



Introducing NCCL Extensions: Communication Patterns for Modern AI

Sreeram Potluri

Distinguished Software Architect, NVIDIA

Artem Polyakov

Senior Software Engineer, NVIDIA

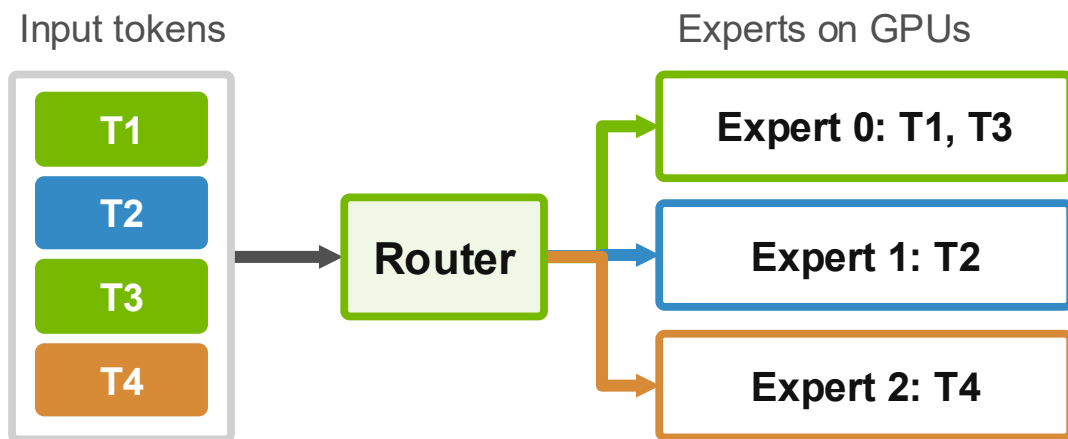
PyTorch Conference Oct 2026

AI needs specialized communication

MoE and reinforcement learning add patterns beyond standard collectives

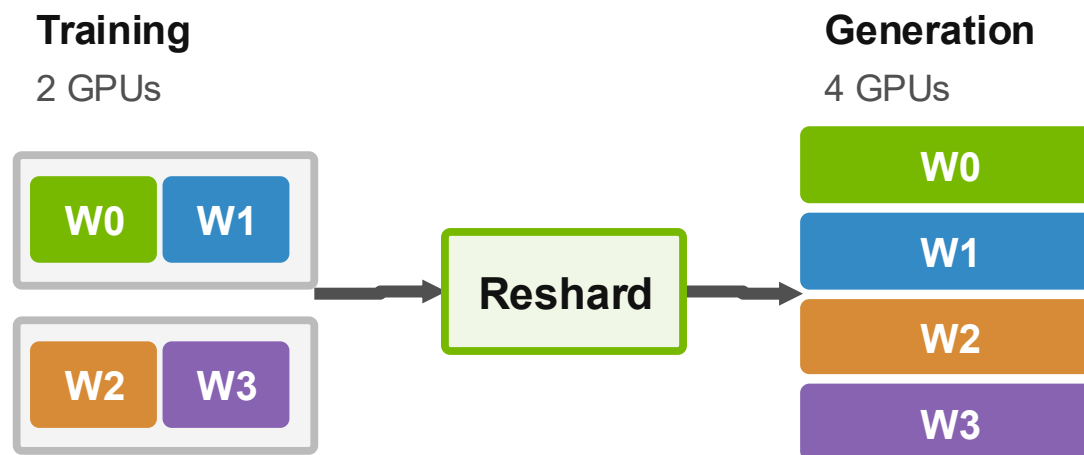
MoE token routing

Tokens choose different experts



RL weight rollout

Updated weights need a different layout



Same weights, new sharding

Each new pattern adds communication tuning and integration work

NCCL Extensions: delivering modern communication patterns

Specialized patterns. Shared NCCL foundation.

Your AI application

Python bindings

NCCL Extensions

EP

Token routing

M2N

Weight rollout

And more

NCCL host and device APIs

Optimize communication

Use implementations designed for your workload.

Reduce engineering effort

Reuse Python bindings and shared packaging.

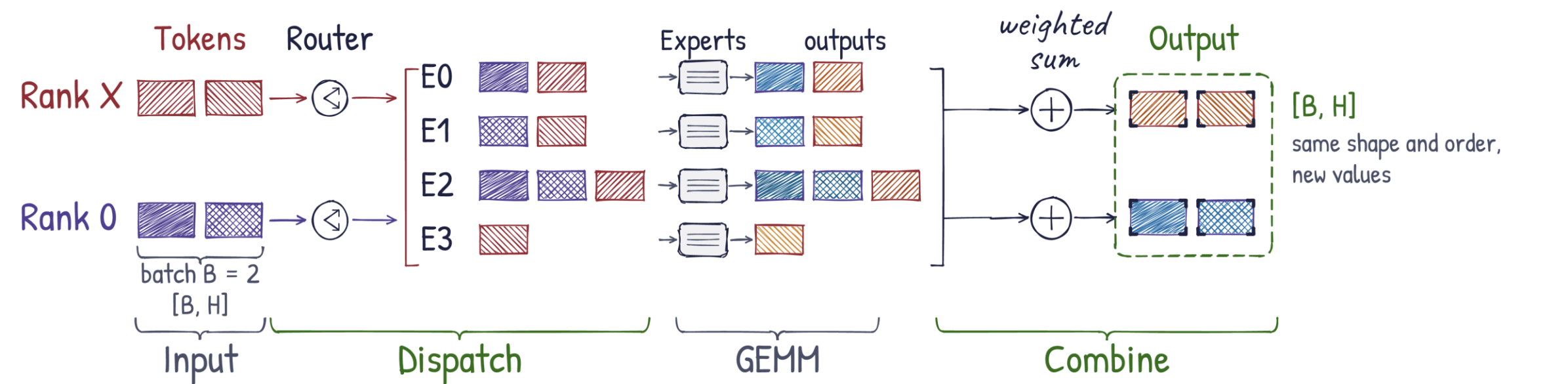
github.com/NVIDIA/nvcl-extensions



NCCL EP

MoE communication requirement

Dispatch tokens to their experts, Combine the outputs back



Token

one row of activations $[H]$; a rank holds a batch of B of them

Router

picks the top- k experts for each token, with a weight for each

Expert

an FFN that lives on one GPU; experts are spread over the EP group

Weighted sum

$y = \sum_k w_k \cdot E_k(x)$ over the token's k experts — a token-sized result

Dispatch

deliver every token to its k experts, on any GPU, with its routing metadata

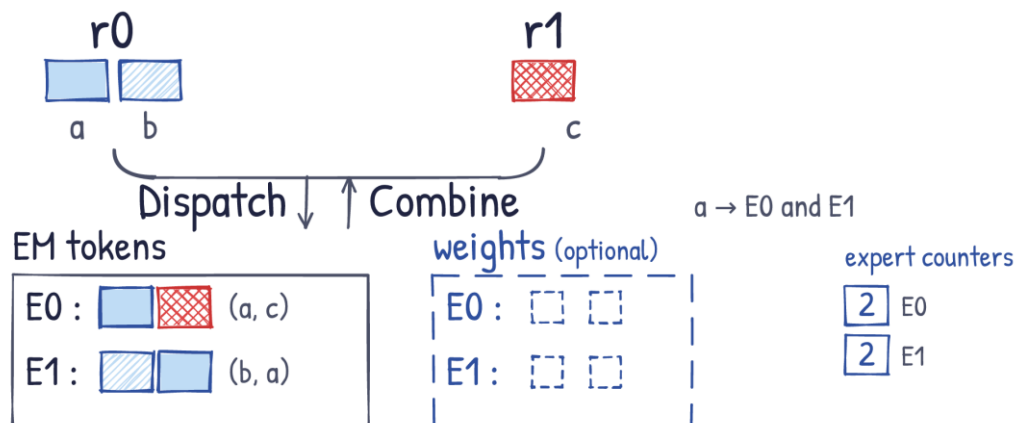
Combine

return the expert outputs to the token's home rank and reduce them

Expert-side layout and metadata

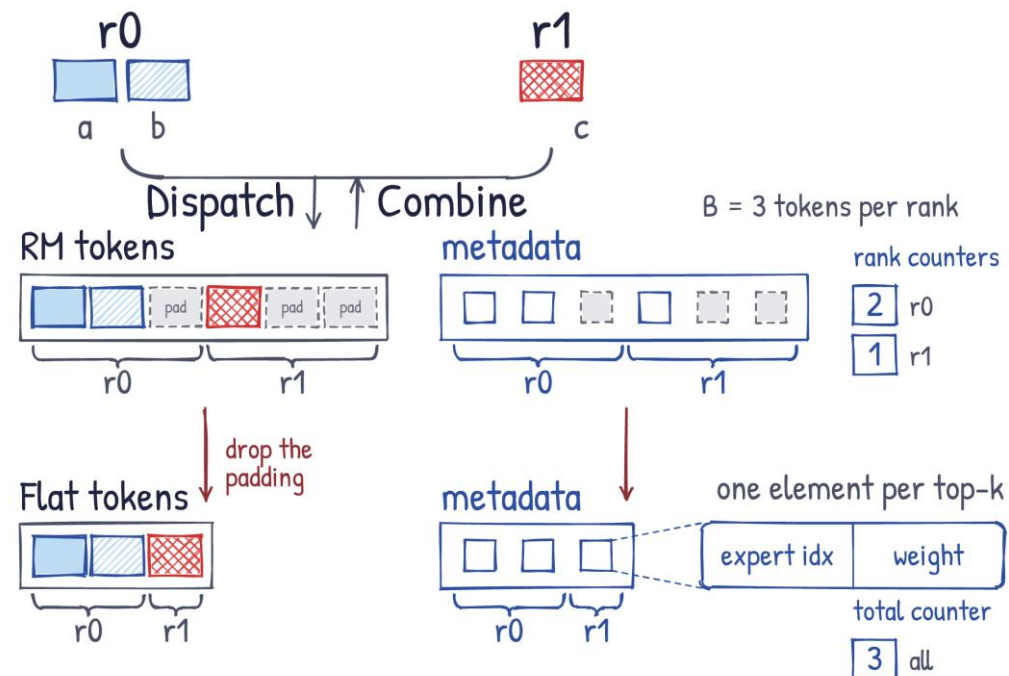
One layout for both operations: Dispatch writes it, experts consume it, Combine reads it back

EXPERT-MAJOR — rows pre-sorted by local expert



- Each expert reads its own contiguous slice; a token routed to two local experts appears in both
- Metadata out: **expert_counters** (+ **expert_offsets** in HT)
- Weights: returned on request, or applied by NCCL EP inside Combine — nothing to carry

RANK-MAJOR (RM) / FLAT — one slot per received token



- Grouped by source rank (LL rank-major) or not grouped at all (HT flat)
- Metadata out: **recv_topk_idx**, **recv_topk_weights**, per-rank / total counts
- The application routes each slot to its local experts and pre-reduces before Combine

NCCL Expert Parallelism (EP) Extension

Expert-parallel communication, built on NCCL

FEATURES

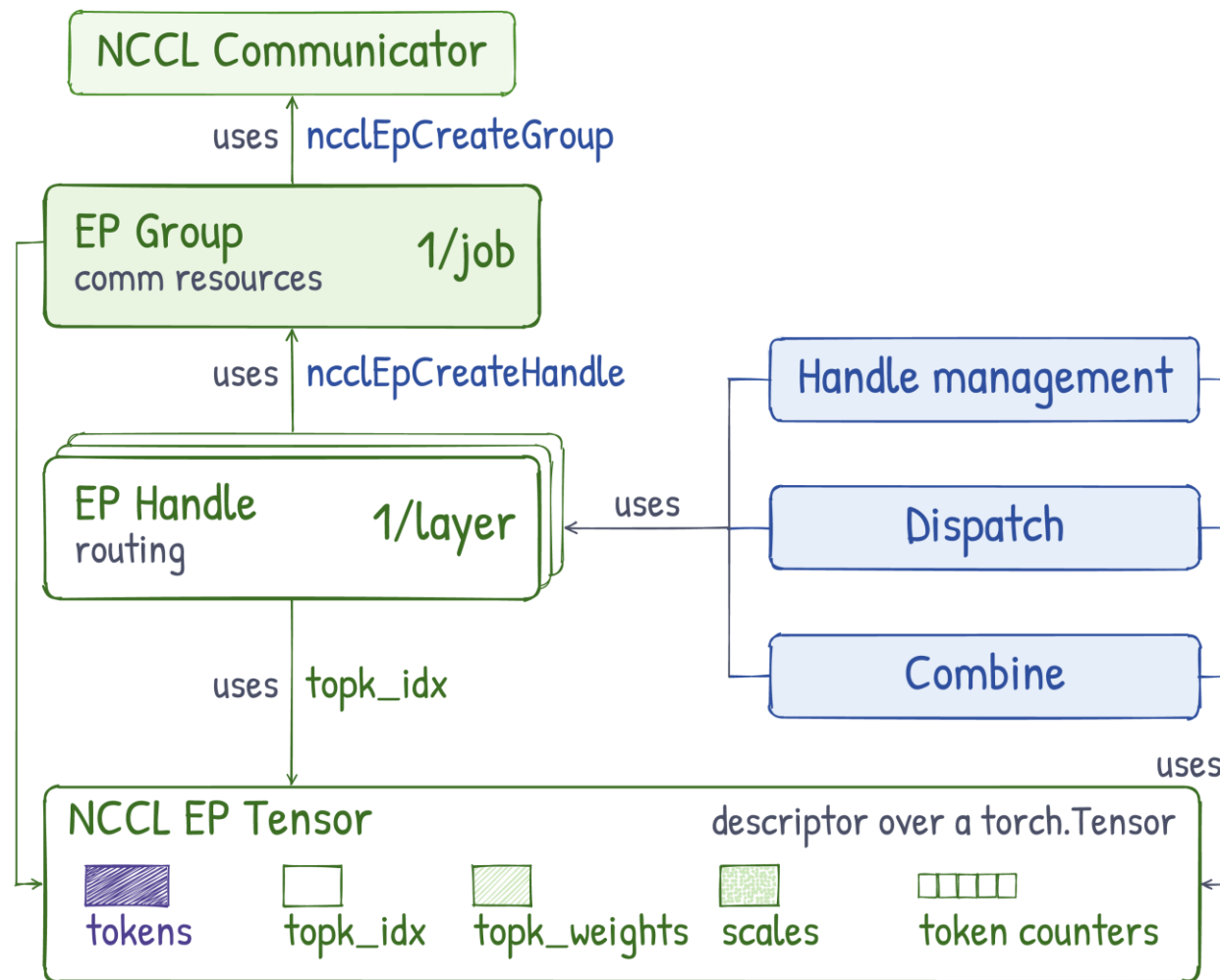
- Fully relies on NCCL for communication
- CUDA Graph support
- Framework integration
 - Expert-major and deduplicated layouts
 - Flexible expert indexing formats
- Native quantization
 - MXFP8 (HT), DeepSeek FP8 and NVFP4 (LL)
- Fault tolerance (LL)
- Zero-copy: works directly on torch tensors
- Token overflow policies (HT)

COMING IN v0.3 — end of October 2026

- Compute Fabric Transport (CFT) support, including counted writes
- Attention-FFN Disaggregation (AFD) support
- Optimized training algorithms (Pull-push mode)
- Improved decode latency (NVLink specialization)

NCCL EP — a dedicated EP communication library

Objects hold the state, operations move the tokens



NCCL EP Group (`ncclEpGroup_t`)

`ncclEpCreateGroup(comm, config)`

- Once per job: HT / LL, experts, token budgets
- Owns buffers, QPs, channels

NCCL EP Handle (`ncclEpHandle_t`)

`ncclEpCreateHandle(group, layout, topk_idx)`

- One per MoE layer: routing + output layout
- Many handles per group; updated in place

NCCL EP Tensor (`ncclEpTensor_t`)

`{ data, ndim, sizes, datatype }`

- A descriptor over an existing buffer, no copies
- Tokens, indices, weights, scales, counters

NCCL EP critical path

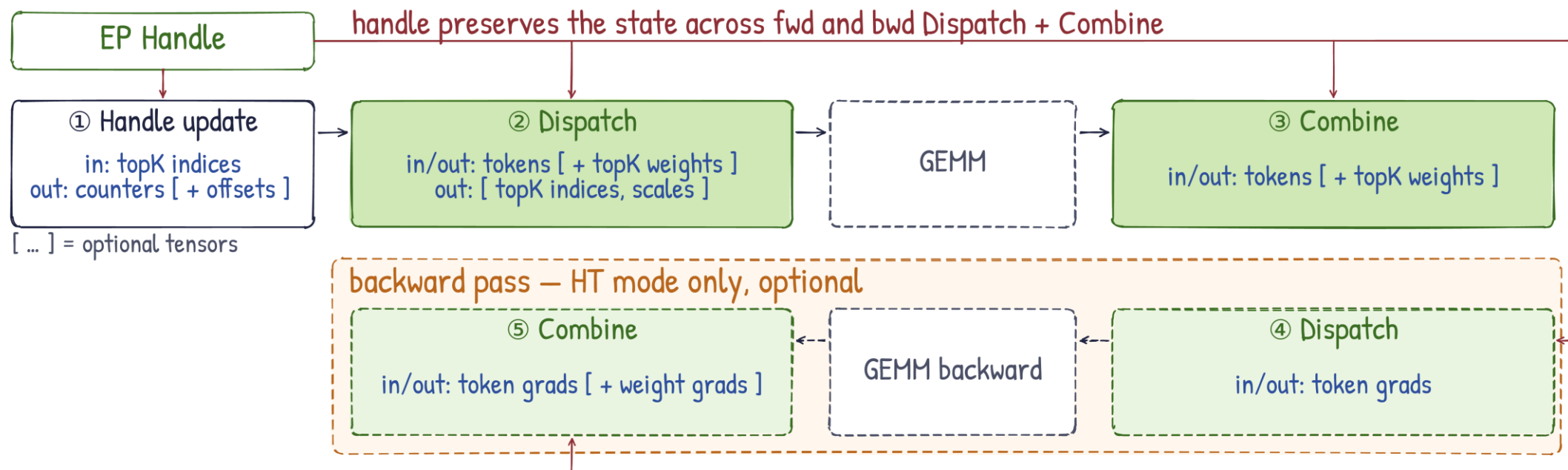
Three calls per step on one handle — the group picks the mode

LOW LATENCY (LL)

- Decode — small batches, every microsecond counts
- Direct communication (NVLink or RDMA)
- Layouts: expert-major, rank-major

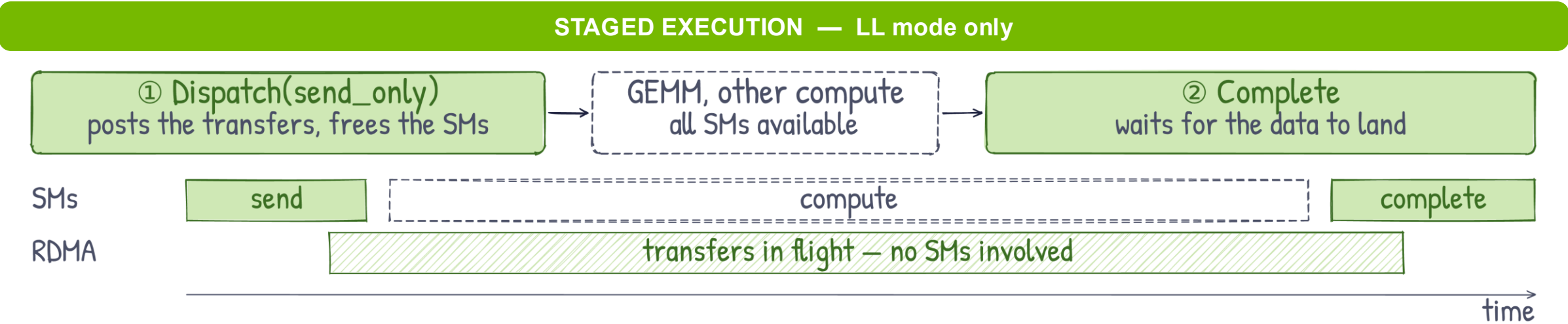
HIGH THROUGHPUT (HT)

- Training and prefill — large batches
- Hierarchical: NVLink inside the node, RDMA across nodes
- Layouts: expert-major, flat · forward and backward passes

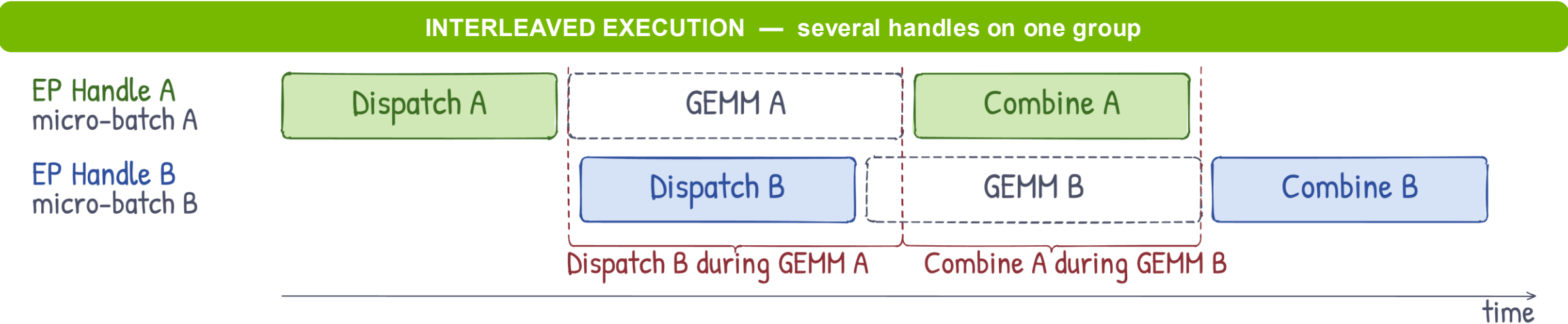


NCCL EP communication overlap options

Overlap communication with compute



RDMA moves the data, the GPU only posts and completes · Combine stages the same way



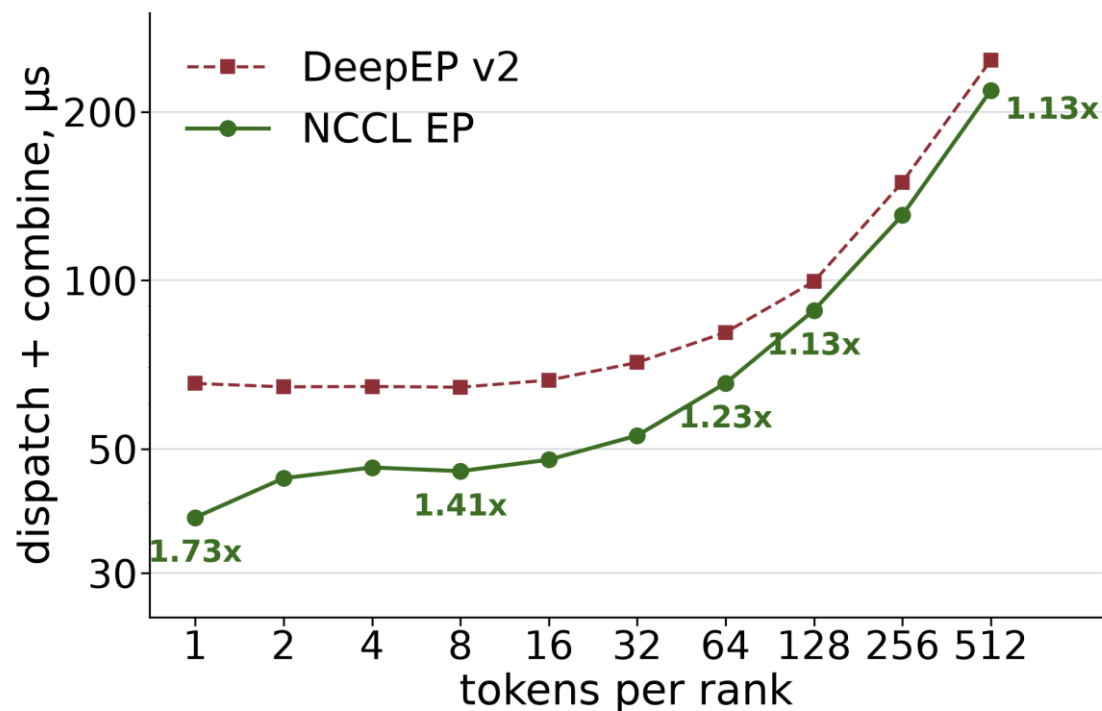
one EP Group, several live handles · updating one handle never disturbs another's pending ops

Decode performance (communication benchmarks)

GB200, one NVLink (MNNVL) domain, EP32 = 8 nodes · round trip time, μs

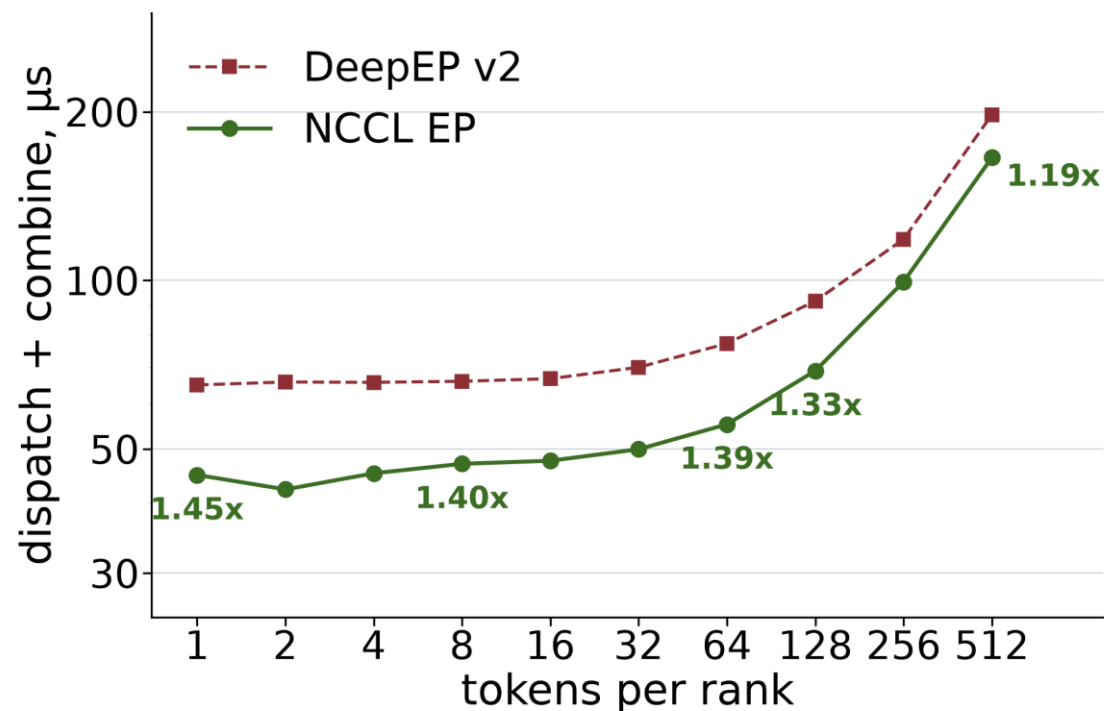
DeepSeek-V3 (DSv3) configuration

256 experts · top-k 8 · hidden 7168
EP32, BF16 · NCCL EP/LL vs DeepEP v2



Nemotron 3 configuration

512 experts · top-k 22 · hidden 2048
EP32, BF16 · NCCL EP/LL vs DeepEP v2



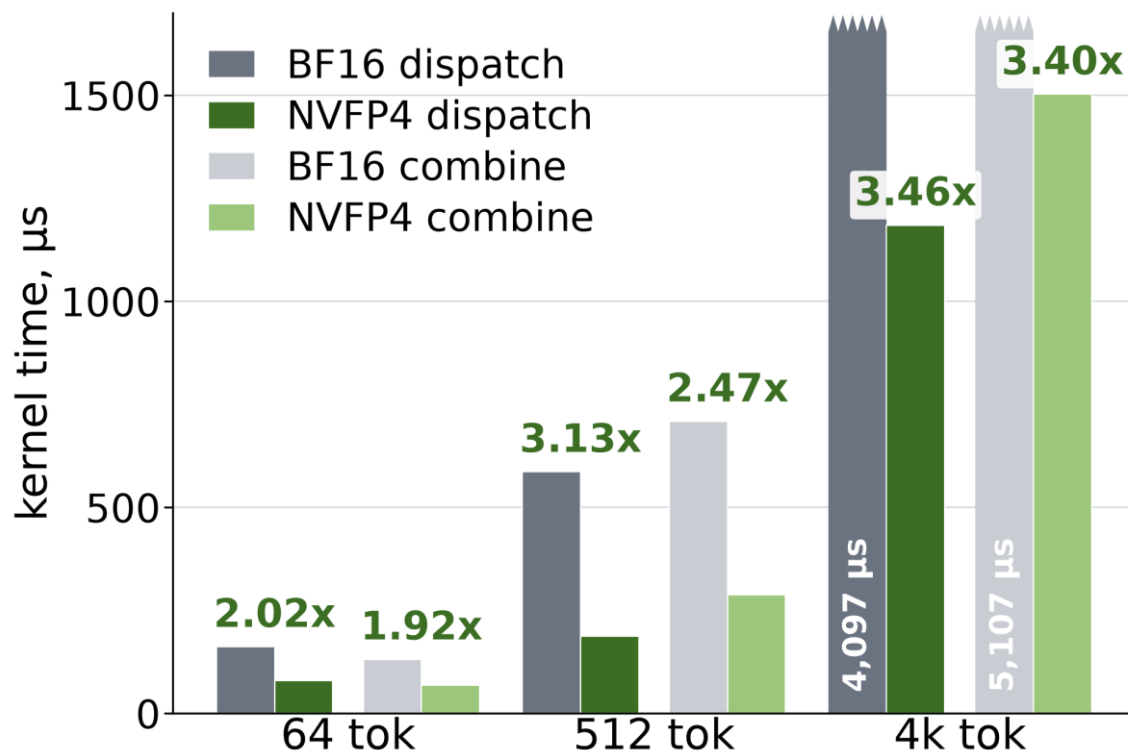
Decode with NVFP4 quantization

Fewer bytes on the wire

LL, rank-major — NVFP4 vs BF16

2 B200 nodes over RDMA (EP16)

64 / 512 / 4096 tokens/rank · kernel time



- For RDMA configurations, quantization provides substantial advantage
 - Dispatch forwards external scales
 - Combine quantizes internally
- Decode: more than 2x speedup
- Prefill: approaching the 4x SoL

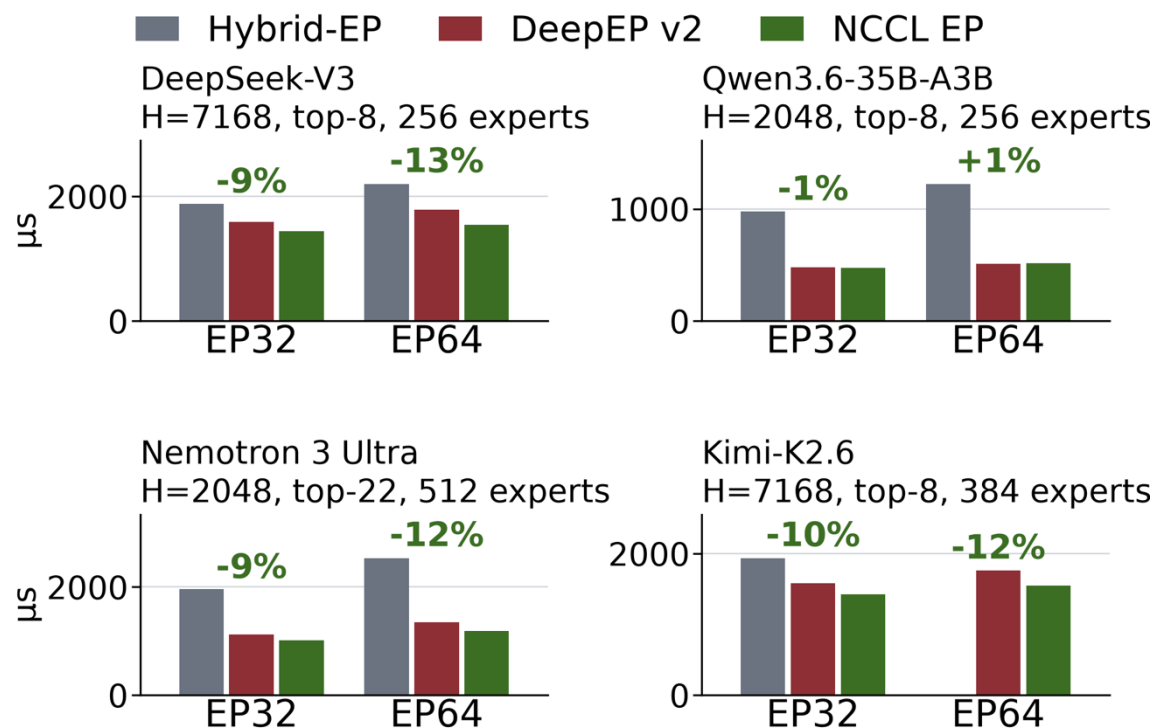
Training performance (BF16 and MXFP8 quantization)

Microbenchmarks, **MNNVL** (single NVL domain), **4k tokens/rank**, **32 SMs** (communication)

HT mode (Pull-Push), expert-major — BF16

GB300, 8 / 16 nodes

total communication time, all kernels

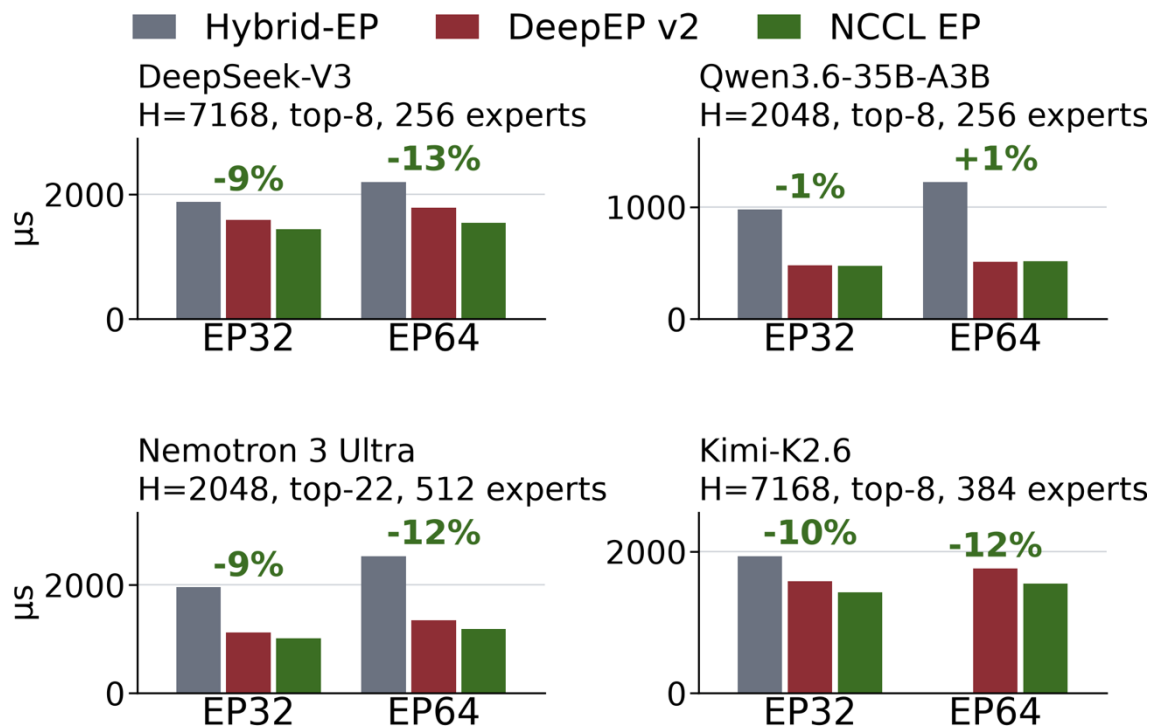


label = latency change, %: NCCL EP vs the faster baseline (left)

Training performance (BF16 and MXFP8 quantization)

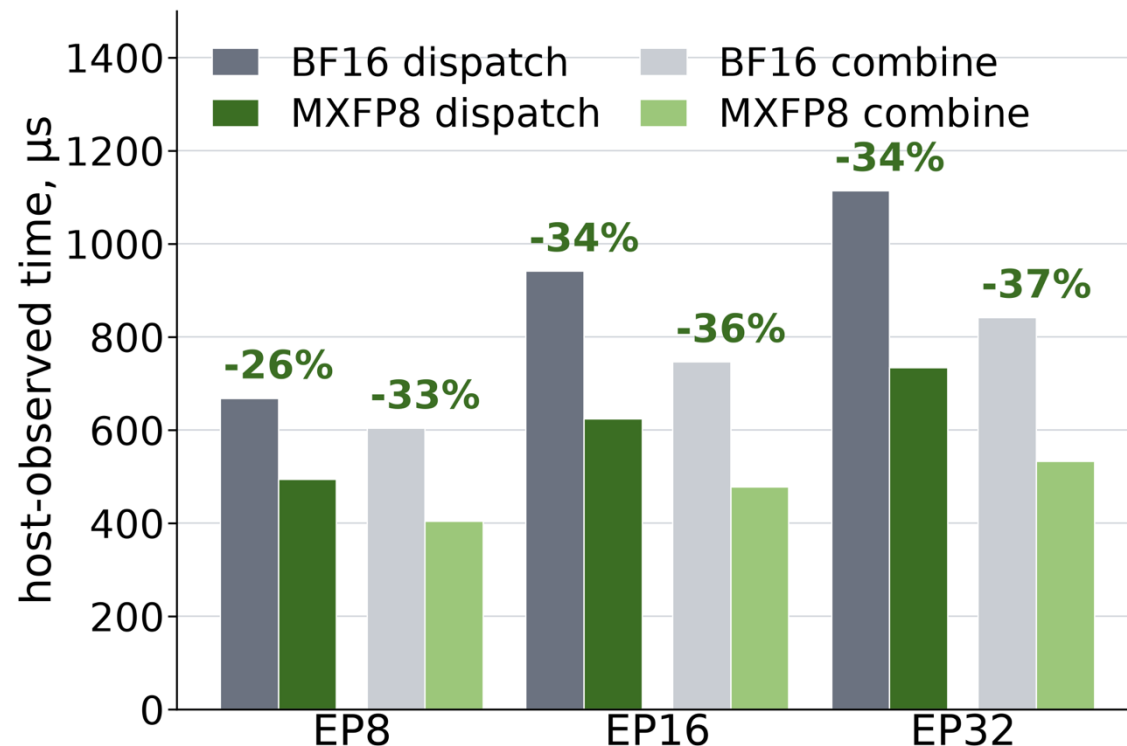
Microbenchmarks, **MNNVL** (single NVL domain), **4k tokens/rank**, **32 SMs** (communication)

HT mode (Pull-Push), expert-major — BF16
GB300, 8 / 16 nodes
total communication time, all kernels



label = latency change, %: NCCL EP vs the faster baseline (left)

HT, expert-major — MXFP8 vs BF16
DeepSeek-V3 · GB200
host-observed time

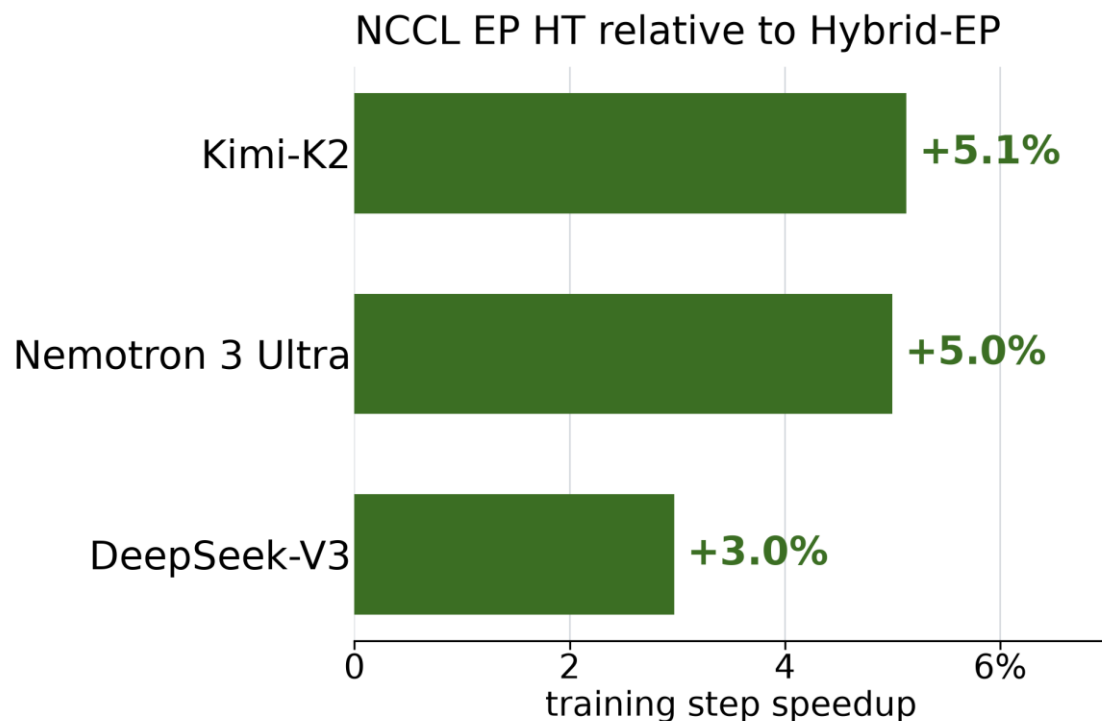


End-to-End training

End-to-end Megatron training

MEGATRON TRAINING — EP64

GB300, 256 GPUs · training step time vs Hybrid-EP



- Megatron integration via NVIDIA Transformer Engine (TE)
- Up to 5%+ improvement in the End-To-End training

PyTorch TokenSwitch: an MoE example

A PyTorch abstraction for NCCL EP dispatch and combine

```
from torch.distributed._token_switch import TokenSwitchNCCL

switch = TokenSwitchNCCL(
    process_group=pg, num_experts=E,
    max_dispatch_tokens_per_rank=B,
    max_recv_tokens_per_rank=M, max_token_bytes=H * 2,
)

routing = switch.create_routing(topk_idx)

recv, recv_w, recv_idx = switch.dispatch(
    routing, x, topk_weights, max_recv_tokens=M,
)

# Apply local experts, routing weights, etc
expert_out = local_experts(recv, recv_idx, recv_w)

out = switch.combine(routing, expert_out)

loss_fn(out).backward() # Gradients flow to x
```

One switch per group

NCCL EP handles the communication underneath.

Routing per forward

Share routing between dispatch and combine.

Autograd supported

Backward pass automatically ships gradients back to token owners.

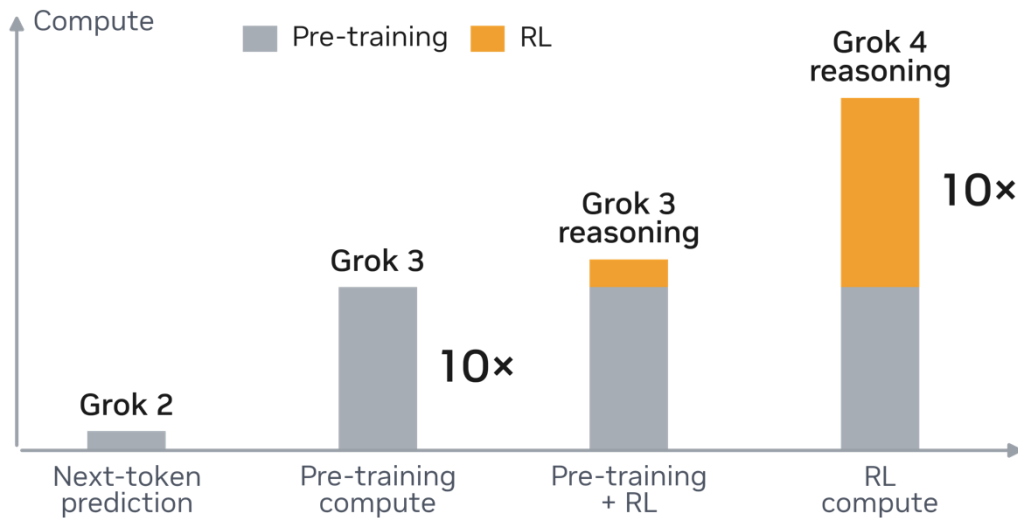


NCCL M2N

Why RL weight sync matters

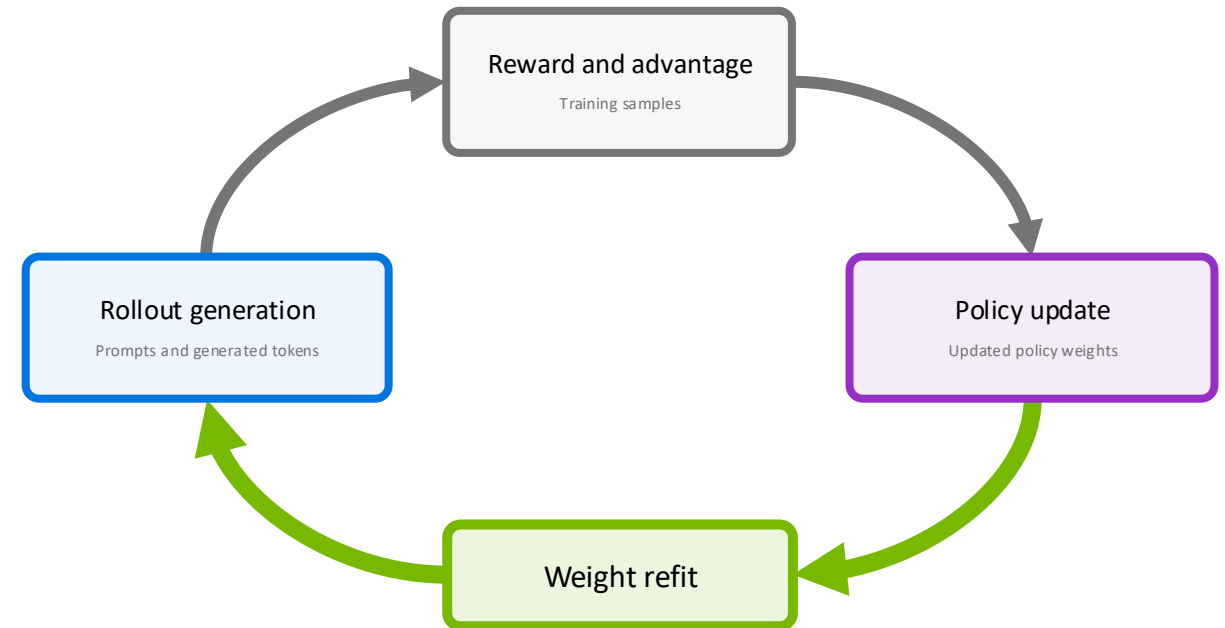
RL compute is approaching pretraining scale, and every policy update must reach the generators

Training compute across Grok versions



Source: xAI Grok 4 launch, Jul 2025. White: pretraining. Orange: RL. Relative scale only. OpenAI likewise describes o3 as a 10x training-compute scale-up over o1.

RL loop: weights cross from trainer to generator every step



Problems (1): Different Communication Requirements

Two endpoint layouts determine the peers and tensor regions to transfer

1. Two endpoint groups

NCCL Collectives use one rank group.

Weight sync has source and destination groups.

Weight sync: two groups



Problems (1): Different Communication Requirements

Two endpoint layouts determine the peers and tensor regions to transfer

1. Two endpoint groups

NCCL Collectives use one rank group.

Weight sync has source and destination groups.

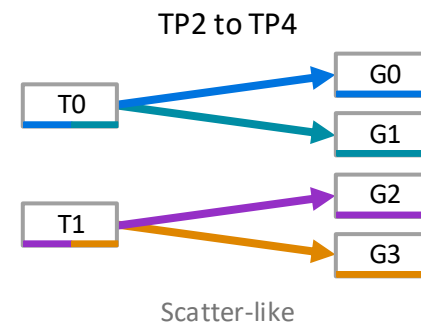
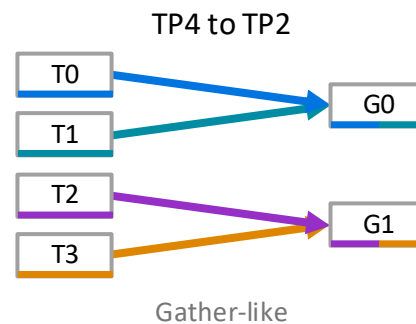
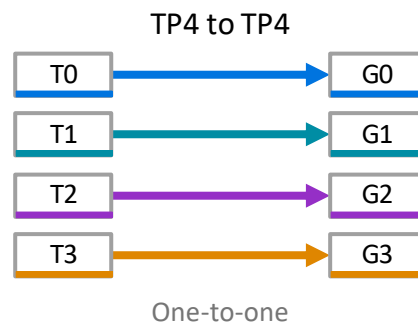
Weight sync: two groups



2. Parallelism degree affects comm pattern

Trainers and generators can have varying degrees of data and model parallelism

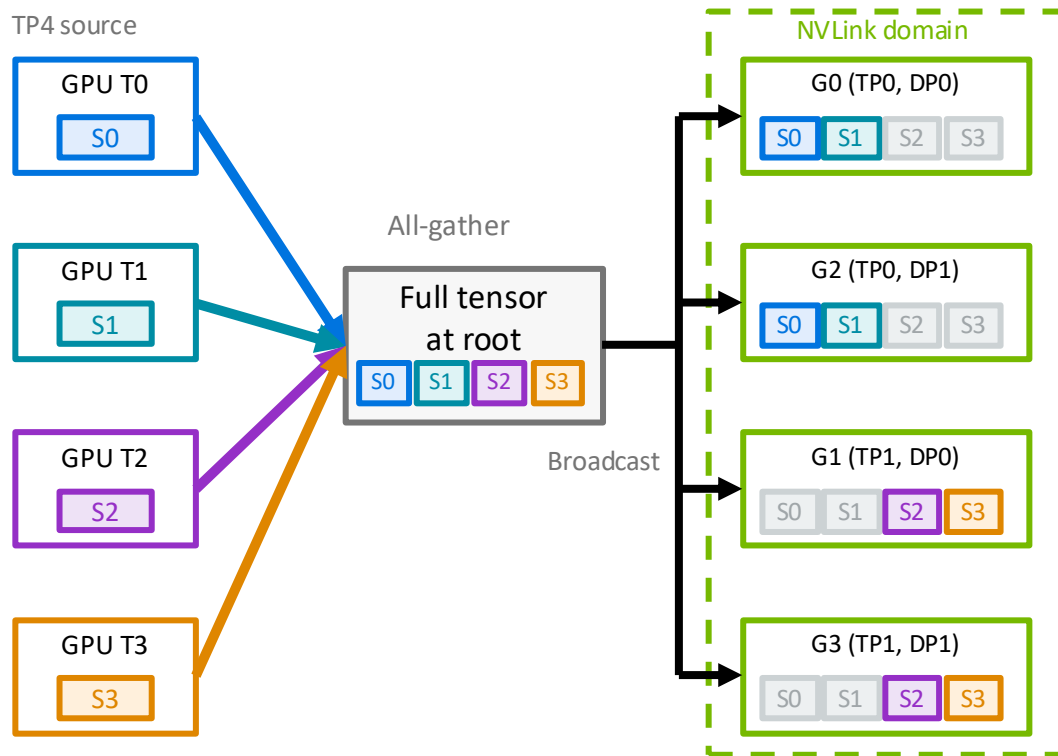
Different shard counts change the data transfer graph



Problems (2) : Redundant network transfers

Example: source TP4, destination TP2 × DP2 in one NVLink domain

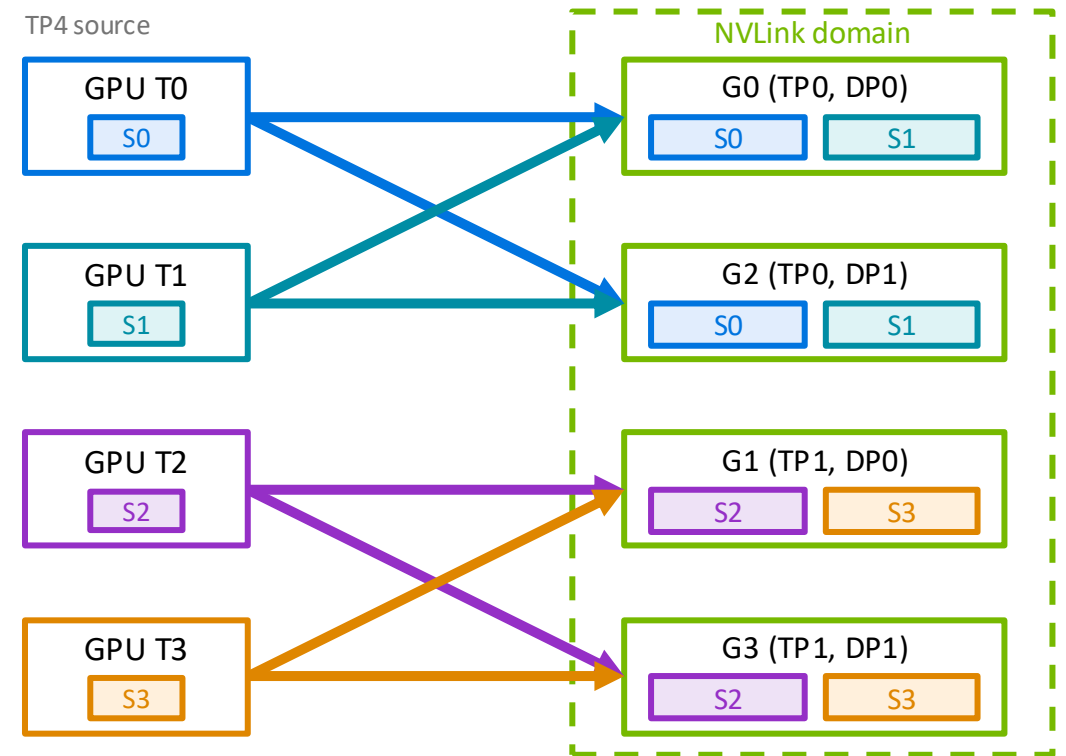
All-gather plus broadcast



Full tensor delivered, local slice retained

Gray shards are discarded. Broadcast arrows show logical delivery; physical NCCL routing depends on topology.

Direct Send Recv



8 network transfers for 4 unique shards

NCCL M2N Reshard API

Describe both layouts; M2N plans the transfer

```
import nccl.m2n as m2n

m2n.reshard(
    src,                # local source shard, or None
    dst,                # local destination buffer, or None
    comm,               # one NCCL comm over both groups
    stream,
    src_mesh=..., src_placements=...,
    dst_mesh=..., dst_placements=...)

# Group a whole refit into one submission
with m2n.group():
    for src, dst in params:
        m2n.reshard(src, dst, comm, stream, ...)
```

Layouts, not peer lists

A mesh plus Shard / Replicate per axis: the same model as PyTorch DTensor.

Same call on every rank

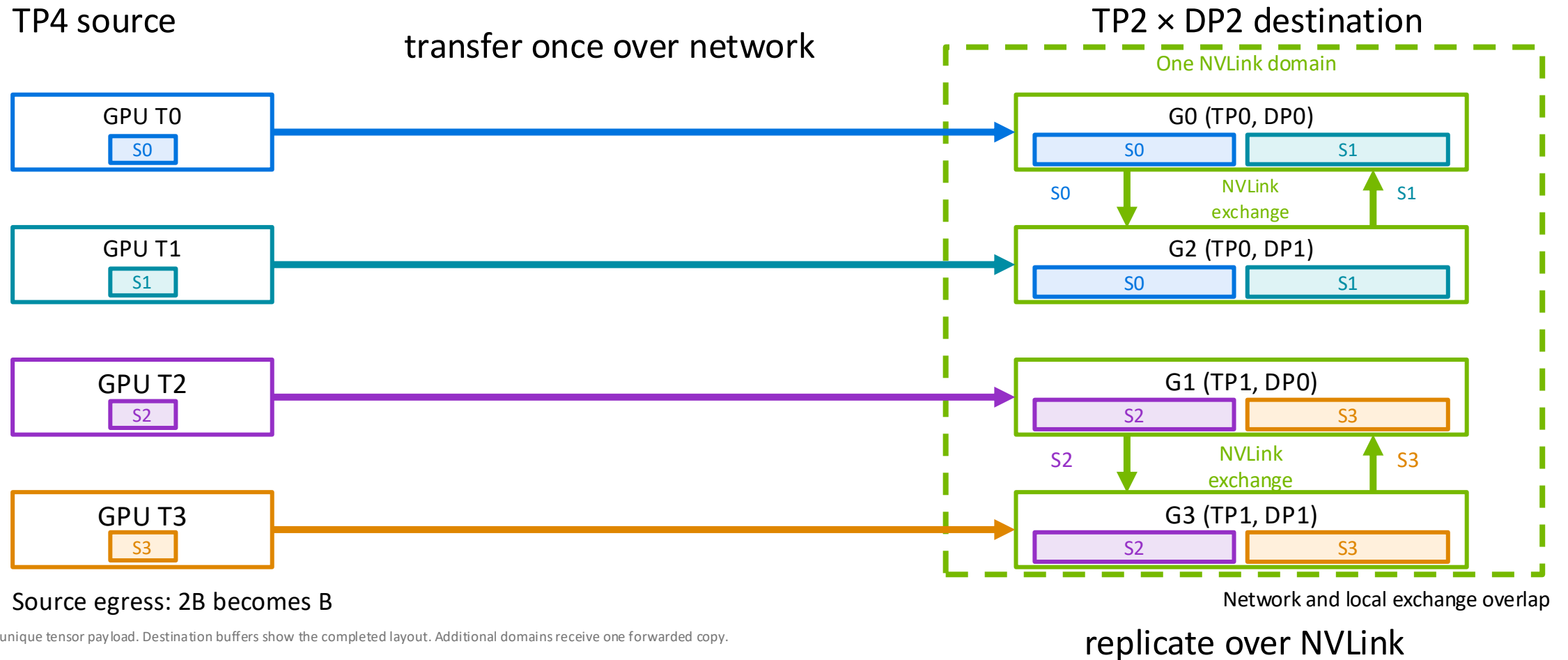
Ranks that don't own a side pass None. M2N derives peers, tensor regions and the topology-aware schedule.

Grouped submission

`m2n.group()` submits all reshards of a refit together, with one abort and error boundary. `ncclM2nGroupStart` / `End` in C.

Benefits of Reshard primitive (1): Topology-aware data movement

TP4 to TP2 × DP2: one network copy per shard, local NVLink exchange



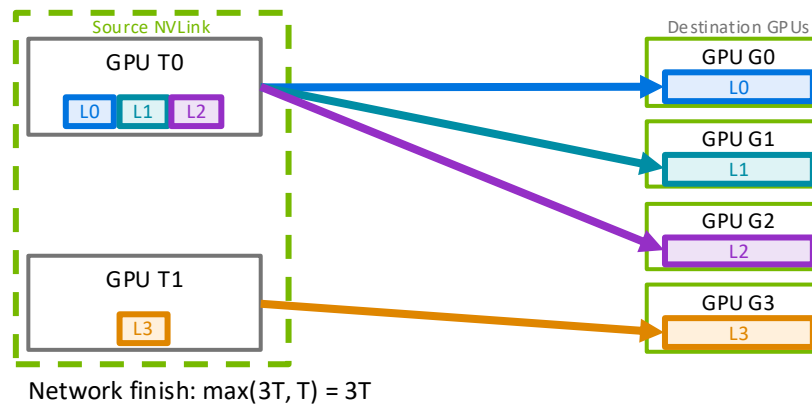
Benefits of Reshard primitive (2): Load balancing

Redistribute layer traffic over NVLink to balance source network paths

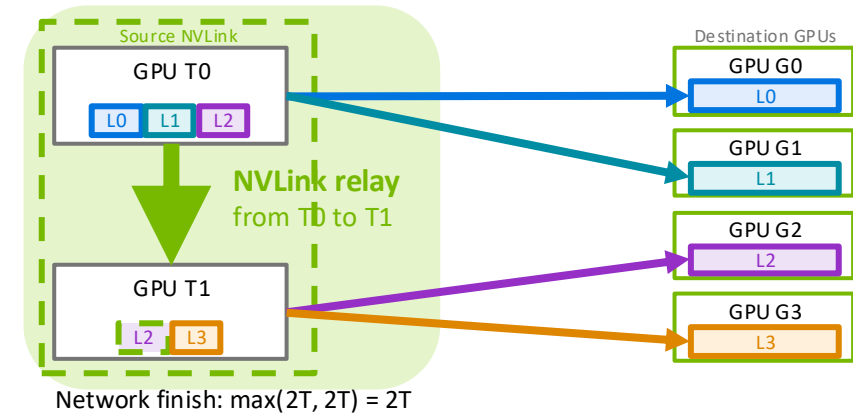
Application imbalance

3 layers vs 1
Equal NIC bandwidth

Direct Send Recv



Reshard: source redistribution

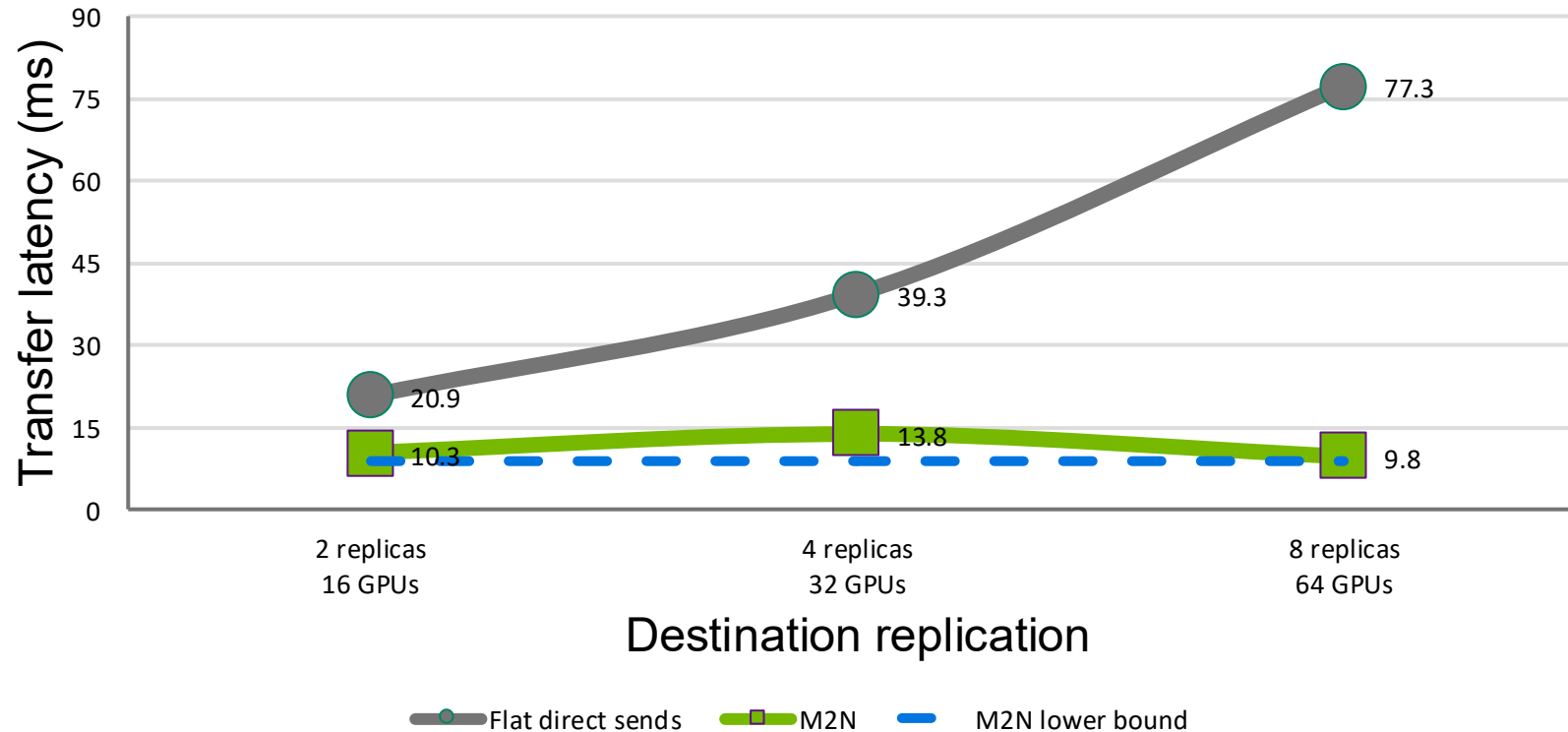


- Application imbalance: uneven layer ownership (e.g. pipeline stages) overloads some NICs. M2N relays layers over NVLink so every NIC sends an equal share.
- Network imbalance and faults: a degraded link, a lost rail or a failed NIC is handled the same way, by routing traffic over NVLink to the healthy NICs.

Illustrative model. T = time to send one layer at full NIC bandwidth; NVLink relay cost omitted.

Evaluation (1) : Single-layer benchmark results

GB200 NVL72 + NDR, EP16 trainer, TP8 generator, BF16 expert gate projection



7.9×
at 8 replicas

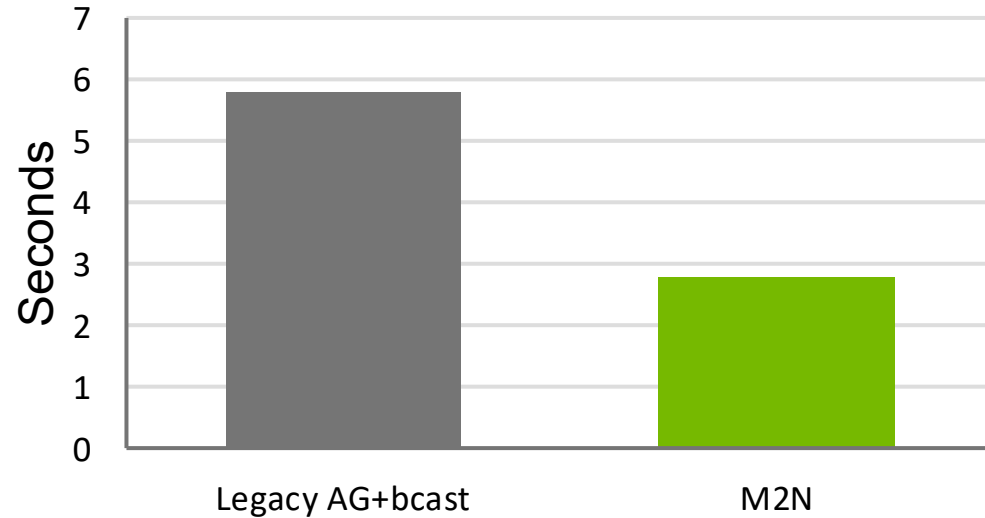
77.3 ms to 9.8 ms
8.96 ms lower bound

Each point uses the lowest validated M2N result. Network lower bound excludes preparation and startup.

DeepSeek-V3 NeMo-RL results

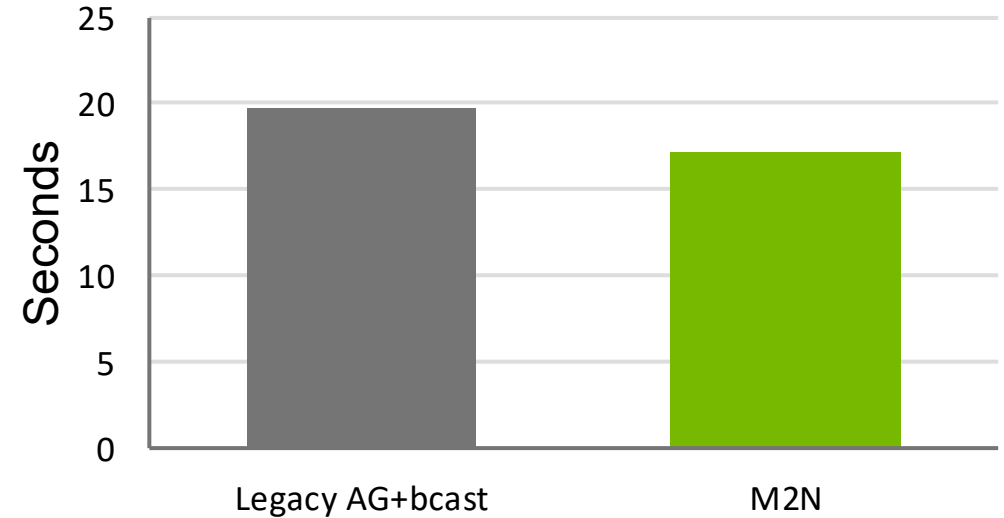
256 GB200 GPUs, disjoint 128-GPU trainer and generator groups

Weight synchronization



2.09× faster

Reported RL step time



12.7% lower

Reported non-warmup timings: baseline median and one measured M2N PACK step. Sequence length 1536; one-step off-policy asynchronous RL.

Release and integration

NCCL M2N is available in the public NCCL Extensions repository

M2N v0.2

Released with NCCL Extensions v0.1.0

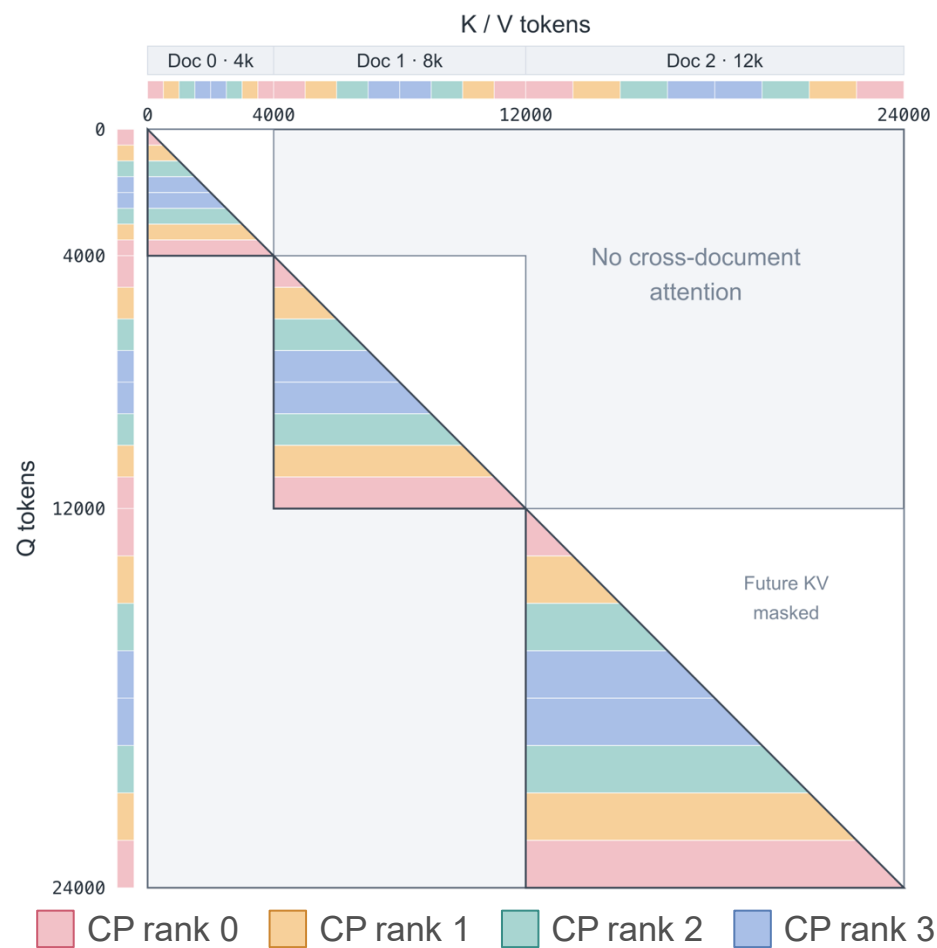
github.com/NVIDIA/nccl-extensions/tree/main/nccl_m2n

Framework	Status	Integration effort
NeMo-RL	Merged	Integration done by Youngeun from E2E Training Performance team
vLLM	Merged	Integration efforts led by Pouya Kousha, Ke Wen from GPU Comms Teams
veRL	PR open	FSDP and Megatron export with destination-local weights
Miles	PR open	Integration by Nicolas Castet
ModelExpress	PR open	Integration by Pouya Kousha and Nicolas Noble

Future work: NCCL Context Parallelism (CP)

Planned `nccl_cp` module, in collaboration with MiniMax

Packed documents with causal masks



Attention defines the traffic

Masks and document boundaries change which KV ranges each rank needs.

Sparse selection adds variability

Selected context can change with the input and attention pattern.

The challenge: communicate only useful context

We're hiring at NVIDIA

Deep Learning Framework Communications Engineer

Make multi-GPU AI faster

PyTorch, JAX, vLLM, SGLang

NCCL, NCCL Extensions

Python, C++, CUDA, cuteDSL

Distributed training and inference

Compute–communication overlap

Performance profiling



Scan to apply

[JR2011908](#)