

Bridging CPU and GPU I/O with a Device-Resident File System

Qi Chen, Guanyi Chen, Yujie Ren*, Jian Zhang

CUHK(ShenZhen)* & HKUST(GuangZhou)

1 Background & Motivation

2 Design

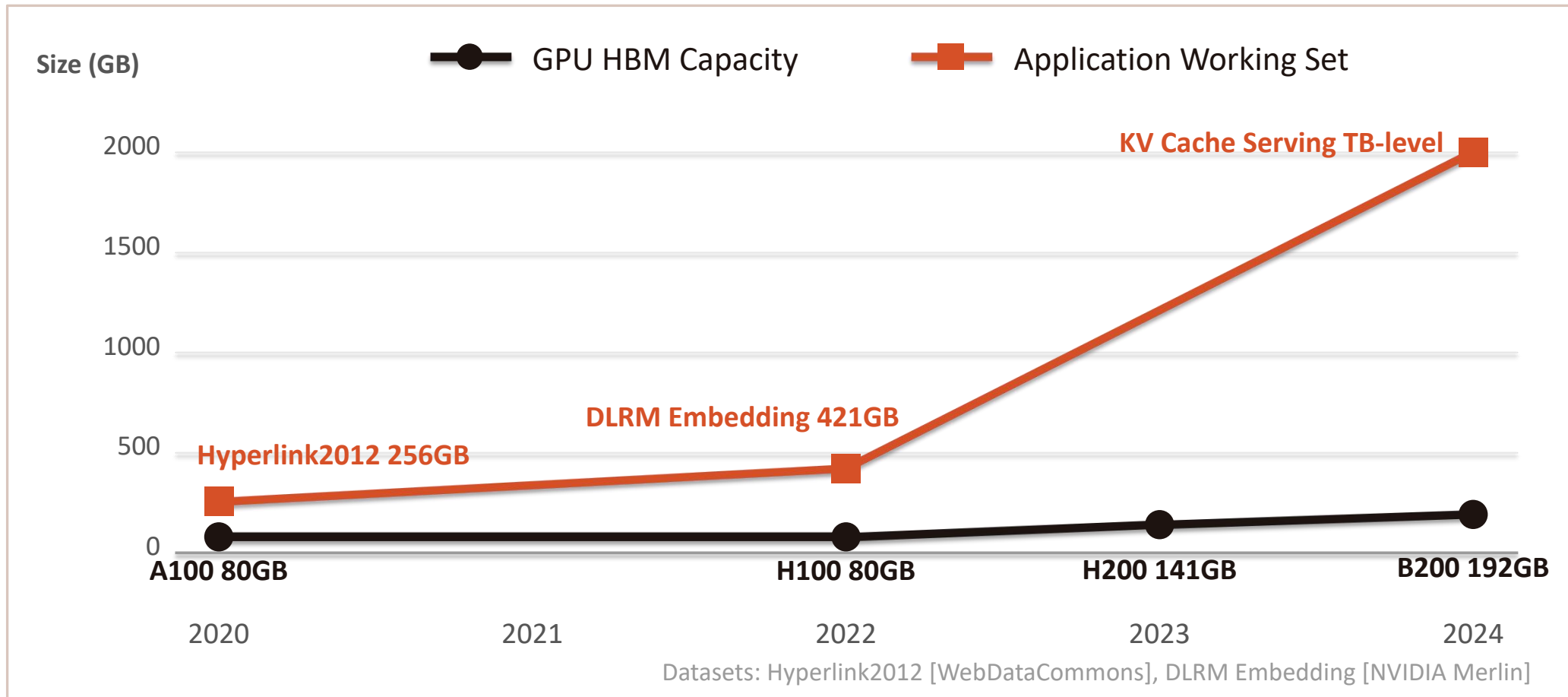
3 Evaluation

4 Open Challenges

5 Conclusion

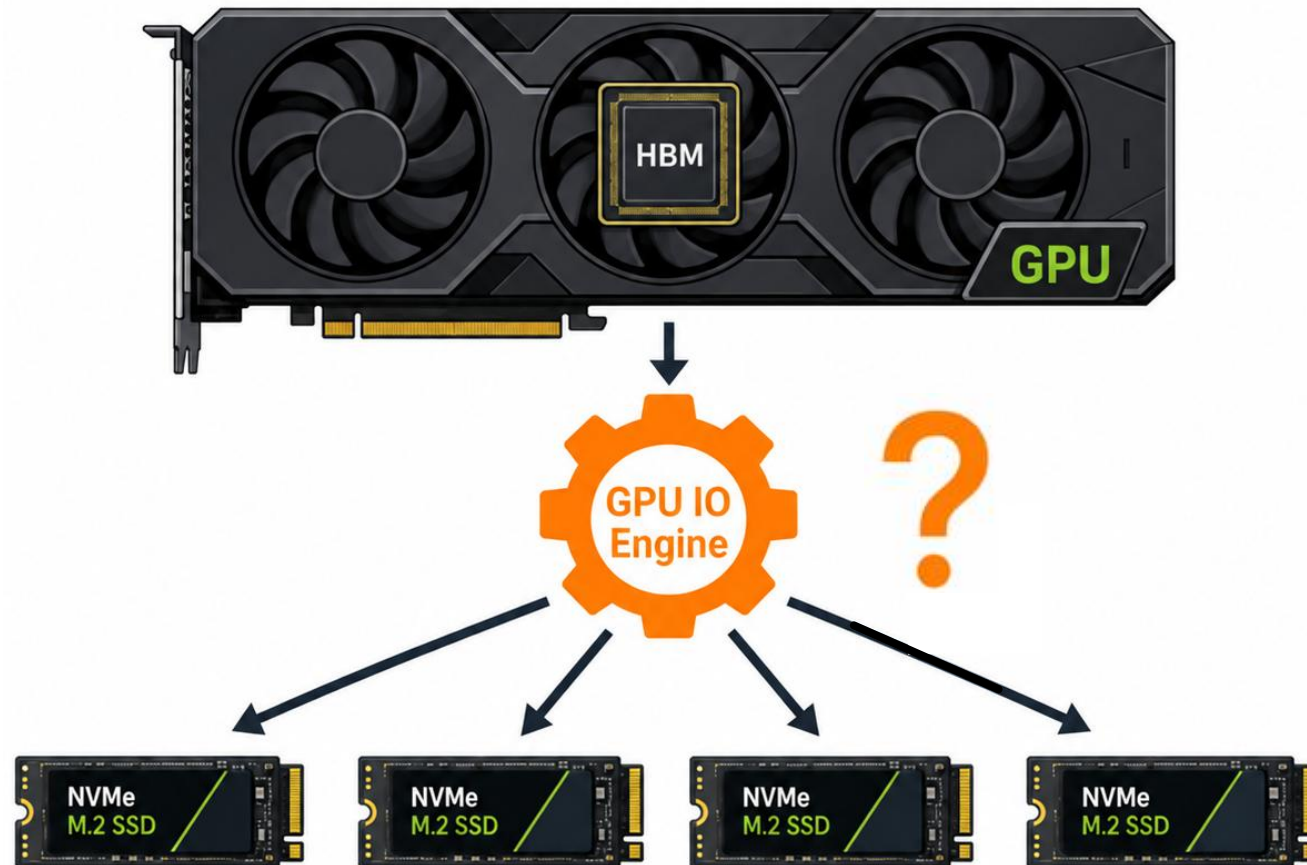
GPU Memory is not enough

GPU memory capacity grows **far slower** than application data size



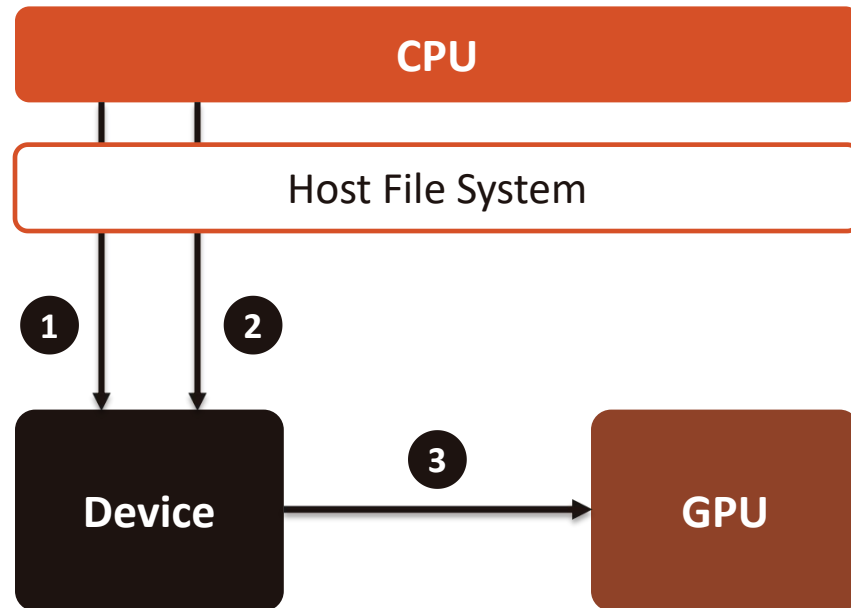
Extending GPU Memory with SSDs

NVMe SSDs provide **TB-scale** storage to feed data-hungry GPU applications

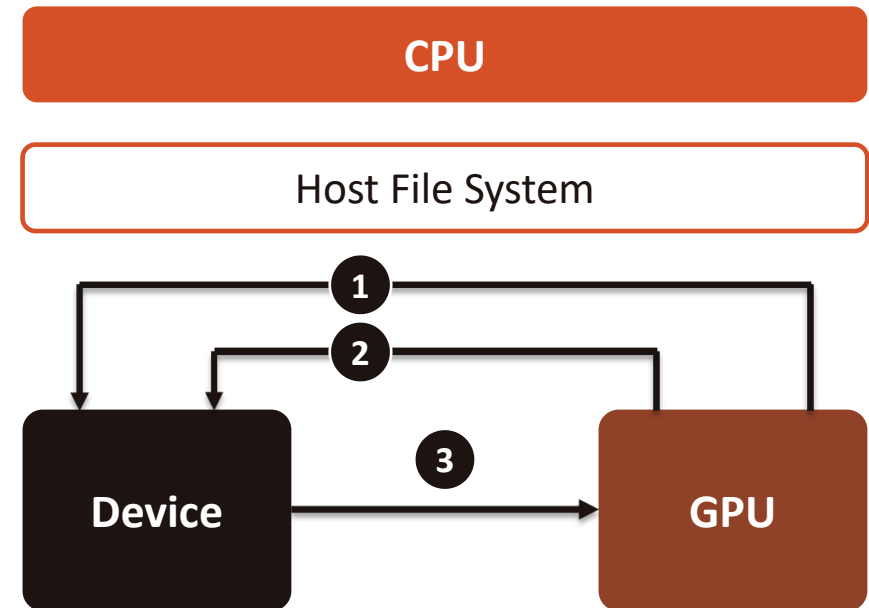


Current GPU IO Engines

CPU-initiated



GPU-initiated

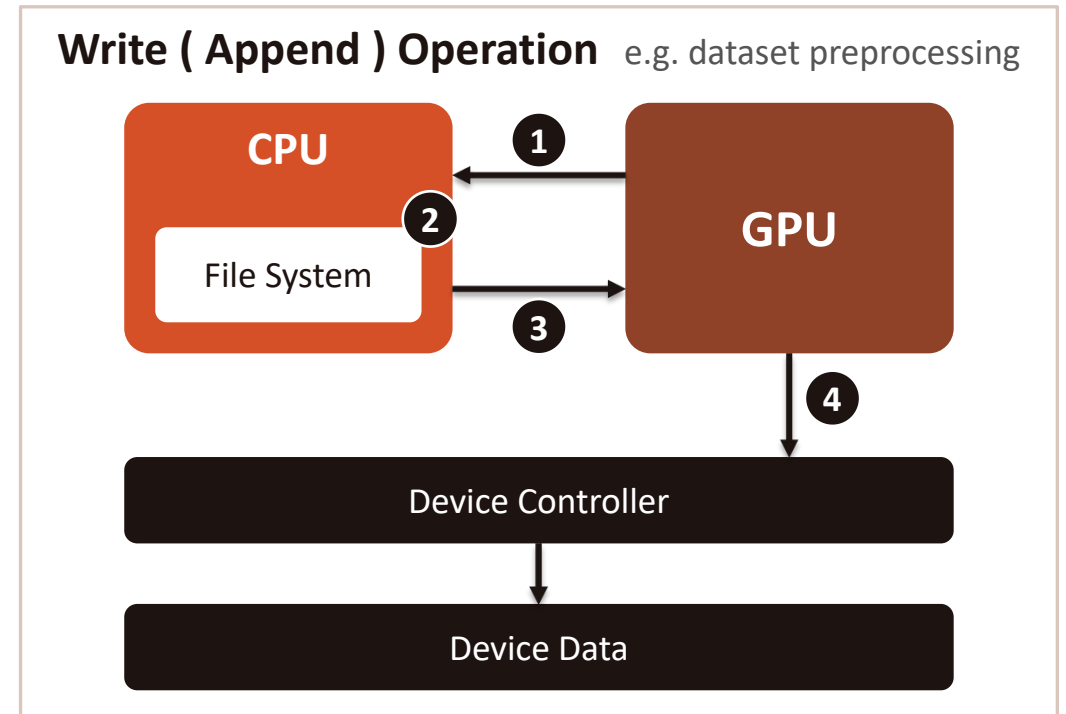
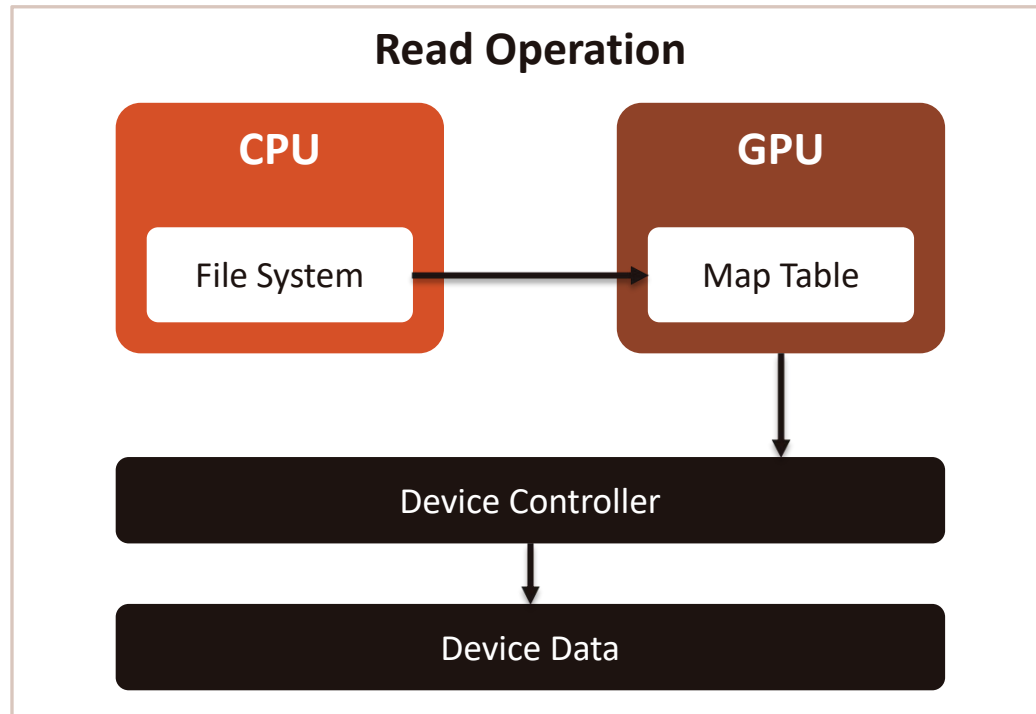


1 Submit I/O Request 2 Polling I/O

3 Transfer data

Current GPU File Systems

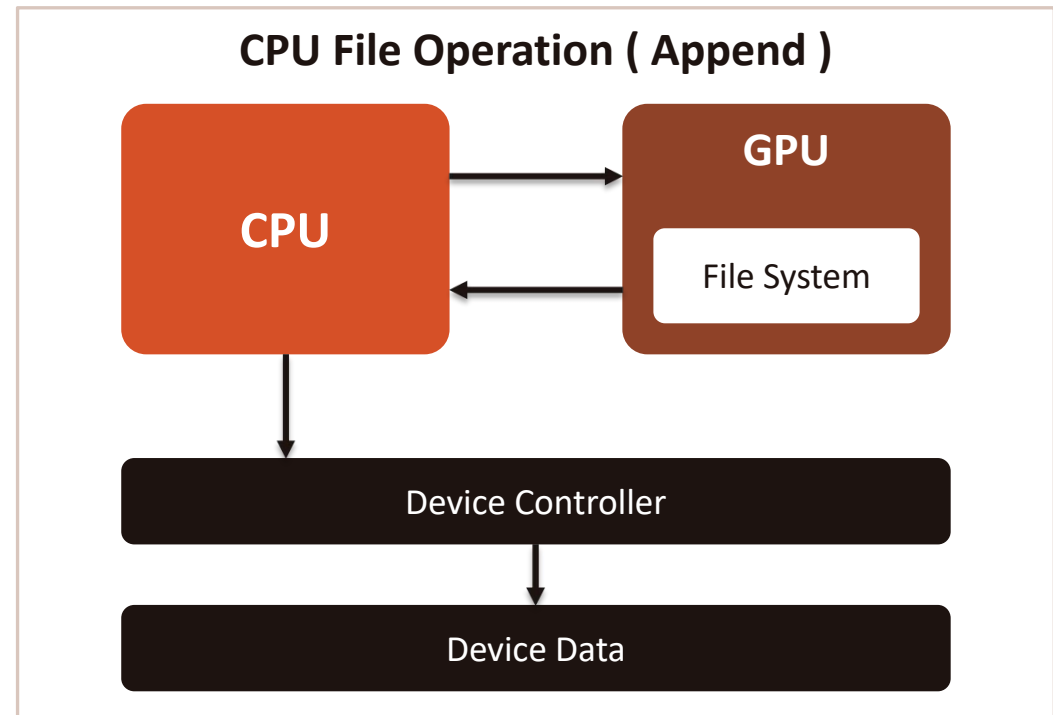
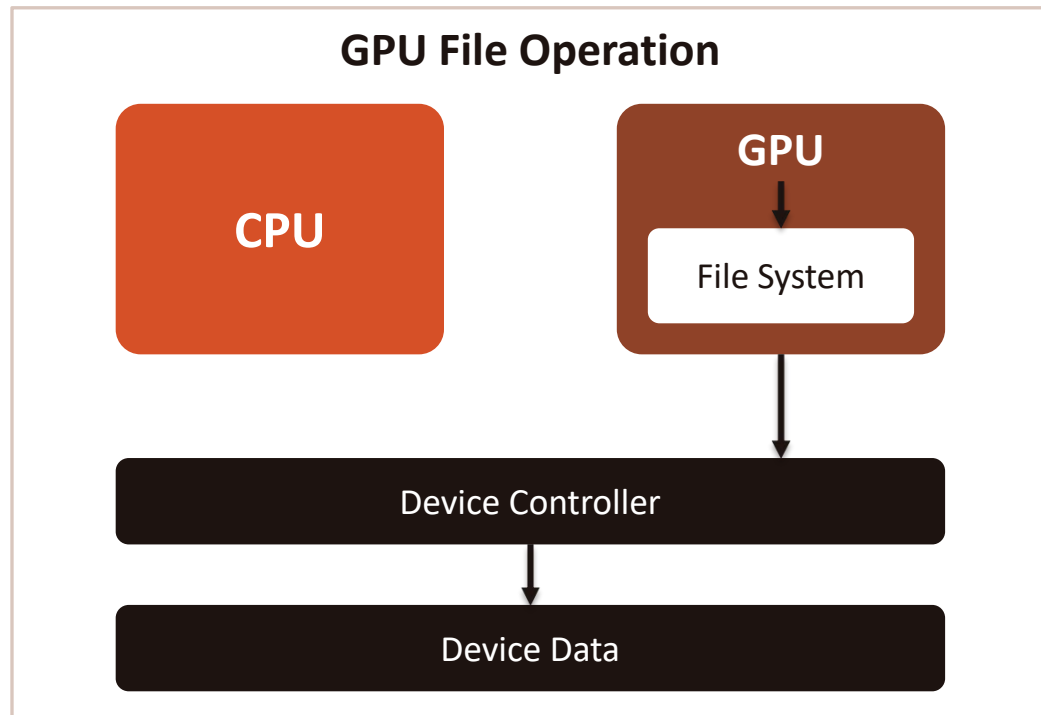
- **CPU-Resident FS** place file system on the CPU side
 - GPU have a partial ability of file system
 - Problem: GeminiFS assume read-only GPU workloads



CPU-Resident (e.g. GeminiFS [FAST' 25])

Current GPU File Systems

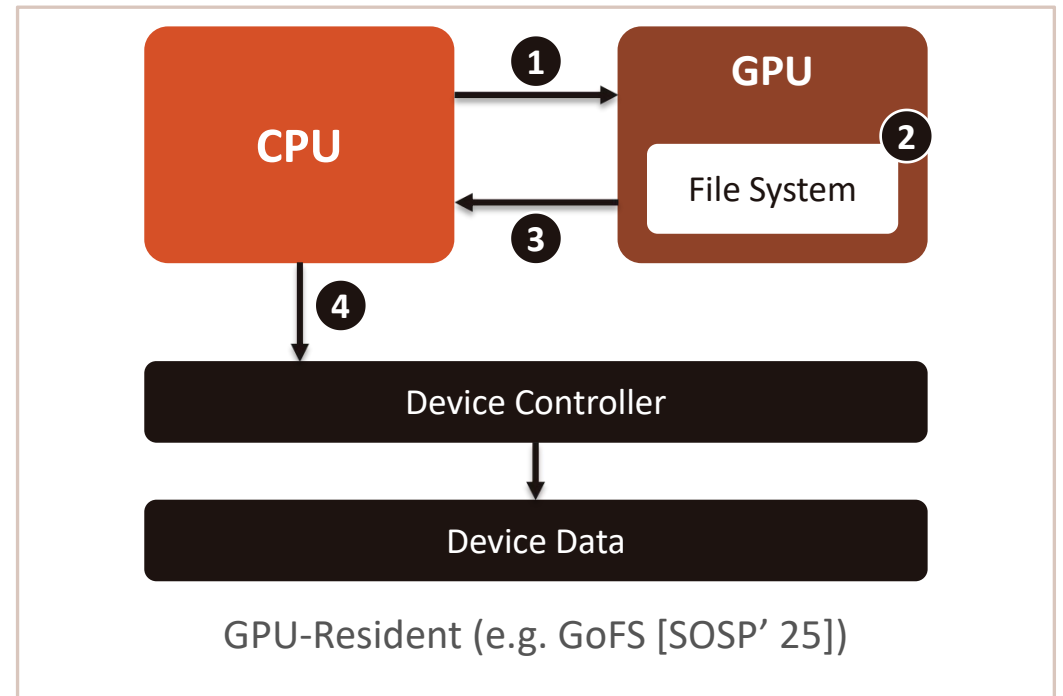
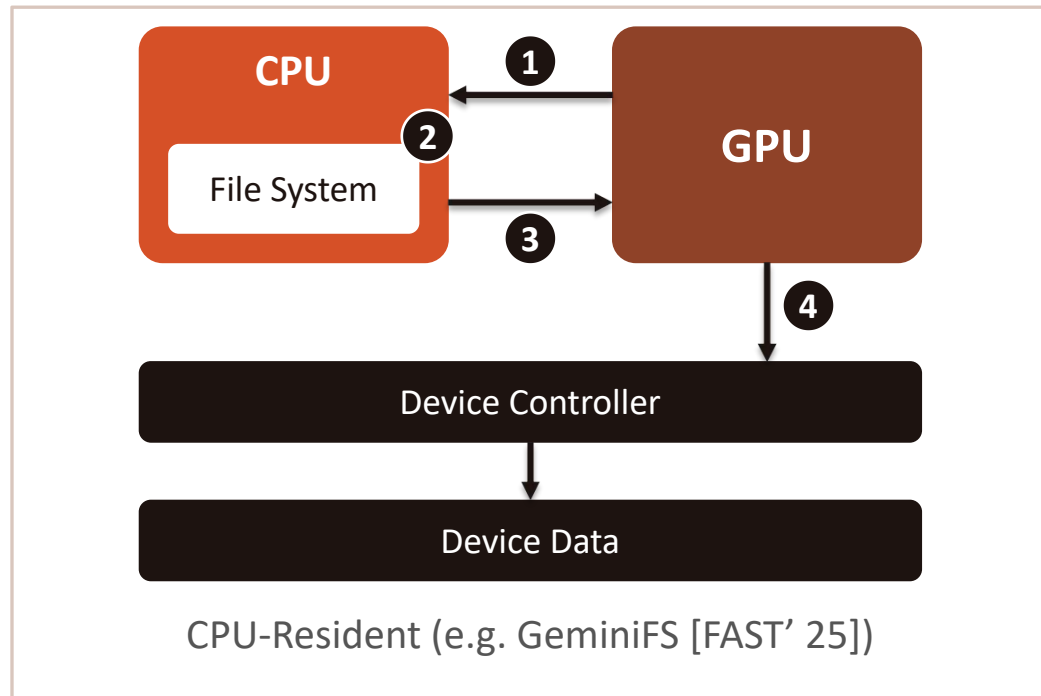
- **GPU-Resident FS** place file system on the GPU side
 - GPU preserve full file system functionality
 - Problem: GoFS syncs CPU side metadata-modifying requests with GPU File System



GPU-Resident (e.g. GoFS [SOSP' 25])

Motivation Summary

- **The metadata synchronization is necessary whether file system is on CPU or GPU.**
 - Distributed file system are designed to scale metadata management, not to eliminate metadata synchronization
 - Device file system can eliminate metadata synchronization, but fail to handle massive GPU IO request because of limited CPU core parallelism



Outline

1 Background & Motivation

2 Design

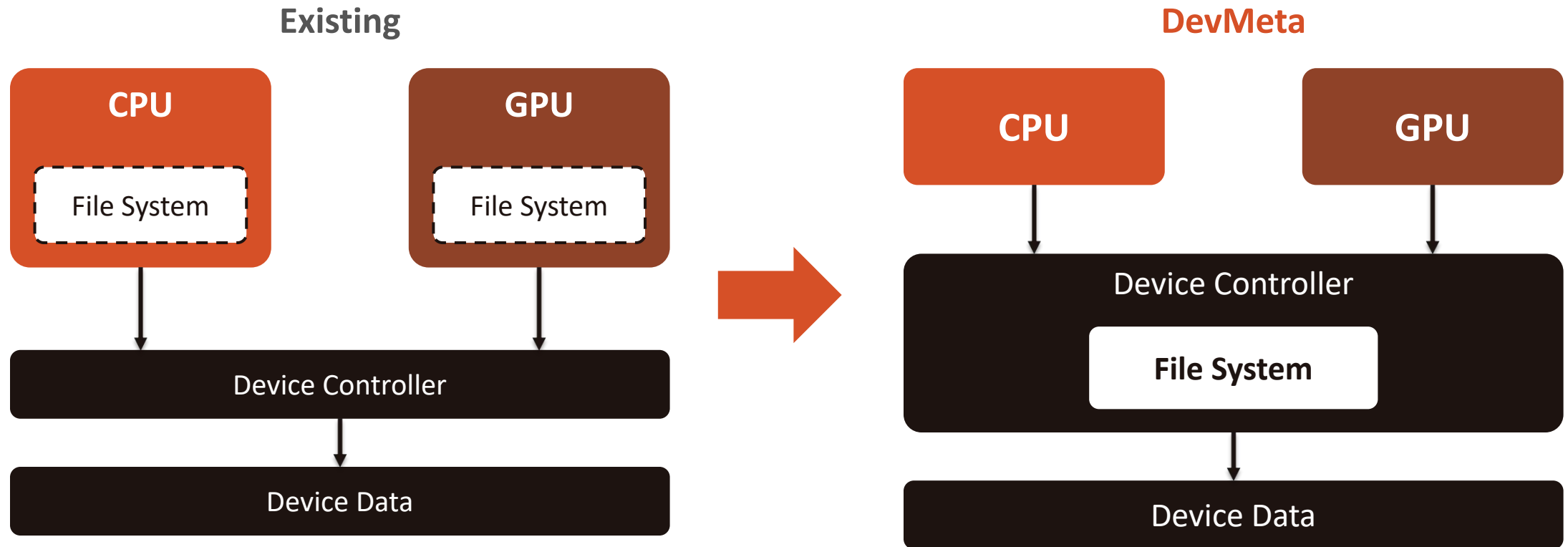
3 Evaluation

4 Open Challenges

5 Conclusion

Our Solution: DevMeta

- A new device side file system that remove metadata synchronization
 - GPU massive IO are handled in time



DevMeta Components

DevMeta-CPU/GPUIOLib

Support POSIX API · Provide CPU/GPU-Initiated IO function

DevMeta-Profiler

Profile and manage hot metadata

GPU Metadata Cache

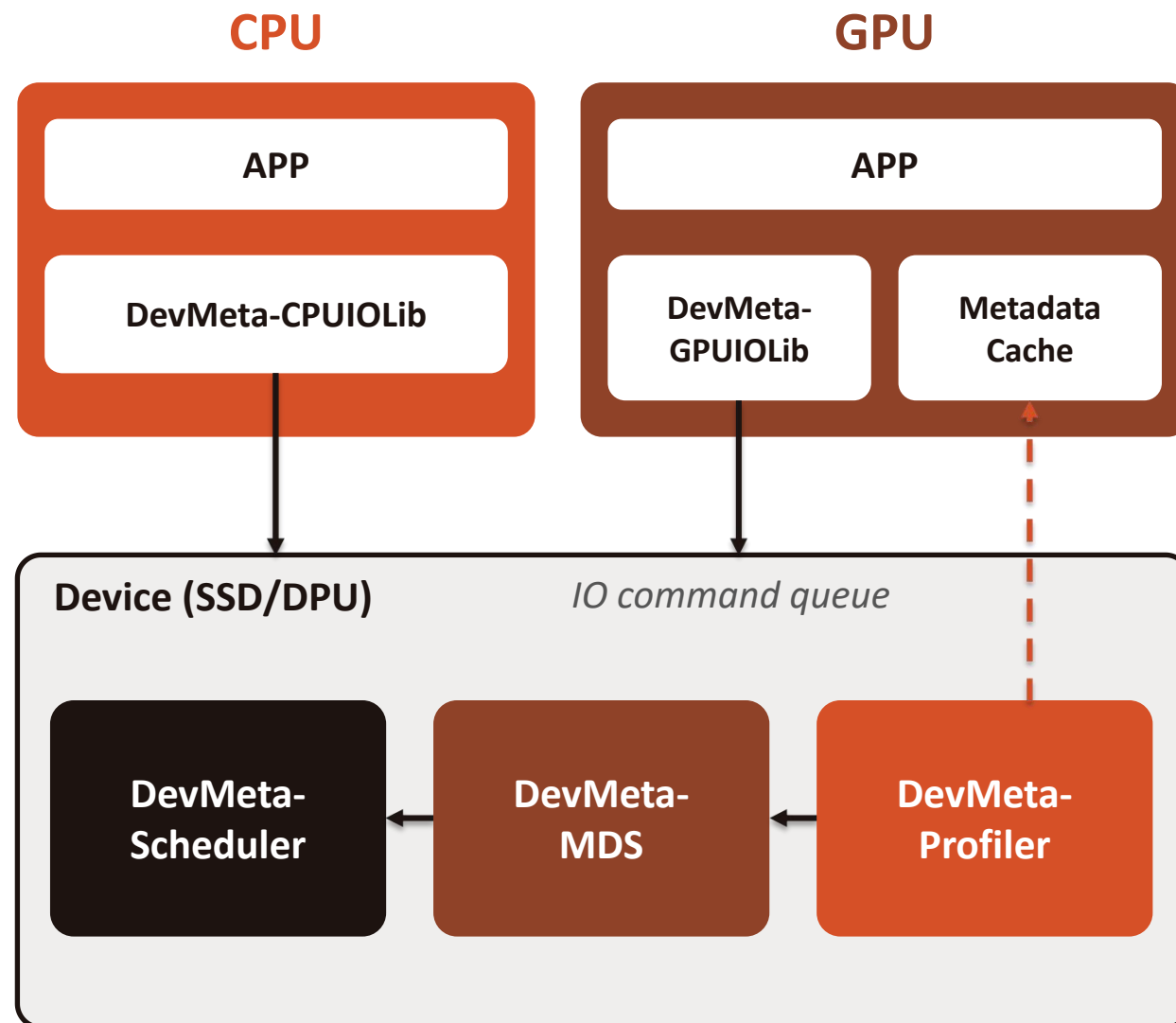
Speedup GPU Metadata lookup

DevMeta-MetaDataServer (MDS)

Store and Manage Metadata

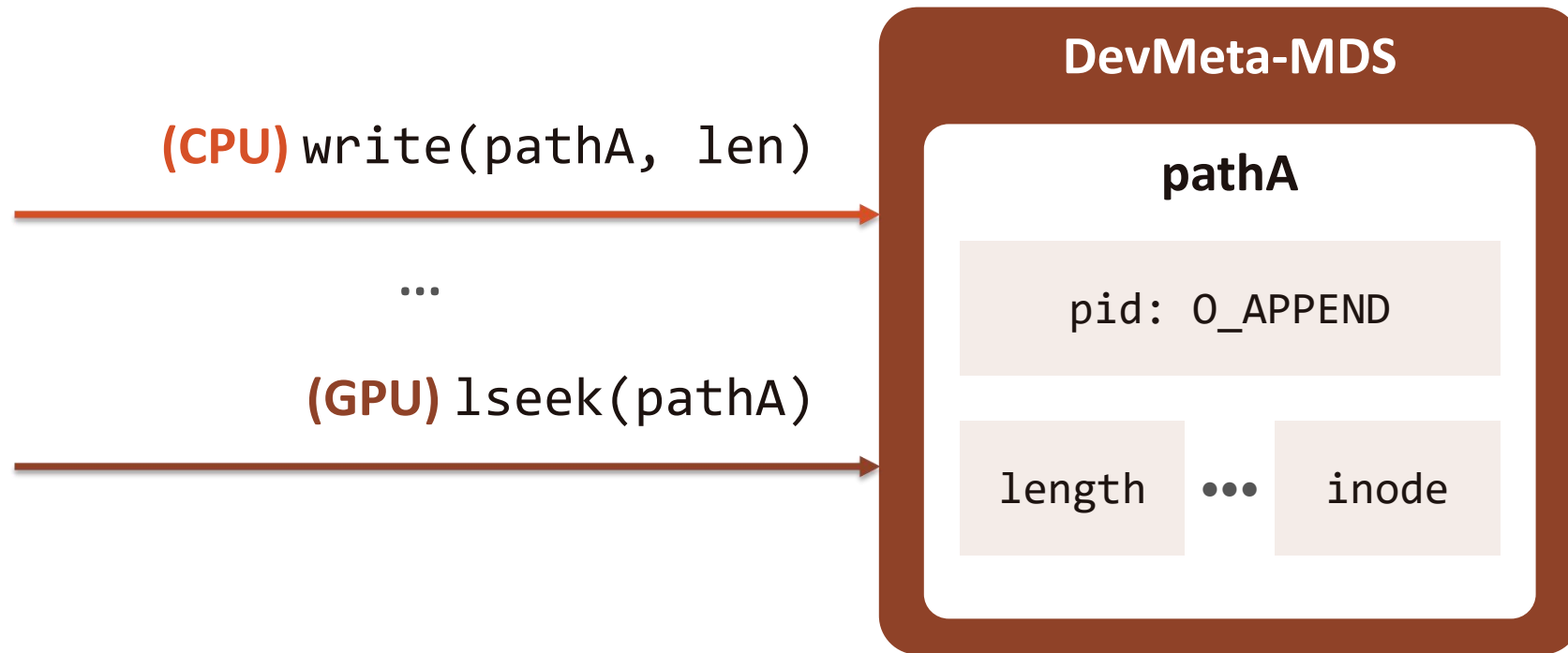
DevMeta-Scheduler

Dynamically combine small requests



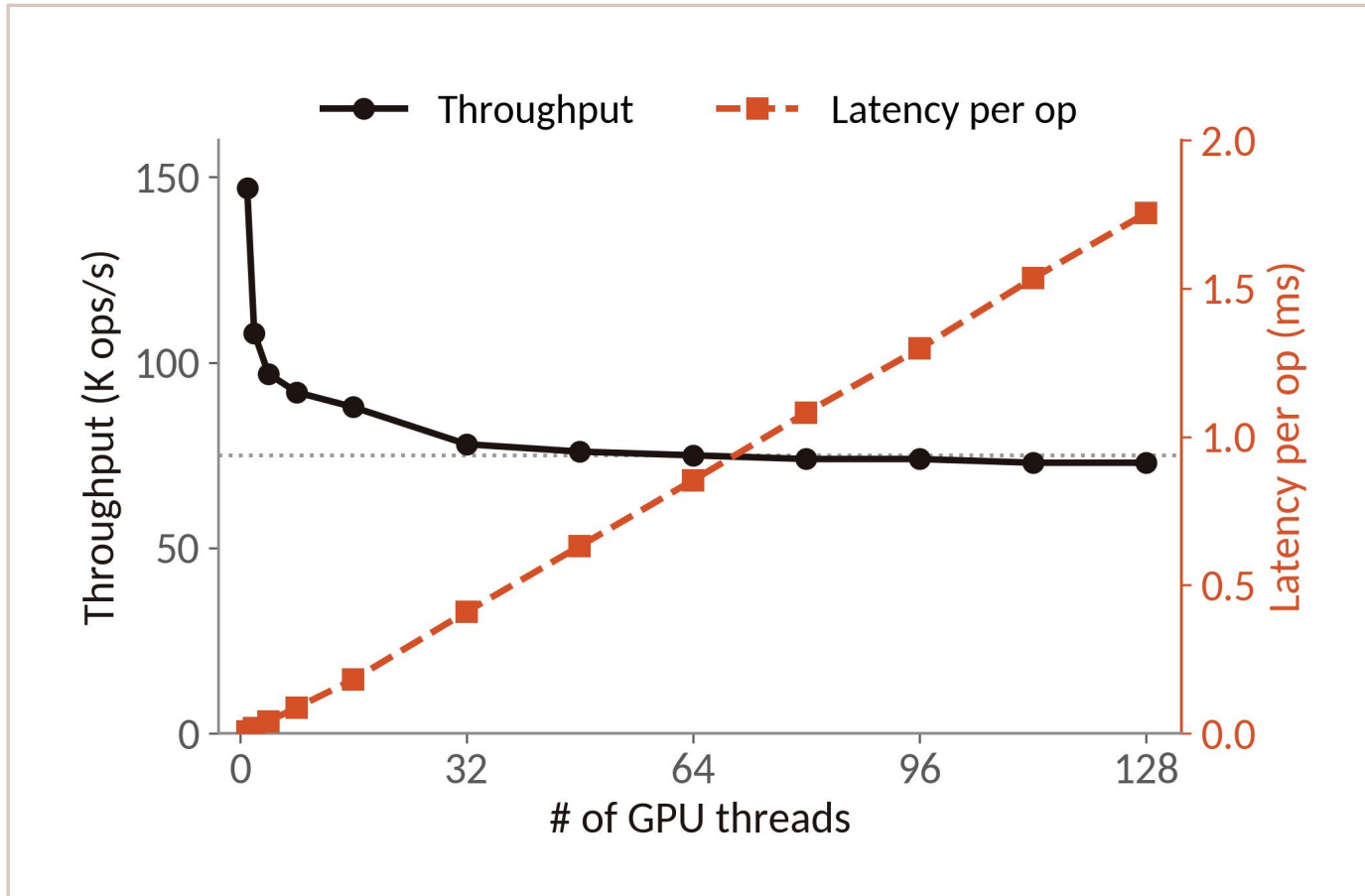
Device-side Metadata Controller

- **DevMeta synchronize updates from CPU or GPU immediately**
 - Updates from CPU and GPU are synchronized on the device side, therefore, any updates from CPU could be observed from GPU immediately, without any overhead.



Staying Efficient under GPU-Scale Concurrency

- Concentrating all metadata work on an embedded controller far weaker than the GPU raises a concern: **the device could become the new bottleneck.**



- **Host File System cannot keep up with many GPU threads**

— Both throughput and latency get worse because of queueing delay

2×

lower throughput

147K → 73K ops/s from 1 to 128 threads

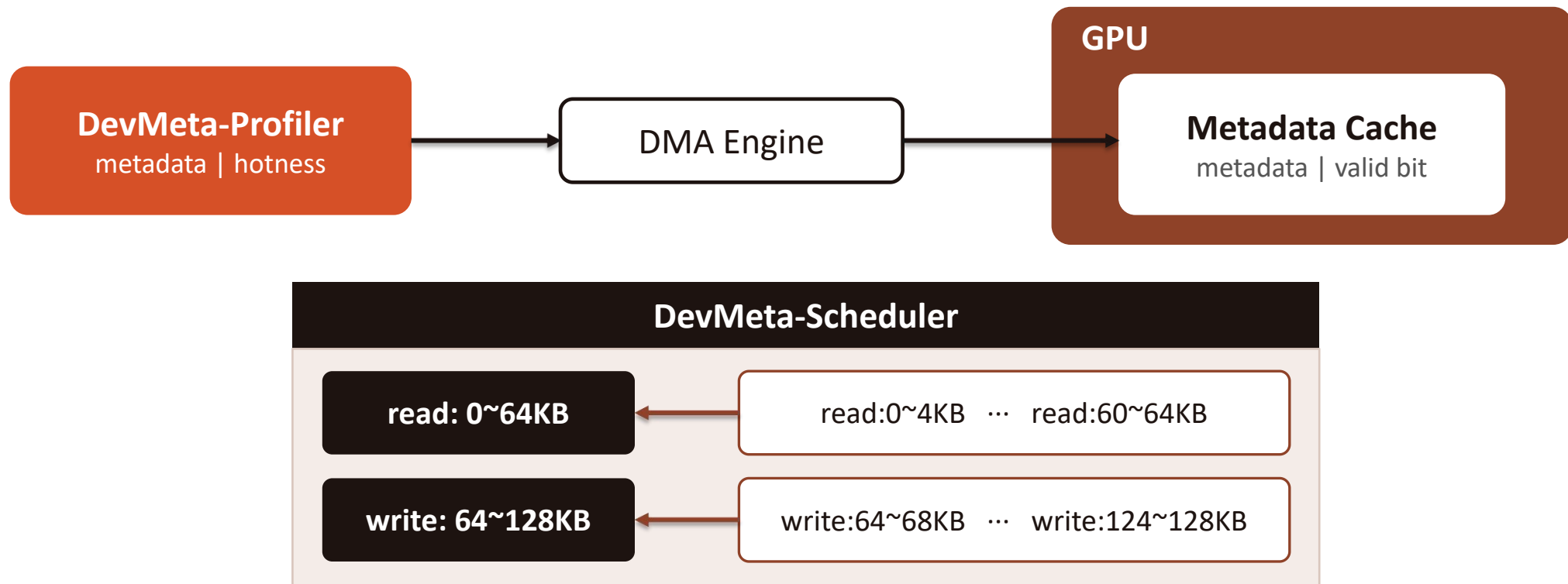
32

threads saturate the host

Throughput plateaus at ~75K ops/s

Staying Efficient under GPU-Scale Concurrency

- Concentrating all metadata work on an embedded controller far weaker than the GPU raises a concern: **the device could become the new bottleneck.**
 - Profiler manage and transfer cache to GPU
 - Scheduler dynamically combine small requests



Outline

- 1 Background & Motivation
- 2 Design
- 3 Evaluation**
- 4 Open Challenges
- 5 Conclusion

Evaluation: Experiment Setup

Platform

■ Hardware

- CPU: Dual-socket Intel Xeon Server
- GPU: NVIDIA A100 with 40GB Memory
- SSD: Samsung 990 Pro 1TB NVMe SSD (PCIe 4.0 ×4)

■ Software

- Ubuntu 22.04, Linux 6.8.0, CUDA 12.4, Driver 550.54

Evaluated file systems

1 **CPU-Resident** GeminiFS [FAST' 25]

A host metadata thread owns the block map; GPU threads send metadata requests through a pinned-memory queue, then issue data I/O themselves at the end of request process.

2 **GPU-Resident** GoFS [SOSP' 25]

A persistent GPU kernel owns the block map; CPU threads send metadata requests to a shared queue polling by the GPU kernel

3 **Device-Resident** DevMeta

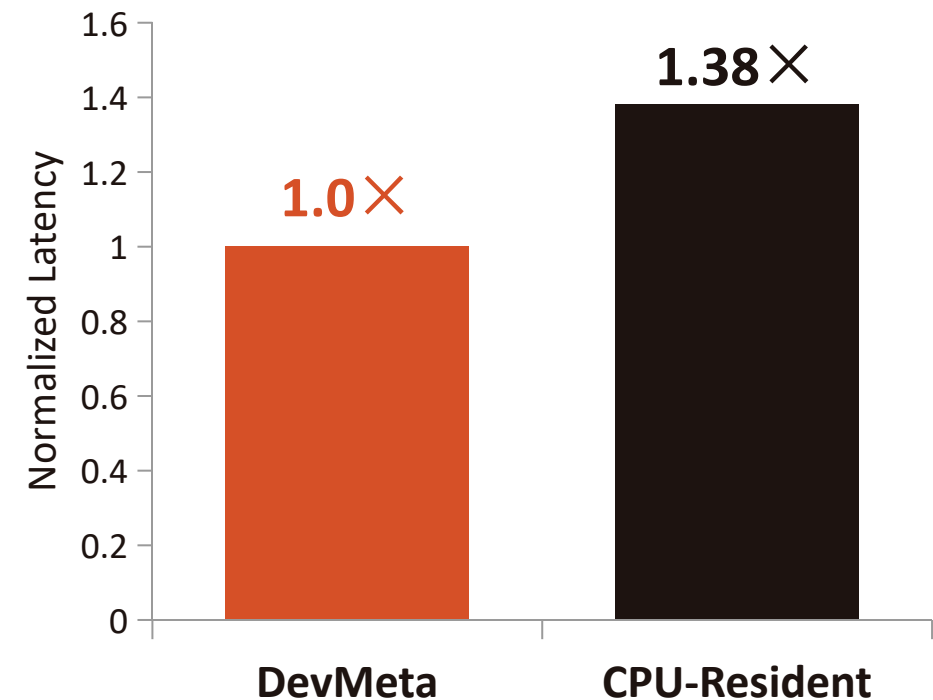
Metadata service on dedicated cores with throttled memory bandwidth and added PCIe latency. Follow [FAST' 18] [OSDI' 20]

CPU-Resident GPU File System

- **GPU append operation waits for block allocation on the host**
 - Requests are delayed by the PCIe round trip and synchronization of CPU

Why measure append?

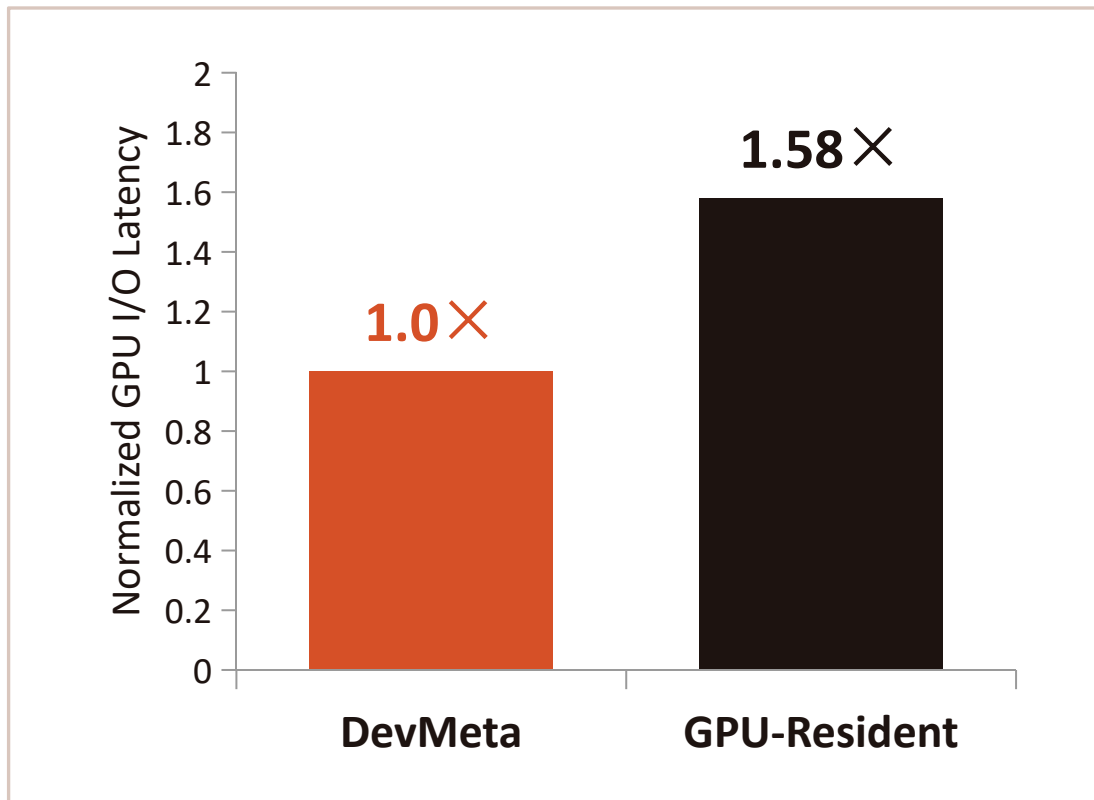
- 1 **Triggers metadata updates**
Allocating new blocks, extending the file length, etc.
- 2 **Common in GPU data pipelines**
A frequent operation in dataset preprocessing and KV cache offloading



GPU-Resident GPU File System

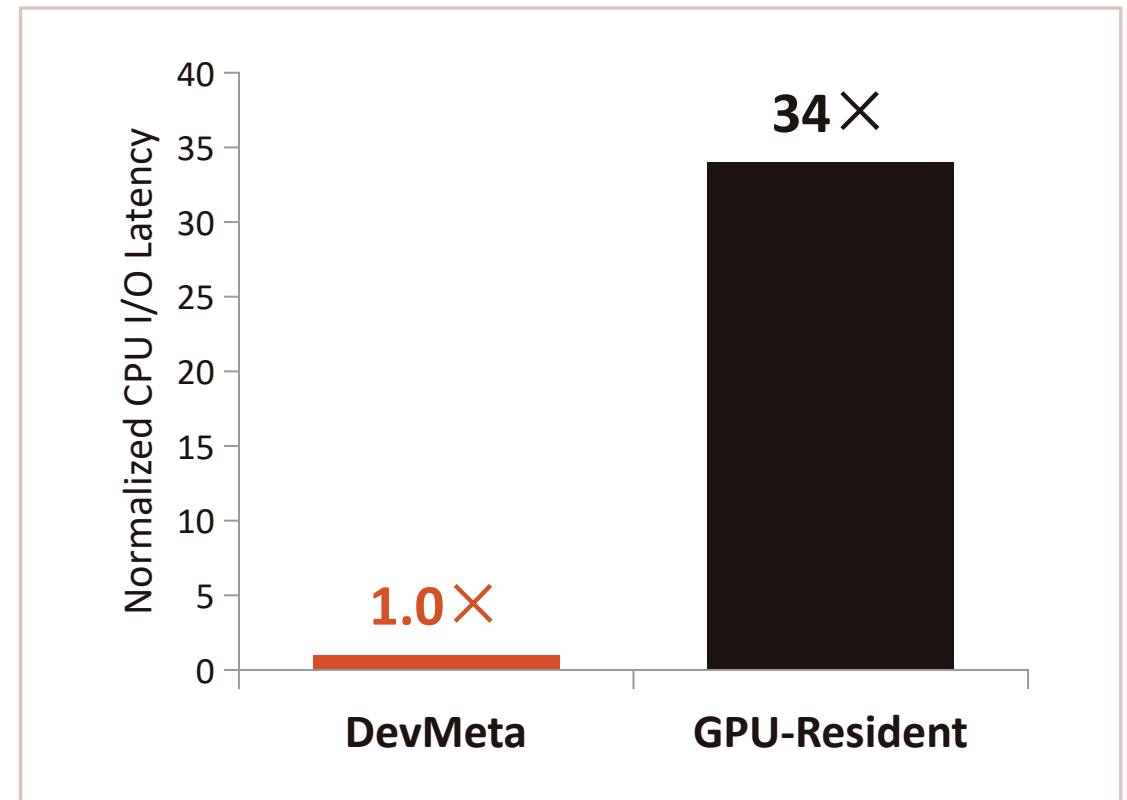
- GPU I/O stalls on CPU-side metadata updates

- Frequent CPU-side metadata updates stall ongoing GPU I/O processing



- CPU I/O must go through the GPU-resident FS

- Massive GPU I/O requests largely delay CPU requests



Outline

- 1 Background & Motivation
- 2 Design
- 3 Evaluation
- 4 Open Challenges**
- 5 Conclusion

Open Challenges

1

Multi-device scalability

Metadata and data distributed and striped to multi-device in a fine-grained manner might help

2

Preserving host-side optimizations

Cooperation between device file system with host processor might help

3

Crash consistency

Keeping a log might help

4

Deployability

Implementation and measure on programmable storage device might help

Outline

- 1 Background
- 2 Motivation
- 3 Design
- 4 Open Challenges
- 5 Conclusion**

DevMeta

Device-side File System that eliminate metadata synchronization

1

**DevMeta
Metadata Server
that synchronizes
CPU/GPU
Request**

2

**Stateless IO
Libraries that
maintain
programmability**

3

**Device side
Profiler that
cache hot data**

4

**Device side
Scheduler that
batch and
combine requests**

DevMeta

Bridging CPU and GPU I/O with a Device-Resident File System

Thanks for Listening!