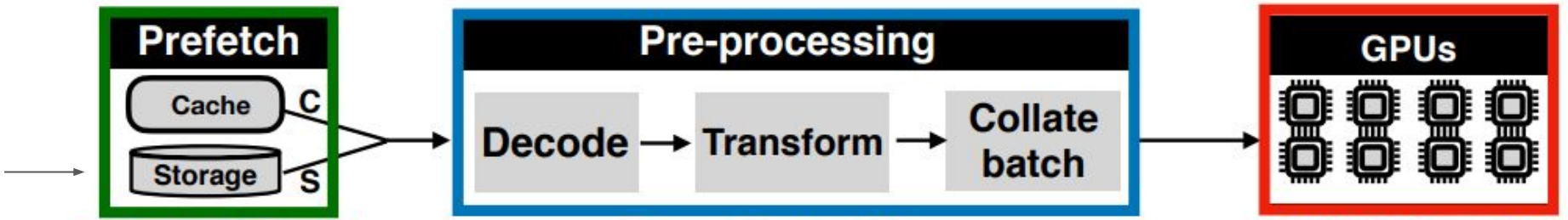
FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline

Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, Woo-Yeon Lee

Background: DL Training Preprocessing



Offline preprocessing

Big Data processing prior to training

- Aggregate
- Clean, Normalize, Encode
- …

CPU Intensive

Online preprocessing

Operate on train set batches

- Load, Decode, Static xform
- Augment (e.g. RIR)
- Tensor Batching

CPU Intensive

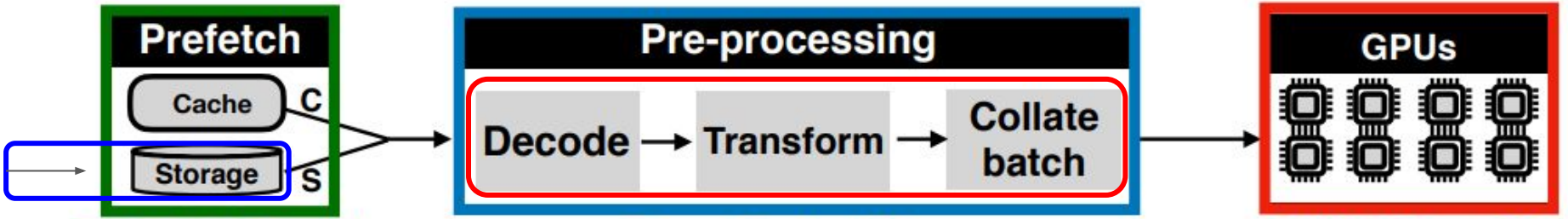
Gradient Computation

Compute on collated tensors

- Forward
- Backward

GPU Intensive

Background: DL Training Preprocessing



Offline preprocessing

Big Data processing prior to training

- Aggregate
- Clean, Normalize, Encode
- …

Execute once → materialize train set to shared storage

~~CPU Intensive~~ – Cached

Online preprocessing

Operate on train set batches

- Load, Decode, Static xform
- Augment (e.g. RIR)
- Tensor Batching

Working set does not fit in memory → incurs **multiple times each training**

CPU Intensive – cannot be Cached

Gradient Computation

Compute on collated tensors

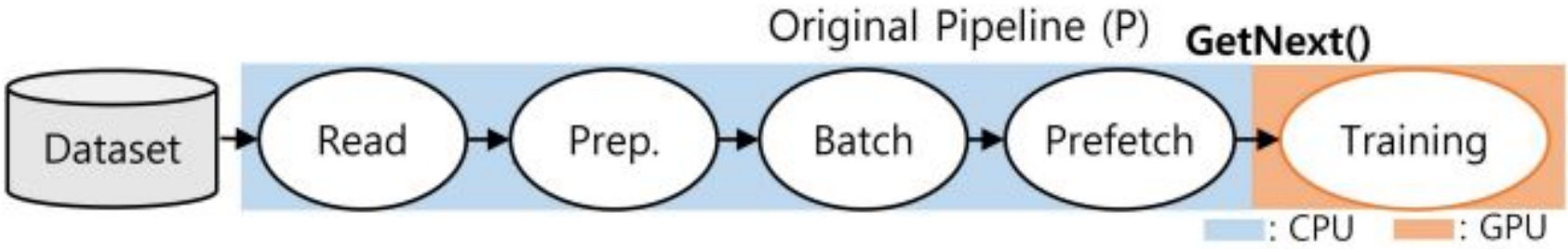
- Forward
- Backward

GPU Intensive

Focus: Online Preprocessing

Representation: **DAG** of coarse-grained operations

- e.g. *Prep.* transformation and augmentation operations (map)



Problem Statement

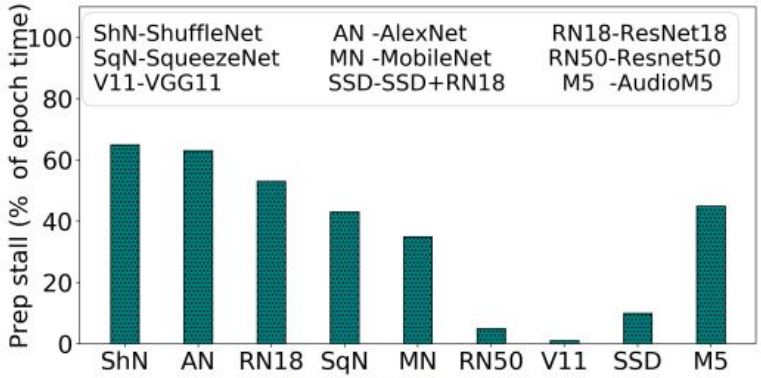
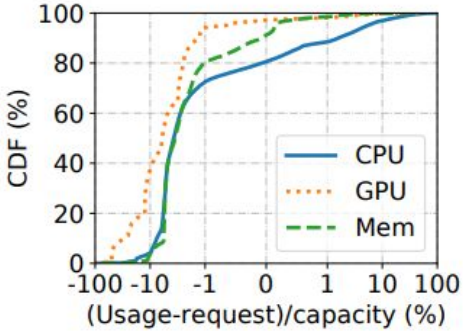


Figure 6: Prep stall across DNNs



(d) Usage minus request, normalized by the machine capacity.

Preprocessing ("prep") stalls characteristic of modern DL pipelines

- Data-centric model view drives CPU demand
- Relative research effort w.r.t GPU – divergence in CPU v. GPU cycle time

CPU Bottlenecks in input preprocessing → GPU underutilization in training

- Significant epoch time stalled by preprocessing

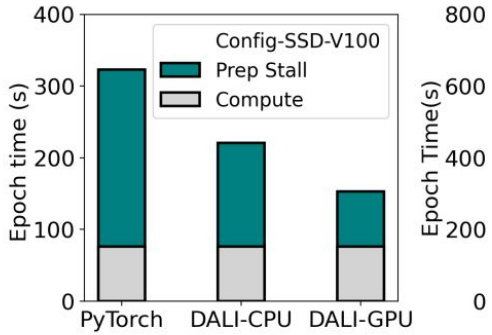
Analyzing and Mitigating Data Stalls in DNN Training (arxiv.org), nsdi22-paper-weng.pdf (usenix.org)

Existing Solutions

Alternative Compute

Input Preprocessing:

- GPU – *DALI (NVIDIA Data Loading Library)*
- FPGA – *DLBooster, TrainBox*



(a) PyTorch vs DALI(3 CPU per GPU)

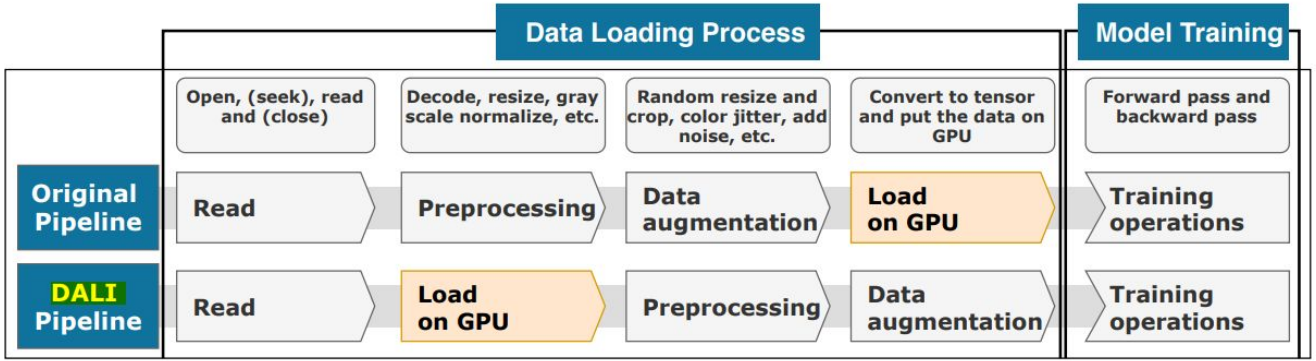


Figure 1: Batch training sequence performed on a GPU with and without DALI.

Analyzing and Mitigating Data Stalls in DNN Training (arxiv.org)

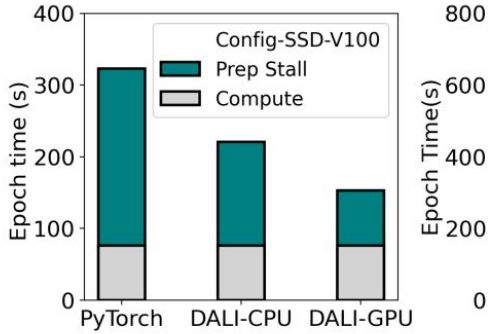
Existing Solutions

Alternative Compute

Input Preprocessing:

- GPU – *DALI (NVIDIA Data Loading Library)*
- FPGA – *DLBooster, TrainBox*

Inflexible to develop/translate CPU operations towards specialized hardware



(a) PyTorch vs DALI(3 CPU per GPU)

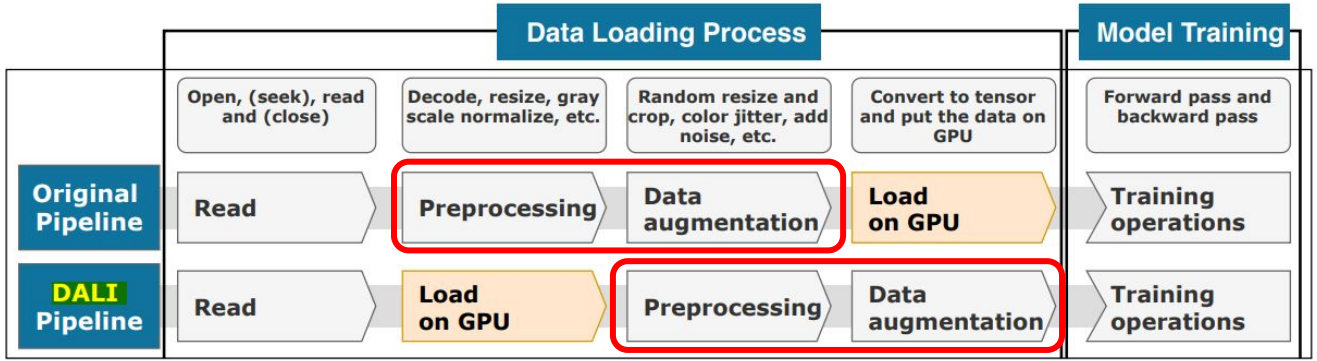
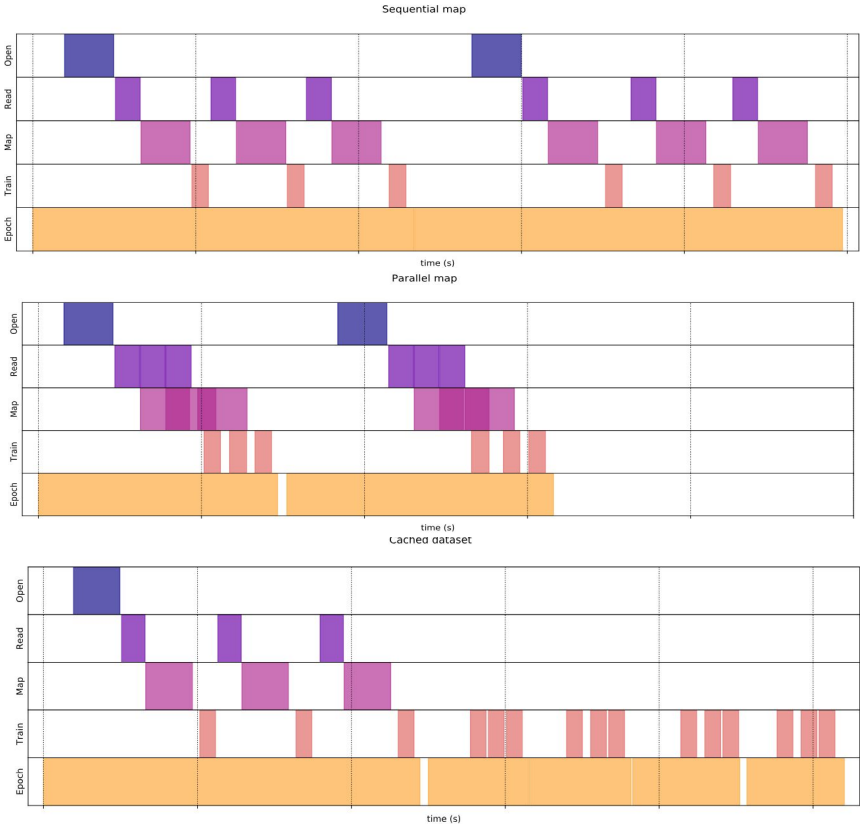
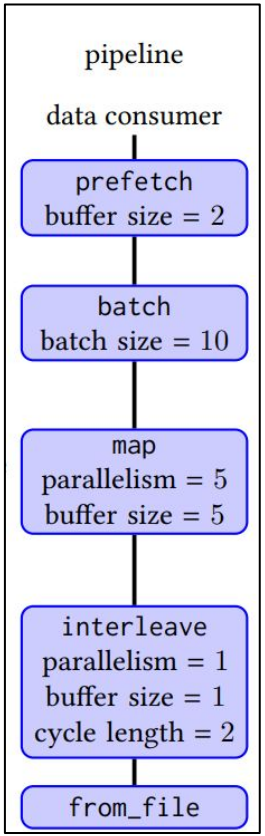


Figure 1: Batch training sequence performed on a GPU with and without DALI.

Analyzing and Mitigating Data Stalls in DNN Training (arxiv.org)

Existing Solutions





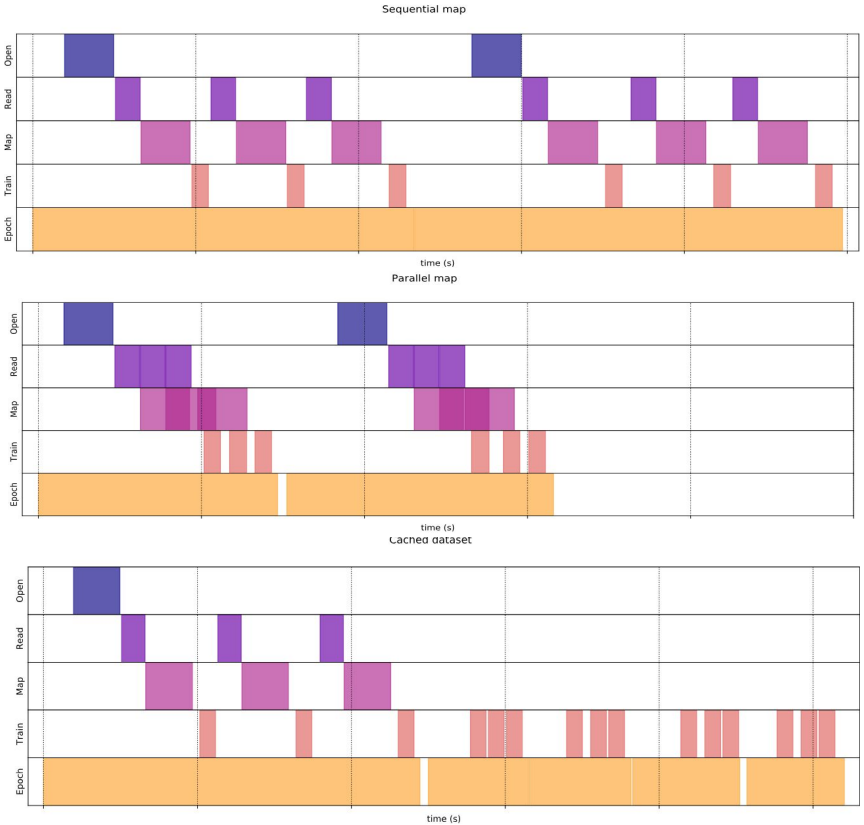
Auto-tuning on local node

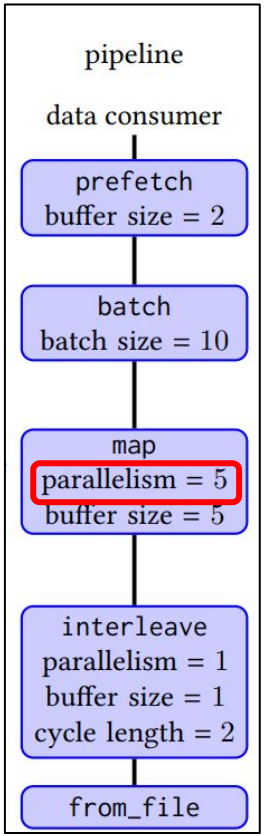
Optimized parallelism and caching **across local resource set**

- Efficient input pipelines – *tf.data*
- Analysis & Tuning – *Plumber*

nsdi22-paper-weng.pdf (usenix.org)

Existing Solutions





Auto-tuning on local node

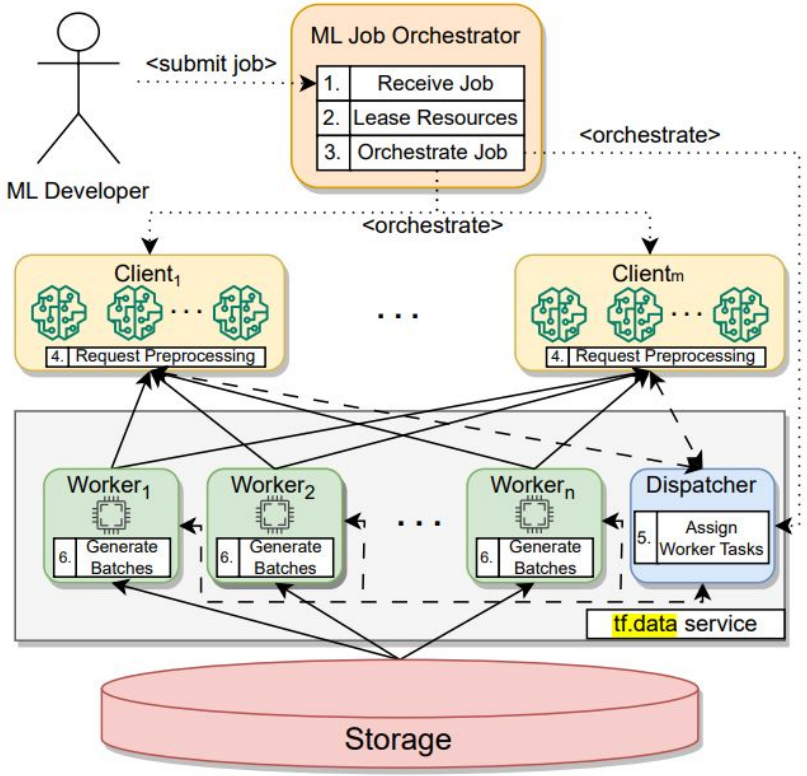
Optimized parallelism and caching **across local resource set**

- Efficient input pipelines – *tf.data*
- Analysis & Tuning – *Plumber*

Capacity restricted to local node

nsdi22-paper-weng.pdf (usenix.org)

Existing Solutions



Remote CPU offload

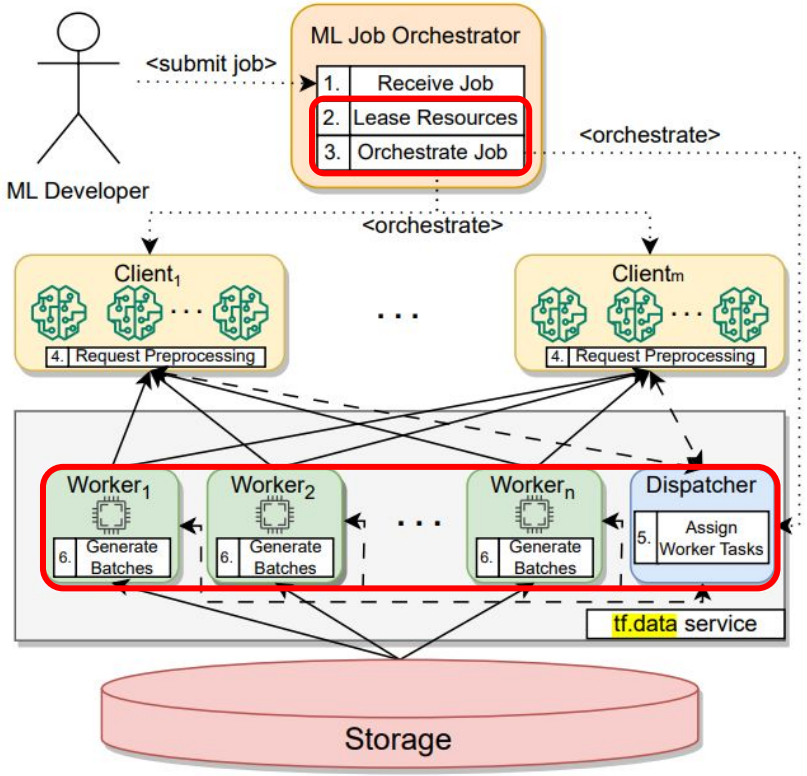
Distributed pre-processing

- Dispatcher / Worker model – *tf.data.service*
- *.distribute*

```
import tensorflow as tf
# Input pipeline
ds = tf.data.Dataset(data_path)
ds = ds.map(...).distribute(dispatcher_addr..)
       .batch(..).prefetch(..)
# define model
class MyModel(tf.keras.Model):
    def __init__(self,...):
        ...
model = MyModel(..)
# training
model.compile(..)
model.fit(ds, epoch=..,)
```

tf.data service: A Case for Disaggregating ML Input Data Processing (arxiv.org)

Existing Solutions



Remote CPU offload

Distributed pre-processing

- Dispatcher / Worker model – *tf.data.service*
- *.distribute*

User-defined

- Suboptimal offloading
- Lack of precision

```
import tensorflow as tf
# Input pipeline
ds = tf.data.Dataset(data_path)
ds = ds.map(...).distribute(dispatcher_addr..)
      .batch(..).prefetch(..)
# define model
class MyModel(tf.keras.Model):
    def __init__(self,...):
        ...
model = MyModel(..)
# training
model.compile(..)
model.fit(ds, epoch=..,)
```

tf.data service: A Case for Disaggregating ML Input Data Processing (arxiv.org)

FastFlow

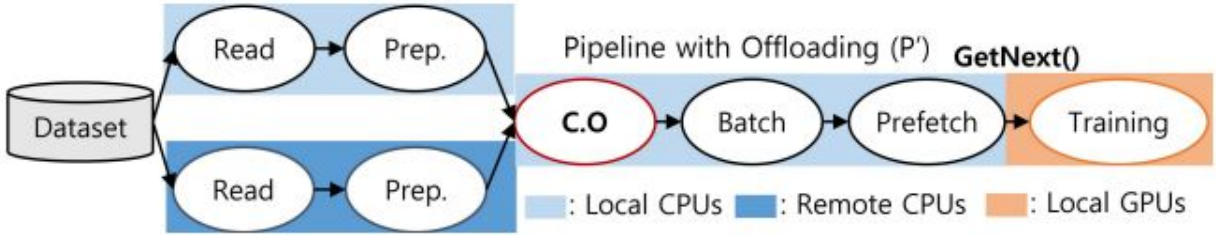
Addresses challenges by proposing **automatic offloading** of input pipeline to remote CPUs

Extends **Local Auto-tuning**

- Autotunes across local and remote set

Extends **Remote CPU Offload**

- Embeds greater precision into pipeline choice and execution
  - When – offload control flow
  - Which – operation choice (Pipeline choice)
  - Quantity – offload ratio
- Decision automation
  - Scalable across diverse resource environments and workloads



FF: *tf.data*, *tf.data.service* integration

(Local Auto-tuning) (Remote Offload)

```python
import fastflow as ff
import tensorflow as tf
# input pipeline
ds = tf.data.Dataset(data_path)
ds = ds.map(...).batch(..).prefetch(..)

# define model
class MyModel( ff.Model ):
    def __init__(self,...):
        ...
    def __deepcopy__(self):
        # must be implemented
model = MyModel(..)
# training with auto-offloading
model.compile(..)
model.fit(ds, epoch=.., ff_config=config)
```

```python
class Model(keras.Model): # FastFlow Model
  ...
  def fit(self, x=None, # input pipeline
          conf=None, # auto-offloading conf
          **kwargs):
    with self.distribute_strategy.scope():
      M = self.__deepcopy__()
      M.compile(..)
    launch_workers(conf)
    P = smart_offloading(x,M,conf)
    # Reuse the main training logic
    super(Model, self).fit(x=P, **kwargs)
    destroy_workers(conf)
```

FF

grad()

Source Code 3: FastFlow's `model.fit` method.

```python
ds = ds.map(...).distribute(dispatcher_addr..)
       .batch(..).prefetch(..)
```

FastFlow

Principal Mechanisms

- **Metric profiling** – *tf.data*, *tf.data.service* extension
- "**Smart Offloading**" policy algorithm

System:

- Single Node/Multi-GPU

FF: Overview

One Epoch:

[FF] **Profile** Model M'=M on Pipelines

- generate + index: *Offloading-Metrics*
- 50-100 batches

[FF] **P** ← **Smart Offloading**(*Offloading-Metrics*, Pipelines)

[Keras] Train Epoch on Model M given **P**

- 100s of batches

FF: Profiling Pass

Before Training: Deep Copy **Model *M*** - retain for profiling

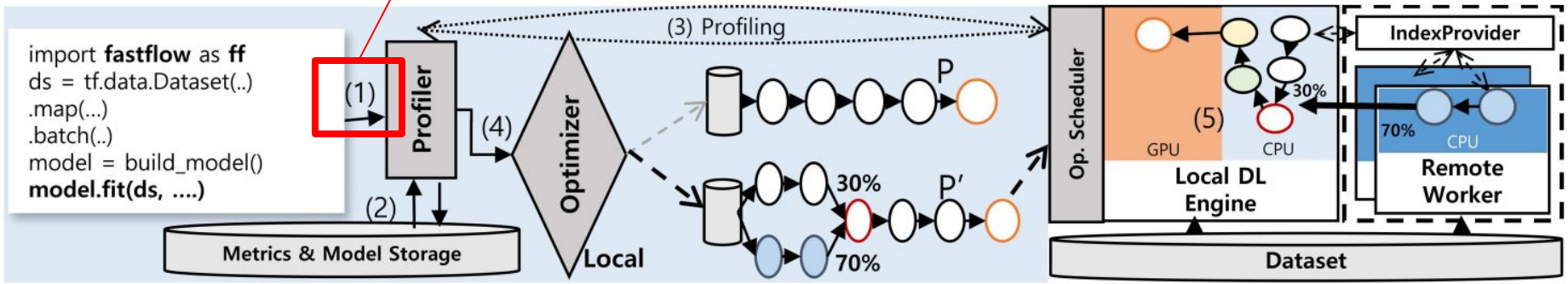


Figure 3: The overall workflow of FastFlow to execute a DL training job with smart offloading.

FF: Profiling Pass

Profile + store indexed metrics

- GPU throughput (M)
- E2E throughput (M + P)
- CPU Cycles (P)

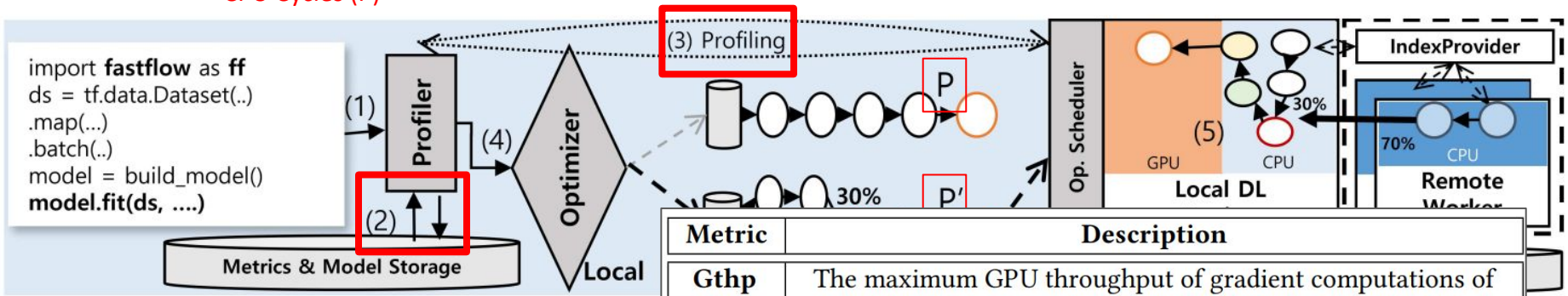


Figure 3: The overall workflow of Fa

Key	Value
M	$Gthp$
(P, M)	$(Lthp, Pcycle)$
(P, M, RemoteNodeInfo)	$(Rthp, P', Ocycle)$

Metric	Description
Gthp	The maximum GPU throughput of gradient computations of model (M) ($thp(\boldsymbol{M})$)
Lthp	The maximum training throughput of input pipeline (P) and gradient computations of model (M) ($thp(\boldsymbol{P+M})$)
Rthp	The maximum training throughput when all data is preprocessed on the remote nodes with the offloaded input pipeline (P') and gradient computations of model ($thp(\boldsymbol{P'+M})$)
Ocycle	CPU cycles on local CPUs for training with offloaded pipeline
Pcycle	CPU cycles on local CPUs for training without offloading

FF: Overview

One Epoch:

[FF] **Profile** Model M'=M on Pipelines

- generate + index: *Offloading-Metrics*
- 50-100 batches

[FF] **P** ← **Smart Offloading**(*Offloading-Metrics*, Pipelines)

[Keras] Train Epoch on Model M given **P**

- 100s of batches

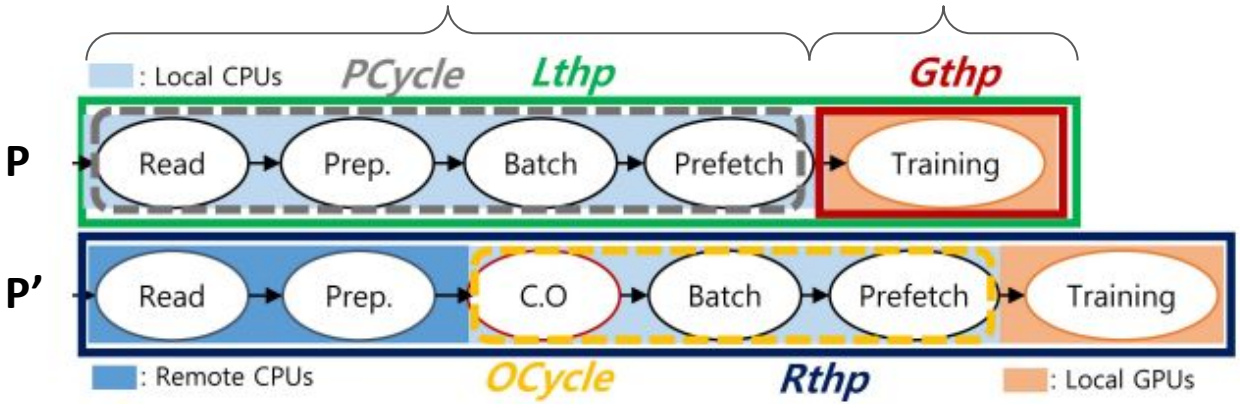
FF: Smart Offloading algorithm

Recall: High Level Policy

1. When – offload control flow
2. Which – operation/pipeline choice
3. Quantity – offload ratio

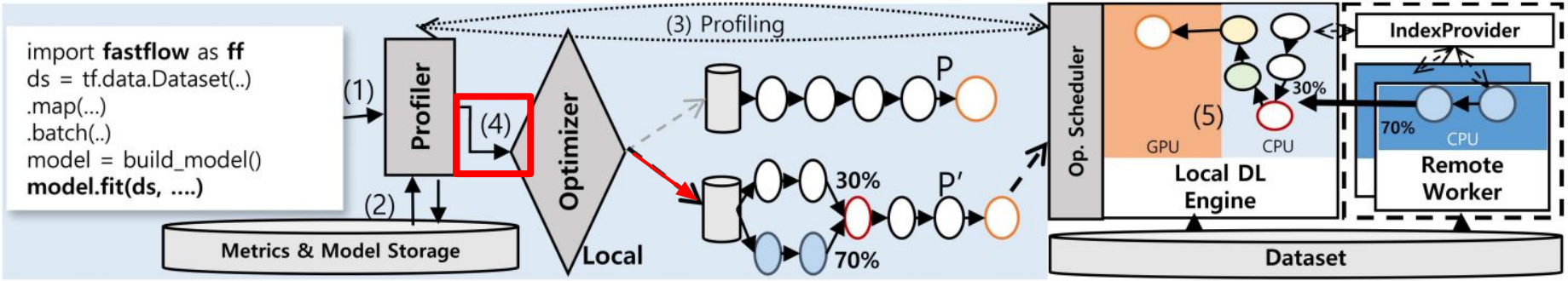
FF: Smart Offloading: 1. Offload Choice

When – offload control flow



FF: Smart Offloading: 2. Pipeline Choice

Which – operation offload choice



Choose pipeline that **best fits M towards R/L system balance:**

- Network I/O
- Storage Device BW
- Offloading Overheads

…

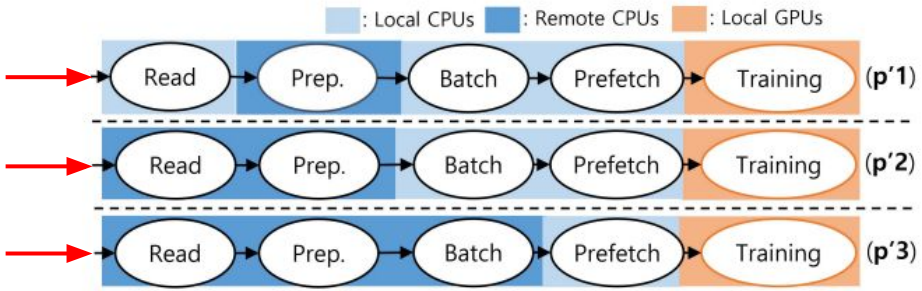


Figure 5: Candidate pipelines for offloading.

FF: Smart Offloading: 2. Pipeline Choice

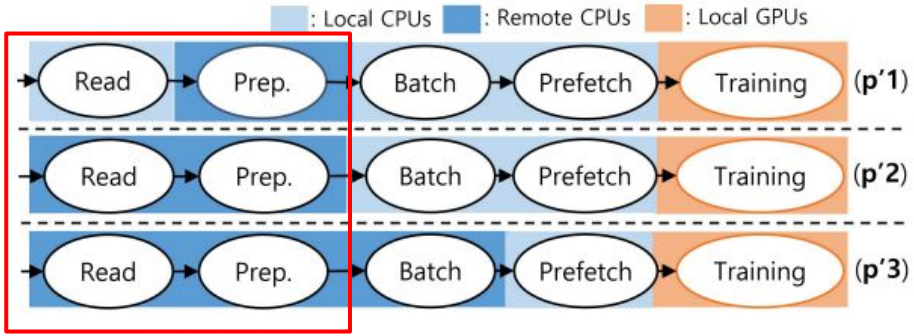


Figure 5: Candidate pipelines for offloading.

Evaluate Trade-offs:

- **p'1 vs p'2/p'3 (Local vs Remote – Fetch)**:
  - **p'1**: local read + network encode
  - **p'2/p'3:** local read

FF: Smart Offloading: 2. Pipeline Choice

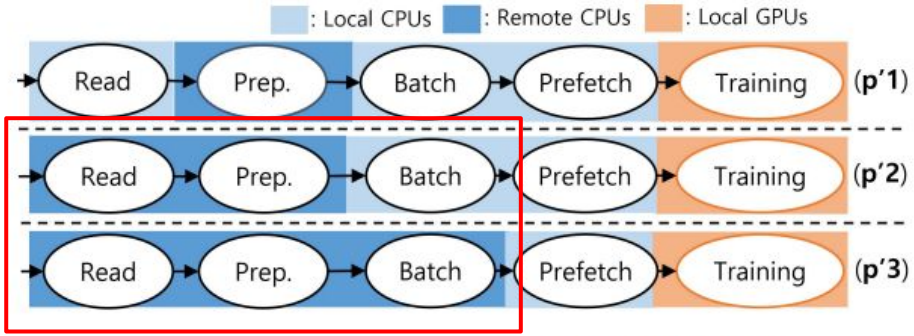


Figure 5: Candidate pipelines for offloading.

Evaluate Trade-offs:

- **p'1 vs p'2/p'3 (Local vs Remote – Fetch)**:
  - **p'1**: local read + network encode
  - **p'2/p'3:** local read
- **p'2 vs p'3 (Both Remote)**:
  - **p'2**: Element-wise offloading ⟶ higher network I/O, lower threading overhead.
  - **p'3**: Batch-wise offloading ⟶ amortized I/O, higher threading overhead.

FF: Smart Offloading: 3. Offload Ratio

Quantity – offload ratio

- ***Upper***: The capacity remote CPUs can hold before becoming a bottleneck.
- ***Lower***: The mismatched throughput between local input pipeline and GPUs (+ offloading overheads)
- ***Lower < Upper***: offload more
- Otherwise, evenly balance the load based on resource capacity

Evaluation

- **Questions to be answered**
  - Does FastFlow improve performance compared to baselines in various workloads and resource environments?
  - How do the offloading decisions affect the performance of FastFlow?

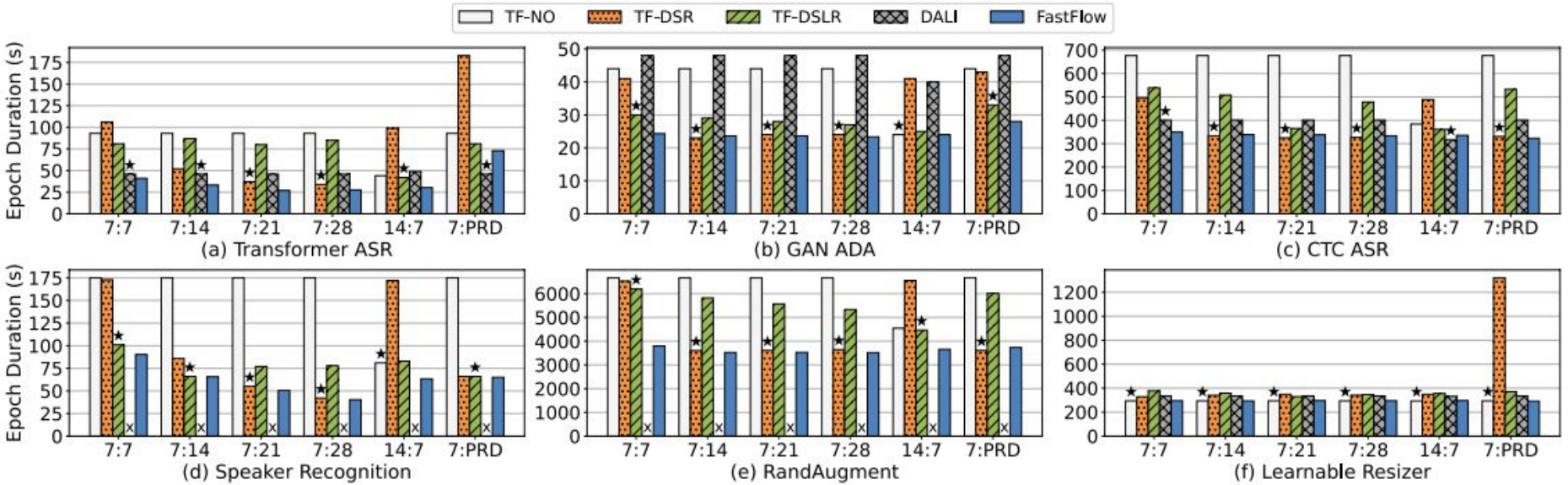
Evaluation

- **Setup**
  - Private cloud system (PCS) built on Kubernetes
  - Resource Environment
    - Local PCS GPU Container: one GPU with x vCPUs
    - Remote PCS CPU Container: y vCPUs; High network bandwidth
    - Remote PRD Node: 30 CPUs; Low network bandwidth

Evaluation

- **Workloads**
  - **Transformer ASR (T)**
  - **GAN ADA (G)**
  - **CTC ASR(C)**
  - **Speaker Recognition (S)**
  - **ResNet50 with RandAugment (R)**
  - **MelGAN (M)**
- **Baselines**
  - **TF-NO** (TensorFlow 2.7.0 with no offloading)
  - **TF-DSR** (TensorFlow+tf.data.service with remote worker only)
  - **TF-DSLR** (TensorFlow+tf.data.service with local and remote)
  - **DALI** (Offloading to the local GPU)

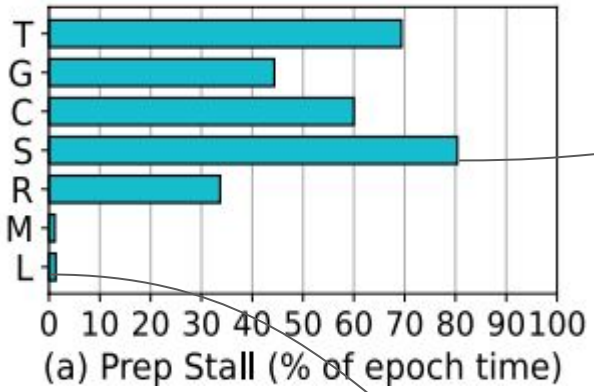
Evaluation - Comparison of Baselines

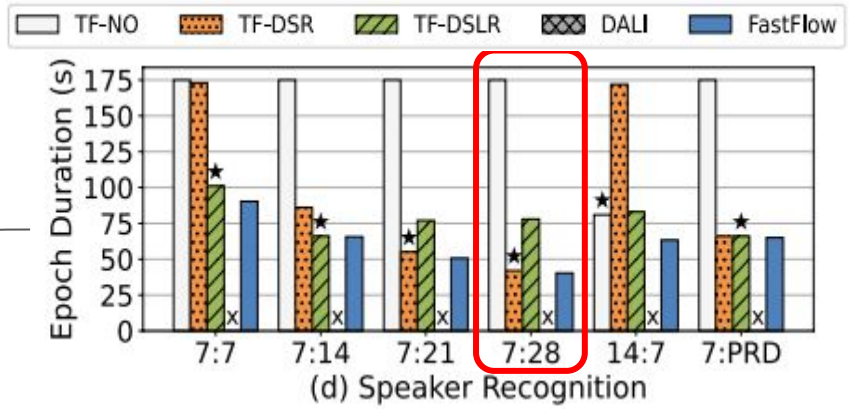


Evaluation - Comparison of Baselines

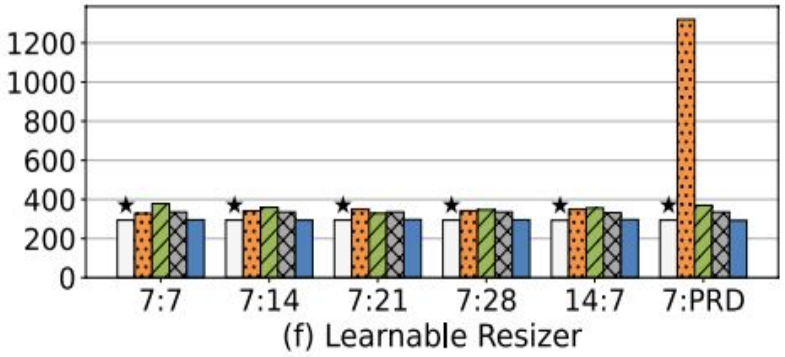
- **TF-NO (no offloading)**

Higher prep stall -> higher speed-up FF achieves



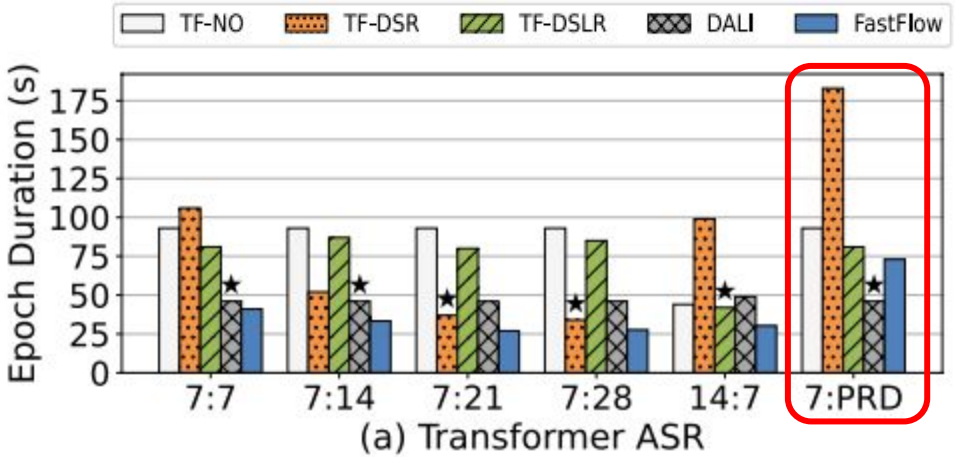


Low prep stall -> similar training time



Evaluation - Comparison of Baselines

- **TF-DSR (w/ remote worker only)**

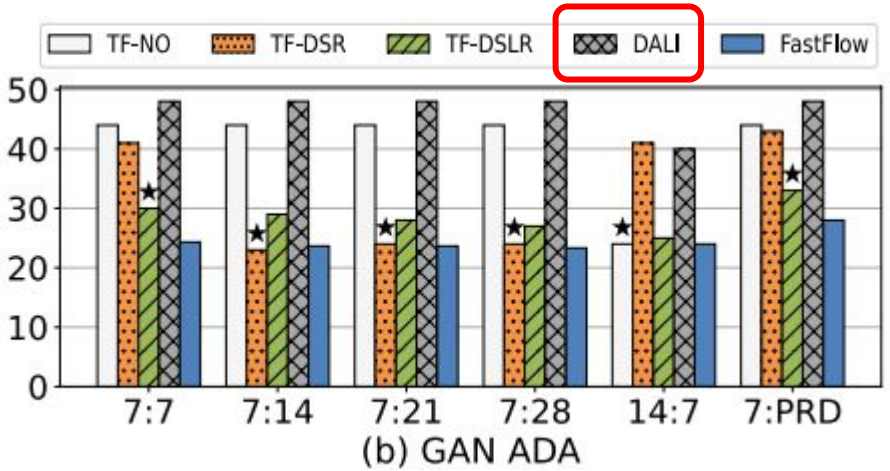


Network bandwidth becomes a bottleneck

Evaluation - Comparison of Baselines

DALI (Preprocessing on GPUs)

- **Slower than FastFlow in most training workloads**
- **Local GPU bottleneck**
- **Degrades performance in GAN ADA**



Evaluation - Comparison of Baselines

- **TF-DSLR (TensorFlow+tf.data.service w/ local and remote)**

  FastFlow has better performance compared to TF-DSLR:
  1. Optimal candidate pipeline
  2. Offloading data ratio decision

Evaluation

- **Performance Impact of Offloading Decisions**

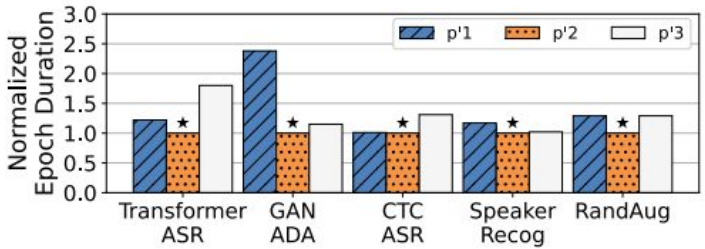


Figure 8: Impact of the offloading operator selection of FastFlow. * mark shows the selection of FastFlow.

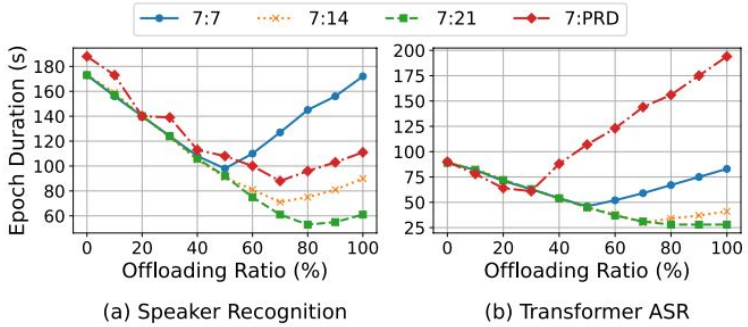


Figure 9: Epoch duration in various offloading ratios on different remote workers. (a) is compute-intensive, and (b) is data-intensive preprocessing workload.

Drawbacks of FastFlow

- Granularity of Operation
  - More fine-grained control to optimize specific preprocessing operations
- Pipeline Choice/Enumeration
  - More optimal offloading strategies
  - Workload-specific pipeline
- Multi-node/multi-GPU
  - Cross-node communication, data synchronization or network bottlenecks
- Granularity of FastFlow Execution (Metrics Profile + Pipeline Choice)
  - Are initial profiling and metric reuse sufficient?

Evaluation - Multi-GPU Training

- **Test Environment**: 1 to 4 GPUs using **synchronous data parallelism**.
- **vCPU:GPU Ratio**: 7:1, scaling both local and remote CPUs from 7 to 28 as GPUs increase.

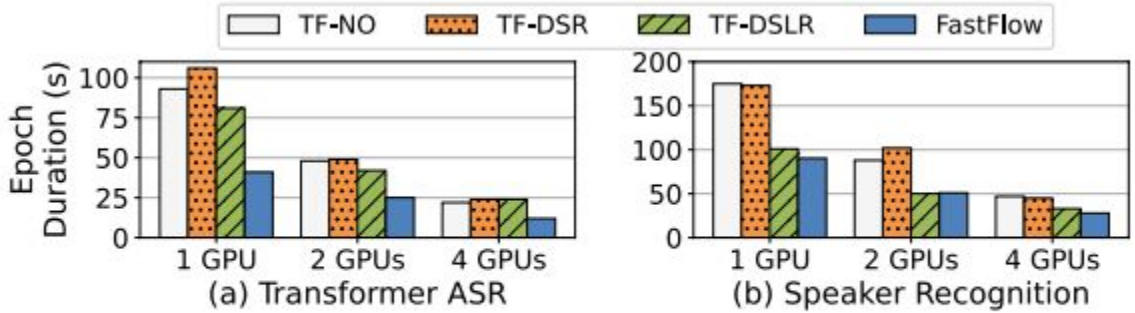


Figure 12: Multi-GPU training for (a) Transformer ASR and (b) Speaker Recognition.

Rail-only: A Low-Cost High-Performance Network for Training LLMs with Trillion Parameters

Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, Naader Hasani

Problem Statement

Problem: the growth rate of LLM size and computation requirement **exceeds** the advancement of accelerators, making hyper-scale GPU data centers inevitable

However, the current design of such data centers has room for improvement

Problem Statement

Problem: the growth rate of LLM size and computation requirement exceeds the advancement of accelerators, making hyper-scale GPU data centers inevitable

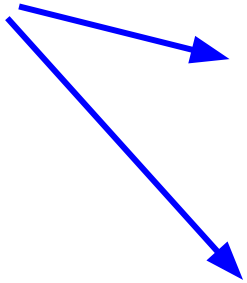
However, the current design of such data centers has room for improvement

Observation: efficiently training LLMs does not require **any-to-any connectivity** across all GPUs in the network

Proposed solution: Remove the spine layers in the current network design and build a rail-only network!

Problem Statement

Problem: the growth rate of LLM size and computation requirement exceeds the advancement of accelerators, making **hyper-scale GPU data centers** inevitable



HB domain: intra-platform connectivity

GPU-centric platforms optimized for multi-GPU workloads, provisioning several Tbps of internal bandwidth across GPUs.

NIC domain: inter-platform connectivity

Inter-platform network, relying on traditional network technology, such as *Ethernet* or *Infiniband*.

SOTA: Rail-optimized network

Recap: Clos network

Clos network: multistage switching network, providing **any-to-any** connectivity between server pairs.

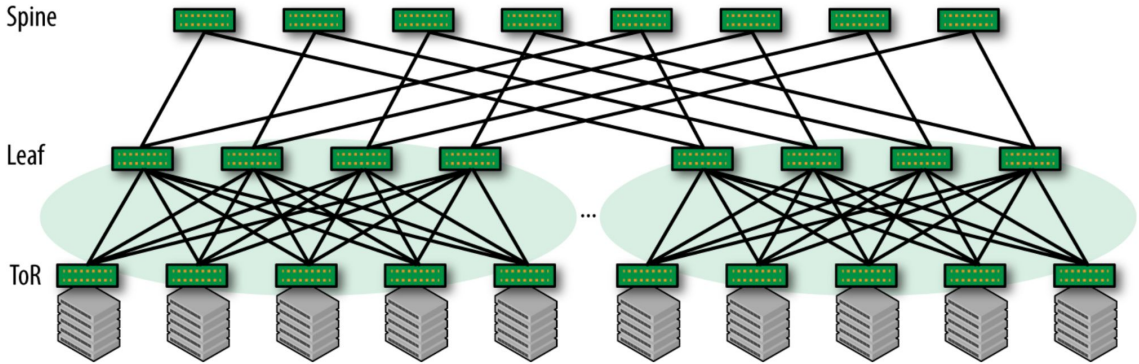


Figure 1: 3-tier Clos topology in a datacenter networking.

Rail-optimized network

What is a "rail"? A rail comprises GPUs with the same local rank that belong to different HB domains.

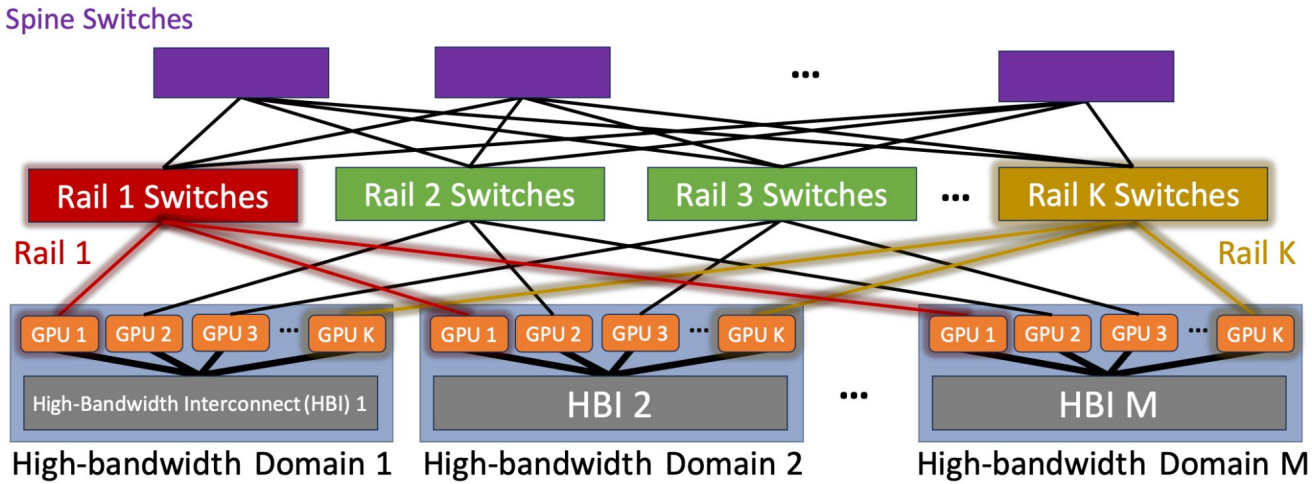


Figure 2: A GPU datacenter with Rail-optimized, any-to-any Clos networks.

Rail-optimized network

What is a "rail"? A rail comprises GPUs with the same local rank that belong to different HB domains.

What is "rail-optimized"? Connecting same-rank GPUs to the same rail switches ensures the lowest possible latency across them.

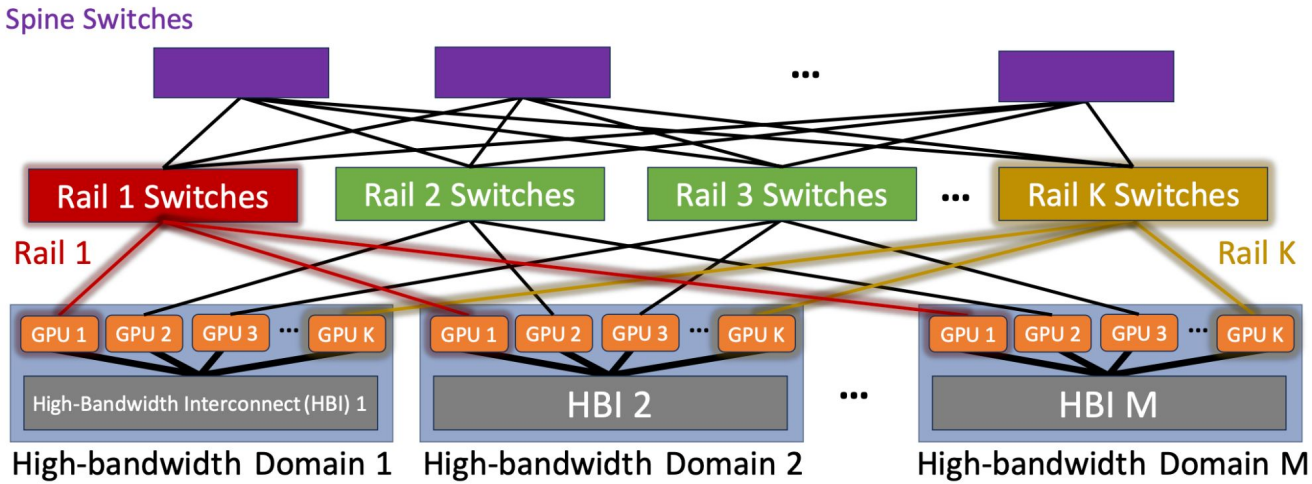
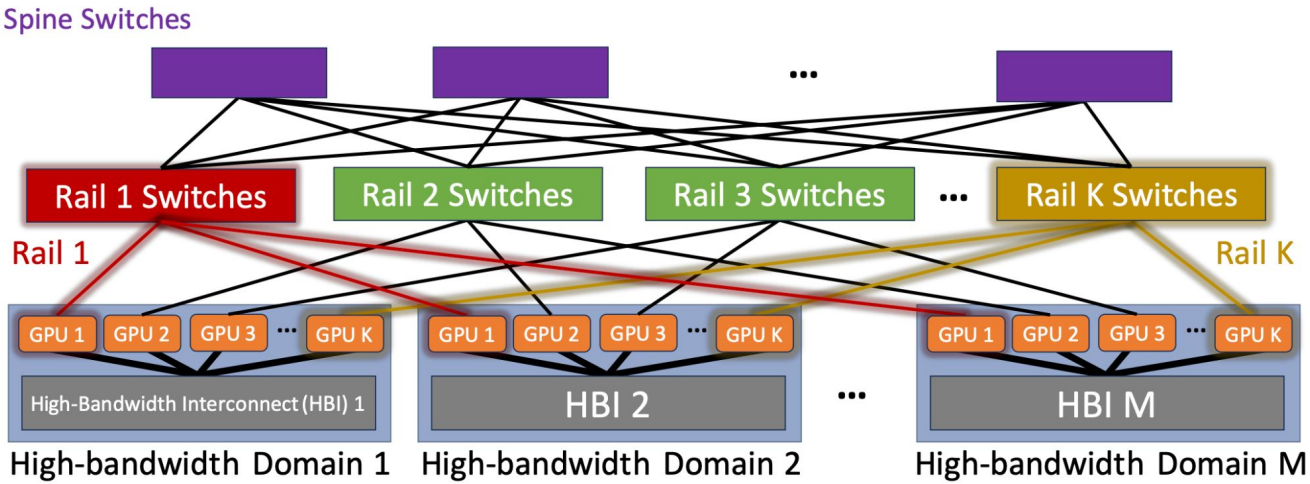


Figure 2: A GPU datacenter with Rail-optimized, any-to-any Clos networks.

Rail-optimized network

What is a "rail"? A rail comprises GPUs with the same local rank that belong to different HB domains.

What is "rail-optimized"? Connecting same-rank GPUs to the same rail switches ensures the lowest possible latency across them.



Did not leverage the unique traffic patterns of LLM training workloads.

Are the spine switches necessary?

Figure 2: A GPU datacenter with Rail-optimized, any-to-any Clos networks.

LLM Traffic Pattern Analysis

Analyze LLM traffic patterns to answer the previous question:

“are the spine switches necessary?”

- Traffic pattern of MegatronLM
- Traffic in the NIC domain for LLMs
- All-to-all traffic for Mixture-of-Expert models

Traffic Pattern of MegatronLM

Setup:

- 4 GPT models:
  - 145.6 billion parameters
  - 310.1 billion parameters
  - 539.6 billion parameters
  - 1 trillion parameters
- Distributed in a cluster of up to **384** DGX A100 servers (up to **3072** GPUs in total) with an HB domain of size 8

Traffic Pattern of MegatronLM

Figure 3(a): volume percentage for each type of communication for one training iteration.

Takeaways:

- DP and PP traffic volume is significantly **smaller** than TP traffic
- TP traffic accounts for **over 75%** of the total transmitted data

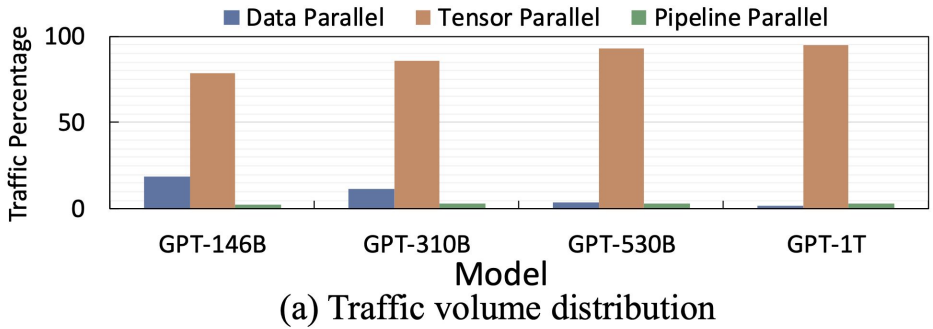


Figure 3(a): The traffic volume from different parallelization dimensions.

Recall:
DP and PP traffic: happen in the NIC domain

TP traffic: stays within HB domains

Traffic Pattern of MegatronLM

Figure 3(b): the communication type distribution across all GPU pairs.

Takeaways:

- Over 99% of GPU pairs carry no traffic and less than 0.04% of GPU pairs carry TP traffic

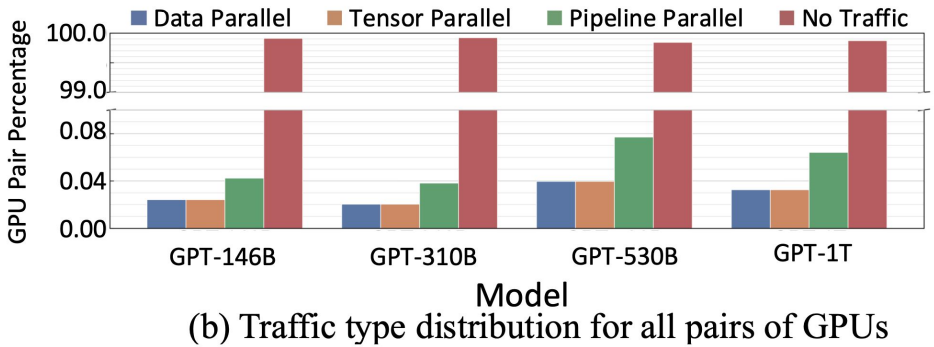


Figure 3(b): The communication type across all GPU pairs.

Traffic Pattern of MegatronLM

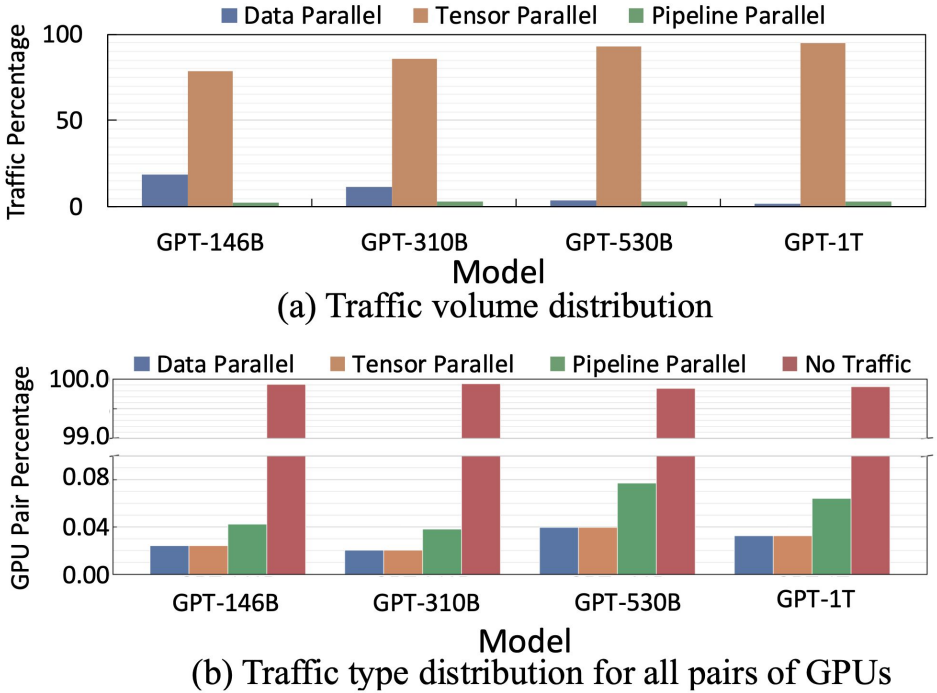
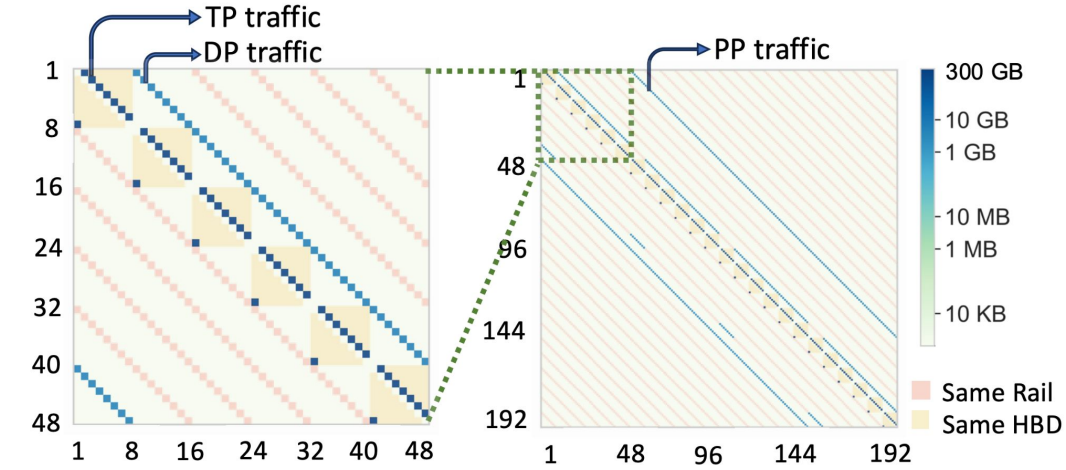


Figure 3: (a) The traffic volume from different parallelization dimensions; (b) The communication type across all GPU pairs.

Conclusion:

Building a GPU datacenter with **any-to-any** connectivity on top of HB domains for LLM models is excessive!

Traffic in the NIC Domain for LLMs



(a) GPT-1T MegatronLM traffic matrix GPU 1 to 48 (one pipeline stage)

(b) GPT-1T MegatronLM traffic matrix GPU 1 to 192 (four pipeline stages)

Takeaways:

- Traffic volume: ~ 300 GB in an HB domain, while only ~ 6 GB in the NIC domain
- Traffic in the NIC domain never traverse through the spine switches – only within the rails

Figure 4: Traffic heatmaps for GPT-1T in MegatronLM. Highlights show GPUs in the same HB domains and rails.

Traffic in the NIC Domain for LLMs

Argument: all LLMs without sparse MoE layers distributed with an optimal parallelization strategy always induce sparse, low-volume traffic *within the rails*.

Proof:

Traffic exiting the HB domains:

- Point-to-point traffic from PP
- Collective communication traffic (AllGather, ReduceScatter, and AllReduce) from TP and DP

Traffic in the NIC Domain for LLMs

Proof:

Traffic exiting the HB domains:

- Point-to-point traffic from PP

  symmetric LLM parallelization → traffic across stages traverses the GPUs of the same rank in the NIC domain only → stays *within the same rail*

- Collective communication traffic (AllGather, ReduceScatter, and AllReduce) from TP and DP

Traffic in the NIC Domain for LLMs

Proof:

Traffic exiting the HB domains:

- Point-to-point traffic from PP
- Collective communication traffic (AllGather, ReduceScatter, and AllReduce) from TP and DP
  - Common patterns such as in MegatronLM: TP and DP contained within HB and NIC domains only, respectively, traffic stays *within the same rail*
  - Others cases: could use *hierarchical collective communication* algorithms → we will prove that they also stay *within the same rail*

Hierarchical Collective Communication Algorithms

Hierarchical collective communication algorithms (use AllGather as an example):

1. **Each rail**: collect partial data for each rail of GPUs without transferring data in the HB domain
2. **Each HB**: conduct AllGather operations within each HB domain

Hierarchical Collective Communication Algorithms

Takeaways:

- Network traffic stays within a rail
- HB domain: ~ 98% of the traffic, while NIC domain: ~2%

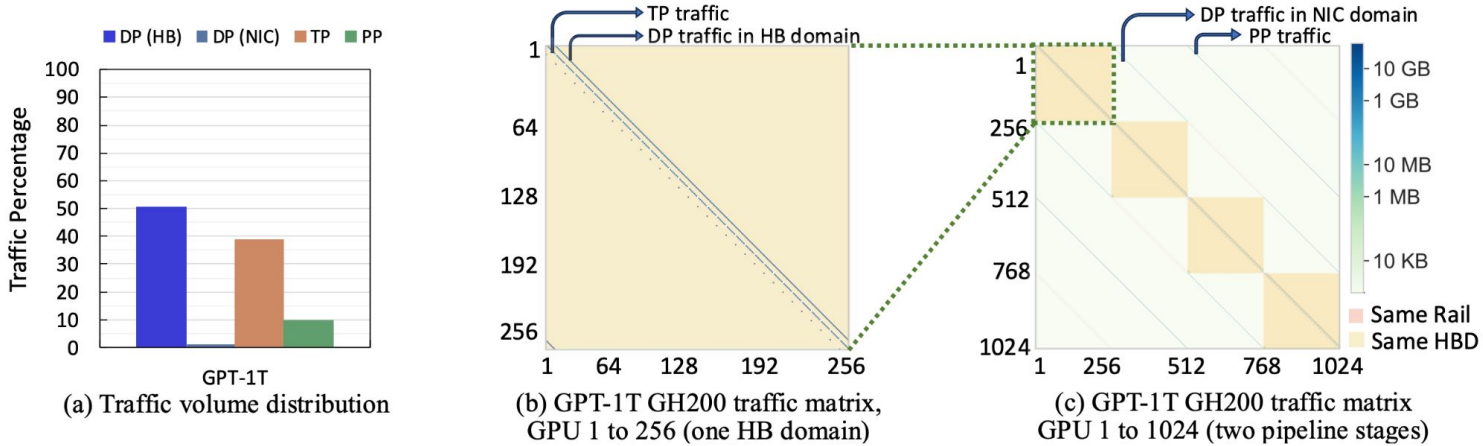
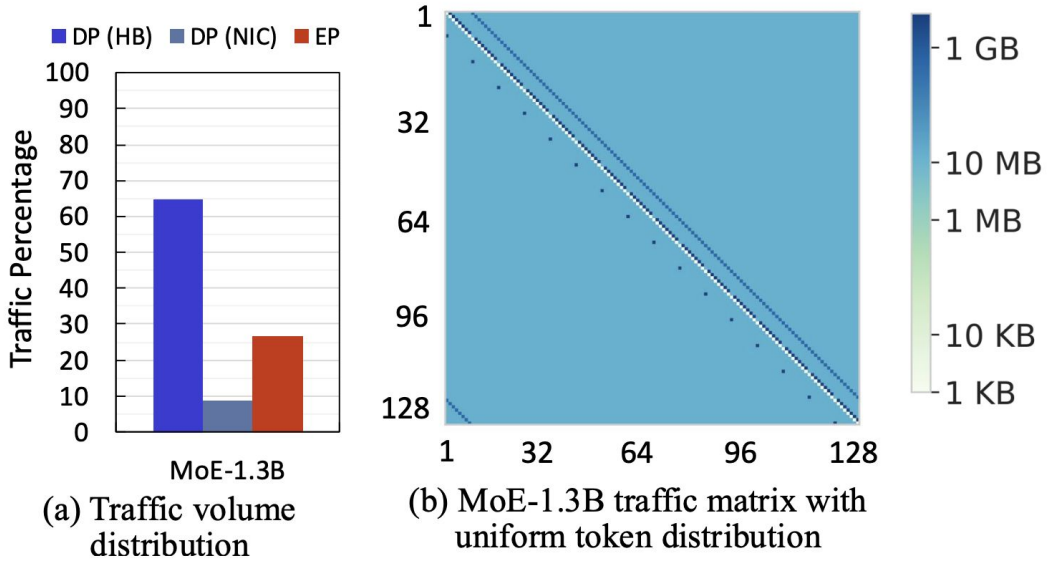


Figure 5: Traffic distribution and heatmaps for GPT-1T, distributed on 16 DGX GH200s. Note that DP (NIC) accounts for 0.8% of the total traffic percentage. The ”Same-Rail” legend on Figure 6 appears for GPUs whose ranks are 256 apart.

All-to-All Traffic for Mixture-of-Expert Models

Expert parallelism: each expert is distributed to a different GPU in the cluster.



Are spine switches crucial in MoE case?

Figure 6: Traffic volume distribution and heatmap for the MoE-1.3B model in DeepSpeedMoE, assuming uniform token distribution.

Rail-only Network Design

Are the spine switches necessary?

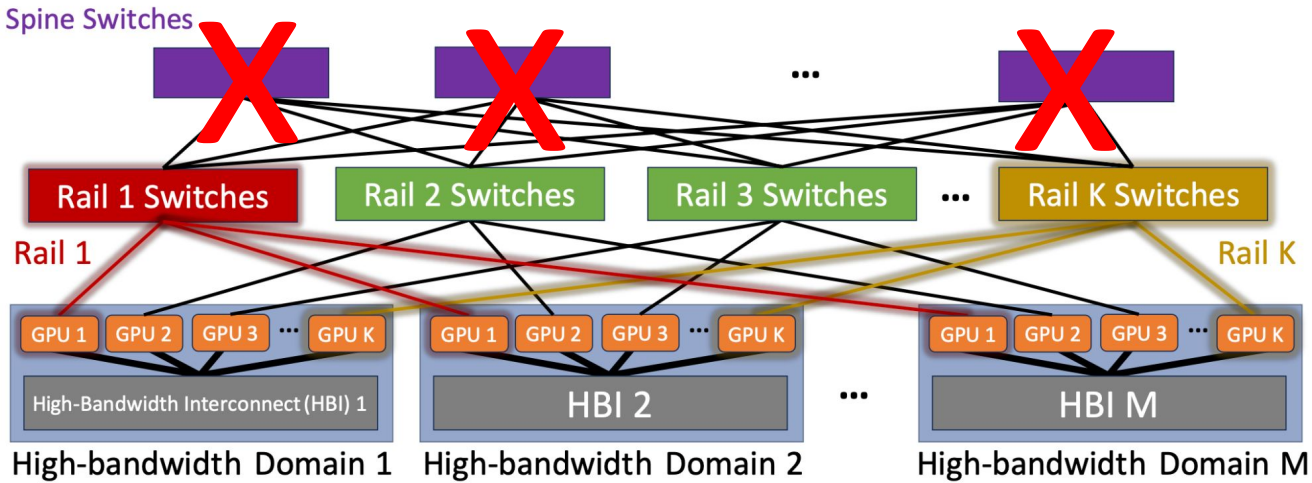


Figure 2: A GPU datacenter with Rail-optimized, any-to-any Clos networks.

Rail-only Network Design

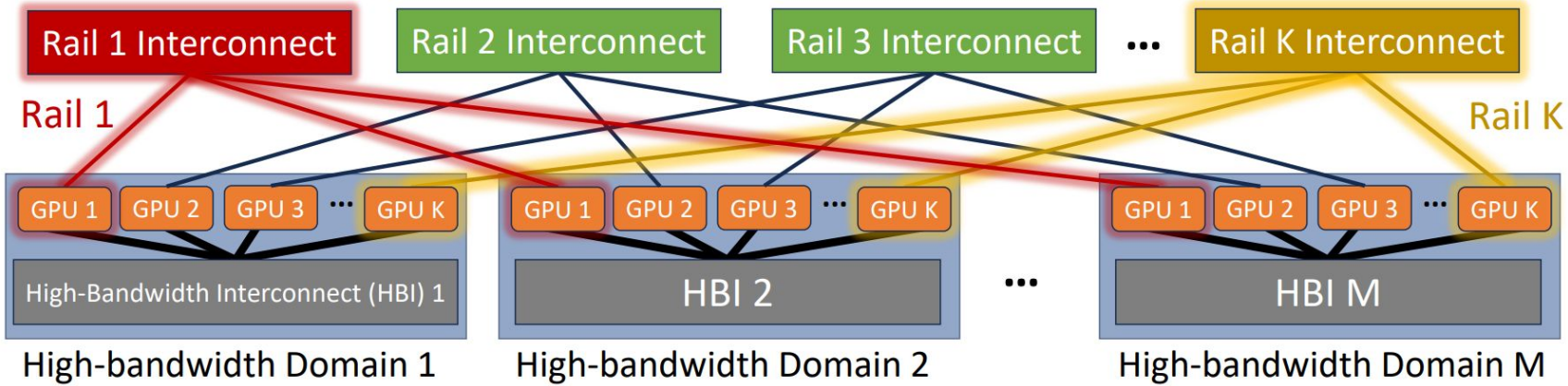
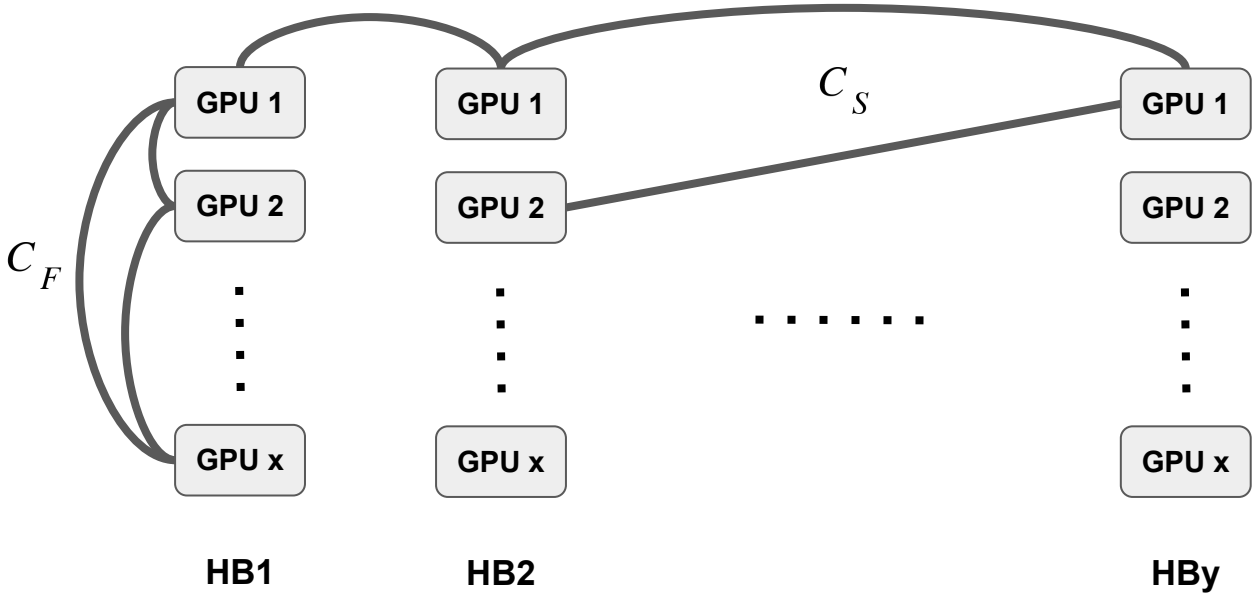


Figure 7: Our proposal: remove the spine layer to form a Rail-only connection.

How to do All-to-All communication?

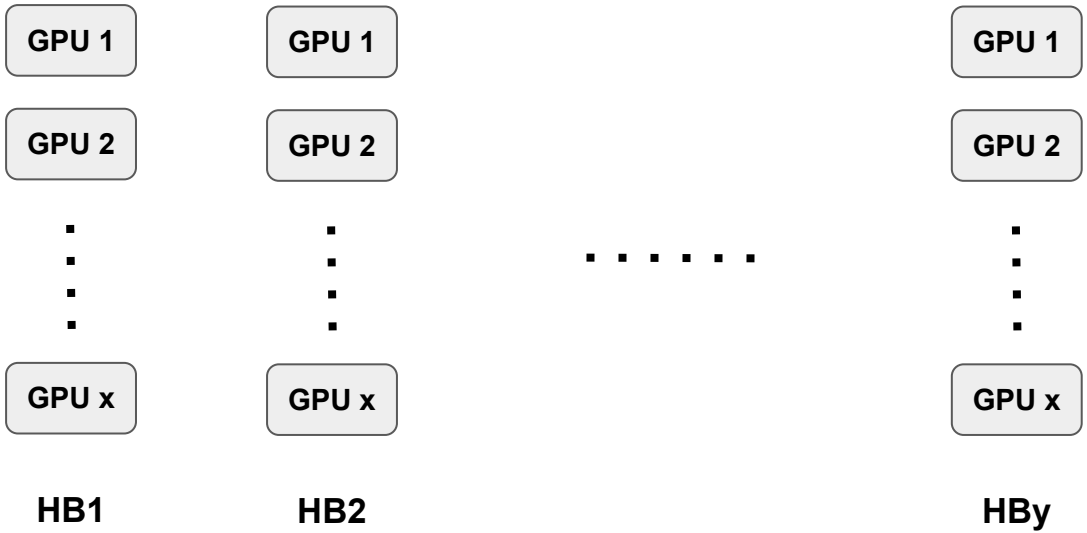
Routing in Rail-optimized Networks



Assume each pair of GPU communicates traffic of size D

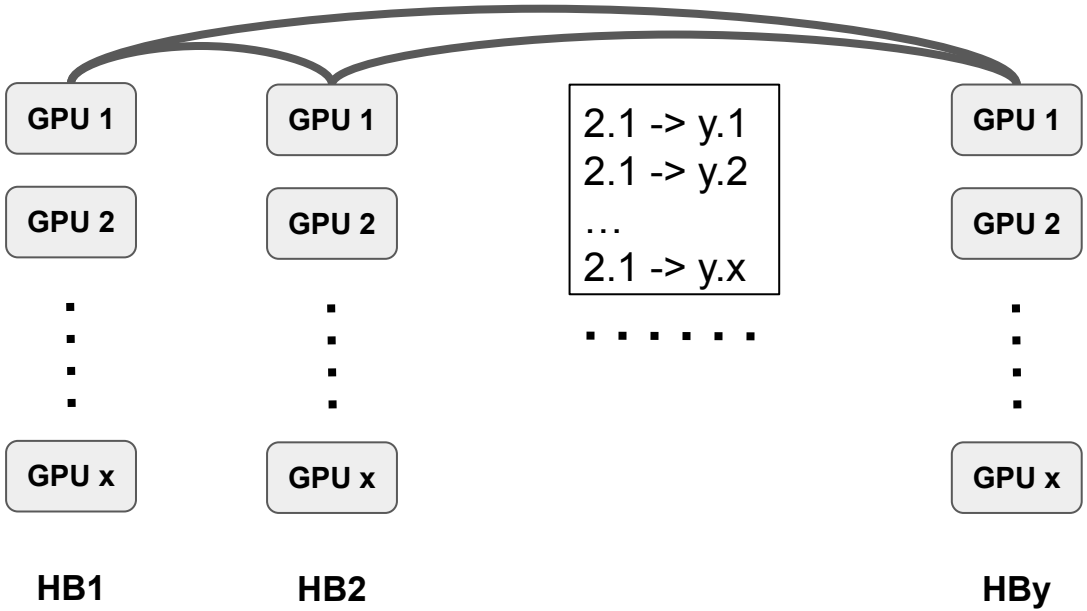
$$T_{a2a}^{Rail-opt} = max\left(\frac{(x-1)D}{C_F}, \frac{x(y-1)D}{C_S}\right) = \frac{x(y-1)D}{C_S}$$

Routing in Rail-only Networks



X.A -> Y.B denotes the message needed to be sent from HB X's GPU A to HB Y's GPU B.
Let's simulate the routing process by focusing on rail 1 (i.e. all messages X.A -> Y.B where A is 1.)

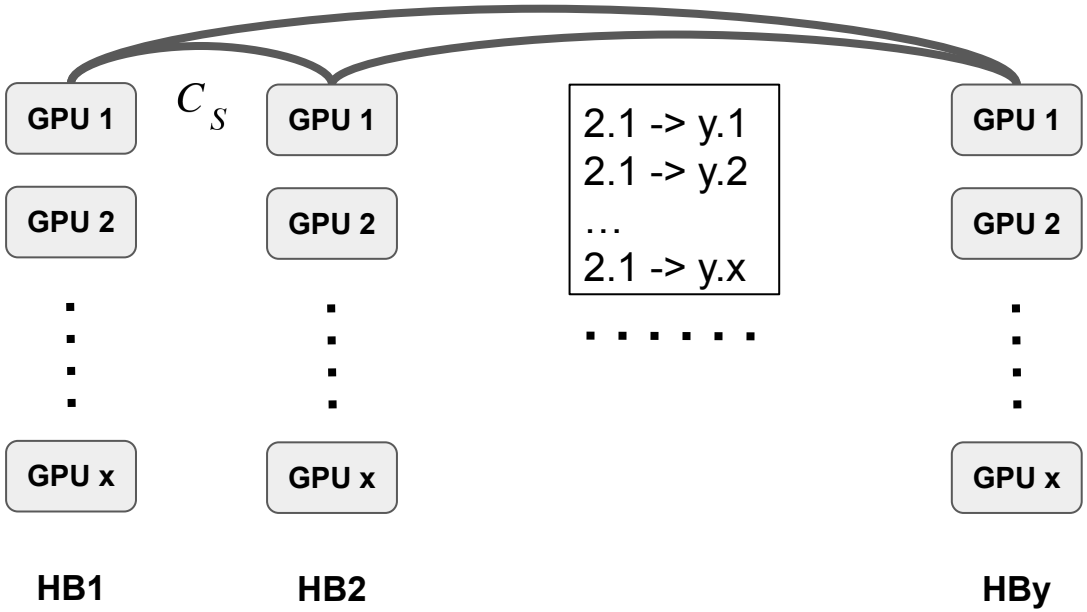
Routing - First Step



X.A -> Y.B : HB X's GPU A -> HB Y's GPU B

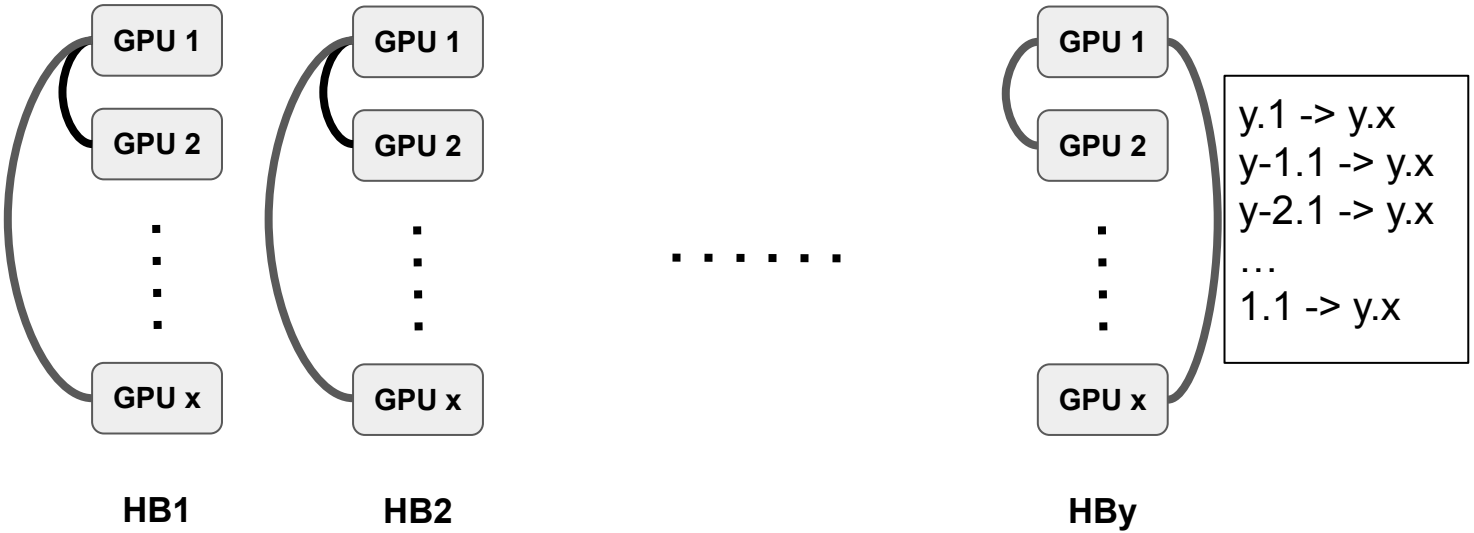
First,with in each rail, each GPU will get all the messages that need to be sent to its HB domain from other GPUs on the same rail.

Routing - First Step



$$T_{all-to-all}^{rail} = \frac{(y-1)\,xD}{C_S}$$

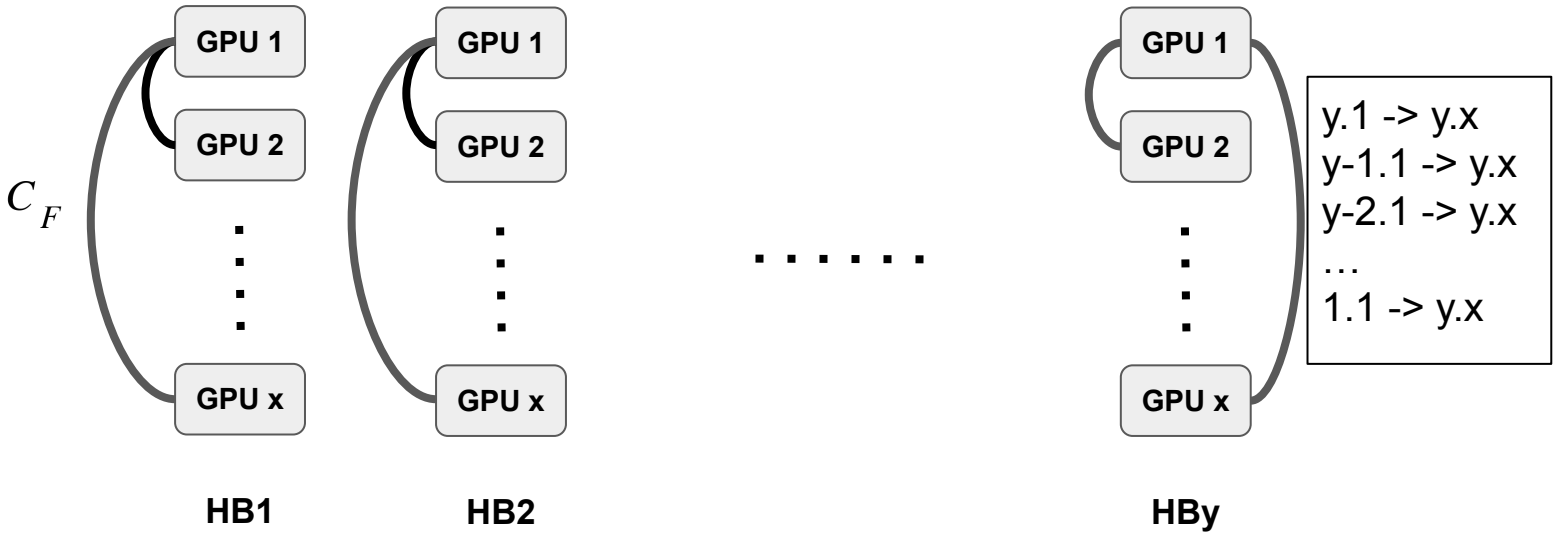
Routing - Second Step



X.A -> Y.B : HB X's GPU A -> HB Y's GPU B

Second, with in each HB domain, GPU 1 will send GPU 2 to GPU x all messages it collects from rail 1.

Routing - Second Step



$$T_{a2a}^{HB} = \frac{(x-1)\,yD}{C_F}$$

Time Comparison

$$T_{a2a}^{Rail-opt} = max(\frac{(x-1)D}{C_F}, \frac{x(y-1)D}{C_S}) = \frac{x(y-1)D}{C_S}$$

$$T_{a2a}^{Rail-only} = T_{a2a}^{rail} + T_{a2a}^{HB} = \frac{x(y-1)D}{C_S} + \frac{y(x-1)D}{C_F}$$

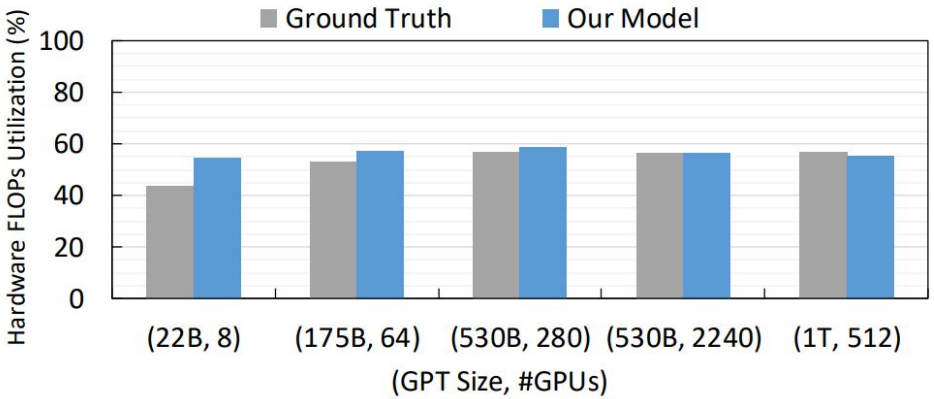
The time differs by y(x-1)D/C_F, which is the time of forwarding within HB domain.

When $y(x-1) \approx x(y-1)$, the slow-down factor is approximately C_S/C_F, which equals to 8.2% and 11.2% for the DGX A100 and DGX H100 generations of GPU platforms, respectively. This slow-down only applies to the all-to-all communication, which accounts for 27% of the total traffic of MoE.

Fault Tolerance

- **Link and switch failures:** GPUs connected to the failed switch or link become unavailable for both Rail-optimized and Rail-only.
- **GPU platform failure:** Migrates the task to another healthy server. Rail-only network connectivity remains the same.
- **Single GPU failures with idle GPU in the HB domain:** The optical switch reconfigures to bring a healthy, idle GPU to replace the failed one.
- **Single GPU failure in fully occupied HB domains:** Two solutions
  - Small HB domains: migrate the tasks on the entire HB domain with the failed GPU to a new one.
  - Large HB domains: add optical switch.

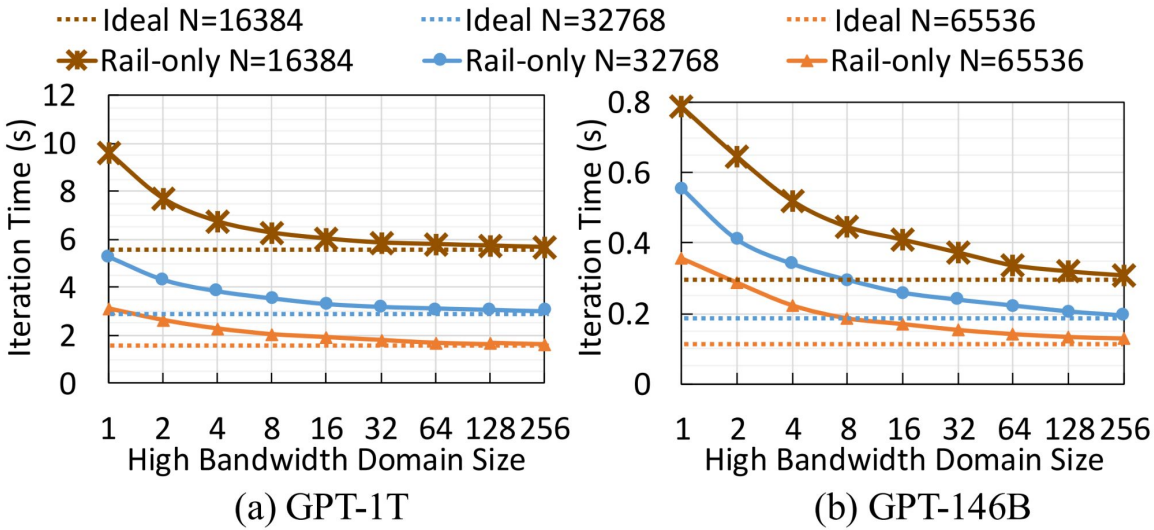
Evaluation



The evaluation result is produced by using an analytical model that considers the critical path for LLM training with TP, DP, and PP.

Ground Truth: published results in the literature.

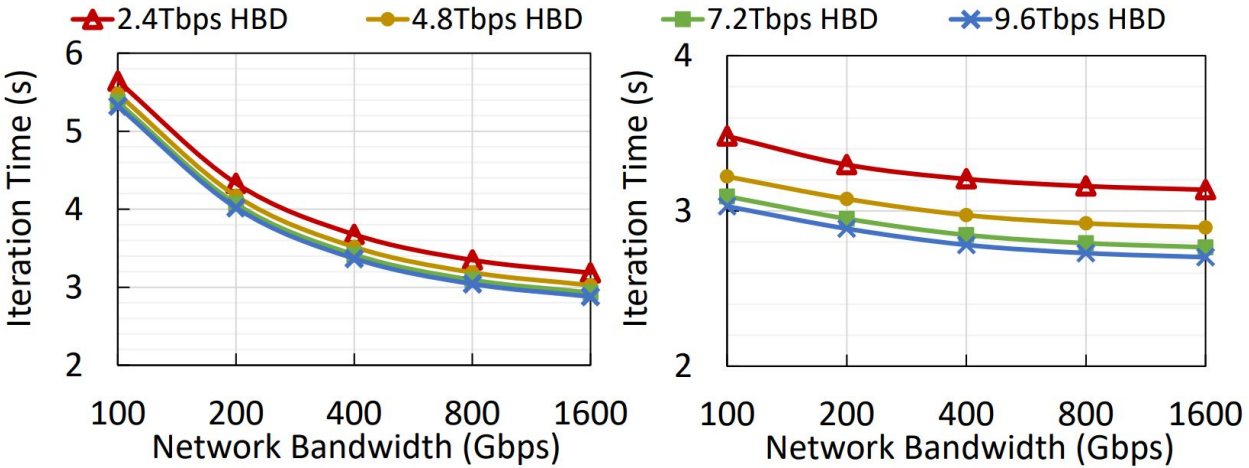
Evaluation - HB Domain Size



Ideally, each pair of GPUs are connected via full-bisection monolithic NVLink. N is the number of GPUS in the cluster.

HB Domain Size: number of GPUs in one High Bandwidth Domain

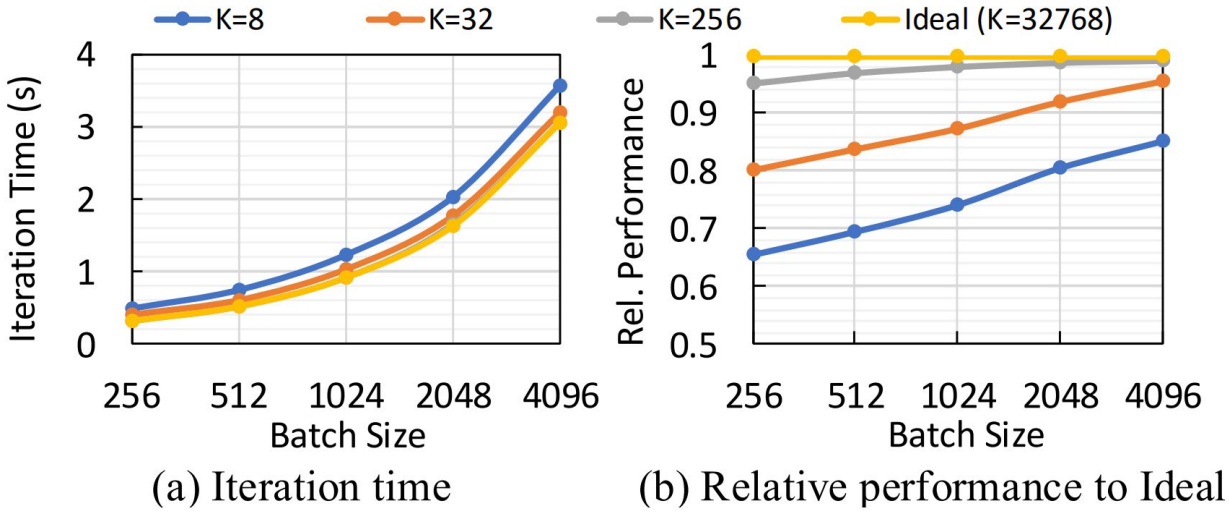
Evaluation - Bandwidth



K is the HB domain size (number of GPUs in a HB domain).

Each different line represents different HBD bandwidth.

Evaluation - Batch Size



(a) Iteration time (b) Relative performance to Ideal

K is High Bandwidth domain size (number of gpus in a HB domain).

Ideally all GPUs are in the same HB domain.

Contribution

Compared to SOTA (rail-optimized network)

- Reduce the network cost by 38% to 77% (117 to 234 million dollars)
- Reduce the power consumption by 37% to 75% (1.7 to 6.9 megawatts)
- while achieving equivalent performance

Q&A

FastFlow

Rail-Only

FF: Smart Offloading algorithm

Recall: High Level Policy

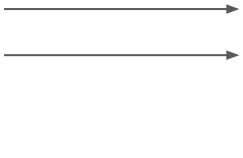
1. When – offload control flow
2. Which – operation/pipeline choice
3. Quantity – offload ratio

Algorithm 1: The smart offloading policy.

Input: Input Pipeline ***P***, model ***M***, conf ***Conf***
Profile $Gthp = thp(\boldsymbol{M})$ and $Lthp = thp(\boldsymbol{P}+\boldsymbol{M})$
if $Gthp <= Lthp$ or $\frac{Gthp}{Lthp} < SpeedUpThreshold$ **then**
    Return ***P*** // No data offloading
else
    $\mathbb{P} = \{p'1, p'2, p'3\}$ = CandidatePipelines(***P***, ***Conf***)
    P' = $\text{argmax}_{\boldsymbol{p'}\in\mathbb{P}} \{Lthp * (1 - \frac{Ocycle(p')}{Pcycle}) + Rthp(p')\}$
    $Upper = Rthp(P')$
    $Lower = (Gthp - Lthp) * (1 + \frac{Ocycle(P')}{Pcycle})$
    if $Lower < Upper$ **then**
        OffRatio = $Upper / Gthp$
    else
        OffRatio = $Rthp / (Lthp + Rthp)$
    Return ApplyOffRatio(***P'***, OffRatio)

FF: Smart Offloading: 1. Offload Choice

When – offload control flow



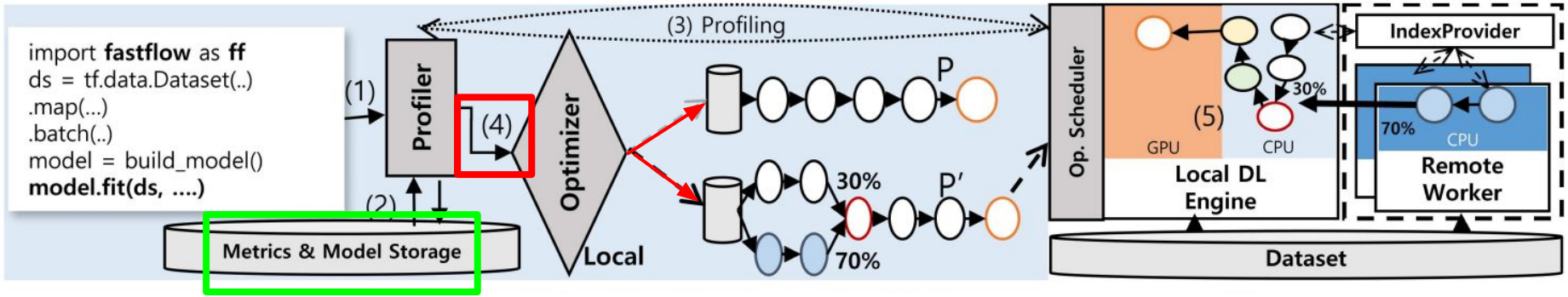
Algorithm 1: The smart offloading policy.

1 **Input:** Input Pipeline ***P***, model ***M***, conf ***Conf***
2 Profile $Gthp = thp(\boldsymbol{M})$ and $Lthp = thp(\boldsymbol{P}+\boldsymbol{M})$
3 **if** $Gthp <= Lthp$ or $\frac{Gthp}{Lthp} < SpeedUpThreshold$ **then**
4 **Return** ***P*** // No data offloading
5 **else**
6 $\mathbb{P} = \{p'1, p'2, p'3\}$ = CandidatePipelines(***P***, ***Conf***)
7 ***P'*** = $\text{argmax}_{p' \in \mathbb{P}} \{Lthp * (1 - \frac{Ocycle(p')}{Pcycle}) + Rthp(p')\}$
8 $Upper = Rthp(P')$
9 $Lower = (Gthp - Lthp) * (1 + \frac{Ocycle(P')}{Pcycle})$
10 **if** $Lower < Upper$ **then**
11 OffRatio = $Upper / Gthp$
12 **else**
13 OffRatio = $Rthp / (Lthp + Rthp)$
14 **Return** ApplyOffRatio(***P'***, OffRatio)

FF: Smart Offloading: 1. Offload Choice

(2)(3) Profile **locally** + store indexed metrics before training

- GPU throughput
- E2E throughput
- CPU Cycles



(4.a) Choose offload decision

FF: Smart Offloading: 2. Pipeline Choice

Which – operation/pipeline choice

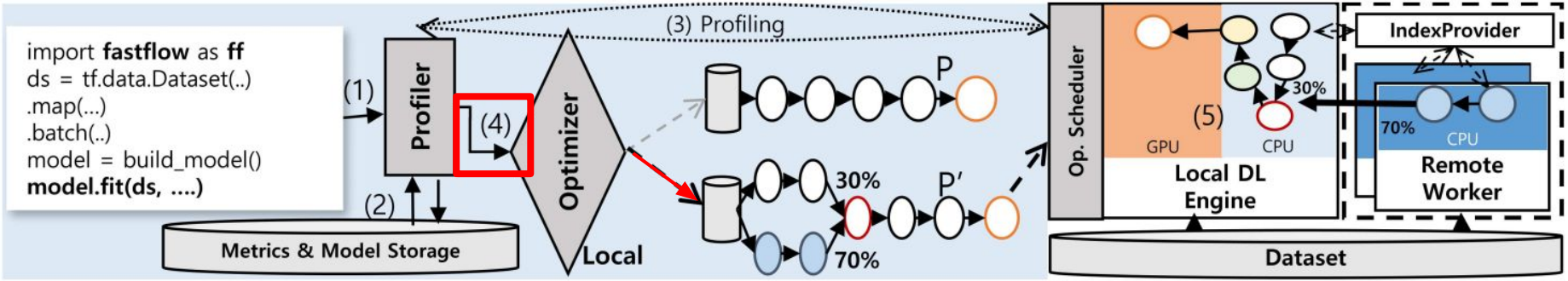
Algorithm 1: The smart offloading policy.

1 **Input:** Input Pipeline ***P***, model ***M***, conf ***Conf***
2 Profile $Gthp = thp(\boldsymbol{M})$ and $Lthp = thp(\boldsymbol{P}+\boldsymbol{M})$
3 **if** $Gthp <= Lthp$ or $\frac{Gthp}{Lthp} < SpeedUpThreshold$ **then**
4 **Return** ***P*** // No data offloading
5 **else**
6 $\mathbb{P} = \{p'1, p'2, p'3\}$ = CandidatePipelines(***P***, ***Conf***)
7 $\boldsymbol{P'} = \text{argmax}_{\boldsymbol{p'}\in\mathbb{P}} \{Lthp * (1 - \frac{Ocycle(p')}{Pcycle}) + Rthp(p')\}$
8 $Upper = Rthp(P')$
9 $Lower = (Gthp - Lthp) * (1 + \frac{Ocycle(P')}{Pcycle})$
10 **if** $Lower < Upper$ **then**
11 OffRatio = $Upper / Gthp$
12 **else**
13 OffRatio = $Rthp / (Lthp + Rthp)$
14 **Return** ApplyOffRatio(***P'***, OffRatio)

FF: Smart Offloading: 2. Pipeline Choice

(2)(3) Profile **locally** + store indexed metrics before training

- GPU throughput
- E2E throughput
- CPU Cycles



(4.b) Choose advantageous pipeline

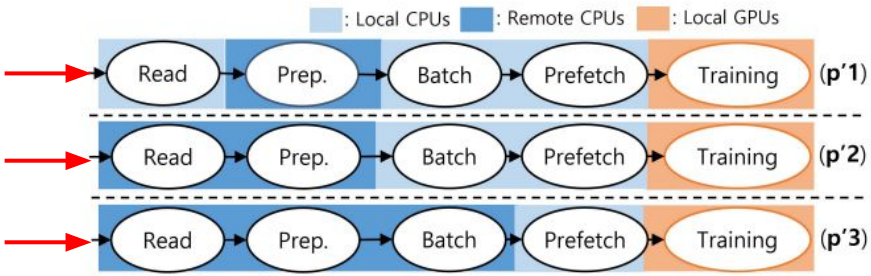


Figure 5: Candidate pipelines for offloading.

FF: Smart Offloading: 3. Offload Ratio

- ***Upper***: The capacity that remote CPUs can hold before CPU bottlenecks occur.
- ***Lower***: The mismatch throughput between local input pipeline and GPUs (+ offloading overheads)

Algorithm 1: The smart offloading policy.

Input: Input Pipeline ***P***, model ***M***, conf ***Conf***
Profile $Gthp = thp(\boldsymbol{M})$ and $Lthp = thp(\boldsymbol{P}+\boldsymbol{M})$
if $Gthp <= Lthp$ or $\frac{Gthp}{Lthp} < SpeedUpThreshold$ **then**
    Return ***P*** // No data offloading
else
    $\mathbb{P} = \{p'1, p'2, p'3\}$ = CandidatePipelines(***P***, ***Conf***)
    P' = $\text{argmax}_{p' \in \mathbb{P}} \{Lthp * (1 - \frac{Ocycle(p')}{Pcycle}) + Rthp(p')\}$
    $Upper = Rthp(P')$
    $Lower = (Gthp - Lthp) * (1 + \frac{Ocycle(P')}{Pcycle})$
    if $Lower < Upper$ **then**
        OffRatio = $Upper / Gthp$
    else
        OffRatio = $Rthp / (Lthp + Rthp)$
    Return ApplyOffRatio(***P'***, OffRatio)

Limitations

TABLE II

NUMBER OF SWITCHES AND TRANSCEIVERS FOR DIFFERENT CLUSTERS.

#GPUs	Switch Radix	#Switches (N_{SW})		#Transceivers (N_{TR})		Savings	
		SOTA	Rail-only	SOTA	Rail-only	Cost	Pwr
32768	64	2560	1536	196608	131072	38%	37%
	128	1280	256	196608	65536	77%	75%
	256	384	128	131072	65536	62%	60%
65536	64	5120	3072	393216	262144	38%	37%
	128	2560	1536	393216	262144	38%	37%
	256	1280	256	393216	131072	77%	75%

Related Work & Discussion

- **LLM trend**
  - Ongoing other works: reduce language models' size and resource requirements without compromising performance
  - Rail-only
- **LLM inference**
  - Each HB domain becomes an inference-serving domain with low latency
  - Rail-only connections help load-balance multiple inference domains
- **Multi-job training**
  - The entire cluster can be arbitrarily partitioned by tiling into smaller rectangular partitions
  - Each partition then independently executes a smaller training job
- **ML infrastructure and other ML workloads**
  - While Rail-only architecture focuses on network design specifically for LLMs, the design is efficient for many other DNN workloads when combined with other efforts

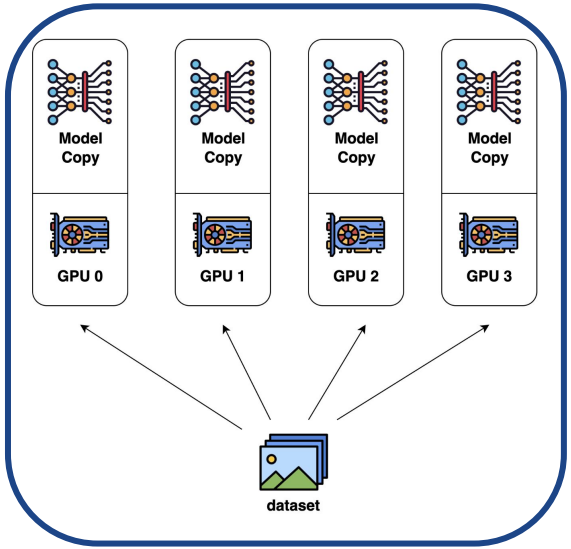
Backup Slides

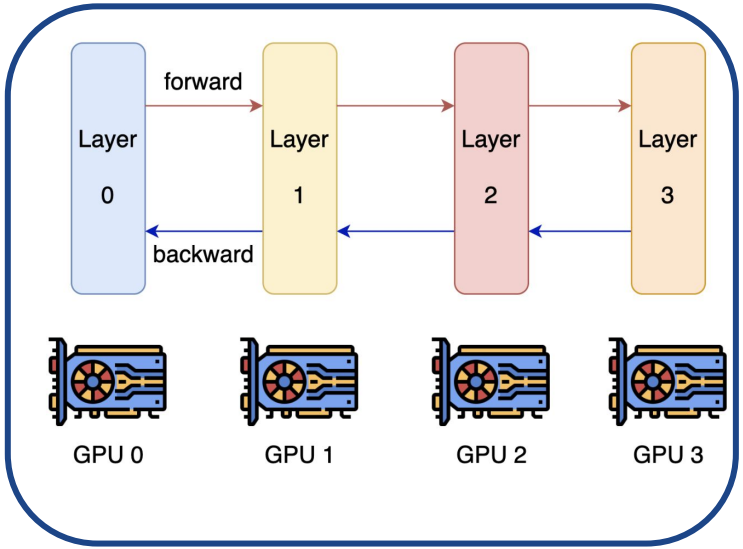
Recap: MegatronLM

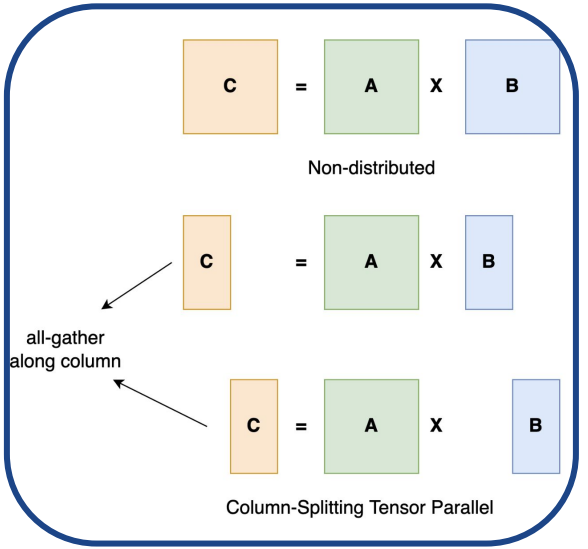
MegatronLM: A pipelining schedule to improve LLM training throughput, composing TP, PP, and DP to scale to thousands of GPUs.

- Symmetric LLM parallelization: When used on models with the same transformer block repeated, each device can be assigned an equal number of transformer layers. Do not consider more asymmetric model architectures in MegatronLM.

Recap: DP, PP, and TP







Data parallelism (DP):

Shard the input dataset, but each worker has a copy of the full model

Pipeline parallelism (PP):

Split the model by layer into several chunks, and each chunk is given to a device

Tensor parallelism (TP):

Split a tensor into N chunks along a specific dimension and each device only holds 1/N of the whole tensor

Intra-platform Connectivity: High-bandwidth Domain

High-bandwidth (HB) domain: GPU-centric platforms optimized for multi-GPU workloads, provisioning several Tbps of **internal** bandwidth across GPUs.

Platform	Technology	Bandwidth
Nvidia's DGX H100 server	8 H100 GPUs interconnected with NVSwitches	7.2 Tbps of nonblocking bandwidth internally
GB200 NVL72 computer	36 GB200 Superchips interconnected with fifth-generation NVLink	14.4 Tbps per GPU
AMD MI300X platform	8 MI300X accelerators interconnected with AMD's Infinity Fabric	7.2 Tbps per GPU
Nvidia's DGX GH200 Supercomputer	256 GPUs interconnected with multi-tiered NVSwich topologies	7.2 Tbps full-bisection intra- GPU bandwidth

Hierarchical Collective Communication Algorithms

Total AllGather runtime:

$$\texttt{AGtime}(D, x, y, C_F, C_S) = \frac{(y-1)D}{xyC_S} + \frac{(x-1)D}{xC_F}$$

TABLE I
VARIABLES USED IN SECTION III AND IV-B.

x	GPU grid dimension in HB domains.
y	GPU grid dimension in NIC domains.
C_F	Bandwidth of HB domains.
C_F	Bandwidth of NIC domains.
D	Data transfer size between a pair of GPUs.
$T_{a2a}^{Rail-opt}$	All-to-all traffic completion time for Rail-optimized networks.
$T_{a2a}^{Rail-only}$	All-to-all traffic completion time for Rail-only networks.

Hierarchical Collective Communication Algorithms

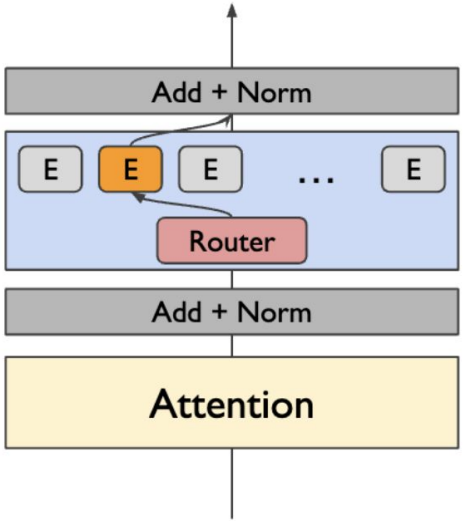
Total AllGather runtime:

$$\texttt{AGtime}(D, x, y, C_F, C_S) = \frac{(y-1)D}{xyC_S} + \frac{(x-1)D}{xC_F}$$

TABLE I
VARIABLES USED IN SECTION III AND IV-B.

x	GPU grid dimension in HB domains.
y	GPU grid dimension in NIC domains.
C_F	Bandwidth of HB domains.
C_F	Bandwidth of NIC domains.
D	Data transfer size between a pair of GPUs.
$T_{a2a}^{Rail-opt}$	All-to-all traffic completion time for Rail-optimized networks.
$T_{a2a}^{Rail-only}$	All-to-all traffic completion time for Rail-only networks.

Recap: Mixture-of-Expert (MoE)



Transformer MoE Layer

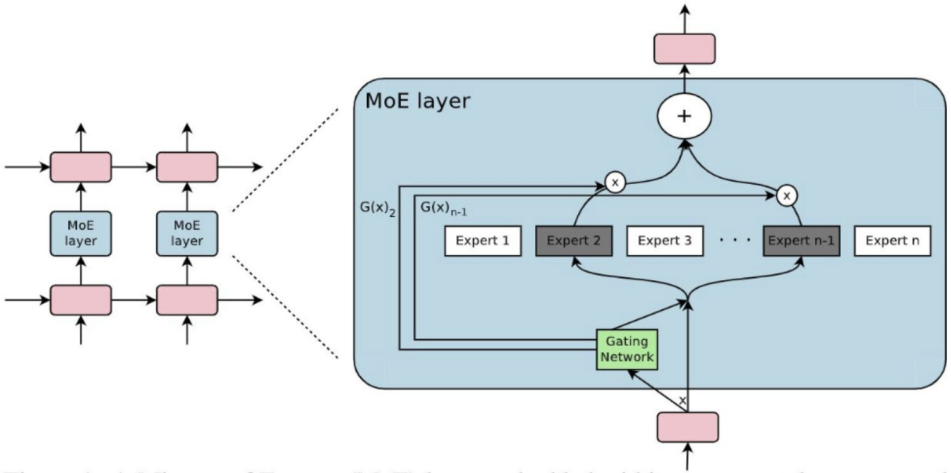


Figure 1: A Mixture of Experts (MoE) layer embedded within a recurrent language model. In this case, the sparse gating function selects two experts to perform computations. Their outputs are modulated by the outputs of the gating network.

Rank in MPI

Rank in MPI (message passing interface)

Rank is a logical way of numbering processes. For instance, you might have 16 parallel processes running; if you query for the current process's rank via MPI_Comm_rank you'll get 0-15. Rank is used to distinguish processes from one another.

The rank value is between 0 and #procs-1. The rank value is used to distinguish one process from another.

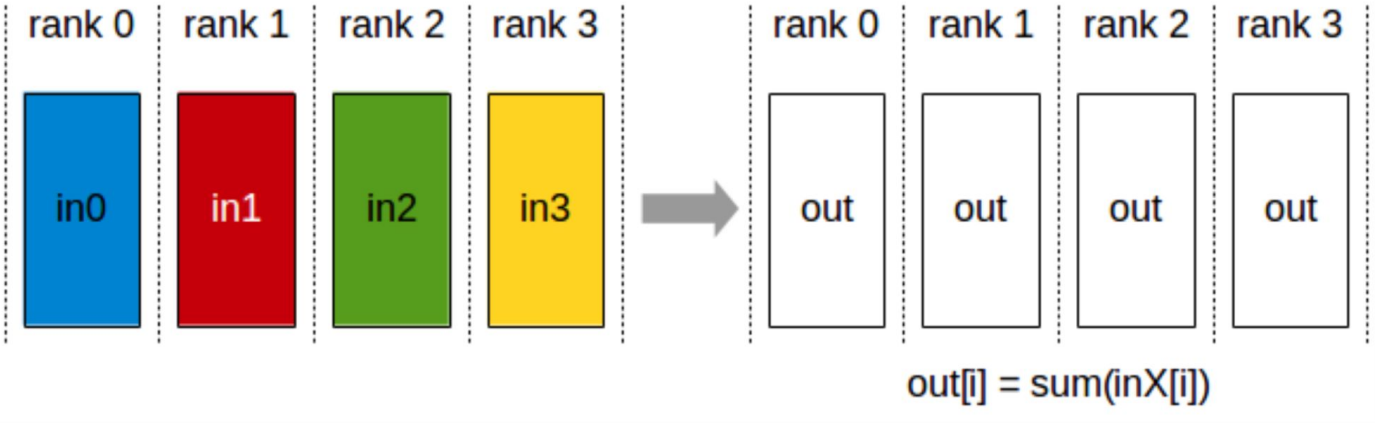
Collective Operations

AllGather and ReduceScatter traffic from tensor parallelism (TP), AllReduce traffic from data parallelism (DP)

https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html

AllReduce

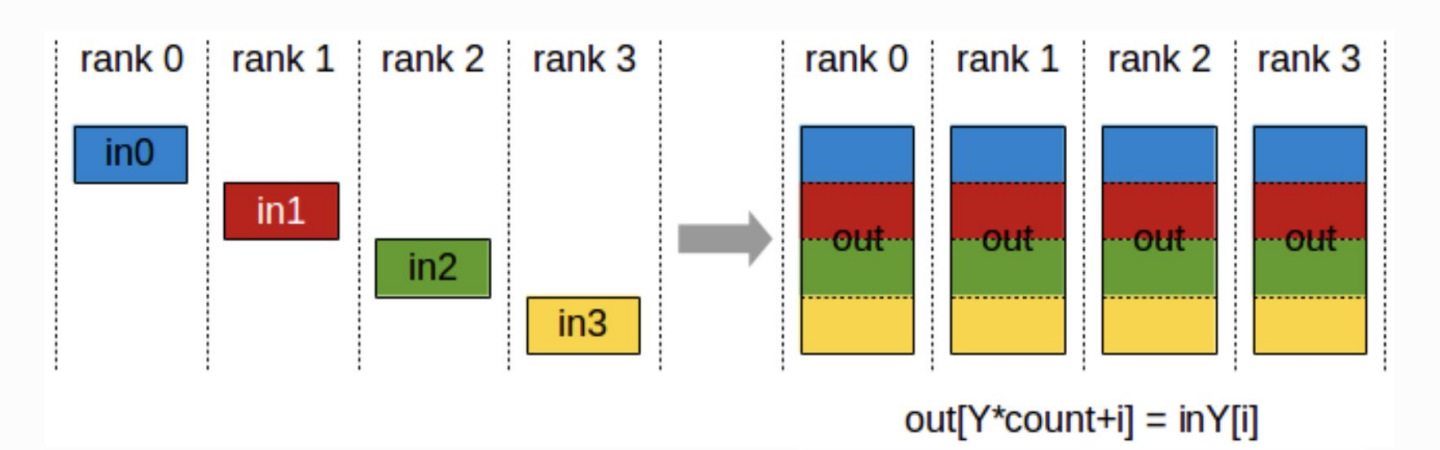
The AllReduce operation performs reductions on data (for example, sum, min, max) across devices and stores the result in the receive buffer of every rank.



All-Reduce operation: each rank receives the reduction of input values across ranks.

AllGather

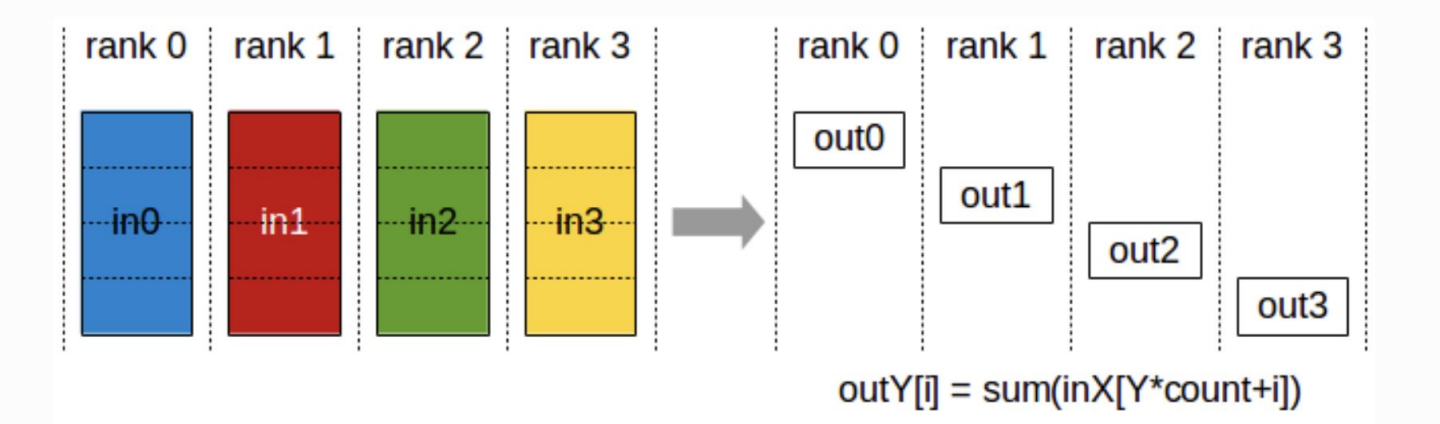
The AllGather operation gathers N values from k ranks into an output buffer of size k*N, and distributes that result to all ranks.



AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

ReduceScatter

The ReduceScatter operation performs the same operation as Reduce, except that the result is scattered in equal-sized blocks between ranks, each rank getting a chunk of data based on its rank index.



Reduce-Scatter operation: input values are reduced across ranks, with each rank receiving a subpart of the result.