Understanding Stragglers in Large Model Training Using What-if Analysis

Using a 5-Month Production Trace from ByteDance

Jinkun Lin, New York University; Ziheng Jiang, Zuquan Song, Sida Zhao, and Menghan Yu, ByteDance Seed; Zhanghan Wang, New York University; Chenyuan Wang, ByteDance Seed; Zuocheng Shi, Zhejiang University; Xiang Shi, ByteDance; Wei Jia, Zherui Liu, Shuguang Wang, Haibin Lin, and Xin Liu, ByteDance Seed; Aurojit Panda and Jinyang Li, New York University

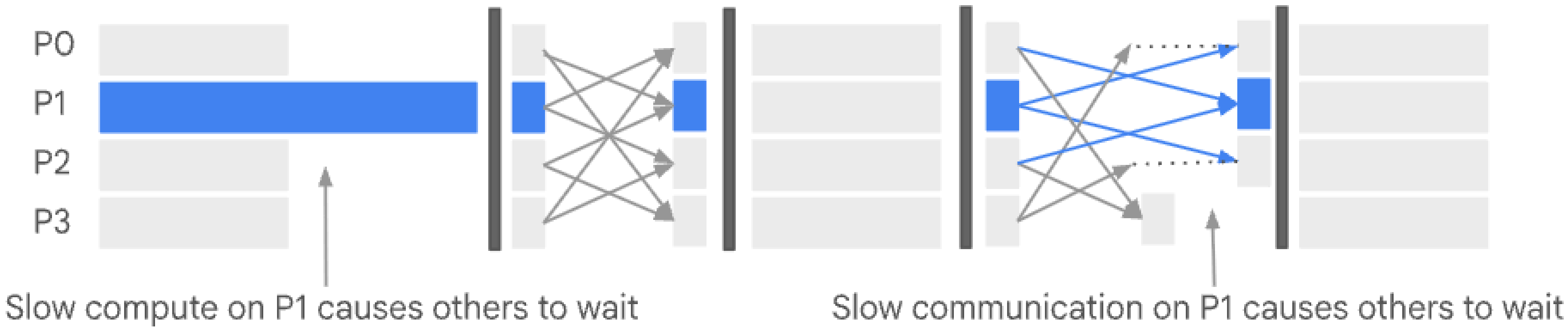
Anant Goyal

What Are Stragglers?

Problem: Distributed LLM training requires frequent synchronization and tighter coordination

- **Straggler**: Worker that lags behind peers, stalling the entire training job
- **Worker**: Single GPU + controlling CPU process

Why it matters: All workers must synchronize → slowest worker blocks everyone



Source: Google Cloud Blog

Scale of the Problem

- Empirically shown: larger model size = improved task accuracy
  - Meta Llama: 65B params (Llama-1) → 405B params (Llama-3)

- Traditional straggler mitigation **doesn't work**:
  - Backup workers → bad training performance (assumes infrequent synchronization)
  - Drop slow updates → hurts model accuracy

Key Question: Do stragglers pose performance issues in real-world LLM training?
Answer: YES!

Key Takeaways

🚨 **42.5%** of jobs are ≥10% slower due to stragglers; tail jobs waste **45% of GPU hours**

- Root causes are **software & workload**, not hardware failures (only 1.7% of cases)

⚠️ **Most Surprising:**

- **Compute**, not communication, is the bottleneck — especially on a well-optimized network
- Straggling has **no correlation with job size** — more GPUs doesn't mean more stragglers
- A simple **Python Garbage Collection tweak** was hiding in plain sight

Study Overview

Setup

- 3,079 training jobs with each using between 128 and > 5000 GPUs
- Network tuned to ensure **no slowdown from congestion**

Design Choice: Jobs don't share servers

- Eliminates stragglers from resource contention
- Any observed stragglers are due to training dynamics, **not infrastructure**

Estimating "Straggler-Free" Performance

Key Insight: Comparable operations should have same duration in absence of stragglers

1. Log (trace) operation type, start/end timestamps, metadata
2. Calculate **idealized duration** for a job, T_{ideal}
3. Calculate **slowdown** metrics and **GPU hours wasted**

$$S = \frac{T}{T_{ideal}} \quad \text{and} \quad \%\, resource\ waste = 1 - \frac{1}{S}$$

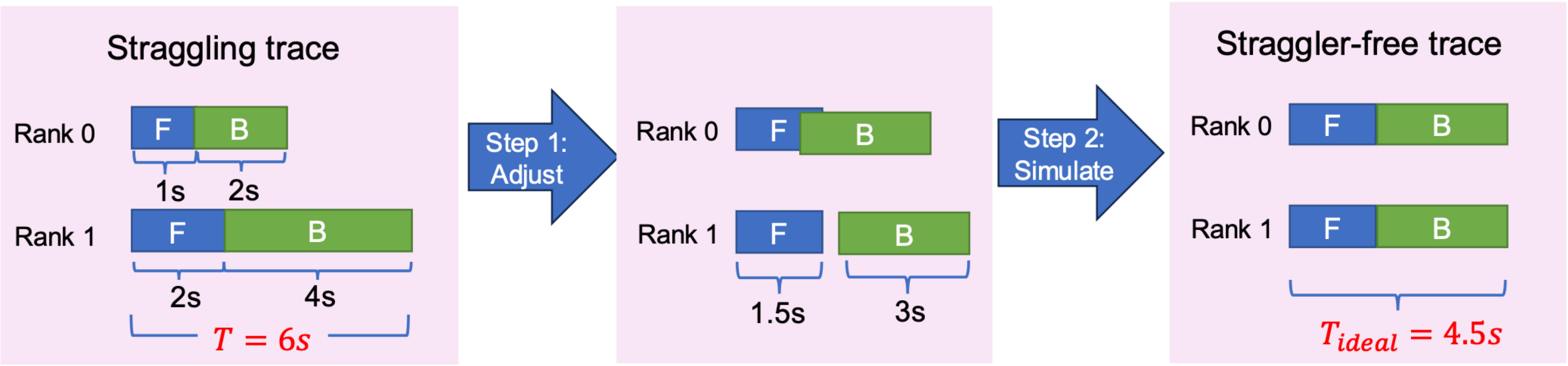
How do we calculate T_{ideal}?

For Compute Operations: Use **average** duration across all workers

- Rationale: Same workload → should take same time

For Communication Operations: Use **median** duration

- Rationale: Affected operations take very long (switch/NIC flapping)



Source: Original author slides from OSDI '25

Finding #1 - GPU Hours Wasted

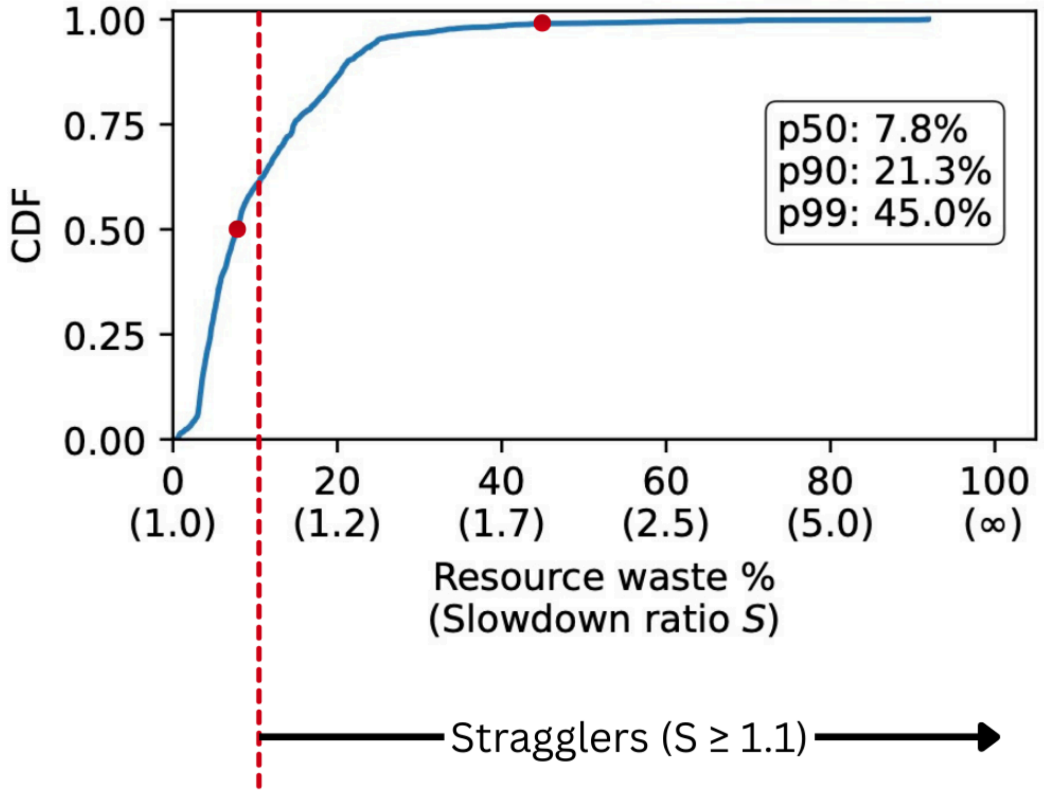
→ Stragglers cause **significant resource waste**

- **>50% of jobs**: waste 7.8% of GPU hours
- **~1% of jobs**: waste 45% of GPU hours

At distrubuted LLM scale with thousands of GPUs:

- 7.8% waste = hundreds of GPU-hours
- GPU-hours wasted → tens of thousands $

Cumulative distribution function (CDF) of resource waste among all jobs



Finding #2 - Compute vs Communication

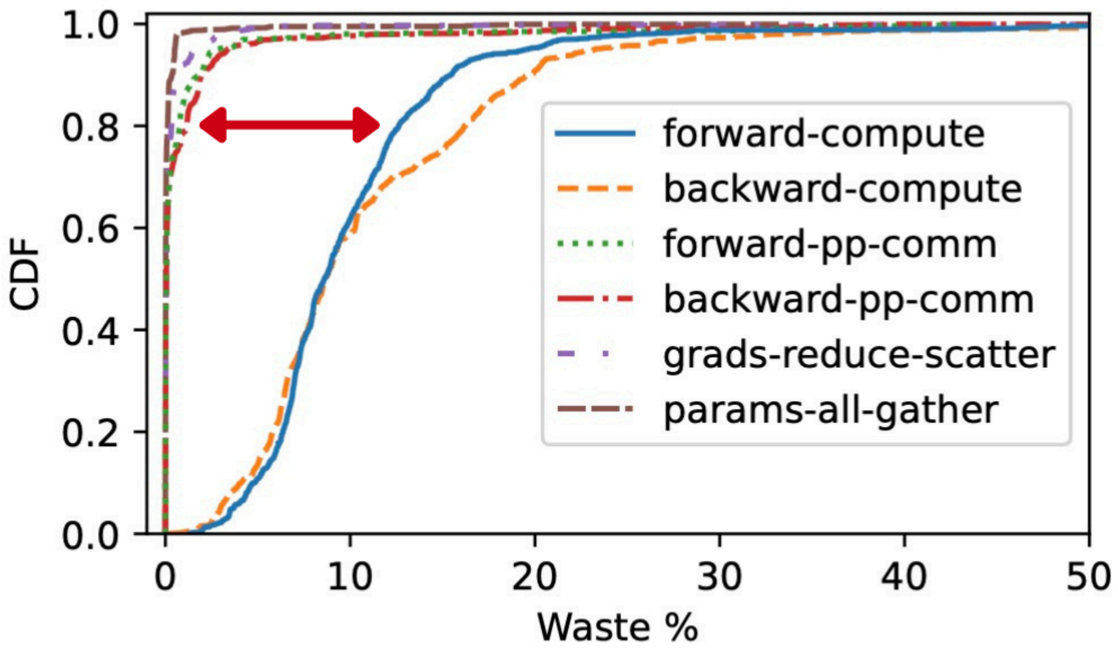
→ Most waste caused by **compute operations, NOT communication**

Why?

- ByteDance optimized in-house network
- Ample network bandwidth provisioned
- Network tuned to avoid congestion

⚠️ **Counterintuitive**: Most assume network is the bottleneck

Cumulative distribution function (CDF) of resource wasted by operation type



Other Findings

Step-Level Slowdown is Persistent

- 90th percentile of steps: slowdown of only **1.06x**
- Small per-step slowdowns compound over thousands of training steps

⚠️ Sufficient to sample just a few steps to identify stragglers

No Correlation Between Job Size and Straggling ⚠️ Counterintuitive!

- Larger jobs = more GPUs = *should* mean more straggler risk
- But data shows small and large jobs are equally affected
- Straggling is inherent to **training dynamics** (model type, workload), not scale

Root Cause #1 - Stage Partitioning Imbalance

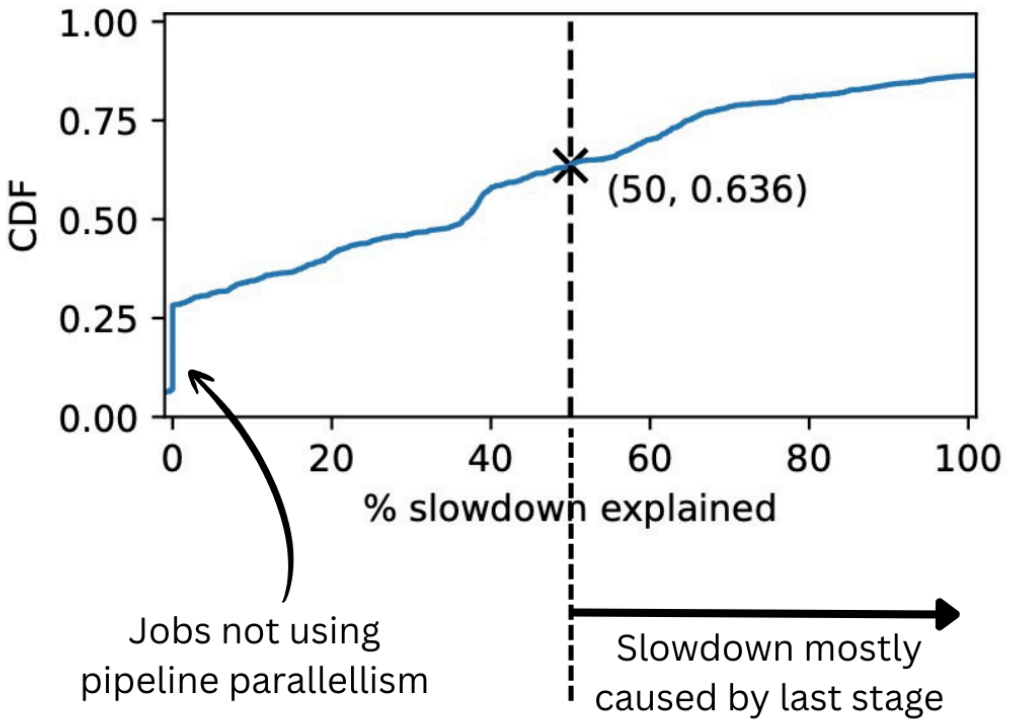
Problem: Pipeline stages divided evenly by layer count

- Last pipeline stage includes **loss layer**
- Loss layer computation >> transformer layers
- Last stage becomes bottleneck

Attempted Fix: Assign fewer layers to last stage

- Result: **<10% speedup** (minimal improvement) due to manual tuning

Cumulative distribution function (CDF) of slowdown caused by the last pipeline stage



Root Cause #2 - Python Garbage Collection

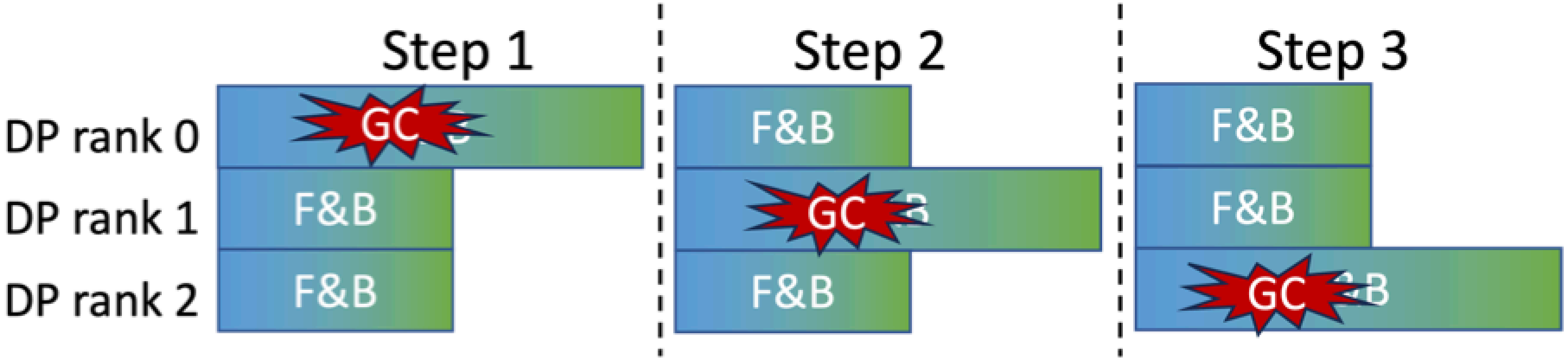
Problem: Forward compute operations run in Python with automatic GC

- "Stop-the-world" GC pauses Python code execution
- Different processes can trigger GC at different times

Fix: Turn off Python automatic GC

- Manually run GC every 500 training steps
- Result: **12.6% improvement**

Implemented, but not widely adopted due to memory challenges



Other Root Causes

Sequence Length Imbalance

- Microbatch compute time ∝ sum of sequence length squares
- **21.4% of jobs** slowed down by sequence length imbalance
- Greedy balancing fix → **23.9% improvement** not deployed; doesn't fix PP-level imbalance

Uncommon Causes

- PyTorch CUDA memory allocator slowness → slows forward & backward pass
- Unrelated kernels sharing CUDA hardware queue → false dependencies block kernel launch

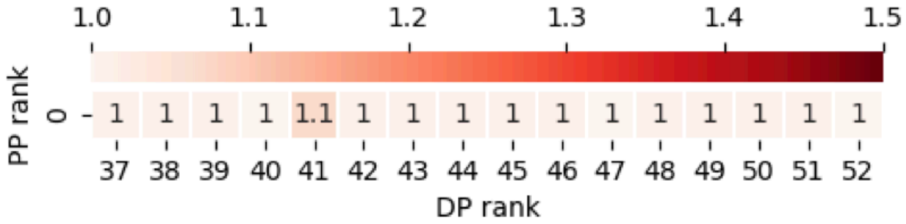
→ These are rare but can cause significant slowdowns in specific jobs

SMon - Straggler Monitoring System

Displays:

- Estimated overall slowdown per job
- Per-step slowdown trends
- Per-worker slowdown identification

Used in production by on-call team to **diagnose and identify stragglers** in real-time



***(a)** Worker issue*

***(b)** Stage partitioning imbalance*

***(c)** Sequence length imbalance*

Related Work

Traditional Straggler Mitigation (MapReduce era)

- Mantri [OSDI '10], Dolly — backup tasks, speculative execution

  ❌ Assumes infrequent sync; breaks under LLM's tight coordination requirements

Stragglers in LLM Training

- FALCON [arXiv 2024]
  - Shared cluster only → sees resource contention stragglers
  - Manual root cause analysis
  - Misses stragglers that affect **most steps**

This paper's contribution: A large-scale, semi-automated empirical study of stragglers in LLM training, with a deployed monitoring tool (SMon)

Study Limitations

Data Coverage

- **Discarded jobs:** repeated failures, trace issues, simulation discrepancy >5%
- May underrepresent worst-case stragglers and hardware-related failures

Profiling Tool (NDTimeline)

- Cannot capture stragglers **within TP or CP groups** — these slow all microbatches uniformly

Cluster-Specific Findings

- Network is **deliberately overprovisioned** — communication bottlenecks may matter more elsewhere

Discussion & Critique

Strengths:

- Large-scale empirical study (5 months, production workload)
- Counterfactual, What-if analysis is rigorous and novel
- Led to practical tool (SMon) and deployed fixes (Planned GC)

Questions

- Forward compute runs in Python, backward runs in C++ — **why not rewrite forward in C++?**
- Can stage partitioning be done **automatically** based on FLOPs rather than manual tuning?
- SMon detects stragglers — can it **actively mitigate** them (e.g., rebalance sequences at runtime)?
- How do findings change in **heterogeneous or shared clusters** where network is not overprovisioned?

Q: Why Not Deploy Sequence Length Balancing?

Result: Greedy balancing → 23.9% throughput improvement in experiments

Why not deployed?

- Only balances across **DP microbatches** — does not fix imbalance at the **PP level**
- Balanced sequence lengths ≠ balanced memory consumption
  - Long sequences require more activation memory → risk of OOM on some workers
- WLB-LLM (the companion paper) addresses this more completely

Takeaway: The fix works locally but doesn't solve the full problem

Q: How Does This Compare to FALCON?

	This Paper	FALCON
Cluster type	Dedicated	Shared
Resource contention stragglers	Not observed	Present
Large job traces (≥512 GPUs)	**562 jobs**	27 jobs
Analysis method	Semi-automated (simulation + what-if)	Manual
Persistent step stragglers	✅ Studied	❌ Overlooked

Key insight: Shared vs. dedicated cluster changes which root causes dominate

- FALCON sees contention stragglers; this paper sees workload imbalance
- Neither result is "wrong" — they reflect different real-world deployment settings

WLB-LLM: Workload-Balanced 4D Parallelism for Large Language Model Training

Zheng Wang, Anna Cai, Xinfeng Xie, Zaifeng Pan, Yue Guan, Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, Yuchen Hao, Yufei Ding

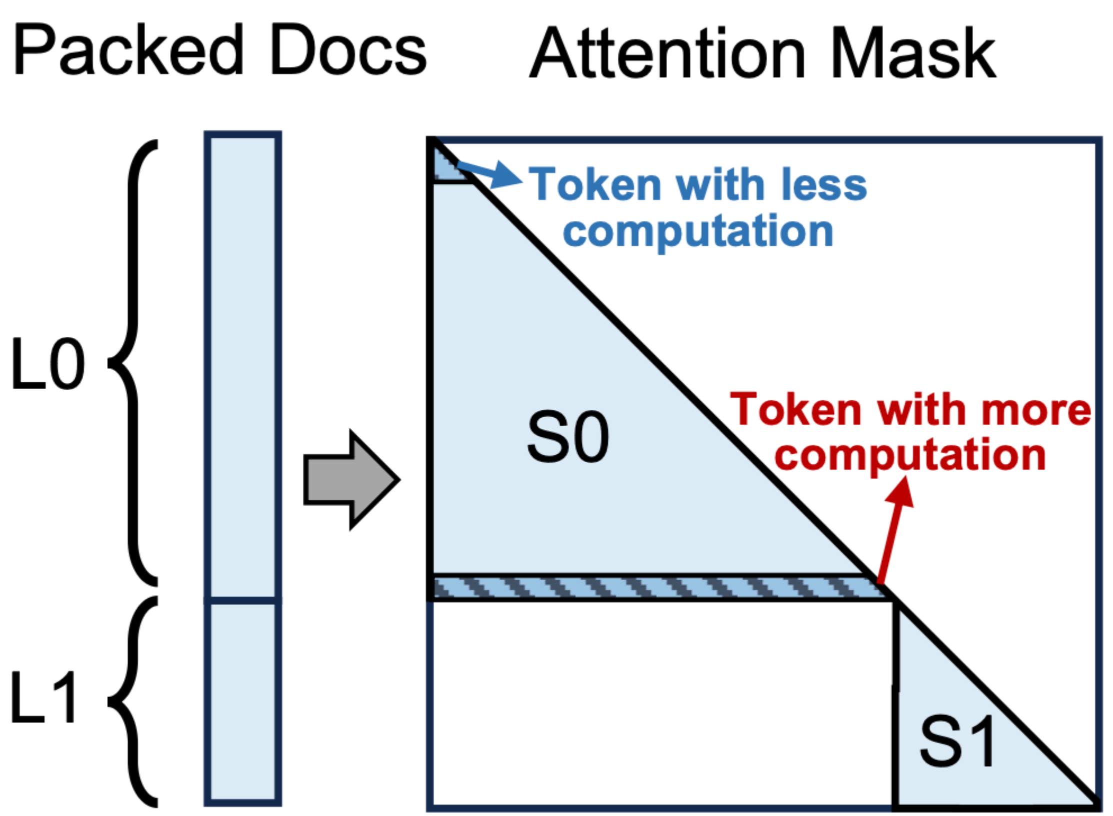
Kartik Ramesh

Introduction

- Llama 3.1 405B trained over 16k GPUs for 30M H100 hours.
- Using AWS H100 pricing **$212M**.
- A 1.2x increase in training speed would have saved **$36M** in cloud costs.
- WLLB-LLM is a work from Meta and UCSD, released ~6 months after their training Llama 3 that achieves this.

Self Attention

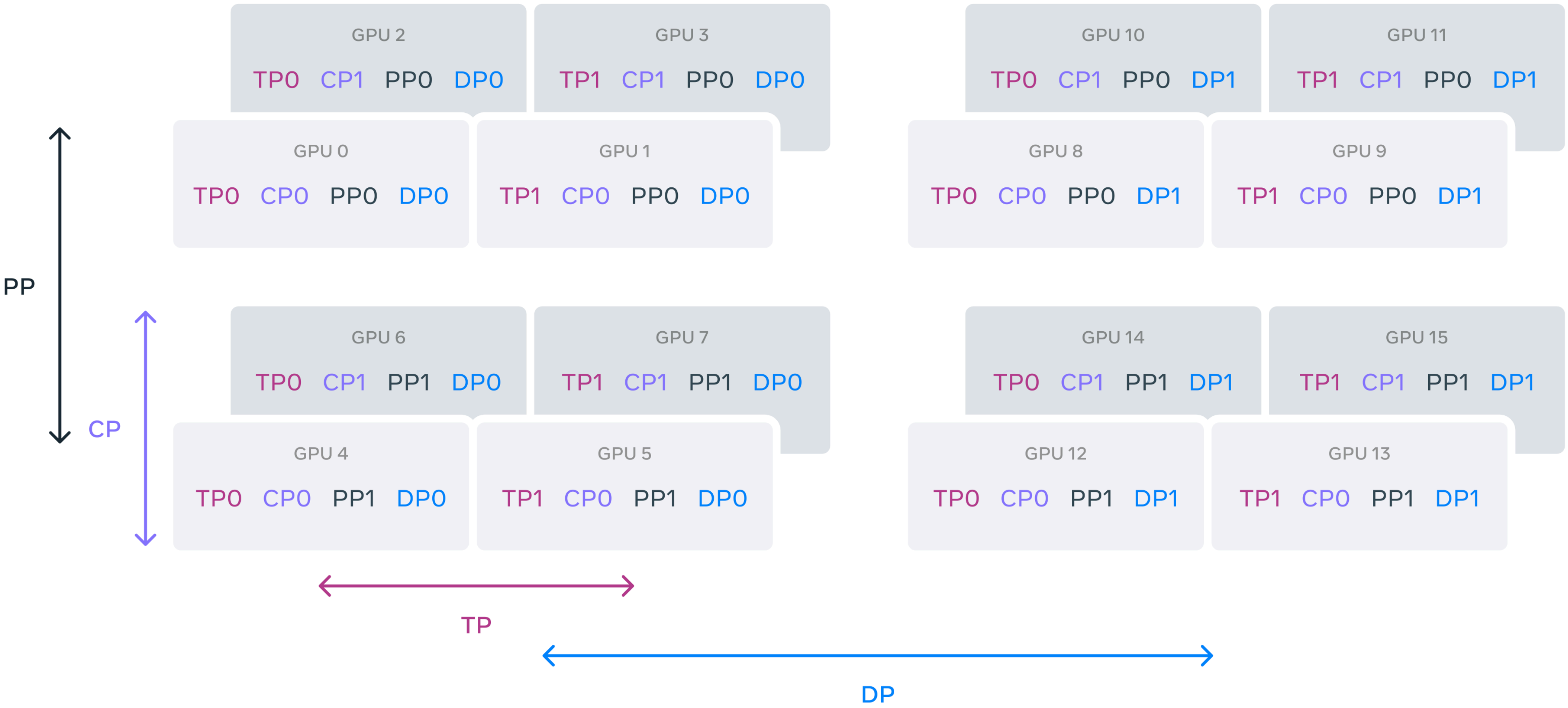
- Attention scales quadratically with the number of past tokens $O(T^2)$
- Not all tokens are equal: Processing later tokens in a document require more computation than earlier.
- When packing multiple documents for training, control attention span using masks.



4D Parallelism

- Huge models are trained on huge clusters because of scale and necessity.
- Tensor shape $[B, T, H]$ across several layers.
- 4D Parallelism splits this shape in various ways.
    - Data - Split B across replicated model.
    - Pipeline - Split model into groups of layers.
    - Context - Split along sequence length T
    - Tensor - Split across the H.

4D Parallelism



$$DP > PP > CP > TP$$

Key Takeaways

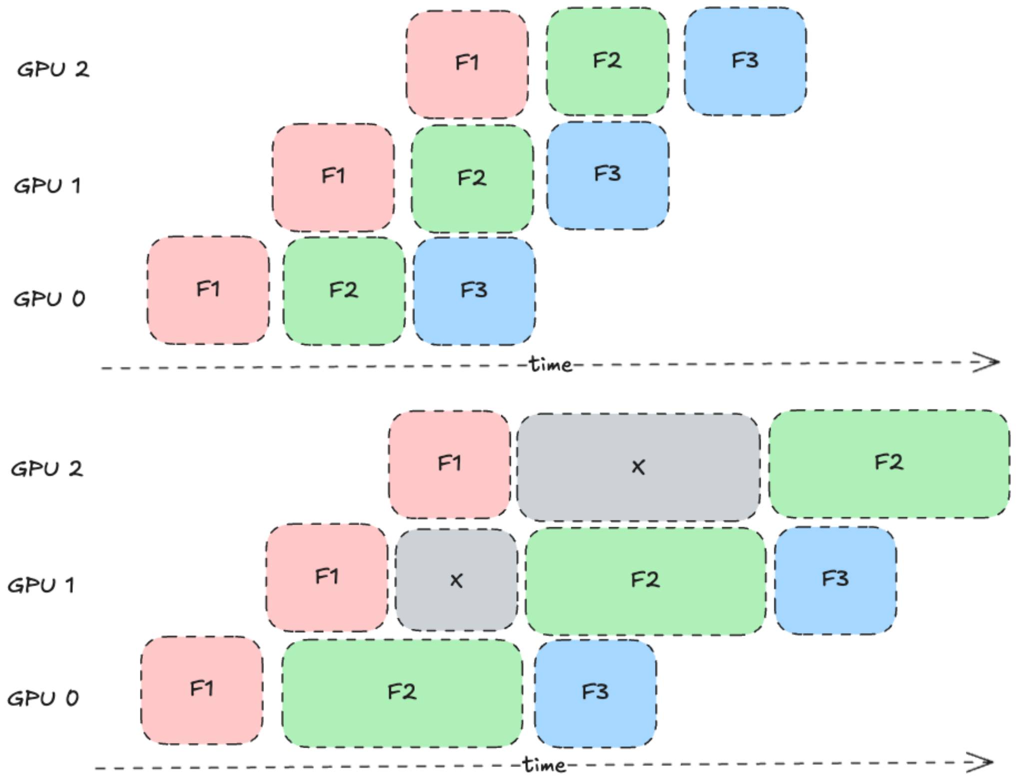
Problem: In long-context 4D training, token count is a weak proxy for compute; attention cost is highly non-uniform.

- **Core idea #1 (PP)**: Reduce PP imbalance via attention-aware micro-batch packing.
- **Core idea #2 (CP)**: Reduce CP imbalance via fine-grained per-document sharding.

Bottom line: WLB-LLM improves training throughput ($\approx 1.23x$) without hurting convergence.

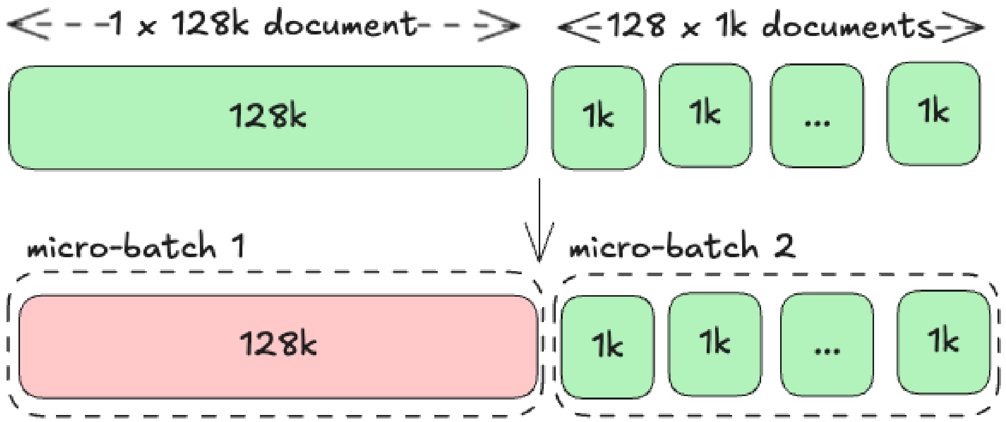
Motivation #1 - Pipeline Imbalance

- Pipeline Parallelism splits B into N μB to hide delays.
- Balanced Batches $\rightarrow$ Higher throughput



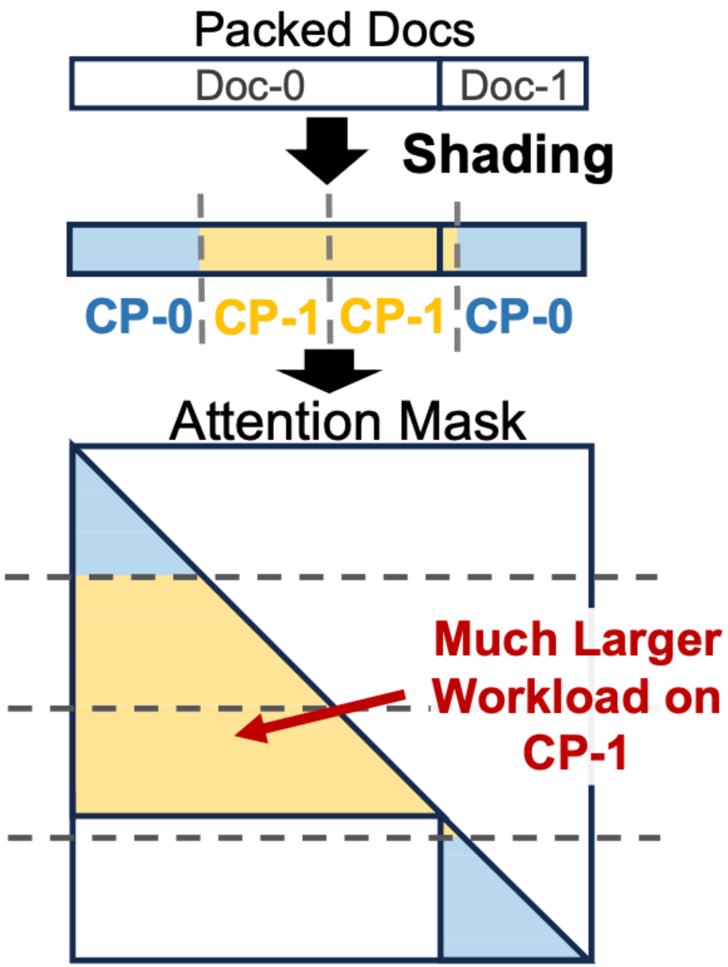
Motivation #1 - Pipeline Imbalance

- Naive approach $\rightarrow$ split by tokens.
- Each μB has uniform sequence length = `CONTEXT_WINDOW_SIZE`
- Does this balance LLM workload?
- $k.128^2 >> k.128 * 1^2$



Motivation #2 - Context Imbalance

- Workload across CP workers should be balanced.
- After packing, split the sequence into $2.CP$ parts and assign one from front and one from back to balance attention.
- Good heuristic, fails for multiple packed documents. Common in long context training.
- Every small delay adds up to higher-order delays.



Baseline: Attention-Aware packing.

Idea: Divide B into μB by estimating d_i^2 as attention cost for each document.

- It works, but limited balancing improvements $\rightarrow$ limited speedup.
- Higher balancing across μB requires balancing across multiple global B. This disturbs the random order of training and loss convergence.
- It might be impossible to come up with such a μB construction, if there are no candidates.

$$
\begin{aligned}
\text{minimize} \quad & \max\Big(\boxed{\sum_{i=1}^{N} x_{ij} \cdot d_i^2}\Big), \quad j = 1, \cdots, M \\
\text{subject to} \quad & \sum_{j=1}^{M} x_{ij} = 1, \qquad i = 1, \cdots, N \\
& \sum_{i=1}^{N} x_{ij} \cdot d_i \leq L, \qquad j = 1, \cdots, M \\
& x_{ij} \in \{0, 1\}
\end{aligned}
$$

Attention Cost

Each Document is present in only one batch

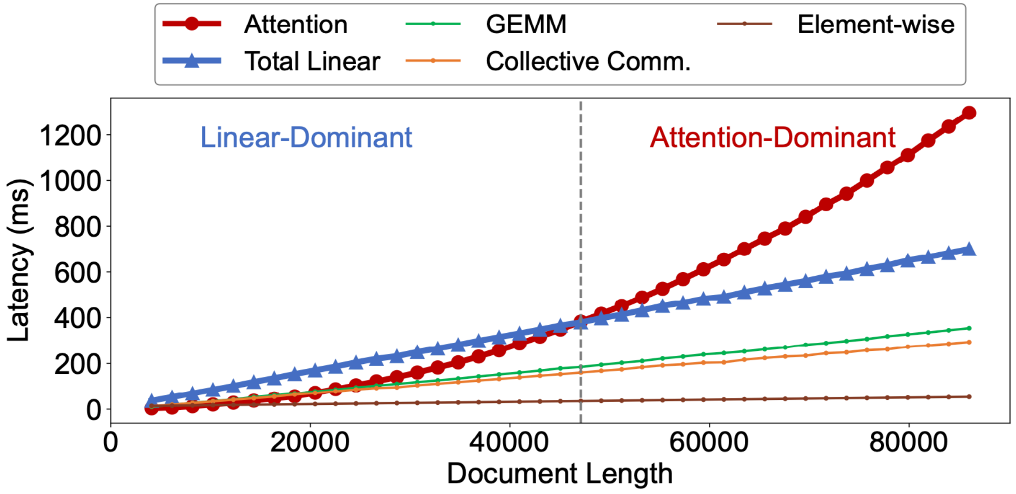
L = Context Size

Variable-Length Packing

Idea: Allow $len(\mu B) > \texttt{CONTEXT\_SIZE}$ for weaker μB

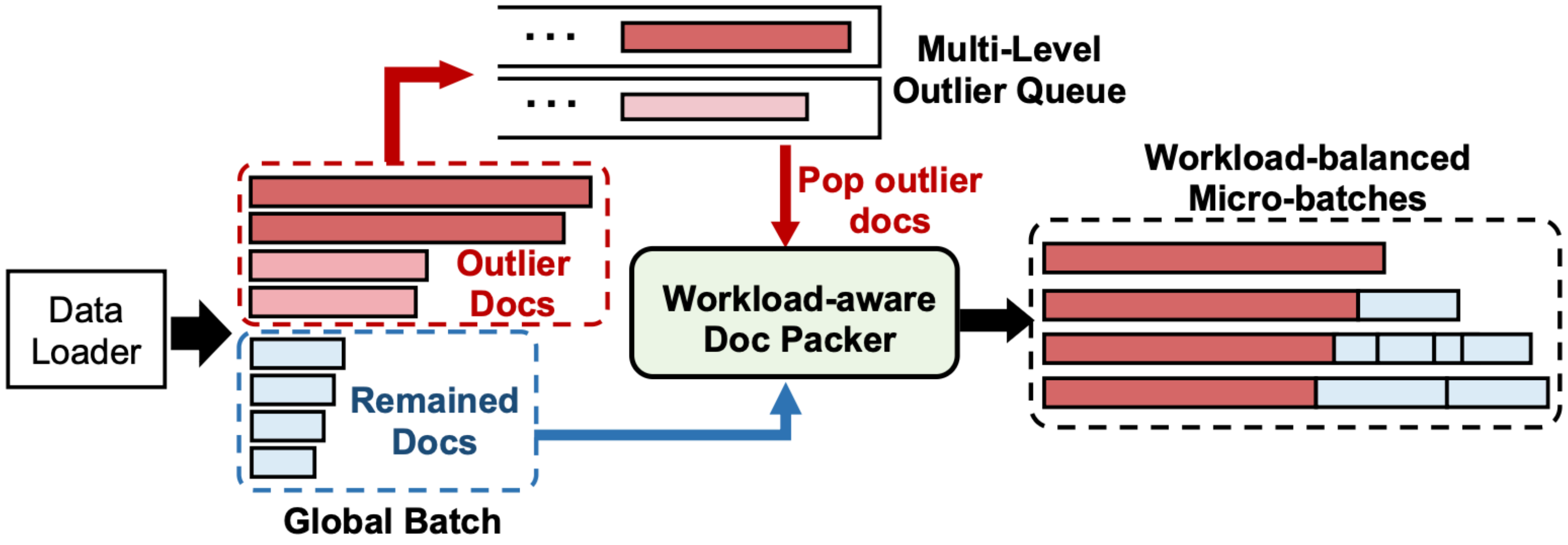
- Attention is Quadratic, but other operations are linear (feed-forward, comms etc.)
- Balance the total workload, not just attention.
- Balance long documents against many shorter documents.

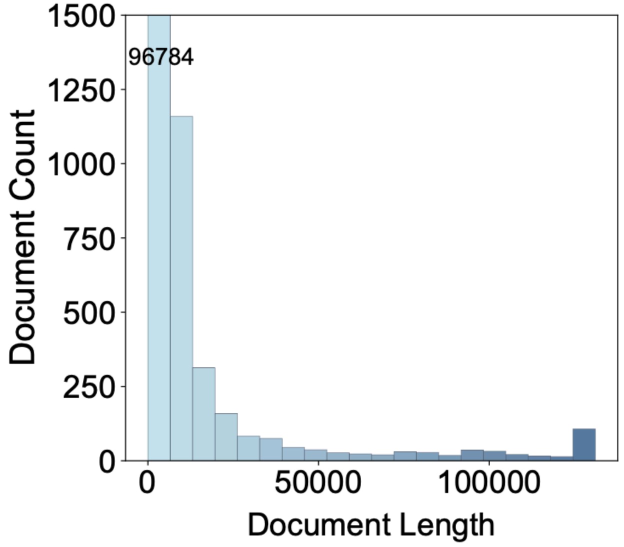
$$\min\left(\max\left(\sum_{i=1}^{N}\left(W_a(x_{ij}\cdot d_i)+W_l(x_{ij}\cdot d_i)\right)\right)\right)$$



Outlier document detection

- Still, you might not have sufficient smaller documents to balance the load of a long document.
- Observe: there aren't that many ultra-long documents.
- Instead of balancing across multiple batches, delay the few long documents.
- Model convergence should not hurt significantly.

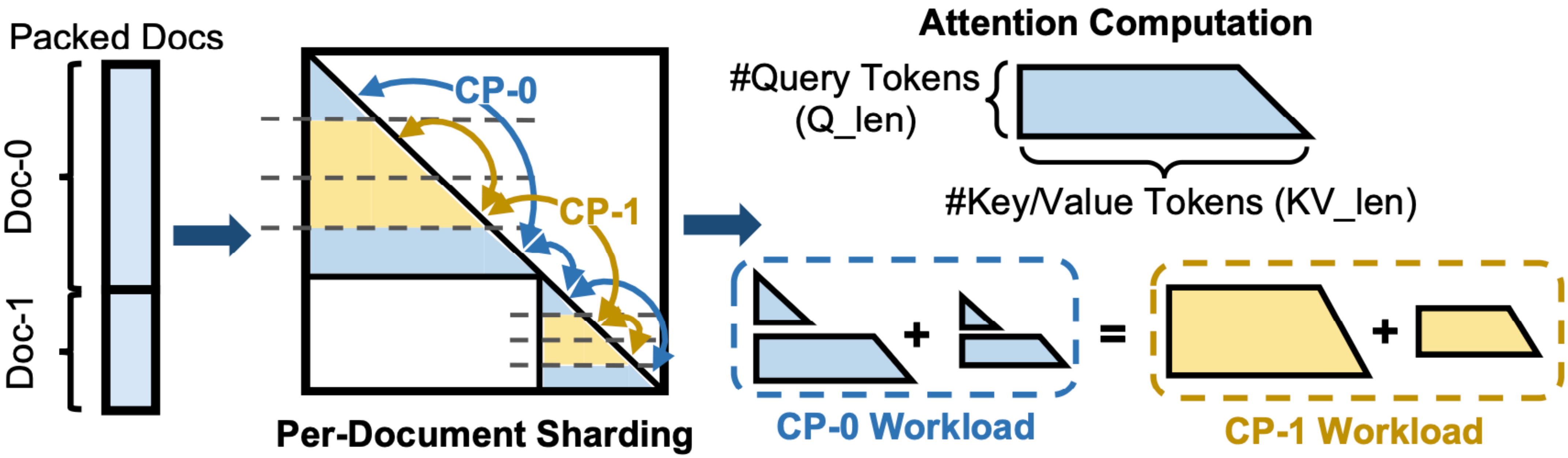




Improved CP sharding

Idea: Apply CP indexing logic to each individual document.

- This should yield a more balanced workload across multiple CP workers.
- They also implement an optimization to avoid padding tokens.

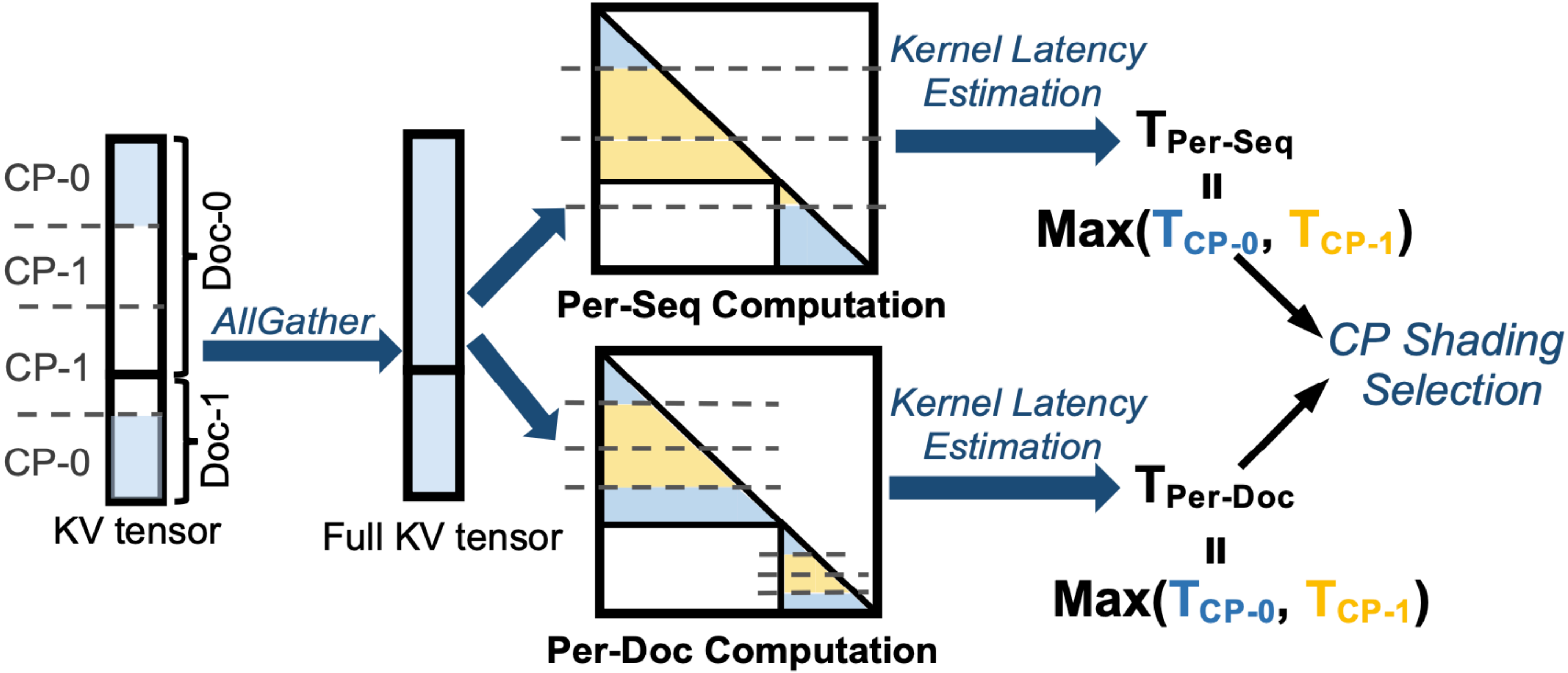


Kernel inefficiencies

- Per-Document sharding achieves better balance, but it does not always guarantee better performance.
- Smaller per-rank attention problems reduce kernel efficiency:
  - Poor tile utilization → padding overhead for short sequences (<128 tokens).
  - Lower effective FLOPs utilization → higher time per token.
  - Reduced KV tile reuse → weaker Hopper TMA multicast benefits

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Kernel Inefficiencies



Experimental Setup

- **Cluster**: 32 nodes, each with 8x NVIDIA H100 SXM 80GB GPUs.
- **Interconnect**: NVLink intra-node, RoCE inter-node.
- **Models**: LLaMA-like 550M, 7B, 30B, 70B; each tested at 64K and 128K context.
- **Training config**: 4D parallelism, global batch size = `PP_size x DP_size`, `bfloat16` precision.
- **Baselines**:
  - `Plain-4D`: default 4D training with per-sequence CP sharding.
  - `Fixed-4D`: fixed-length packing + fixed CP sharding (per-sequence or per-document).

Speedup Breakdown

Which optimization helps us the most?

- `PP-Var-Len` alone $\rightarrow$ 1.28x
- Orthogonal optimizations that combine well.
- Every second counts!

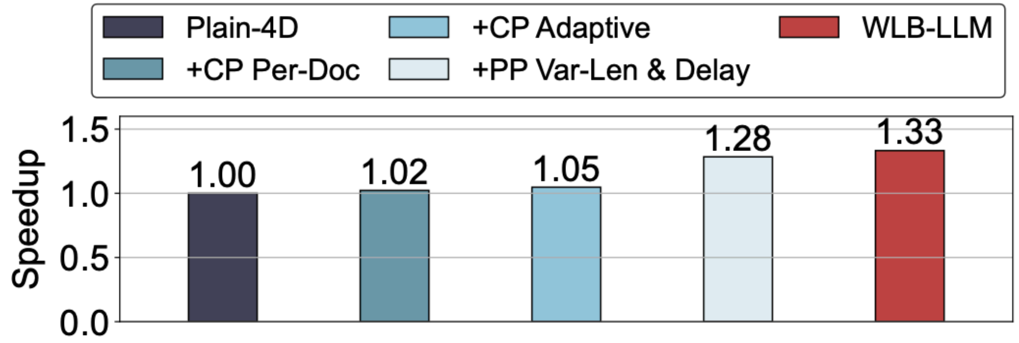


Figure 13: Performance breakdown of *WLB-LLM* on the 7B model with a 128K context window.

Speedup across Model + Context

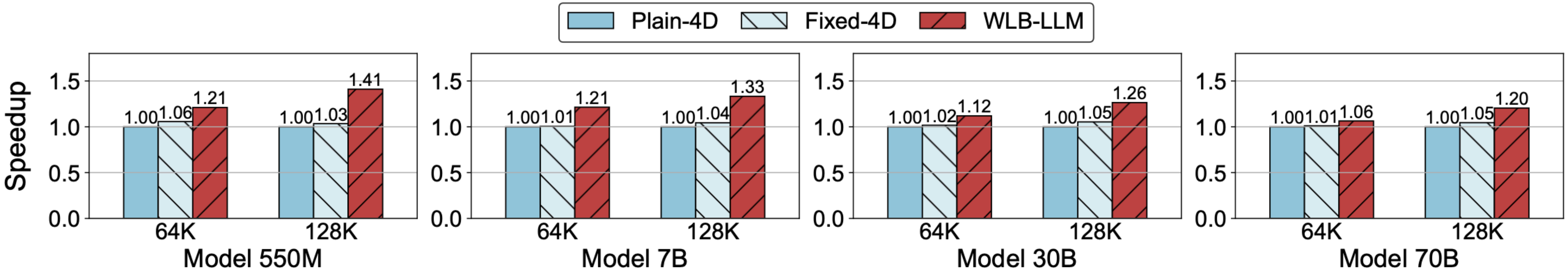


Figure 12: Training performance speedups of *WLB-LLM* and *Fixed-4D* over *Plain-4D* across various configurations.

- WLB-LLM consistently outperforms baseline for all tested configurations.
- Naive attention balancing is insufficient.
- Relative speedup decreases with increased model size.

Other Experiments (Summary)

- **Context sensitivity (Fig. 14)**: Speedup increases with context length (about `1.07x @64K` to `1.40x @160K` on 7B), consistent with worse imbalance at longer contexts.
- **Packing overhead vs balance (Table 2)**: WLB-LLM reaches near-optimal imbalance with low runtime overhead (tens of ms), while solver-based packing can be prohibitively slow.
- **CP sharding ablation (Fig. 15)**: Adaptive sharding consistently outperforms always-per-sequence or always-per-document sharding.
- **Convergence / quality**: Their training-loss curves indicate no clear quality regression from the system optimizations.

Discussion & Critique

Strengths

- Identifies and fixes a bottleneck for training long-context Llama models with 4D parallelism.
- Joint PP+CP optimization gives meaningful real-system gains, especially as context windows grow.
- Engineering is practical: low overhead and no obvious convergence regression in their reported runs.

Weaknesses

- Workload dependence and limited generalization evidence outside their evaluated distribution.
- Strong dependence on a small number of extreme outliers; unclear benefit when length distributions are flatter.
- Heavy use of heuristics (packing + sharding selection) without strong guarantees in worst-case settings.

Related Work

(1) Efficient Long-context Language Model Training by Core Attention Disaggregation (DistCA)

- Split out "core attention" ($\mathrm{softmax}(QK^\top)V$) as a weightless compute service, separate from the rest of the transformer.
- Better than WLB-style baselines at scale: reports ~1.15–1.35× throughput gains over their WLB "ideal" baseline in 4D (with PP), depending on workload.

(2) ByteScale Efficient Scaling of LLM Training with a 2048K Context Length on More Than 12,000 GPUs

- Hybrid Data Parallelism (HDP): unify DP + CP into one dynamic device mesh.
- Length-aware sharding: use the minimum number of devices per sequence.
- Short sequences stay local (skip CP comm), long sequences shard across more GPUs.

(3) Ordering efficiency

- Reduce pipeline bubbles by optimizing scheduling.
- PipeDream, 1F1B, Seq1F1B.

What did you think?

Problem: In long-context 4D training, token count is a weak proxy for compute; attention cost is highly non-uniform.

- **Core idea #1 (PP)**: Reduce PP imbalance via attention-aware micro-batch packing.
- **Core idea #2 (CP)**: Reduce CP imbalance via fine-grained per-document sharding.

Bottom line: WLB-LLM improves training throughput ($\approx 1.23x$) without hurting convergence.

Packing Method		Imbalance Degree	Packing Overhead (ms)
Method	**Config**		
Original Packing	/	1.44	0
Fixed-Len Greedy	#global batch=1	1.41	4
	#global batch=2	1.22	5
	#global batch=4	1.11	5
	#global batch=8	1.08	5
Fixed-Len Solver	#global batch=1	1.40	467
	#global batch=2	1.18	1488
	#global batch=4	1.09	25313
WLB-LLM	#queue=1	1.24	8
	#queue=2	1.05	20
	#queue=3	1.05	23

Table 2: Packing imbalance degree and overhead analysis.