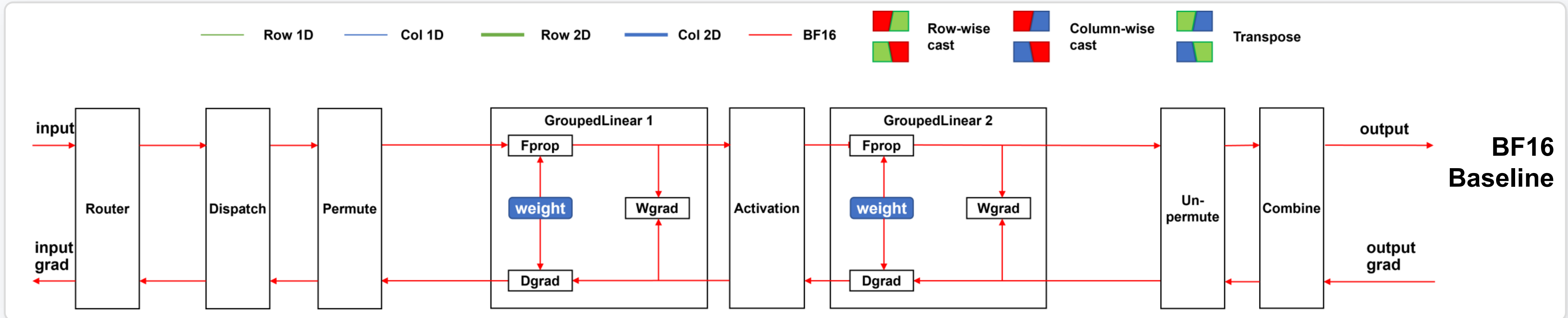
# FP8-Flow-MoE: A CASTING-FREE FP8 RECIPE WITHOUT DOUBLE QUANTIZATION ERROR

Fengjuan Wang, Zhiyi Su, Xingzhu Hu, Cheng Wang, Mou Sun

Zhejianglab

MLSys 2026

# 1. Background and Motivation： The Computational Bottlenecks of MoE




- BF16 MoE: the critical path of one MoE layer

## Background

MoE routes each token to only a small subset of experts, increasing model capacity without a proportional increase in compute

## System View

One MoE layer includes routing, dispatch, permute, expert GEMMs, activation, and combine.

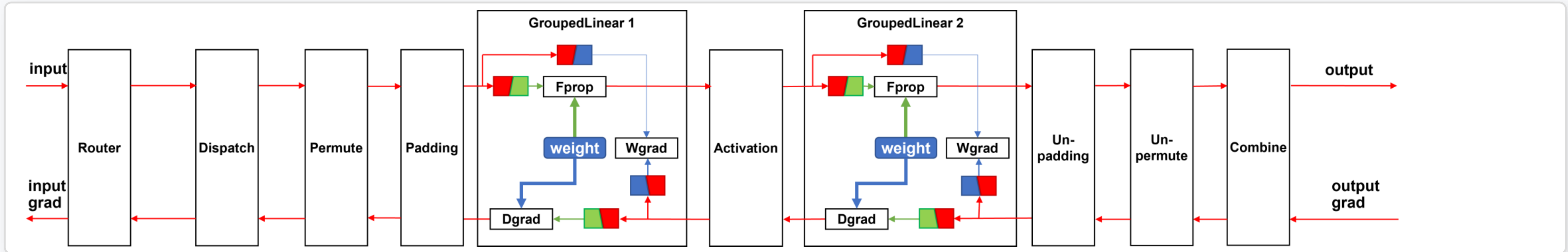
## Bottleneck

Under the BF16 baseline, communication, token reordering, and expert computation together dominate the training cost.

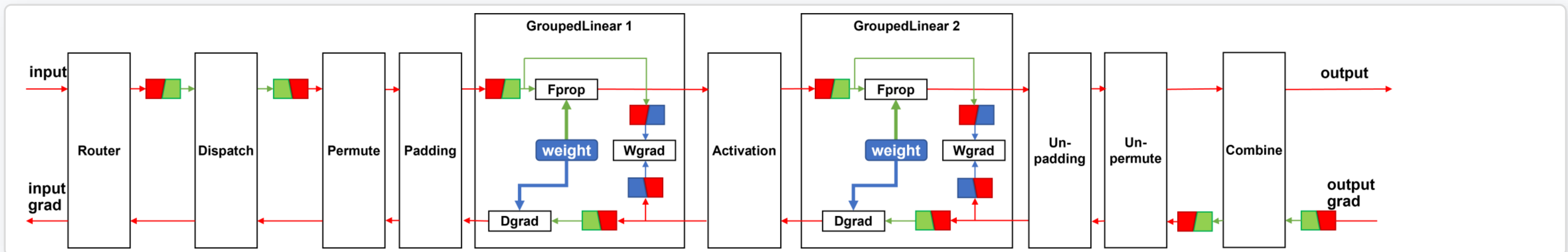
## Motivation

MoE reduces active computation, but not communication, data movement, or memory pressure; this motivates low-precision training such as FP8.

# 1. Background and Problem: Why Existing FP8-MoE Recipes Are Still Not Enough



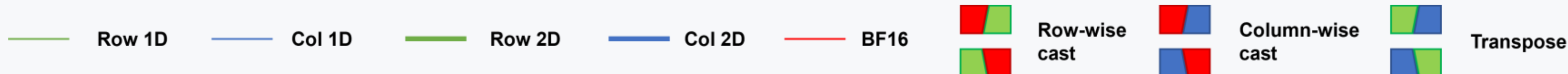

- NVIDIA TransformerEngine Blockwise FP8 : FP8 mainly accelerates grouped GEMM, while communication and data movement remain in higher precision; Q/DQ overhead remains around grouped linear layers.



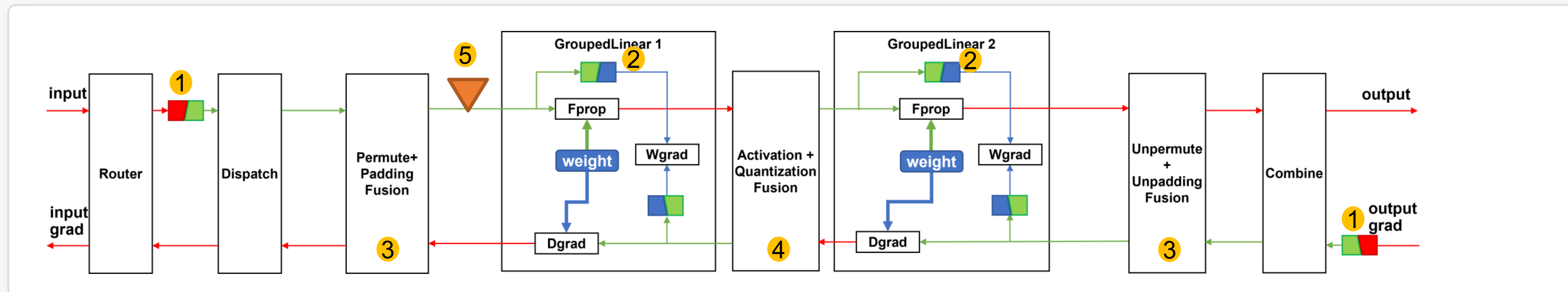

- DeepSeek-V3: FP8 is extended to communication, but one MoE layer still incurs many explicit casts. in the Wgrad path, row-wise FP8 activations must be dequantized, transposed, and requantized into column-wise FP8, which introduces double quantization error.

**Keynote：** The problem is not the lack of FP8 kernels, but the lack of an FP8-centric and quantization-consistent dataflow.

# 2. Method: FP8-Flow-MoE




💡 Core principle: Keep tensors in FP8 along the MoE expert path whenever possible, and retain BF16 only at numerically sensitive boundaries。




### Persistent FP8 Path

1 1 Quantize before dispatch and keep dispatch → permute → expert in FP8

### Scaling-aware Transpose

2 Convert layouts directly in FP8 and avoid double quantization error.

### Fused FP8 Operators

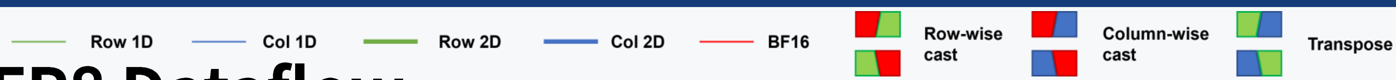
3 4 Fused permute_padding、unpermuted_unpadding、act + quant，to reduce kernel fragmentation.

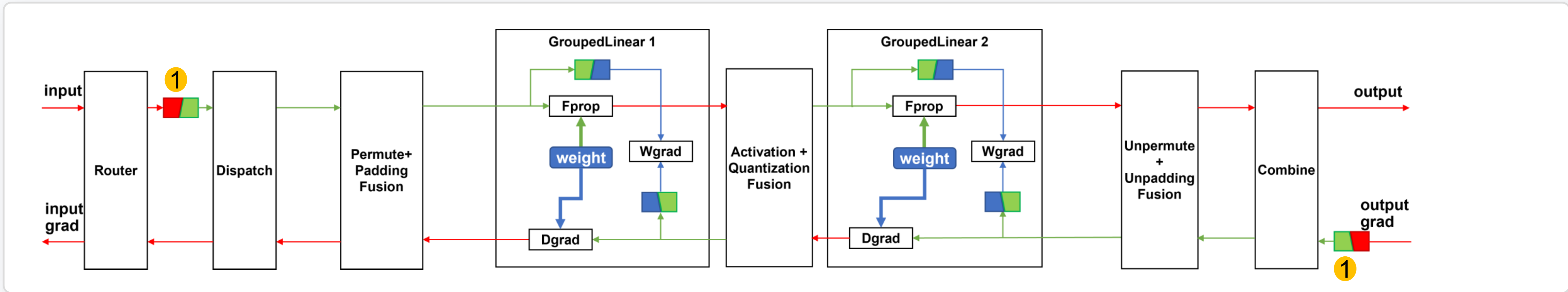
### Fine-grained Recomputation

5 Recompute at the expert level with FP8 activation compression

| Explicit casts | TGS gain | Peak memory reduction | Convergence |
|---|---|---|---|
| 12 → 2 | +21% | -16.5 GB | ≈ BF16 Baseline |

# 3.Key Technique I：Casting-free FP8 Dataflow




## Keep the critical MoE path in FP8 (Casting-free)

- Quantize before dispatch, then keep dispatch → Permute → GroupedLinear(Expert) in FP8
- **Benefits**: reduced cast overhead, FP8 communication, smaller FP8 buffers, and lower data-movement cost

## Retain BF16 only at two numerically sensitive boundaries

- Grouped Linear 1 ↔ Activation：nonlinear operations are sensitive to quantization noise
- Grouped Linear 2 ↔ Combine/Reduction：reduction requires larger dynamic range and is more prone to overflow.

Keep the high-cost MoE path in FP8 whenever possible, while retaining BF16 only at necessary boundaries for stability.

# 4.Key Technique Ⅱ： Scaling-aware FP8 Transpose

**Why it is needed: Wgrad requires column-wise FP8, while Fprop and Dgrad use row-wise FP8**

## Naive solution: Double Quantization

FP8(row) → dequantize to BF16 → transpose → requantize to FP8(col). This introduces two independent quantization steps and accumulates rounding 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:

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

Formally, let $x \in \mathbb{R}^n$ denote the high-precision tensor, let $Q_{\text{row}}$ and $Q_{\text{col}}$ 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 $x$:

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

## Our solution： Scaling-aware Transpose

- If scaling factors are powers of two, layout conversion reduces to exponent adjustment.
- Align scales within each block and convert directly in FP8.
- This avoids BF16 fallback and replaces three operators with one.

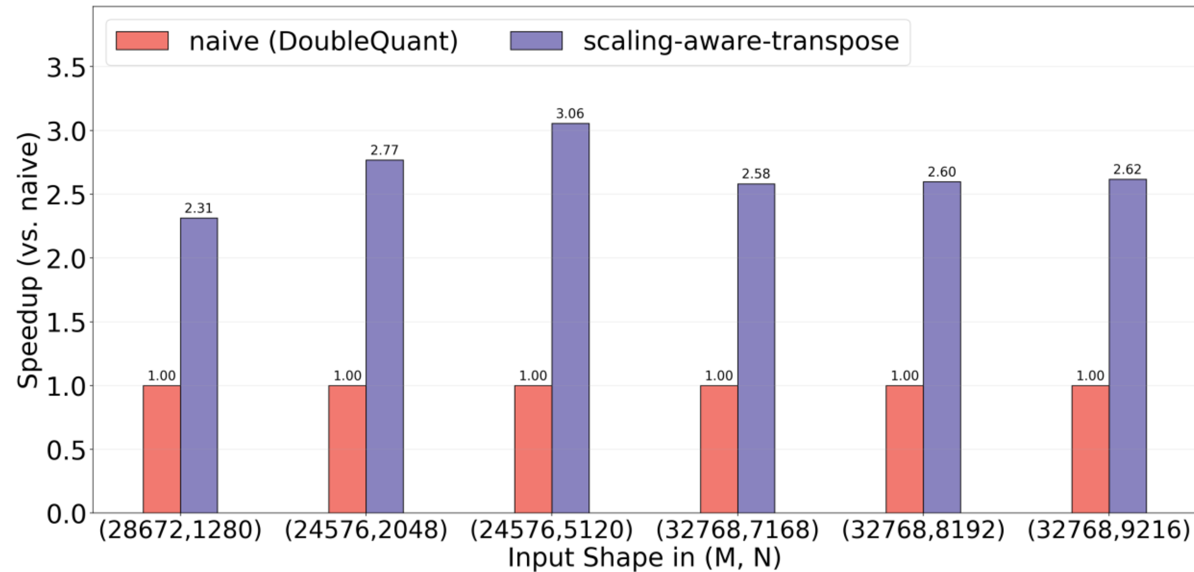
**★ Takeaway: Solves both a numerical stability problem and a system efficiency problem**

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

1: **Input:** row-wise data $X^{row}_{i,j}$, scaling factor $S^{row}_i$ for $i \in [i_0, i_0{+}B{-}1]$, $j \in [j_0, j_0{+}B{-}1]$
2: **Output:** column-wise data $X^{col}_{j,i}$ and scaling factor $S^{col}_j$

3: $S_{\max} = \max_{i\in[i_0,i_0+B-1]} S^{row}_i$
4: $\forall j \in [j_0, j_0{+}B{-}1]$, $S^{col}_j = S_{\max}$
5: **for** each element $(i, j)$ in the block **do**
6: $k = \log_2(S_{\max}/S^{row}_i)$ $\{k \in \mathbb{N}\}$
7: **if** $k = 0$ **then**
8: $X^{col}_{j,i} = X^{row}_{i,j}$
9: **else**
10: $SN = X^{row}_{i,j}$ & `0x80` {sign bit}
11: $E = X^{row}_{i,j}$ & `0x78` {exponent bits}
12: $M = X^{row}_{i,j}$ & `0x07` {mantissa bits}
13: $E_{\text{new}} = E - (k \ll 3)$
14: $E_{\text{new}} = 0$ **if** $E_{\text{new}} <$ `0x08` {flush-to-zero}
15: $X^{col}_{j,i} = SN \mid E_{\text{new}} \mid M$
16: **end if**
17: **end for**

# 4.Key Technique Ⅱ： Scaling-aware FP8 Transpose (Micro-benchmark)




- Compared with the naive (dequantize → transpose → quantize) implementation, scaling-aware FP8 transpose operator up to 3x speedup

# 5. Key Technique III： High-Performance FP8 Kernels （Permute + Padding)

## Fused Permute + Padding

### Motivation

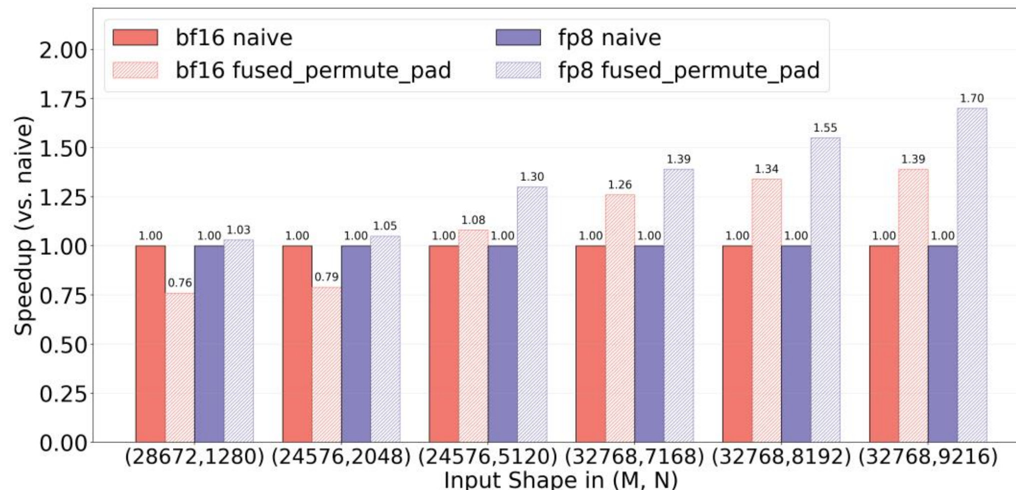
- Permute and padding are memory-bound operators and together account for a large fraction of MoE-layer latency

### Benefits

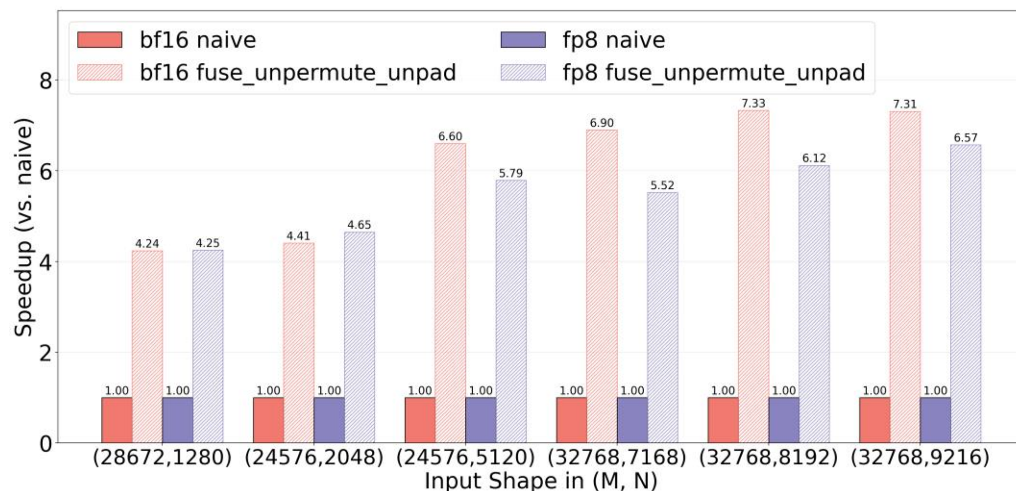
- Fusing them reduces HBM traffic, kernel launches, and intermediate buffer overhead

### Micro-benchmarks

- Fused permute+pad: up to 1.7× speedup
- Fused unpermute+unpad: up to 6.6× speedup
- The fusion also reduces peak memory by about ~1 GB in end-to-end training.




- Fused permute+padding case, we observe up to 1.7× speedup




- Fused unpermute+unpadding kernel, the improvement reaches as 6.6×

# 5. Key Technique III： High-Performance FP8 Kernels(SwiGLU + Quant)

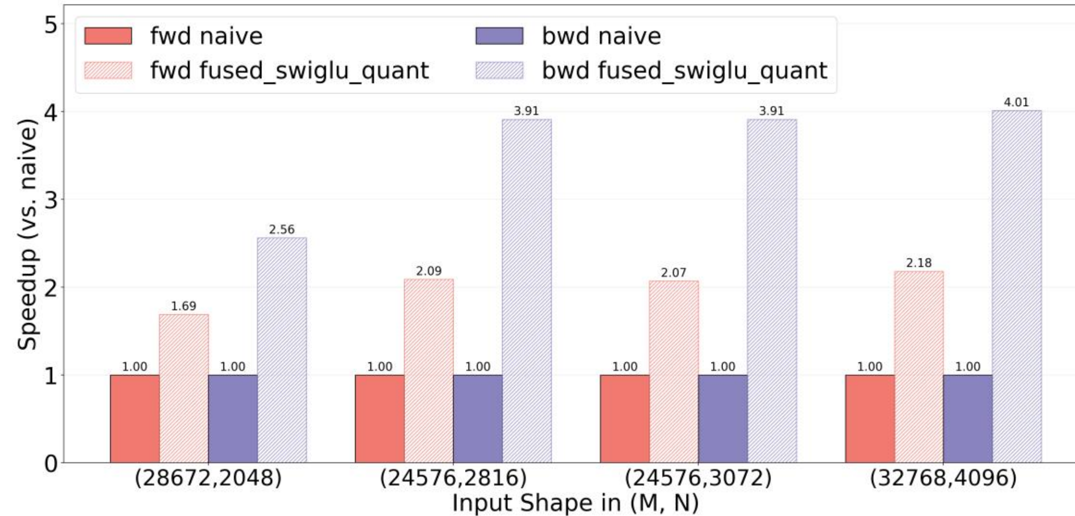
## Fused SwiGLU + Quant

### Motivation

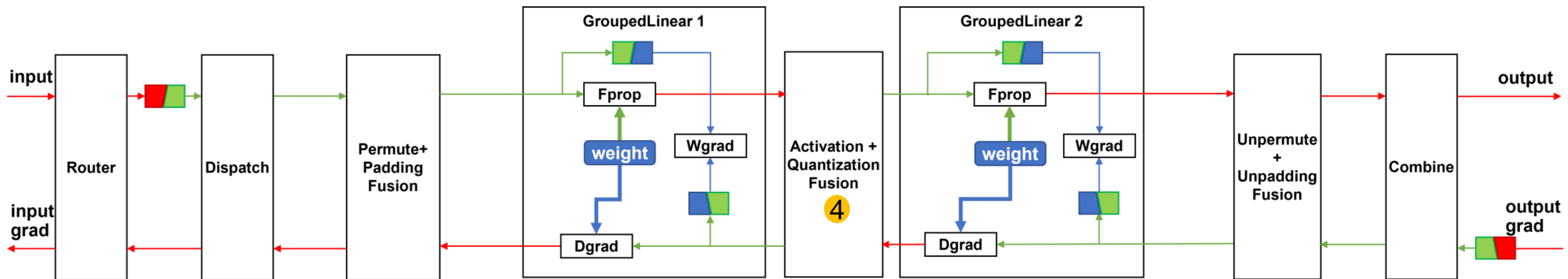
- In a naive FP8 pipeline, quantization around activation introduces extra kernels on the critical path.
- We fuse SwiGLU and quantization so that activation directly produces FP8 outputs.

### Micro-benchmarks

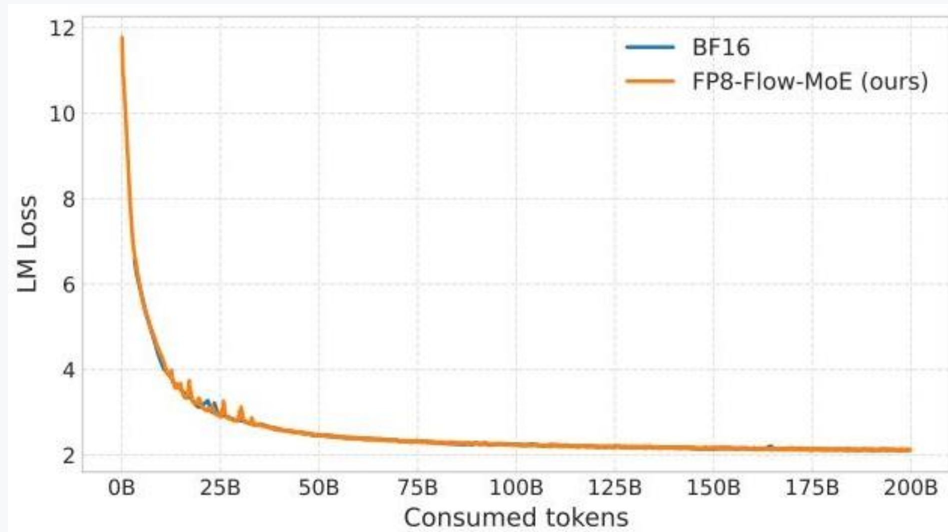
- Forward: up to 2.18× speedup
- Backward: up to 4× speedup
- This reduces kernel launch overhead and avoids extra activation round-trips.



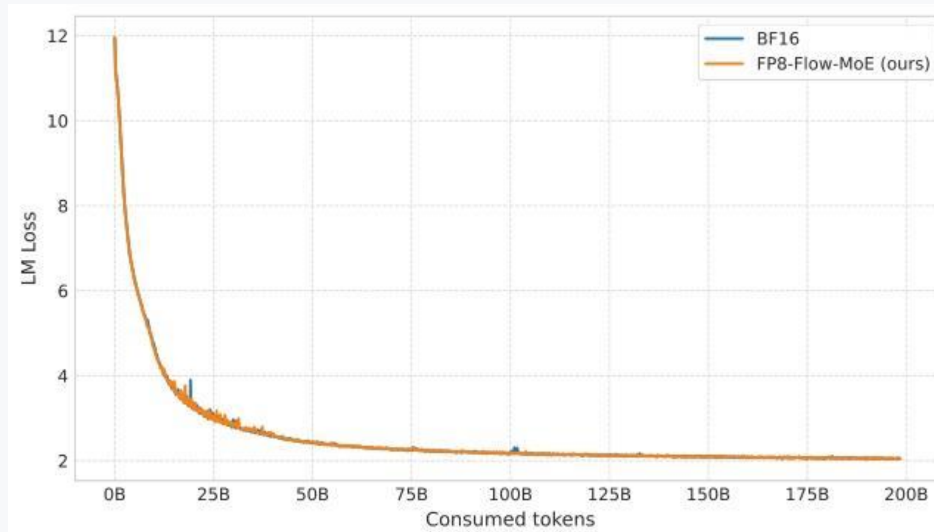

- In the forward pass, the fused kernel achieves 2.18× speedup.
- In the backward pass, reaching 4×



# 6. Convergence Validation




Training loss on DeepSeek-V2-Lite (16B)




Additional validation on Qwen3-30B-A3B

**Setup**： DeepSeek-V2-Lite is trained from scratch for 200B tokens; Qwen3-30B provides additional convergence validation
**Conclusion**：**Train in FP8, converge like BF16**； Relative loss error remains below 0.19%.

# 7. End-to-End Efficiency: higher TGS, lower memory, and better scalability at larger EP

TGS (vs BF16)
**+16%**

TGS (vs TE Blockwise FP8)
**+21%**

Peak Mem (vs BF16)
**-8 GB**

Peak Mem (vs TE Blockwise FP8)
**-16.5 GB**

| 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)**

**Primary benchmark**: DeepSeek-V3 (671B), evaluated under multiple expert-parallel and checkpointing settings.
**Platform:** 32-node NVIDIA Hopper cluster, 8 GPUs/node, 80 GB/GPU
**Throughput:** up to +16% vs BF16, and +21% vs TE blockwise FP8
**Peak memory:** up to -8 GB vs BF16, and -16.5 GB vs TE FP8 baselines At EP=32, some baselines run out of memory, while FP8-Flow-MoE remains stable.

# THANKS

## Q & A

**Upstream progress: TransformerEngine merged several PRs; Megatron-LM support is in progress.**

- https://github.com/NVIDIA/Megatron-LM/pull/2764 （Open）
- https://github.com/NVIDIA/TransformerEngine/pull/2544 （Open）
- https://github.com/NVIDIA/Megatron-LM/pull/2763 （Merged）
- https://github.com/NVIDIA/Megatron-LM/pull/4038 （Merged）
- https://github.com/NVIDIA/TransformerEngine/pull/1921 （Merged）
- https://github.com/NVIDIA/TransformerEngine/pull/2144 （Merged）
- https://github.com/NVIDIA/TransformerEngine/pull/2026 （Merged）
- https://github.com/NVIDIA/TransformerEngine/pull/1866 （Merged）
- https://github.com/NVIDIA/TransformerEngine/pull/2547 （Merged）