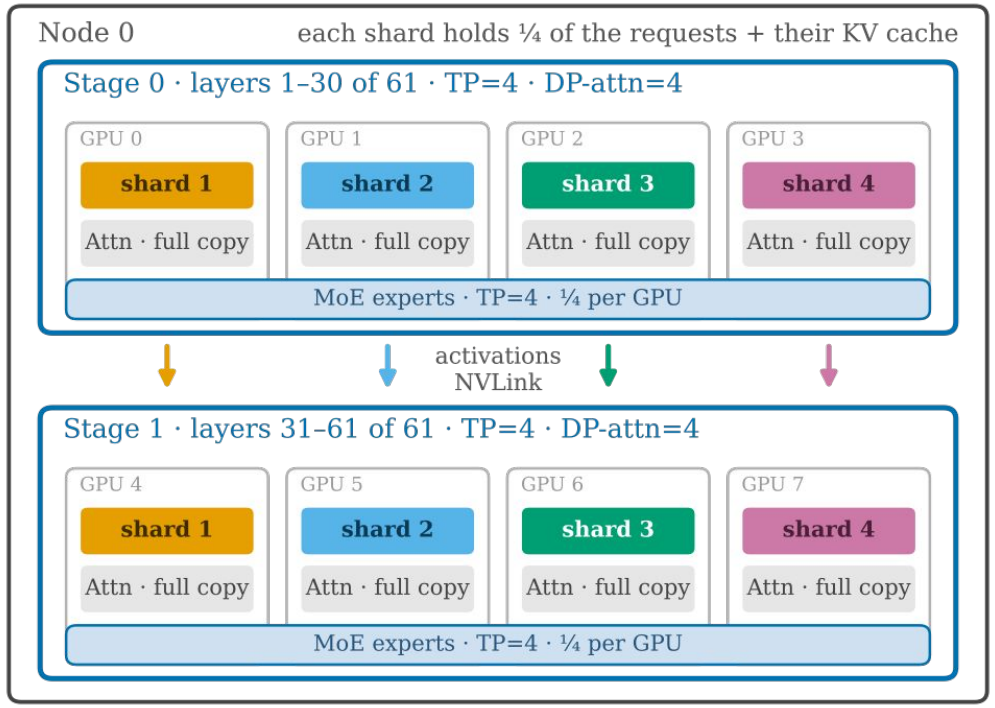
BID 2026 workshop at ICPP, Singapore, 28 September 2026

Less is More

Optimising SGLang distributed DeepSeek-R1 inference on a two-node H200 cluster



Luca Lowndes and Nathan Culshaw

Monash University, Melbourne

We had 16 H200s. Our fastest layout used 8.

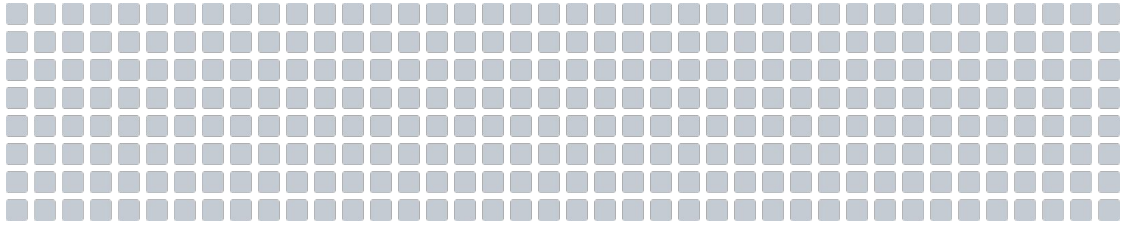
17,342 tok/s on one node, 76% faster than the 9,852 tok/s baseline that used all 16.

All mean of three runs.

Other Deployment Playbooks use far more GPUs than we had

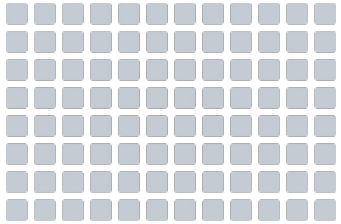
DeepSeek-V3 decode unit

40 nodes, 320 GPUs



SGLang large-scale EP recipe

12 nodes, 96 H100s



This work

2 nodes, 16 H200s

Our question: how many tokens per second can these 16 GPUs produce?

Each square is one GPU; each column of eight is one node.

Sources: DeepSeek-V3 Technical Report (2024); LMSYS SGLang blog (May 2025)

The fixed benchmark

Fixed		Ours to change
Model	DeepSeek-R1: 671B parameters, 37B active	Parallelism layout
Engine	SGLang 0.5.3rc1	DP-attention
Workload	2,000 ShareGPT V3 prompts, offline	Scheduler memory settings
Precision	dummy weights, seed 2025	CUDA graph batch limit
Metric	Total token throughput (tok/s)	NCCL settings
Baseline	TP=16 on both nodes: 9,852 tok/s (3 runs)	Container and software stack
		torch.compile

Benchmark was fixed as a part the 2025 APAC HPC-AI Competition, where this work originated from.
Baseline: 9,850 tokens per second (average over three runs)

NVLink is about 10x faster than InfiniBand per flow

NVLink, inside a node

Per GPU: 18 links at 26.6 GB/s each, raw, one direction

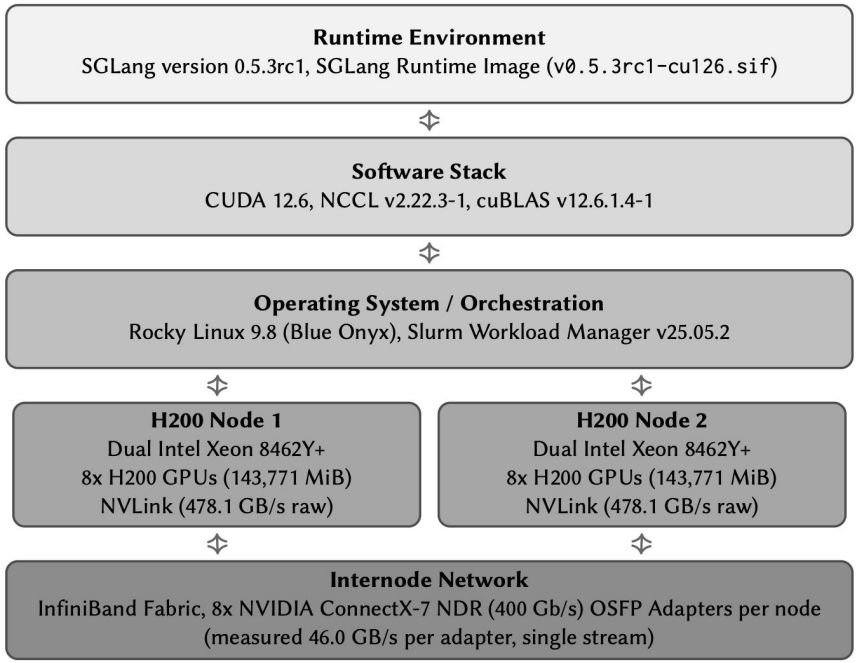
478 GB/s

InfiniBand, between nodes

Measured with **ib_write_bw**, one stream; each node has 8 ConnectX-7 NDR 50 GB/s adapters

46 GB/s

Transmitting 16MB over NVLink took 74 µs, transmitting the same data over InfiniBand took 421 µs.

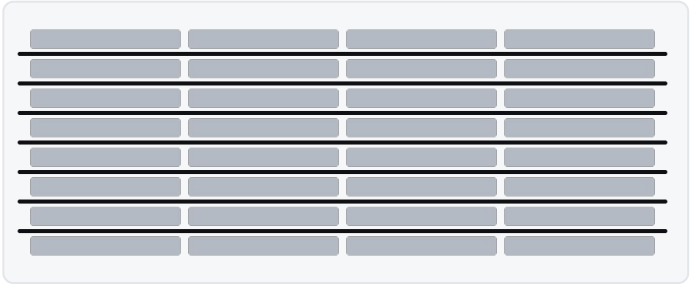


2 × Dell XE9680-F (immersion cooled), 8 × H200 (141 GB) each, Sustainable Metal Cloud, Singapore

Each way of splitting the model has a different cost

Tensor parallelism, TP

Splits every layer across the GPUs



Traffic

Two all-reduces per layer: 122 per forward pass

Cost

Communication sits on the critical path

Pipeline parallelism, PP

Gives each GPU a block of layers

Traffic

Activations at each stage boundary, once per micro-batch

Cost

Stages idle if the pipeline runs dry

Data parallelism, DP

Runs full copies and splits the requests

Traffic

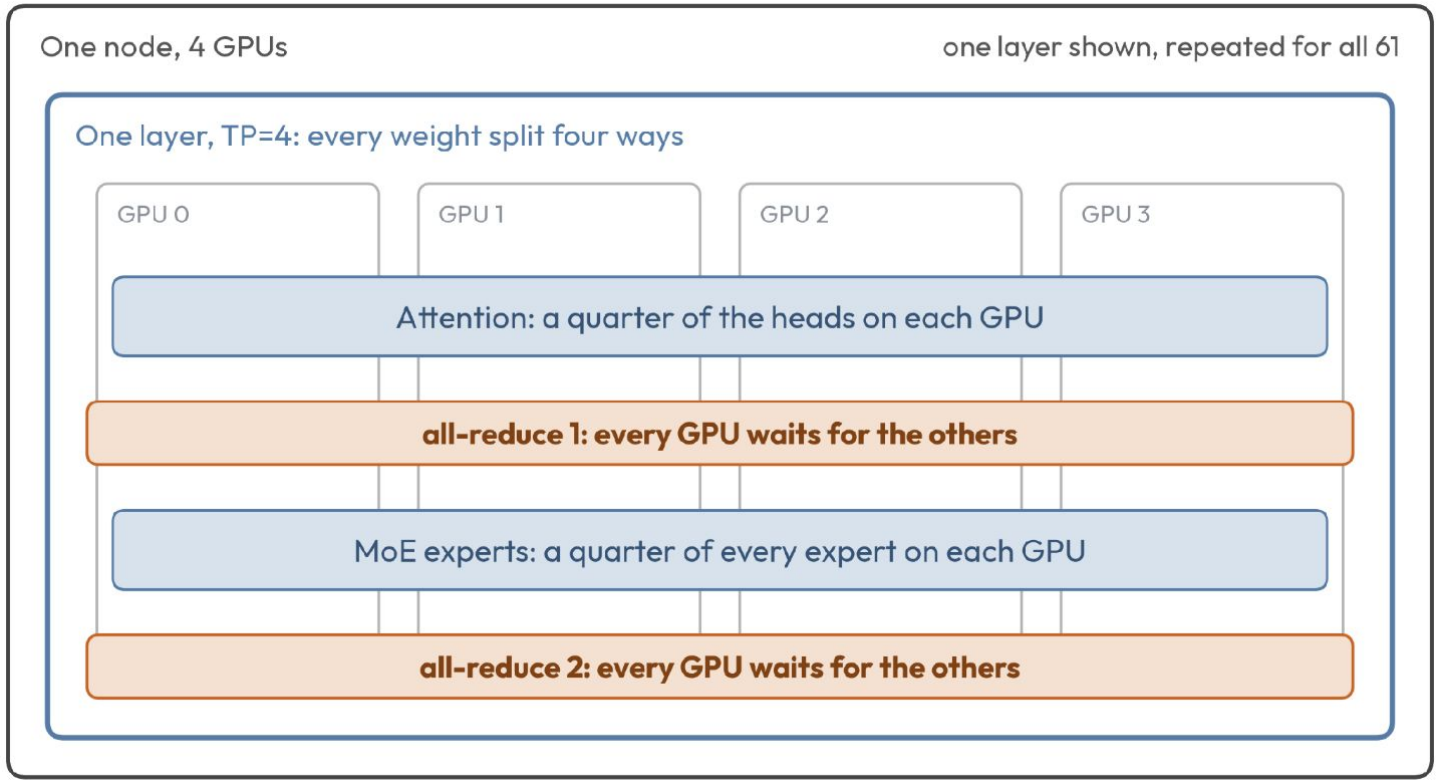
None during inference

Cost

Every copy needs the full weights

In the diagrams each column is a GPU, each row a layer, and dark lines represent when an all-reduce has to happen.

Tensor parallelism splits every layer and syncs twice per layer



Splits

Every weight matrix, a quarter per GPU at TP=4

Traffic

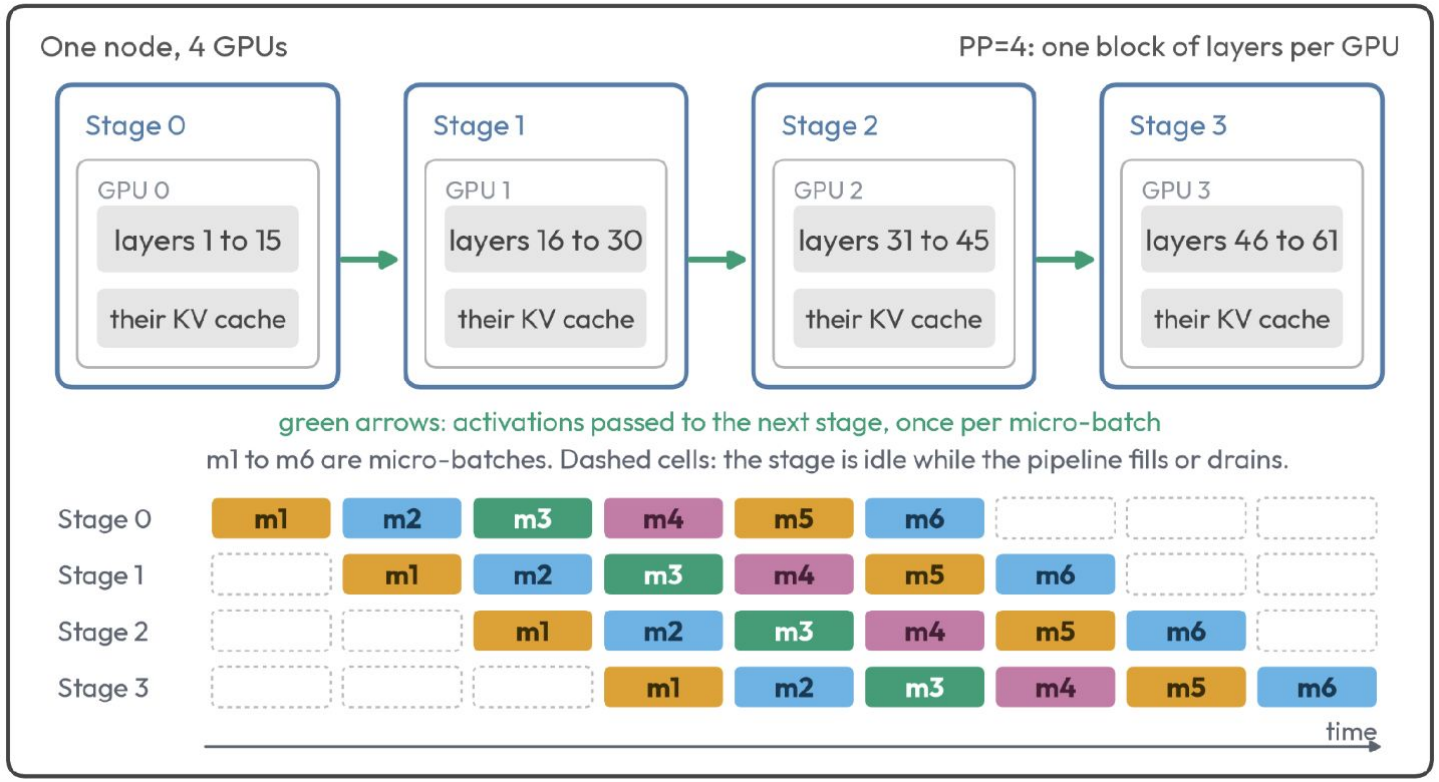
Two all-reduces per layer: 122 per forward pass

Cost

Communication sits on the critical path

Shown for 4 GPUs. DeepSeek-R1 has 61 layers, each with an attention block and a mixture-of-experts (MoE) block.

Pipeline parallelism gives each GPU a block of layers



Splits

The layers, in consecutive blocks

Traffic

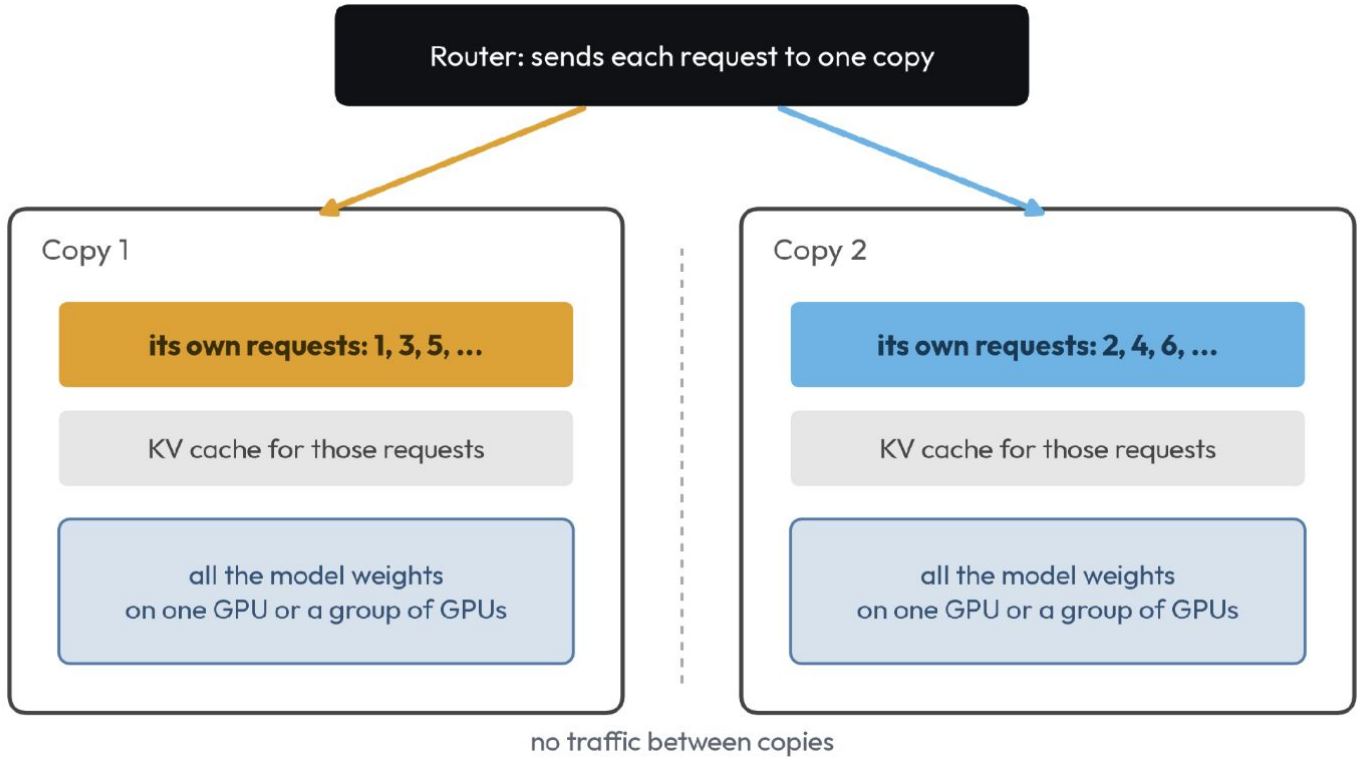
Activations at each stage boundary, once per micro-batch

Cost

Stages sit idle while the pipeline fills or runs dry

Shown for 4 stages. Each stage keeps the KV cache for its own layers, so PP splits the cache instead of copying it.

Data parallelism runs full copies and splits the requests



Splits

The requests. Each copy runs the full model

Traffic

None between copies during inference

Cost

Every copy needs the full weights in memory

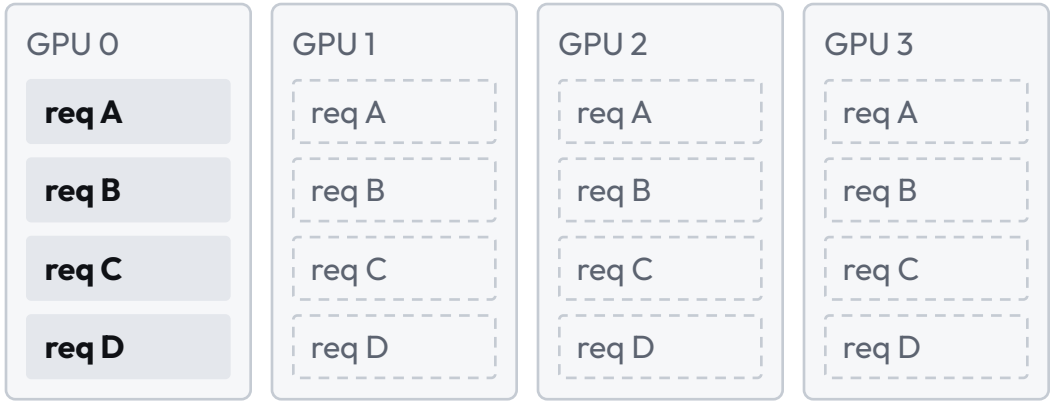
A fourth option for MoE models, expert parallelism, spreads the experts across GPUs. It pays off at much larger scale than ours.

With MLA, tensor parallelism copies the KV cache

MLA compresses the keys and values of every head into one latent vector per token. With a single KV head there is nothing for TP to split, so every TP rank keeps a full copy of the cache.

Tensor-parallel attention, TP=4

SGLang's default



4 requests fit. GPUs 1 to 3 hold copies (dashed).

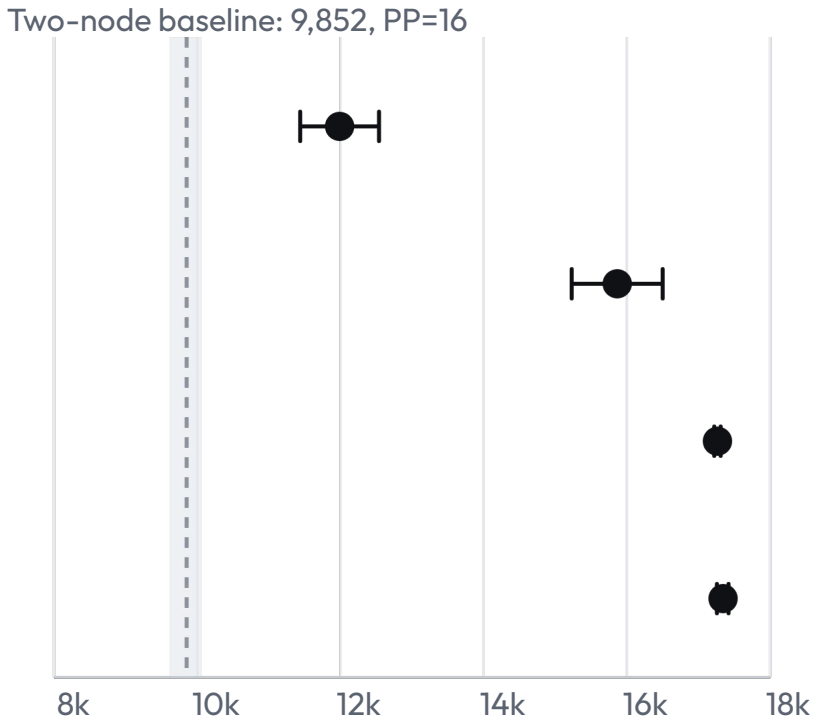
DP Attention: LMSYS Blog (DP Attention for DeepSeek Models)

DP-attention, DP=4

--enable-dp-attention

16 requests fit in the same memory.

On one node, DP-attention was the biggest lever



TP=4, DP=1, PP=2 no DP-attention	11,987 11,436 to 12,537
TP=2, DP=2, PP=4 DP-attention	15,864 15,229 to 16,499
TP=4, DP=4, PP=2 DP-attention	**17,263** 17,218 to 17,308
TP=4, DP=4, PP=2 DP-attention, torch.compile	**17,342** 17,255 to 17,417

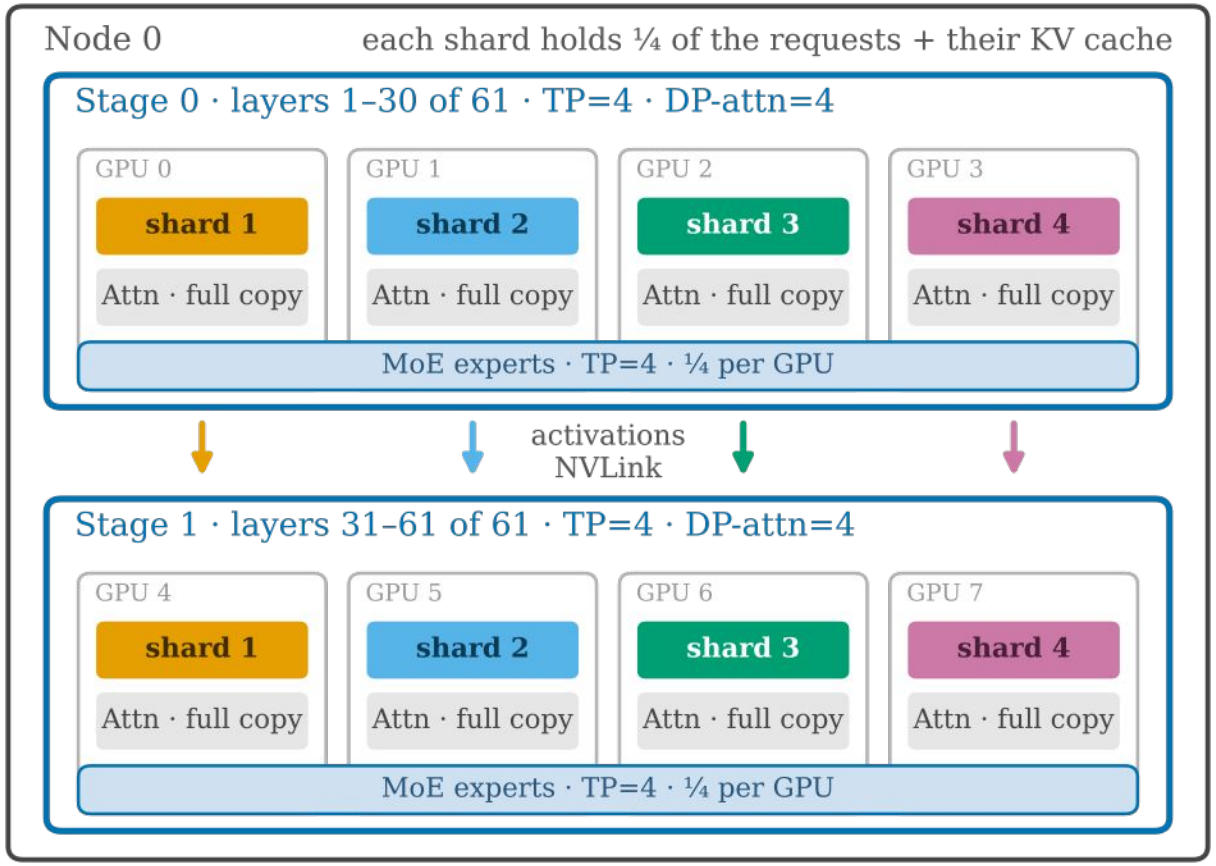
+44%

from DP-attention alone, with TP=4 and PP=2 fixed

Mean of two runs each:
11,987 to 17,263 tok/s

Throughput in tok/s. Dots are mean of two runs.

The fastest layout keeps all GPU traffic on NVLink



Attention: DP=4

Each GPU serves a quarter of the requests and keeps only their KV cache.

Experts: TP=4

Within a stage, the four GPUs split every expert layer.

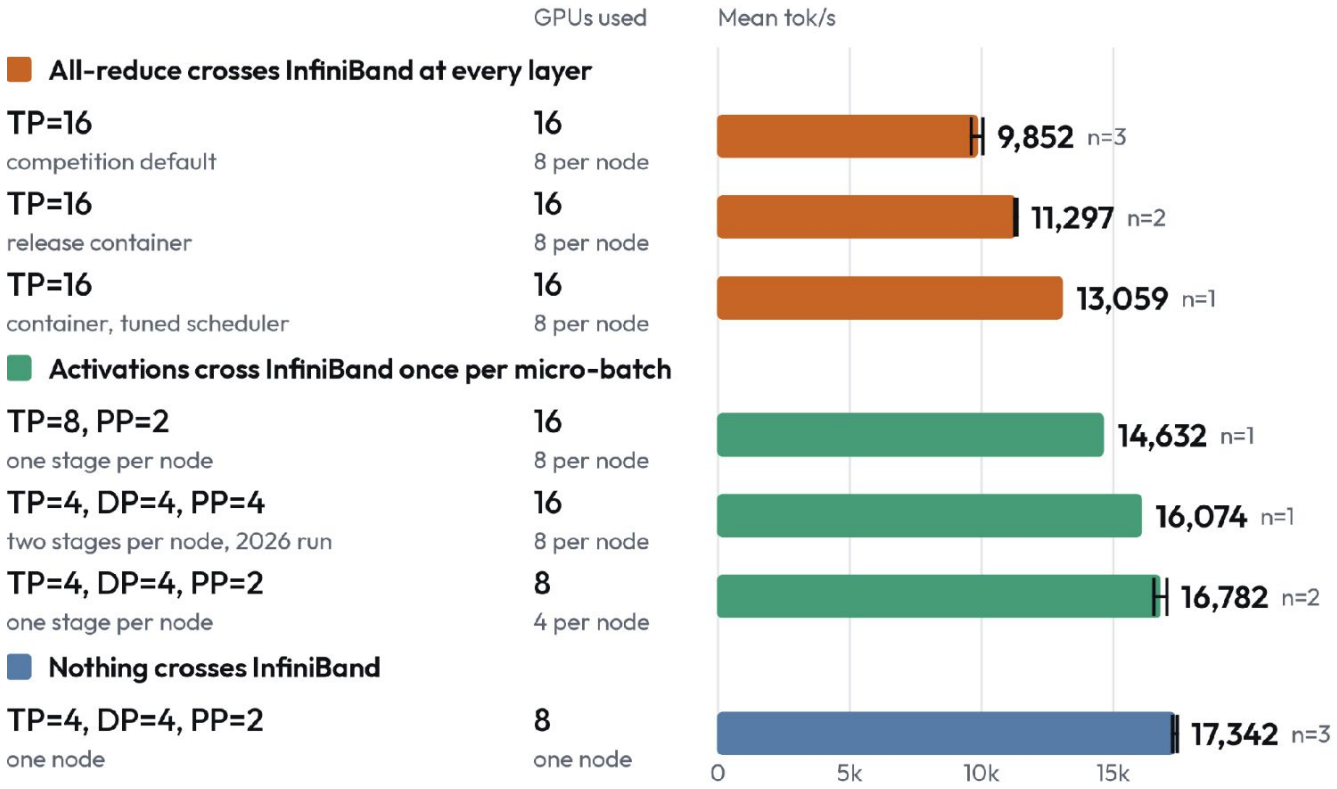
Stages: PP=2

Two blocks of layers, with only activations moving from one to the next.

17,342 tok/s

mean of three runs, with torch.compile

Across two nodes, the fastest engine used 8 GPUs



2.07×

throughput per allocated GPU, one node against the best two-node engine

Mean of 17,342 tok/s on 8 GPUs, against a mean of 16,782 tok/s on a 16-GPU allocation

Squares show the GPUs each engine used; n is the number of runs, and ticks span repeated runs. Both TP=4, DP=4, PP=2 bars used torch.compile. All values shown as mean of runs.

Tuning NCCL did not close the gap

0 of 40

NCCL configurations beat the library's autotuned defaults on two-node TP=16

The best reached 9,561 tok/s,
below all three default runs (9,611 to 10,054).

What we swept

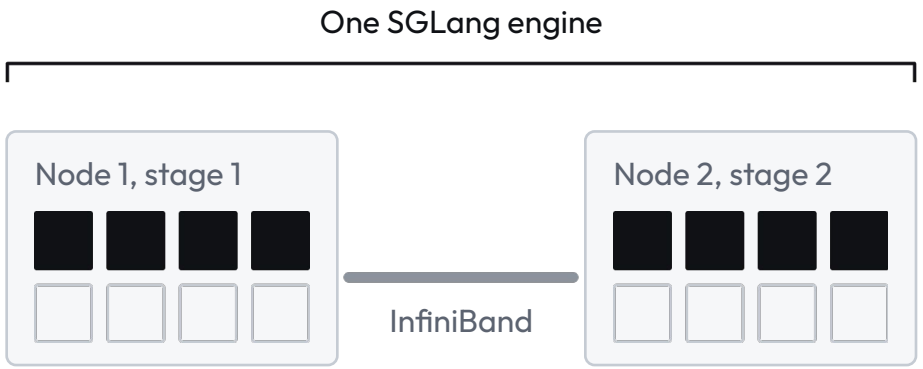
NCCL_PROTO	LL, LL128, Simple
NCCL_NTHREADS	64 to 256
NCCL_MIN_NCHANNELS	4 to 16

Also: GPUDirect RDMA level, PCI relaxed ordering, socket threads

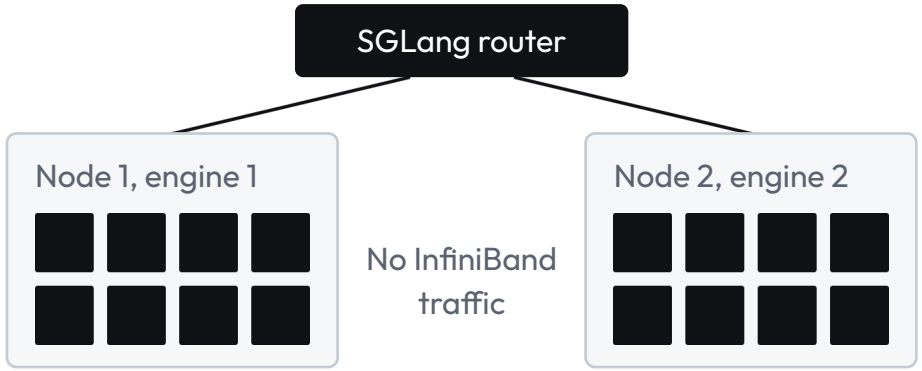
What helped was moving the node boundary:
TP=8, PP=2 ran 30% faster than TP=16 on the same 16 GPUs.

Across nodes, replicate instead of distributing

Distribute: one engine across both nodes



Replicate: one engine per node, behind a router



16,782 tok/s

mean of two runs (16,532 to 17,032), 8 of 16 GPUs busy

23,118 tok/s

projected, not yet measured

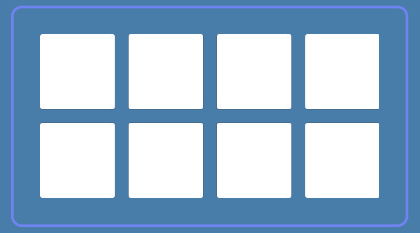
Projected as 2 × 11,559 tok/s, the mean of two runs of one node on half the prompts (11,231 and 11,887). That is 38% above the best distributed engine.

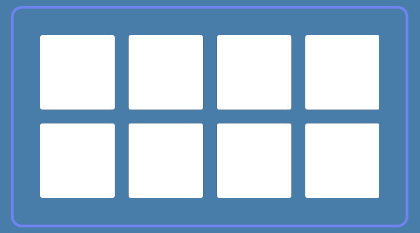
Several results still need direct tests

Limitation	What would test it
The replica result is projected from half-workload runs	Run both replicas at once behind the SGLang router
One to three runs per configuration	More repeats, reported as mean and standard deviation
Dummy weights, so expert routing was not realistic	Re-run the key layouts with the real R1 weights
BF16 on one SGLang release	FP8, newer SGLang, and other engines such as vLLM
Two nodes on one interconnect class	More nodes, NVL72-class racks, and wide EP at scale

Keep collectives on NVLink

1. Fill one node before spanning two.
2. Across nodes, run independent replicas behind a router.
3. If one engine must span nodes, split it with PP, not TP.





Measured on DeepSeek-R1 with SGLang on two H200 nodes over NDR InfiniBand. At larger scale, wide expert parallelism and prefill-decode disaggregation take over.

Thank you

Thanks to Simon Michnowicz (Monash eResearch), Firmus for compute access, and Pengzhi Zhu for running the competition.

luca@lowndes.net

github.com/LucaLow/APAC-AI-HPC-2025

Scheduler tuning only helped TP=16

`--mem-fraction-static`

Share of GPU memory for weights plus KV cache. Sets the ceiling on concurrent requests.

`--cuda-graph-max-bs`

Largest decode batch captured as a CUDA graph. Bigger batches fall back to eager execution.

On TP=4, DP=4, PP=2, any setting above the defaults ran out of memory. Each GPU holds twice the weight share of TP=16, and the KV cache already takes the rest.

13,059

tok/s peak on TP=16, at 0.92 and batch 724: 16% above the defaults

Out of memory at 0.93 with batch 724, and at 0.94 for every batch size. One run per setting; the defaults are a two-run mean.

The release container beat our hand-built environment

Hand-built virtual environment

11,559

tok/s, TP=8 on one node, one run. Needed a lower memory fraction (0.87) and expandable segments to avoid out-of-memory crashes.

SGLang release image

12,449

tok/s, same layout, one run: 7.7% faster, with no manual memory flags.

Image v0.5.3rc1-cu126: CUDA 12.6, NCCL 2.22.3, cuBLAS 12.6.1.4, run with Singularity under Slurm. Not yet separated: the container itself versus the pinned versions inside it.