NeMo AutoModel Guest Lecture – Sept. 2026

Huiying Li, Zhiyu Li and Alexandros Koumparoulis

Agenda

Part 1 - Introduction

Part 2 - Deepseek v4 Case Study

Agenda

Part 1 - NeMo Automodel Introduction

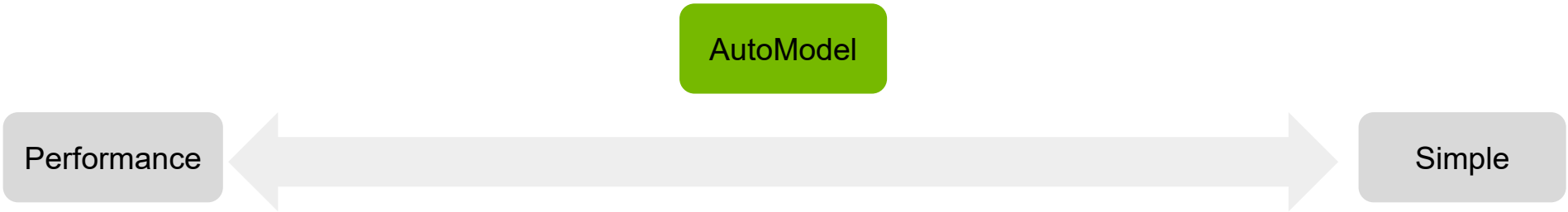
- Overview
  - What is Automodel
  - Why do we need Automodel
  - API & Checkpoints
- Two paths in Automodel - OOTB & Custom Layer acceleration

What is NeMo Automodel?

OSS PyTorch based training framework within NeMo designed to bridge the gap between ease of use and peak performance for large-scale LLM/VLM training.

Available on Github and NGC

NeMo Automodel - Balancing between Ease of Use and Speed



- **Complex to use.**
- **Limited model coverage**
- **Optimal speed**

- **Easy to use**
- **Broad model coverage**
- **Very good speed**

- **Easy to use.**
- **Broad model coverage**
- **Lacks Speed**

NeMo AutoModel Functionality Overview

Supported Tasks & Project structure

App Layer / Recipes

- LLM finetune-SFT/LoRA/Qlora Single/Multi-turn
- VLM finetune-SFT/LoRA
- LLM Knowledge Distillation
- LLM SeqCLS (sentiment classification)
- LLM pretrain
- LLM QAT

Component Layer

- Transformers
- Distributed
- Checkpointing
- Datasets
- Launcher
- Loggers
- Quantization
- PEFT
- MoE
- Loss
- Config
- Utils

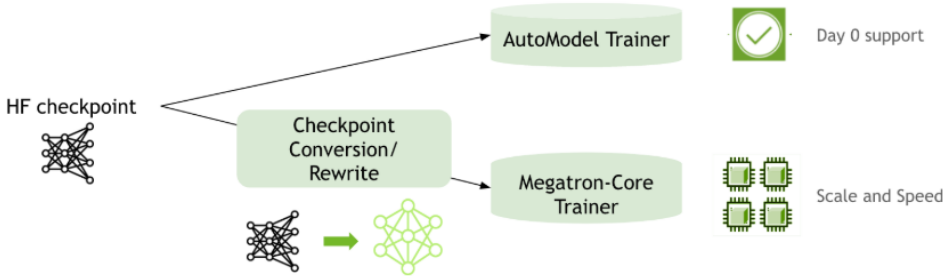
NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

NVIDIA

NeMo AutoModel

PyT-based Training Library. HF compatible Acceleration. Without Checkpoint Conversion

- **Day 0 support**
  - HF-native interface: Drop-in, HF-style APIs (e.g., AutoModel-like) for model instantiation
- **Broad Model Coverage**
  - ‘000s of models on HF supported OOTB. see full list (LLM, VLM/OMNI)
- **Frictionless Finetune**
  - No checkpoint conversion is needed!
- **Acceleration & scale**
  - Uses liger-kernel, FSDP2, DeepEP, full n-d parallelism



Back to main

Day 0 support - The only library with Day 0 support for Large MoEs

NeMo AutoModel

codecov 84% | CICD NeMo | python 3.10+ | contributions welcome | Star 963

Documentation • Ready-to-Use Recipes • Examples • Model Coverage • Performance • Contributing

News and Discussions

- [09/11/2026] **DeepSeek-V4.1-Flash** We now support SFT and LoRA for DeepSeek's causal encoder/decoder backbone with CSA2, single-pass mHC, and Engram, many thanks to @Khazic. Check out the HellaSwag EP64 16-node recipe and model coverage page.
- [08/29/2026] **GLM-5.3** We now support full-parameter fine-tuning of Z.ai's GLM-5.3 with cuDNN DSA and HybridEP. Check out the EP64/PP4 recipe and model coverage page.
- [08/28/2026] **Ox Alpha / GLM-5.3-Flash** We now support fine-tuning Z.ai's 320B-A18B hybrid-attention MoE VLM with packed CP and EP. Check out the MedPix EP72/CP2 recipe and model coverage page.
- [08/26/2026] **Qwen3.8-Flash-Next** We now support full-parameter fine-tuning for the 180B hybrid-attention Engram MoE with FlexAttention sparse QSA, packed (THD) sequences, and model-owned context parallelism. Check out the HellaSwag EP64 recipe and model coverage page.
- [08/21/2026] **DFlash 2** We now support training DFlash 2 drafters: the DFlash block-diffusion backbone plus a two-tap dynamic convolution and a pairwise path selector for higher acceptance lengths. Check out the Qwen3-8B recipe and the speculative decoding overview.
- [08/14/2026] **Qwen3.8-27B** We now support full-parameter SFT and LoRA fine-tuning for Qwen's dense 27B vision-language model. Check out the SFT and LoRA recipes, fine-tuning guide, and model coverage page.
- [08/12/2026] **Qwen3.8-2.4T-A95B** We now support full-parameter fine-tuning for `Qwen/Qwen3.8-2.4T-A95B` checkpoints. Check out the HellaSwag EP32/PP8 recipe and model coverage page.
- [08/12/2026] **North Micro Vision** We now support LoRA fine-tuning for Cohere Labs' 2.4B-parameter native-resolution vision-language model. Check out the RDR recipe and model coverage page.
- [08/10/2026] **MuseGlimmer** We now support SFT/LoRA the dense 30B MuseGlimmer vision-language model, including TP/CP packed sequence recipes. Check out the recipes and model coverage page.
- [08/08/2026] **Nemotron 3.5 Lightning** We now support LoRA fine-tuning for NVIDIA's 30B-A3B hybrid MoE model with multi-token prediction. Check out the HellaSwag recipe and model coverage page.
- [07/30/2026] **Inkling-Small** We now support full-parameter fine-tuning for the 276B-parameter, 12B-active Inkling-Small model on 64 H100 GPUs. Check out the MedPix EP64 recipe and model coverage page.
- [07/29/2026] **Kimi K3** We now support full-parameter fine-tuning for Moonshot AI's 2.8T-parameter MoE model on NVIDIA GB200. Check out the HellaSwag EP32/PP8 recipe on 256 GB200.

poolside.ai/blog/introducing-laguna-s-2-1

aus | pio | Embedding | Inf | training | general

output and $0.01 cache-read per 1M tokens.

For developers who want to go further, Laguna S 2.1 can be post-trained with NVIDIA NeMo AutoModel and Prime Intellect's Prime Lab. ZML's LLMD framework runs it across a wide range of hardware.

Qwen
@Alibaba_Qwen

Big thanks to Day 0 support! Developers can start building today for free on nim and finetune with NeMo recipe:

NVIDIA-NeMo/
Automodel

NVIDIA

Pytorch Distributed native t
LLMs/VLMs with OOTB Hug

46 Contributors | 120 Issues

Automodel/examples/vlm_finetune

From github.com

blog.google/innovation-and-ai/technology/develo

A Links | bank | aus | pio | Embedding | In

Hackable Diffusion, a modular JAX toolbox designed for
Unsloth and NVIDIA NeMo. Additionally, official support f

Back to main

Model Y is just released, is it supported? Simple to validate!

- Does NeMo Automodel support my model? Try it!
- LLM

```
python/torchrun –nproc-per-node 8 examples/llm_finetune/finetune.py \
--config /path/to/llm_config.yaml \
--model.pretrained_model_name_or_path NEW_MODEL_ID
```

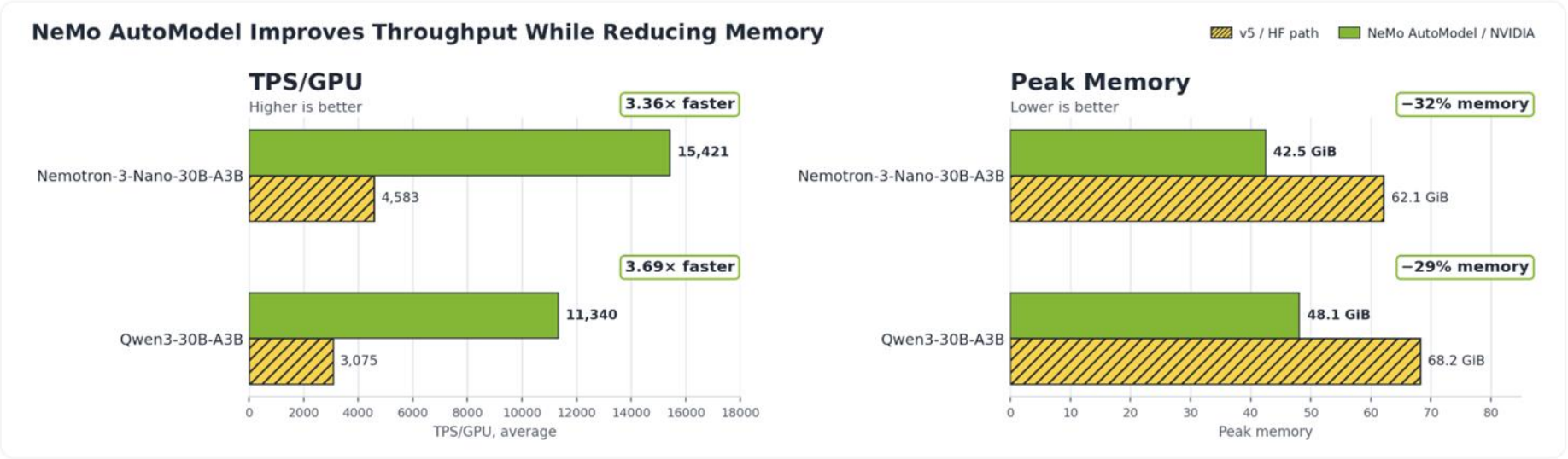
- VLM
  - check the vlm's multimodal preprocessor
    - if it uses HF's AutoProcessor:

```
python/torchrun –nproc-per-node 8 examples/vlm_finetune/finetune.py \
--config /path/to/vlm_config.yaml \
--model.pretrained_model_name_or_path NEW_MODEL_ID
```

    - if it has custom preprocessing code, put them in your own `collate_fn` and use this `collate_fn` in your own yaml.

NeMo AutoModel Improves Throughput While Reducing Memory

https://huggingface.co/blog/nvidia/accelerating-fine-tuning-nvidia-nemo-automodel



More performance numbers are https://docs.nvidia.com/nemo/automodel/latest/performance/performance-summary

Two paths in Automodel

OOTB Path

Custom Model Path

Wraps Models from transformers

- FSDP2, TP, PP support via AutoPipeline
- Faster CE loss with lower mem consumption
- Torch.compile()
- Torch AO
- Liger kernel, Triton Kernel (peft)
- AutoProcessor for audio/visual input

Model Coverage (hyperlink)

- LLMs: Dense LLMs, small MOE LLM
- VLMs/Omni: dense VLMs

Provides optimized model implementations

- Re-write layers
  - Attention & dense layers with TE kernels enabling speedups & CP + SP
  - MOE layers with DeepEP
- Rewrite Model Skeleton
- Enforce contract on HF checkpoint format (i.e., no checkpoint conversion)
- Custom audio/visual processing pipeline
- Reference example for user-defined architectures (eg. DSV3)

Agenda

Part 2 - Deepseek v4 Case Study

- Performance Profiling
- DSv4 Architecture
- Running Moonlight OOTB
- Profiling Tools
- Kernel Level Optimizations
- Scaling up and out with nD parallelisms
- Performance with nD parallelisms
- Final performance profile
- Correctness testing
- Checkpointing
- Training Loop Considerations

Case Study Introduction

We start off with introductions to Performance metrics, and Performance profilers to lay the background

Then we walk through an end-to-end optimization journey for training **DeepSeek v4-style MoE models**, starting from an OOTB baseline and progressively applying optimizations.

The choice of this specific architecture is motivated by the prevalence of sparse MoE configurations in today's popular LLMs. Additionally, the Moonlight-v4-16B variant serves as an accessible small-scale proxy, and the architecture permits a comprehensive exploration of optimizations involving specialized attention mechanisms.

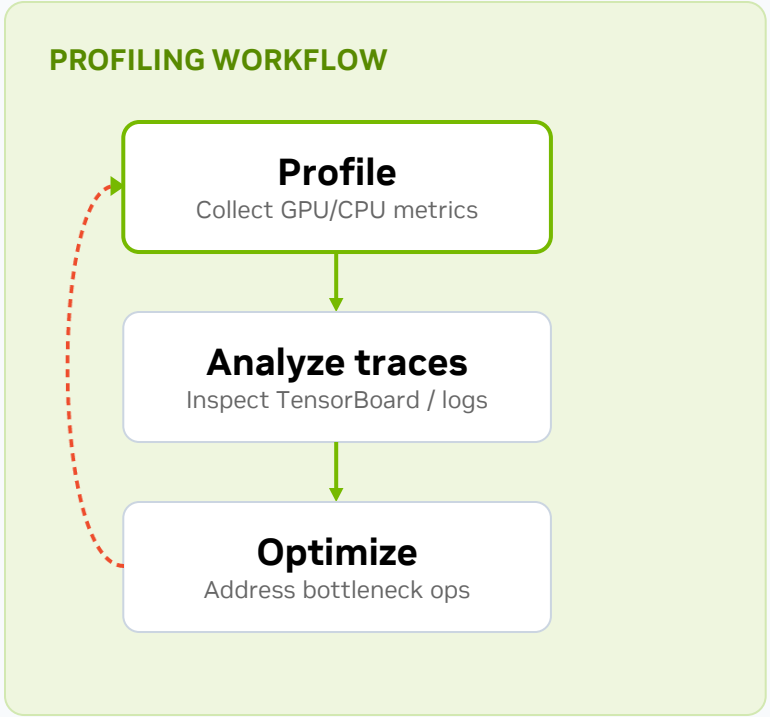
Case Study Notes

- By performance optimization, we refer to optimizing the throughput or step times related to training. We do not focus on improving functional performance like accuracy on benchmarks, etc
- We will focus on framework level optimizations i.e. We won't dive down into the inner workings and details of optimized kernels like FlashAttention, but we will try to use the kernels and tools available to us to get the best performance in our training loop
- A lot of the things shared in this case study are based on our empirical experience, and there are certainly other ways or methods to profile and optimize performance as well

Profiling Tools - PyTorch Profiler 1) torch.profiler

What it's best for:

- **Quick Python-level + operator-level profiling.**
- Correlates CPU and CUDA activity
- Exports traces to TensorBoard
- Can record shapes, memory, and stack traces (with overhead)
- **General rule of thumb:** it is more heavy/overhead than nsys.



Profiling Tools - PyTorch Profiler 1) torch.profiler

What it's best for:

- **Quick Python-level + operator-level profiling**
- Correlates CPU and CUDA activity
- Exports traces to TensorBoard
- Can record shapes, memory, and stack traces (with overhead)

Use cases:

- Find expensive ops (both CPU and GPU)
- Spot CPU bottlenecks and CUDA launch overhead
- Memory profiling with **torch.cuda.memory**

Find more info in **profiler_recipe** & **memory_profile**

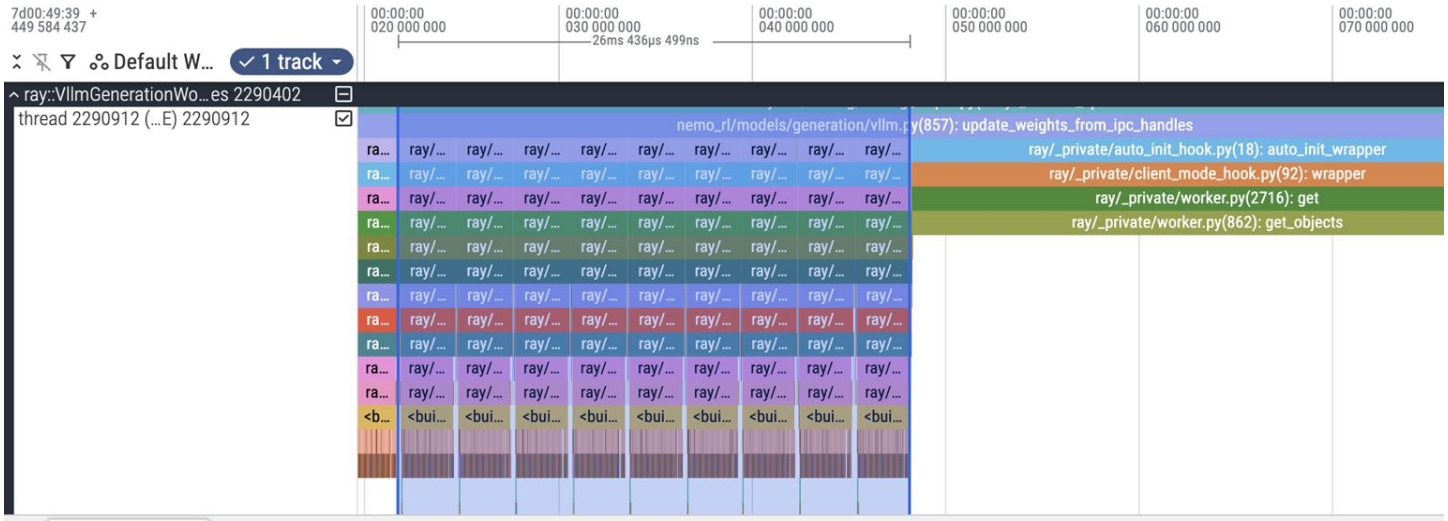
Profiling Tools - PyTorch Profiler 1) torch.profiler

```
import torch
from torch.profiler import profile, ProfilerActivity

with profile(
   activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
   schedule=torch.profiler.schedule(wait=1, warmup=1, active=3),
   on_trace_ready=torch.profiler.tensorboard_trace_handler("./logs"),
   record_shapes=True,
   profile_memory=True,
   with_stack=True,
) as prof:
   for step, batch in enumerate(dataloader):
       loss = model(batch)
       loss.backward()
       optimizer.step()
       prof.step()
```

Profiling Tools - PyTorch Profiler 1) torch.profiler

https://ui.perfetto.dev/



Current Selection

Area Selection | Slices | Pivot Table | Slice Flamegraph

Name	Wall duration ↓	Self duration	Avg Wall duration	Occurrences
	Σ 508.0ms	Σ 26.44ms		Σ 55365
nemo_rl/models/generation/vllm.py(857): update_weights_from_ipc_handles	26.44ms	49.18µs	26.44ms	1
ray/_private/async_compat.py(50): wrapper	26.44ms	0s	26.44ms	1
ray/_private/function_manager.py(689): actor_method_executor	26.44ms	0s	26.44ms	1
ray/util/tracing/tracing_helper.py(463): _resume_span	26.44ms	0s	26.44ms	1

Profiling Tools - Nsys 2) Nsight Systems

What it's best for:

- End-to-end timeline: CPU threads, GPU work, CUDA runtime, NVTX ranges
- Communication visibility (NCCL) and overlap with compute
- Identifying synchronization stalls and idle gaps

Use cases:

- "Why is my GPU idle?"
- "Is data loading blocking the step?"
- "Are NCCL collectives overlapping with compute?"

Find more info in **nsight user guide**

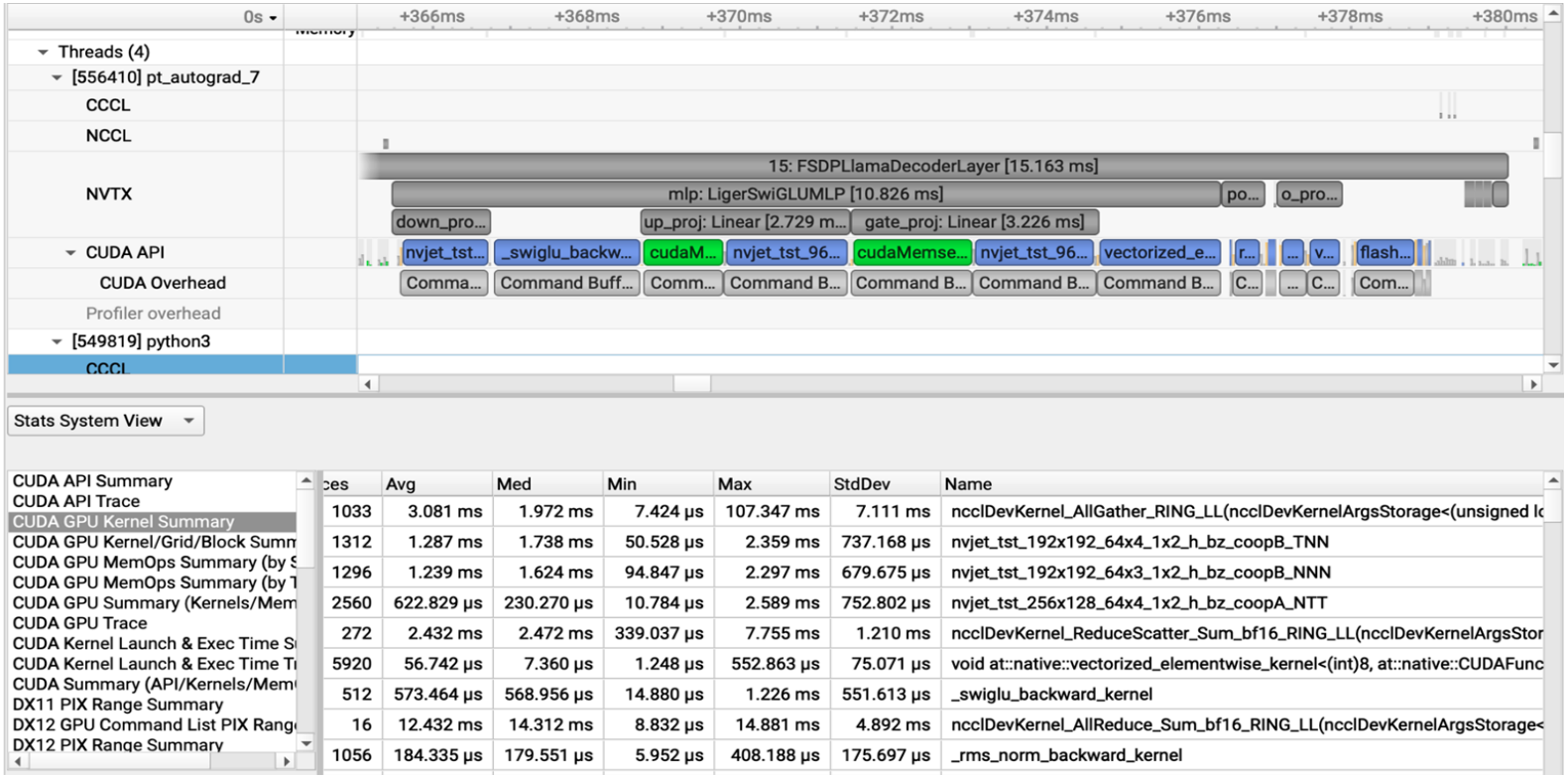
Profiling Tools - Nsys 2) Nsight Systems

```
# Profile a training run
$ nsys profile \
--trace=cuda,nvtx,osrt \
--cuda-memory-usage=true \
--output=profile_output \
python train.py
```

```
# View in Nsight Systems GUI
$ nsys-ui profile_output.nsys-rep
```

Find more info in **nsight user guide**

Profiling Tools - Nsys 2) Nsight Systems



Timeline inspection and agent measurement

The GUI shows trace shape. The agent computes a number and tests it.

BEFORE
nsys-ui, human at the timeline
Scroll, zoom, eyeball
Recognize a pattern
Know what to look for in advance

Intuition carries the method

NOW
Ask in words, agent measures and reveal bottleneck
> Where does the step time go?
A table from SQL over the export
> Is that real or noise?”
Measurement mean, and an error bar

Intuition becomes optional

GUI What does this look like?

AGENT What is the accurate metrics

What a single timeline cannot answer

Counting, aggregation, and cross-profile comparison require computation.

EYEBALLING A TIMELINE

Counting
“The gaps look frequent”

Aggregating
“Attention looks busy”

Comparing profiles
Open five windows and squint

AGENT OVER THE EXPORT

Counting
1,770 gaps (Mean 23.8 µs, 81%host-side)

Aggregating
22.8% of the stall (226 gaps, 37 µs each)

Comparing profiles
One query (Five profiles, one table)

Optimization impact is measured more accurately across runs
Compare the same metric before and after the change to distinguish real improvements from noise

Verification keeps the agent on track

Each claim needs a cheap check that could prove it wrong.

CHEAP CHECKS

“Metric X shows a 20% win”
Read what the metric actually measures **(1 query)**

“Module Y regressed”
Use 3 unchanged runs as replicates **(1 query)**

“The host is blocked”
Trace OS calls and test overlap with the stall **(1 flag)**

“No data for that”
List the available tables **(1 command)**

VERIFICATION PRINCIPLES

DERIVE A NOISE FLOOR

Runs that should not differ become replicates. A delta inside their spread is not a finding.

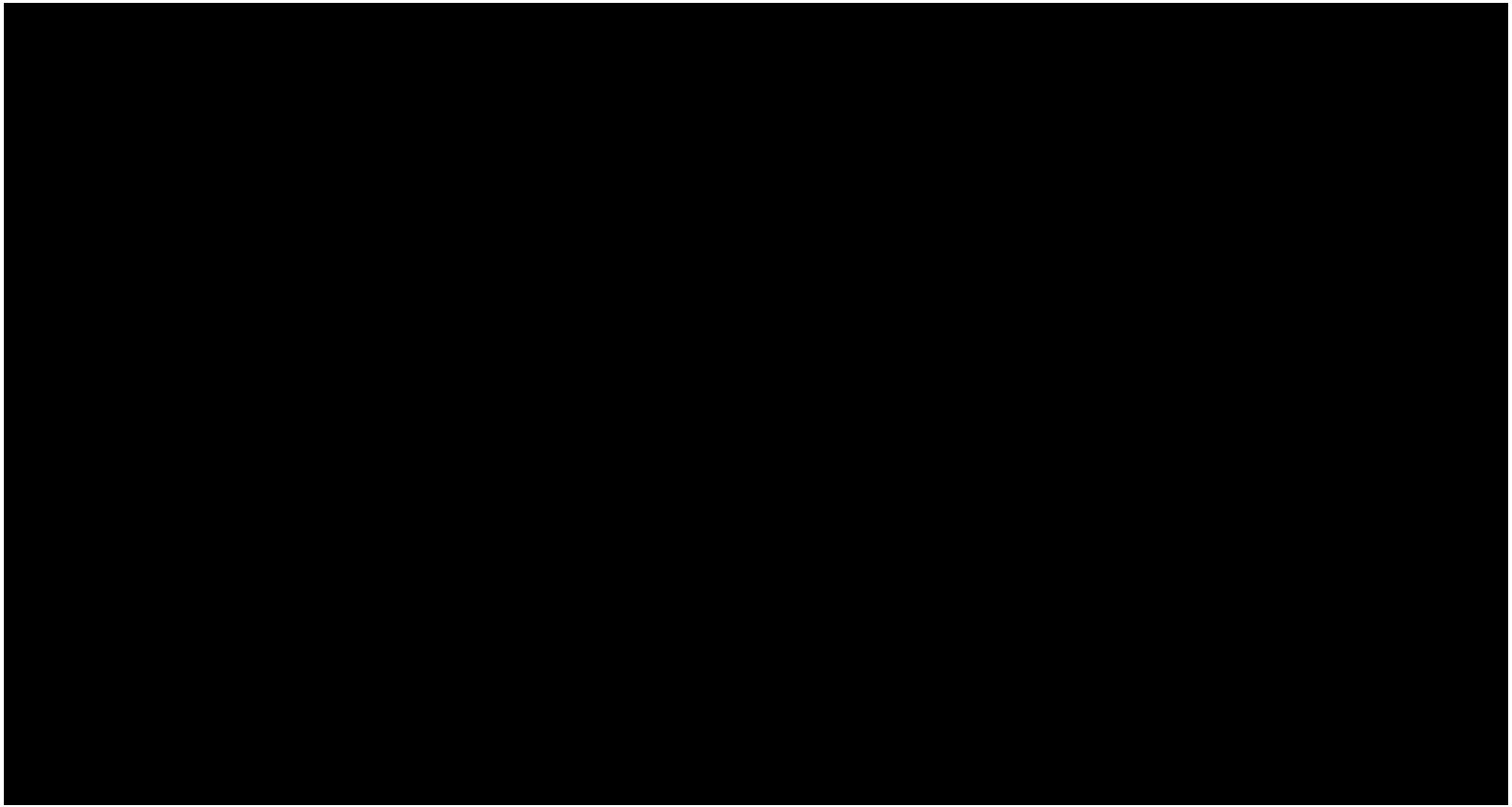
CHECK THE MEASUREMENT

Span versus busy. Forward versus forward plus backward. Warm versus cold. The metric name rarely tells you.

Verification makes performance claims trustworthy across runs

Low-cost checks prevent false wins and misattributed regressions.

Vibe Nsight Analysis



Minimal setup and human oversight

SETUP

NEEDED
Profile like:
nsys export --type sqlite
Shell access to the exported SQLite file

OPTIONAL
Domain analysis scripts
The GUI for a final human sanity check

NOT NEEDED
Agent familiarity with your model
Prior knowledge of which nsys report to open

OPERATING BOUNDARIES

1. Human suspicion remains required
Every correction in this session began with suspicion and challenge

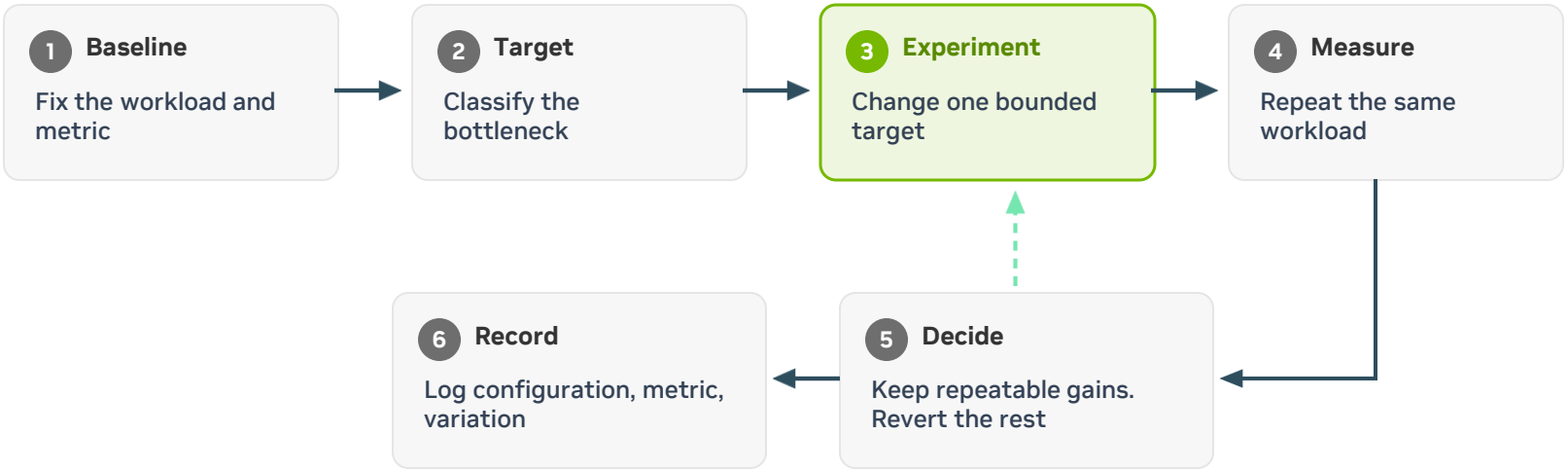
2. The question limits the answer
A narrow question can produce a confidently wrong answer in context.

3. Verification is the transferable skill
Ask how the number would look if it were false.

The agent makes repeated verification affordable. The human supplies suspicion.

The Optimization Experiment Loop

Fix the workload and metric. Change one target. Keep or discard the result.



FIXED EVALUATION CONTRACT Hold the workload, hardware, metric, and acceptance rule constant across experiments.

One loop, three search surfaces

The editable surface changes. The evaluation contract and decision rule stay fixed.

Autoresearch

SEARCH SURFACE

Training code

EVALUATION & DECISION

Fixed data, evaluator, budget

Keep lower val_bpb

Kernel Factory

SEARCH SURFACE

Kernel implementation

EVALUATION & DECISION

Fixed workloads, GPU, tolerances

Keep fastest correct candidate

Full application

SEARCH SURFACE

Kernel, parallelism, hyperparameter tuning

EVALUATION & DECISION

Fixed workload, hardware, metric

Keep repeatable end-to-end gain

SAME DECISION RULE

Keep the candidate only when its metric improves under the fixed evaluation contract.

Kernel Factory: RMSNorm in DSV4

A backend migration, not a model rewrite

1 RESOLVED CONTRACT

Field	Moonlight V4 contract
campaign	moonlight_v4_rmsnorm_h2048
model	hidden 2,048; eps 1e-6; 27 layers
input	[tokens, 2,048] BF16
math	FP32 accumulation; one BF16 store
scope	forward only

NUMERICS **BF16 input/output; FP32 accumulation; eps 1e-6**

BOUNDARY **Forward only; gradients require a second campaign**

2 NSIGHT WORKLOADS

Source	tokens	Role
Nsight stage	1,024	capture workload
Layer profile	4,096	microbenchmark
Training config	8,192	2× weight; e2e gate
Inference	separate	no token-1 case

Nsight supplied the shapes. The training campaign does not include token-1 decode.

3 BACKEND MIGRATION

Model Change **None — block and final norms already use the factory**

One YAML line **rms_norm: torch_fp32 → kf_triton_h2048**

Search on the GPU used for the final benchmark

All reference measurements here came from 8× H100

A B200 winner is not an H100 winner

Block and final norms migrate. Attention q/kv, compressor, and indexer norms stay torch_fp32.

Kernel Factory: RMSNorm in DSV4

The kernel is near the roofline; Moonlight runs below the crossover

1 KERNEL RESULT

Measurement	8,192 tokens
HBM roofline	0.0200 ms
KF Triton	0.0238 ms · 84% HBM
TE 2.11	0.0256 ms · 78% HBM
torch	0.0271 ms · 74% HBM
shipped torch_fp32	0.1473 ms · 14% HBM

KERNEL — **KF wins all nine tested sizes**

CROSSOVER — **torch wins ≤16K; KF wins ≥32K tokens**

2 WALL-CLOCK RESULT

Backend	kernel	wall
KF Triton	23.8 µs	83.5 µs
TE 2.11	25.6 µs	46.7 µs
torch	27.1 µs	28.8 µs
torch_fp32	147.3 µs	153.1 µs

At 8K tokens, dispatch dominates: 83.5 µs wall around 23.8 µs of CUDA.

3 OPTIMIZATION ORDER

First — **CUDA-graph the launch path**

Second — **Remove the full FP32 copy**

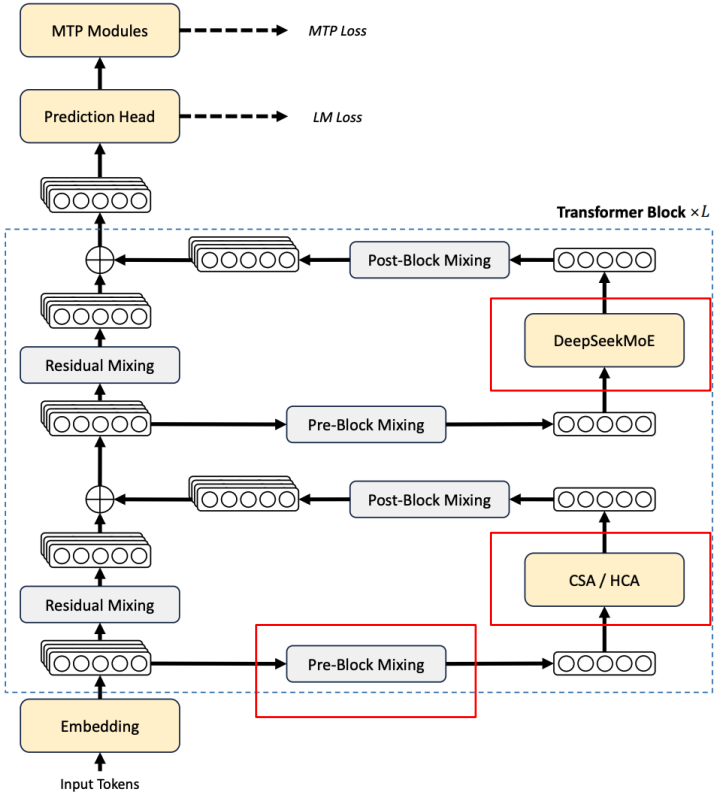
Moonlight: 8,192 tokens < 32K crossover

Dispatch floor, not kernel time, decides wall clock

Campaign third — workflow demo, not a speedup claim

Sparse MQA remains the architecture-specific second campaign.

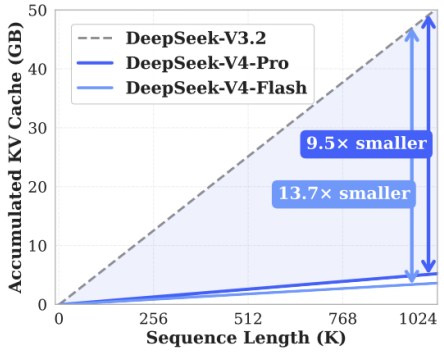
From Deepseek V3 to V4 Architecture

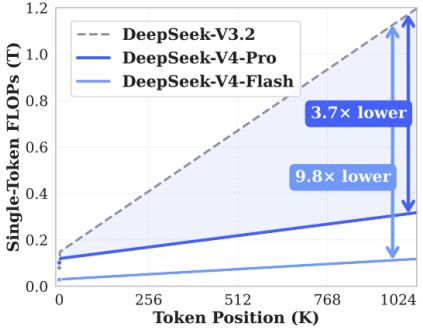


Ref: https://arxiv.org/pdf/2606.19348

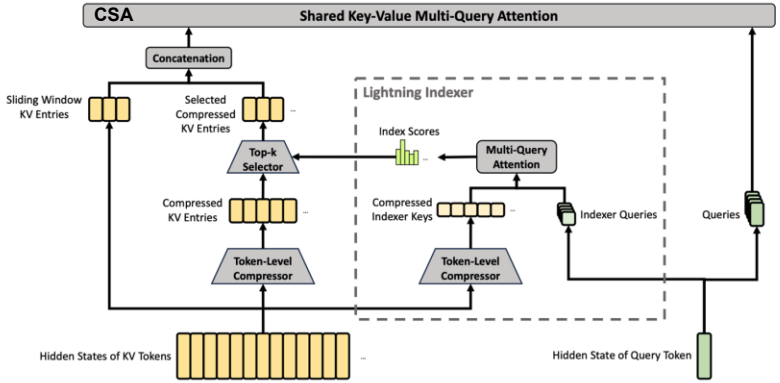
Highlights:

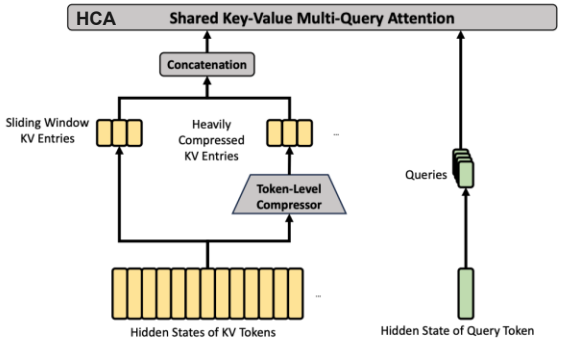
- **Efficiency is all you need**
- Hybrid attention to improve long context efficiency (CSA / HCA)
- Manifold-Contained Hyper-Connections (mHC)
- @ 1M token context
  - DSv4-Pro requires 27% FLOPs and 10% KV$ compared to DSv3.2





From Deepseek V3 to V4 Architecture





- Attention → MLA (low rank tokens) → CSA/HCA (compress seq len)
- KV → down_proj → token-compressor (CSA/HSA) N/m; m compr. factor

Ref: https://arxiv.org/pdf/2606.19348

Deepseek V4 Configuration

Model	Total Params	Active Params	# Layers	Experts	Top-K	CSA / HCA / SWA / Hash-route
DeepSeek-V4-Pro	1.573T	49.8B	61	384 routed + 1 shared (width 3072)	6	30 / 31 / 0 / 0,1,2
DeepSeek-V4-Flash	284.3B	13.8B	43	256 routed + 1 shared (width 2048)	6	21 / 20 / 2 / 0,1,2
Moonlight-V4-16B-A3B	16.5B	3.0B	27	64 routed + 1 shared (width 1408)	6	13 / 12 / 2 / 0

Key insight: DeepSeek-V4 extends **sparsity from the FFN to attention**. Routing stays fine-grained (V4-Pro activates **3.1%** of its 1.6T parameters), and the hybrid CSA/HCA attention compresses the KV cache 4x or 128x and lets each query attend to only 128 recent tokens plus a few hundred compressed entries, so per-token attention cost is nearly flat in context length. (recall: 27% of V3.2's FLOPs and 10% of its KV cache at 1M tokens)

Deepseek V4 Memory and Compute Characteristics

For DeepSeek-V4 the MoE profile carries over from v3, and the hybrid attention changes the long-context profile:

- Parameter memory: Scales with total parameters (all experts must be stored)
- Activation memory: Scales with active parameters (only selected experts compute)
- **KV cache: no longer linear in full-resolution tokens (new)**: 10% of V3.2 at 1M tokens.
- **Compute: attention cost nearly flat in context length (new)**. A CSA query attends to 128 window entries + 512 selected compressed entries, an HCA query to 128 + L/128; V4-Pro needs 27% of V3.2's per-token FLOPs at 1M tokens.
- Communication: Expert parallelism requires all-to-all communication to route tokens
- Load balancing: Uneven expert utilization can create stragglers

Starting the optimization journey with Moonlight-V4-16B-A3B

- Same architecture as Deepseek-v4, but downscaled
- Originally: downsize(Deepseek-v3) → Moonlight-16B-A3B
  - Similarly, downsize(Deepseek-v4) → Moonlight-V4-16B-A3B; created for the purposes of this lecture
- Only 16BA3B, small enough to profile/tune (single node)
- Trained checkpoints available for fine-tuning and correctness verification (tbd)

Memory footprint for Moonlight-V4-16B-A3B

Model States (M_states) = Parameters (P) * (Precision (M_p) + Gradients (M_g) + Optimizer States (M_opt))

- **Model Size (P):** 16.5 x 10^9 parameters
- **Precision (M_p):** 2 bytes (BF16)
- **Gradients (M_g):** 2 bytes (BF16)
- **Optimizer States (M_opt):** 4 bytes (Adam)

Global Model States:

M_states = 16.5B * (2B + 2B + 4B) = 132 GB

Per-Device Static Memory (with FSDP sharding on 8 GPUs):

M_device = 132 GB / 8 = 16.5 GB per GPU

Memory Capacity for Activations (M_act):

Assuming 80GB VRAM (A100/H100):

M_avail ≈ Total VRAM - 16.5 GB ~= 63.5 GB (assuming 80GB VRAM)

	Moonlight-16B-A3B	**Moonlight-V4-16B-A3B**	**Gain**
KV cache 8K context	243 MiB 31.1 KB/token	18.6 MiB 2.4 KB/token	13x
KV cache, 1M context	30.4 GiB	2.1 GiB	14.6×
Attention FLOPs per token, 8K / 1M	2.3 / 290 GF	0.8 / 59 GF	2.9× / 4.9×

This remaining capacity provides substantial headroom for scaling sequence lengths or batch sizes.

Case study setup

Branch and code: https://github.com/NVIDIA-NeMo/Automodel/tree/scalable-ai-2026_sept

Two ways to setup environment:

1. Automodel container - nvcr.io/nvidia/nemo-automodel:26.08.00
    - Note: Please mount code from the scalable-ai-2026_sept branch when running container (-v {git_repo}:/opt/Automodel)
2. uv sync in a CUDA environment

Code pointers and commands will be provided in the lecture document.

Stage 1: Running Moonlight-V4-16B-A3B OOTB

```
$ torchrun --nproc-per-node 8 nemo_automodel/recipes/llm/benchmark.py --
config examples/scalable_ai/configs/moonlight_v4_16b_hf.yaml
```

What does the **benchmark.py** recipe do?

- Sets up model and optimizer
- Creates mock batches to mimic pretraining by generating random sequences with tokens sampled from the Model's vocabulary size.
- Ensures a fixed sequence length and sets up the next token prediction loss on the batch
- Runs training steps (forward+backward, optimizer) in a loop

Stage 1: Moonlight-V4-16B-A3B OOTB Performance

8xH100, BF16 params, grads, optimizer states; GBS=256, MBS=1

Sequence Length	2048
MFU	**3.84499%**
Tokens/sec	18.25k (2.28k tokens/s/GPU)
Step time	28.732s (0.898 per mbs ; 32 grad acc steps)
Max memory utilization	49.31GB

Stage 1: Why is MFU low?

Currently, we can make some assumptions based on the implementation of the model:

- Generic kernels not optimized for the specific operations
- Suboptimal memory access patterns
- Python overhead e.g., with looped experts / module tree traversals etc.

We also need to objectively reason about performance, using the tools described earlier.

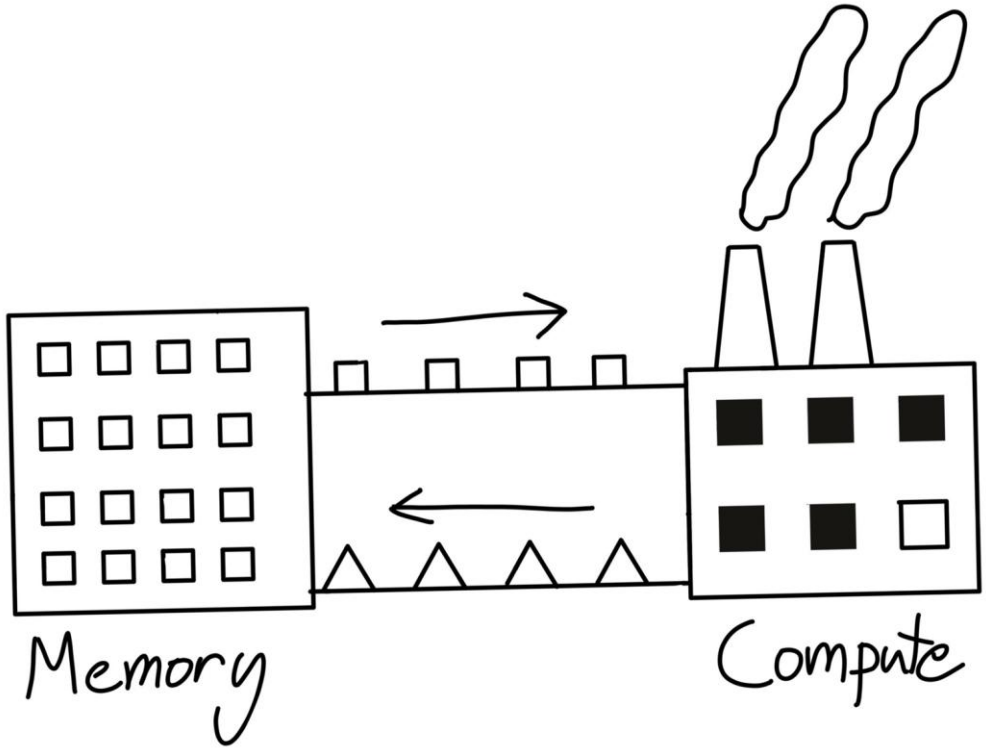
Initial Profiling Results

Profile: examples/scalable_ai/nsys_profiles/moonlight_hf_small.nsys-rep

Command:

```
$ nsys profile --force-overwrite true --trace=cuda,nvtx \
    --cuda-memory-usage=true --output=moonlight_hf_small.nsys-rep \
    torchrun --nproc-per-node 2 \
      nemo_automodel/recipes/llm/benchmark.py \
      --config examples/scalable_ai/configs/moonlight_v4_16b_hf.yaml \
      --benchmark.nsys_start 3 \
      --benchmark.nsys_end 5 \
      --step_scheduler.max_steps 6
      --benchmark.warmup_steps 1 \
      --step_scheduler.global_batch_size 4 \
      --dataset.seq_len 1024 \
      --model.config.num_hidden_layers 3
```

Mental Model for Performance Optimization



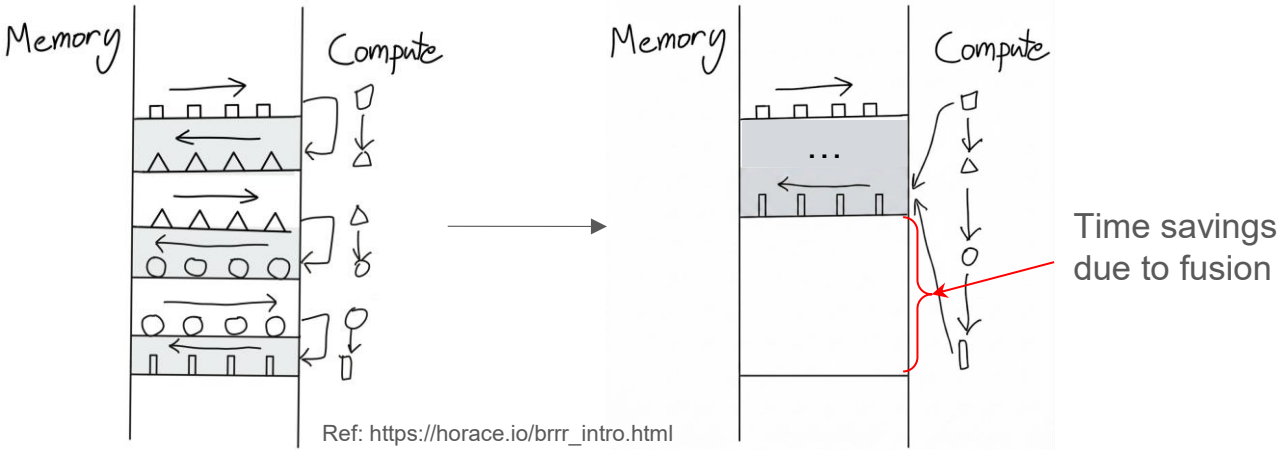
Ref: https://horace.io/brrr_intro.html

Optimization Themes

1. Kernel Level Optimizations
    - RMSNorm
    - mHC with torch.compile
    - GroupedGEMM
2. Scale up and out with nD parallelization
    - Expert Parallelism
    - Fully Sharded Data Parallelism
    - Pipeline Parallelism
    - Context Parallelism

Kernel Level Optimizations

1. Kernel fusion avoids unnecessary + expensive HBM traffic
   - H100: peak-BW(Shared memory)= ~33TB/s ; peak-BW(HMB3)= ~3.35TB/s ; 10x diff.
   - Epilogue fusion in CUTE/Cutlass → GEMM + Activation → fuse activation
2. Avoid launch overhead and command buffer stalls
   - Tiny kernels (e.g., add scalar to vector) can time(launch kernel) >> time(kernel runtime)
   - Queueing a lot of tiny kernels can fill command buffer → stall launch thread

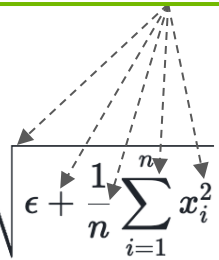


Kernel Level Optimizations – RMSNorm

1 math symbol ~
1 kernel launch

Approach: profile individual kernels in isolation for quick iteration

Optimization #1: RMSNorm FWD: $y_i = \frac{x_i}{\mathrm{RMS}(x)} * \gamma_i, \quad \text{where} \quad \mathrm{RMS}(x) = \sqrt{\epsilon + \frac{1}{n}\sum_{i=1}^{n} x_i^2}$



Implementation Approach:

1. Torch.compile / TE RMSNorm / Triton

Implementa	Kernels fwd	Kernels bwd	Time fwd (ms)	Time bwd (ms)	Time total (ms)	Speedup vs eager	Mem peak fwd (MB)	Mem kept for bwd (MB)
PyT eager	8	15	5.92	15.15	21.07	1.0×	1536	2304
torch.compile	1	3	0.81	1.12	1.94	10.9×	256	304
Triton (liger)	1	3	0.87	1.12	1.99	10.6×	256	256
F.rms_norm	1	2	0.69	1.93	2.62	8.0×	256	256
TE RMSNorm	1	2	0.79	1.08	1.87	11.3×	256	261

Note: bf16 input 8×4096×4096 (256 MB), NVIDIA RTX 5880 Ada Generation, PyTorch 2.11.0a0+eb65b36914.nv26.02, TransformerEngine 2.11.0+c188b533
- **Eager forward peaks at 6x the input** because the fp32 upcast, the squared tensor and the normalized tensor are all materialized; 4x the input stays alive for backward. Fused

Kernel Level Optimizations – mHC

Eager mHC	torch.compile(mHC)

`x32 = streams.float()`
`mix = linear(rmsnorm(x32), W)`
`pre, post, logits = gates(mix)`
`comb = sinkhorn(sigmoid(logits))`
`collapsed = (pre*streams).sum(hc)`
`expanded = post * y + matmul(comb.T,streams)`
	`pre, post, logits = compile(weights)(streams)`
	`comb = sinkhorn(...) # unchanged`
	`collapsed = compile(sum(pre * streams))`
	`expanded = compile(post * y + sum_i(comb_i *stream_i))`

23 FWD + 51 BWD = 74	9 FWD + 25 BWD = 34

27 layers Moonlight-v4, seq 2048 GBS 256, 8x H100	Off	On	Gain
step time	14.520 s	13.721 s	5.5% lower
throughput	36.11k tok/s	38.21k tok/s	**5.8% higher**
MFU	7.61%	8.05%	+0.443 abs.
rank-0 peak memory	47.98 GiB	44.65 GiB	−3.33 GiB

Pros: easy and quick to implement

Cons:

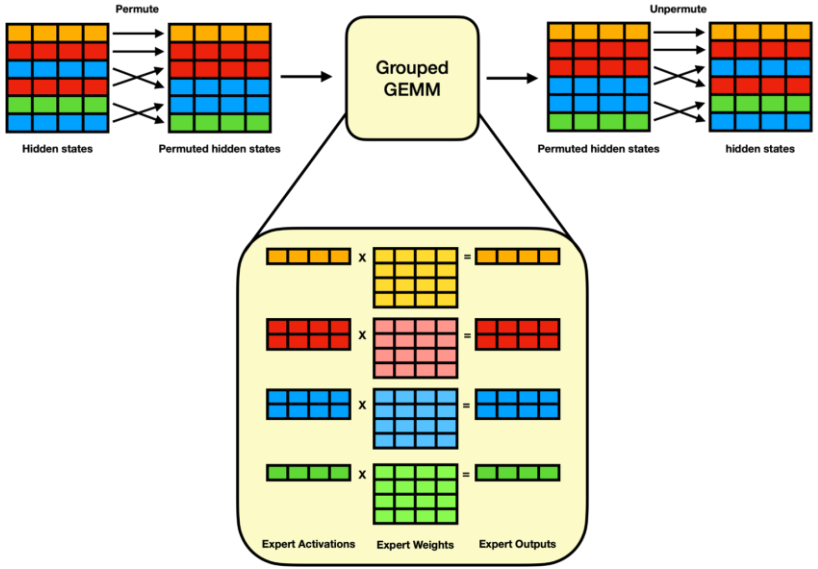
- Dynamic shapes
- Compilation time / bugs

Kernel Level Optimizations – GroupedGEMM

Approach: Test and profile individual kernels in isolation for quick iterations

Optimization #3: GroupedGEMM for Expert computation to fuse multiple differently-sized (input) matrix multiplications into a **single** kernel launch.

Before	After
N kernel launches	1 kernel launch
Poor GPU utilization	High GPU utilization
Launch overhead dominates	Compute dominates

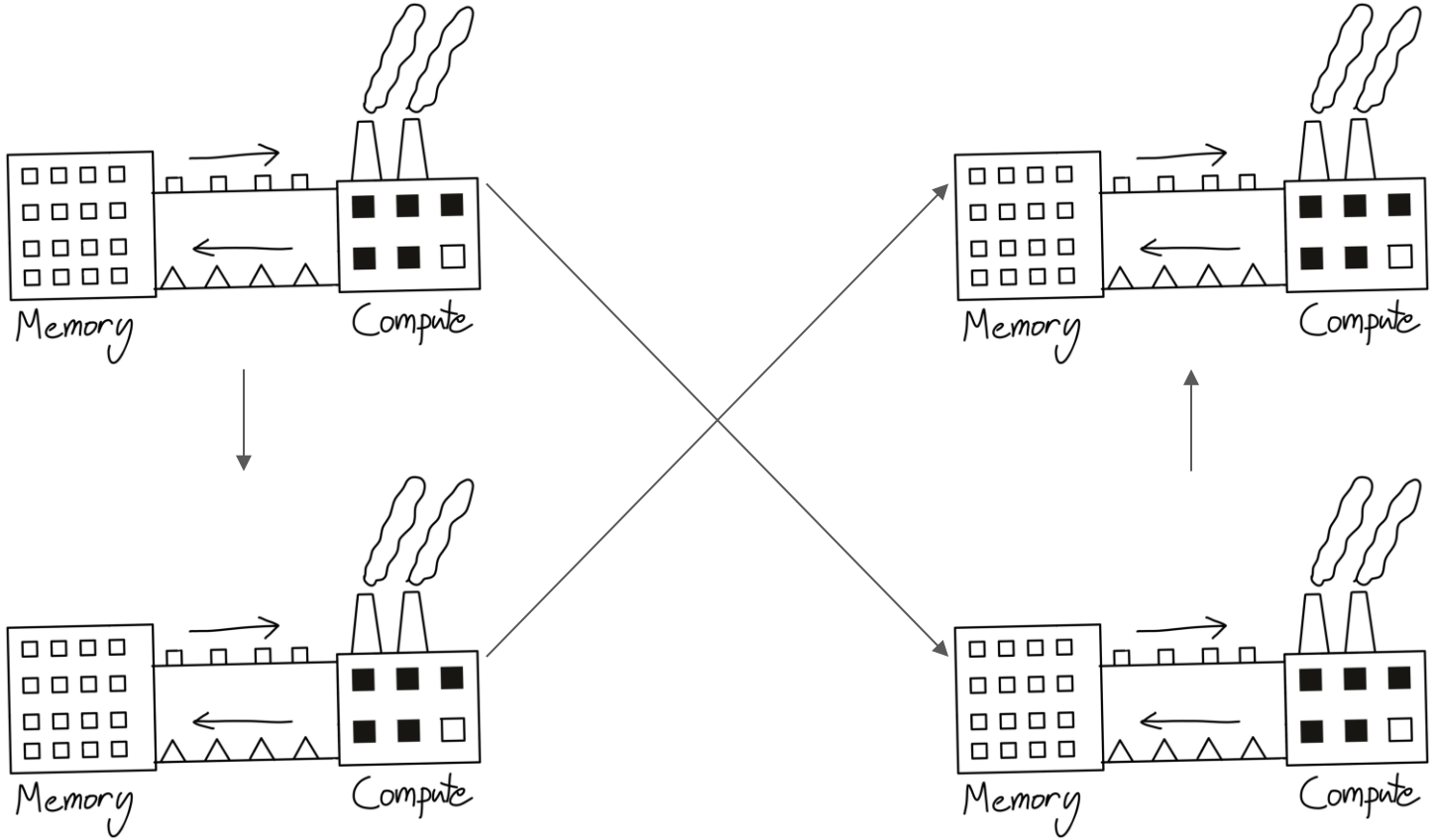


https://tech.kakao.com/posts/716

E2E optimizations for Moonlight-V4-16B-A3B

Change	Deferred sync	LBS / GA	Step time	MFU	Tokens/s	Rank-0 peak	Increm. gain	Attribution
Stock HF	FALSE	1 / 32	28.734 s	3.85%	18.25k	49.31 GiB	Baseline	—
Native EP8, eager attention	TRUE	1 / 32	19.748 s	5.59%	26.55k	58.76 GiB	31.30%	Other: native implementation plus expert parallelism; not isolated to a kernel
grouped_gemm + DeepEP	TRUE	1 / 32	19.853 s	5.57%	26.41k	58.80 GiB	−0.5%	Combined kernel/communication change; slightly slower at this microbatch
TileLang sparse attention	TRUE	1 / 32	14.520 s	7.61%	36.11k	47.98 GiB	26.9%	Kernel-path gain: sparse attention, indexer, and Sinkhorn kernels; replacement rather than fusion
`compile_hc`	TRUE	1 / 32	13.721 s	8.05%	38.21k	44.65 GiB	5.5%	**Kernel fusion:** compiled mHC mixer and collapse/expand elementwise cores
BF16 gradient reduction	TRUE	1 / 32	13.584 s	8.13%	38.60k	42.62 GiB	1.0%	Precision/communication: halves gradient-reduction traffic; no kernel fusion
BF16 mHC and LM-head projections	TRUE	1 / 32	13.341 s	8.28%	39.30k	38.83 GiB	1.8%	Precision: two FP32 projections changed together; no kernel fusion
Local batch 2	TRUE	2 / 16	**9.005 s**	**12.268%**	**58.22k**	56.20 GiB	32.5%	Batching/utilization; no kernel change

Scaling Out w nD parallelism



Ref: https://horace.io/brrr_intro.html

EP Communication Within and Across NVLink Domains

Peer-memory access inside a domain; GPU-initiated RDMA between domains.

EP LIBRARY	DeepEP v1	HybridEP	DeepEP v2
WITHIN AN NVLINK DOMAIN	**CUDA peer-memory access** LD/ST · TMA over NVLink	**CUDA peer-memory access** LD/ST · TMA over NVLink	**NCCL LSA** Peer-memory access over NVLink
ACROSS NVLINK DOMAINS	**NVSHMEM / IBGDA** GPU-initiated RDMA	**DOCA** GPU-initiated RDMA	**NCCL GIN** GPU-initiated networking RDMA over IB / RoCE

DATA PATH

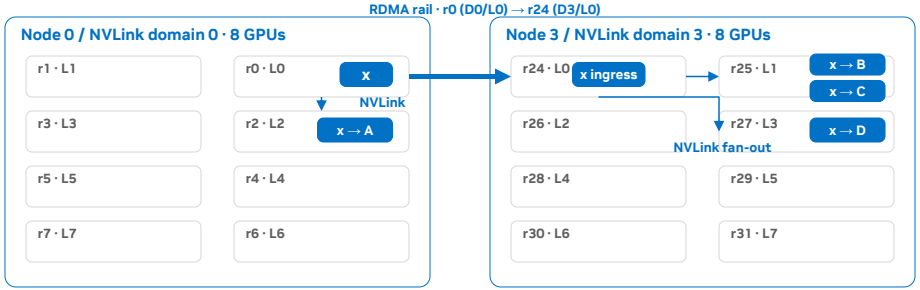
Within: GPU A → NVLink / NVSwitch → GPU B
Across: GPU A → PCIe → NIC → IB / RoCE → NIC → PCIe → GPU B
GPUDirect RDMA lets the NIC access GPU memory without CPU staging.

One Token Through an MoE Layer

H100 assumption: four nodes; each node is one 8-GPU NVLink domain. This token uses Node 0 and Node 3.

ORIGINAL TOKEN	SELECTED EXPERT	GPU
x on rank 0	**A**	rank 2
same x	**B, C**	rank 25
same x	**D**	rank 27

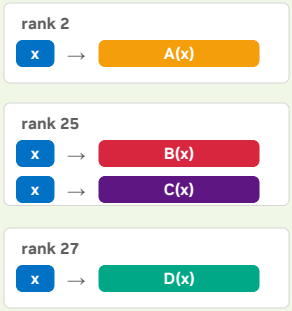
1 · Dispatch — how does x reach A, B, C, D?



Across domains, x travels once on the L0 rail; NVLink fans out to ranks 25 and 27.

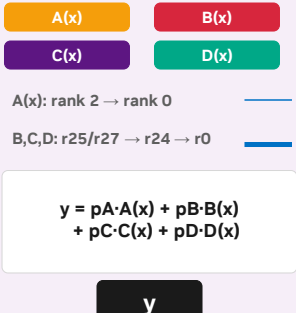
First, send the token over RDMA along the same rail to the same local rank in the destination domain.
Then use NVLink to distribute it to the GPUs that need it.

2 · What runs on each GPU?



3 GPUs run 4 expert MLPs with different weights.

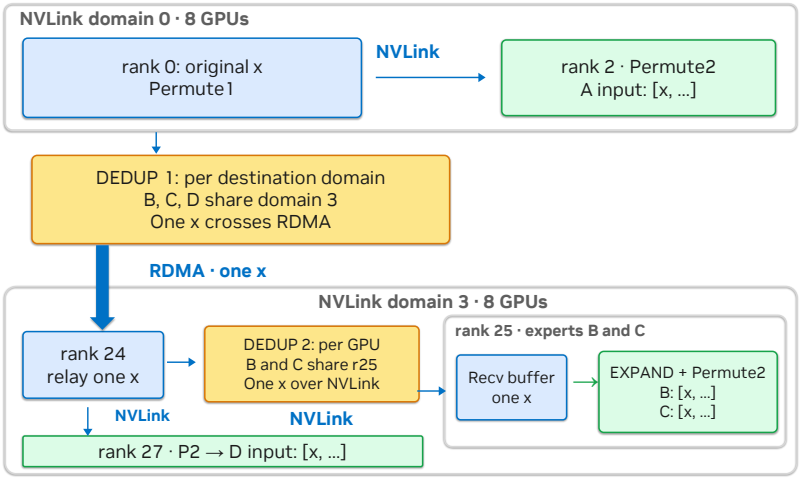
3 · How do outputs return to x?



Dispatch: Deduplicate, Then Expand for Experts

x on rank 0 selects A@r2, B/C@r25, D@r27. Each node has 8 GPUs.

Shared relay path: v1 normal · HEP · v2 hybrid



DeepEP v1 normal
r0 → r24 → r25 / r27; the caller runs local expand + P2 after communication.

HEP
Same relay path; dispatch_with_permute includes expand + P2, with optional fusion.

DeepEP v2
Hybrid: same path. Direct: r0 → r25 and r27, sending two RDMA copies of x.
do_expand=True creates expert-major inputs in the dispatch copy epilogue.

Permute1 + dedup: input → send layout

Input on rank 0: [x, u, v]

x → A@r2, B@r25, C@r25, D@r27
u → A@r2
v → B@r25

↓ Group by destination rank

r2: [x, u]
r25: [x, v] (one x for both B and C)
r27: [x]

DEDUP: suppress repeated payloads

x → B@r25 and x → C@r25
share one x; keep both routes in metadata.

EXPAND: create one row per expert

Received at r25: [x:{B,C}, v:{B}]
Expand: [x_B, x_C, v_B]

P2: from source-rank-major to expert-major

[x_B, v_B | x_C]
B rows C rows → grouped GEMM

x_B and x_C contain the same x vector.

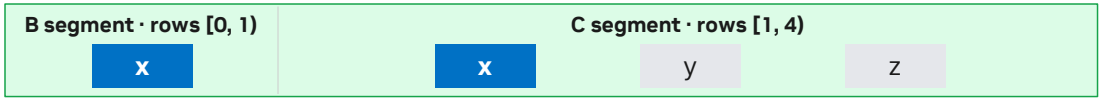
Push Dispatch in all three: the source / relay writes into destination buffers.
Shown relay path: 4 experts · 3 target GPUs · 1 hidden vector across RDMA.

Logical flow; DEDUP boxes are not separate kernels. Sources: DeepEP v1 · HEP · DeepEP v2

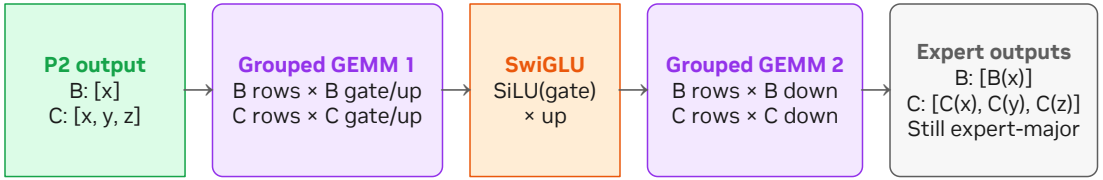
Expert Compute: Two Grouped GEMMs

P2 supplies expert-major rows.

Inside rank 25 · experts B and C



tokens_per_expert = [1, 3] offs = [1, 4] (cumulative ends)



Segment boundaries select the weights
Counts or offs locate each expert's rows. The two blue x entries contain the same vector; B and C process it with different weights.

Grouped GEMM schedules multiple expert matmuls together
Each expert keeps its own weight matrix. Token counts can differ between experts and change with routing on every step.

Gate + up share the first grouped GEMM
AutoModel packs both projections into one weight tensor; then applies SwiGLU and the grouped down projection.

Schematic omits bias, padding and router weights. Router weights apply in activation or after down, subject to model/backend support. FP8 / MXFP8 requires supported hardware and libraries; the torchao path can fall back to torch._grouped_mm.

Source: NVIDIA NeMo Automodel · experts.py / mxfp8.py · e2c47c5

AutoModel Expert Compute Backends

Config experts → implementation → acceleration

gmm
grouped_gemm.ops.gmm
Schedules multiple expert GEMMs as a group.

torch_mm
torch._grouped_mm
Native PyTorch grouped GEMM; GPU-resident offs describe the segments.

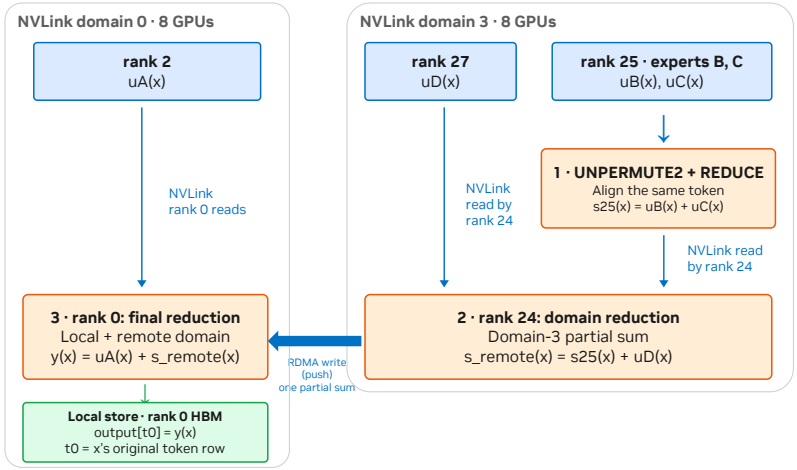
te
TE GroupedLinear
Optimized grouped GEMM with TE FP8 / MXFP8 recipes.

torch_mm_mxfp8
torchao MXFP8 grouped GEMM
Dynamically quantizes both operands to MXFP8 for low-precision compute.

Combine: Restore Token Order and Add Contributions

Rank 0 computes top-k weights from Wrouter·x (e.g., softmax; normalization/scaling is model-specific).
Dispatch carries these weights with x; each expert applies its weight before Combine, giving uE(x).

HEP return path: NVLink pull, then RDMA push



Same return path; different GPU initiates the NVLink transfer

DeepEP v1 normal — NVLink PUSH + RDMA PUSH
r25 / r27 write r24's buffer; r24 reduces locally, then writes the partial sum to r0.

HEP — NVLink PULL + RDMA PUSH
r24 reads the outputs on r25 / r27, reduces them, then writes the partial sum to r0.

DeepEP v2 hybrid — NVLink PUSH + RDMA PUSH
r25 / r27 write r24's buffer; r24 reduces* and uses NCCL GIN put to send to r0.

* v2 can defer reduction to the origin; direct mode pushes straight to the original rank.

Rank 25: What Unpermute2 Does

GEMM output is still expert-major

B: [uB(x)]
C: [uC(x), uC(y), uC(z)]

Unpermute 2: Expert major -> source rank major(saved received token order)

1 · UNPERMUTE2 + LOCAL REDUCE
4 token–expert rows → 3 token rows

Restored token row	Rank-25 partial sum
x	uB(x) + uC(x)
y	uC(y)
z	uC(z)

1 Within a GPU
B and C contribute to the same x row.

2 Within the remote NVLink domain
Rank 24 reads and adds ranks 25 and 27.

3 At the original rank
Rank 0 adds local and remote contributions.

Dispatch fans inputs out; Combine adds expert contributions back together.

The diagram shows HEP. Push / pull describes who initiates the cross-GPU payload transfer; the arrows show data flow.

Sources: v1 NVLink sender / RDMA put · HEP peer reads / RDMA write · v2 remote stores / GIN put · unpermute

Mixture of Kittens: Dispatch, Expert Compute and Combine in One Megakernel

Same B / C example; now ranks 0 and 25 share one GB200 / GB300 NVL72 domain. Only supports DSV3 style MoE.

Our mental model: route / layout → expert FFN → restore / reduce

Inside the persistent megakernel on the expert GPU

PULL dispatch
Rank 25 reads source tokens directly into expert segments
B: [x] | C: [x, y, z]

→

Expert FFN
Gate / up GEMMs
→ SwiGLU → down
Each expert keeps its weights

→

PUSH combine
Write B(x), C(x) back to rank 0's separate route slots
Reuse the dispatch schedule

NVLink / TMA moves the payloads; GPU-local counters connect communication and compute.

Rank 25: expert-major schedule
Expert row ← (source rank, token, top-k slot)
B row 0 ← (r0, t0, k1) → x
C row 0 ← (r0, t0, k2) → x
C row 1 ← (r1, t0, k0) → y
C row 2 ← (r2, t0, k0) → z
Logical rows shown; implementation pads segments.

Rank 0: restore x at token row t0
Return slots: [A(x), B(x), C(x), D(x)]
Local epilogue applies routing weights:
y(x) = pA·A(x) + pB·B(x) + pC·C(x) + pD·D(x)

output[t0] = y(x)
Routed experts shown; shared expert omitted.

Expansion happens while pulling: B and C each read x.
Their outputs return separately; the source rank performs the weighted sum.

What Changes in Execution?

1 · The schedule defines the layout

All-gather routing IDs; build a GPU schedule.
Each row names its source and expert slot.
Pull places x directly into B / C rows:
expand + P2 happen during dispatch.

2 · Overlap at minibatch granularity

Communication SMs move tokens while compute SMs run ready expert work.
Tune minibatch_size and fwd_num_comm_sms / bwd_num_comm_sms.

3 · Reuse a bounded activation ring

macrobatch_size sets ring capacity.
Fill → compute → drain → reuse.
GPU counters govern safe slot reuse.

BF16 and MXFP8 · Forward + Backward

The same schedule is reused in backward.
In MXFP8 mode, dispatch pulls BF16 and quantizes for local expert GEMMs.

MoK schedules data movement and expert math together, in the layout each stage needs.

Routing / schedule preparation and the final weighted-sum epilogue are outside the megakernel. Ring capacity bounds routed activations.

Sources: MoK design blog · scheduler · pull / push kernels · weighted-sum epilogue · configuration

DeepSeek V4 Flash: Full-Model MoK Performance

43-layer pretrained model · 64 GB200 GPUs · End-to-end training

Training stack	Step time	Speedup vs HEP BF16	Configuration
HF + FSDP (OOM)	OOM	—	Out of memory; no valid training-step timing
FSDP + HybridEP (BF16)	9.1640 s	1.000× Baseline	Separate dispatch / expert GEMMs / combine BF16 experts (torch_mm)
+ MoK (BF16)	8.0644 s	1.136×	Fused dispatch + expert compute + combine BF16 experts
+ MoK MXFP8	7.4245 s	1.234×	Same MoK execution path MXFP8 routed-expert GEMMs

Matched MoK benchmark: FineWeb · 8K sequence · Global batch 64 · EP64 / PP1 · 5 warmup + 15 timed steps.

MoK BF16: +13.6% throughput; MoK MXFP8: +23.4% vs HEP BF16.

MXFP8 gives another 1.086× over MoK BF16. Full pretrained weights; MTP disabled.

Sources: merged Automodel PRs #3606 (MoK BF16 / MXFP8), #3422 (MoK integration). HF OOM: reported baseline.

NVIDIA.

Scaling Out w nD parallelism - Combining EP w FSDP

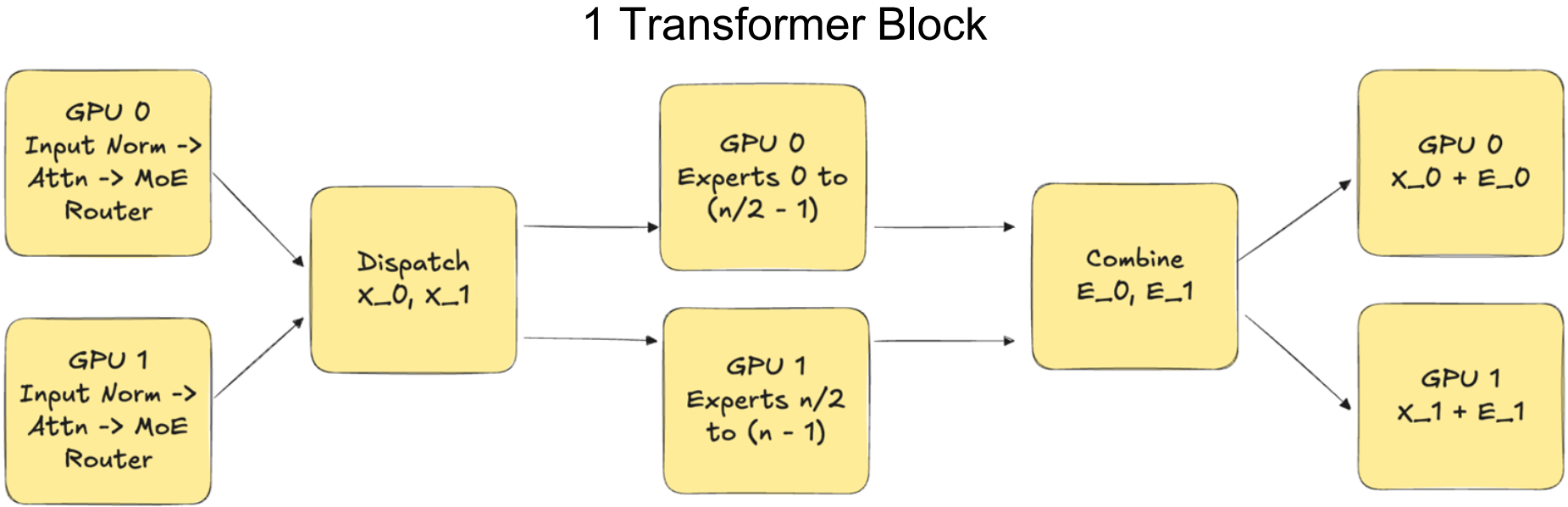
Key Concept: FSDP and EP share the same axis in a DeviceMesh

- FSDP shards non-expert layers (attention, shared FFN)
- EP distributes expert layers
- Requires careful communication scheduling

Benefits:

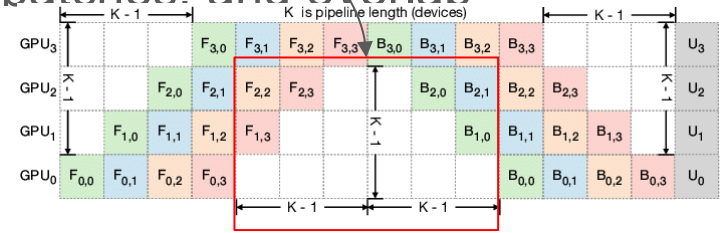
- Better step times (experts computed in parallel)
- Scales to sparser and bigger MoEs (more experts, fewer active)
- Enables training models with hundreds of experts

Shared axis visualization for EP and FSDP



Scaling Out w nD parallelism - Pipeline Parallelism

- PP – pros: enables scaling; cons: utilization bubbles, code complexity
- Used with **700B+ parameter models**, such as DSV4-pro(1.6T), Qwen3.8-2.4T, Kimi-K3 (2.8T) etc
- Splits the model into **sequential stages**; each stage is sharded on its own with FSDP + EP
- Each stage waits on the previous one, so GPUs sit idle while the pipeline fills and drains: **the bubble**
- **1F1B**, interleaved 1F1B, etc schedules shrink the bubble by interleaving forward and backward passes of different microbatches, and overlap activation transfers with compute
- Implemented in Automodel via Autopipeline



Under the Hood Deep Dive on AutoPipeline

Applying Pytorch PP to HF models via `AutoPipeline`

Pipeline Parallel(PP) is important when memory exceeds a single device - by splitting model's layers across devices. `AutoPipeline` in NeMo AutoModel enables Pytorch native PP for any Hugging Face Transformer LLMs models(AutoModelForCausalLM category). (VLMs WIP).

How `AutoPipeline(...).build(model, loss_fn)` works:

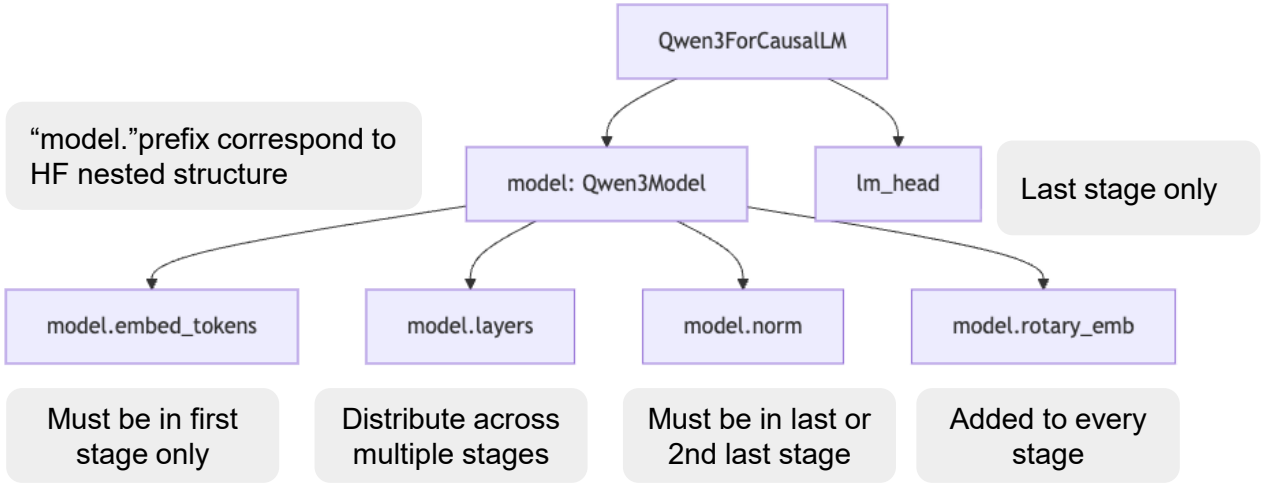
1. **Analyze models** structure – in particular the nn.Module tree
   Frequent model structure [embedding | encoder] → [[transformer + MoE] x depth] → [lm_head | MTP]
2. **Calculate virtual stages** based on `layers_per_stage`
3. **Module assignment** (which layers go to which stage)
4. **Model Splitting** by deep copying the full model for each stage and removed other stages
5. **Stage creation:** Wrap each model chunk in a `PipelineStage`
6. **Parallelization:** Apply additional parallelization if a `parallelize_fn` is provided
7. **Schedule Building**: Create the pipeline schedule (1f1b, interleaved, etc.) with the stages and loss function

Under the Hood Deep Dive on AutoPipeline

Applying Pytorch PP to HF models via `AutoPipeline`

Highlight 1 - The `generate_hf_model_fqn_per_model_part` function in `functional.py` makes sure different components are placed according to its own placement requirements handles this complexity autom

- A model has different components (Input Embeddings, Transformer Layers, Final Normalization, Language Modelling Head, Rotary Embeddings, each component have different placement requirements.
  - Even more pronounced for VLMs and other complex model architectures



Under the Hood Deep Dive on AutoPipeline

Applying Pytorch PP to HF models via `AutoPipeline`

Highlight 2 - Maintaining accurate layer indexing to ensure correct checkpoint loading and saving:

✗ With `nn.ModuleList` - during model splitting across pp stages, each stage automatically reindex its contents starting from 0

```
# Standard HuggingFace pattern
class TransformerModel(nn.Module):
    def __init__(self):
        self.layers = nn.ModuleList([
            TransformerLayer() for _ in range(32)  # layers 0-31
        ])
# After splitting across 4 stages:
# Stage 0 gets: layers[0:8]   -> Re-indexed as layers[0:8]   ✓ Correct
# Stage 1 gets: layers[8:16]  -> Re-indexed as layers[0:8]   ✗ Wrong!
# Stage 2 gets: layers[16:24] -> Re-indexed as layers[0:8]   ✗ Wrong!
# Stage 3 gets: layers[24:32] -> Re-indexed as layers[0:8]   ✗ Wrong!
```

✓ By converting to `nn.ModuleDict` with explicit layer naming

```
elif isinstance(module, nn.ModuleList):
    indices_to_keep = {int(idx) for idx in layers_to_keep if idx.isdigit()}
    new_layers = nn.ModuleDict(
        {str(i): layer for i, layer in enumerate(module) if i in indices_to_keep}
    )
    setattr(parent_module, name, new_layers)

# After conversion and splitting:
# Stage 0: {"0": layer_0, "1": layer_1, ..., "7": layer_7}
# Stage 1: {"8": layer_8, "9": layer_9, ..., "15": layer_15}
# Stage 2: {"16": layer_16, "17": layer_17, ..., "23": layer_23}
# Stage 3: {"24": layer_24, "25": layer_25, ..., "31": layer_31}
```

Under the Hood Deep Dive on AutoPipeline

Applying Pytorch PP to HF models via `AutoPipeline`

Highlight 3 - Patch HF forward methods - Standard HF forward method introduces two problems

(1) Modules are missing when model is split across stages
(2) Each stage should return a single tensor output to be passed to the next stage

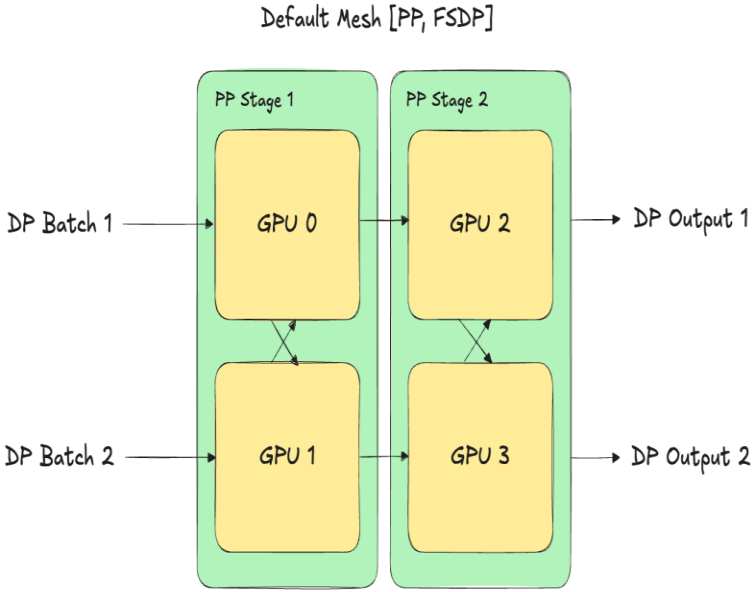
```
# Problem 1: Missing modules
# Stage 1 trying to run standard forward method
hidden_states = self.embed_tokens(input_ids)  # ← AttributeError!
# embed_tokens is None on this stage

# Problem 2: Complex return types
def forward(self, input_ids, **kwargs):
    # ... forward computation ...
    return CausalLMOutputWithPast(  # ← Pipeline expects simple tensor!
        loss=loss,
        logits=logits,
        past_key_values=outputs.past_key_values,
        hidden_states=outputs.hidden_states,
        attentions=outputs.attentions,
    )
```

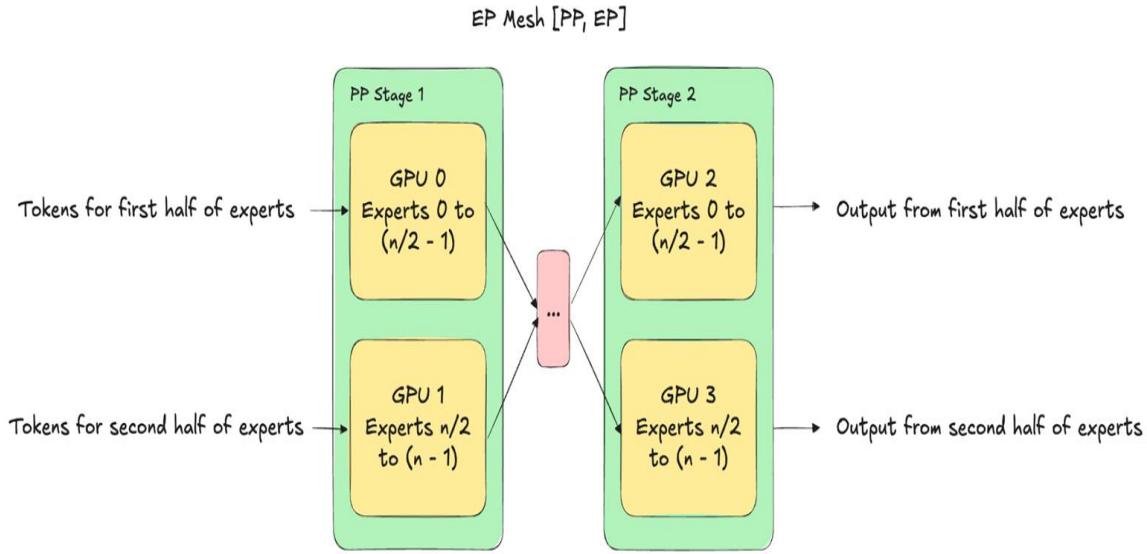
```
# Problem 1 Solution: Intelligent module detection
if hasattr(self, "embed_tokens") and self.embed_tokens is not None:
    hidden_states = self.embed_tokens(input_ids)#First stage
else:
    hidden_states = input_ids  # Middle/last stage - already embeddings

# Problem 2 Solution: Pipeline-aware return types
def pipeline_forward_causal_lm(self, input_ids, **kwargs):
    # Get outputs from the inner model
    outputs = self.model(input_ids=input_ids, **kwargs)
    hidden_states = outputs if isinstance(outputs, torch.Tensor) else outputs.last_hidden_state
    # Apply language modeling head if it exists on this stage
    if hasattr(self, "lm_head") and self.lm_head is not None:
        logits = self.lm_head(hidden_states)
        return logits  # ← Simple tensor for pipeline
    else:
        return hidden_states  # Pass to next stage
```

Scaling Out w nD parallelism - FSDP + EP + PP



Default Mesh for non-expert layers

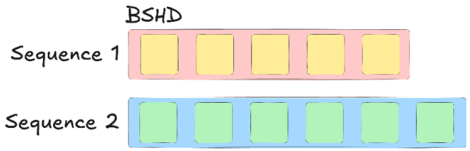


MoE Mesh for expert layers

Scaling Out w nD parallelism - Context Parallelism

- Necessary for long sequence lengths as activation memory quickly runs out of capacity with increasing sequence length
  - FA mem space complexity O(N); compute O(N**2)
- Implemented through Transformer Engine's DotProductAttention which internally implements various context parallel techniques
- Key considerations for the practical implementation include formatting and passing in the correct input format (Visualization in next slide)

Context Parallelism - Input visualization

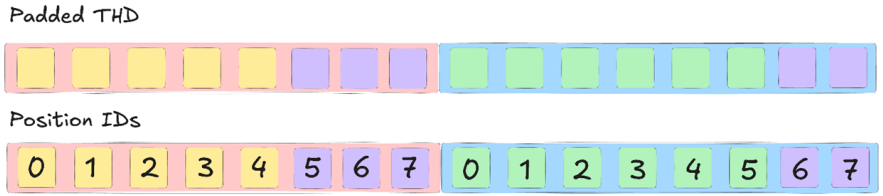


Legend
B - Batch Size
S - Sequence Length
H - Number of Heads
D - Head size

THD args
cu_seqlens: [0, 5, 11], max_seqlen: 6

Context Parallelism Constraints: Each seq needs to be padded to be a multiple of 2 x cp_size

Let's consider a cp_size of 2, so each sequence needs to be a multiple of 4
The padded THD sequence then becomes:



Splitting Input for CP:
- Split each sequence separately
- Split in zig-zag fashion

Context Parallelism with Sparse Attention: Deepseek V4

DeepSeek-V4-Flash · Compressed Sparse Attention (CSA) · AM / TileLang · 16,384 tokens / CP = 4

1 · Split the sequence into contiguous query shards

CP rank 0
Q[0 : 4096] stays local
Produce local KV + compressed states

CP rank 1
Q[4096 : 8192] stays local
Produce local KV + compressed states

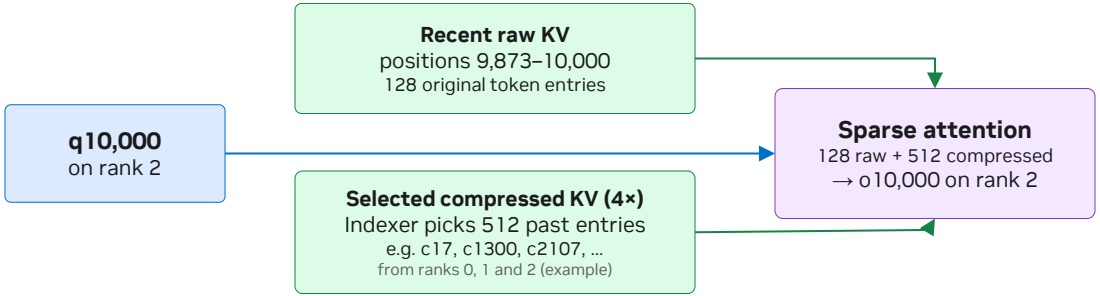
CP rank 2
Q[8192 : 12288] stays local
Produce local KV + compressed states

CP rank 3
Q[12288 : 16384] stays local
Produce local KV + compressed states

2 · All-gather the KV-side tensors

Raw KV (16,384) + compressed KV (4,096) + indexer keys (4,096) → available on every rank

3 · Rank 2 computes attention only for its local queries



Repeat for all 4,096 query rows in rank 2's [8192, 12288) shard.

What Each CP Rank Does

Keep Q and output local

4,096 query rows on rank 2.
Output keeps the same local token rows.

Select keys per query

Indexer scores the global compressed-key pool.
Causality removes future entries; top-k can choose entries originating on any rank.

Backpropagate to KV owners

Autograd sums gradients from all query shards back to the original KV owners.

CP splits Q and outputs; sparsity limits the KV entries read by each query.

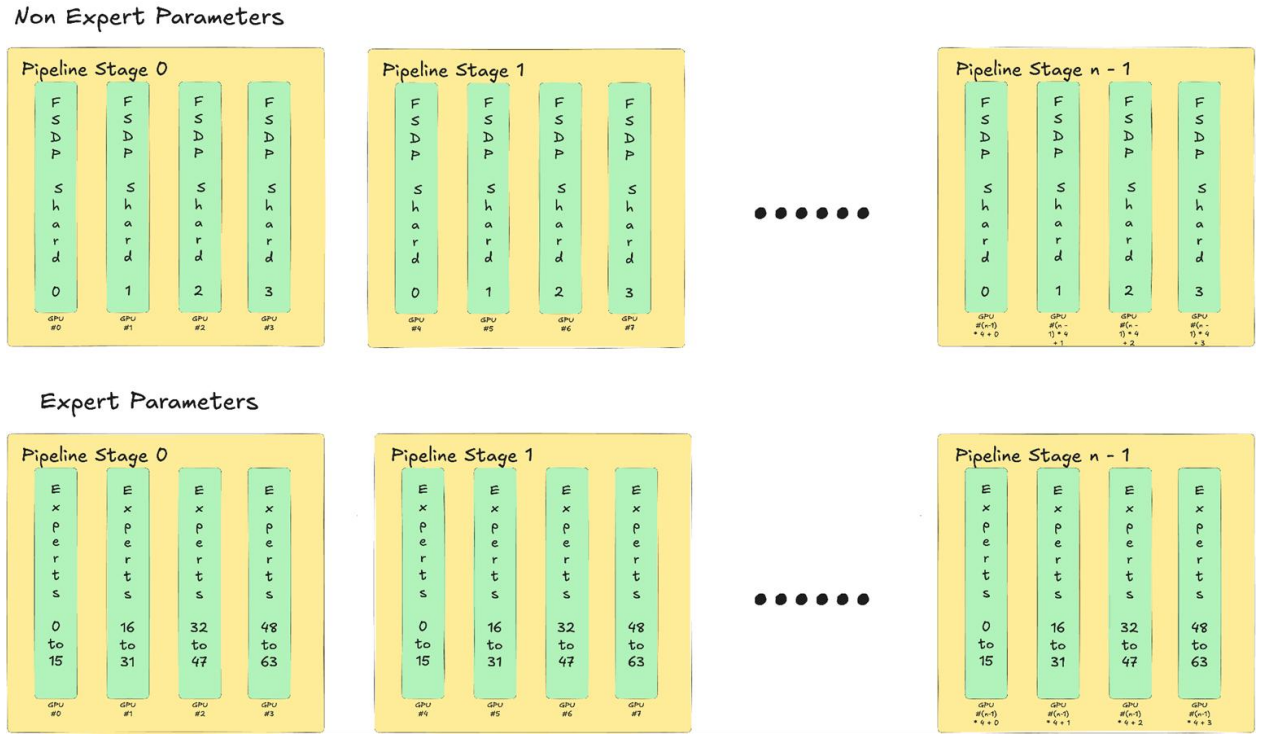
AM gathers KV-side tensors before top-k selection. HCA layers use 128× compression and all past compressed entries.

Sources: DeepSeek-V4-Flash config · AM CP helpers · AM attention / indexer · sparse-index construction

NVIDIA

Scaling Out w nD parallelism - Order

1. Split the model into stages based on PP size via `AutoPipeline`
2. Apply CP via TE's `DotProductAttention` or model's own CP implementation
3. Apply EP via `GroupedExperts`
4. Apply activation checkpointing
5. Apply FSDP with `default` mesh and `moe` mesh



n*4 GPUs with PP = n, fsdp = 4, ep = 4

Kimi K3: Training Performance Improvements

Full 2.78T K3 · 256 GB200 GPUs (64 nodes) · PP8 / EP32 · 2K synthetic benchmark

Optimization area	What changed	Reported impact	MFU (before → after)
Memory / PP buffers	Save BF16 inputs; recompute FP32 math in chunks Reuse 1F1B receive buffers as a ring	PP buffers: ~45 GiB saved on middle stages Enables 2K rows × global batch 4096	9.8% → 13.6%‡ 1K → 2K + pool / norm
GPU kernel fusion	Compile SiTU / router-weight / RMSNorm chains Triton SiTU and attention-residual kernels	Compile islands +11.7%*; SiTU +5.3%* Attention-residual fusion +6.6%	18.7% → 19.9% Attention-residual A/B
KDA kernels	CUDA short conv; avoid duplicate recomputation Fused CUDA forward + Triton backward	KDA knobs +1.66%; fused KDA +2.04% Fused KDA: −1.5 GiB per layer-microbatch	Knobs: 20.2% → 20.5% Fused: 20.7% → 21.1%
Host sync / overlap	Reuse static metadata; bounded-capacity dispatch Overlap shared experts on a side stream	Static pad pin +0.65%; shared overlap ≈ neutral Dynamic dispatch +8.5%†	Pad pin: 18.6% → 18.7% Overlap: ≈ unchanged
Runtime stalls	Run Python garbage collection every 10 steps	+8.4% throughput (same-job A/B) Iteration spread: 1.7% → 0.1%	14.5% → 15.8%‡
Startup	Compile pipeline stages concurrently Share HybridEP JIT cache across processes	Iteration 0: 607 → 71 s Separate prewarm: 72 → 13 s; steady-state unchanged	Unchanged Startup only

Full-model results: synthetic data + balanced routing. †+8.5%: 32-GPU, 16-layer model with dynamic routing.

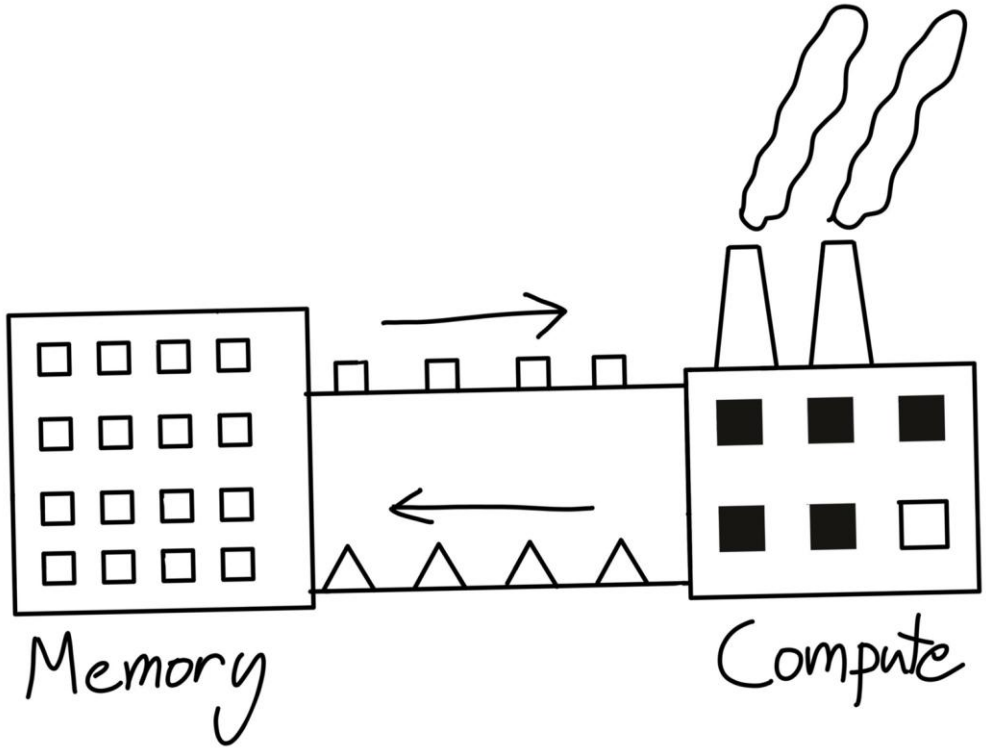
Latest reported stack: 839.6 tokens/s/GPU · 21.1% recipe MFU.

Independent A/Bs; gains are not additive. *Cross-job. MFU: PR-reported, 2.5 PFLOP/s GB200 peak; ‡rescaled from 2.25 PFLOP/s.

Yisong Li · PRs #3731, #3732, #3779, #3780, #3792, #3808, #3893–#3896, #3931, #3932, #3954.

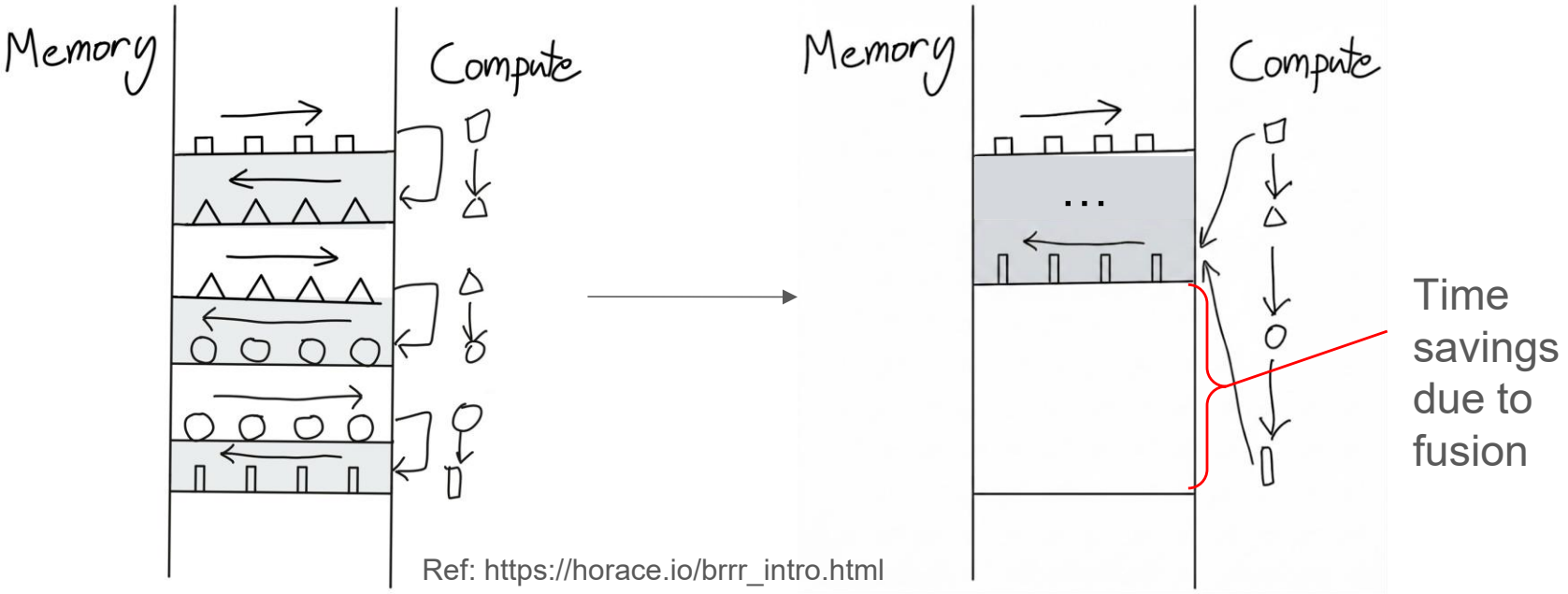
NVIDIA.

Recap - Mental Model



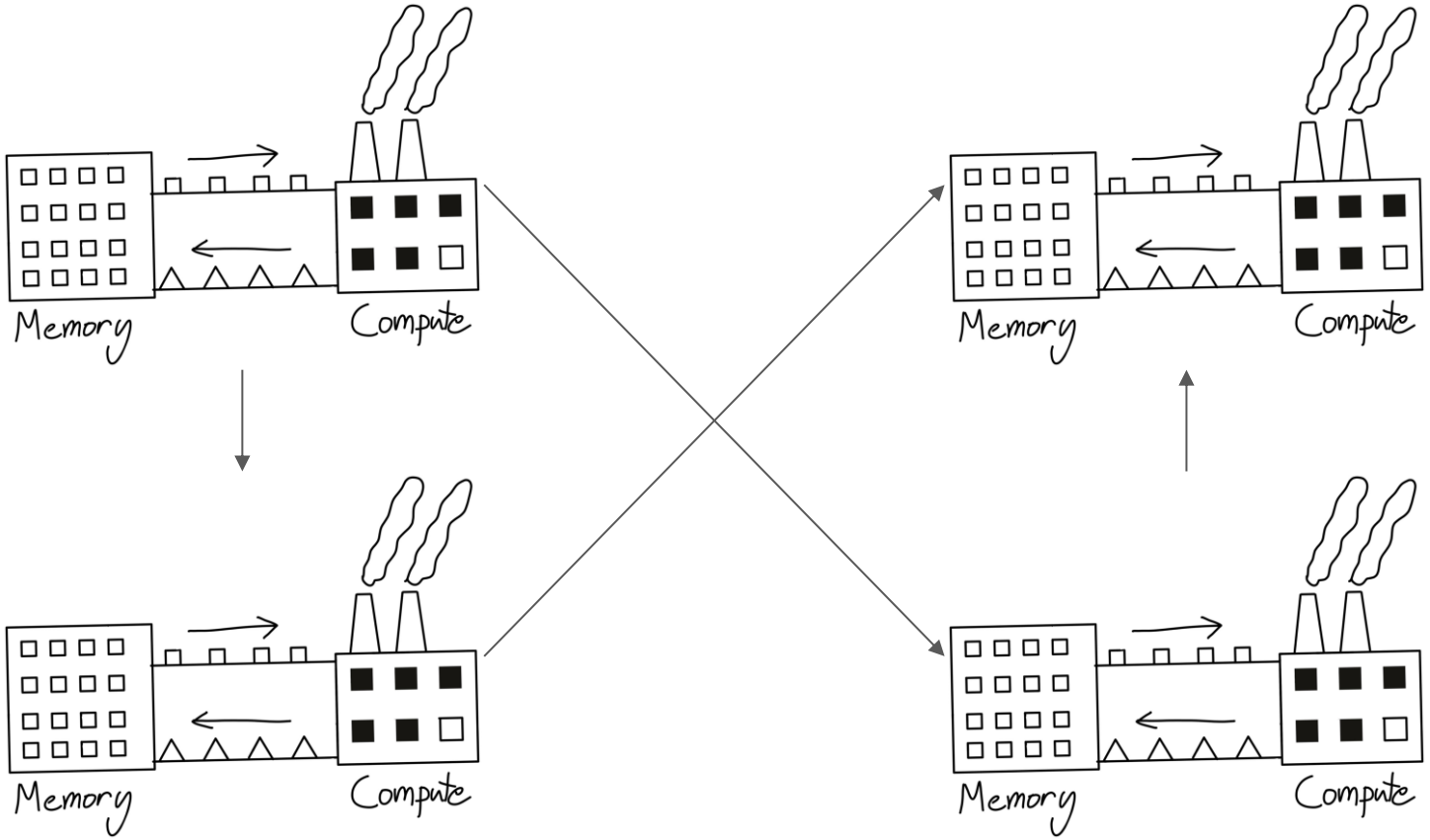
Ref: https://horace.io/brrr_intro.html

Recap - Kernel Level Optimizations



Ref: https://horace.io/brrr_intro.html

Recap - Scaling Out w nD parallelism



Ref: https://horace.io/brrr_intro.html

Verification Testing

- Numerical equivalence: Compare activations **and** gradients of each layer against reference (OOTB) implementation to ensure within tolerance
- Loss curve validation: Training loss should match reference on same data
- Downstream task validation Evaluate on benchmarks after training

Takeaways

- Optimize at the kernel level using a combination of torch.compile and specialized kernels
- Start small and scale out with nD parallelism for large models and larger sequences
- Performance and correctness go hand in hand
- Always read the source code

Acknowledgement

All the work presented in this talk is the culmination of efforts of a variety in NVIDIA as well as the OSS community, and we sincerely thank all of them for making such frameworks like Automodel possible, and aim to contribute as much as possible back to the community.

Thank You!

Code: https://github.com/NVIDIA-NeMo/Automodel

Blogs:

https://huggingface.co/blog/nvidia/accelerating-fine-tuning-nvidia-nemo-automodel

https://huggingface.co/blog/nvidia/scale-diffusers-finetuning-nemo-automodel

Implementing large scale MoE training with native Pytorch in Nemo Automodel

Accelerating-large-scale mixture-of-experts training in Pytorch

Challenges in Enabling PyTorch Native Pipeline Parallelism for Hugging Face Transformer Models

Fine-tune Hugging Face Models Instantly with Day-0 Support with NVIDIA NeMo AutoModel