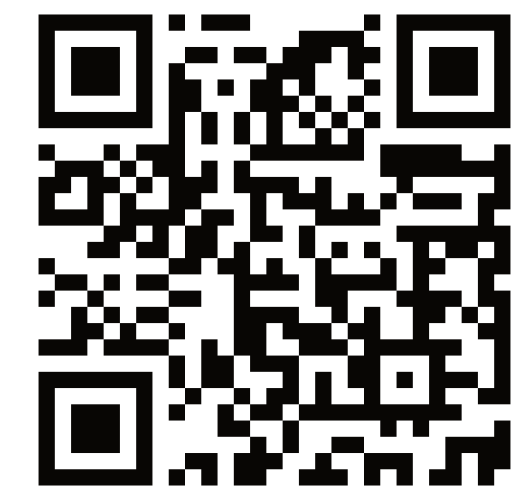




One slow rank makes every rank wait.

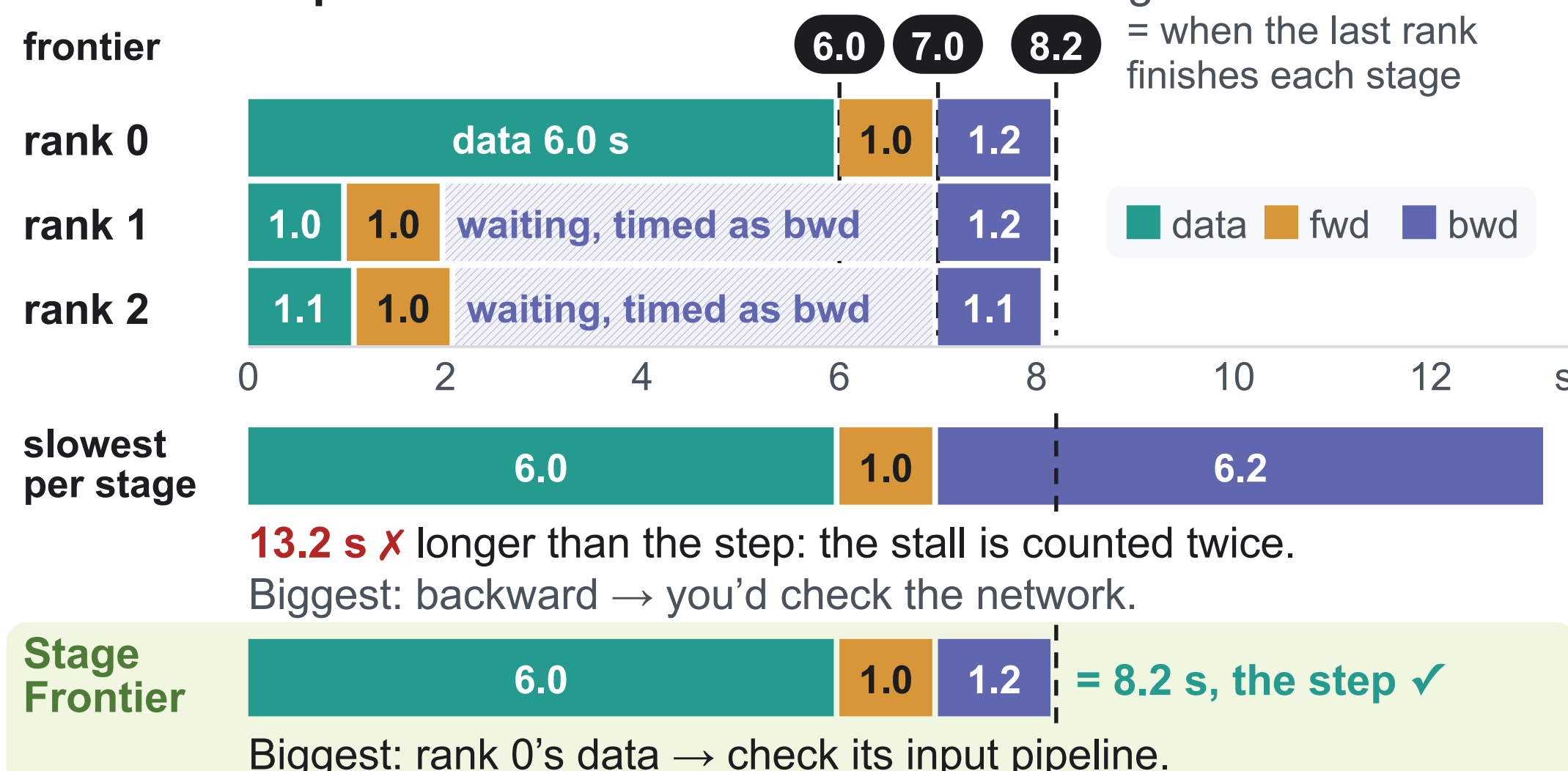
In data-parallel training, ranks sync at every step, so a delay on any one rank, whether in data loading, compute or a callback, can make all the others wait. Their timers may then log that wait in another stage: one rank's slow data load shows up as slow backward on the rest. Finding the real cause took heavy profiler traces.

StageFrontier times every stage for every rank with CPU timers and **works out how much of the step time each stage really takes**, counting each delay once. It names the stage and the rank to check first. With negligible overhead, it can stay on in every job, catching recurring bottlenecks, transient slowdowns and fleet trends.

METHOD

1 Follow the frontier, not the waits

One 8.2 s step on three ranks. Rank 0's data loading is slow.



THE RULE

Charge each stage only for how far it moved the frontier.

$d_{r,s}$: time rank r spends in stage s , on its own CPU clock; $s = 1 \dots S$; $F_0 = 0$

$$P_{r,s} = \sum_{j \leq s} d_{r,j}$$

rank r 's time up to stage s

$$a_s = F_s - F_{s-1}$$

charge to stage s

$$F_s = \max_r P_{r,s}$$

frontier: when the last rank finishes

$$\sum_s a_s = F_s$$

the slowest rank's step time

A stall is charged once, where it first holds everyone back.
Waits on other ranks are not charged again and never need measuring.

Six stages, CPU-timed: data fwd bwd cb opt other

Exact when a stage's work is visible to the CPU within it; GPU work only if the stage waits for it (a sync). Otherwise it shows up in a later stage that waits.

IN SHORT

Low overhead, always on: how much training time each stage really takes.

A few CPU timers per step, no profiler. If one GPU is slow and the others wait, the delay counts once, **where it first holds everyone back**.

2 How is it different from a profiler?

	Profilers	StageFrontier
	our old way: a few steps, on demand	every step, every job
Records	Every op and kernel, on each GPU you choose to trace	A few CPU timers per step, for every GPU
Finds the slow GPU	Trace the waits: align the traces to see which GPU the others wait for	Follow the frontier: each stage's share from each rank's CPU timers
Output	1.2–16 GB of traces for 40 steps on 32 GPUs	0.11 MB for the same 40 steps: cheap enough to leave on

Use both: StageFrontier tells you where and when (which stage, which GPU, since when); a profiler then tells you why (which op or kernel).

APPLICATION

3 What it prints every window

Every step: each rank times its own stages → Every window: rank 0 gathers all ranks' times → Rank 0 computes the frontier and prints:

example: 64 GPUs, one window of 100 steps (illustrative); cb = callbacks

[4200-4299] n=100 step(mean=1.992s p50=1.954s ...)

A crit-path: data=48% bwd=27% fwd=23% cb=1%

B straggler: rank=37 host=... stage=data persist=91% ...

C stages(p50/p95/max): data=0.181s/0.207s/1.623s

fwd=0.451s/0.472s/0.553s bwd=1.294s/1.561s/1.988s ...

How to read it

A **crit-path:** data takes 48% of step time (≈0.96 s per step); backward 27%.

B **straggler:** rank 37 had the longest data time in 91% of steps.

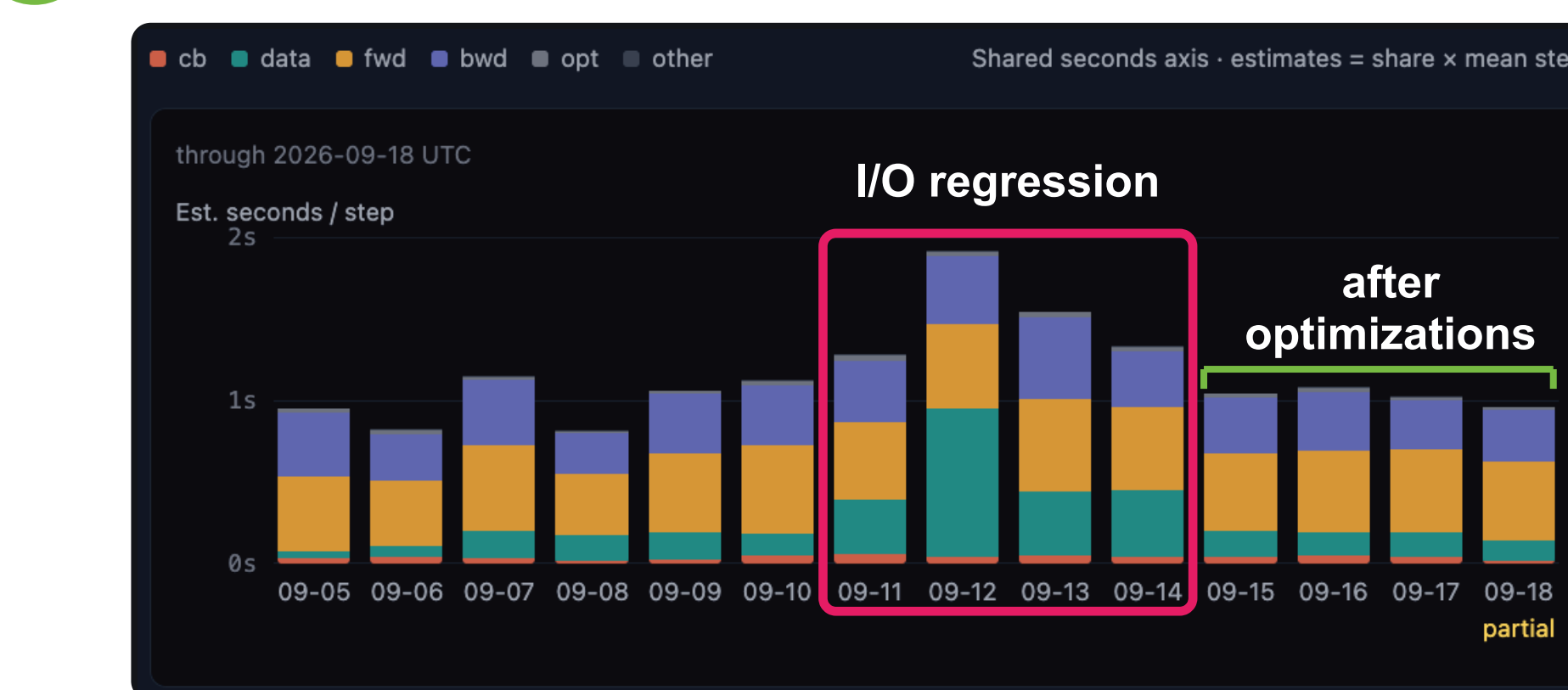
C **stages:** raw times; backward's median 1.29 s is mostly waiting for rank 37.

Next: check rank 37's node and other jobs at that time; profile only if it persists.

4 Per job: recurring bottlenecks

Case	StageFrontier showed	What we found	Outcome
Vision-A	data on top: rare long stalls from varying ranks	loader workers' handoff to the main process stalls at times; fix: prefetch batches	230 → 340 samples/s
Vision-B	callbacks 24% though usually 2 ms (spikes up to 7 s); data 27%	image logs every 10 steps, pin_memory off, too many loader workers	63 → 85 samples/s
Mixed-A	data 90%, yet batches were ready: no queue or worker wait	post-processing in the main process: .cpu() syncs, Python loops, file reads	Confirmed by turning it off
QAT-B	forward 67%, every step alike: steady, no spikes	fake-quant ops: .item() syncs and tens of thousands of tiny kernels	Confirmed by profiler

5 Transient slowdowns and fleet trends



All jobs on one cluster, per day. Sep 11–14: a new feature added many reads from shared Lustre storage; on Sep 12, data took ~0.9 s (usually ~0.15 s) of a ~1.9 s step.

RELATED POSTER how the video data loading was optimized
From MP4 to Tensor: Optimizing Multi-Camera Video Data Loading for PyTorch Training · Kyle Huang and Mihai Alden

CONCLUSION

One job
Which stage and which GPU to check, for slowdowns that recur or come and go.

Across jobs
Other jobs slowed too? Slow data suggests storage; slow backward, the network. Track regressions per project or cohort.

Fleet
Whether an optimization helped across all jobs, and which stage costs the most GPU-hours.

In production
On by default in almost all Alpamayo training jobs since March 2026.

PAPER
arXiv:2606.06751
CONTACT
byoon@nvidia.com