



CONFERENCE

—NORTH AMERICA 2026—

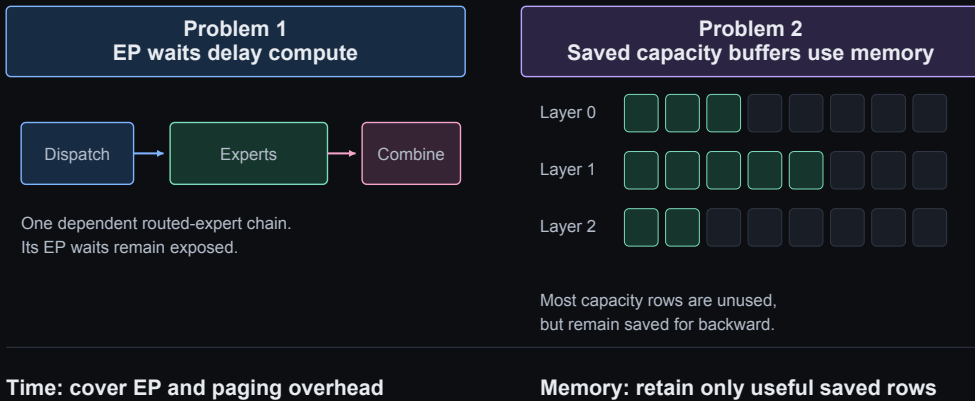
EP-Overlap: Hiding Expert-Parallel Communication in MoE Training

Sanket Purandare (PyTorch, Meta)

Syed Ahmed (NVIDIA)

Two challenges to scaling MoE training

Exposed EP communication costs time. Capacity-padded activations cost memory.



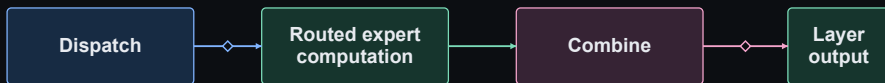
MoE scaling faces exposed EP communication and retained worst-case activation memory. EP overlap addresses time, while paged stashing compacts saved state.

Why EP communication is exposed

Dispatch and combine must complete before their dependent computations can read the results.

One routed-expert call has a dependent communication / compute path

Combine must complete before the layer can consume expert results.



The expert computation depends on dispatch. The layer output depends on combine. This single path has no peer computation that can cover its own pending exchange.

→ Data / completion

◇ Completion wait

The dispatch-to-expert and combine-to-output dependencies expose EP time. A call's own dependent compute cannot cover its pending exchange.

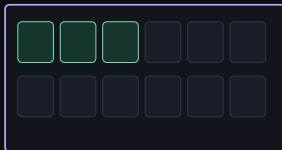
Arrows show required data and completion dependencies before reads.

The cost of saving capacity-padded activations

Capacity buffers avoid host synchronization and preserve fixed CUDA graph addresses.

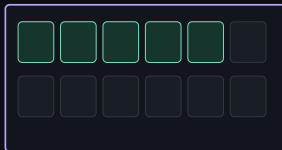
Backward retains the entire capacity-padded tensor from every live layer

Layer 0: saved [C, H]



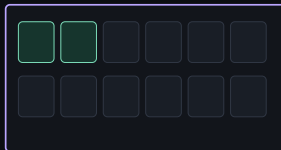
All C rows retained; only 3 rows useful

Layer 1: saved [C, H]



All C rows retained; only 5 rows useful

Layer 2: saved [C, H]



All C rows retained; only 2 rows useful

$O(\text{num_layers} \times \text{worst-case activation bytes})$

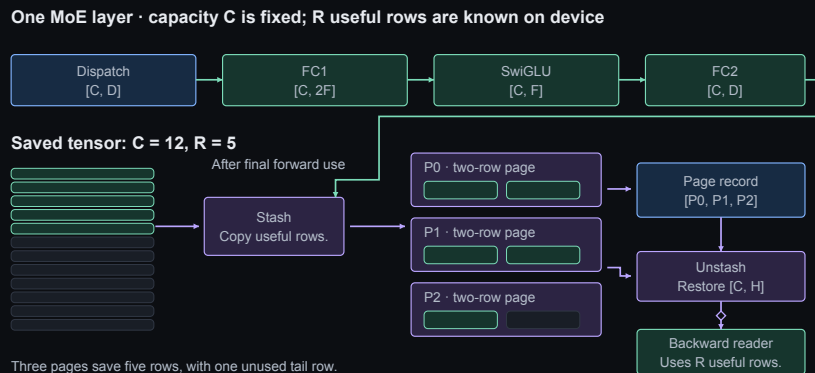
Saving actual rows would scale with their useful sizes instead.

Device-only counts and fixed CUDA replay addresses motivate capacity buffers. Saving full padded tensors scales as $O(\text{num_layers} \times \text{worst-case activation bytes})$, rather than useful rows.

Worst-case tensor bytes = $C \times H \times \text{bytes per element}$. PP can keep several layer/microbatch instances live.

Paged Stashing: Solving the worst-case memory by retaining actual activation rows

Save useful rows in reusable fixed-size GPU pages, then restore them for backward.



Logical pages compact useful saved rows into persistent GPU storage. A per-tensor page record guides stash/unstash; completion protects reads and buffer reuse.

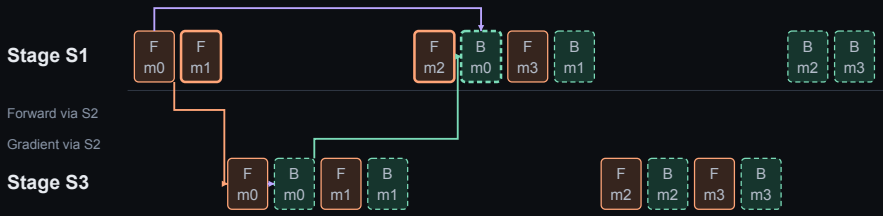
Illustrative $C = 12, R = 5$, two-row pages. Real default pages contain 64 tokens.

Paged stashing handles out-of-order microbatch access

Retrieve and free each record when its microbatch needs it, without external fragmentation.

Example: two local stages S1 and S3 on one PP rank

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$; S_0 and S_2 run on the other rank.



Paged storage supports out-of-order record access in multi-stage PP. Uniform free pages can be reused independently of record lifetime, avoiding external fragmentation.

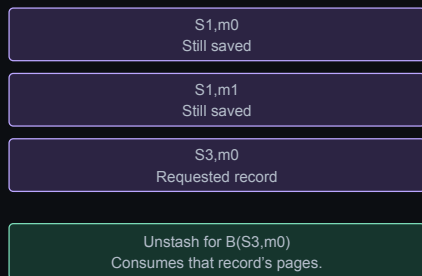
Ordinary interleaved 1F1B is one example of a multi-stage PP schedule. Tail pages can contain unused rows.

Out-of-order activation access in multi-stage PP

Multiple stages on one rank can request older and newer saved records in a mixed order.

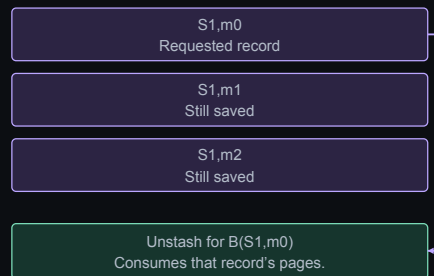
The shared pager must follow stage + microbatch records, not one FIFO or LIFO stack

Case 1 · not FIFO



Older S1 records remain saved.
Access is not oldest-first.

Case 2 · not LIFO



Newer m1/m2 records remain saved.
Access is not newest-first.

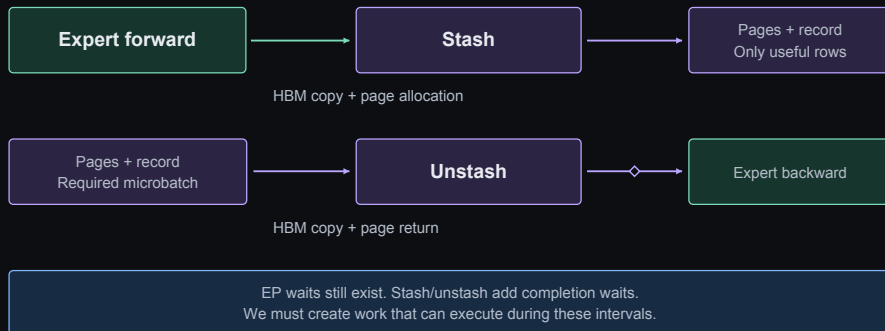
Multi-stage PP schedules can request records out of forward or stack order. The saved record belongs to a stage/layer/microbatch and must be retrieved by that identity.

This is stage/microbatch lifetime order. Tensor accesses within each layer call retain their normal dependencies.

Stash and unstash overhead

Memory compaction adds HBM copies, page bookkeeping and completion waits.

Smaller saved state introduces GPU data movement and page-management work



Stash/unstash reduce retained capacity-buffer memory, but add HBM copies, page management, and completion waits. The execution-time problem remains.

Paged stashing solves retained worst-case memory. EP and page-copy intervals still need independent computation.

EP-Overlap: Eager chunks and joint F/B graphs

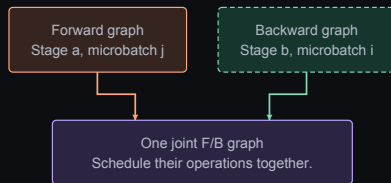
Paged stashing fixes retained memory. Now cover exposed EP communication and page transfers.

SPMD: Eager token chunking



Trace both calls into one graph.
Use peer chunks and shared experts
to cover EP and page transfers.

PP: Joint forward/backward graph



Use the opposite source's computation
to cover EP and page transfers.
Keep each source's data dependencies.

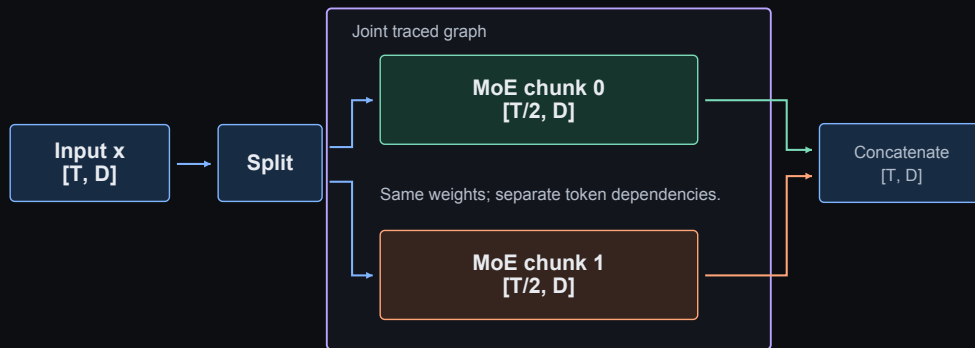
SPMD splits MoE token rows into eager calls and traces them together, creating peer chunk computation. PP joins independent forward/backward source graphs, exposing their internal operations to one scheduler. Both approaches schedule EP and paging against the resulting compute windows.

Shared-expert computation provides another EP-independent window when the MoE has shared experts.

Eager token chunks create peer computation

Split MoE token rows before tracing, then interleave both calls by their dependencies.

Trace both calls into one joint forward/backward graph

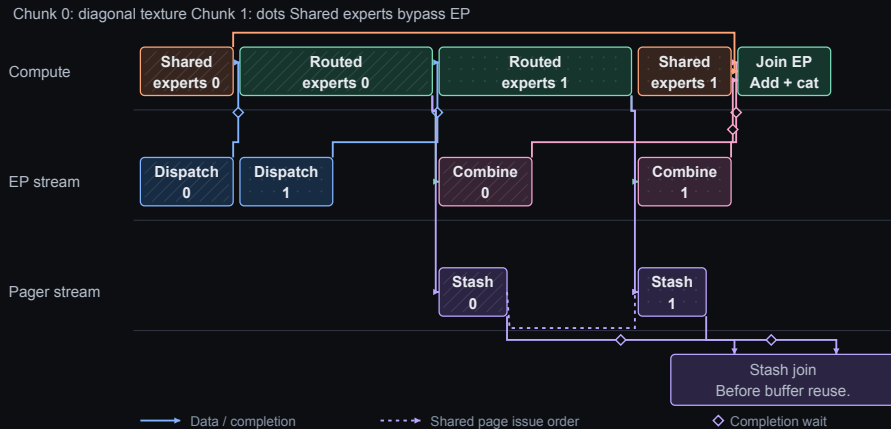


The scheduler can interleave the traced units. Concatenation preserves token order.

Eager token splitting creates two activation-independent MoE calls. Trace them together, schedule their internals, and concatenate outputs in token order.

SPMD overlap with routed and shared experts

Shared experts bypass EP. The peer token chunk supplies additional routed-expert work.

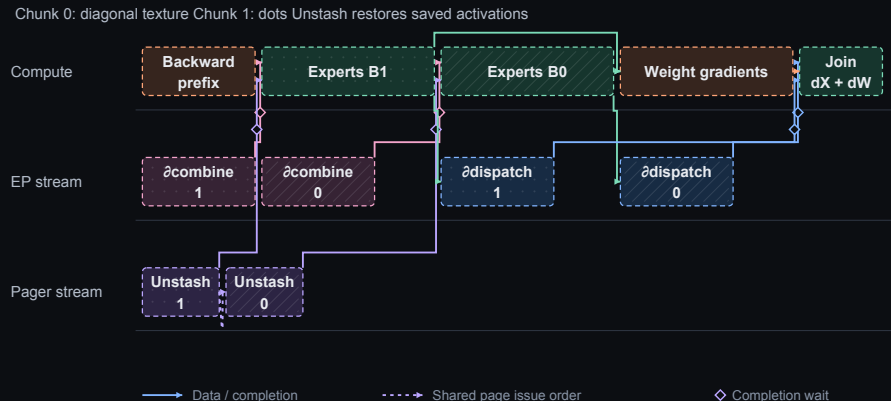


Shared experts cover dispatch/combine without consuming EP results. The peer token chunk adds routed-expert computation to overlap EP and stashing. Own-source data and copy lifetimes remain protected.

Illustrative legal schedule. Shared-expert inputs and weights must be ready before their computation can run.

SPMD backward: Overlap EP and activation restoration

Restore saved rows and exchange gradients while independent computation proceeds.



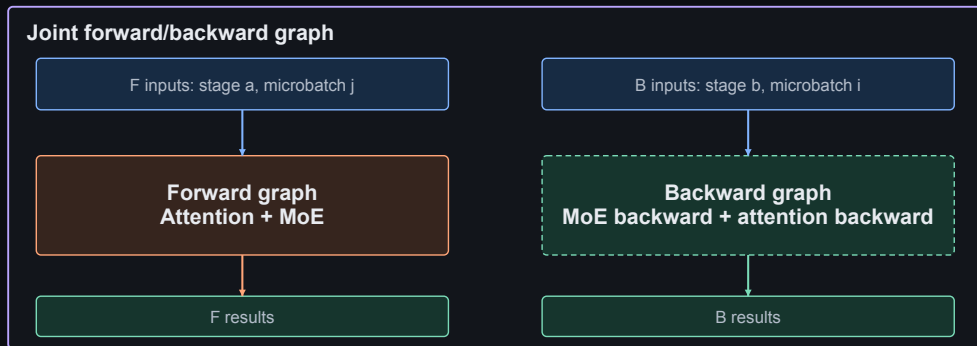
Unstash and incoming EP gradients both precede each expert backward. Peer and weight-gradient work cover later exchanges; native completion and page issue ordering remain explicit.

$\partial\text{combine}$ sends gradients to experts; $\partial\text{dispatch}$ returns input gradients. Dashed boxes are backward operations.

PP overlap starts with a joint forward/backward graph

DualPipeV and Interleaved1F1BCombined supply independent forward/backward actions.

One executable graph exposes both sources to a common scheduler

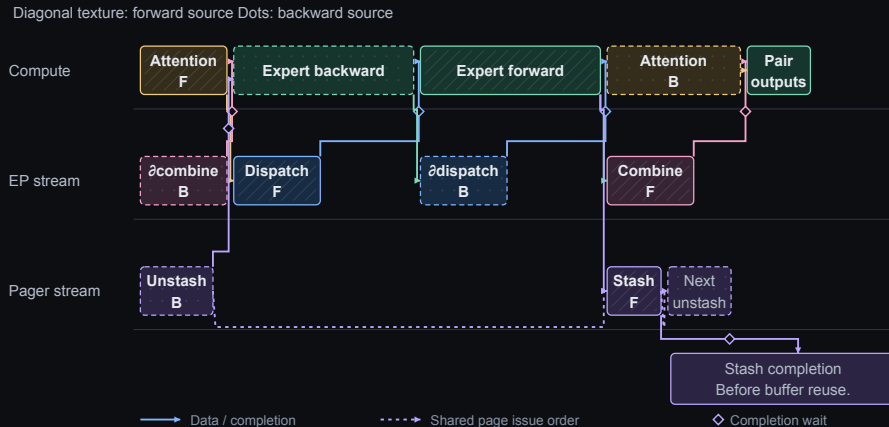


Inputs, outputs and dependencies stay source-owned. The scheduling scope is now shared.

Joining independent F/B source graphs exposes operation-level overlap in one executable region while preserving source-owned inputs, outputs and dependencies.

Scheduling EP and paging across forward/backward graphs

Keep each source's dependencies while using the other source's computation to cover transfers.

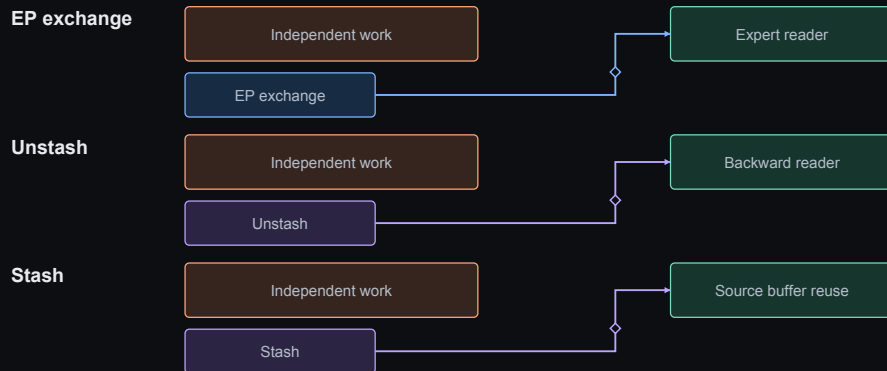


The joint F/B region creates an opposite-source compute window around EP and stash/unstash. Each source's data dependencies and the shared pager/lifetime ordering remain intact.

Dotted arrows retain shared pager issue order. Stash completion constrains buffer reuse, not unrelated F/B data.

Safe overlap: Wait before reads and buffer reuse

Issue transfers early, do independent work, and wait at the dependent read or reuse.



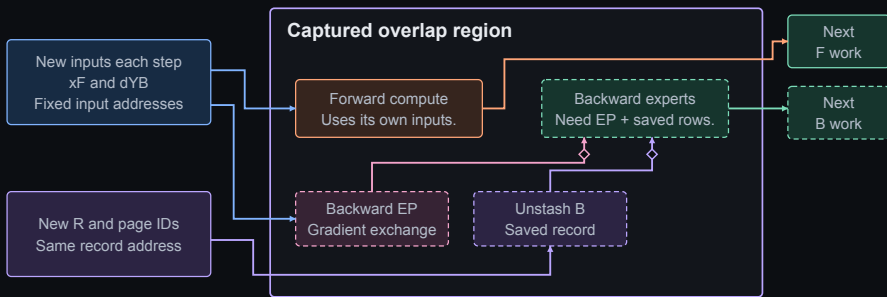
Use independent work between transfer launch and completion. Wait before EP/unstash results are read, and retain stash-copy inputs until their storage can safely be reused.

Shared pager issue order is preserved separately from the completion required before reads or buffer reuse.

Replay preserves the captured overlap schedule

Input values and page IDs change. Captured operations, edges and addresses remain fixed.

The operation graph and addresses are captured once



Work-buffer addresses α and page-pool address β stay fixed



Values change; the captured nodes and edges persist.

CUDA replay changes input/count/page-record values while preserving the captured overlap operations, completion dependencies and fixed GPU addresses.

Illustrative excerpt of a full captured step. Two-row pages show changing record values inside fixed storage.

SPMD and PP: Different constructions, shared guarantees

SPMD creates token chunks. PP joins independent graphs. Both share storage and lifetime rules.

Contract	SPMD construction	PP construction
Created work units	Eager MoE calls on two token chunks	One joint graph from independent F/B actions
New scheduling scope	The traced chunk forward/backward program	The mixed forward/backward source program
Output capacity	Split the configured raw budget, then align	Keep the full per-microbatch bound
Data protection	Each chunk's EP/unstash waits before reads	Each source's EP/unstash waits before reads
Page policy	Useful rows; record-based access; ordered issue	Useful rows; record-based access; ordered issue

SPMD constructs eager token units; PP constructs a joint F/B region. Both then schedule communication against the created work and preserve the same page/lifetime contracts.

The PP consumer is defined by independent F/B actions, rather than one schedule name.

Throughput gains on DeepSeek V3 15.7B

HybridEP and FA4 varlen, full CUDA graphs, no AC/SAC. Page stashing is active in both arms.

Case	No overlap (tokens/s)	Overlap (tokens/s)	Speedup
SPMD32, forced balanced capacity 1×	893,028	1,084,136	1.214×
SPMD32, real routing capacity 8×	360,551	445,641	1.236×
PP64, forced balanced capacity 1×	1,239,218	1,588,677	1.282×
PP64, real routing capacity 8×	605,265	764,450	1.263×

Overlap improves throughput by 21.4–28.2% across these four matched comparisons.

Correctness and benchmark setup

TorchTitan GraphTrainer on GB300 NVL. DeepSeek V3 15.7B, HybridEP and FA4 varlen.

Seven exact recorded samples across every rank

Loss: maximum difference 0

Gradient norm: maximum difference 0

SPMD: 32 GPUs

DP = EP32, TP1

Local batch 16

Two eager chunks

PP: 64 GPUs

PP2, DP = EP32, TP1

Four stages per rank

64 microbatches of 4

Interleaved1F1BCombined, no EP chunking

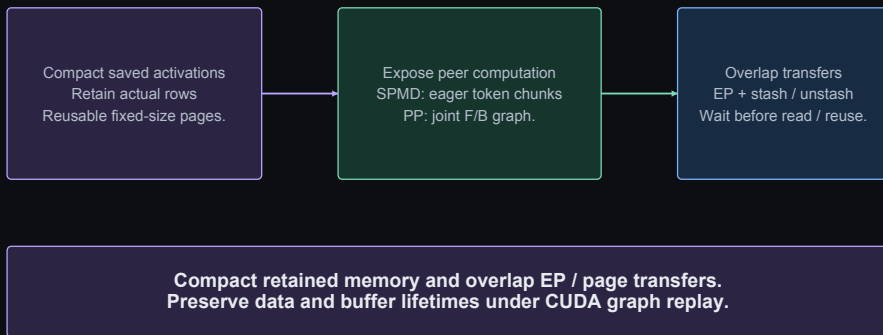
1× and 8× refer to the HybridEP capacity factor.

All seven recorded loss and gradient-norm samples match exactly across every rank, with maximum difference 0.

Sequence length 1024. Throughput uses steps 11–50 of 60. Both arms use full CUDA graphs with no AC/SAC.

Putting paged stashing and EP overlap together

Retain actual rows and create peer computation to overlap EP communication and page transfers.



Paged stashing compacts memory. EP-Overlap constructs independent work and schedules exchanges/copies with it. Dependency joins make the combined execution safe for CUDA graph replay.

The remaining exposure depends on startup/tail work, unpaired actions, and how much independent compute the constructed units provide.