DeepSeek Open Infra

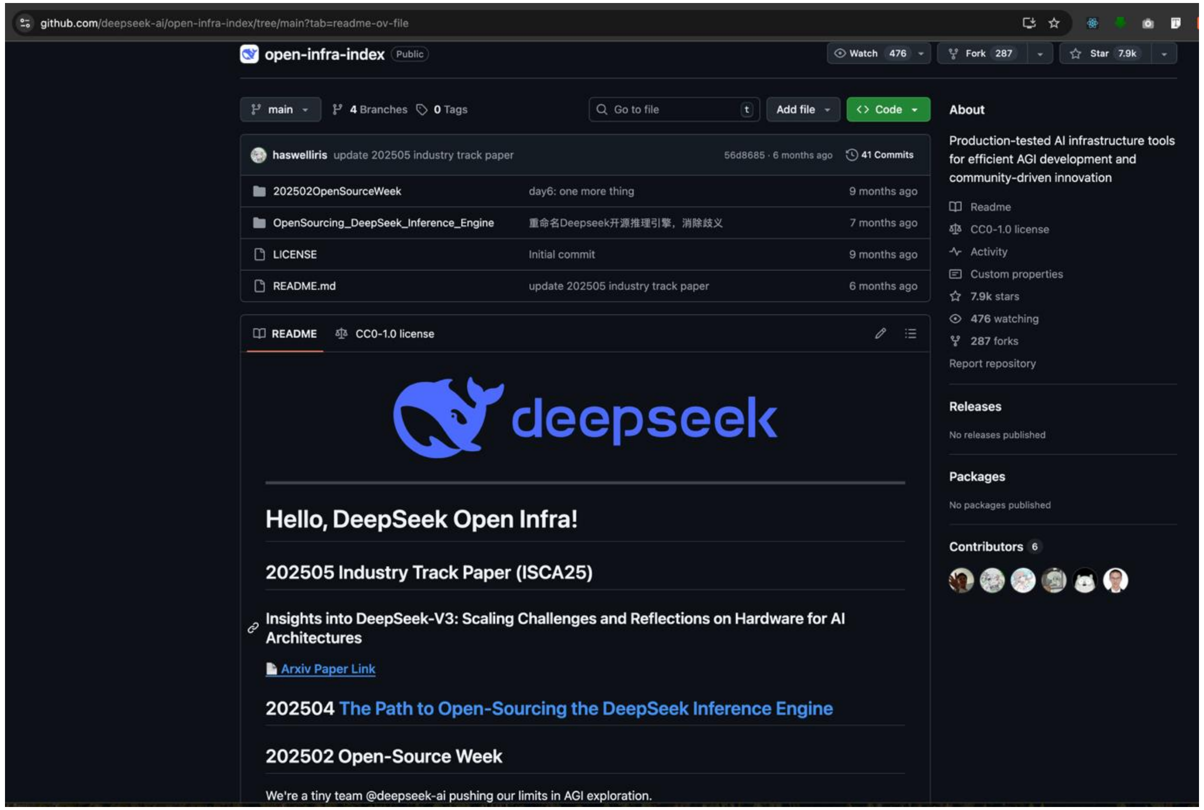
Presenter: Sudarsan Srivathsun

DeepSeek V3 Introduction

Model	GPUs	GPU Model
GPT-4	~25,000+	**NVIDIA A100 (80GB) SXM**
Claude 3	~20,000+	**NVIDIA H100 (plus some AWS Trainium)**
Qwen 2.5	~10,000+	**NVIDIA H800**
LLaMA 3.1	~16,000+	**NVIDIA H100**
DeepSeek-V3	**2,048**	**NVIDIA H800**

DeepSeek-V3 matched or exceeded many of these results with just 2,048 NVIDIA H800 GPUs

Deepseek Open Infra



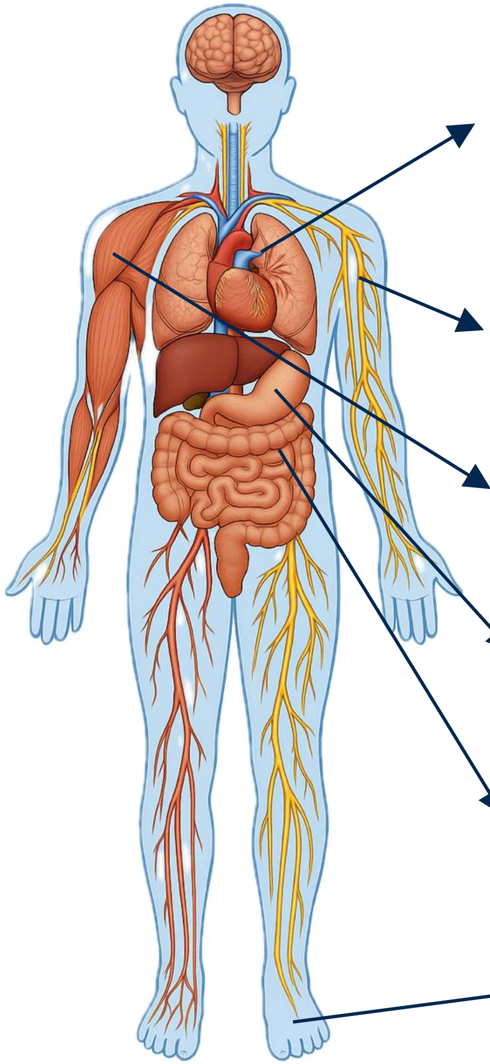
- **A fully open-source infrastructure stack powering DeepSeek-V3/R1**
  Built from real production systems, exposing the exact compute kernels, communication libraries, and data-processing tooling used inside DeepSeek's large-scale inference and training pipelines.

- **Designed for efficiency on commodity H800 clusters**
  Includes ultra-optimized components (FlashMLA, DeepGEMM, DeepEP, DualPipe, EPLB, 3FS) that drastically reduce cost-per-token by maximizing GPU utilization, overlapping compute/communication, and removing bottlenecks.

- **Modular, transparent building blocks—not vaporware**
  Each repo is small, focused, and high-quality: no monolithic framework, but a `set of standalone components you can pick, reuse, or study independently (kernels, parallelism strategies, RDMA/NVLink communication, file systems, etc.).

- **Built for the community to learn from real AGI-scale engineering**
  DeepSeek open-sourced actual production code—not demos—letting researchers and engineers study real kernels, real networking patterns, and real parallelism algorithms used at massive scale.

Deepseek Open Infra



Heart: Compute Layer
FlashMLA kernels
DeepGEMM FP8 GEMM kernels
Attention kernels, Quantization kernels
Activation & normalization kernels

Nerves: Communication/IO Layer
(DeepEP, RDMA, NVLink, DualPipe, EPLB, Smallpond, 3FS)

Muscles: Distributed Training Engine
(DeepEP, DualPipe, EPLB, Parallelism utilities)

Stomach: Runtime & Serving
(KV cache, token streaming, speculative decode)

Intestines: Utilities & profiling Benchmarks (MLA, GEMM, EP), Profilers (CUDA, GPU, network), Memory profiler, Debugging tools

Bones: Hardware Infrastructure
(H800 Mesh, Multi-Plane Network, NVLink Mesh, RDMA Fabric)

COMPUTE LAYER
- FlashMLA
- DeepGEMM

Distributed Training Layer
- DeepEP
- DualPipe
- EPLB
- Smallpond

Communication / IO Layer
- DualPipe
- RDMA / NVLink Kernels
- DeepEP
- Smallpond

Runtime Layer
- KV-cache lookup (3FS)
- Prefill/Decode kernels (DeepEP)

Tooling/Profiling Layer
- Benchmarks (MLA/GEMM/EP)
- Profilers

Hardware Layer
- H800 Mesh, NV Link Mesh, RDMA Fabric
- Multi-plane network

Compute Layer – Generic Terms

A few terms so the following makes sense:

- **Q, K, V**:
  - `q` = query vectors (current tokens we're decoding)
  - `k`, `v` = key/value vectors from past tokens (stored in the **KV cache**)
- **KV cache**:
  - Memory of previous tokens for each sequence in the batch.
  - Organized in **blocks / pages** (`block_table`, `page_block_size`) so we don't store one giant contiguous tensor per sequence.
- **Variable length / varlen**:
  - Different prompts in a batch can have different lengths.
  - Need offsets like `cu_seqlens_*` (cumulative sequence lengths) to know where each sequence starts/ends in a flat tensor.
- **Sparse vs dense attention**:
  - **Dense**: every query attends to all previous keys.
  - **Sparse**: each query only attends to a subset (top-k positions) given by `indices`.
  - FlashMLA can read KV cache in FP8 (`is_fp8_kvcache=True`).
- **BF16 / FP8**:
  - **BF16 (bfloat16)**: 16-bit floating point, good dynamic range, used widely for training/inference.
  - **FP8**: 8-bit float, even smaller, faster and more memory-efficient (but trickier to use).

Compute Layer – Flash MLA

K = [k1, k2, k3, ..., k4096]   (4096 floats)

V = [v1, v2, v3, ..., v4096]   (4096 floats)

4096 + 4096 = 8192 floats

If stored in BF16 (2 bytes per float) → **16 KB per token**

If you have a sequence of **32,000 tokens** (32k context):
→ 32,000 × 16 KB = **512 MB KV cache**
for one layer! multiply by ~30 for a full model
This becomes a **memory disaster**.

Let: **K** is a vector of size (4096 × 1)
Wk_proj is a projection matrix of size (512 × 4096)
latent_K of size (512 × 1)
Same for V

Multi-Head Latent Attention:

latent_K = Wk_proj * K   → 512 floats
latent_V = Wv_proj * V   → 512 floats

latent_K = [512 floats]
latent_V = [512 floats]
Total = 1024 floats
At BF16 (2 bytes) = 2048 bytes = 2 KB

Compare this to traditional **16 KB → 8× reduction**.
DeepSeek claims **70 KB/token compressed**, including all other internal buffers.

Flash Multi-Head Latent Attention:

reconstructed_K = Wk_decode * latent_K
reconstructed_V = Wv_decode * latent_V

FlashMLA is a high-performance CUDA kernel that implements the decoding step of MLA's latent attention mechanism

Analogy:

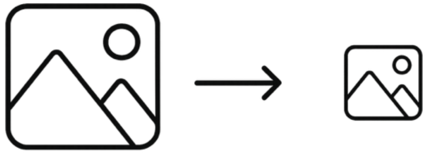


Image → Thumbnail

Compute Layer – Flash MLA

FlashMLA is a highly optimized CUDA attention kernel—exposed to PyTorch as a custom operator, that implements DeepSeek's Multi-Head Latent Attention (MLA) for Hopper GPUs (H100/H800), enabling extremely fast variable-length decoding by efficiently reading compressed KV-cache pages directly on GPU.

Architecture:

Input From Model
Model produces Q(queries for ctokens and store past K/V in a compact KV-c (latent space)

→ **PyTorch Wrapper**
Exposes functions to Python Code, calls C++ binding layer

→ **Get Metadata**
Calculate tile scheduling and splits

→ **Launch CUDA Kernel**
Run the main kernel on GPU

→ **(optional) Handle Sparse Attention**
Attend only to top-k tokens

→ **Suport Var-Length Sequences**
Pack sequences efficiently, Avoid Padding

→ **Backward Pass for Training**
Compute gradients by a custom kernel

Compute Layer – Flash MLA

FlashMLA = optimized implementation of MLA decoding

- Written for **Hopper GPUs** (H100 / H800)
- Extremely fast kernel for reconstructing from latent space
- Works with **variable sequence lengths** (important for real workloads)
- Supports **BF16**
- Works with **paged KV cache** (block size 64)

DeepSeek reports:

- **3000 GB/s memory-bound throughput**
- **580 TFLOPS BF16 compute throughput**
- On **H800**, which is bandwidth-limited (this is insane efficiency)

Why This Is So Important for DeepSeek

- Because DeepSeek trains on **H800 GPUs**, which have:
- Slow NVLink bandwidth
- Slow memory throughput
- MLA + FlashMLA dramatically reduce:
- Memory usage
- Memory bandwidth needed
- Communication volume
- Without MLA, **MoE would be too slow** on H800 GPUs.

✔ **Why it matters**

- FlashMLA makes MLA **practical at scale**:
- Faster than regular attention
- Much smaller KV memory
- Lower inference latency
- Huge cost savings
- This is literally the *engine* behind DeepSeek-V3's long-context efficiency.

Compute Layer – DeepGEMM

DeepGEMM exists because **LLMs are dominated by one operation**:

MATRIX MULTIPLICATION (GEMM)

This operation accounts for **80–95%** of compute in transformers — especially in:

- attention projections
- feed-forward networks
- MoE experts
- QKV projections
- output layers

If your GEMM is slow → **your entire model is slow**.
If your GEMM is inefficient → **your training cost explodes**.

Compute Layer – DeepGEMM

NVIDIA provides cuBLAS, Transformer Engine, etc.
But these libraries:

❌ **are optimized for general workloads, not LLM-specific shapes**
LLMs have very specific GEMM shapes:

- Very tall/skinny matrices
- MoE-style grouped matrices
- FP8/low-precision projections
- Irregular batch sizes
- Dynamic shapes at decode time

❌ **Problem 1: FP8 is powerful, but hard to implement**
NVIDIA's Transformer Engine supports FP8,
but not in the ultra-specialized MoE or MLA ways DeepSeek needs.

❌ **Problem 2: MoE Makes Everything Harder**
Without special kernels → MoE becomes slow and communication-bound.

DeepSeek solved this with DeepEP (for routing) *and* DeepGEMM (for compute).

❌ **Problem 3: Hopper GPUs need hand-optimized kernels**
Generic libraries do not give DeepSeek the **absolute maximum throughput** needed to hit high efficiency.

Compute Layer – DeepGEMM

DeepGEMM is nxeeded because default GPU libraries cannot deliver the ultra-specialized, FP8-optimized, MoE-aware, Hopper-tuned matrix multiplication performance required for DeepSeek-V3.

It solves the bottleneck at the heart of every LLM: **fast, stable, low-precision GEMM**, enabling DeepSeek to train a 671B-parameter model cheaply on 2,048 H800s.

Model produces A and B
Transformer layer outputs matrices that need fast multiplication.

PyTorch calls DeepGEMM
Python code simply passes these matrices into DeepGEMM.

C++ binding layer dispatches to GPU
Validates tensors and sends raw pointers to CUDA kernels.

FP8 scaling + metadata setup
DeepGEMM calculates scaling factors so FP8 math stays accurate.

JIT compiles best FP8 kernel
Attend only to top-k tokens

Custom CUDA kernel computes A × B
Pack sequences efficiently, Avoid Padding

MoE layout handled if needed
Only the active MoE experts are multiplied → huge compute savings.

Output written to GPU memory
C = A×B is stored for the next transformer layer.

Output written to GPU memory
C = A×B is stored for the next transformer layer.

Compute Layer – DeepGEMM

• Major Speedup for All Linear Layers
LLMs spend 80–90% of compute on matrix multiplication, DeepGEMM makes them 2–3× faster, hitting **1000–1350+ FP8 TFLOPS** on H800 (much faster than NVIDIA's Transformer Engine).

• Core to DeepSeek's Low-Cost Training
Faster GEMM = fewer GPU hours. FP8 cuts compute + memory traffic, enabling DeepSeek-V3 to train on **2,048 H800s** instead of tens of thousands of H100s.

• Purpose-Built for MoE Models
Custom expert-parallel kernels activate only needed experts → massive FLOP & memory savings for DeepSeek-V3's **671B-parameter MoE**.

• Optimized for Real LLM Shapes
JIT-tuned kernels match exact transformer matrix sizes → better hardware utilization than generic GPU libraries.

• Lower Memory & Bandwidth Load
FP8 + optimized layouts reduce HBM/NVLink traffic — crucial for the **bandwidth-limited H800** cluster.

• Lightweight & Easy to Use
Minimal, clean code (~300 core lines) and simple PyTorch API → drop-in GEMM replacement.

Why DeepGEMM Matters:

- LLMs spend ~80–90% of training compute on GEMM. Making GEMM faster = making the entire model faster.
- Crucial for DeepSeek's low-cost training strategy. Efficient GEMM means fewer GPUs and lower electricity cost.
- Optimized for MoE models, unlike generic GPU libraries. DeepSeek-V3 uses MoE heavily → DeepGEMM is purpose-designed.

Compute Layer

Let's follow a single new token as it moves through a DeepSeek-V3 layer.

Step 1: Model produces a new hidden vector h
The transformer layer outputs a hidden embedding for the current token.

Step 2: DeepGEMM computes Q/K/V through fast FP8 matrix multiplies
Each Transformer layer needs **Query (Q)**, **Key (K)**, and **Value (V)** vectors. DeepGEMM replaces standard GEMM to compute Q = hWq, K = hWk, V = hWv.

Step 3: Build / update the KV cache
The model stores K/V for future attention, often in MLA compressed format.

Step 4: Run Attention over Q vs KV cache → FlashMLA
Fast CUDA attention kernel computes softmax(QK^T)V to get the context vector.

Step 5: Residual + normalization combines context with original h
Attention output blends back into the model via residual connection.

Step 6: DeepGEMM runs MoE/MLP layers through massive FP8 GEMM operations
Router + Expert MLP projections rely heavily on DeepGEMM FP8 kernels.

Step 7: DeepGEMM computes the final output projection (logits)
logits = h_final @ W_vocab also runs on DeepGEMM.

DeepEP (Primary – Communication layer)

DeepEP is DeepSeek’s high-performance all-to-all communication engine for MoE models, enabling fast, GPU-driven data exchange between experts during training and inference. It removes CPU bottlenecks and maximizes throughput on H800 clusters.

MoE routing selects experts per token
MoE routing selects experts per token

→

Python Wrapper Dispatches → C++ Binding Layer
Python code simply passes these matrices into DeepGEMM.

→

Expert Dispatch Metadata Preparation
DeepEP computes internal metadata needed for MoE communication

↓

Launch CUDA “Dispatch” Kernel (Token → Expert Sharding)
Tokens destined for remote experts are reorganized into contiguous GPU buffers.

←

Launch High-Performance All-to-All Communication
Each GPU sends its token batches to the GPUs hosting the corresponding experts.

←

Expert Compute Happens (DeepGEMM or Other Ops)
Each GPU receives expert-specific batches and runs FFN/MLP using DeepGEMM (FP8) or another compute kernel.

↓

Launch CUDA “Combine” Kernel (Expert Output → Token Order)
Outputs from each expert are reassembled back into the original token order.

→

Return to PyTorch Execution
C = A×B is stored for the next transformer layer.

DeepEP (Primary – Communication layer)

Problems it solves:

- Mixture-of-Experts requires every GPU to exchange token data with every other GPU. DeepEP removes this bottleneck with custom GPU-driven communication
- Eliminates CPU involvement by enabling GPU-side RDMA for ultra-low-latency transfers.
- Reduces cross-node traffic with expert-aware routing and token sharding.
- Overlaps communication (by scheduling chunks) with computation to prevent GPU idle time.
- Handles both intra-node (NVLink) and inter-node (InfiniBand) communication efficiently.
- Allows MoE training to scale efficiently even on bandwidth-limited H800 GPUs.

DeepEP (Primary – Communication layer)

Why It Matters for DeepSeek:

- DeepGEMM and MLA help reduce compute & memory cost; DeepEP ensures the communication layer does not undo those savings by becoming a bottleneck.
- Eliminates CPU involvement by enabling GPU-side RDMA for ultra-low-latency transfers.
- Reduces cross-node traffic with expert-aware routing and token sharding.
- Overlaps communication (by scheduling chunks) with computation to prevent GPU idle time.
- Handles both intra-node (NVLink) and inter-node (InfiniBand) communication efficiently.
- Allows MoE training to scale efficiently even on bandwidth-limited H800 GPUs.

DeepEP Capabilities:

- Efficient and optimized all-to-all communication
- Both intranode and internode support with NVLink and RDMA
- High-throughput kernels for training and inference prefilling
- Low-latency kernels for inference decoding
- Native FP8 dispatch support
- Flexible GPU resource control for computation-communication overlapping

Distributed Training layer - DualPipe

Why it exists?

In traditional pipeline parallelism (Megatron-LM, DeepSpeed):

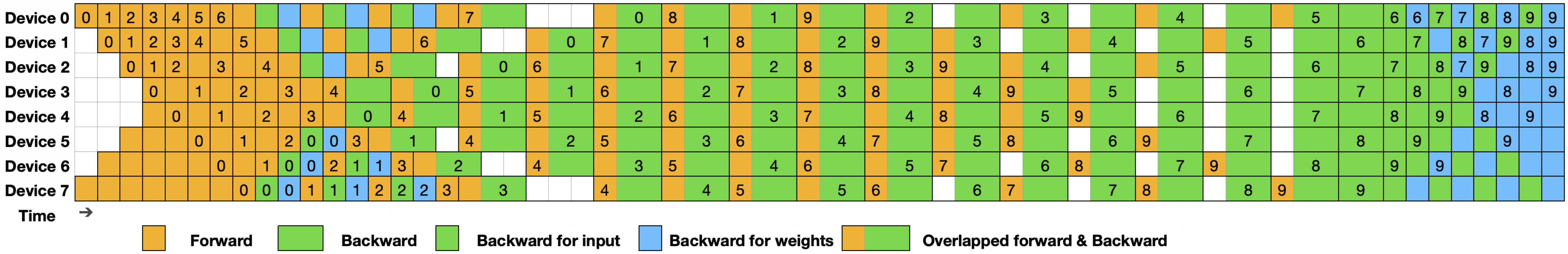
- You run the forward pass across devices
- THEN the backward pass returns
- GPUs wait often → **pipeline bubble**
- Communication (dispatch/combine, gradient exchange) blocks compute

DualPipe fixes this by running forward/backward in opposite overlapping directions + overlapping communication with compute.

DualPipe is DeepSeek's optimized pipeline-parallel training algorithm that runs two pipelines at the same time — one mostly forward → backward and one backward →f orward — so that GPU compute and communication overlap almost perfectly.

DualPipe eliminates pipeline bubbles by running forward and backward passes simultaneously and overlapping all communication with compute — maximizing GPU utilization.

Distributed Training layer - DualPipe



DualPipe (Primary – Communication layer)

Benefits:

- Higher GPU utilization (compute + comm overlap).
- Massively reduced pipeline bubbles in large-scale MoE training.
- Less sensitivity to network bottlenecks.
- Directly reduces training time and cost for DeepSeek-V3/R1 clusters.

Why This Matters for DeepSeek
DualPipe is critical because DeepSeek-V3 uses massive model parallelism.

Without DualPipe:
- GPUs stall during pipeline transitions
- Backward pass blocks forward pass
- Training cost increases
- Token throughput collapses

With DualPipe:
- Every GPU runs forward + backward simultaneously
- Pipeline bubbles disappear
- Training becomes significantly faster & cheaper

Distributed Training layer - EPLB

EPLB (Expert-Parallel Load Balancer **) is DeepSeek's load-balancing system for Mixture-of-Experts (MoE) layers.** It dynamically redistributes expert workloads across GPUs so no GPU becomes overloaded.

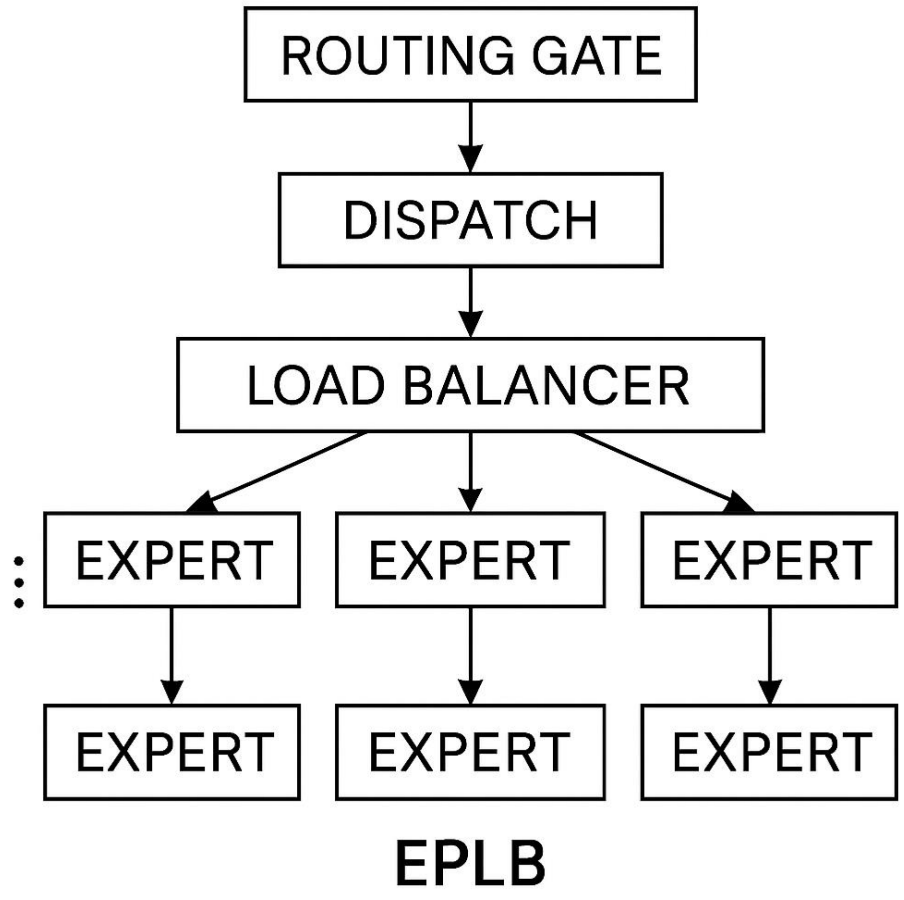
Why EPLB Is Needed (Problem It Solves)

- MoE models route tokens to experts, but **token distribution is uneven**:
- Some experts get many tokens → **overloaded GPU**, slow step
- Some experts get few tokens → **under-utilized GPU, wasted compute**
- Causes **stragglers**, bottlenecks, and low hardware efficiency
- EPLB solves this by **balancing the number of tokens per expert across GPUs** in real time.

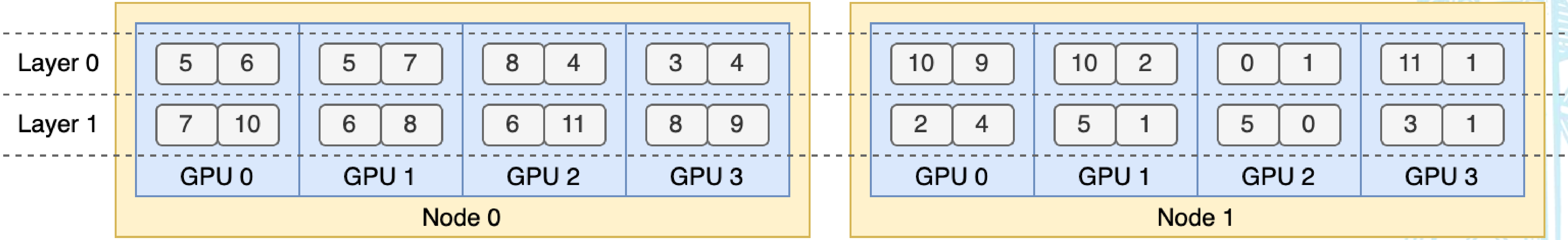
Why EPLB Matters for DeepSeek?

- **Up to 2–3× higher expert utilization**
- **Eliminates bottleneck GPUs** → higher cluster throughput
- **Lower training cost** because fewer GPUs sit idle
- **Stable scaling** to 64–256+ experts
- **Essential for V3's massively parallel MoE design**
- DeepSeek-V3 is extremely MoE-heavy, so **good load balancing = huge efficiency gains**.

Distributed Training layer - EPLB



Distributed Training layer - EPLB



- Each small box is a token load assigned to an expert (the number = how many tokens that expert must process).
- Each big blue box is a GPU, containing several experts.
- The two yellow regions are Node 0 and Node 1 (each with 4 GPUs).
- The diagram shows load imbalance: some GPUs get heavy expert workloads, others get very little.
- This imbalance slows down MoE training because the slowest GPU becomes the bottleneck.
- EPLB (Expert-Parallel Load Balancer) fixes this by redistributing experts across GPUs/nodes so every GPU has similar load, reducing stalls and communication.

Runtime Layer – 3FS

3FS (Fire-Flyer File System) is DeepSeek's **high-performance, parallel distributed file system** designed to feed massive AI training and inference pipelines with extremely fast data access using **SSD + RDMA + multi-node parallelism**.

Why EPLB Is Needed (Problem It Solves)

Training DeepSeek-V3/R1 hits a major bottleneck:

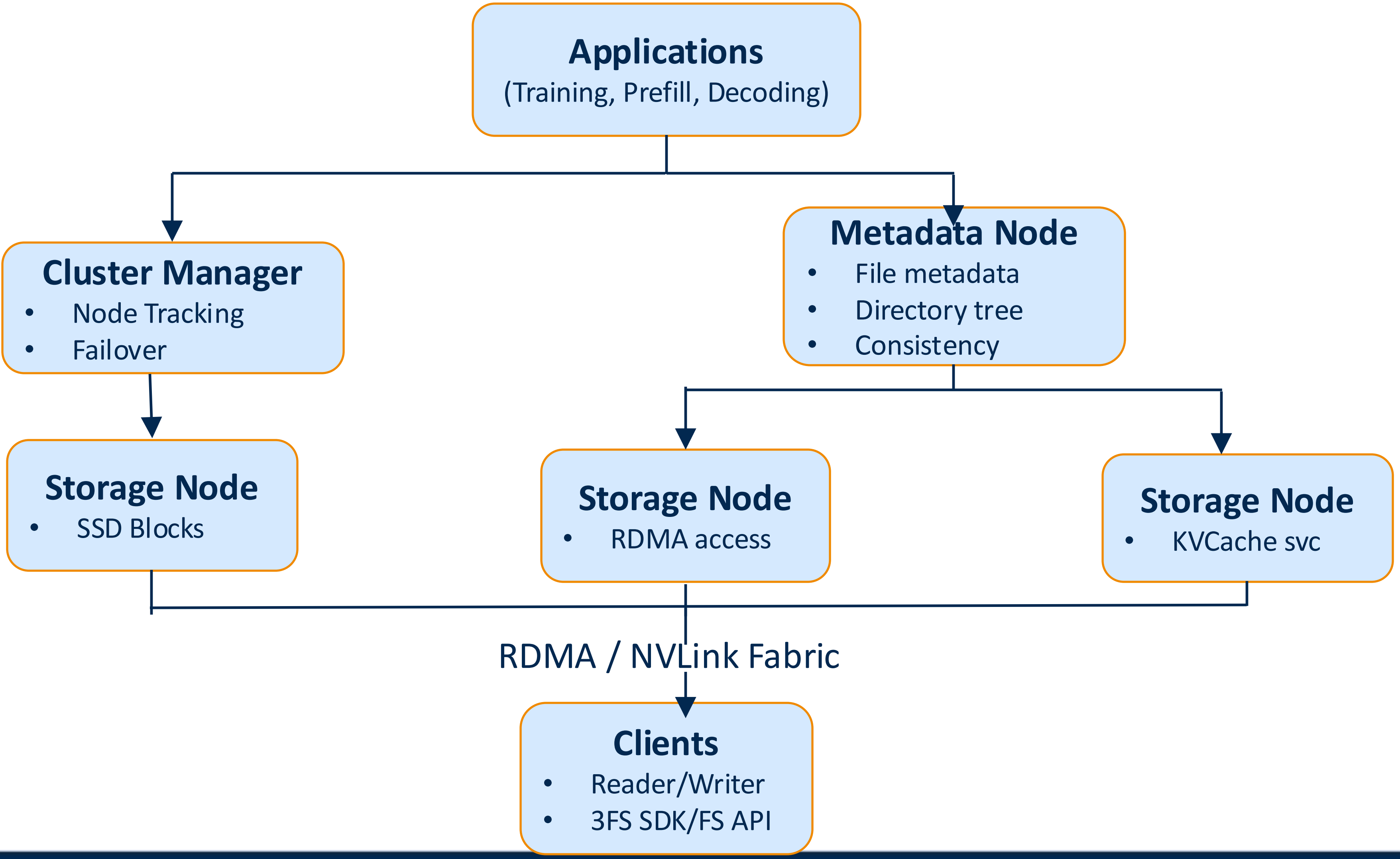
- LLMs require **petabytes of training data** and huge KV-cache traffic
- Storage I/O becomes the **#1 limiter** in large-scale clusters
- Existing filesystems (NFS, Lustre, Ceph) cannot deliver the **bandwidth-per-node needed**
- MoE models require fast access to embeddings, checkpoints, and KV-cache lookups

3FS solves this by becoming a **fast, distributed, RDMA-powered data highway**.

Why EPLB Matters for DeepSeek?

- **Up to 2–3× higher expert utilization**
- **Eliminates bottleneck GPUs** → higher cluster throughput
- **Lower training cost** because fewer GPUs sit idle
- **Stable scaling** to 64–256+ experts
- **Essential for V3's massively parallel MoE design**
- DeepSeek-V3 is extremely MoE-heavy, so **good load balancing = huge efficiency gains**.

Runtime Layer – 3FS



Runtime Layer – 3FS

Benefits:

- Eliminates data loading bottlenecks for large-scale MoE models
- Feeds GPUs at full speed, preventing idle time
- Enables fast checkpoint load/save, speeding recovery & experiments
- Supports high-throughput RDMA for distributed inference (KV-cache access)
- Works with Smallpond, enabling distributed preprocessing at cluster scale
- Makes DeepSeek's infrastructure cheaper and more efficient by maximizing SSD + RDMA performance

Capabilities:

- 6.6 TiB/s aggregate read throughput in a 180-node cluster
- 3.66 TiB/min throughput on GraySort benchmark in a 25-node cluster
- 40+ GiB/s peak throughput per client node for KVCache lookup
- Disaggregated architecture with strong consistency semantics
- Training data preprocessing, dataset loading, checkpoint saving/reloading, embedding vector search & KVCache lookups for inference in V3/R1

Runtime Layer – Smallpond

Smallpond is DeepSeek's high-performance data-processing framework built on top of 3FS.

It provides fast, distributed ETL (Extract–Transform–Load) for huge datasets used in LLM training—tokenization, preprocessing, shuffling, batching, filtering, and data transformations.

Why Deepseek needed Smallpond:

DeepSeek trains multi-trillion-token datasets. Traditional ETL pipelines (Spark, Ray, Dask, Arrow, etc.) struggle because:

1. Huge datasets (petabytes) need preprocessing quickly

Tokenization, shuffling, packing, and feature generation can take **weeks** on standard tools.

2. Existing ETL systems can't saturate RDMA/NVLink bandwidth

Most frameworks are CPU-heavy, network-slow, and not GPU-aware.

DeepSeek needs **60+ GB/s per node** sustained throughput—standard systems can't reach that.

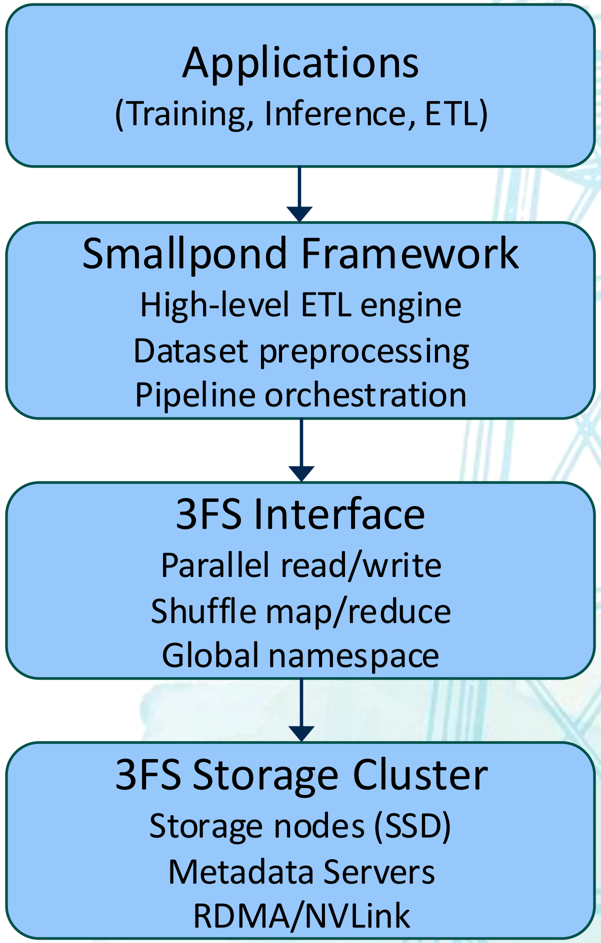
3. Training throughput collapses when data pipelines are slower than GPUs

GPUs sit idle because:

- dataset shuffling is slow,
- I/O is inconsistent,
- preprocessing isn't parallelized properly.

4. Need unified API for training, inference, and ETL (Existing ETL tools do *not* integrate tightly with distributed training engines**)**

DeepSeek wanted **one system** that, preprocesses data, reads/writes data during training, fetches KVCache during inference



Runtime Layer – Smallpond

What Smallpond Enables

- High-speed distributed dataset preprocessing
- Shuffle + MapReduce over RDMA
- Seamless integration with 3FS parallel filesystem
- Balanced pipeline orchestration matching DeepSeek-V3's training engine
- Zero-copy data movement to GPU nodes
- Massive throughput for dataset loading + checkpoint I/O

Benefits:

1. Keeps GPUs 100% Utilized

No stalls from slow dataloaders.

2. Removes CPU Bottlenecks

Moves bottleneck from CPUs → SSDs + RDMA network.

3. Predictable, Balanced Throughput

Deterministic sampling + sharding prevent data skew.

4. Massive Scale Support

Designed for:

180-node clusters

Tens of TB of data

Distributed MoE training workloads

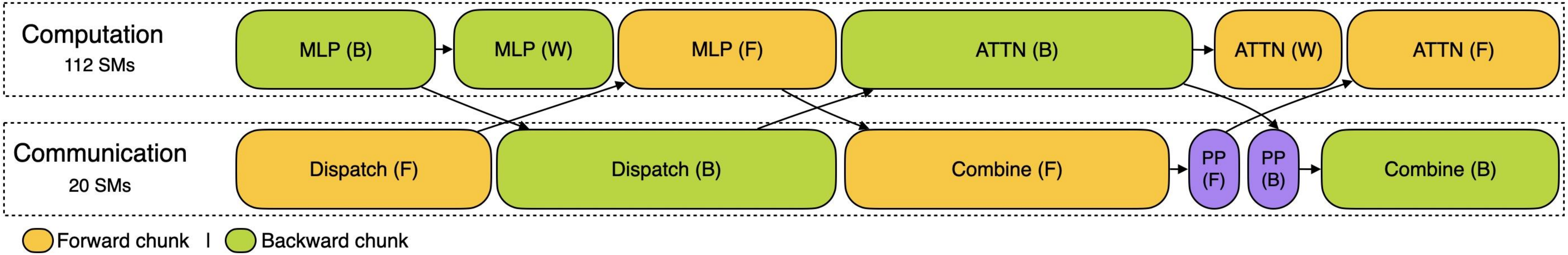
5. Deep Integration With DeepSeek Stack

Smallpond + 3FS + DeepEP =

fully optimized data → model pipeline at DeepSeek scale.

Profiling Layer

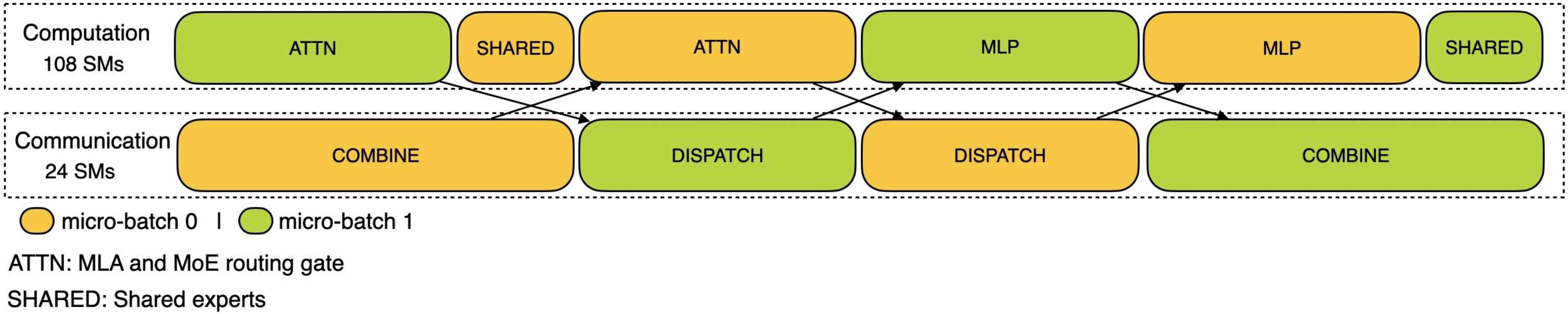
Training:



The training profile data demonstrates our overlapping strategy for a pair of individual forward and backward chunks in DualPipe. Each chunk contains 4 MoE (Mixture of Experts) layers. The parallel configuration aligns with DeepSeek-V3 pretraining settings: EP64, TP1 with 4K sequence length. And the PP communication is not included during profiling for simplicity.

Profiling Layer

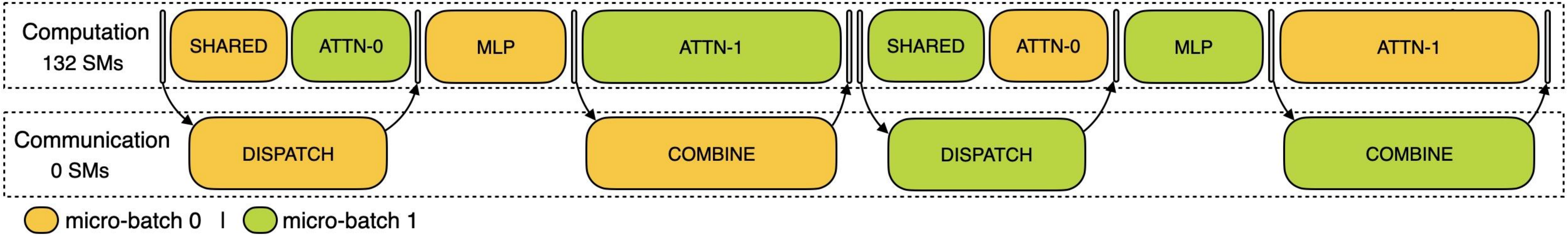
Prefilling:



For prefilling, the profile employs EP32 and TP1 (in line with DeepSeek V3/R1 's actual online deployment), with a prompt length set to 4K and a batch size of 16K tokens per GPU. In our prefilling stage, we utilize two micro-batches to overlap computation and all-to-all communication, while ensuring that the attention computation load is balanced across the two micro-batches — meaning that the same prompt may be split between them.

Profiling Layer

Decoding:



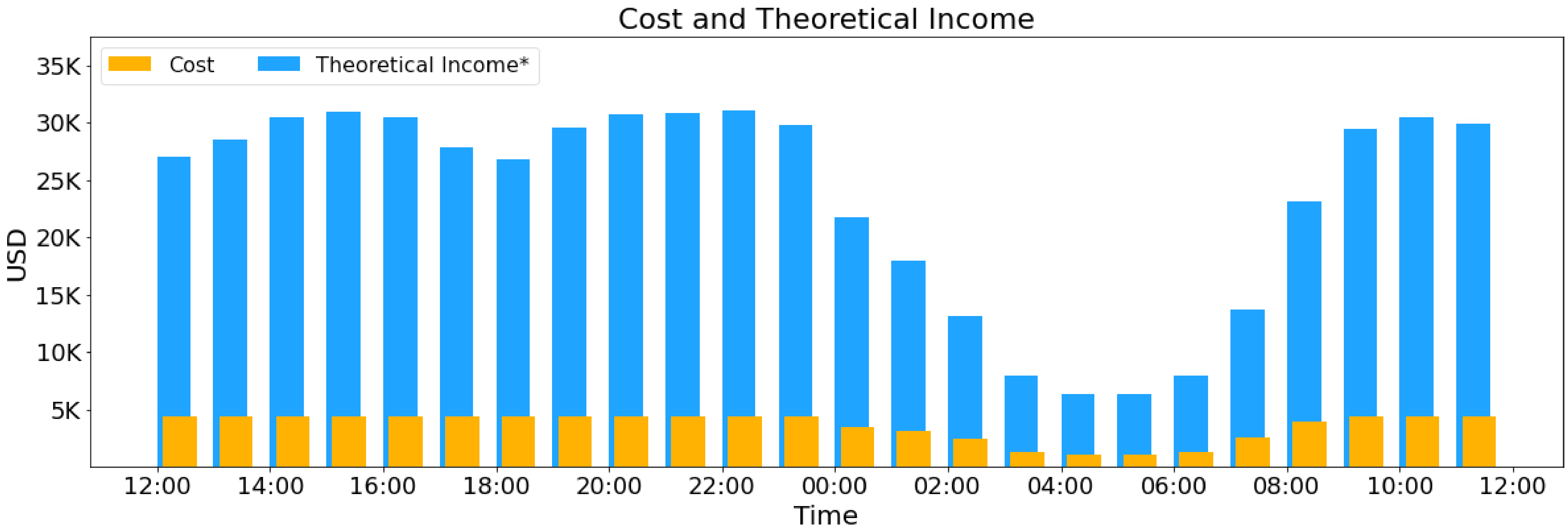
ATTN-0: MLA down/up projection and other ops after combine all-to-all and before core attention

ATTN-1: Core attention, attention output projection and MoE routing gate

SHARED: Shared experts

For prefilling, the profile employs EP32 and TP1 (in line with DeepSeek V3/R1 's actual online deployment), with a prompt length set to 4K and a batch size of 16K tokens per GPU. In our prefilling stage, we utilize two micro-batches to overlap computation and all-to-all communication, while ensuring that the attention computation load is balanced across the two micro-batches — meaning that the same prompt may be split between them.

DeepSeek-V3/R1 Inference: Cost vs Theoretical Income



* The theoretical income is calculated based on R1's standard API pricing, taking into account all tokens across web, APP, and API. It is not our actual income.

UCDAVIS

References

Open Infra Index Github Repo:

https://github.com/deepseek-ai/open-infra-index