends FP8 beyond expert GEMMs to token dispatch, but introduces 2 additional (de)quantization conversions around dispatch and another 2 on the backward combine path, resulting in a total of 12 (de)quantization conversions per MoE layer. Moreover, in the Wgrad stage, FP8 activations are typically quantized row-wise, dequantized, and then re-quantized in a column-wise manner. Since these per-token quantization schemes apply scaling along a particular direction, the repeated (de)quantization can amplify rounding and scaling errors ("double quantization error"), negatively affecting numerical stability and convergence.

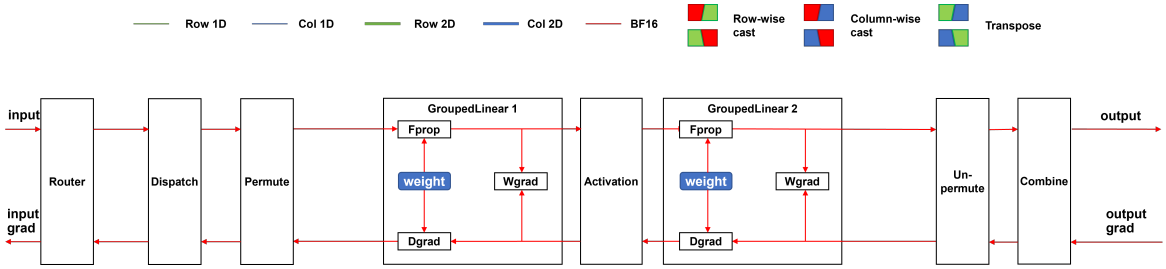
These challenges motivate FP8-Flow-MoE: a quantization-consistent, casting-free FP8 dataflow that eliminates redundant (de)quantization operations and maintains numerical stability throughout the MoE pipeline.

3 METHOD

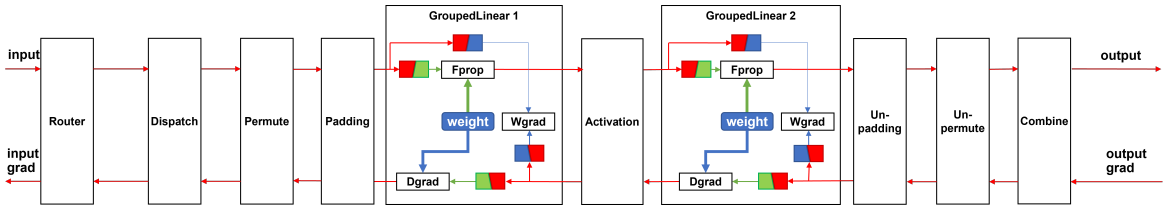
There have been extensive studies on applying FP8 quantization in transformer-based model training, particularly, dense models such as BERT, GPT, etc., achieving significant performance gain with negligible accuracy loss. This contribution focuses on another archetype of models, MoE models, which have emerged as one of the most performant LLM architectures due to their high capability and reduced resource requirements for inference, i.e. only small parts of all experts are activated.

3.1 Scaling-aware transpose

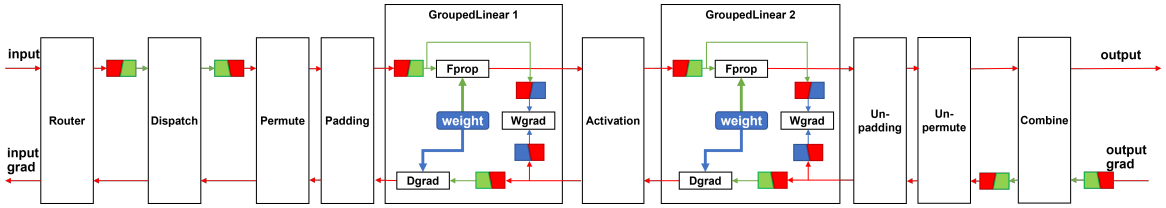
One of the key challenges in fully adopting FP8 precision in the MoE module is the two different quantization layouts required by the grouped linear operations. Specifically, the activations (and their gradients) are consumed in a row-wise quantized format (referred to as *Fprop* and *Dgrad* in



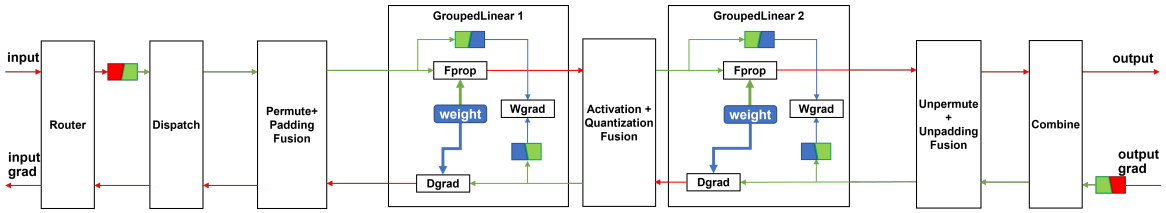
(a) BF16 baseline.



(b) TransformerEngine blockwise recipe.



(c) according to DeepSeek V3 technical report.



(d) FP8-Flow-MoE.

Figure 1: The computational graph of an MoE module

Figure 1), while the weight-gradient computation requires column-wise quantized inputs (*Wgrad*). However, the preceding all-to-all communication (*dispatch*) typically transmits only one format, usually the row-wise quantized data, to leverage the reduced-communication advance brought by using a low-precision format.

As a consequence, when the grouped linear layer subsequently needs column-wise quantized data, a naive implementation must first dequantize → transpose → requantize, introducing two rounds of (de)quantization. This process leads to what we call the double quantization error, whose root cause is that 1D per-token quantization uses scaling factors computed over contiguous segments (e.g., 128 elements per scale), and the scales differ across the two layouts. Since FP8 quantization maps scaled values onto a discrete representable grid determined by the exponent and mantissa bits, performing two independent quantizations with different scaling factors effectively remaps values twice to non-overlapping discrete sets. Formally, the double quantization error is defined by

$$E = Q_{col}(D(Q_{row}(X))) - X \tag{1}$$

where $Q_{row}(\cdot)$, $Q_{col}(\cdot)$ are the row-wise and column-wise quantization operator and $D(\cdot)$ is the de-quantization operator. Both row-wise and column-wise quantization are performed in a per-tile fashion, with 128 continuous elements in each tile. In other words, the scaling factor is calculated as

$$s = \frac{\max_{0 \le i < 128} \|x_i\|}{448} \tag{2}$$

where 448 is the maximum representable number of FP8_E4M3 format. After scaling, a quantization operator essentially maps x_i into the nearest discrete value

$$Q_{row}(x_i) = \text{round}(\frac{x_i}{s}) \tag{3}$$

A de-quantization operator scales the FP8 data back to its original range by multiplying the scaling factor, but the rounding error remains,

$$D(Q_{row}(x_i)) = \text{round}(\frac{x_i}{s}) \cdot s \tag{4}$$

Following this representation, it is trivial that performing a second row-wise quantization do not introduce any extra error since the same 1x128 elements are again tiled for quantization, which gives same maximum value and eventually leaving s unchanged. A number is strictly mapped to the identical discrete value, or formally,

$$\begin{aligned} Q_{row}(D(Q_{row}(x_i))) &= \text{round}(\frac{\text{round}(\frac{x_i}{s}) \cdot s}{s}) && (5)\\ &= \text{round}(\text{round}(\frac{x_i}{s})) && (6)\\ &= \text{round}(\frac{x_i}{s}) && (7)\\ &= Q_{row}(x_i) && (8) \end{aligned}$$

where we assumed a deterministic rounding algorithm, e.g. round to nearest (RtN), is used and rounding the same number twice does not change its value.

However, in case that the second quantization is performed column-wise, its scaling factor will be calculated from the 128-element tile after transpose, marked as s', which is generally not the same as s. We have

$$Q_{col}(D(Q_{row}(x_i))) = \text{round}(\frac{\text{round}(x_i/s) \cdot s}{s'}) \tag{9}$$

where the two rounding operators cannot be combined because they are applying to two different values. However, this error amplification can be mitigated if the scaling factor is constrained to powers of two, in which case, rescaling between two quantization domains (row-wise and column-wise) involves only adjusting exponent bits, provided that no overflow or underflow occurs. Formally, let us assume $s = 2^T, s' = 2^{T'}$, where $T \in \mathbb{N}, T' \in \mathbb{N}$. At row-wise quantization, the value of an element is discretized to

$$Q_{row}(x) = (-1)^{SN} \cdot 2^{E-7} \cdot (1 + M/8) \cdot 2^T \tag{10}$$

where SN, E and M are the sign bit, exponent bits and matissa bits of the its FP8 coding respectively, 7 and 8 are constants derived from FP8_E4M3 definition. While applying column-wise quantization, the same value is discretized to a different representation,

$$Q_{col}(x) = (-1)^{SN'} \cdot 2^{E'-7} \cdot (1 + M'/8) \cdot 2^{T'} \tag{11}$$

Since both T and T' are natural numbers, there exist another natural number D, where $D = T' - T$. Notice that if we set

$$SN' = SN, \quad E' = E - D, \quad M' = M.$$

, we have

$$\begin{aligned} Q_{col}(x) &= (-1)^{SN} \cdot 2^{E-D-7} \cdot (1 + M/8) \cdot 2^{D+T} && (12)\\ &= (-1)^{SN} \cdot 2^{E-D-7+D+T} \cdot (1 + M/8) && (13)\\ &= (-1)^{SN} \cdot 2^{E-7+T} \cdot (1 + M/8) && (14)\\ &= (-1)^{SN} \cdot 2^{E-7} \cdot (1 + M/8) \cdot 2^T && (15)\\ &= Q_{row}(x) && (16) \end{aligned}$$

Building on this insight, we further align all scaling factors within a transposed block to the largest one (to avoid overflow) and directly convert between layouts by exponent manipulation alone. In other words, we can transform row-wise quantized FP8 tensors into column-wise quantized ones without any dequantization or re-quantization, by simply modifying the exponent bits of the FP8 encodings. This principle underlies our Scaling-aware FP8 Transpose operator, as outlined in Algorithm 1. In our implementation, we follow the Transformer Engine's blockwise FP8 recipe

Algorithm 1 Scaling-aware FP8 Transpose (per block B×B)

Input: row-wise data $X_{i,j}^{row}$, scaling factor S_i^{row} for $i \in [i_0, i_0{+}B{-}1]$, $j \in [j_0, j_0{+}B{-}1]$
Output: column-wise data $X_{j,i}^{col}$ and scaling factor S_j^{col}

$S_{\max} = \max_{i \in [i_0, i_0+B-1]} S_i^{row}$
$\forall j \in [j_0, j_0{+}B{-}1],\ S_j^{col} = S_{\max}$
for each element (i, j) in the block **do**
  $k = \log_2(S_{\max}/S_i^{row})$ $\{k \in \mathbb{N}\}$
  if $k = 0$ **then**
    $X_{j,i}^{col} = X_{i,j}^{row}$
  else
    $SN = X_{i,j}^{row}$ & 0x80 {sign bit}
    $E = X_{i,j}^{row}$ & 0x78 {exponent bits}
    $M = X_{i,j}^{row}$ & 0x07 {mantissa bits}
    $E_{\text{new}} = E - (k \ll 3)$
    $E_{\text{new}} = 0$ **if** $E_{\text{new}} <$ 0x08 {flush-to-zero}
    $X_{j,i}^{col} = SN \mid E_{\text{new}} \mid M$
  end if
end for

and set the quantization block size to B=128; however, the algorithm applies to an arbitrary block size B.

During the scaling-aware transpose, we select the maximum scaling factor within each block. Since scaling factors are defined as quantization divisors, this guarantees the exponent adjustment only shifts FP8 exponents downward, eliminating overflow risk. Underflow is possible and is a general property of FP8 quantization. In our case, its impact is limited for two reasons. First, the transpose affects only weight gradient computation and does not propagate to dgrad. Second, even with E4M3, the exponent range spans multiple orders of magnitude, so the resulting underflow is typically not large enough to dominate gradient updates.

3.2 Casting-free FP8 dataflows

To fully exploit the computational advantages of FP8 while maintaining numerical stability, we design an FP8-centric dataflow, referred to as FP8-Flow-MoE (Figure 1(d)), that minimizes format conversions and preserves FP8 representations along the MoE expert path whenever possible. Concretely, FP8-Flow-MoE introduces the following design choices:

FP8 communication. We perform quantization before dispatch and feed row-wise FP8 tensors into grouped linear. This keeps the dispatch → permute/padding → expert computation path entirely in FP8. In the TransformerEngine implementation, we store the scale factors in a compact layout to avoid layout-induced transposes overhead. Detailed benchmarks are provided in Section 4.4.2.

Fusion of operators to reduce kernel fragmentation. We apply quantization-aware and data-movement-aware fusion along the MoE path to reduce kernel launches and intermediate memory traffic. This includes fusing token reordering (permute/unpermute and padding/unpadding) into symmetric kernels across forward/backward passes, and fusing activation with FP8 quantization to produce FP8 outputs on the-fly without introducing standalone quantization operators on the critical path. Details are provided in Section 3.3.1 and Section 3.3.2.

Scaling-aware FP8 transpose. To meet the column-wise FP8 requirement of Wgrad FP8 GEMM while avoiding the costly dequantize → transpose → requantize sequence, we use our previously introduced Algorithm 1 to directly convert row-wise FP8 activations and output gradients into column-wise FP8.

Fine-grained recomputation. Because dispatch → permute/padding → expert computation stays in FP8, we enable fine-grained recomputation before the first grouped linear layer. This can reduce peak activation memory by up to ∼50% and avoids recomputing the router, compared to recomputation at the full moe module level.

While maintaining numerical stability, we retain BF16 at two specific boundaries: (i) between the first grouped linear and the activation function, and (ii) between the second grouped linear and the combine operation (i.e., reduction across experts). These two exceptions correspond to fundamental numerical challenges for FP8 arithmetic. The first involves nonlinear transformations (e.g., GELU or ReLU), which can amplify quantization errors due to their sensitivity to input perturbations. The second involves reduction operations (e.g., summations), where accumulated values significantly increase the required dynamic range and are therefore more susceptible to overflow under limited exponent precision. The same considerations apply symmetrically to the corresponding backward computations. Since these two computations are adjacent in the graph, we retain BF16 precision locally to ensure stability, while keeping all other tensors strictly in FP8 format.

3.3 High-Performance FP8 Kernels

To make the proposed FP8-centric computation flow practical and efficient, we further develop a comprehensive set of supporting operators specifically tailored to the data dependencies and numerical patterns within the MoE layer. These operators address key challenges that arise when applying FP8 to large-scale MoE training, including cumulative rounding and scaling errors, redundant data conversions, and schedule inefficiencies caused by fractured kernels. Together, they enable a seamless and high-throughput execution of the FP8-Flow-MoE, ensuring that the benefits of FP8 tensor cores extend beyond isolated kernels to the full

end-to-end training pipeline.

3.3.1 Fused Permute and Padding

In FP8 data pipeline, a *padding* step is introduced to satisfy the shape constraint of FP8 GEMM kernels—each expert's input must contain a multiple of 16 entries to fully utilize tensor cores. Before this, a *permute* operation reorganizes dispatched tokens so that samples belonging to the same expert are contiguous in memory. Executing these two lightweight data movement kernels separately incurs redundant HBM accesses and kernel launches. Since both operations are element-wise and independent, they can be naturally fused into a single kernel.

Based on these observations, we design a Fused Permute+Padding operator that directly performs expert-wise reordering and shape alignment in one pass. The fusion is implemented through a thread-block mapping scheme that dynamically computes the target offset in the padded layout while streaming reordered input elements from global memory. Similarly, in the backward propagation phase, we apply the same fusion strategy to combine unpermute and unpadding operations into a single kernel. This symmetric fusion ensures consistent data handling between forward and backward flows. A detailed benchmark and analysis are provided in Section 4.4.1.

3.3.2 Fused SwiGLU and Quantization

The motivation for designing an FP8-centric training flow originates from a key performance observation: naively inserting FP8 kernels (e.g., GEMM and communication) into a BF16 computation graph introduces frequent quantization and dequantization operations, which can substantially offset the benefits of FP8 acceleration.

To illustrate this issue, we analyze FP8 communication using the DeepEP library released by DeepSeek. In its default usage, quantization and dequantization are performed immediately before and after the communication, introducing extra data conversion and synchronization on the critical path. A detailed latency breakdown and quantitative analysis are provided in Section 4.4.2.

It is worth noting that although FP8 communication theoretically halves the data transfer size, in practice, the need to transmit both the FP8 tensor and its corresponding scaling factors increases the number of data buffers and synchronizations, thereby constraining the achievable communication speedup. Nevertheless, even a single pair of quantization and dequantization operators can noticeably reduce the performance gain of FP8 communication. Considering that a typical MoE forward or backward pass involves multiple such pairs (see Figure 1(c)), the cumulative overhead can almost completely neutralize the benefits of isolated FP8 kernels.

To mitigate this issue, we apply quantization-aware fusion to surrounding compute kernels whenever possible. In particular, we fuse the SwiGLU activation and quantization steps into a single kernel, producing FP8-quantized outputs for subsequent computation without introducing standalone quantization operators on the critical path. This design preserves the numerical efficiency of mixed precision while removing the quantization bottleneck that previously limited end-to-end performance. Operator-level micro-benchmark results are reported in Section 4.4.3.

4 EXPERIMENTS

We evaluate FP8-Flow-MoE from two perspectives: numerical convergence and training efficiency. And training efficiency includes end-to-end efficiency results (Section 4.2), component-level ablations (Section 4.3), and operator-level micro-benchmarks (Section 4.4).

All end-to-end experiments (convergence, efficiency, and ablations) are conducted on a 32-node NVIDIA Hopper cluster, where each node is equipped with eight GPUs (80 GB memory per GPU). Expert parallelism (EP) is scaled across nodes to assess training efficiency and scalability under realistic large-model workloads.

4.1 Convergence Validation

To verify that FP8-Flow-MoE maintains numerical stability and convergence behavior, we train a DeepSeek-V2-Lite model (16 B parameters) from scratch under two precision settings: BF16 and FP8-Flow-MoE. Both models are trained for 200 B tokens, using identical optimization hyperparameters, learning rate schedule, data ordering, and parallelism configurations. All runs use a realistic web-scale pretraining mixture consistent with the DeepSeek-V3 accuracy validation setting (DeepSeek-AI Team, 2024), including FineWeb, Nemotron-CC-v2, and DCLM-baseline.

As shown in Figure 2, the two loss curves are nearly indistinguishable across the entire training trajectory, from the early warm-up phase through the stabilized loss regime. The FP8-Flow-MoE model shows no sign of divergence, gradient underflow, or numerical drift, the relative loss error remains consistently below 0.19%, which is within the typical variance introduced by stochastic training dynamics. These results demonstrate that our FP8-centric dataflow does not compromise convergence fidelity even under extended-scale workloads.

This observation confirms that the scaling-aware transpose and fused operators in FP8-Flow-MoE preserve the numerical dynamics of BF16 training.

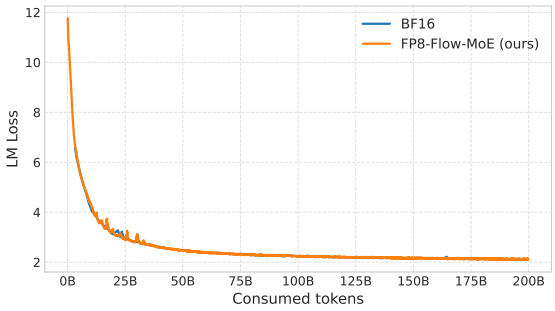


Figure 2: Training loss on DeepSeek-V2-Lite

4.2 Efficiency Evaluation

We evaluate FP8-Flow-MoE on multiple large-scale MoE models, with DeepSeek-V3 (671B) as the primary benchmark and additional evaluations on Qwen3-235B and DeepSeek-V2 (236B) to assess generality across model scales and architectures.

We next benchmark end-to-end training efficiency across these models under multiple precision and dataflow configurations. Specifically, we compare the following four representative configurations: **(1) BF16**: baseline configuration using standard FP32-BF16 mixed precision. **(2) Blockwise**: TE-style blockwise recipe with FP8 grouped GEMM kernels. **(3) Tensorwise**: TE-style tensorwise recipe with FP8 grouped GEMM kernels. **(4) FP8-Flow-MoE (ours)**: FP8-centric dataflow with fused operators and minimal quantization overhead.

Training Hyperparameters. All end-to-end efficiency experiments use AdamW (β_1=0.9, β_2=0.95) with sequence length 4096, tensor parallelism TP=1, and context parallelism CP=1. All runs are continued from the same publicly released checkpoint to ensure comparable initialization across methods. For DeepSeek-V3 (671B) and Qwen3-235B, we use a global batch size of 15,360 tokens with learning rate 7.3×10^{-6} (min LR = 0). For DeepSeek-V2 (236B), we use a global batch size of 8,192 tokens with learning rate 3×10^{-6} (min LR = 3×10^{-7}).

DeepSeek-V3 (671B) serves as the primary benchmark for end-to-end efficiency experiments, enabling us to evaluate the scalability and robustness of FP8-Flow-MoE under realistic large-scale training conditions. Models are trained using standard expert and pipeline parallelism configurations (EP/PP = 8/32, 16/16, 32/8) and evaluated under two activation checkpointing (AC) strategies: *AC = full*, which applies full checkpointing to all modules; and *AC = sel (+MoE expert)*, which selectively checkpoints the MoE layer while excluding experts to evaluate FP8 activation compression. The latter represents the most memory-efficient configuration compatible with the 1F1B-overlap schedule in Megatron, as checkpointing the entire MoE layer would otherwise cause OOM across all recipes. To isolate the effect of computation efficiency from communication overlap, all end-to-end experiments are performed without computation–communication overlap.

Throughput Analysis. Table 1 and 2 summarizes the measured throughput (TGS, tokens/GPU/s) and peak memory (Mem, GB) under three EP settings. Across all activation checkpointing modes, FP8-Flow-MoE consistently outperforms the BF16 baseline, achieving +6 % (EP8), +8 % (EP16), and +16 % (EP32) throughput improvements. Compared with the Blockwise recipe, FP8-Flow-MoE further improves throughput by 3 % (EP8), 8 % (EP16), and up to 21 % (EP32). It also surpasses the Tensorwise recipe, suggesting that the benefits of FP8-Flow-MoE extend beyond BF16 and Blockwise FP8 to a more quantization-efficient baseline, with the advantage reaching up to 18% at EP32. The performance gap widens at higher EP levels, where the cumulative impact of reduced quantization overhead and kernel fusion becomes more significant.

Memory Efficiency. Table 2 shows that, under *AC = sel (+MoE expert)* at EP8, FP8-Flow-MoE reduces peak memory by approximately 8 GB vs. BF16 and 16.5 GB vs. TE FP8 baselines(Blockwise and Tensorwise), directly benefiting from the FP8 checkpoint compression mechanism, which stores intermediate activations in FP8 instead of BF16. At the largest scale of EP = 32, both the BF16 and FP8 baselines (Blockwise and Tensorwise) encounter out-of-memory (OOM) errors, while FP8-Flow-MoE remains stable, clearly demonstrating its superior scalability and memory efficiency.

Method	EP8		EP16		EP32	
	TGS	Mem	TGS	Mem	TGS	Mem
BF16	1,109	39	939	**36**	671	**43**
Blockwise	1,146	37	938	41	644	51
Tensorwise	1,130	38	989	42	660	53
FP8-Flow-MoE	**1,176**	37	**1,012**	39	**779**	49

Table 1: Throughput analysis on DeepSeek-V3 under **AC=full**

Method	EP8		EP16		EP32	
	TGS	Mem	TGS	Mem	TGS	Mem
BF16	1,178	64	1,055	71	OOM	OOM
Blockwise	1,178	73	1,031	77	OOM	OOM
Tensorwise	**1,231**	73	OOM	OOM	OOM	OOM
FP8-Flow-MoE	1,193	**56**	**1,111**	**66**	**912**	**75**

Table 2: Throughput analysis on DeepSeek-V3 under **AC=sel (+MoE expert)**

Beyond the primary benchmark on DeepSeek-V3 (671B),

we further evaluate FP8-Flow-MoE on two additional large-scale MoE models, Qwen3-235B and DeepSeek-V2 (236B), to assess robustness across model scales and architectures. Detailed results are reported in Table 3, Table 4, Table 5, and Table 6.

In particular, FP8-Flow-MoE delivers up to +16% throughput improvement on Qwen3-235B and up to +10% on DeepSeek-V2 under *AC=full*. Moreover, under *AC=sel (+MoE expert)* with EP32 on Qwen3-235B, FP8-Flow-MoE is the only configuration that remains runnable, whereas BF16 and TE FP8 baselines encounter OOM. On DeepSeek-V2 with EP32, FP8-Flow-MoE further reduces peak memory by up to 15 GB per GPU compared to TE FP8 baselines and by up to 11 GB compared to BF16.

These results are consistent with our findings on DeepSeek-V3, indicating that the benefits of FP8-Flow-MoE generalize across different model scales and configurations.

Method	EP8		EP16		EP32	
	TGS	Mem	TGS	Mem	TGS	Mem
BF16	1,150	**40**	884	**36**	**559**	38
Blockwise	1,215	48	830	42	520	43
Tensorwise	1,268	46	883	40	543	37
FP8-Flow-MoE	**1,300**	46	**961**	39	558	**34**

Table 3: Throughput on **Qwen3-235B** under **AC=full**.

Method	EP8		EP16		EP32	
	TGS	Mem	TGS	Mem	TGS	Mem
BF16	1,264	74	996	79	OOM	OOM
Blockwise	OOM	OOM	OOM	OOM	OOM	OOM
Tensorwise	OOM	OOM	OOM	OOM	OOM	OOM
FP8-Flow-MoE	**1,413**	**64**	**1,091**	**60**	**651**	**68**

Table 4: Throughput on **Qwen3-235B** under **AC=sel**.

Method	EP8		EP16		EP32	
	TGS	Mem	TGS	Mem	TGS	Mem
BF16	1,286	**27**	1,319	**36**	1,147	**35**
Blockwise	**1,450**	33	1,330	43	1,113	40
Tensorwise	1,365	34	1,277	44	1,094	42
FP8-Flow-MoE	1,393	32	**1,344**	41	**1,209**	38

Table 5: Throughput on **DeepSeek-V2** under **AC=full**.

4.3 End-to-End Component Ablations

FP8-Flow-MoE's end-to-end gains come from several co-designed optimizations, including FP8 communication, scaling-aware FP8 transpose, and operator fusion. Because these components are tightly coupled in the execution graph, ablating one component often requires adjusting the training pipeline and scheduling, which may slightly deviate from the fully optimized behavior. Nevertheless, end-to-end ablations remain useful to quantify the contribution of key optimizations on the critical path. We conduct component ablations on DeepSeek-V3 (671B) under the same setup as our main efficiency evaluation with full activation checkpointing. We focus on two components: (i) fused permute+padding and (ii) scaling-aware FP8 transpose.

Method	EP8		EP16		EP32	
	TGS	Mem	TGS	Mem	TGS	Mem
BF16	1,474	55	1,499	63	1,315	66
Blockwise	**1,600**	60	1,547	70	1,293	70
Tensorwise	1,548	60	1,599	70	1,391	70
FP8-Flow-MoE	1,572	**52**	**1,601**	**57**	**1,408**	**55**

Table 6: Throughput on **DeepSeek-V2** under **AC=sel**.

For the fused permute+padding operator, we compare against a naive implementation that executes permute and padding as separate operators. As shown in Table 7, fused permute+padding consistently improves end-to-end throughput up to +4%, while also reducing peak GPU memory by approximately 1 GB. These results indicate that kernel fusion on the MoE routing path provides a stable and non-trivial contribution to the overall throughput gains.

For the scaling-aware FP8 transpose, we compare against the dequantize–transpose–requantize path(DoubleQuant). Replacing DoubleQuant with scaling-aware transpose yields substantially larger end-to-end improvements, improving throughput by up to +18%. In addition to throughput gains, this replacement reduces peak memory by approximately 1 GB depending on the EP configuration.

Method	EP8		EP16		EP32	
	TGS	Mem	TGS	Mem	TGS	Mem
FP8-Flow-MoE	**1,176**	**37**	**1,012**	**40**	**779**	**49**
Permute+Padding	1,130	**37**	967	41	768	50
DoubleQuant	1,028	38	859	41	655	51

Table 7: Component ablations on DeepSeek-V3

4.4 Operator-level micro-benchmarks

We conduct a series of unit-level benchmarks that isolate each kernel's performance. The test inputs are generated through simulation, and their tensor shapes are chosen to reflect the real-world configurations of representative MoE models—DeepSeek-V2-Lite, DeepSeek-V2, and DeepSeek-V3 under different expert parallelism (EP) settings. Specifically, these micro-benchmarks allow us to directly measure raw kernel performance, demonstrating how each operator

contributes to the overall efficiency of the FP8-Flow-MoE system.

Operator fusion benchmarks are run on a single GPU to isolate kernel-level speedups. For communication benchmarks, we follow the same EP-to-node mapping as in end-to-end training: EP8 is evaluated on 1 node, EP16 on 2 nodes, and EP32 on 4 nodes.

4.4.1 *Fused Permute and Padding*

We compare the fused and naive implementations under both BF16 and FP8 precision settings. Across different tensor sizes corresponding to typical MoE workloads, the fused version demonstrates a consistent acceleration. For the forward permute+padding case, we observe up to 1.7× speedup (Figure 3), while for the backward unpermute+unpadding kernel, the improvement reaches as high as 6.6× on large-scale configurations (Figure 4).

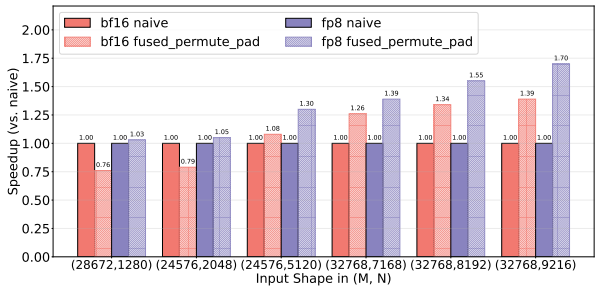


Figure 3: Speedup of fused permute+pad over the naive (separate) implementation.

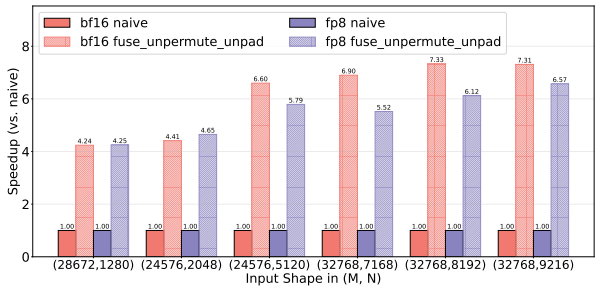


Figure 4: Speedup of fused unpermute+unpad over the naive (separate) implementation.

4.4.2 *FP8 Communication*

Table 8 reports a latency breakdown of FP8 communication in DeepEP, where quantization and dequantization are executed immediately before and after the communication kernel. The results reveal two key findings. First, for smaller-scale workloads, quantization and dequantization each consume a similar amount of time as the communication kernel, nearly eliminating any performance gain. Second, while the communication cost grows roughly linearly with workloads, the (de)quantization cost remains constant; consequently, such data conversion reduces the end-to-end FP8 speedup from 1.6× to 1.4×. It is worth noting that although FP8 communication theoretically halves the data transfer size, in practice, the need to transmit both the FP8 tensor and its corresponding scaling factors doubles the number of data buffers and synchronizations, limiting achievable speedup to around 1.6×. Nevertheless, a single pair of quantization and dequantization operations reduces the performance gain of FP8 communication by roughly one third (from 0.6× to 0.4× improvement).

(M,N,EP)	BF16	FP8 Time (ms)			Speedup	
	(ms)	Q/D	Comm	All	Comm	All
(24576,2048,8)	0.537	0.127/0.084	0.325	0.535	1.65×	1.00×
(24576,5120,8)	0.785	0.087/0.089	0.526	0.703	1.49×	1.12×
(32768,7168,8)	1.276	0.086/0.089	0.905	1.080	1.41×	1.18×
(24576, 2048,16)	1.224	0.091/0.083	1.176	1.350	1.04×	0.91×
(24576,5120,16)	2.213	0.082/0.082	1.400	1.564	1.58×	1.42×
(32768,7168,16)	2.934	0.084/0.092	1.847	2.023	1.59×	1.45×
(24576,2048,32)	3.005	0.094/0.083	2.740	2.918	1.10×	1.03×
(24576,5120,32)	5.003	0.082/0.081	2.868	3.031	1.74×	1.65×
(32768,7168,32)	7.327	0.082/0.082	4.319	4.483	1.70×	1.63×

Table 8: Communication Performance with Speedup

4.4.3 *Fused SwiGLU and Quantization*

Figure 5 presents unit-level benchmarks for the fused SwiGLU+Quant kernel. Compared with the naive implementation (standalone SwiGLU followed by a separate quantization kernel), fusion consistently improves performance in both passes. In the forward pass, the fused kernel achieves $1.69\times$–$2.18\times$ speedup across the tested shapes. The gain is even more pronounced in the backward pass, reaching $2.56\times$–$4.01\times$, indicating that removing the extra kernel launch and intermediate activation round-trips is particularly beneficial when backward-side memory traffic is heavier. The accompanying latency breakdown further confirms that fusion substantially reduces the end-to-end operator time while preserving the original SwiGLU computation, and meanwhile directly produces FP8-quantized outputs for subsequent computation.

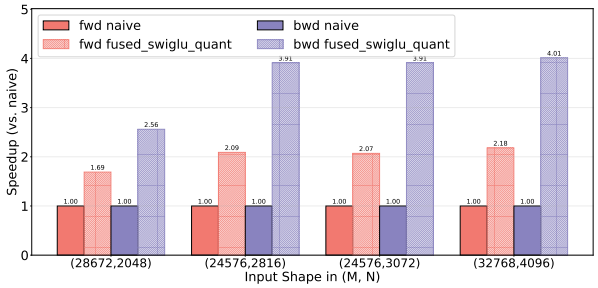


Figure 5: Speedup of fused SwiGLU+quantization over the naive (separate) implementation.

4.4.4 *Scaling-aware FP8 Transpose*

Figure 6 presents unit-level benchmarks for scaling-aware FP8 transpose operator, Compared with the naive implementation (standalone dequantize → transpose → quantize), it is clear that scaling-aware FP8 transpose operator delivers 2 to 3 times speedup across all tensor shapes, confirming its effectiveness in computational efficiency.

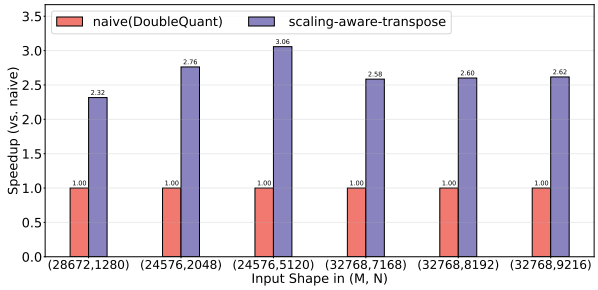


Figure 6: Speedup of scaling-aware transpose over the naive (dequantize + transpose + quantize) implementation.

4.5 Discussion

Persistent FP8 dataflow matters Simply replacing GEMM or communication kernels with FP8 counterparts yields limited benefit, since quantization/dequantization costs dominate at small batch sizes or high communication frequency. The proposed scaling-aware transpose and fused operator design eliminates redundant precision transitions, directly improving kernel launch efficiency and memory bandwidth utilization.

Scaling amplifies FP8-Flow-MoE's gains As EP increases, communication and quantization overheads compound in baseline methods, whereas FP8-Flow-MoE's low-precision persistence minimizes these costs, resulting in up to $1.2\times$ end-to-end acceleration at large expert parallelism.

Practical training stability Despite adopting lower precision for a larger portion of the training pipeline than baseline implementations, FP8-Flow-MoE preserves identical convergence behavior and robustness across hundreds of billions of tokens. This stability stems from our insight of potential double quantization errors at transpose boundaries. By introducing a scaling-aware transpose, FP8-Flow-MoE effectively eliminates this redundant quantization while streamlining data movement, yielding both improved numerical fidelity and higher efficiency.

Portability and Generality FP8-Flow-MoE is evaluated on NVIDIA Hopper, but its key ingredients are not tied to any specific GPU generation. The scaling-aware transpose assumes blockwise quantization with power-of-two scaling, so rescaling between layouts reduces to an exact exponent shift (Algorithm 1) without extra rounding from dequantization or requantization. Under this assumption, the transpose is FP8-format agnostic and applies to both E4M3 and E5M2; otherwise, one must fall back to explicit dequantize→transpose→requantize.

Beyond the transpose, fused operators are driven by MoE dataflow and tensor-shape constraints (token reordering, padding/alignment, and conversion boundaries) rather than Hopper-specific features. They are implemented using general kernel programming frameworks and primarily require a comparable programmable kernel stack and support for the target low-precision datatypes and scaling representation.

These assumptions also align with the broader trend of microscaling-style low-precision formats, where block scales are represented as exponent-only encodings (e.g., UE8M0/E8M0), and are standardized across FP8/FP6/FP4 variants (e.g., MXFP8/6/4). As such, FP8-Flow-MoE naturally extends to future hardware that adopts blockwise quantization with exponent-only scaling, while preserving the design principles of eliminating redundant (de)quantization and reducing kernel fragmentation along the MoE path.

5 Conclusions

This study identifies two additional factors that play critical roles in achieving high end-to-end efficiency for FP8 precision in large-scale Mixture-of-Experts (MoE) training.

First, maintaining data in low precision throughout the computation flow minimizes redundant conversions and alleviates memory bandwidth pressure. A representative example is our FP8 transpose operator, which directly transforms row-wise to column-wise quantized tensors without reverting to higher precision—achieving high data throughput while avoiding double quantization error.

Second, FP8 precision inevitably fragments the computation graph by introducing small operators such as quantization and padding, leading to excessive kernel launches and memory traffic. By fusing lightweight operations and developing custom high-performance implementations, FP8-Flow-MoE effectively eliminates this structural inefficiency.

In summary, FP8-Flow-MoE presents an FP8-centric training approach for MoE models that preserves convergence while improving end-to-end efficiency. Our results highlight that the effectiveness of FP8 depends on dataflow design rather than a drop-in replacement, and demonstrates the practical viability of low-precision MoE training.

Acknowledgments

This work is supported by National Key Research and Development Program of China (No. 2023YFE0108600)

REFERENCES

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., Nori, H., Palangi, H., Tulio Ribeiro, M., and Zhang, Y. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *arXiv e-prints*, art. arXiv:2303.12712, March 2023. doi: 10.48550/arXiv.2303.12712.

DeepSeek-AI Team. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

DeepSeek-AI Team. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025a.

DeepSeek-AI Team. Deepgemm: clean and efficient fp8 gemm kernels with fine-grained scaling. https://github.com/deepseek-ai/DeepGEMM, 2025b.

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems*, volume 35, pp. 30318–30332, 2022.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

Inflection AI. Inflection-2: The next step up. https://inflection.ai/inflection-2, 2023.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. {GS}hard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=qrwe7XHTmYb.

Meta AI Team. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv e-prints*, art. arXiv:2307.09288, July 2023. doi: 10.48550/arXiv.2307.09288.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training. *International Conference on Learning Representations (ICLR)*, 2018.

Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., Mellempudi, N., Oberman, S., Shoeybi, M., Siu, M., and Wu, H. FP8 Formats for Deep Learning. *arXiv e-prints*, art. arXiv:2209.05433, September 2022. doi: 10.48550/arXiv.2209.05433.

NVIDIA Corporation. Nvidia hopper architecture whitepaper. *Technical Whitepaper*, 2022.

NVIDIA Corporation. Nvidia transformer engine documentation. https://github.com/NVIDIA/TransformerEngine, 2023.

OpenAI. Gpt-4 technical report. *arXiv e-prints*, art. arXiv:2303.08774, March 2023. doi: 10.48550/arXiv.2303.08774.

Peng, H., Wu, K., Wei, Y., Zhao, G., Yang, Y., Liu, Z., Xiong, Y., Yang, Z., Ni, B., Hu, J., et al. Fp8-lm: Training fp8 large language models. *arXiv preprint arXiv:2310.18313*, 2023.

Rajbhandari, S., Li, C., Yao, Z., Zhang, M., Aminabadi, R. Y., Awan, A. A., Rasley, J., and He, Y. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 18332–18346. PMLR, 17–23 Jul 2022.

Wang, N., Choi, J., Brand, D., Chen, C.-Y., and Gopalakrishnan, K. Training deep neural networks with 8-bit floating point numbers. *Advances in neural information processing systems*, 31, 2018.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Singh Koura, P., Sridhar, A., Wang, T., and Zettlemoyer, L. OPT: Open Pre-trained Transformer Language Models. *arXiv e-prints*, art. arXiv:2205.01068, May 2022. doi: 10.48550/arXiv.2205.01068.

Zhao, C., Deng, C., Ruan, C., Dai, D., Gao, H., Li, J., Zhang, L., Huang, P., Zhou, S., Ma, S., Liang, W., He, Y., Wang, Y., Liu, Y., and Wei, Y. Insights into deepseek-v3: Scaling challenges and reflections on hardware for ai architectures. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, pp. 1731–1745, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712616. doi: 10.1145/3695053.3731412. URL https://doi.org/10.1145/3695053.3731412.

APPENDIX

Analysis of double quantization error

To analyze the error introduced by naive double quantization, we consider the following procedure for obtaining a columnwise-quantized FP8 tensor from a rowwise-quantized one:

$$x \xrightarrow{\text{quant}} Q_{\text{row}}(x) \xrightarrow{\text{dequant}} D(Q_{\text{row}}(x)) \xrightarrow{\text{quant}} Q_{\text{col}}(D(Q_{\text{row}}(x))) \tag{17}$$

Formally, let $x \in \mathbb{R}^n$ denote the high-precision tensor, let Q_{row} and Q_{col} denote the rowwise and columnwise quantization operators, and let D denote the dequantization operator.

For brevity, define $\widetilde{x} = D(Q_{\text{row}}(x))$. The overall double-quantization error measures how far the final output $Q_{\text{col}}(\widetilde{x})$ deviates from the original x:

$$\begin{aligned} \left\|Q_{\text{col}}(\widetilde{x}) - x\right\|_2 &= \left\|Q_{\text{col}}(\widetilde{x}) - Q_{\text{row}}(x) + Q_{\text{row}}(x) - x\right\|_2 \\ &\le \left\|Q_{\text{col}}(\widetilde{x}) - Q_{\text{row}}(x)\right\|_2 + \left\|Q_{\text{row}}(x) - x\right\|_2 \\ &= \underbrace{\left\|Q_{\text{col}}(\widetilde{x}) - \widetilde{x}\right\|_2}_{\text{2nd quant error}} + \underbrace{\left\|Q_{\text{row}}(x) - x\right\|_2}_{\text{1st quant error}} \end{aligned} \tag{18}$$

where the second line inserts and cancels $Q_{\text{row}}(x)$, the third line applies the triangle inequality, and the fourth line uses the fact that dequantization is lossless, i.e., $\widetilde{x} = D(Q_{\text{row}}(x)) = Q_{\text{row}}(x)$ since BF16 subsumes the full representable range of FP8. This decomposition makes explicit the extra discrepancy introduced by the naive dequantize-transpose-requantize path.

Additional Convergence Validation Results

To further strengthen the stability analysis in Section 4.1, we provide additional convergence results on Qwen3-30B and DeepSeek-V2-Lite.

Figure 7 shows that the training-loss trajectories of BF16 and FP8-Flow-MoE on Qwen3-30B model remain closely matched throughout the entire run. FP8-Flow-MoE tracks the BF16 baseline without visible divergence or instability. The mean relative loss difference stays below 0.1%. Figure 8 further supports this observation from the perspective of optimization stability. Both gradient-norm trajectories remain closely aligned throughout training. FP8-Flow-MoE does not exhibit larger or more frequent spikes than BF16, and both methods converge to a similar low-gradient regime in the late stage. No sign of gradient explosion, gradient collapse, or sustained numerical drift is observed.

As a complement to the loss curves in Section 4.1, Figure 9 shows the gradient-norm trajectories for DeepSeek-V2-Lite. BF16 and FP8-Flow-MoE remain closely aligned throughout training. No sign of gradient explosion, collapse, or accumulated numerical instability is observed under FP8-Flow-MoE. Together with the aligned loss curves in the main text, this confirms that FP8-Flow-MoE preserves training stability comparable to BF16.

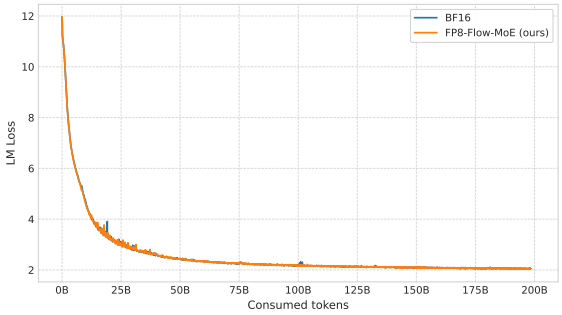


Figure 7: Training loss on Qwen3-30B.

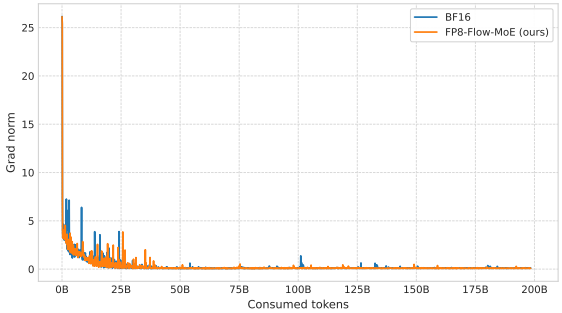


Figure 8: Training gradient norm on Qwen3-30B.

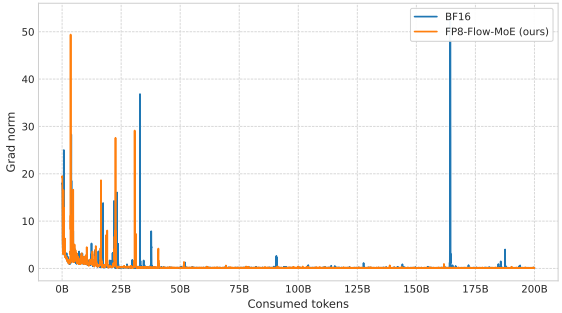


Figure 9: Training gradient norm on DeepSeek-V2-Lite.