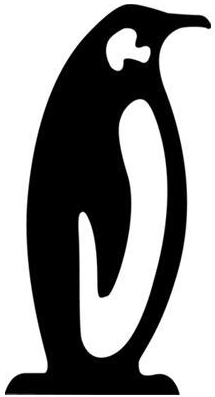
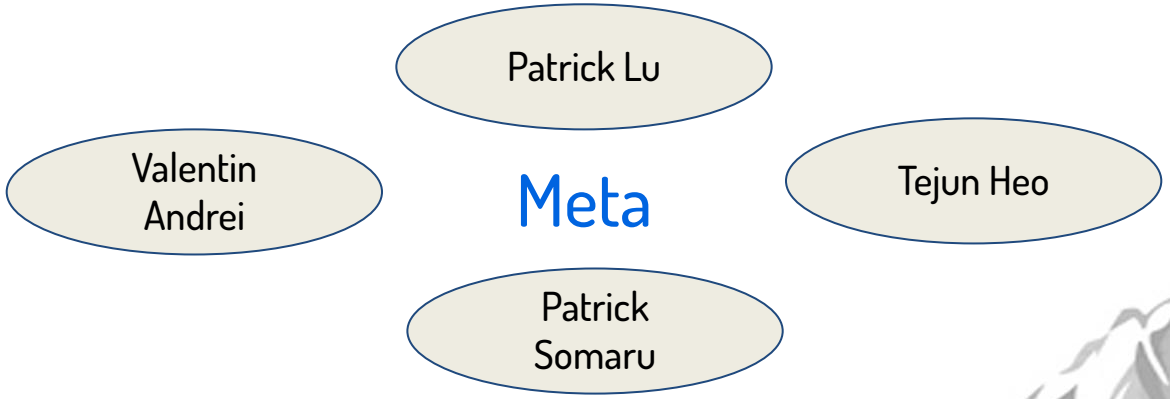


東京 2025 LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025

Accelerating distributed AI training with sched_ext



東京 2025
LINUX PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

Acknowledgement (Meta folks)

- Emil Tsalapatis
- Sam Crossley
- David Vernet
- Dan Schatzberg
- Jamie Cahill
- Maksym Kutsevol
- Andrii Nakryiko
- Joseph Bacik
- Jake Hillon
- Fuat Geleri

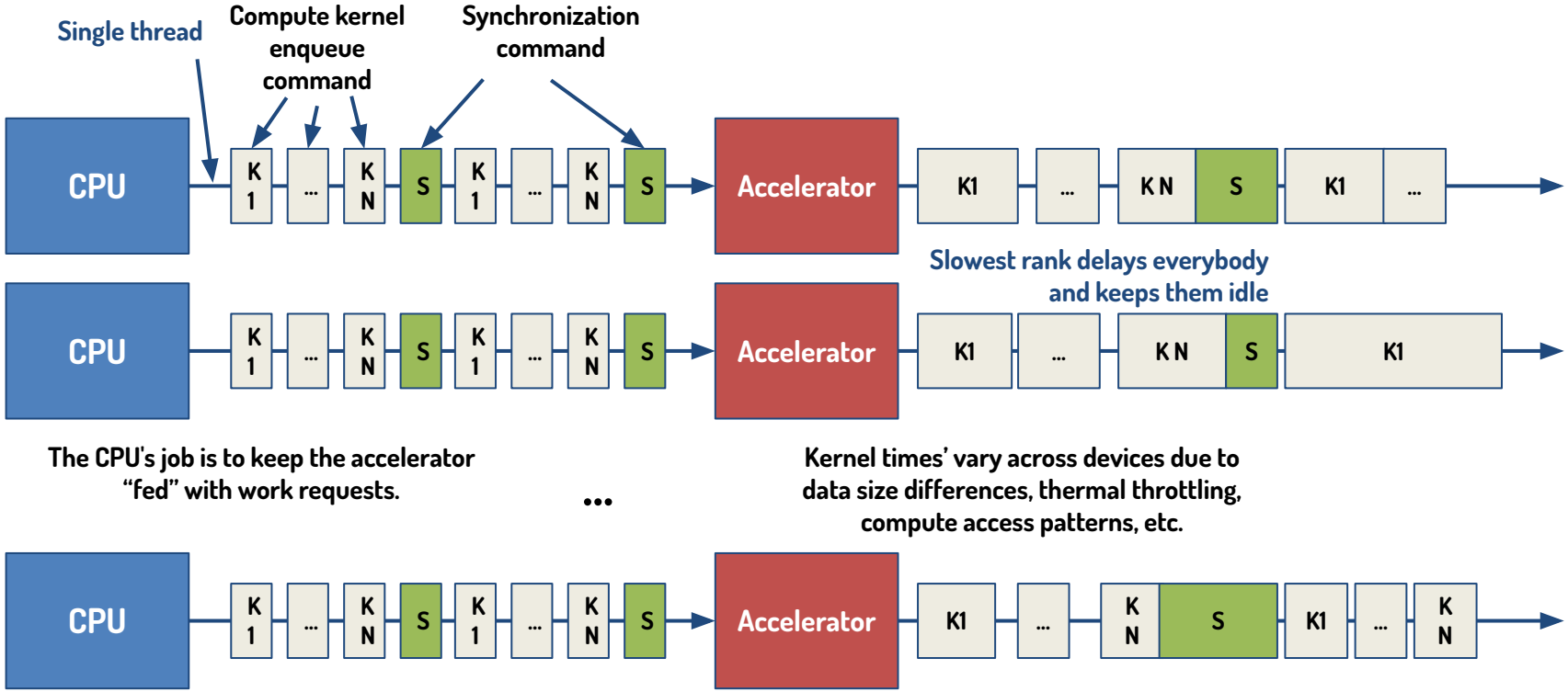
東京 2025
LINUX
PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

Content

- Relevant information about distributed AI
- Why we invested in using sched_ext to accelerate distributed AI
- How we used sched_ext in production:
  - Deployment strategy
  - Encountered issues
  - Observability
  - Improvements
- How can we do better

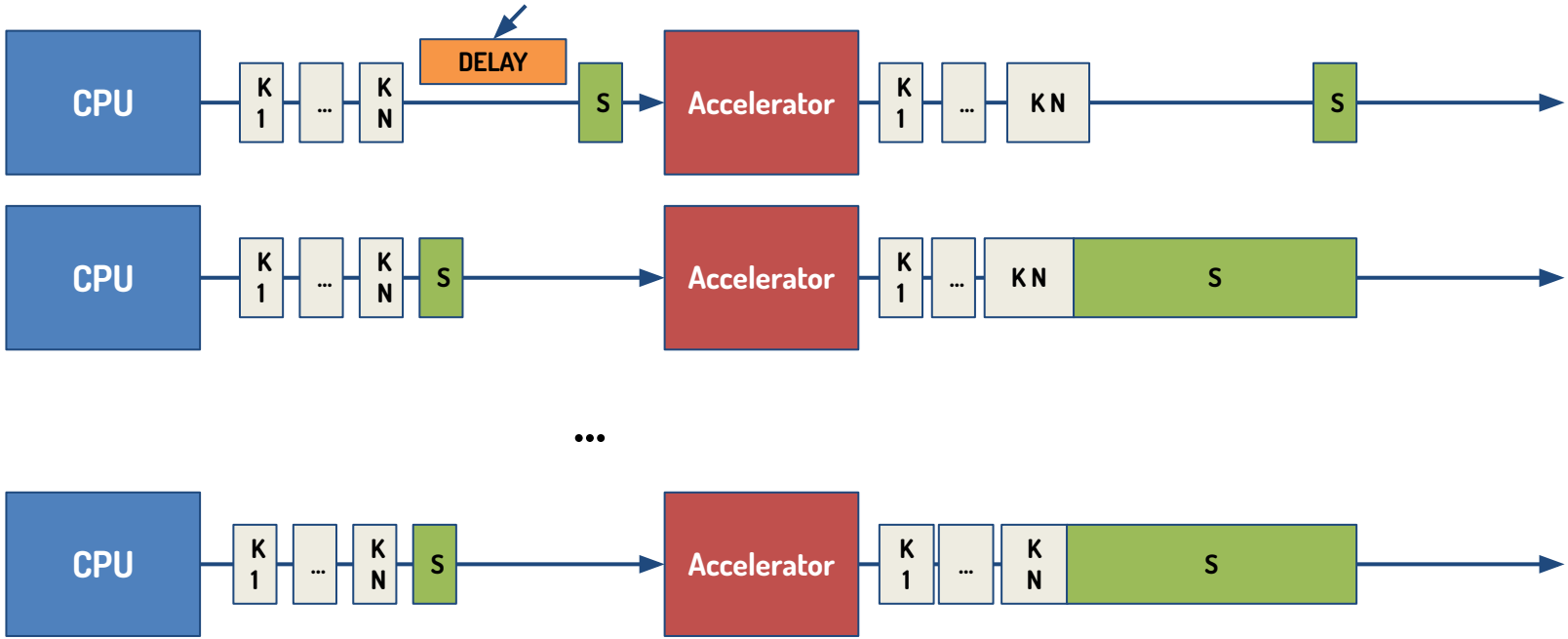
The Theory

Distributed AI training (1) - Critical path, accelerators, synchronization

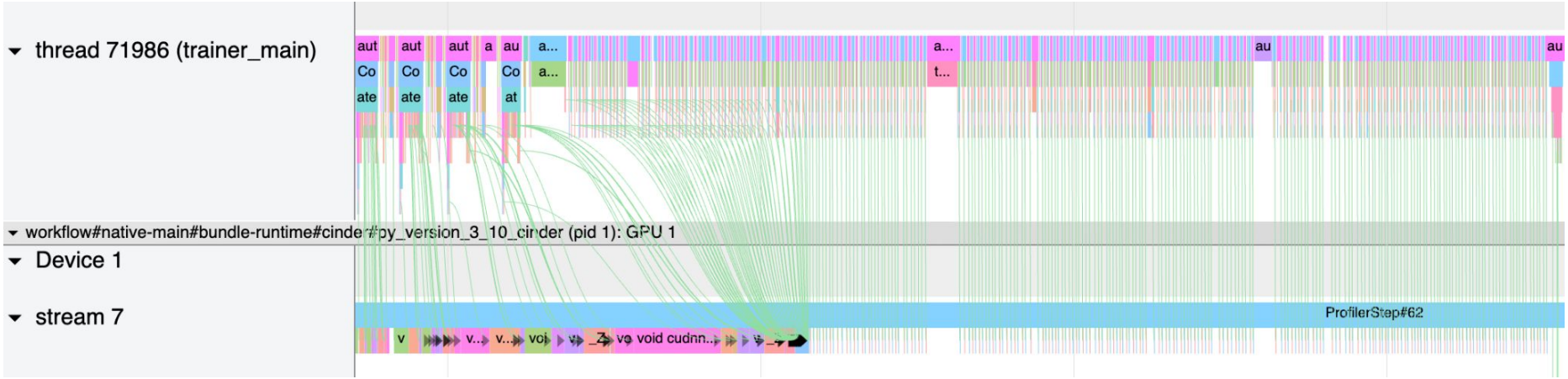


Distributed AI training (2) - Critical path, delays

Whenever a delay (e.g. a preemption) occurs on the critical path, the accelerator become idle or wait for the straggler to complete the synchronization command.



Distributed AI training (3) - Real trace



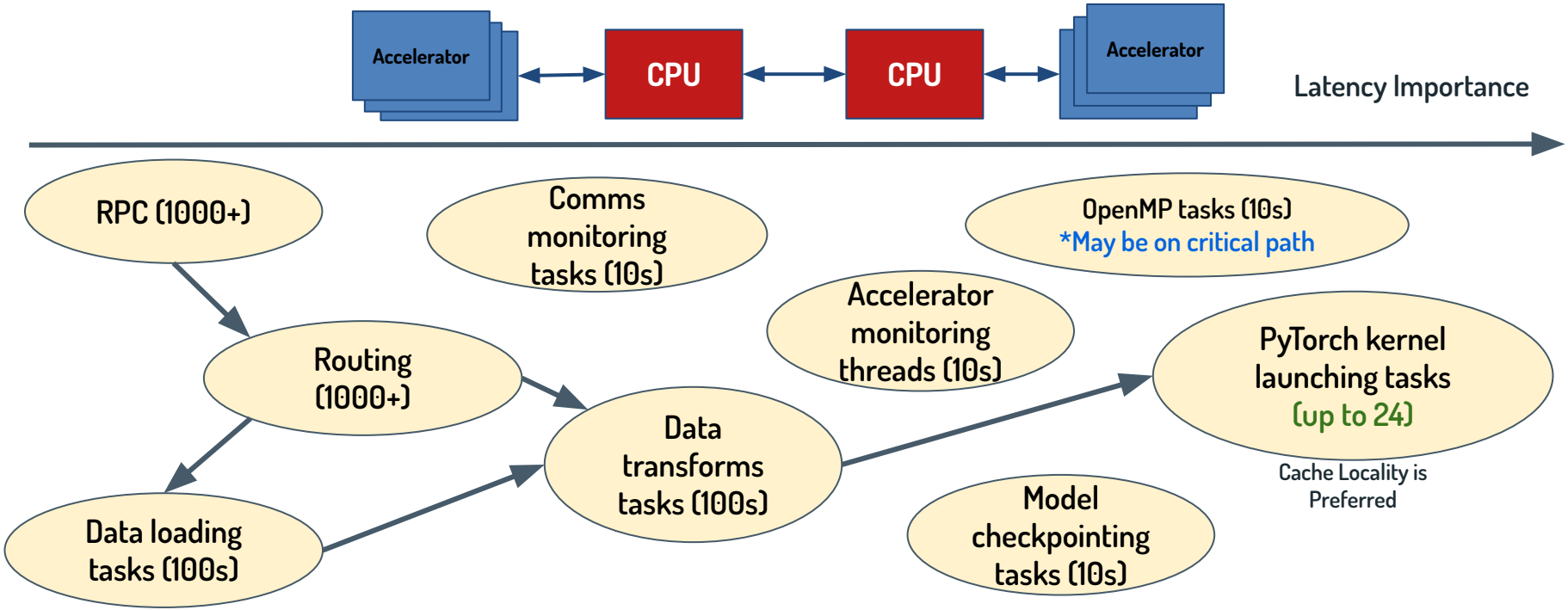
Not so latency bound phase. The GPU has fallen behind because high amount of work required.

Latency bound phase. The GPU runs CUDA kernels very fast because it probably executes upper regions of the model. The GPU is mostly starved waiting for the CPU. If trainer main is preempted due to any reason, the workflow loses QPS - due to synchronization.

Distributed AI training (4) - Example

This chart motivated most of the work presented in the following slides.

Distributed AI training (5) - Thread pools



Challenge → Prioritize latency critical threads without starving them due to data dependencies.

Challenge → Confine threads when thread pools are unreasonably large

Challenge → Dealing with NUMA balancing.

With sched_ext we aim to write a scheduler that:

- **Allow us to use more CPU cycles for useful work (e.g. reduce remote data processing)**
- Truly prioritizes the latency critical tasks of the training application
- Has a mechanism of ranking classes of tasks by importance
- Protect the latency critical tasks from CPU frequency throttling
- Provides preferential access to memory bandwidth for latency critical tasks (when possible)
- Has a short development and deployment cycle
- Is robust enough to be deployed across a fleet with hundreds of corner cases

Scheduler design idea

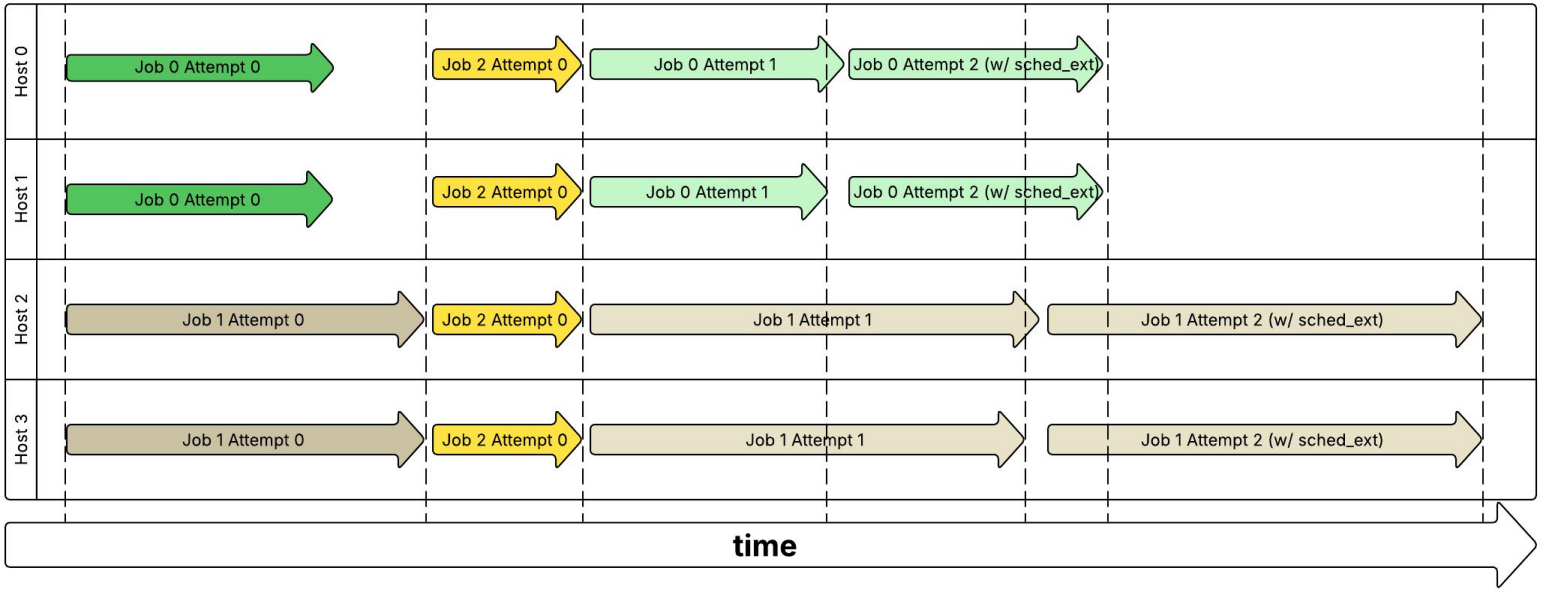
- scx_layered → Bucket the tasks in the system into layers with different properties and scheduling policies:

Latency critical tasks + comms monitoring and signaling	Reduce preemptions, run on cores with higher frequency, preserve cache locality, etc.
Data pipeline tasks	Allow variable CPU count usage, can run on cores at lower frequency, limit memory bandwidth usage, etc.
Supporting frameworks tasks	Confine into a fixed number of CPUs but sufficient enough to avoid stalls
The rest	Whatever is left

The Practice
(Fleet Level Deployment)

What can go wrong?

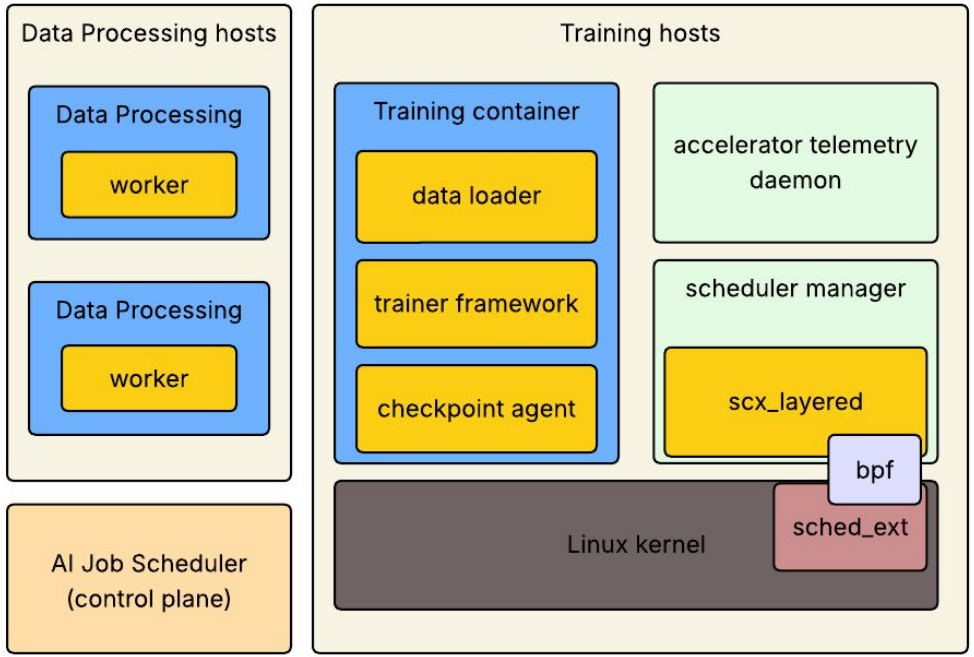
Concept of an AI Training job attempt



We can enable sched_ext on pre-defined attempt(s), and compare scheduler performance between job attempts. This provides an apples to apples comparison

Host level components used in deployment

- AI Job Scheduler → coordinate the lifecycle of training jobs
- Data Processing → transform raw input data into training tensor
- Accelerator telemetry → publish GPU & host utilizations
- Scheduler manager → manage the lifecycle of scx_layered, a bpf based scheduler
- Trainer container → contains the distributed training application



Our journey

- Opt-in / Opt-out mechanism
- Fast enabling / disabling
- Deploy per job, model, training paradigm, product group, team, etc.

Experiments

- Does it make sense to use sched_ext
- Identify as many corner cases as possible
- Which training application can mostly benefit?

Build Fine Grained Deployment

Build Observability

- Monitor for failures and alerts
- Monitor performance
- Quantify the infrastructure cost savings

Target Use Cases / Small Scale Deployments

- Highly heterogeneous training for small-medium content understanding models
- On-box data pipelines and reduce the need for remote data pipelines

Fleet Level Slow Roll-Out

- Slowly enable sched_ext, observe behavior and continue until full deployment

Early experiments findings

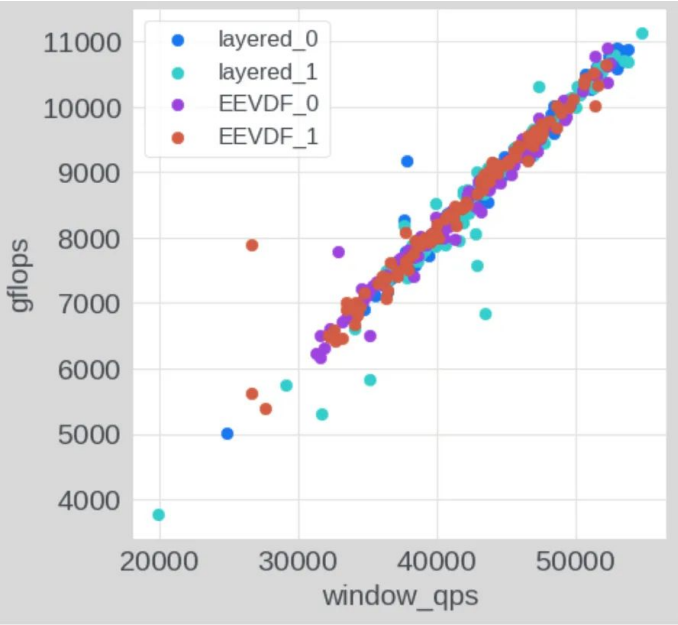
Fundamental

- **How to identify the tasks that are latency critical?**
- How to avoid slowing down the data pipeline while protecting the more important tasks?
- The CPU frequency drops with CPU utilization
- With more load on the system LLC misses become more expensive due to increased memory bandwidth pressure
- **NUMA balancing is difficult**
- **There are no tools for non-experts that can help analyzing a schedulers' efficiency**

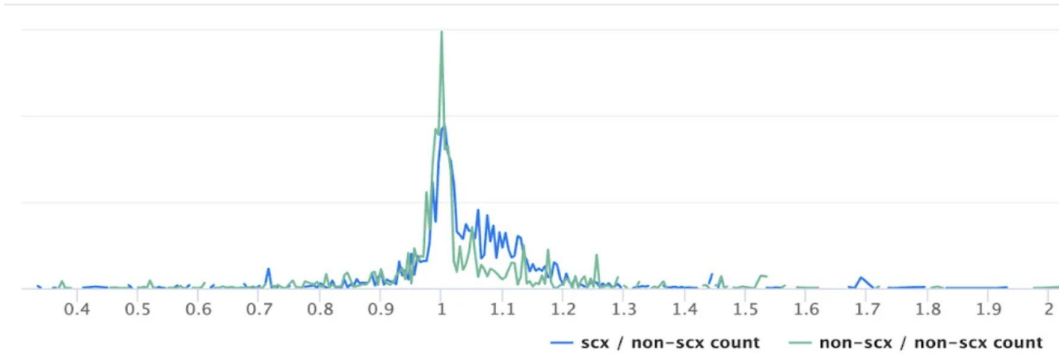
Corner Cases

- Our "latency critical" tasks yield() too often
- Tons of task dependencies (e.g. Python GIL, data transforms → data loaders)
- Thread pools are copy-pasted across applications with default settings creating hundreds of thousands of tasks
- **Tasks get pinned to specific cores across the system**
- Users launch too many tasks for supporting services like data pre-processing and loading, model dynamic compilation, etc.
- Too many tasks that schedule compute kernels are being created
- **IRQs are preempting our important tasks**

Finding a proxy to measure fleet-wide performance

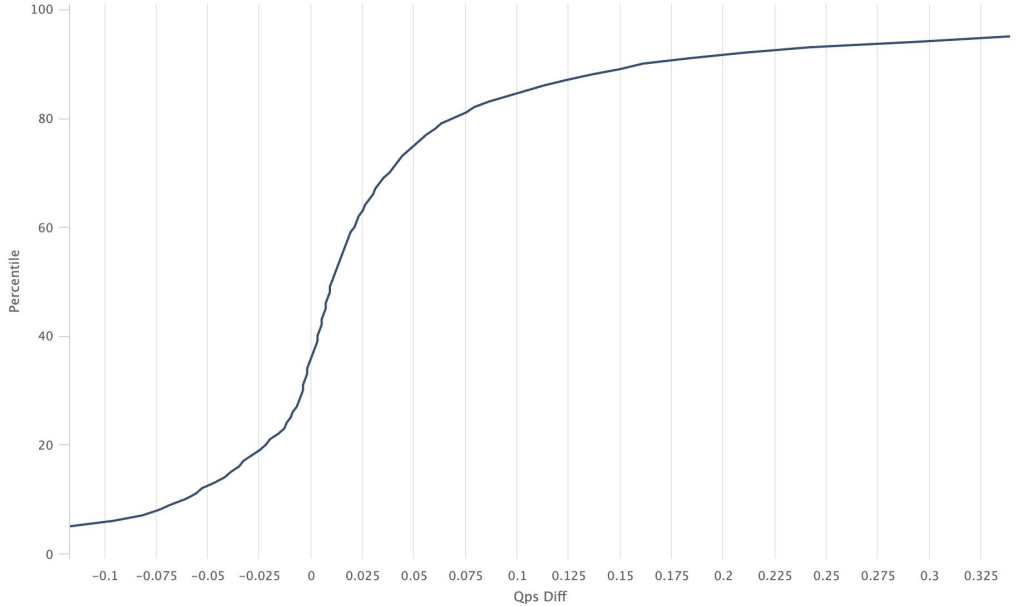


Credit: Sam Crossley (Meta)



- There is no universal application performance metrics.
- Hardware counters (like SM Utilization & tensor core active % correlated well with application performance.)
- Building A/A vs. A/B aggregation can help us reason fleet-wide performance

Fleet-wide training throughput improvement



- Considerably more jobs show throughput improvement
- No signs of pathological regressions
- We saved about 4% of accelerator capacity on a fleet using tens of thousands of accelerators
- We are aiming to save ~30% of remote data processing capacity

The Pain
(How can we do better ?)

Identifying the tasks on the critical path

Issues: Which tasks are latency critical? what if they are created a while after the job starts? what if they change during the execution? What can we use to identify them?

Solutions (Nvidia Specific):

- Kprobe'ing mmap, poll and open in Nvidia drivers to obtain real-time PID to GPU mappings
- Querying NVML to obtain PID To GPU Mappings
- Obtaining GPU to NUMA Node Mappings from NVML
- Heuristics (CommPrefix, Cgroup Name, Group Leader status, etc.)

Potential Improvements:

- A standardized approach for accelerator vendors to report the tasks that schedule work on accelerators. The system can use this information to prioritize those tasks, as it's highly likely that they are latency critical.
- More tracepoints in the accelerator stack that allow us to identify context creation, async copies, kernel launches, etc.

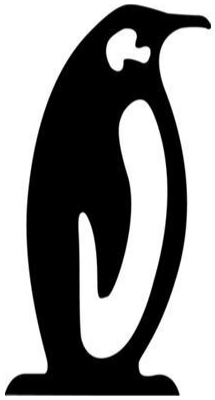
Tasks management

Issues : Latency critical threads do frequent yields() **Solution:** We ignore a percentage of yields() coming from tasks in the high priority layer
Issue: Latency critical tasks maintain persistent metadata structures about tensors throughout the application. We need locality. **Solution:** Implement a “prev over idle policy” where we pick the last CPU where the task previously landed
Issue: Applications are very sensitive to bad placement of tasks across NUMA nodes **Solution:** Implement balancing strategies in scx_layered and query Nvidia libraries for current NUMA placement
Issue: Applications are using custom core pinning and increase the risk of stalls if other tasks use the “dedicated” CPUs **Solution:** Not much besides removing the custom affinities :(

Tasks management (Potential Improvements)

Discussion: We would like be able to move allocated memory with tasks when we move tasks to be local to the GPUs they were using, ideally from bpf, and at least without having to modify workload code.

Discussion: We would like to be able to confine memory allocated to tasks to that local to their current GPU, to help further reduce cross NUMA node traffic.



東京 2025
LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025