SNIA DEVELOPER CONFERENCE
SDC 26 | SANTA CLARA, CA SEPTEMBER 28-30
BY Developers FOR Developers

www.sniadeveloper.org

AiSIO

Orchestrating Storage I/O Across CPUs and Accelerators

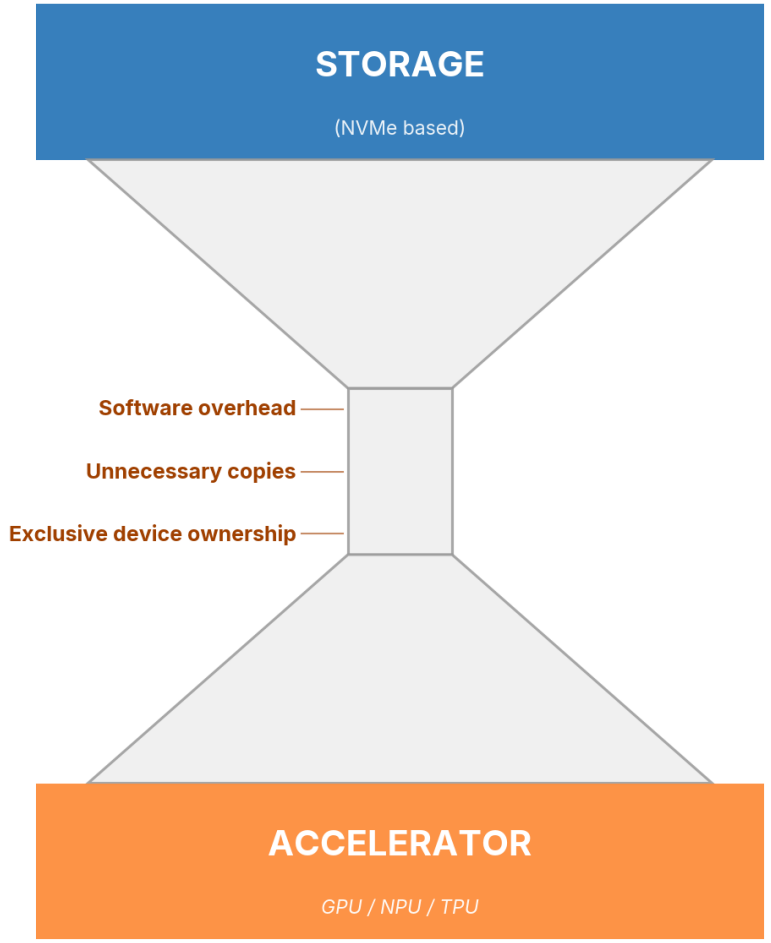
Simon A. F. Lund, Samsung

Feeding Storage to Accelerators

Three challenges to address

- Software overhead at every layer
  - Application, libraries, and language runtime on the I/O hot path
  - User/kernel boundary cost on every I/O
  - OS storage stack runs on every I/O (block layer, filesystem, page cache)
  - Drivers trade per-I/O cost for generality and abstraction support
- Unnecessary copies
  - Data routes through host DRAM before reaching the accelerator
- Exclusive device ownership
  - One driver and usage policy precludes coexistence of kernel, user-space, and accelerator I/O paths

Today's storage stack is built for CPU consumers; the same stack used by an accelerator hits these three obstacles.



 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O (AiSIO)

The vision: Accelerators as first-class storage citizens

- Accelerators: GPUs, NPUs, TPUs, and similar
  - Current focus is on a single class of accelerator: GPUs
- Preserve files, filesystems, and OS control
- Accelerator-integrated Storage I/O modes:
  - **CPU-initiated hostmem**: CPU drives I/O, data lands in host memory (possibly copied to accelerator)
  - **CPU-initiated P2P**: CPU drives I/O, data direct to device memory
  - **Accelerator-assisted P2P**: CPU drives I/O, completion mirrored beside the payload on the accelerator
  - **Accelerator-initiated P2P (aka GPU-initiated)**: Accelerator drives I/O directly into its own memory
- Interoperability over replacement
  - Extend the existing stack without sacrificing performance
- All open source; in or targeting upstream, with an open collaboration model

 ©2026 SNIA Developer Conference. All Rights Reserved.

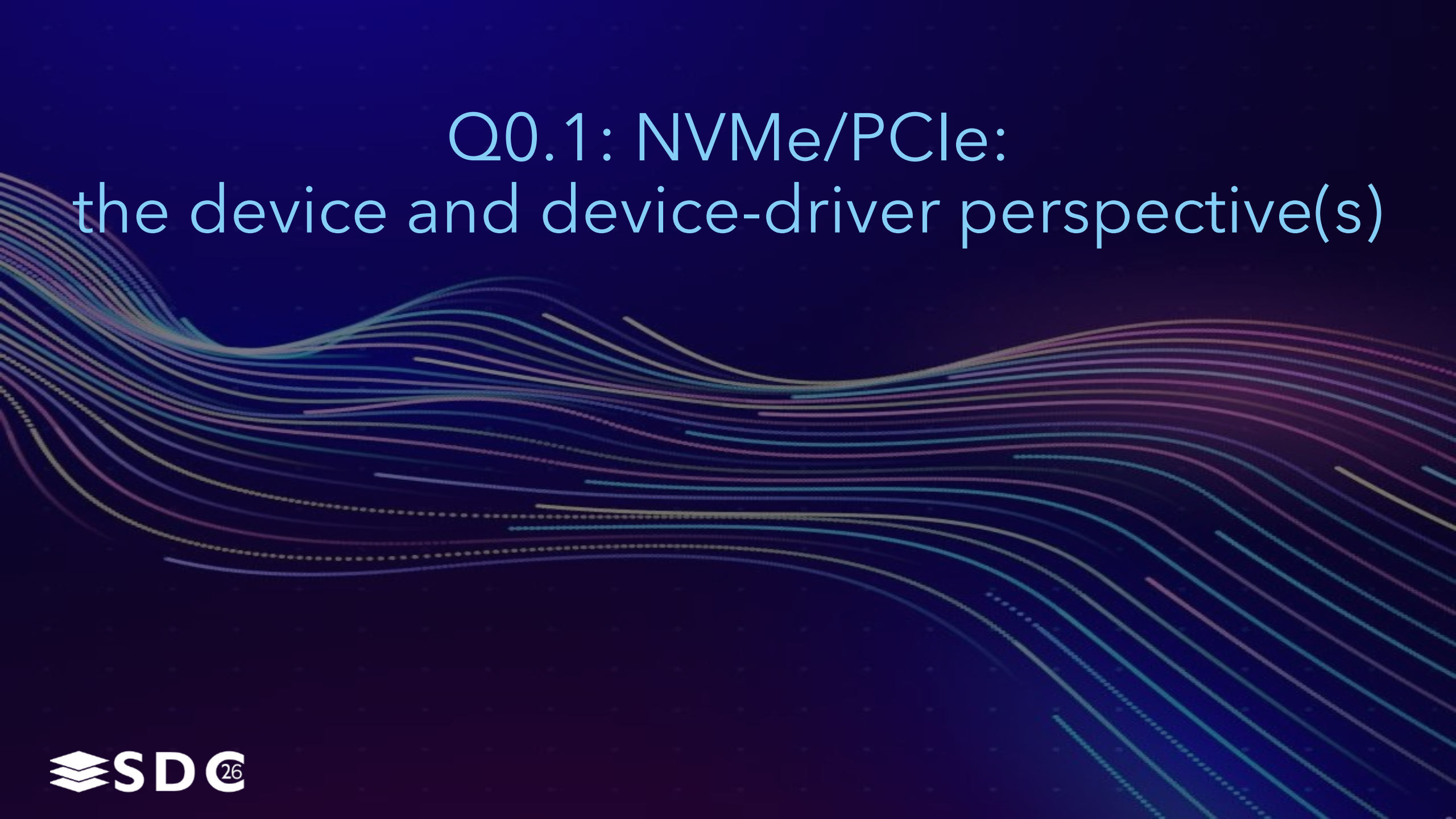
What this talk covers

Roadmap for the presentation

- Q0: What are the I/O modes behind Accelerator-integrated Storage I/O?
  - NVMe/PCIe primer and the Accelerator-integrated Storage I/O modes
- Q1: Where did it start?
  - The initial proof-of-concept (AiSIO PoC) and its limitations
- Q2: What is it now?
  - The open source implementation (AiSIO OSS), Host-orchestrated Multipath I/O, and the component stack
- Q3: How does it behave and perform?
- Q4: Where do we go from here?

 ©2026 SNIA Developer Conference. All Rights Reserved.

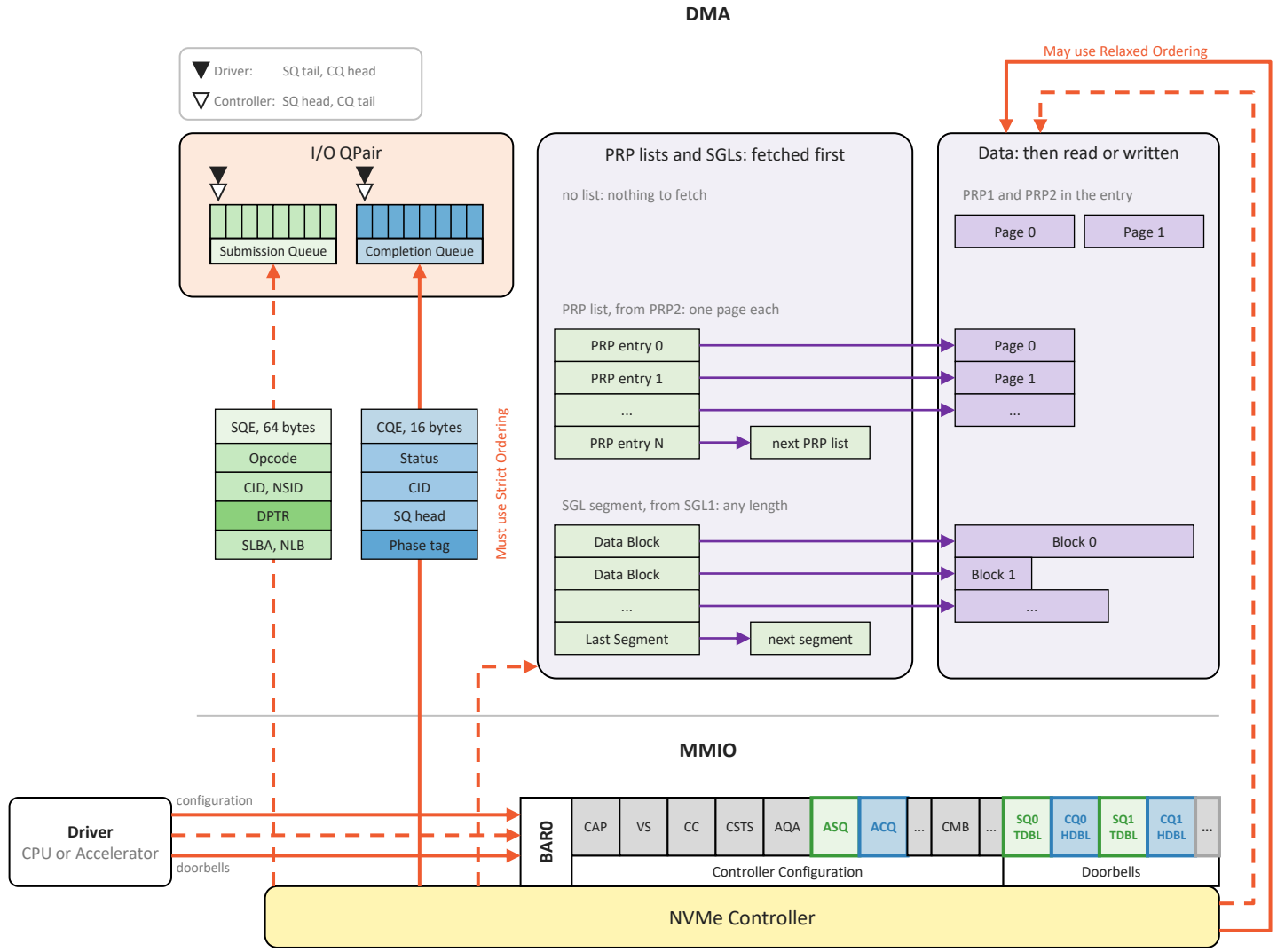
Q0.1: NVMe/PCIe: the device and device-driver perspective(s)



NVMe/PCIe: communication and data movement

From the device and device-driver's point of view

- **DMA:** memory TLPs the controller issues, targeting host or peer memory
  - The controller is the requester; the memory whose addresses it was handed is the destination
  - SQEs and CQEs: fetched from a submission queue, posted to a completion queue
  - PRP lists and SGLs: reached from the pointer in the entry, fetched, never written
  - Data: reached from the PRPs or SGL; written by a Read command, read by a Write command
- **MMIO:** memory TLPs to the controller's BAR0
  - The controller is the destination; whoever drives it is the requester
  - Controller configuration, at enablement: read CAP; clear CC.EN; point ASQ, ACQ and AQA at the admin pair; write CC; set CC.EN and wait for CSTS.RDY
  - Doorbells: ring to tell the controller an index moved, SQ tail or CQ head; one ring can cover several commands



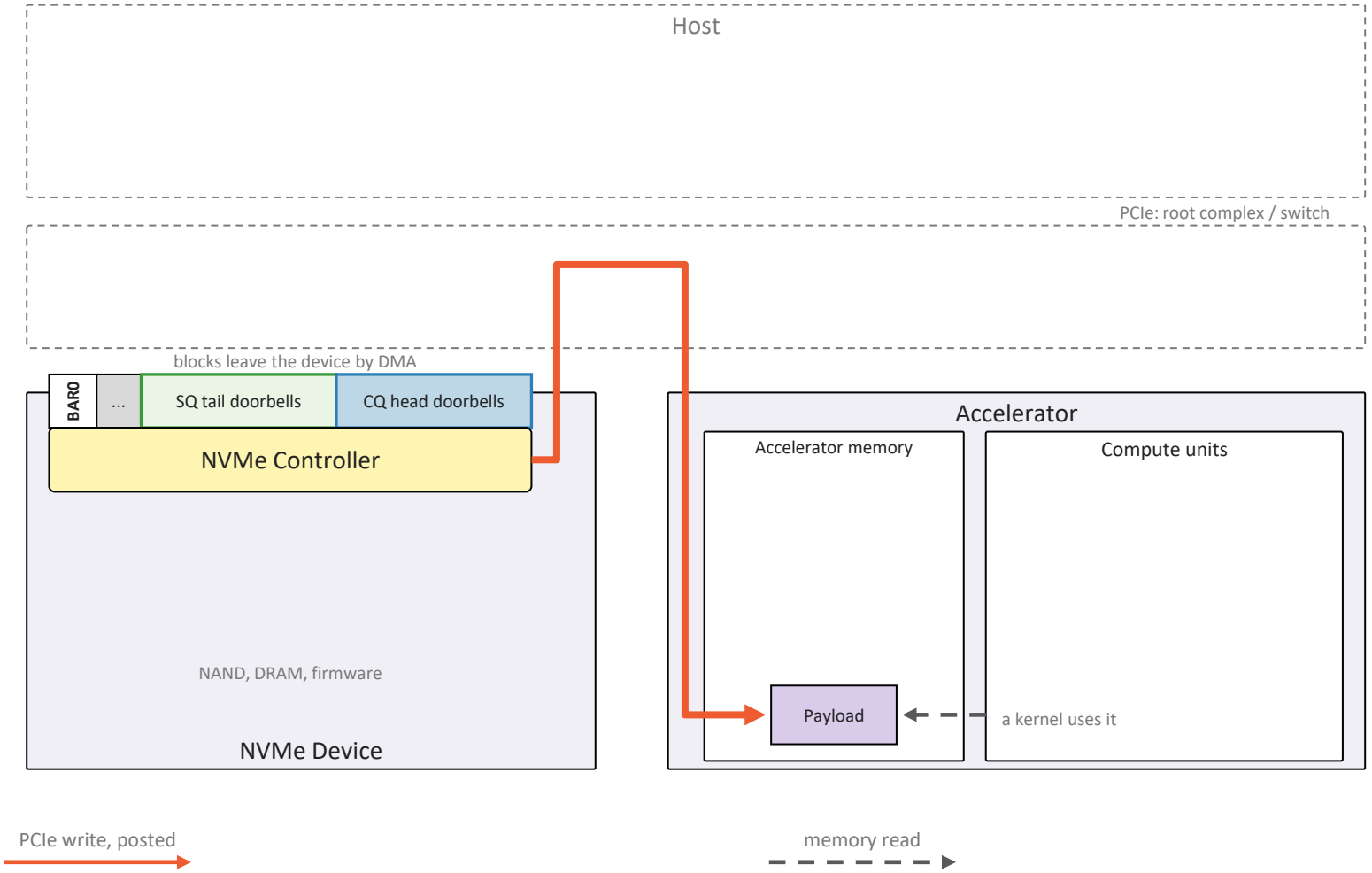
 ©2026 SNIA Developer Conference. All Rights Reserved.

Q0.2: Accelerator-integrated Storage I/O modes



Accelerator-integrated Storage I/O modes

The I/O modes illustrated with an NVMe Read command



Every mode ends the same way

Blocks leave the device by DMA and are used by a kernel in accelerator memory

What the modes decide

Where the queues, the lists and the payload buffer live

Who initiates, who reaps

Two kinds of answer

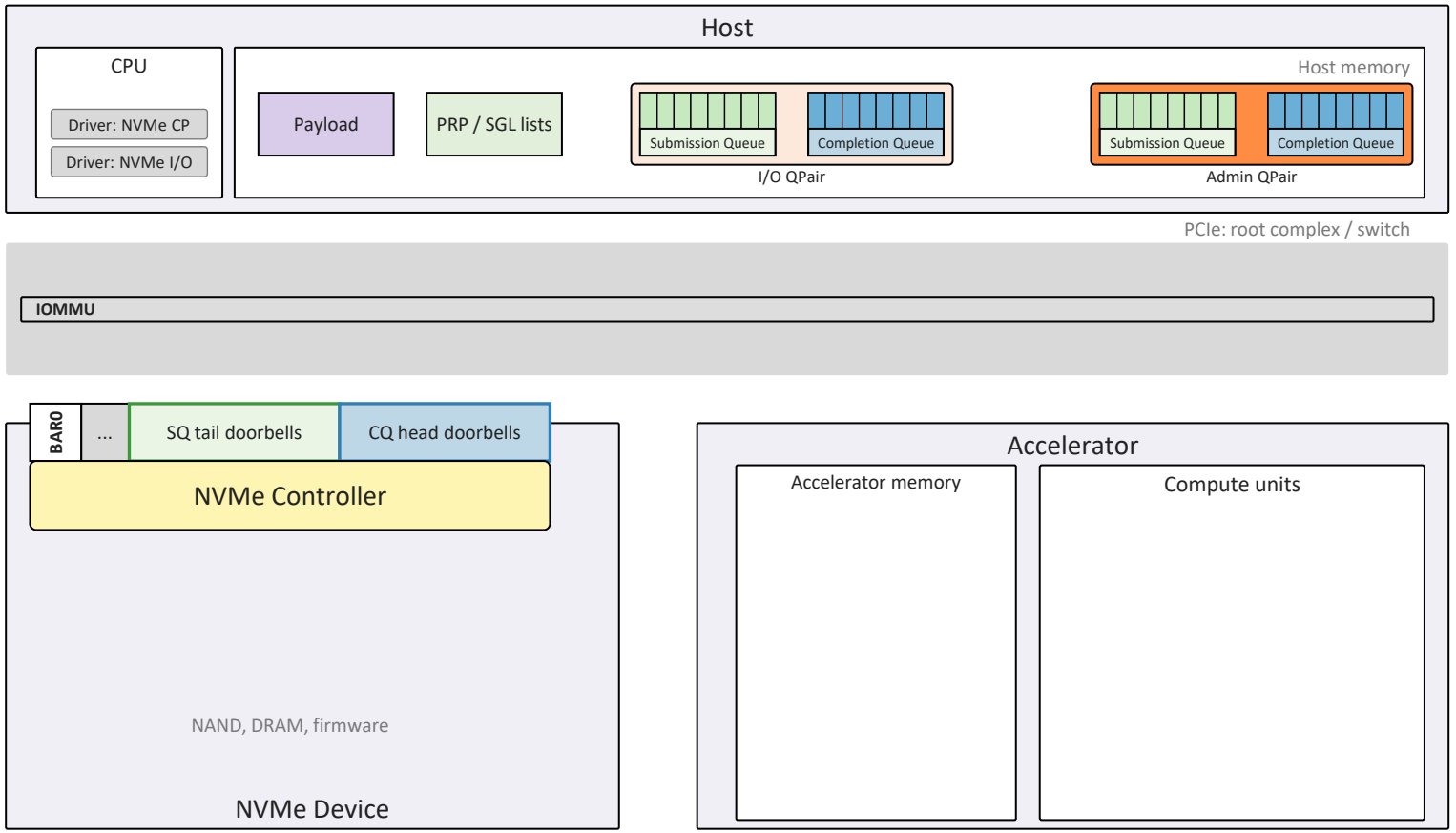
Add a copy step after the command: the conventional path

Change where the command itself lands: the accelerator-integrated modes

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem, at rest before step 1



I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free

COMMAND COMPLETION PROCESSING (REAPER)

7. CPU spins on the phase tag, reads the 16-byte CQE, and validates it via CID

8. CPU rings the CQ head doorbell; the slot is free again

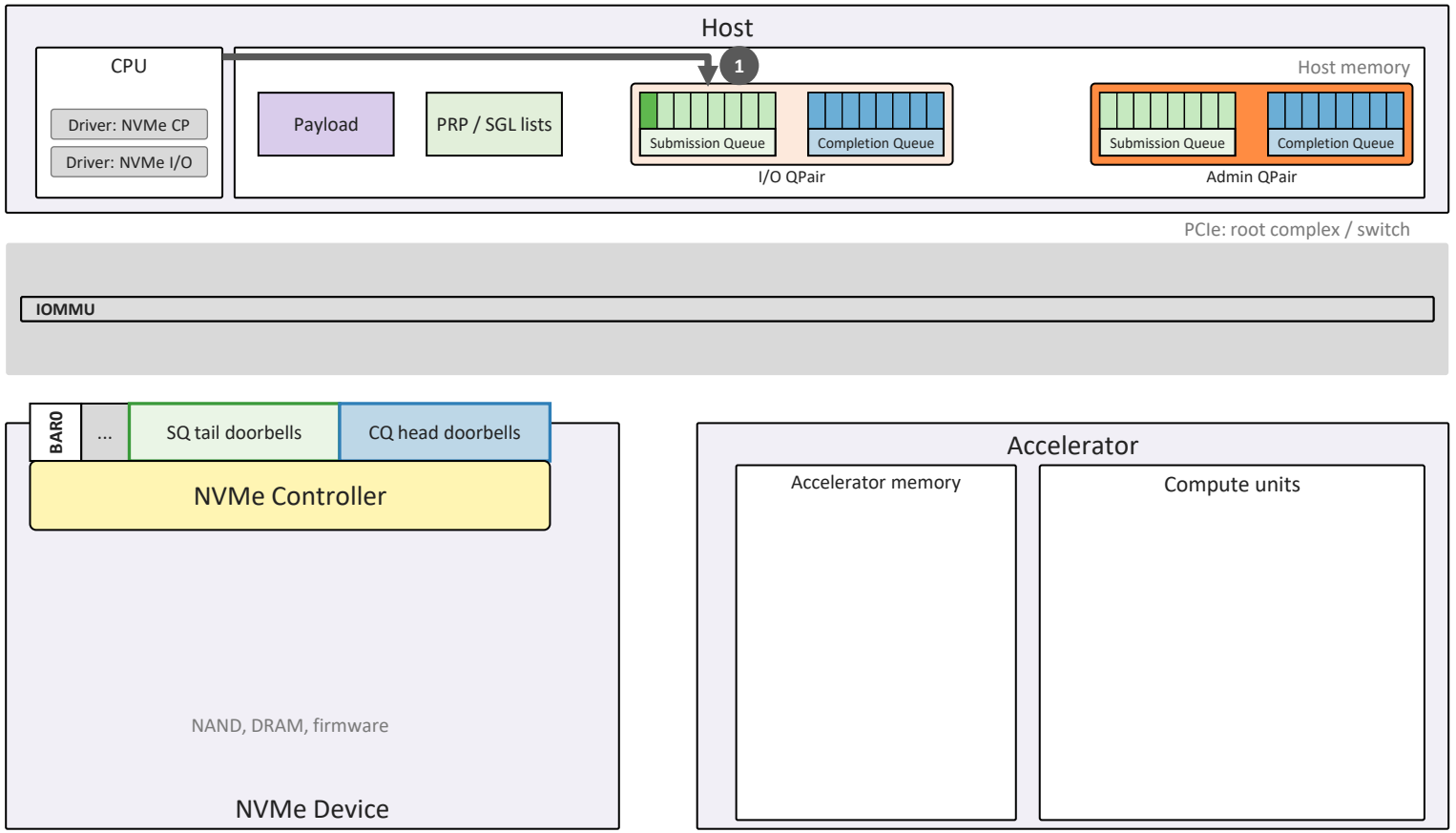
CONSUMPTION (CONSUMER)

9. The accelerator's runtime, CUDA, HIP, OpenCL, whichever manages it, **copies the payload** from host memory into accelerator memory, a host-to-device transfer; a kernel uses it there

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem, step 1 of 9



I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free

COMMAND COMPLETION PROCESSING (REAPER)

7. CPU spins on the phase tag, reads the 16-byte CQE, and validates it via CID

8. CPU rings the CQ head doorbell; the slot is free again

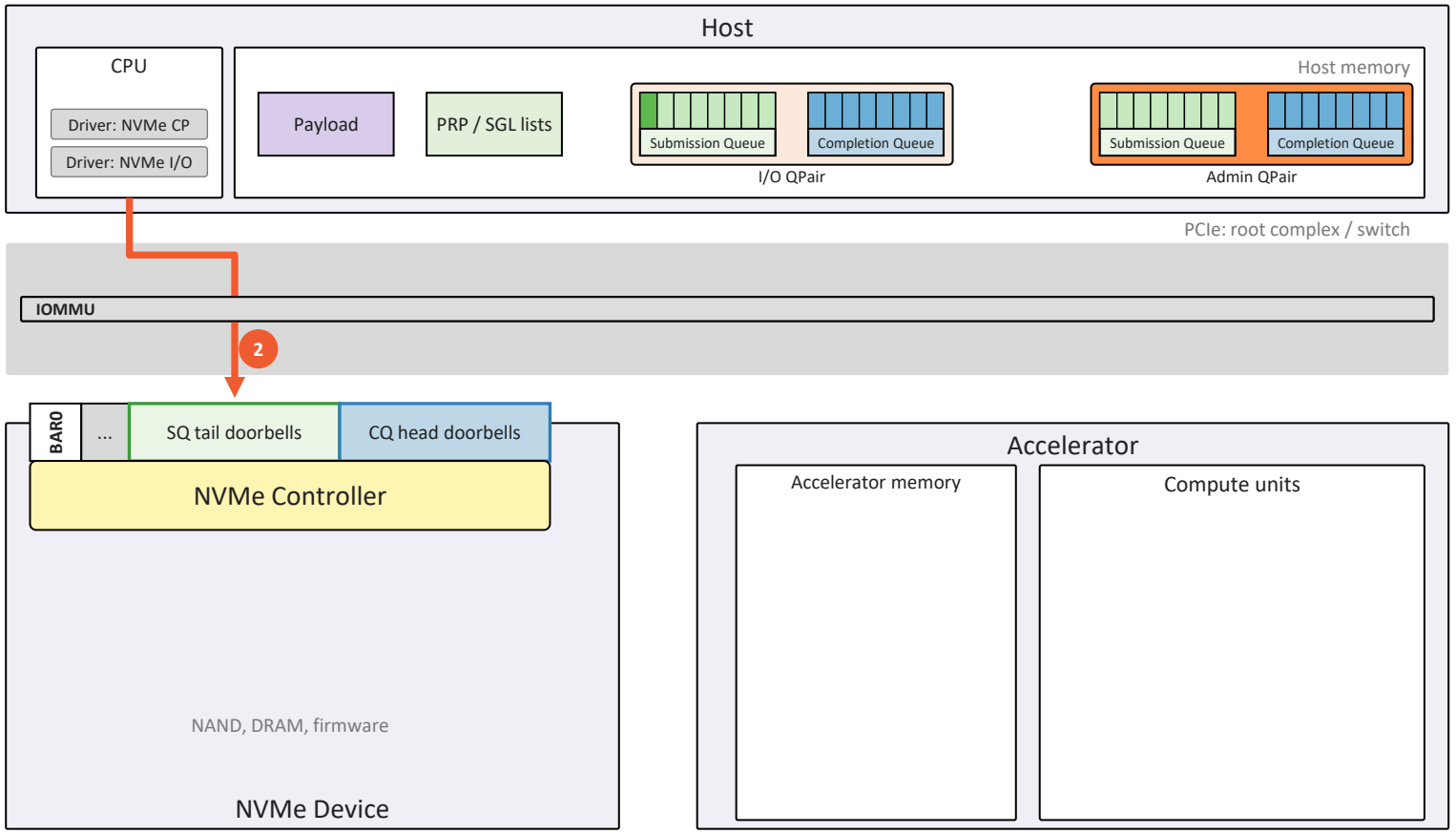
CONSUMPTION (CONSUMER)

9. The accelerator's runtime, CUDA, HIP, OpenCL, whichever manages it, **copies the payload** from host memory into accelerator memory, a host-to-device transfer; a kernel uses it there

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem, step 2 of 9



I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free

COMMAND COMPLETION PROCESSING (REAPER)

7. CPU spins on the phase tag, reads the 16-byte CQE, and validates it via CID

8. CPU rings the CQ head doorbell; the slot is free again

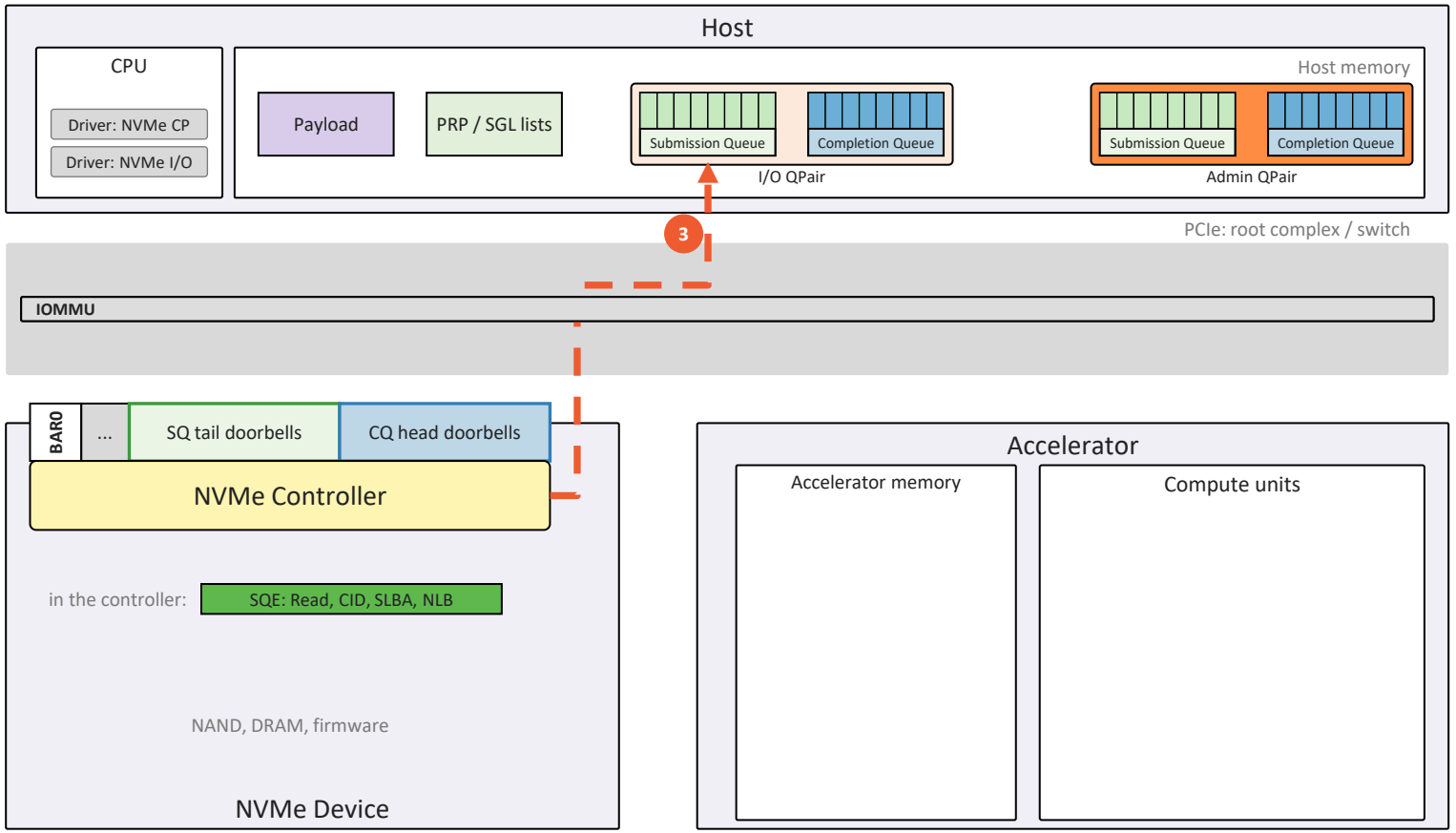
CONSUMPTION (CONSUMER)

9. The accelerator's runtime, CUDA, HIP, OpenCL, whichever manages it, **copies the payload** from host memory into accelerator memory, a host-to-device transfer; a kernel uses it there

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem, step 3 of 9



PCIe write, posted — PCIe read, non-posted — memory write

I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free

COMMAND COMPLETION PROCESSING (REAPER)

7. CPU spins on the phase tag, reads the 16-byte CQE, and validates it via CID

8. CPU rings the CQ head doorbell; the slot is free again

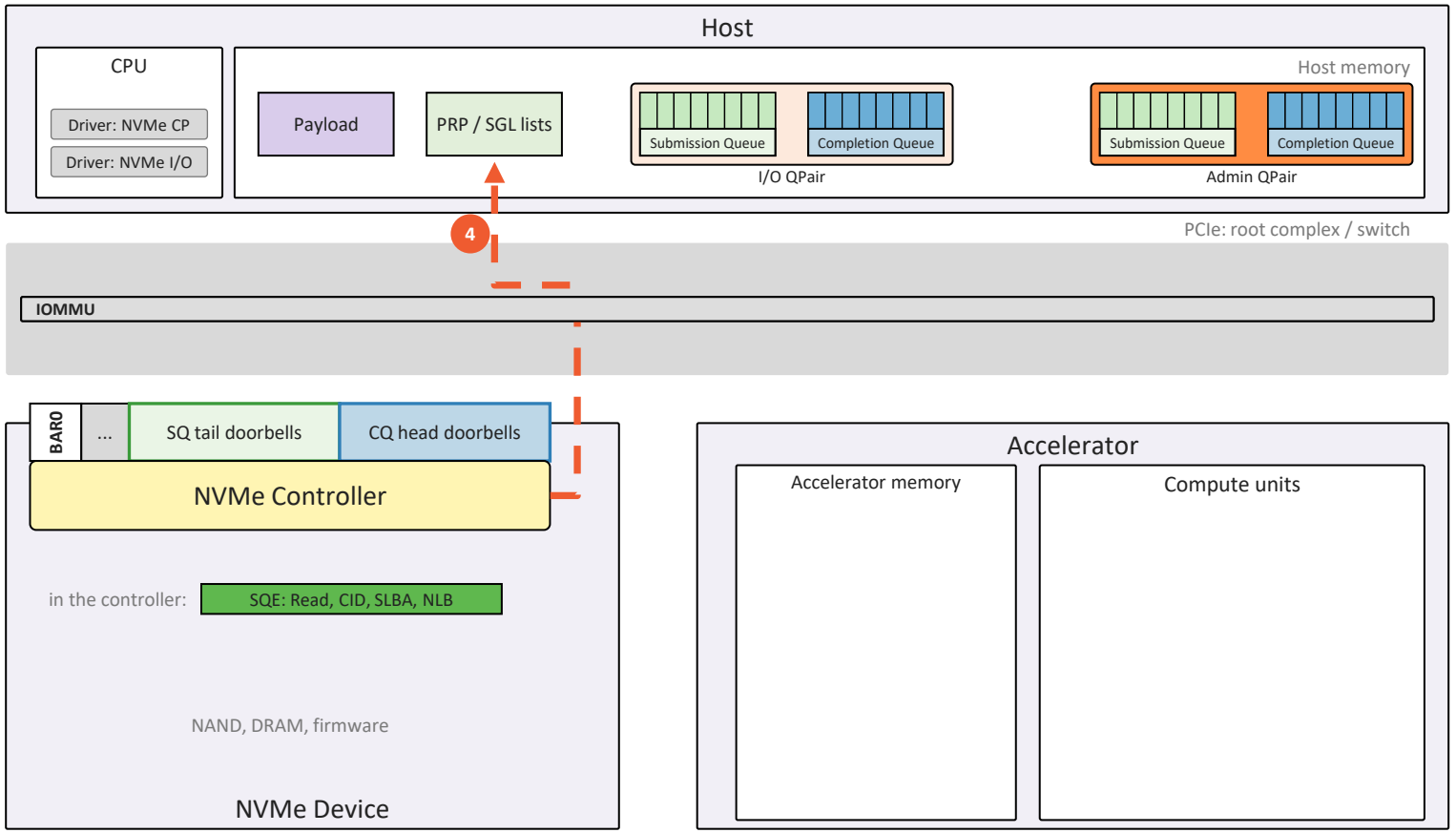
CONSUMPTION (CONSUMER)

9. The accelerator's runtime, CUDA, HIP, OpenCL, whichever manages it, **copies the payload** from host memory into accelerator memory, a host-to-device transfer; a kernel uses it there

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem, step 4 of 9



I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free

COMMAND COMPLETION PROCESSING (REAPER)

7. CPU spins on the phase tag, reads the 16-byte CQE, and validates it via CID

8. CPU rings the CQ head doorbell; the slot is free again

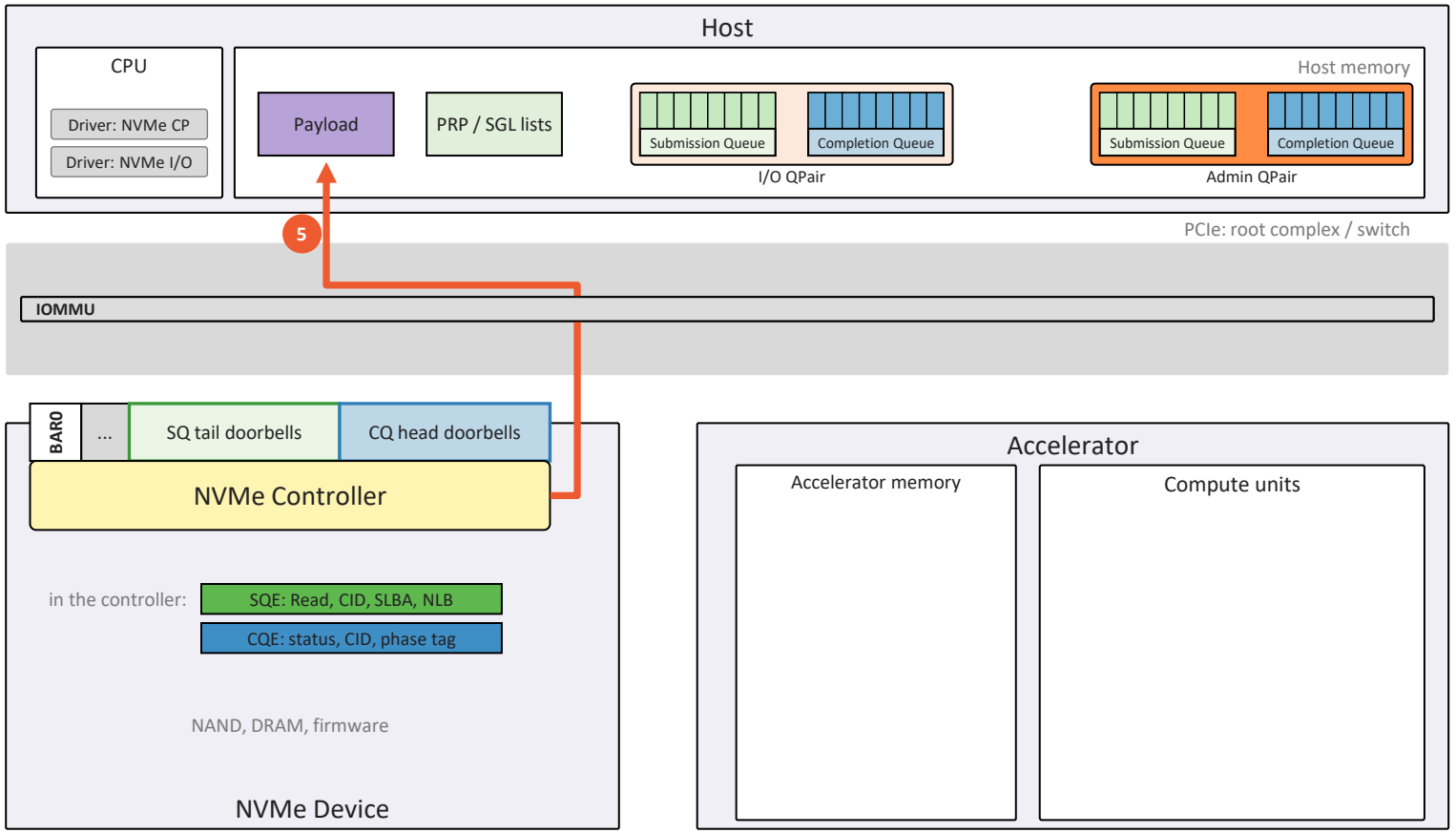
CONSUMPTION (CONSUMER)

9. The accelerator's runtime, CUDA, HIP, OpenCL, whichever manages it, **copies the payload** from host memory into accelerator memory, a host-to-device transfer; a kernel uses it there

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem, step 5 of 9



I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free

COMMAND COMPLETION PROCESSING (REAPER)

7. CPU spins on the phase tag, reads the 16-byte CQE, and validates it via CID

8. CPU rings the CQ head doorbell; the slot is free again

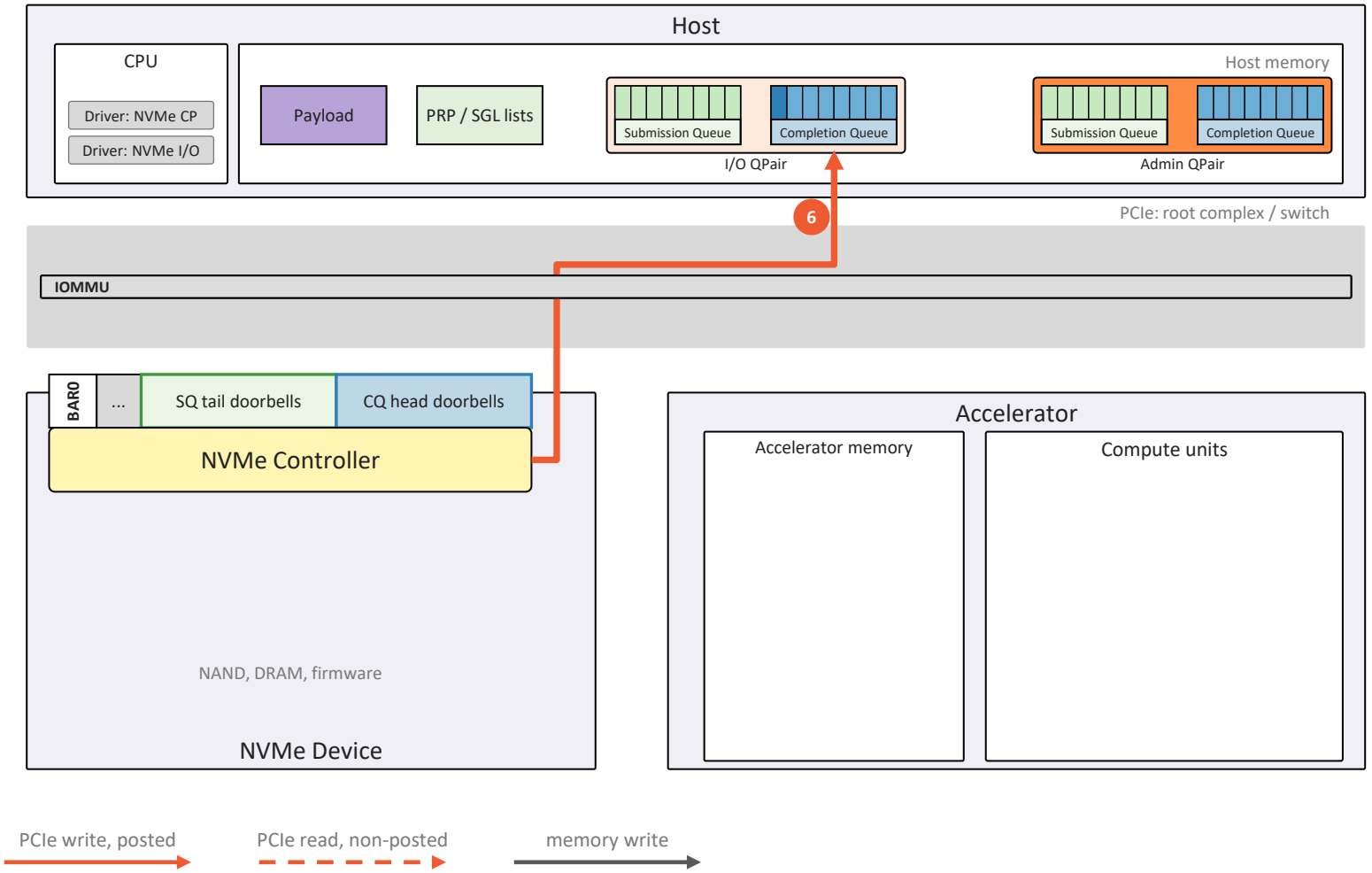
CONSUMPTION (CONSUMER)

9. The accelerator's runtime, CUDA, HIP, OpenCL, whichever manages it, **copies the payload** from host memory into accelerator memory, a host-to-device transfer; a kernel uses it there

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem, step 6 of 9



I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free

COMMAND COMPLETION PROCESSING (REAPER)

7. CPU spins on the phase tag, reads the 16-byte CQE, and validates it via CID

8. CPU rings the CQ head doorbell; the slot is free again

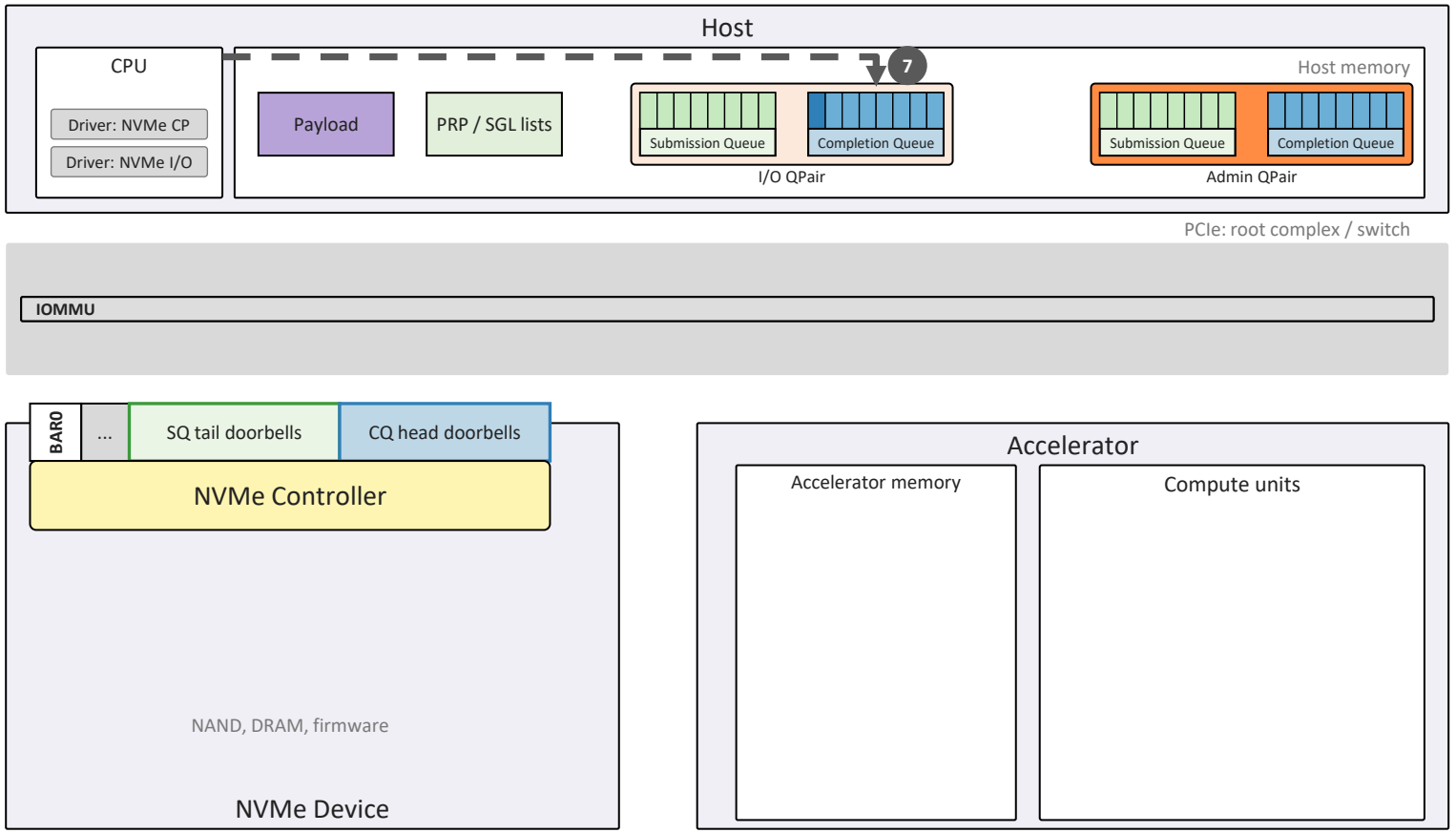
CONSUMPTION (CONSUMER)

9. The accelerator's runtime, CUDA, HIP, OpenCL, whichever manages it, **copies the payload** from host memory into accelerator memory, a host-to-device transfer; a kernel uses it there

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem, step 7 of 9



PCIe write, posted — PCIe read, non-posted — memory write — memory read

I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free

COMMAND COMPLETION PROCESSING (REAPER)

7. CPU spins on the phase tag, reads the 16-byte CQE, and validates it via CID

8. CPU rings the CQ head doorbell; the slot is free again

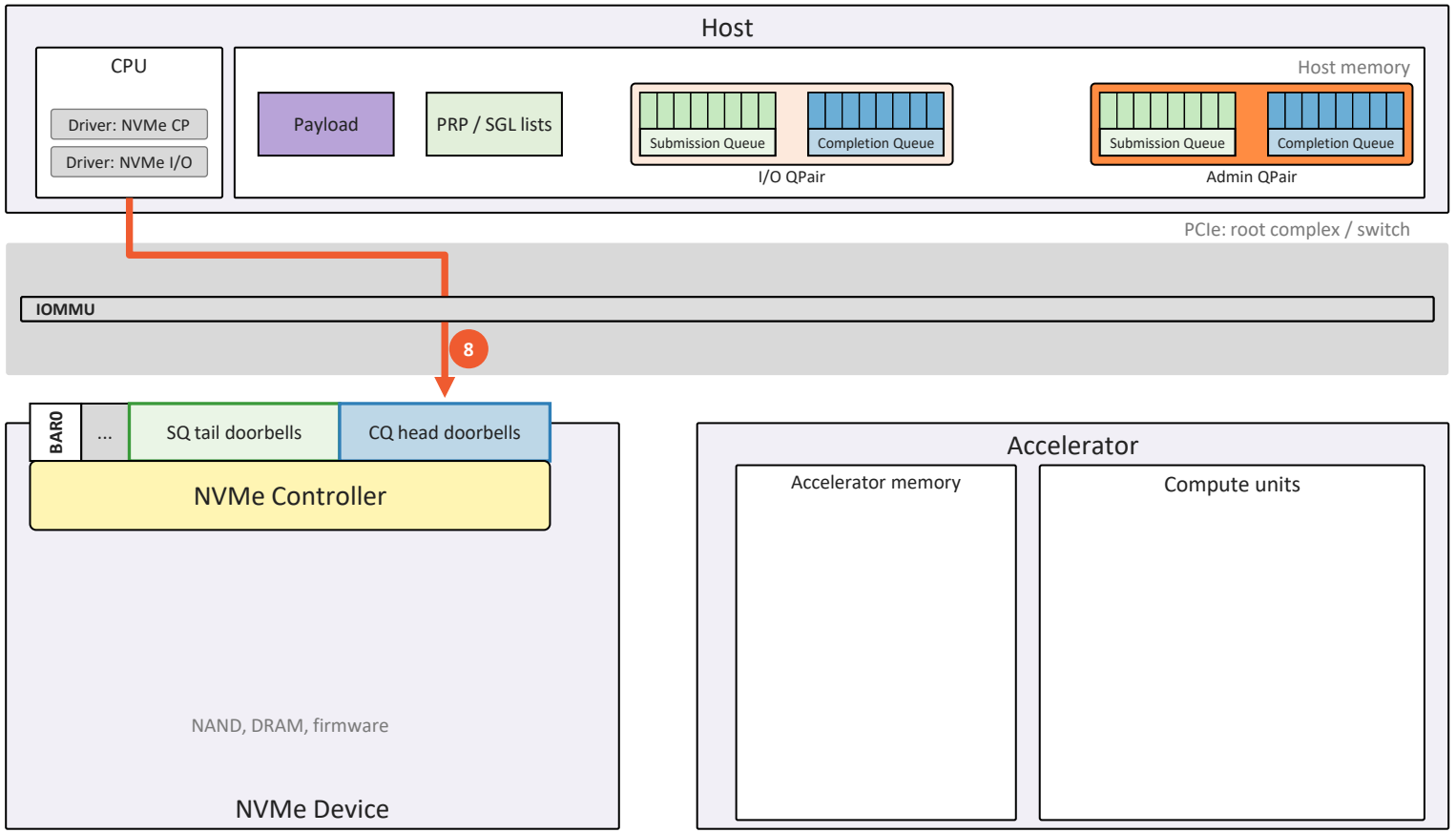
CONSUMPTION (CONSUMER)

9. The accelerator's runtime, CUDA, HIP, OpenCL, whichever manages it, **copies the payload** from host memory into accelerator memory, a host-to-device transfer; a kernel uses it there

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem, step 8 of 9



I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free

COMMAND COMPLETION PROCESSING (REAPER)

7. CPU spins on the phase tag, reads the 16-byte CQE, and validates it via CID

8. CPU rings the CQ head doorbell; the slot is free again

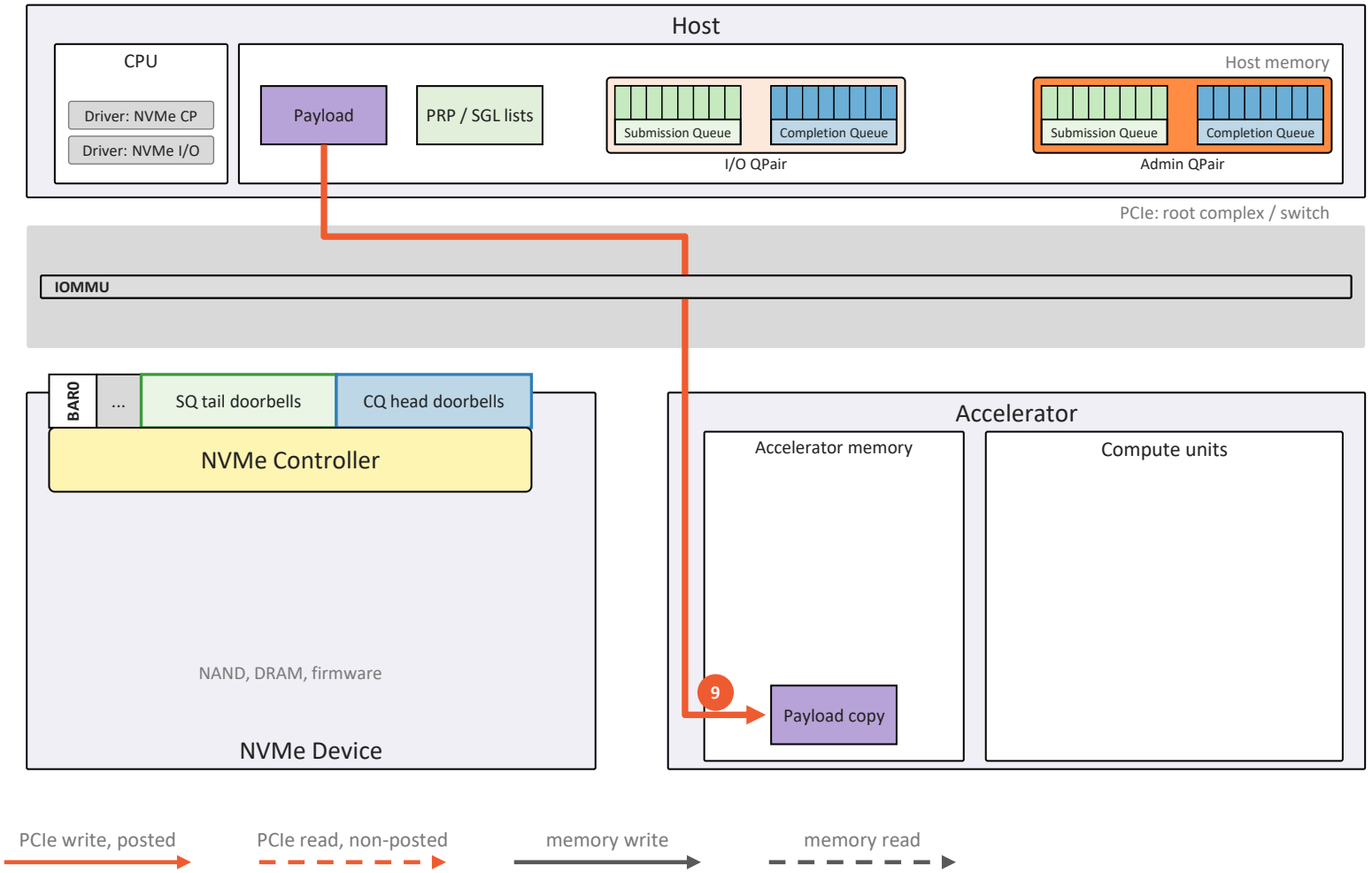
CONSUMPTION (CONSUMER)

9. The accelerator's runtime, CUDA, HIP, OpenCL, whichever manages it, **copies the payload** from host memory into accelerator memory, a host-to-device transfer; a kernel uses it there

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem, step 9 of 9



I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free

COMMAND COMPLETION PROCESSING (REAPER)

7. CPU spins on the phase tag, reads the 16-byte CQE, and validates it via CID

8. CPU rings the CQ head doorbell; the slot is free again

CONSUMPTION (CONSUMER)

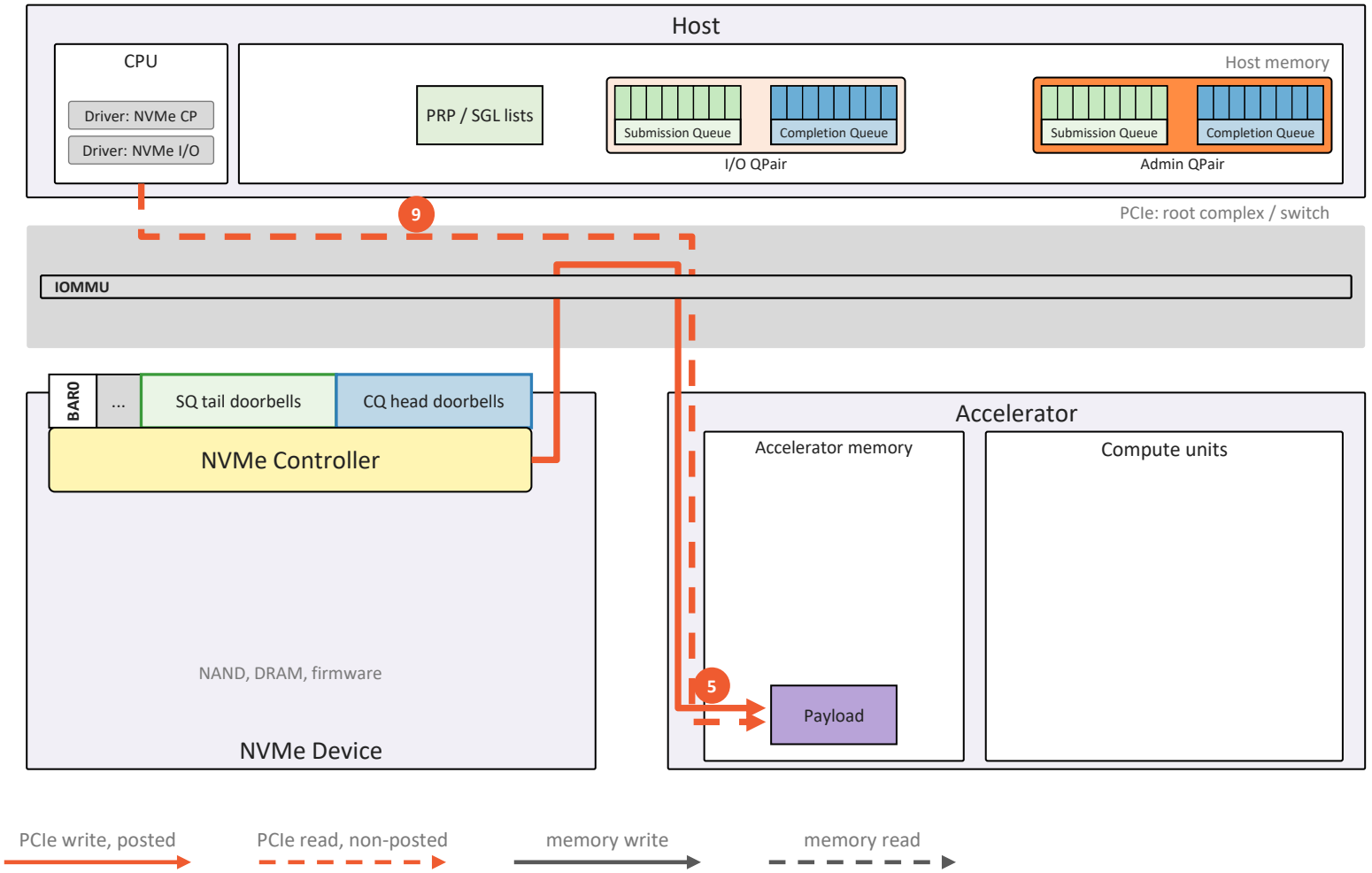
9. The accelerator's runtime, CUDA, HIP, OpenCL, whichever manages it, **copies the payload** from host memory into accelerator memory, a host-to-device transfer; a kernel uses it there

A step added, not a step changed.
The I/O never touches the accelerator; the CPU plays every host-side role. The copy is a second transfer, per record. Every mode that follows points the command somewhere else instead.

 ©2026 SNIA Developer Conference. All Rights Reserved.

CPU-initiated P2P

The command points its payload at accelerator memory



I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to; **the entries hold accelerator-memory addresses**

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE; here **accelerator memory**, over BAR1: peer to peer

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free; **into host memory**: not the payload's sink

COMMAND COMPLETION PROCESSING (REAPER)

7. CPU spins on the phase tag, reads the 16-byte CQE, and validates it via CID

8. CPU rings the CQ head doorbell; the slot is free again

READ FLUSH

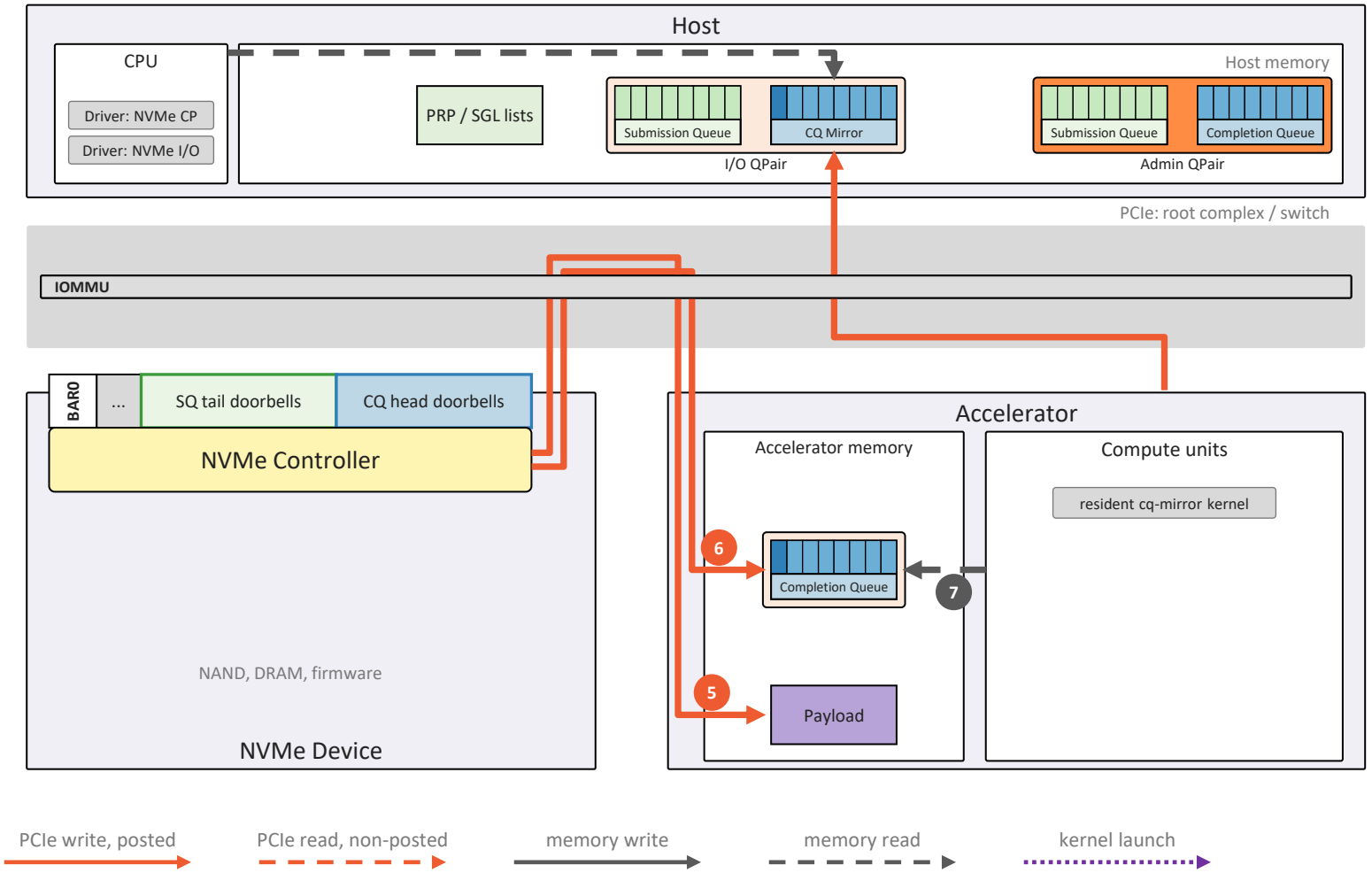
9. **Read-flush**: CPU reads one word of the payload back over PCIe; a read cannot overtake the posted writes ahead of it, so when it returns the payload has landed

The race.
CQE to host memory, payload to accelerator memory: two sinks. The CPU can see the completion before the data has landed. The read-flush closes it at one round trip; the mirror, next, avoids it.

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-assisted P2P NEW

The command points its completion beside the payload



I/O INITIATOR

1. CPU constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry

2. CPU rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE

4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to

5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE; here **accelerator memory**, over BAR1

6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free; **into accelerator memory**: the same sink as the payload

COMMAND COMPLETION PROCESSING (REAPER)

7. Split: **a resident kernel** spins on the accelerator-memory CQ and copies each CQE into the **host-memory mirror**; the CPU spins on the mirror, as it always did

8. CPU rings the CQ head doorbell; the slot is free again; the head it publishes only ever **lags** what the kernel has consumed, **never leads** it

CONSUMPTION (CONSUMER)

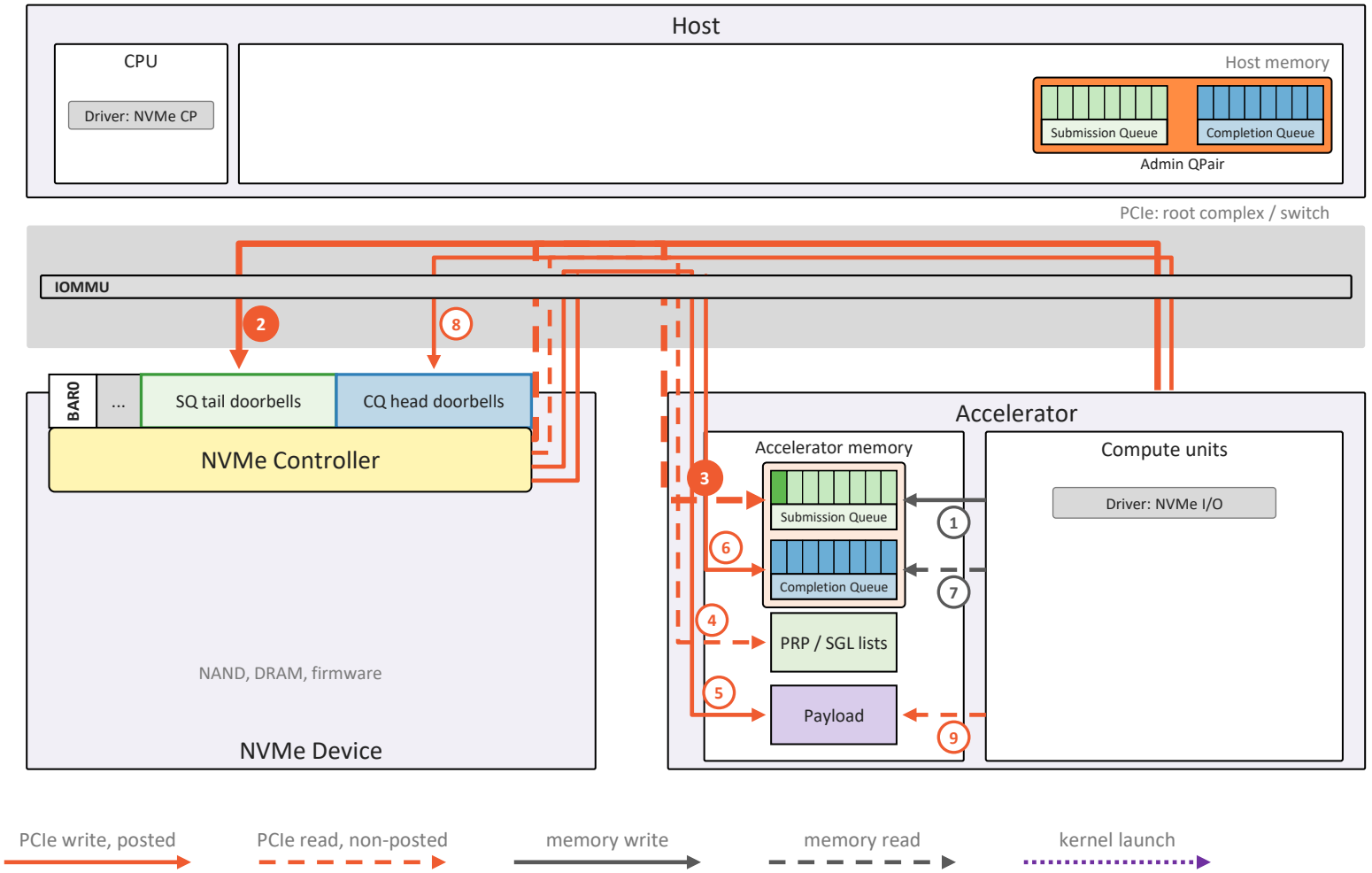
9. A kernel reads the payload **from accelerator memory**

Faster and safer.
The controller no longer holds each CQE behind cross-sink ordering; the CQE is observed on the accelerator, in the sink the payload landed in, before it is published to the host. The issue and reap loop is unchanged.

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-initiated P2P, aka GPU-initiated

The command moves to the accelerator entirely: issued, landed and reaped there



I/O INITIATOR

1. **ACC** constructs the Read command, its payload address, and PRP/SGL lists, then writes it into the SQ tail entry; **one thread per command**; SQ and buffer in **accelerator memory**
2. **ACC** rings the SQ tail doorbell: one 4-byte MMIO write into BAR0, after a write barrier; **thread 0, over P2P**

COMMAND PROCESSING (CONTROLLER)

3. NVMe controller fetches the SQE, 64 bytes, from wherever the queue lives, and advances its own SQ head: bookkeeping it keeps to itself, not signaled to the host until the CQE; here **from accelerator memory**, over BAR1: a P2P read
4. NVMe controller follows the data pointer, PRP or SGL as the SQE's PSDT field says, and fetches the list it points to
5. NVMe controller writes the data it read from media into the payload buffer, and constructs the CQE
6. NVMe controller posts the CQE, carrying status, CID, phase tag, and the SQ head, which tells the host the slot is free

COMMAND COMPLETION PROCESSING (REAPER)

7. **ACC** spins on the phase tag, reads the 16-byte CQE, and validates it via CID; **each thread on its own slot**
8. **ACC** rings the CQ head doorbell; the slot is free again; **thread 0, over P2P**; the phase flips on wrap

CONSUMPTION (CONSUMER)

9. The threads use the payload **in accelerator memory** and start the next round; **the CPU is out of the loop**

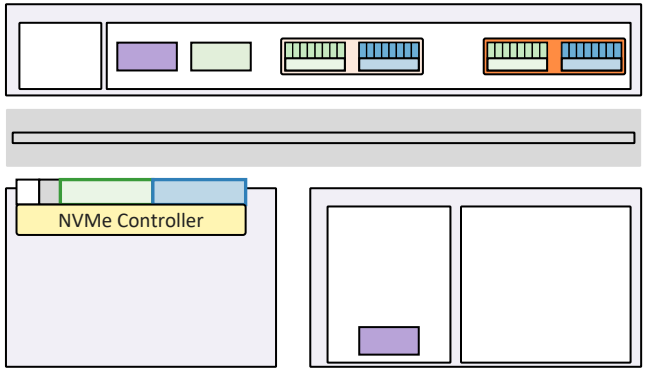
One sink, the accelerator in every role.
The controller writes only into accelerator memory. Doorbell writes are peer writes in the accelerator's IOMMU domain, so the controller's BAR must be mapped there. The polling kernel holds SMs for the run.

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O modes

The I/O modes at a glance and related work

CPU-initiated hostmem

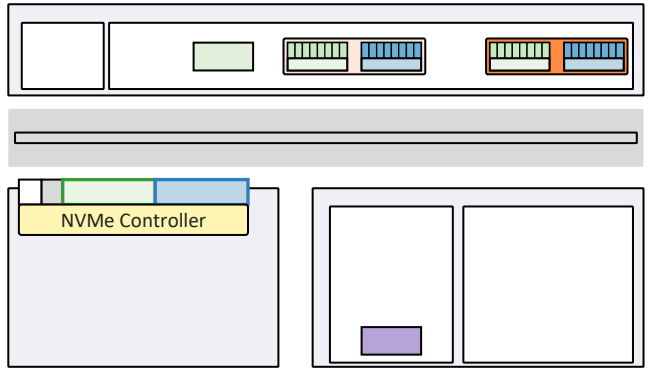


a copy added after the command

The conventional path: every framework's default, staged through host memory

AiSIO OSS: this talk

CPU-initiated P2P



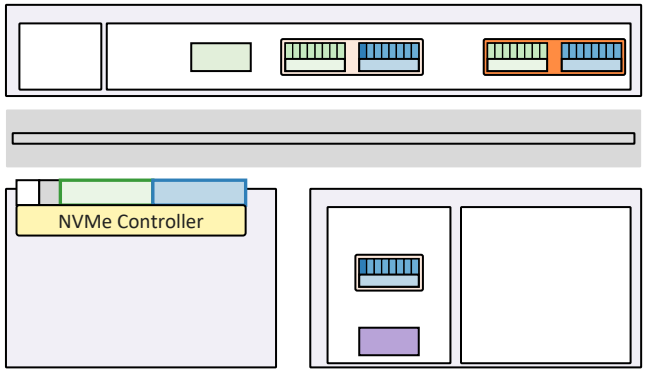
steps changed: payload on the accelerator

NVIDIA GDS: vendor-maintained, widely deployed

AiSIO PoC: built on BaM and libnvm

AiSIO OSS: this talk

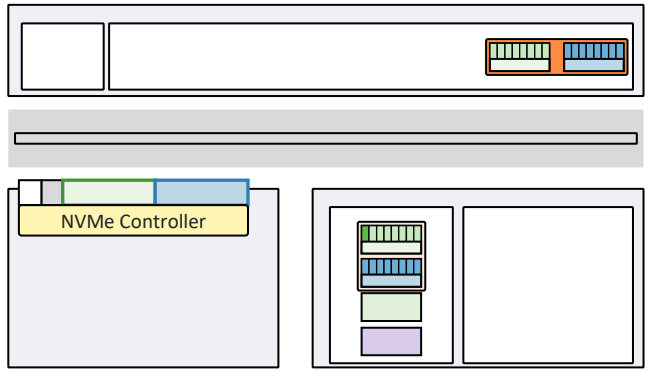
Accelerator-assisted P2P



steps changed: completion beside the payload

AiSIO OSS: this talk; the CQ mirror; nobody else does this yet

Accelerator-initiated P2P, aka GPU-initiated



steps changed: everything on the accelerator

BaM (NVIDIA Research): exclusive device ownership, out-of-tree kernel modules

NVIDIA SCADA: announced direction, not yet public

AMD ROCm-XIO: open source, early access (April 2026)

AiSIO PoC: built on BaM and libnvm

AiSIO OSS: this talk

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q1: Where did it start?



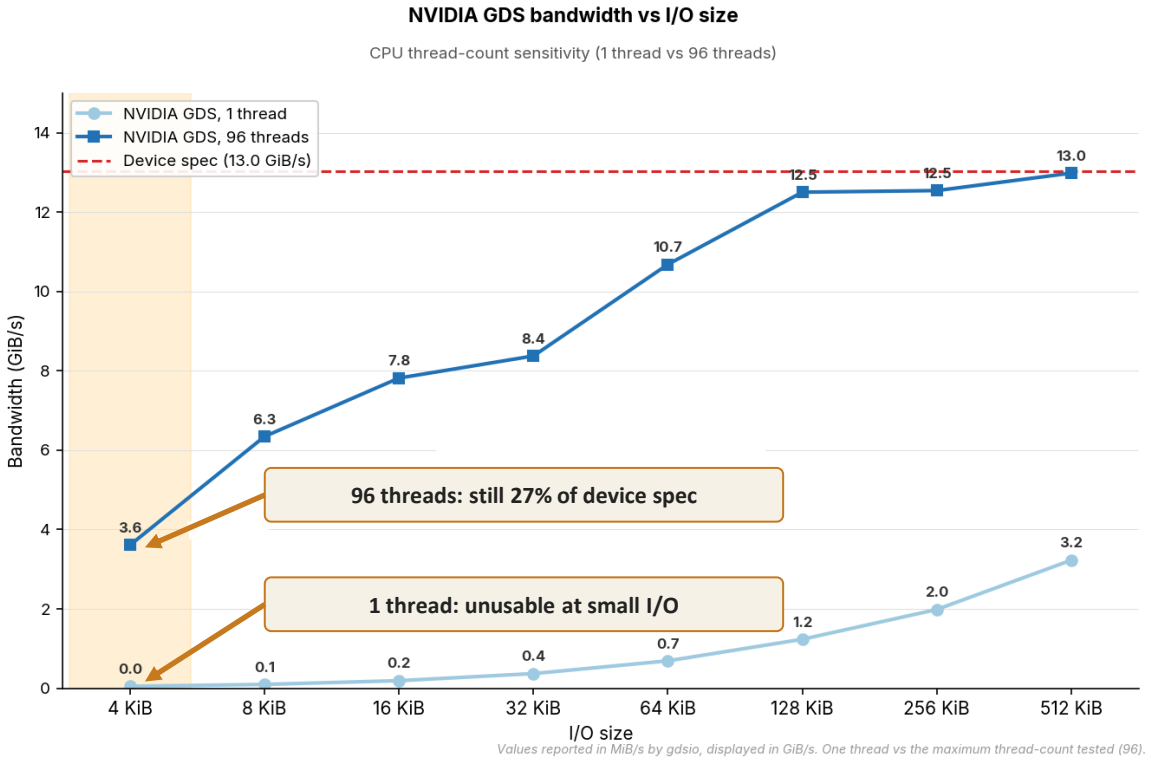
AiSIO PoC: identifying motivations and challenges

The tension: long-lived abstractions vs performance goals

- **GDS's** small-I/O throughput is poor
  - **Pro:** Maintains abstractions (**File I/O**)
  - **Con:** Expensive and bounded scalability
    - 96 CPU threads / 48 cores at 100% utilization to reach 3.6 GiB/s for 4 KiB I/O

- **Industry and academia** take a different path
  - **Drop:** Kernel protection, block devices, files and file systems
  - **Gain:** High IOPS for small I/O (512 bytes)

➔ **Small I/O is essential**



AiSIO's goal: performance without sacrificing the abstractions

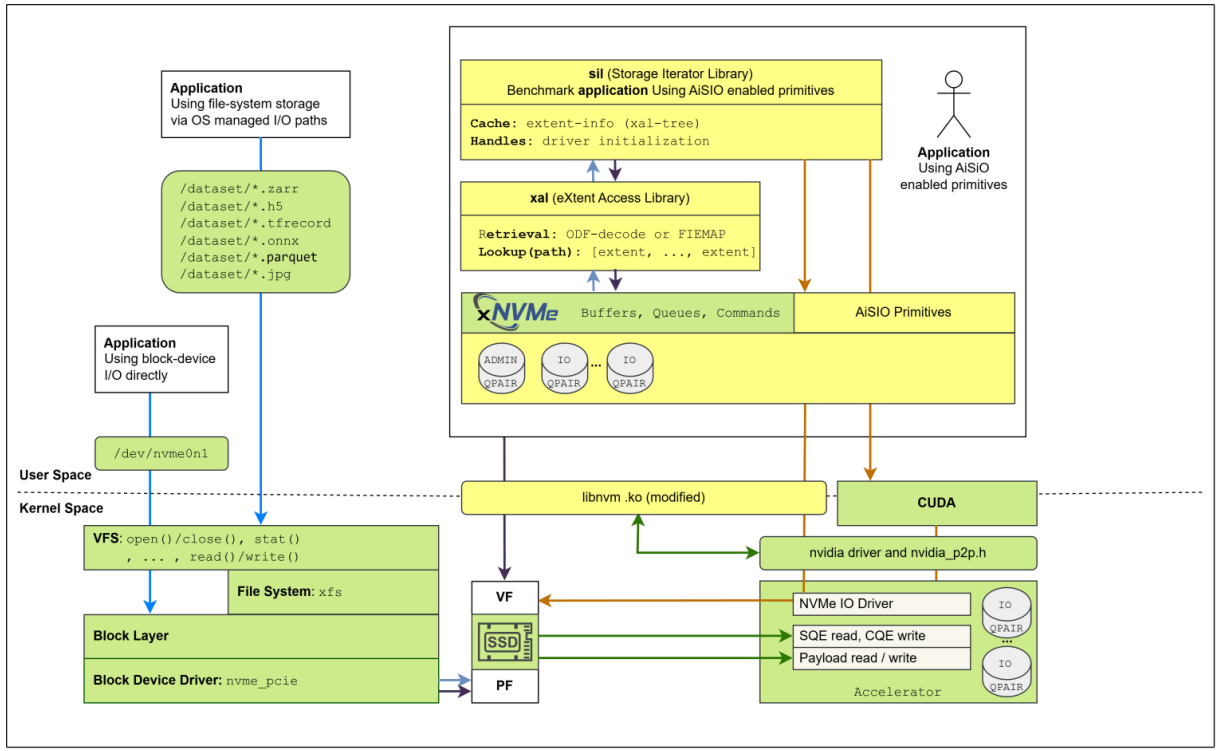
Get the throughput back without dropping the block layer or the file system

 ©2026 SNIA Developer Conference. All Rights Reserved.

AiSIO PoC: stack built on libnvm / BaM

Reproduced from BaM, extended with file-backed access and coexistence

- Built on **BaM** (paper | github), a fork of **libnvm** (paper | github) and prework described in paper
- Reproduced from **BaM**:
  - **GPU-initiated I/O / Accelerator-initiated I/O**:
    GPU drives storage commands directly, not the CPU
  - **Small-I/O performance**:
    Dramatically higher bandwidth and IOPS than GDS
- Extending **libnvm/BaM** with two contributions:
  - **Maintain Abstraction (File I/O)**:
    GPU performs I/O on the data backing files
  - **Coexistence/interoperability**:
    Conventional I/O path remains available, enabling applications and tools depending upon it

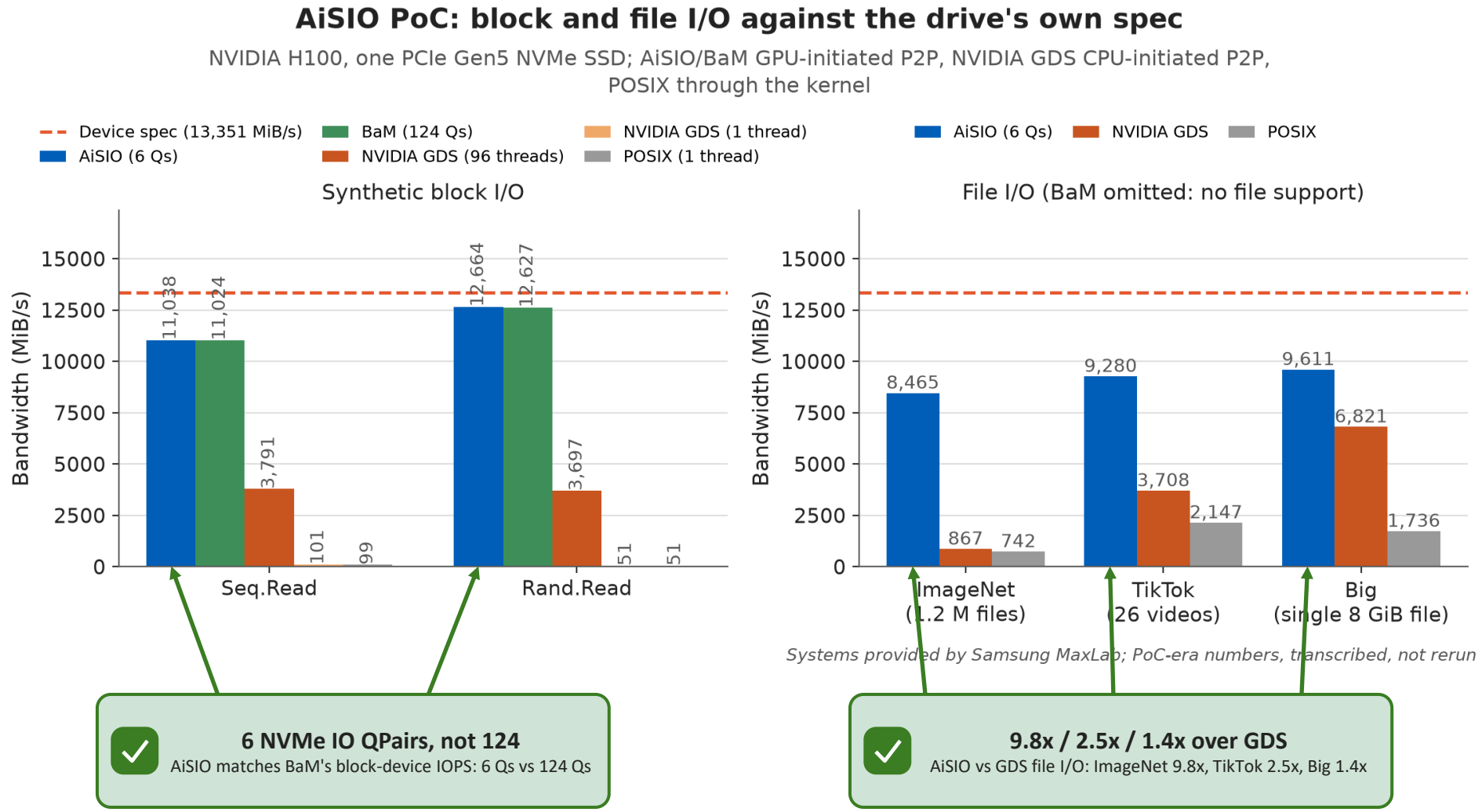


- Explored using NVIDIA V100/H100 and Samsung Gen5 SSDs
  - Presented at FMS'25, demoed at OCP'25 | Hardware courtesy of Samsung MaxLab

 ©2026 SNIA Developer Conference. All Rights Reserved.

AiSIO PoC: block and file I/O

one H100, one PCIe Gen5 SSD, 4 KiB I/O Size; PoC-era numbers from Samsung MaxLab



 ©2026 SNIA Developer Conference. All Rights Reserved.

AiSIO PoC: Limitations of the implementation

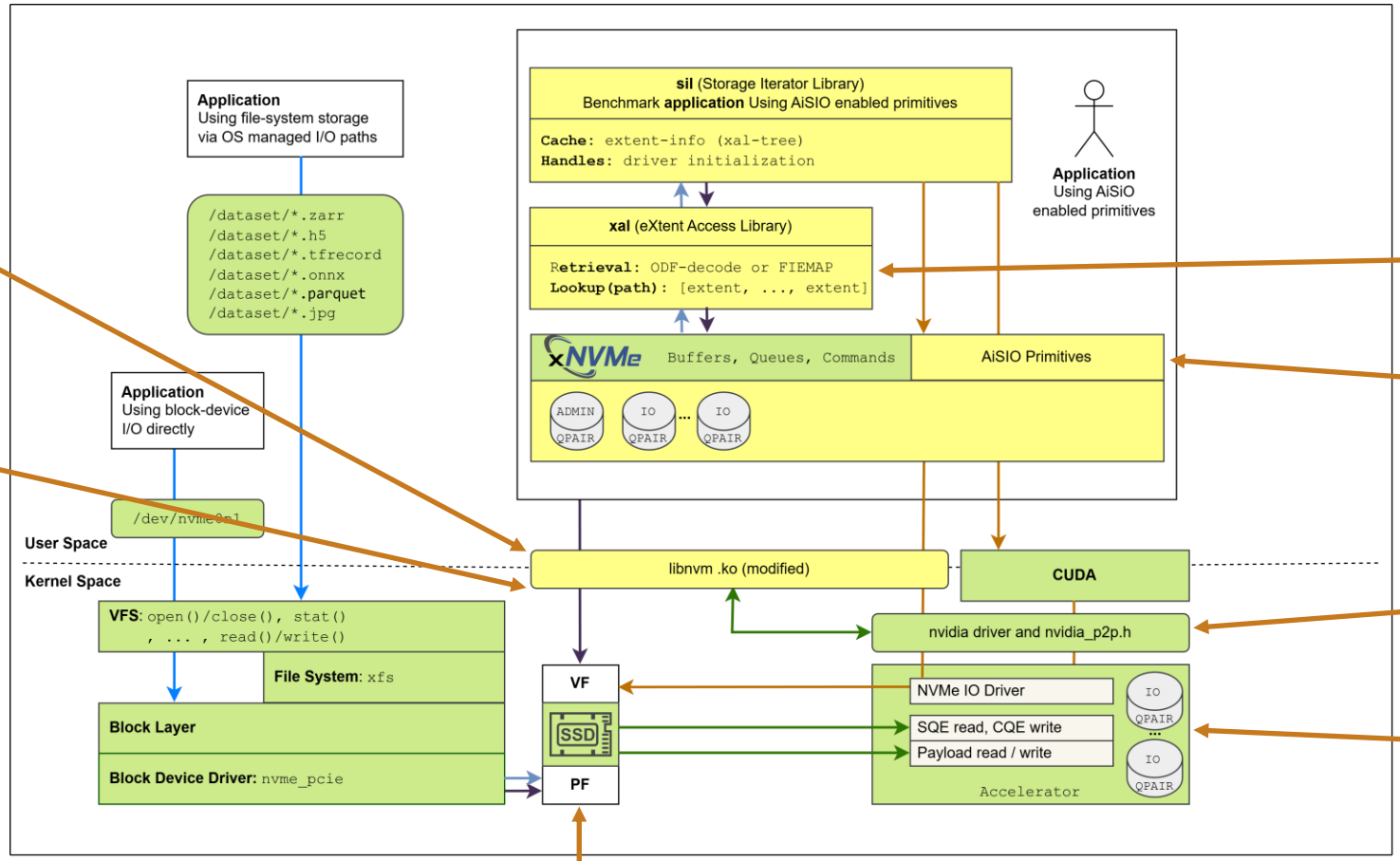
Proprietary P2P driver, one I/O mode, SR-IOV required, PA only and no IOMMU

libnvm.ko (modified)
Its own char-dev, not upstream infrastructure; deprecated and out-of-tree

Claims every NVMe PCI function; /dev/libnvmN: three ioctls (map host memory, map GPU memory via nvidia_p2p_get_pages, unmap) plus an mmap of BAR0.

No vfio-pci, dma-buf, or iommufd, and no co-existence e.g. no /dev/nvmeX
In other words, no file systems: no kernel infrastructure at all

Licensing, an open question
BSD tree, module declares "Dual BSD/GPL" to reach GPL-only symbols (class_create, device_create), built against nvidia.ko's Module.symvers



Extent cache
Efficient, but duplicated per process
Raw decode of the FS on-disk format

Only GPU-initiated I/O is wired up
CPU-initiated P2P and other modes are missing, narrowing use-cases and deployment options

Proprietary, single-vendor
P2P DMA driver

No IOMMU support!
Physical Addresses (PA) only

Requires SR-IOV on the NVMe device;
reduces deployment options

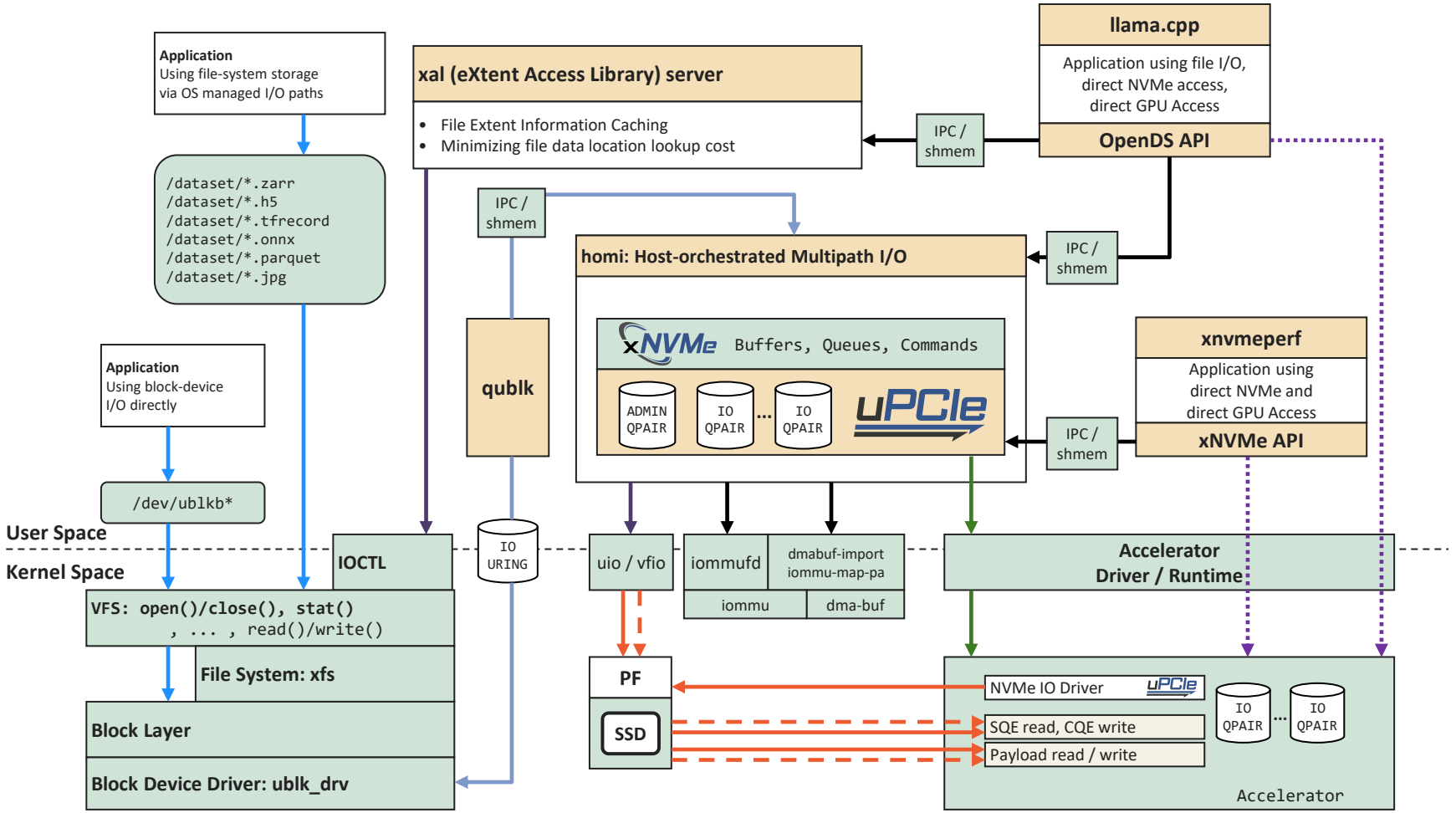
 ©2026 SNIA Developer Conference. All Rights Reserved.

Q2: What is it now?



AiSIO Architecture

Host-orchestrated multi-path I/O, block-layer enablement, and file-extent caching



Shared, with or without SR-IOV

homi opens the controllers through uio or vfio and hands out queue pairs over IPC and shared memory: to clients on the CPU, and to an accelerator

Three ways in for an application

Direct, through the OpenDS or xNVMe API; as files, through the OS with xal resolving extents so the I/O can still go direct; as a block device, through qublk and ublk_drv

Kernel primitives, nothing else

uio or vfio for the controller; iommufd and the dma-buf importer to place accelerator memory behind the IOMMU; ublk_drv for the block path

The accelerator is a peer

Its own queue pairs and NVMe I/O driver; doorbells to the PF and DMA from the SSD without the CPU on the path

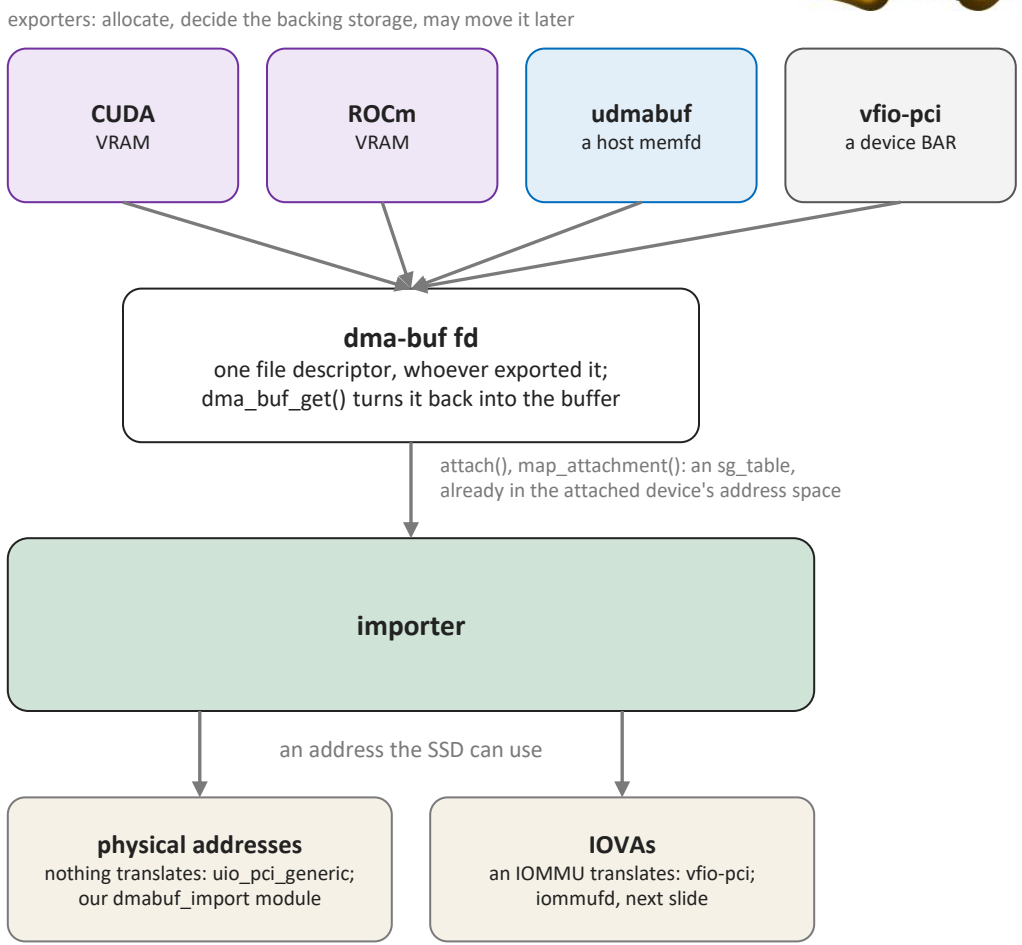
 ©2026 SNIA Developer Conference. All Rights Reserved.

dma-buf: buffer **sharing** and synchronization

Enables using buffers across subsystems, drivers, and user space



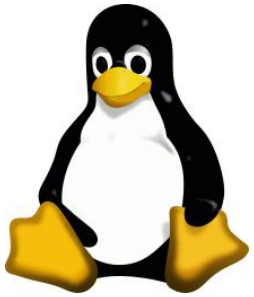
- Conventional **fd** handle
  - Produced by an **exporter**
  - Represent DMA'able memory
  - Including device-memory (non-pageable)
- dma-buf exporter/importer examples
  - **Exporter**: CUDA / ROCm / vfio-pci
  - **Importer**: iommufd
- AiSIO motivation for using dma-buf
  - Buffers suitable for DMA
  - Target host as well as device memory
    - Despite black-box drivers/runtimes!



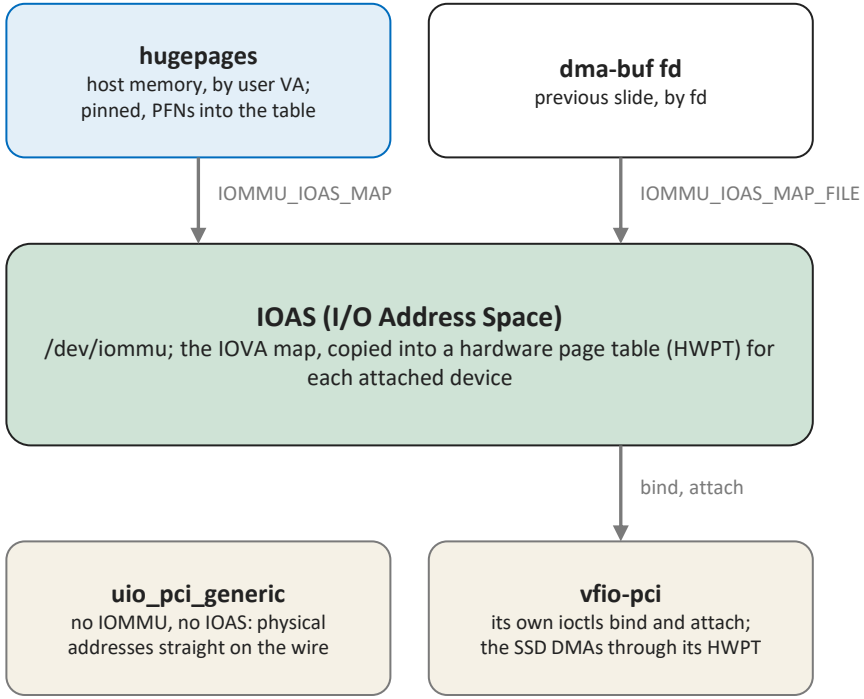
 ©2026 SNIA Developer Conference. All Rights Reserved.

iommufd

Managing IO page tables from user space using file descriptors



- Set up an **I/O address space** (IOAS)
  - MAP memory in → IOVAs out → attach a device
  - When the device-DMA targets the IOVAs, the IOMMU will translate and forward!
- **Mappers**
  - IOMMU_IOAS_MAP takes a user VA
  - **IOMMU_IOAS_MAP_FILE** (memfd / **dma-buf**)
- **AiSIO** leverages this for P2P memory
  - {cuda,hip}Malloc →
    {cu,hip}MemGetHandleForAddressRange(..TYPE_DMA_BUF_FD..) →
    IOMMU_IOAS_MAP_FILE
  - A user space NVMe driver can now construct NVMe commands targeting the just mapped device memory via the IOVAs



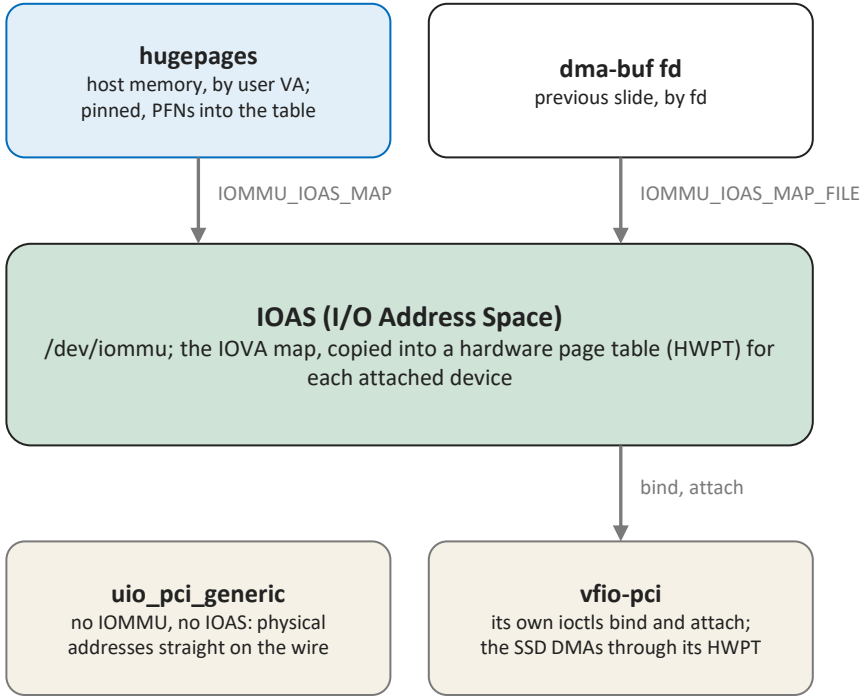
 ©2026 SNIA Developer Conference. All Rights Reserved.

iommufd

Managing IO page tables from user space using file descriptors



- Set up an **I/O address space** (IOAS)
  - MAP memory in → IOVAs out → attach a device
  - When the device-DMA targets the IOVAs, the IOMMU will translate and forward!
- **Mappers**
  - IOMMU_IOAS_MAP takes a user VA
  - **IOMMU_IOAS_MAP_FILE** (memfd / **dma-buf**)
- **AiSIO** leverages this for P2P memory
  - {cuda,hip}Malloc →
    {cu,hip}MemGetHandleForAddressRange(..TYPE_DMA_BUF_FD..) →
    IOMMU_IOAS_MAP_FILE
  - A user space NVMe driver can now construct NVMe commands targeting the just mapped device memory via the IOVAs



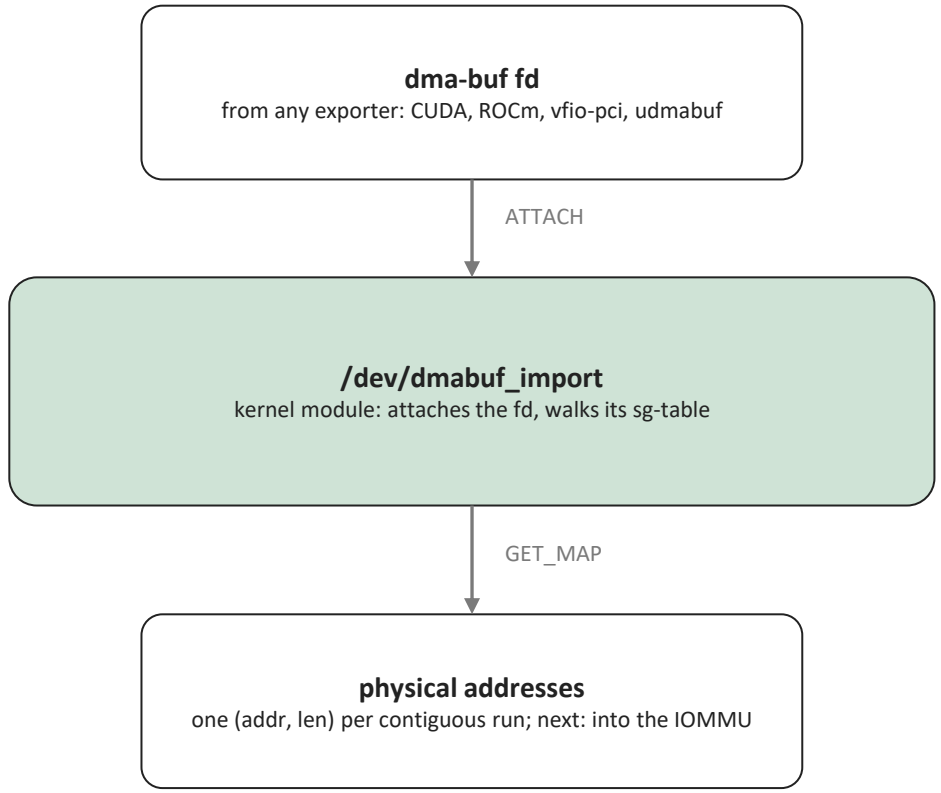
Currently **not** a generic dma-buf import path!
Only a single-range **vfio-pci** dma-buf gets in.
CUDA or **ROCm** export is refused, -EOPNOTSUPP

 ©2026 SNIA Developer Conference. All Rights Reserved.

dma-buf importer

Obtain the addresses of the memory backing a dma-buf **fd**

- Three IOCTLs on /dev/dmabuf_import
  - **ATTACH** / **GET_MAP** / **DETACH**
  - They do what they suggest, accessing the required infrastructure of the dma-buf mechanics from user space to import the given dma-buf using **ATTACH** and retrieve the underlying mapping/addresses with **GET_MAP**
  - Usable as-is with the **IOMMU** disabled, enabled needs one more step (iommu-map-pa on the next slide)
- Out-of-tree module shipped as DKMS
  - Out-of-tree until **IOMMU_IOAS_MAP_FILE** accepts more than vfio-pci exporters
- **Source:** safl/upcie

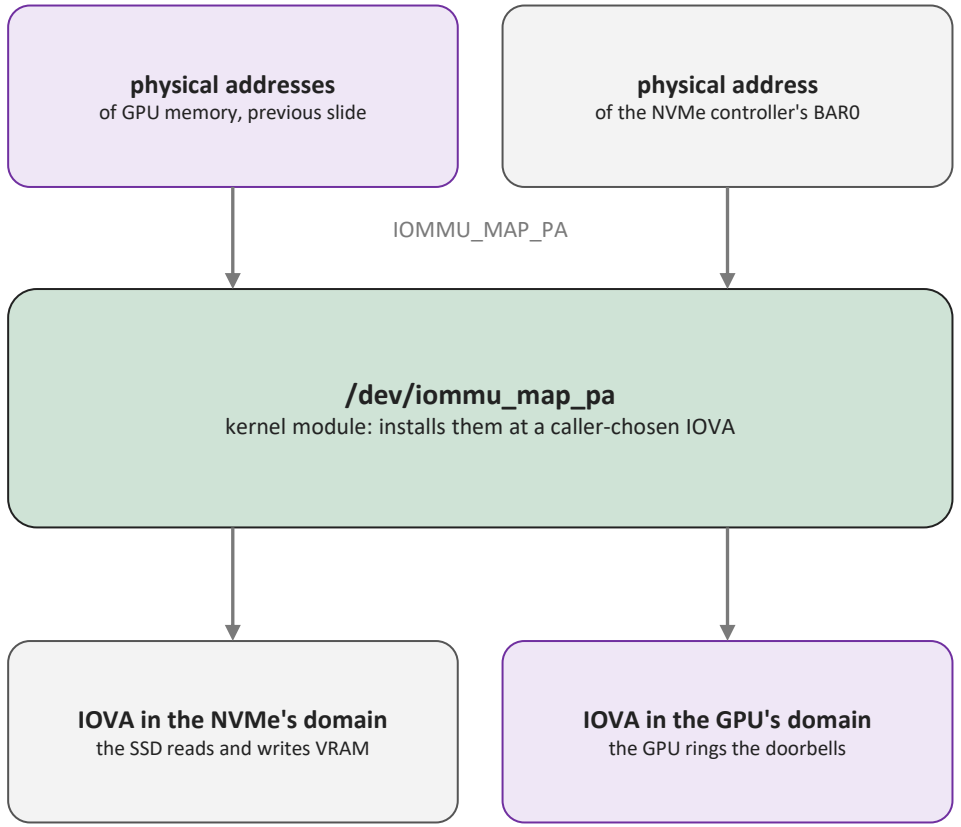


 ©2026 SNIA Developer Conference. All Rights Reserved.

iommu-map-pa

Physical address in, IOVA in the device's domain out

- Two IOCTLs on /dev/iommu_map_pa
  - **IOMMU_MAP_PA** / **IOMMU_UNMAP_PA**
  - They do what they suggest, accessing the required infrastructure of the iommufd mechanics from user space to import the given addresses.
  - This works in combination with the dma-buf import **GET_MAP**
- Out-of-tree module shipped as DKMS
  - Out-of-tree until **IOMMU_IOAS_MAP_FILE** accepts more than vfio-pci exporters
- **Source:** safl/upcie



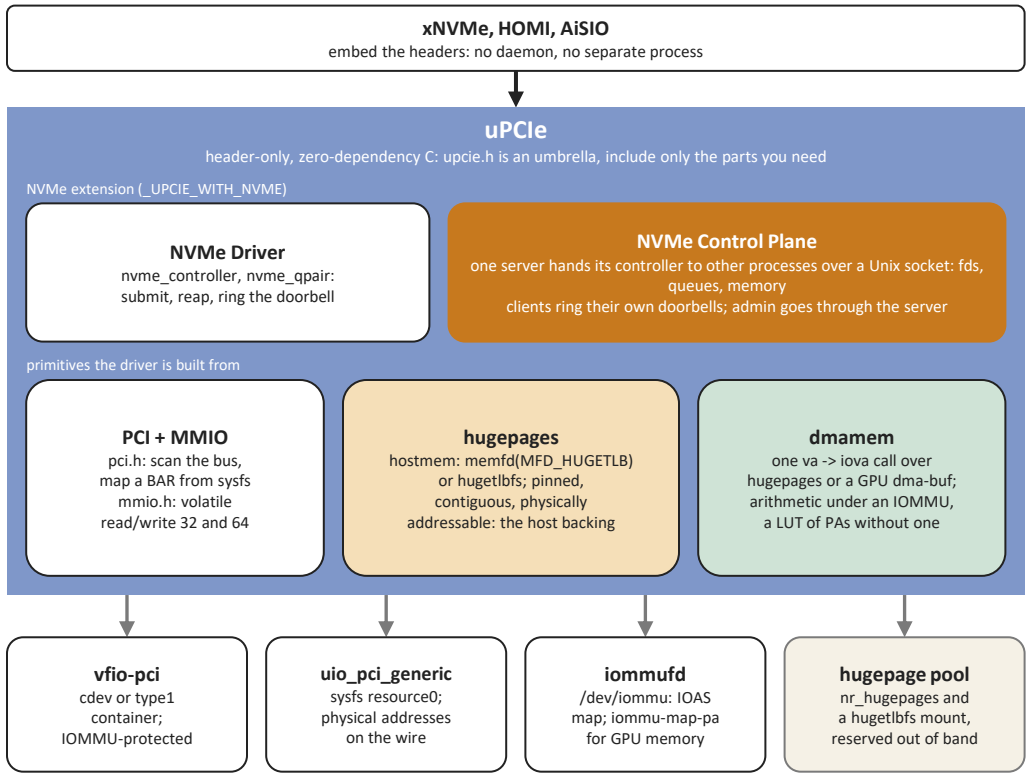
 ©2026 SNIA Developer Conference. All Rights Reserved.

uPCIe

User space PCIe primitives for efficient device drivers

- Primitives for common operations
  - MMIO, DMA, MEMORY
- Wrappers for system interfaces
  - iommufd, vfio-pci, uio_pci_generic, hugepages
- Header-only
  - Embedded in the caller, no separate process
- NVMe Driver and Control Plane
  - PCIe transport
  - SW-fallback for lack of SR-IOV
  - IOMMU setup for payloads, PRPs, etc.

- **Source:** safl/upcie

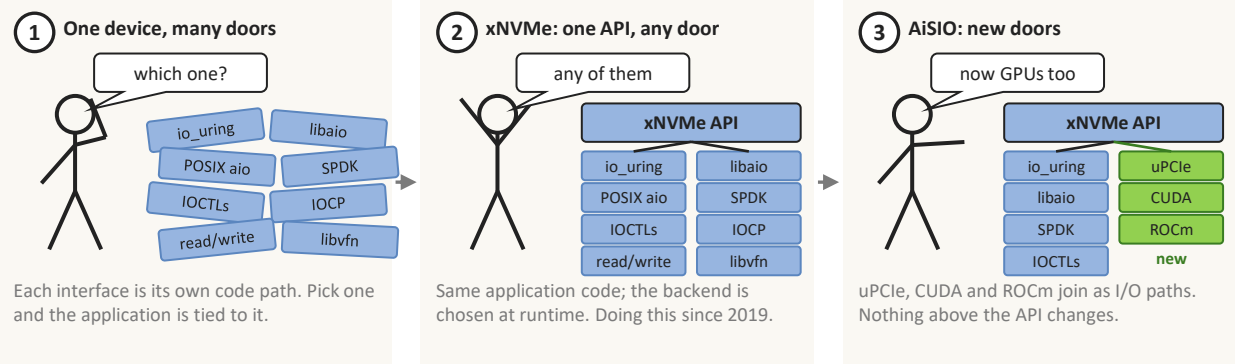


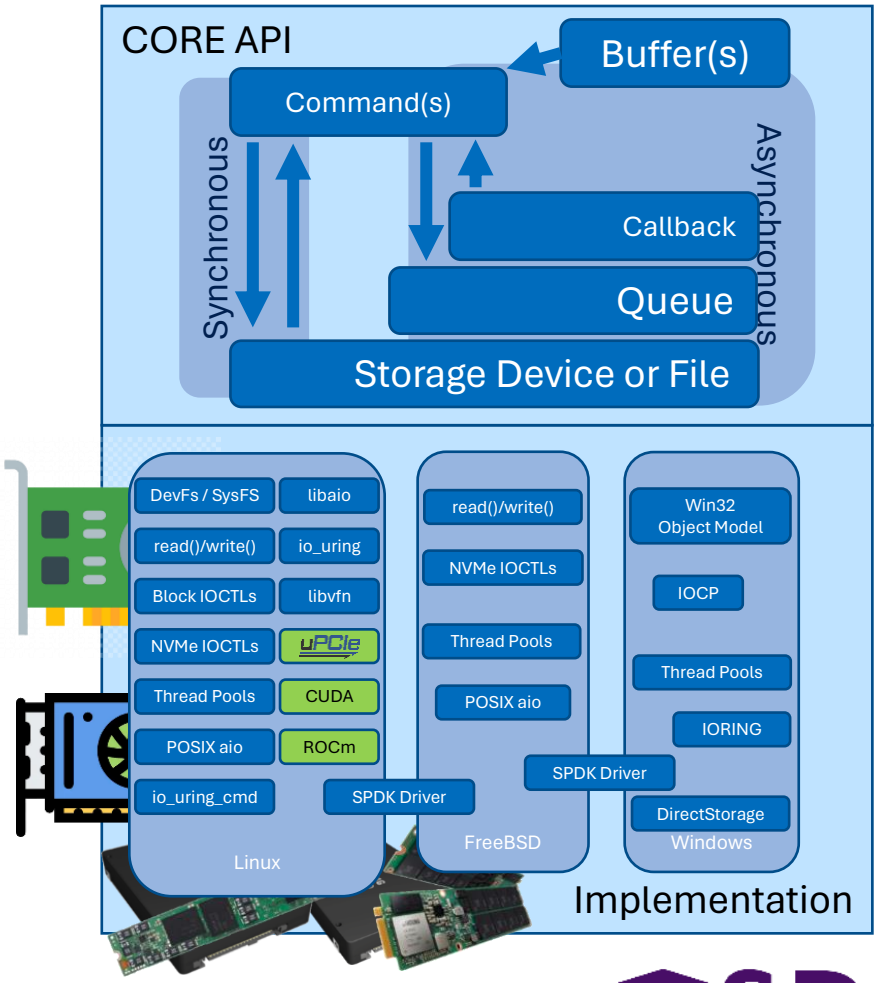
kernel interfaces it wraps directly, no daemon in between; the pool is the one setup step outside the library (the hugepages CLI, or echo N > /proc/sys/vm/nr_hugepages)

 ©2026 SNIA Developer Conference. All Rights Reserved.

Unified Storage I/O APIs and Tools

Minimal overhead, maximum flexibility: any backend, chosen at runtime





- Expanded with **AiSIO** primitives (API)
- Expanded with **AiSIO** I/O paths (driver/runtime)
- Expanded with **AiSIO** tools (HOMI, xnvmeperf, qublk)

- **Docs:** xnvme.io
- **Source:** xnvme/xnvme

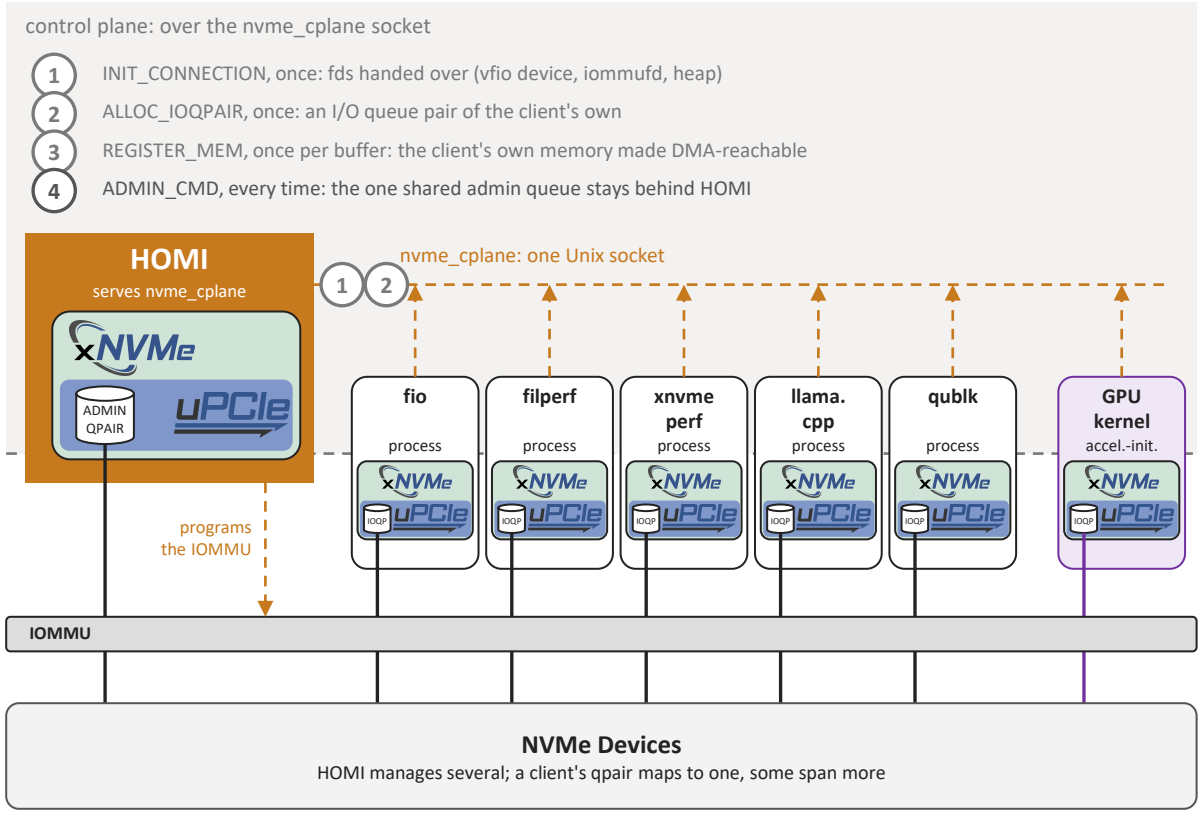
 ©2026 SNIA Developer Conference. All Rights Reserved.

HOMI

Host-orchestrated Multi-path I/O, as a system daemon

- **Host-resident daemon (homi)**
- Centralizes **control-plane** responsibilities
  - Controller bring up and lifetime
  - Admin Command Processing
  - I/O queue-pair provisioning
  - IOMMU setup
- The enabler of **NVMe controller sharing** without relying on SR-IOV

- **Docs:** xnvme.io/tools/homi
- **Source:** xnvme/xnvme/tools/homi



After 1 and 2 the client rings its own doorbell; I/O never crosses the socket again. Admin commands are the one thing that always does.

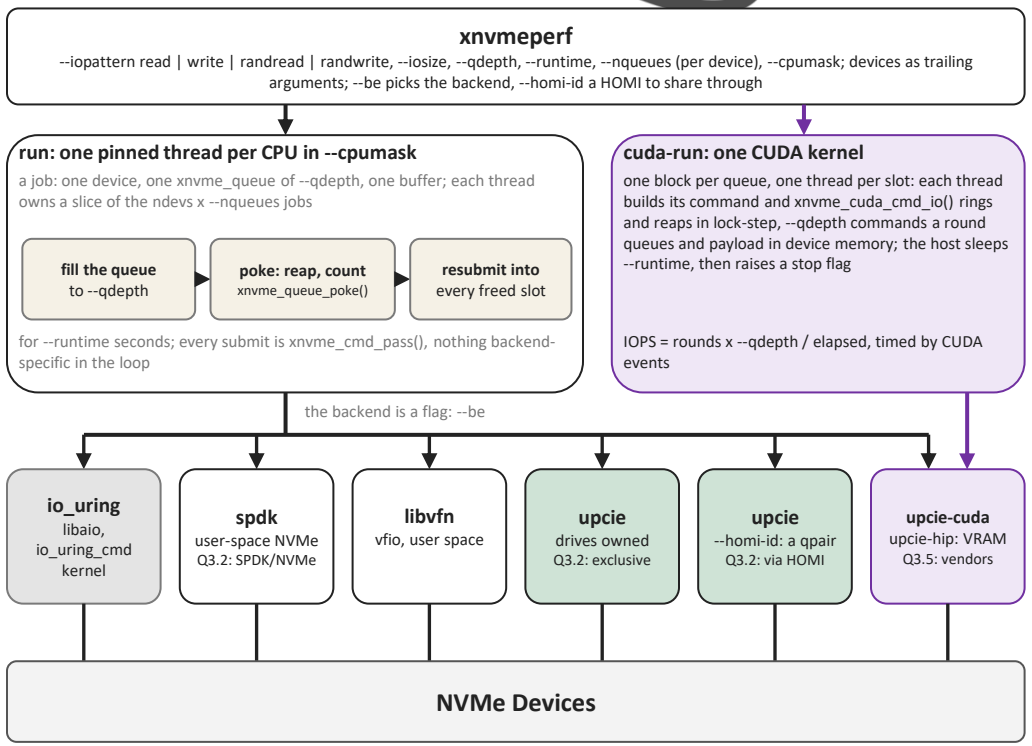
 ©2026 SNIA Developer Conference. All Rights Reserved.

xnvmeperf

Synthetic workloads, on anything xNVMe runs on

- Synthetic workload generation
  - Minimal overhead
  - For kernel-managed block devices
  - NVMe devices (NVM Command Set)
  - Anything that xNVMe runs on!
- Execution
  - **run / verify:** CPU-initiated, one pinned thread per CPU
  - **cuda-run / cuda-verify:** accelerator-initiated, GPU threads submit and reap

- **Docs:** xnvme.io/tools/xnvmeperf
- **Source:** xnvme/xnvme/tools/xnvmeperf

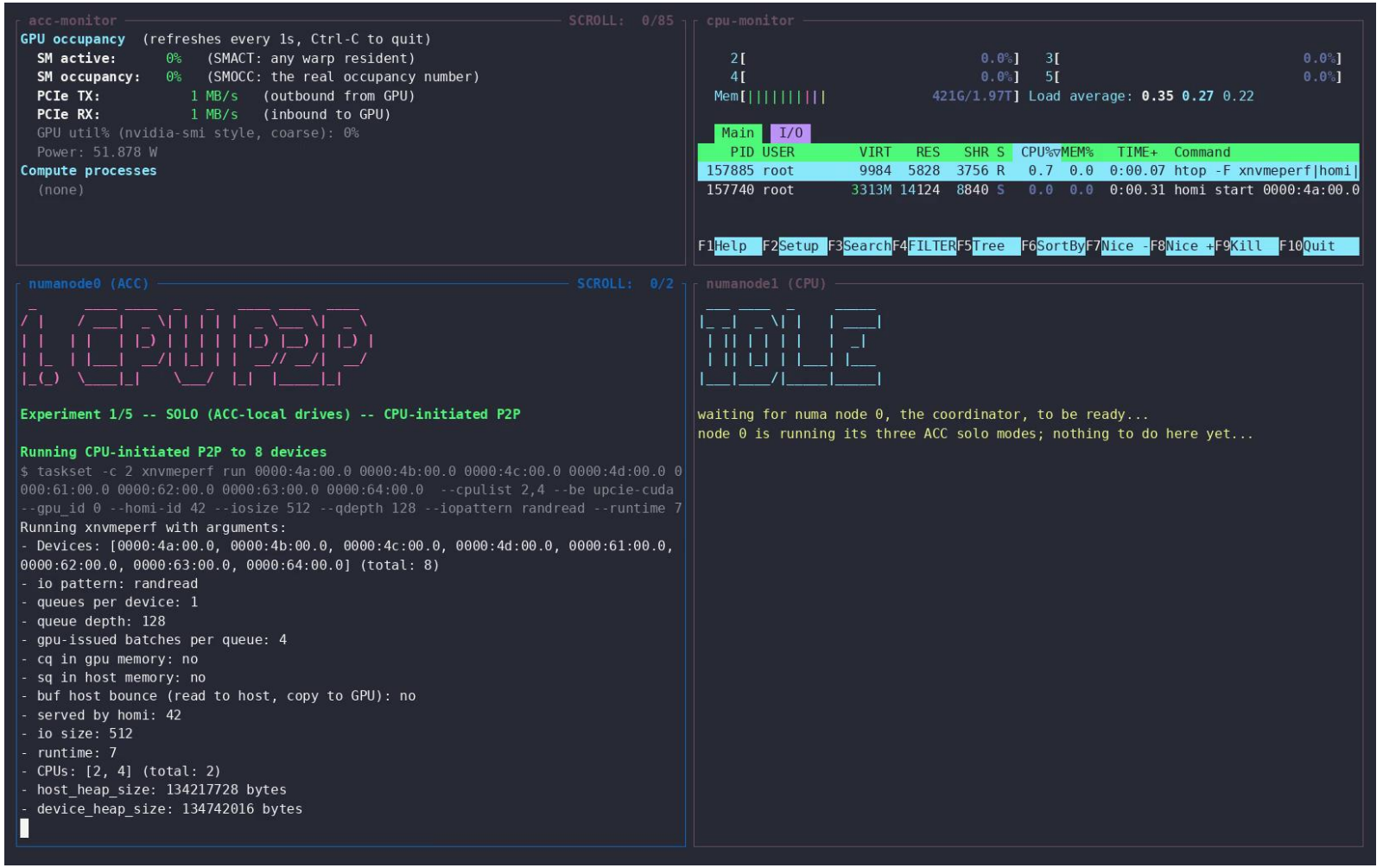


One table per run, per device and total: IOPS, MiB/s, failed, and which CPUs drove it. Q3.2's one-core-peak ladder is this tool with three values of --be; the two-vendor comparison swaps only --be between upcie-cuda and upcie-hip. With run, a GPU backend still means the CPU submits, the payload lands in VRAM: CPU-initiated P2P in the mode vocabulary; cuda-run is the accelerator-initiated one.

 ©2026 SNIA Developer Conference. All Rights Reserved.

xnvmeperf (demo)

Benchmarking the Accelerator-integrated Storage I/O modes



- **1. CPU-initiated P2P:** 8 devices
- **2. Accelerator-assisted P2P:** 8 devices
- **3. Accelerator-initiated P2P:** 8 devices
- **4. Duet:** Accelerator-initiated P2P and CPU-initiated hostmem, at once
- **5. Finale:** CPU-initiated hostmem, across all 16 drives

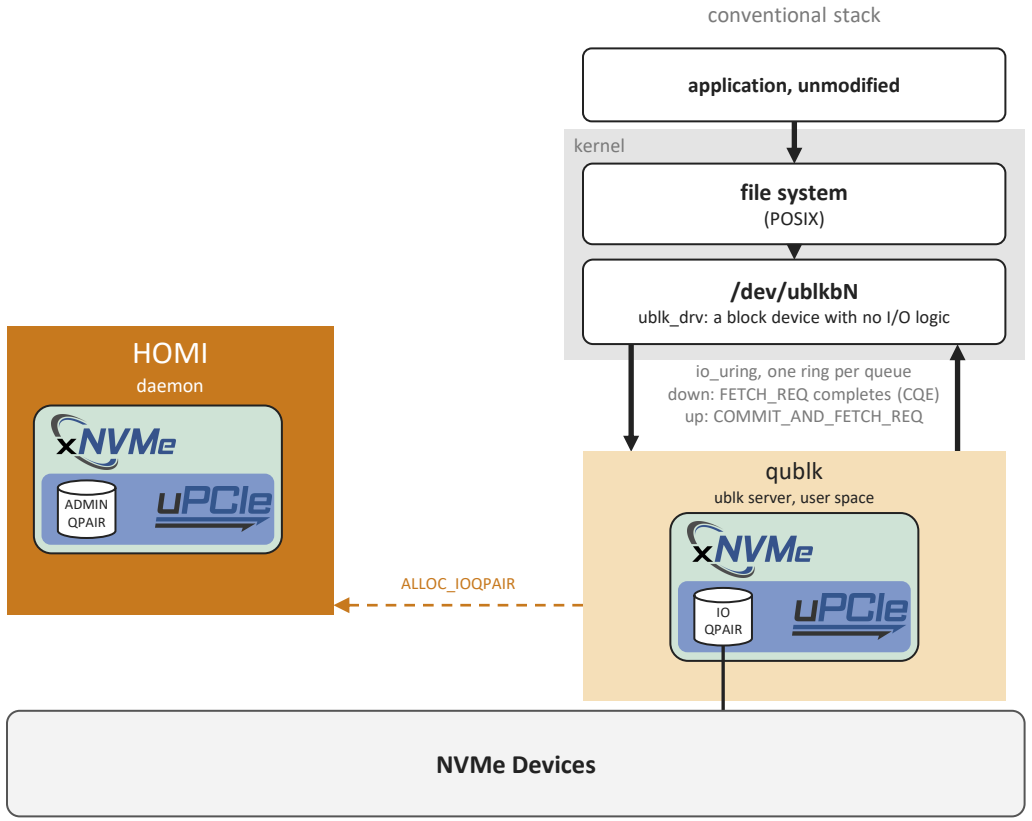
Recording:
asciinema.org/a/Aky5amiptKPfDqSH

 ©2026 SNIA Developer Conference. All Rights Reserved.

qublk

A ublk server on xNVMe: /dev/ublkbN, over io_uring

- Re-introduces the block device
  - A user-space NVMe driver has no kernel block device; qublk provides one, /dev/ublkbN
  - Mountable file systems, and everything else built on that abstraction, come back with it
- ublk server backed by xNVMe
  - /dev/ublkbN is a kernel block device
  - Any xNVMe backend, any NVMe device
  - uPCIe through HOMI; coexists with AiSIO paths, no SR-IOV
- **Docs:** xnvme.io/tools/qublk
- **Source:** xnvme/xnvme/tools/qublk



The same device backs /dev/ublkbN: requests handed to qublk and answered over one io_uring per queue, no data through HOMI.

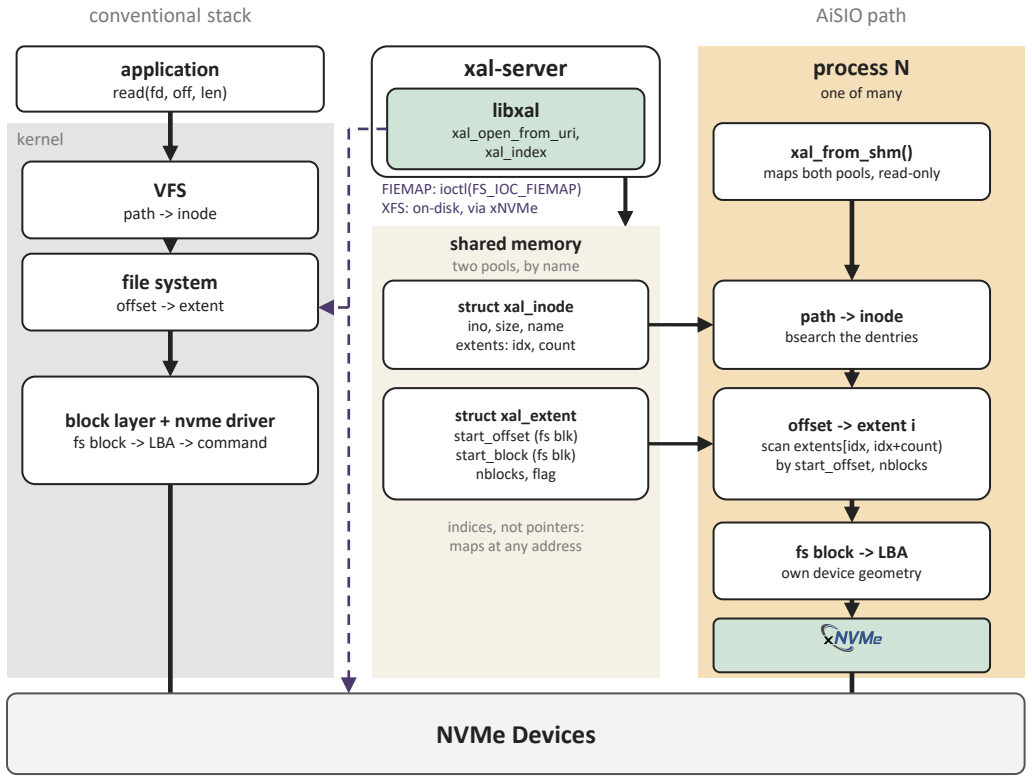
 ©2026 SNIA Developer Conference. All Rights Reserved.

eXtent Access Library (XAL)

Encapsulate and cache retrieval of file extent information

- Extent-info → struct xal_extent: start_offset (in the file), start_block (on the device), nblocks (length), flag (e.g. unwritten)
  - Mapping file's logical offset to its location on the device
  - file path -> inode -> extents -> device LBAs
- Library Backends
  - Watch-modes for file-extent changes
  - Extent retrieval
    - From the OS via FIEMAP (IOCTL)
    - Raw file-system on-disk-format-decode, no mount (done for XFS)
    - Only applicable to unmounted file-system
- The XAL Server (extent-info caching)
  - One process keeps the index (xal-server)
  - Clients access the index via shared memory → no syscall per lookup
- Linux upstream efforts
  - File-system communication for file-leases e.g. pin the mapping
  - Cross file-system portable extent-format and user-space export

- **Source:** xnvme/xal



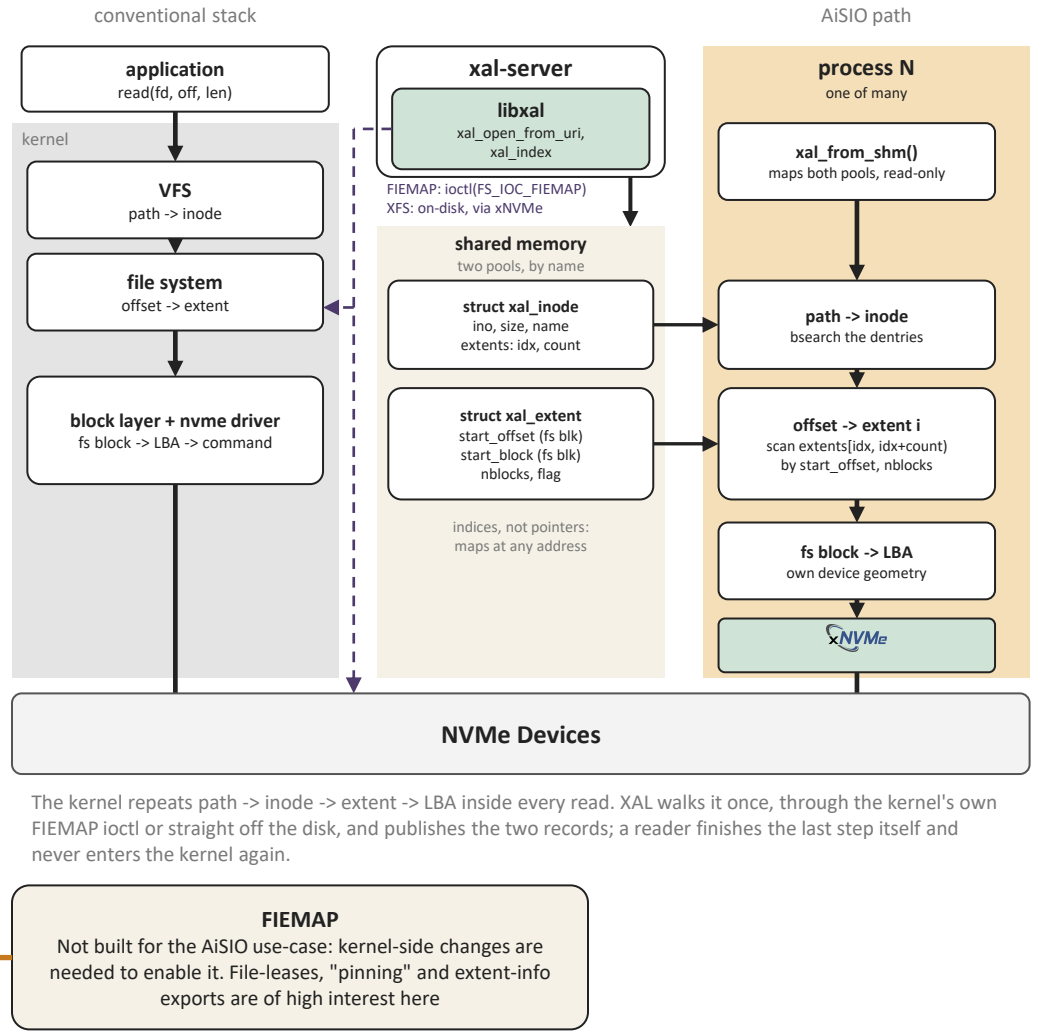
The kernel repeats path -> inode -> extent -> LBA inside every read. XAL walks it once, through the kernel's own FIEMAP ioctl or straight off the disk, and publishes the two records; a reader finishes the last step itself and never enters the kernel again.

 ©2026 SNIA Developer Conference. All Rights Reserved.

eXtent Access Library (XAL)

Encapsulate and cache retrieval of file extent information

- Extent-info → struct xal_extent: start_offset (in the file), start_block (on the device), nblocks (length), flag (e.g. unwritten)
  - Mapping file's logical offset to its location on the device
  - file path -> inode -> extents -> device LBAs
- Library Backends
  - Watch-modes for file-extent changes
  - Extent retrieval
    - From the OS via FIEMAP (IOCTL)
    - Raw file-system on-disk-format-decode, no mount (done for XFS)
    - Only applicable to unmounted file-system
- The XAL Server (extent-info caching)
  - One process keeps the index (xal-server)
  - Clients access the index via shared memory → no syscall per lookup
- Linux upstream efforts
  - File-system communication for file-leases e.g. pin the mapping
  - Cross file-system portable extent-format and user-space export
- **Source:** xnvme/xal



The kernel repeats path -> inode -> extent -> LBA inside every read. XAL walks it once, through the kernel's own FIEMAP ioctl or straight off the disk, and publishes the two records; a reader finishes the last step itself and never enters the kernel again.

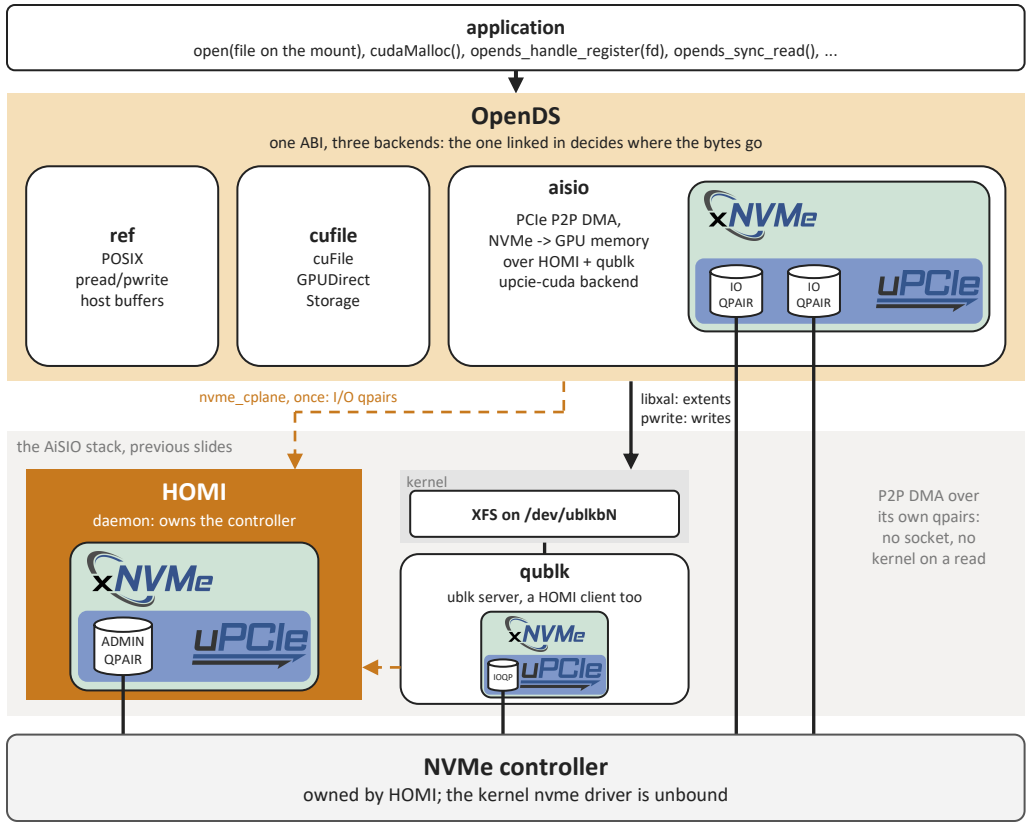
FIEMAP
Not built for the AiSIO use-case: kernel-side changes are needed to enable it. File-leases, "pinning" and extent-info exports are of high interest here

 ©2026 SNIA Developer Conference. All Rights Reserved.

Open Direct Storage (OpenDS)

Vendor-neutral File and Block APIs, cuFile and beyond

- Open Direct Storage
  - cuFile compatible API for adoption of legacy applications
  - Efficient true ASYNC APIs to move beyond
  - Vendor-neutral
- One ABI, three backends
  - POSIX (ref), NVIDIA cuFile (cufile), xNVMe + HOMI + qublk (aisio)
- **Source:** xnvme/opends



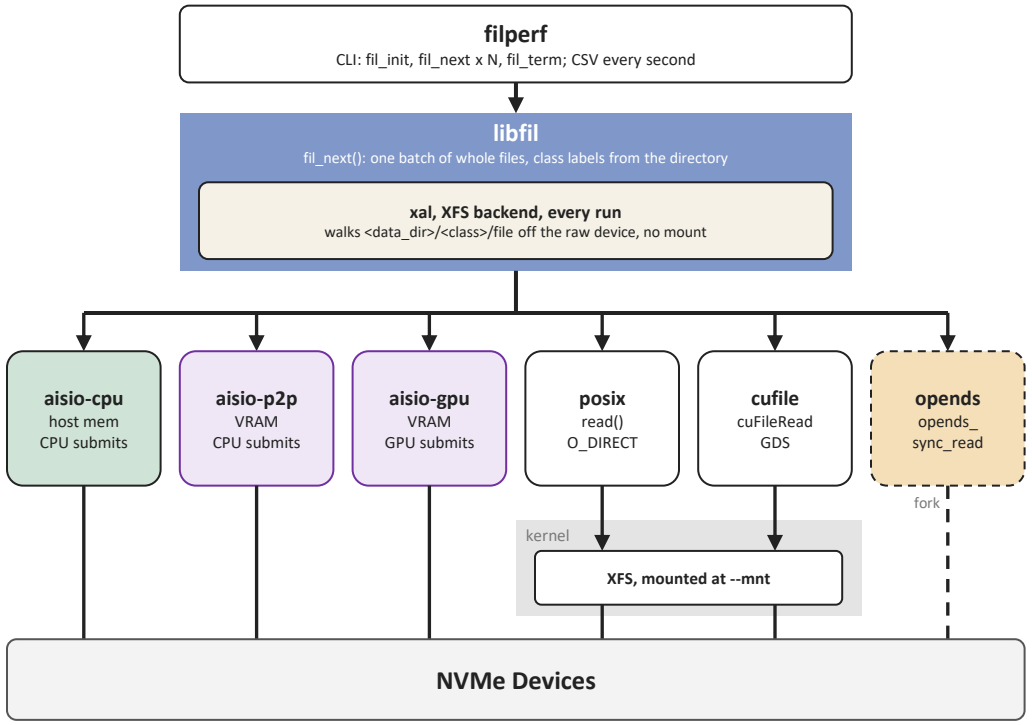
With the aisio backend every byte crosses xNVMe: reads over OpenDS's own qpairs, straight to GPU memory; writes through XFS on qublk, which is xNVMe underneath. ref and cufile go through the kernel's own file system and nvme driver instead, outside this stack.

 ©2026 SNIA Developer Conference. All Rights Reserved.

File Iterator Library (fil) and perf tool (filperf)

Run a synthetic file-loader workload for interface comparison

- Library: libfil
  - Library interfaces to the synthetic file workload
  - Richer workload instrumentation
- Library: backends
  - Who submits, where bytes land
  - Enables relative comparison of interfaces
- Tool: filperf
  - Command-line tool
  - A timing loop around libfil
  - fil_init(), N x fil_next(), fil_term()
  - CSV every second: batches, IOPS, MiB/s
- **Source:** xnvme/fil

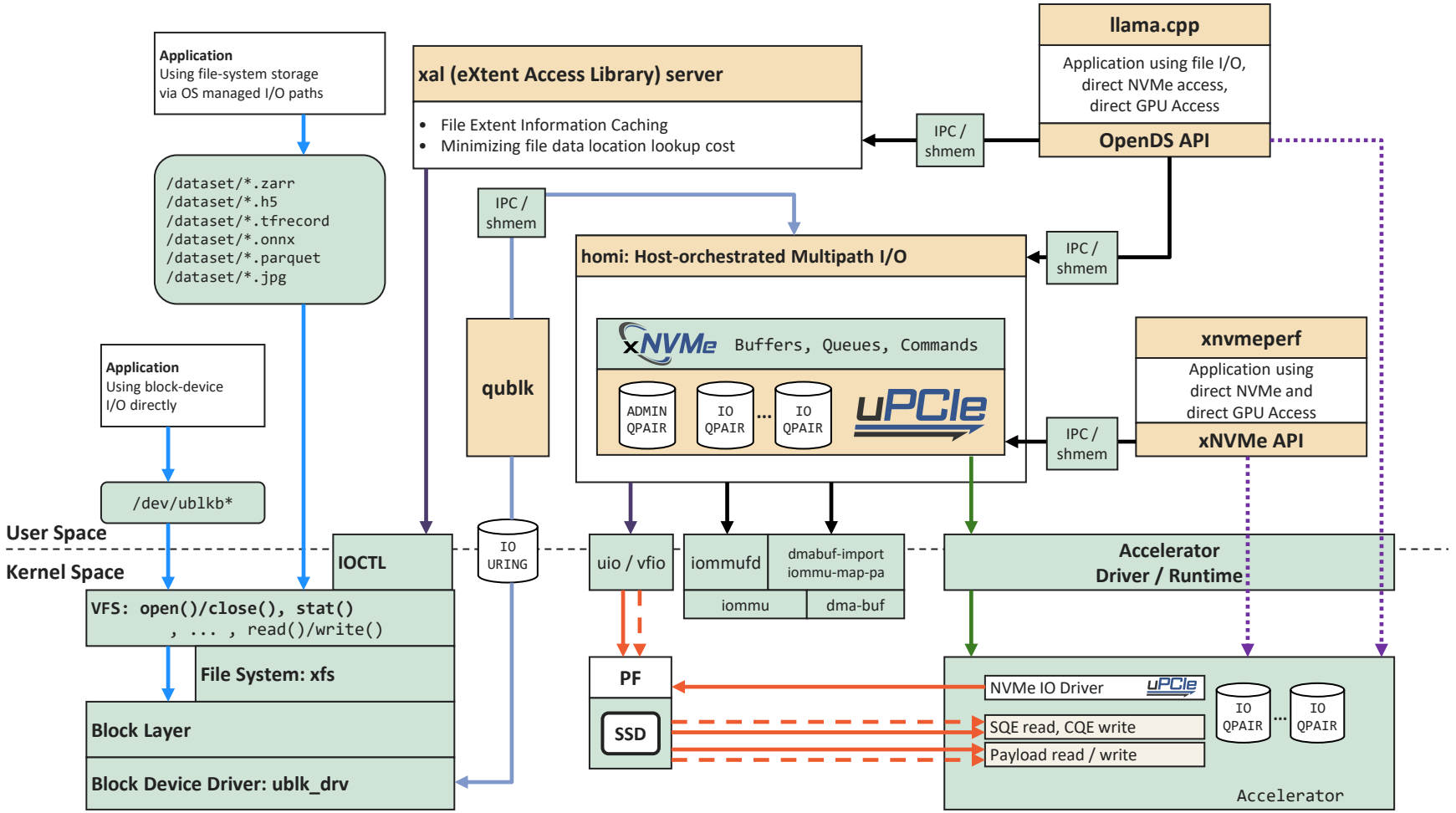


One xal walk names the files for every backend. opends is the fork OpenDS pins (fmma/fil, opends-0.1.2): it opens each file on the qublk mount and reads it through OpenDS's aisio backend, HOMI's qpair, previous slide.

 ©2026 SNIA Developer Conference. All Rights Reserved.

AiSIO Architecture

Host-orchestrated multi-path I/O, block-layer enablement, and file-extent caching



Shared, with or without SR-IOV

homi opens the controllers through uio or vfio and hands out queue pairs over IPC and shared memory: to clients on the CPU, and to an accelerator

Three ways in for an application

Direct, through the OpenDS or xNVMe API; as files, through the OS with xal resolving extents so the I/O can still go direct; as a block device, through qublk and ublk_drv

Kernel primitives, nothing else

uio or vfio for the controller; iommufd and the dma-buf importer to place accelerator memory behind the IOMMU; ublk_drv for the block path

The accelerator is a peer

Its own queue pairs and NVMe I/O driver; doorbells to the PF and DMA from the SSD without the CPU on the path

 ©2026 SNIA Developer Conference. All Rights Reserved.

Reproducibility

AiSIO as a research platform: bare metal to results



- **AiSIO** is a reproducible research platform
  - **Technical report:** xnvme.io/aisio
  - **SDK:** github.com/xnvme/aisio
  - Open source, automated, deterministic
- From bare-metal setup to results in version-controlled **CIJOE** workflows
  - **System setup and configuration**
    - OS and kernel modules, accelerator vendor stack provisioning (e.g. NVIDIA/CUDA and AMD/ROCm)
    - xNVMe/uPCIe/HOMI/qublk/xal/filperf, SPDK, and fio built from source
    - Datasets populated for the benchmark workflows
  - **Experiments, validation, and verification**
    - CPU-initiated baseline: what does the hardware allow?
    - Minimizing software overhead
    - Reaching the accelerator: CPU-initiated and accelerator-initiated P2P
    - P2P bandwidth scaling across NVMe devices
    - Measured through HOMI behind a translating IOMMU

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3: How does it behave and perform?

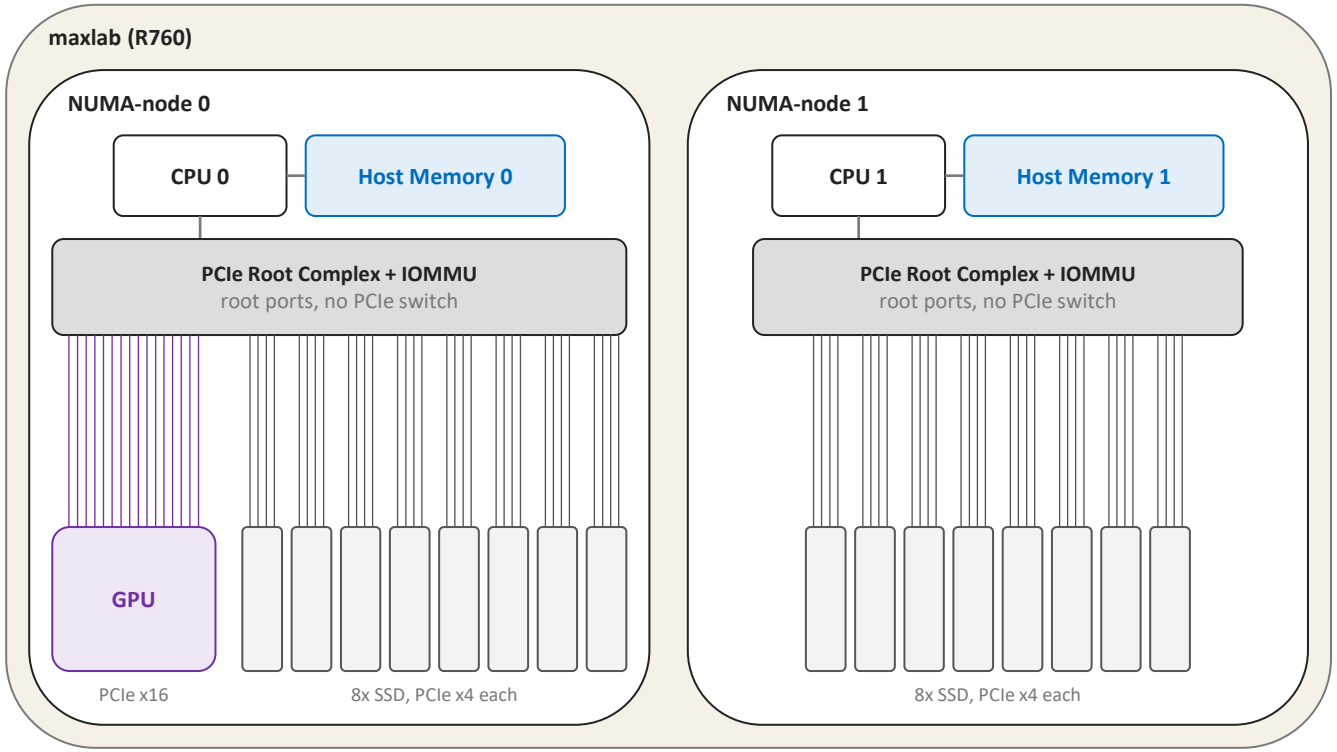


SDC 26

Q3.0 The hardware and software utilized for evaluation

Every number presented through Q3.1-Q3.5, and in Extra, is measured on this system

	maxlab (R760)
CPU	2x Intel Xeon Gold 6442Y
Drives	16x NVMe/PCIe Gen6 x4, in a PCIe Gen5 platform
Per-drive ceiling	7.0M IOPS
Max aggregate IOPS	109.6M (16 drives)
GPU	NVIDIA H100 PCIe
Clock, governor	3.0 GHz pinned, performance
IOMMU	translating, drives on vfio-pci
Software	xNVMe(homi_cplane), SPDK(v26.05), fio(3.42), Linux(7.0.0-27-generic)



 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.0 The workload used for testing

Every IOPS number in Q3.1-Q3.5 is random read, 512 B, queue depth 128

The benchmarks that follow focus on IOPS at small I/O sizes.

Specifically, 512 B. Here's why.

- Small I/O exposes processing overhead and latency sensitivity at every layer of the stack: a 64-byte SQE, a 16-byte CQE, two doorbell writes, and the completion poll are all fixed costs. The smaller the I/O, the closer the result gets to that pure protocol overhead.
- 512 B is the smallest size that still exercises the stack's normal command path. Commands like write-zeroes carry no payload, but come with unwanted side effects.
- Small I/O also matters in practice: it's essential for file-system performance, and for large-scale GNN, vector-search, and other workloads dominated by small, scattered I/O. 512 B random reads are a useful synthetic stand-in for that.

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.1: I/O efficiency: state-of-the-art



State of the art for I/O efficiency

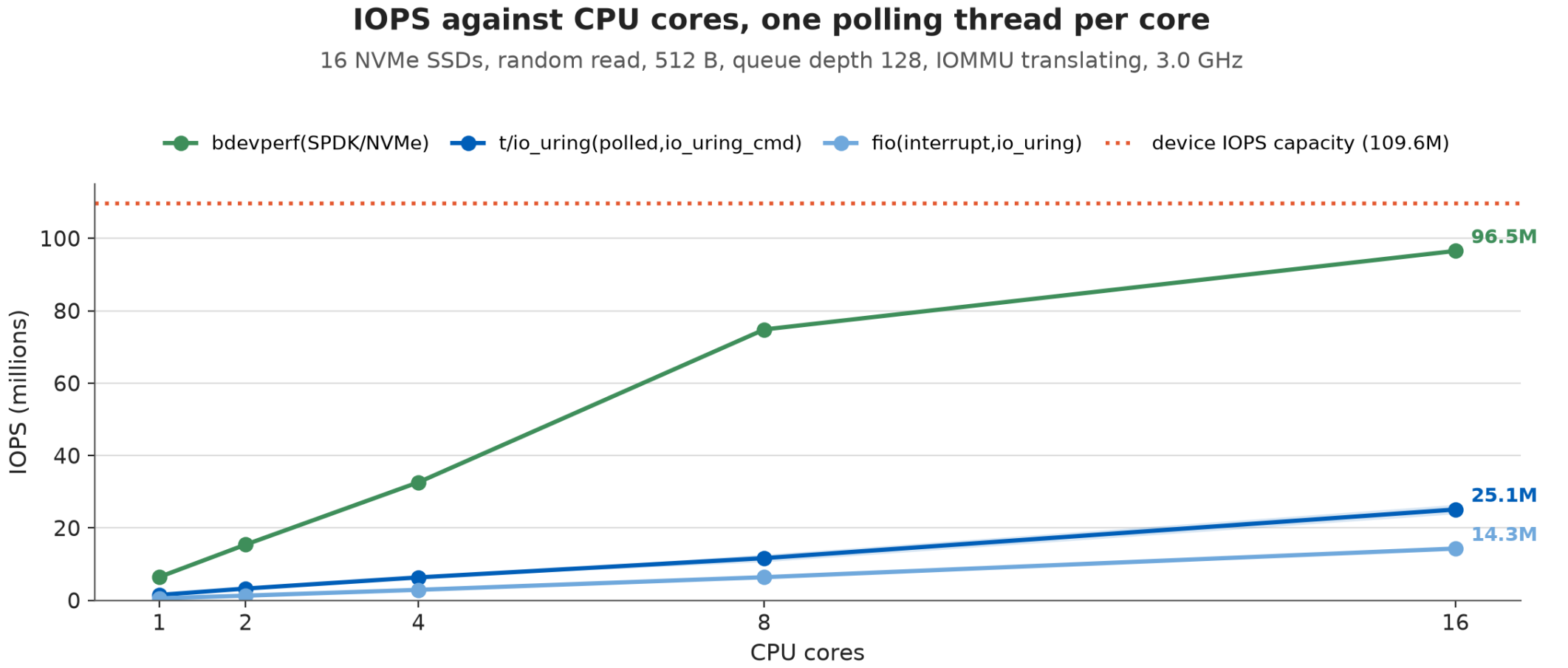
One workload, three I/O paths, each driven by its usual benchmark tool

- Linux Kernel: Block Device I/O
  - The conventional, interrupt-driven, OS-managed path
  - **io_uring** against the NVMe Namespace backed block-device
  - tool: **fio**, using the io_uring I/O engine
- Linux Kernel: "Direct" NVMe I/O
  - The Linux kernel approach to layer-bypass aka NVMe Passthru
  - **io_uring_cmd** against the NVMe Namespace Char-device
  - tool: **t/io_uring**, fio's raw-interface benchmark for it
- The Storage Performance Development Kit (**SPDK**)
  - The user space approach using the polled-mode NVMe driver
  - tool: **bdevperf**, using the SPDK block-device API on top of the NVMe Driver

 ©2026 SNIA Developer Conference. All Rights Reserved.

I/O Efficiency: IOPS as a function of CPU cores

bdevperf + fio (both), over the core count, CPU-initiated hostmem



median of three runs, band is the spread

Disabling the IOMMU roughly doubles the kernel path's own IOPS, landing around 50M at sixteen cores; see Extra for the full kernel-path chart.
dma-buf support on io_uring could cut that overhead without disabling the IOMMU, closing in on ~50M for the kernel-managed path itself.

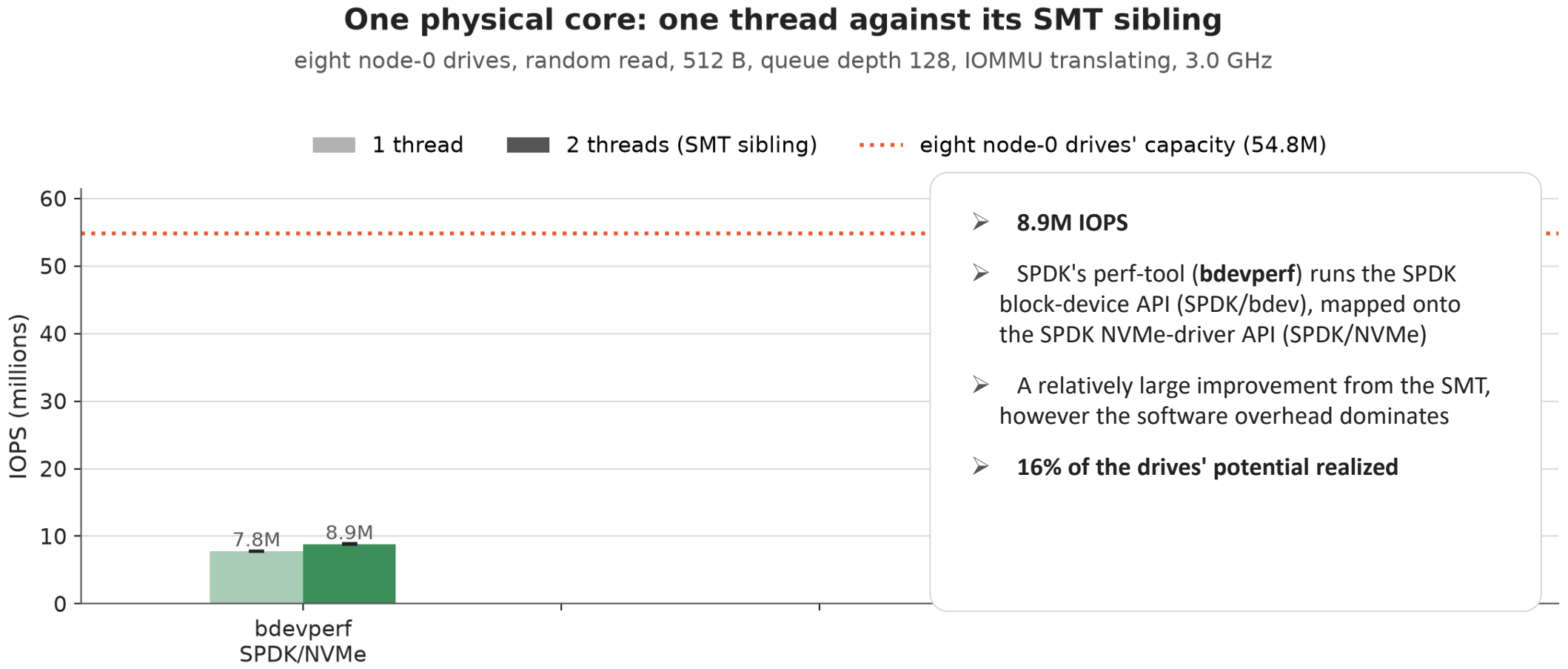
 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.2: I/O efficiency: beyond state-of-the-art



I/O Efficiency: from SPDK to xNVMe

bdevperf (SPDK/NVMe)



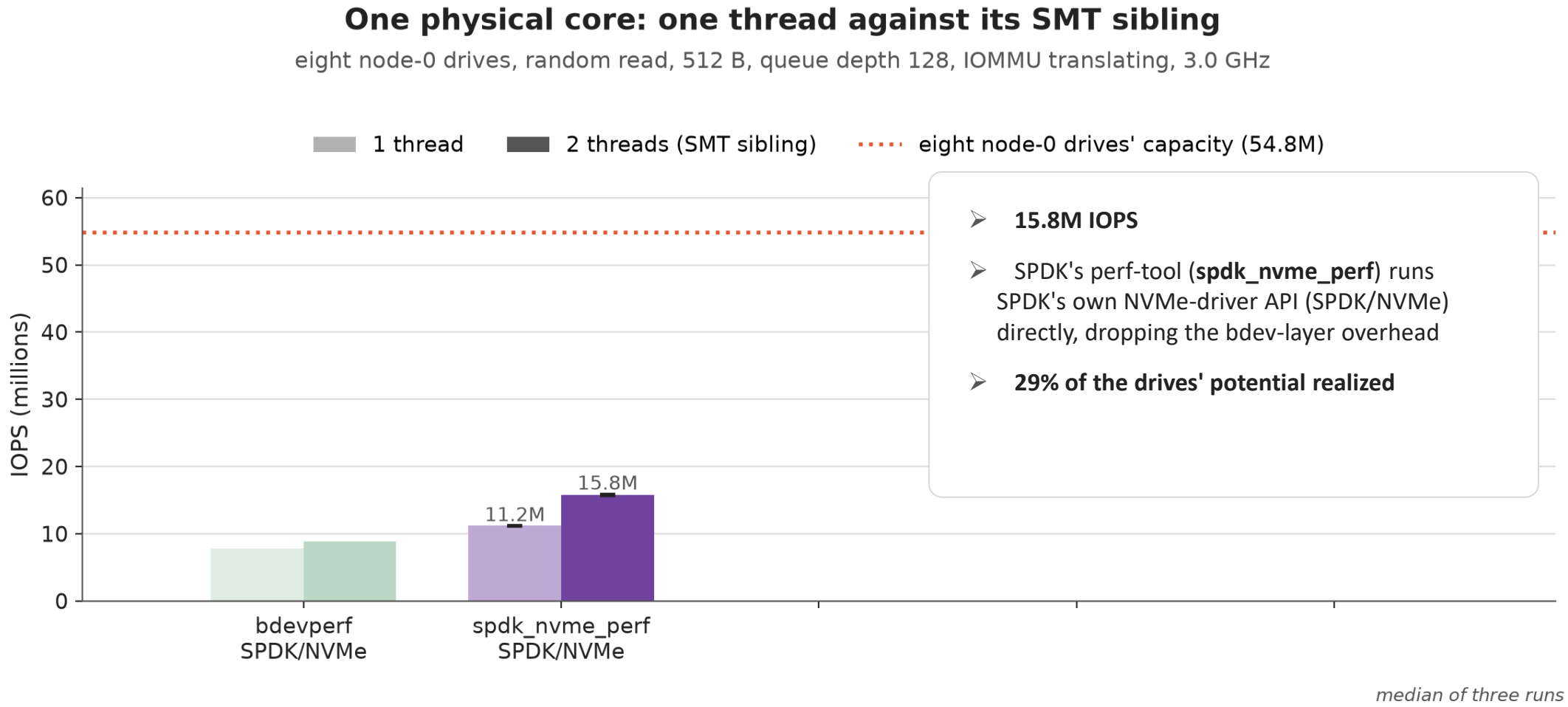
- **8.9M IOPS**
- SPDK's perf-tool (**bdevperf**) runs the SPDK block-device API (SPDK/bdev), mapped onto the SPDK NVMe-driver API (SPDK/NVMe)
- A relatively large improvement from the SMT, however the software overhead dominates
- **16% of the drives' potential realized**

median of three runs

 ©2026 SNIA Developer Conference. All Rights Reserved.

I/O Efficiency: from SPDK to xNVMe

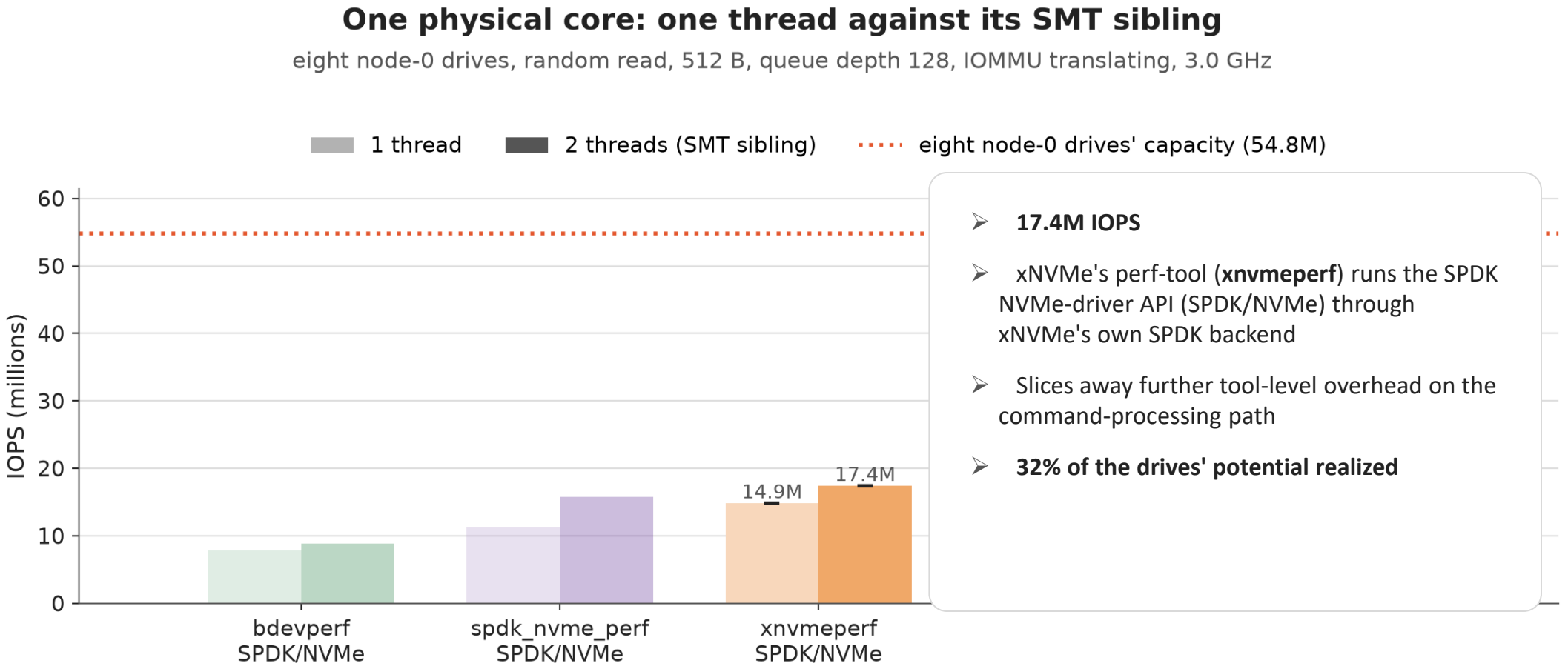
spdk_nvme_perf (SPDK/NVMe)



 ©2026 SNIA Developer Conference. All Rights Reserved.

I/O Efficiency: from SPDK to xNVMe

xnvmeperf (SPDK/NVMe)



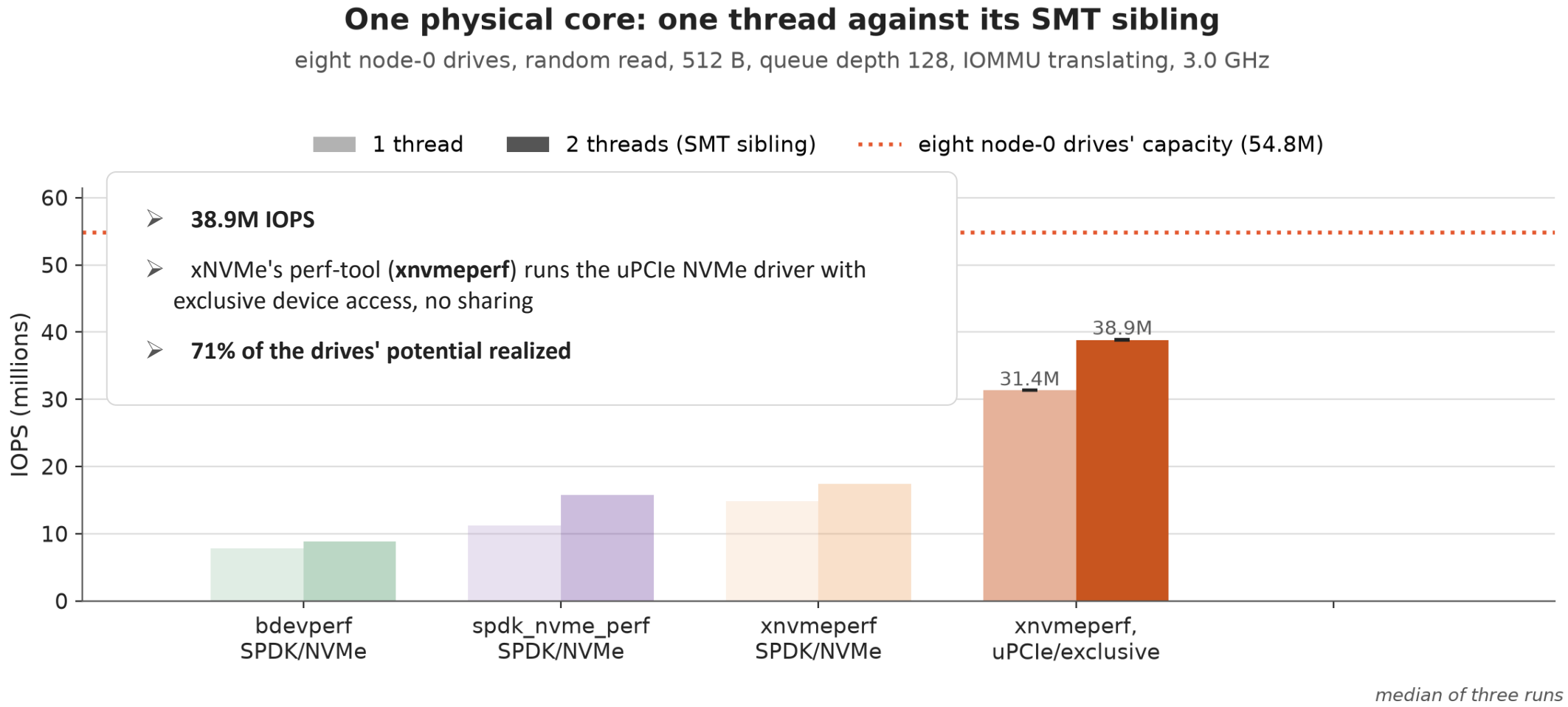
- **17.4M IOPS**
- xNVMe's perf-tool (**xnvmeperf**) runs the SPDK NVMe-driver API (SPDK/NVMe) through xNVMe's own SPDK backend
- Slices away further tool-level overhead on the command-processing path
- **32% of the drives' potential realized**

median of three runs

 ©2026 SNIA Developer Conference. All Rights Reserved.

I/O Efficiency: from SPDK to xNVMe

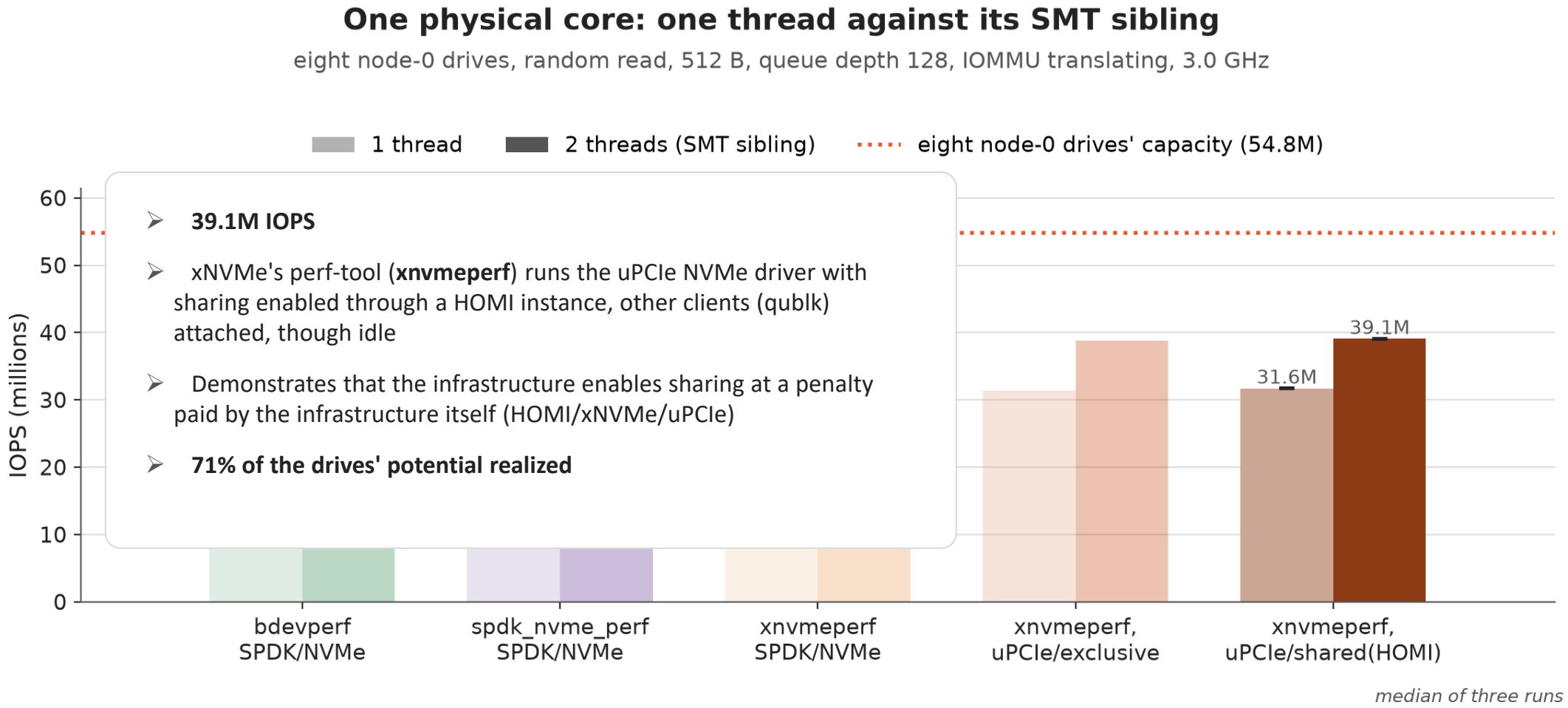
xnvmeperf, uPCIe/exclusive



 ©2026 SNIA Developer Conference. All Rights Reserved.

I/O Efficiency: from SPDK to xNVMe

xnvmeperf, uPCIe/shared (HOMI)



 ©2026 SNIA Developer Conference. All Rights Reserved.

I/O Efficiency: beyond state-of-the-art 1/2

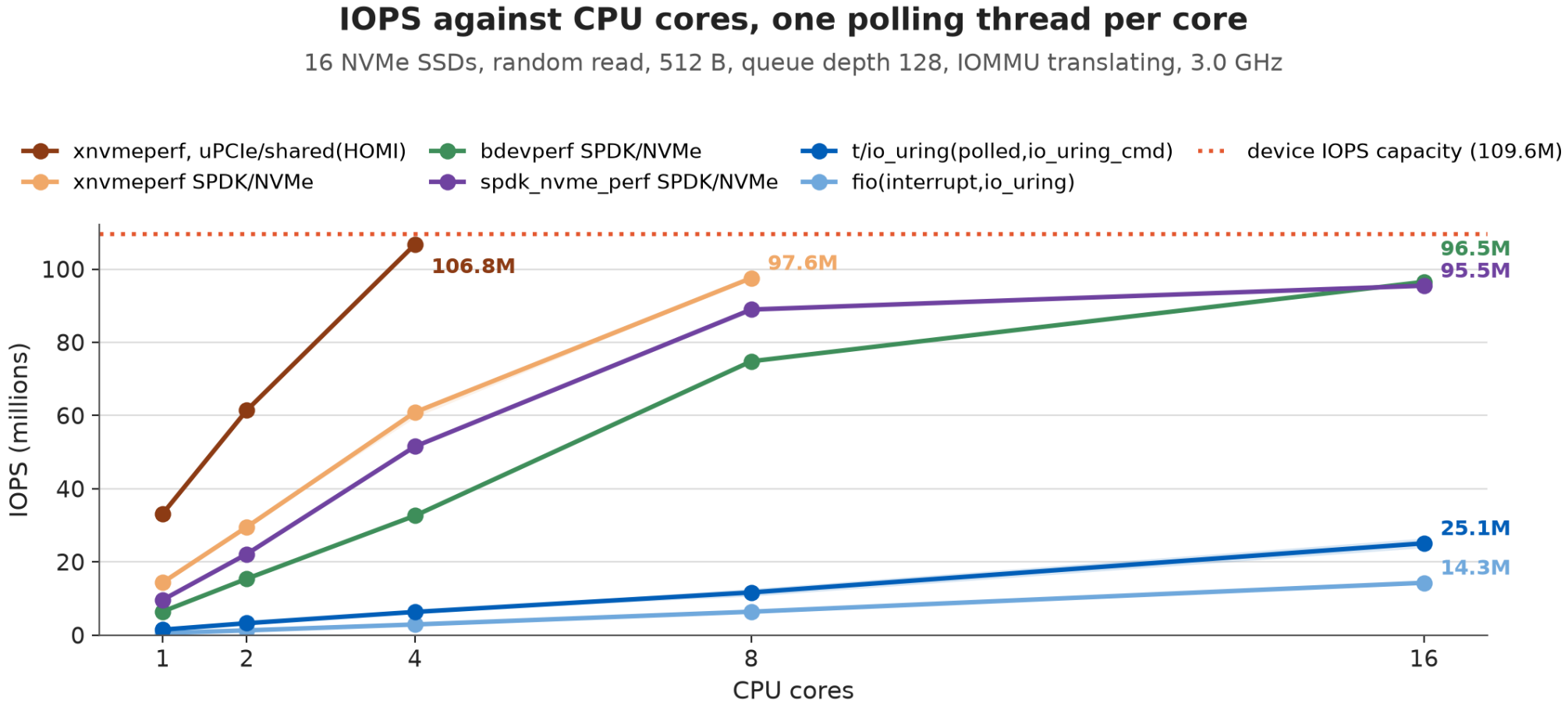
A new upper bound on **efficiency**, scalability, and interoperability

- **Efficiency**
  - **xNVMe/uPCIe** at **39.1M** IOPS using a **single** CPU core @ 3 GHz
  - A **2.2x** improvement over the prior state-of-the-art

 ©2026 SNIA Developer Conference. All Rights Reserved.

I/O Efficiency: IOPS as a function of CPU cores, revisited

Added: xNVMe(uPCIe), xNVMe(SPDK/NVMe), spdk_nvme_perf(SPDK/NVMe)



median of three runs, band is the spread

 ©2026 SNIA Developer Conference. All Rights Reserved.

I/O Efficiency: beyond state-of-the-art 1/2

A new upper bound on efficiency, **scalability**, and interoperability

- **Efficiency**
  - **xNVMe/uPCIe** at **39.1M** IOPS using a **single** CPU core @ 3 GHz
  - A **2.2x** improvement over the prior state-of-the-art
- **Scale**
  - **xNVMe/uPCIe**: **106.8M** IOPS on **4** cores
  - **9.2M** more than the prior state-of-the-art
  - Using **half** the amount of CPU cores

 ©2026 SNIA Developer Conference. All Rights Reserved.

I/O Efficiency: beyond state-of-the-art 2/2

A new upper bound on efficiency, **scalability**, and **interoperability**

- **Efficiency**
  - **xNVMe/uPCIe** at **39.1M** IOPS using a **single** CPU core @ 3 GHz
  - A **2.2x** improvement over the prior state-of-the-art
- **Scale**
  - **xNVMe/uPCIe**: **106.8M** IOPS on **4** cores
  - **9.2M** more than the prior state-of-the-art
  - Using **half** the amount of CPU cores
- **Interoperability**
  - **HOMI** enabled: **SW** controller **sharing** under a **translating** IOMMU

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.3: Accelerator-integrated Storage I/O modes: Comparing peak IOPS of all I/O modes



Q3.3: Comparing peak IOPS of all I/O modes

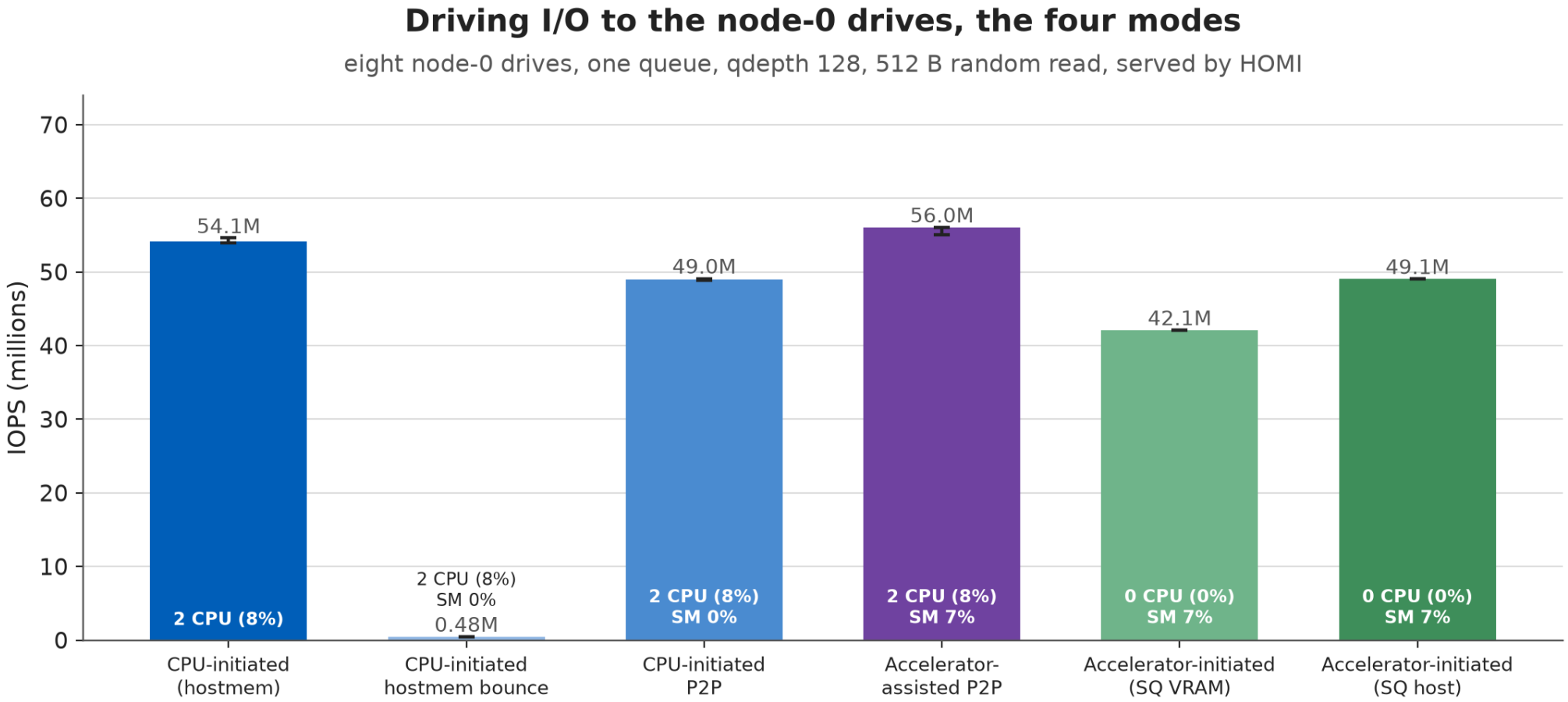
one H100, its eight node-local drives, two node-0 cores, served by HOMI

- Scoped to a **single NUMA node**, to weigh the initiators fairly
  - Two CPU cores, not using SMT, thus one HWT per CPU core
  - One H100, its eight node-local drives
- **HOMI** is running with other clients attached
  - HOMI serves the controllers to the benchmark processes
  - qublk is running, although idle
  - This is done to assert that HOMI and qublk behave as one would require them to
- **A comparative study**
  - IOPS rate of all the I/O modes as well
  - BW utilization in absolute numbers and per I/O
  - CPU-initiated hostmem and CPU-initiated hostmem with HtoD bounce for reference

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.3: Accelerator-integrated Storage I/O modes

eight node-0 drives, one queue, queue depth 128, served by HOMI

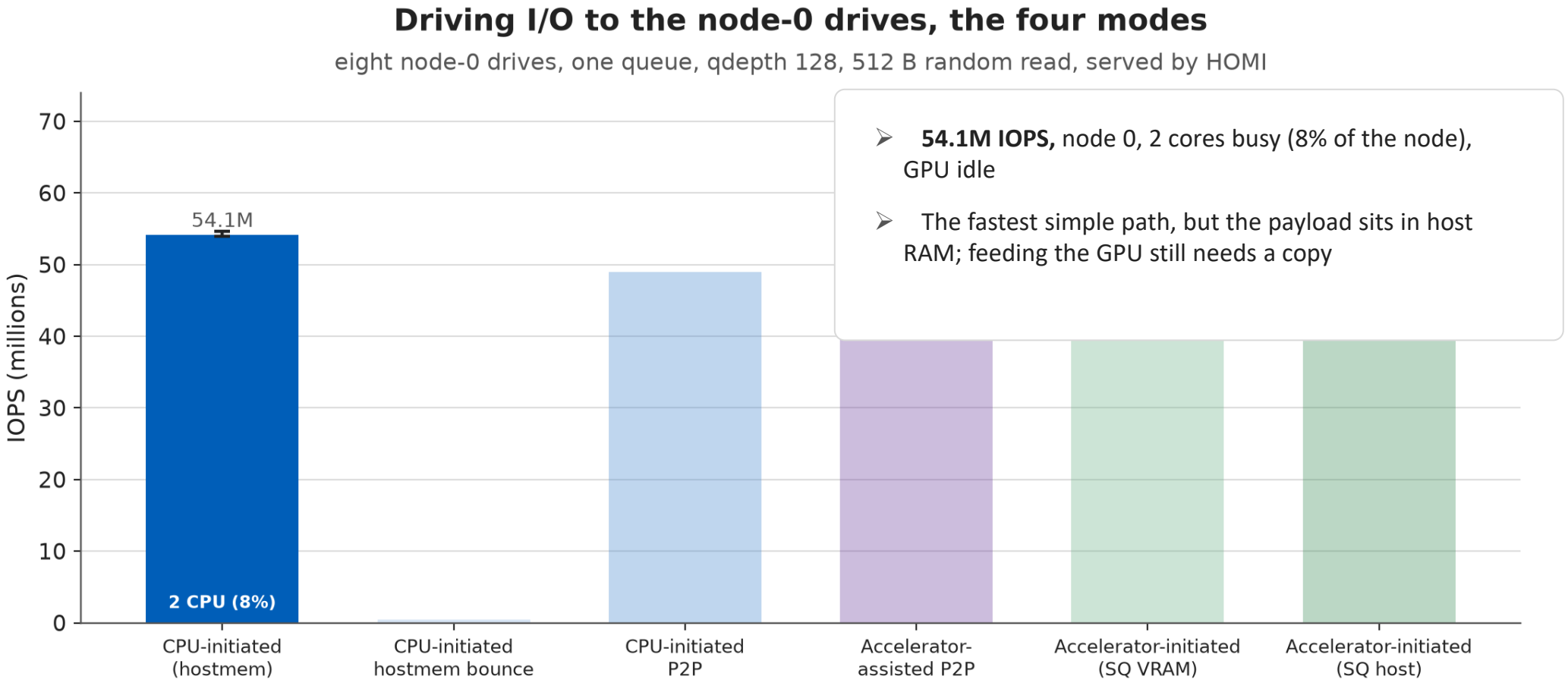


median of three runs; annotations are CPUs busy (qublk's own idle-attached core excluded), as a share of the node's 24 cores, and GPU SM-active during the run

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.3: Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem



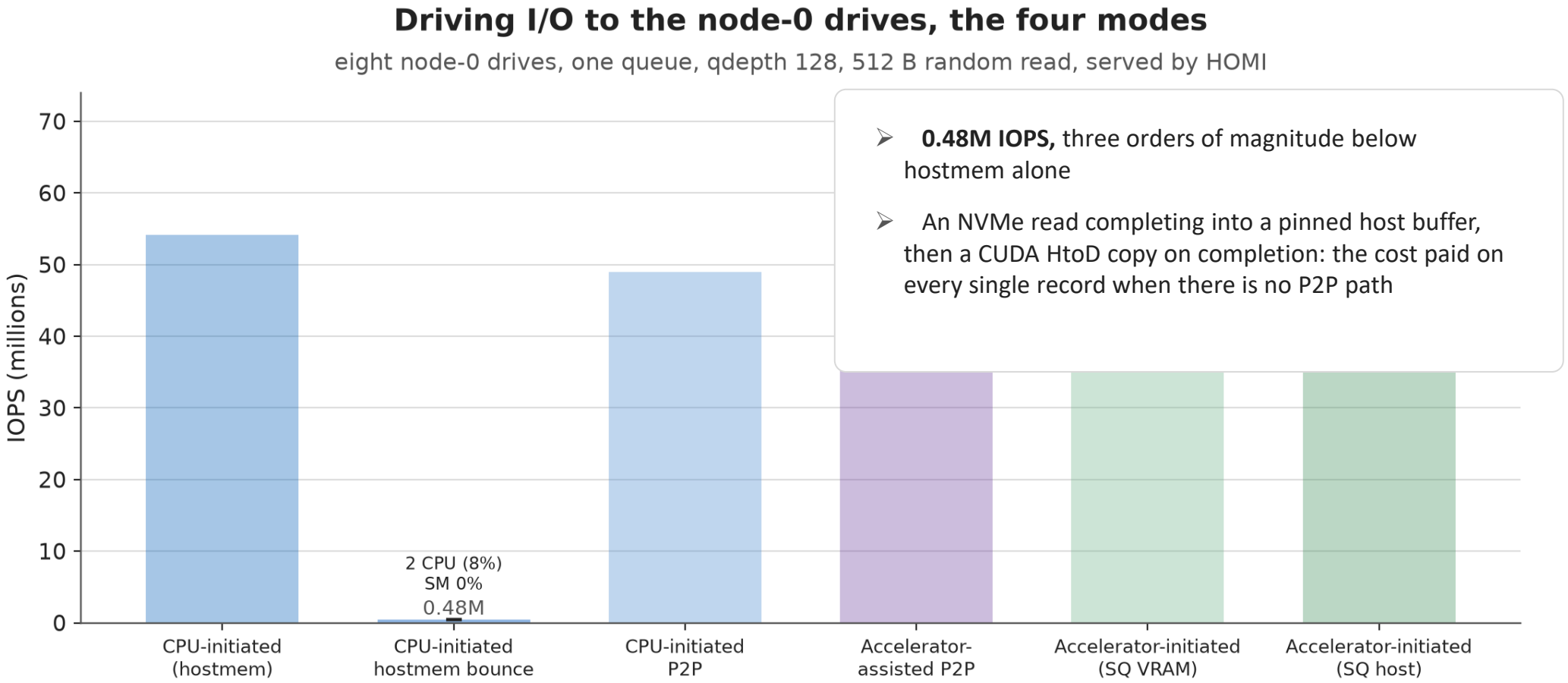
- **54.1M IOPS,** node 0, 2 cores busy (8% of the node), GPU idle
- The fastest simple path, but the payload sits in host RAM; feeding the GPU still needs a copy

median of three runs; annotations are CPUs busy (qublk's own idle-attached core excluded), as a share of the node's 24 cores, and GPU SM-active during the run

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.3: Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated hostmem + bounce-buffer (runtime-invoked HtoD transfer)



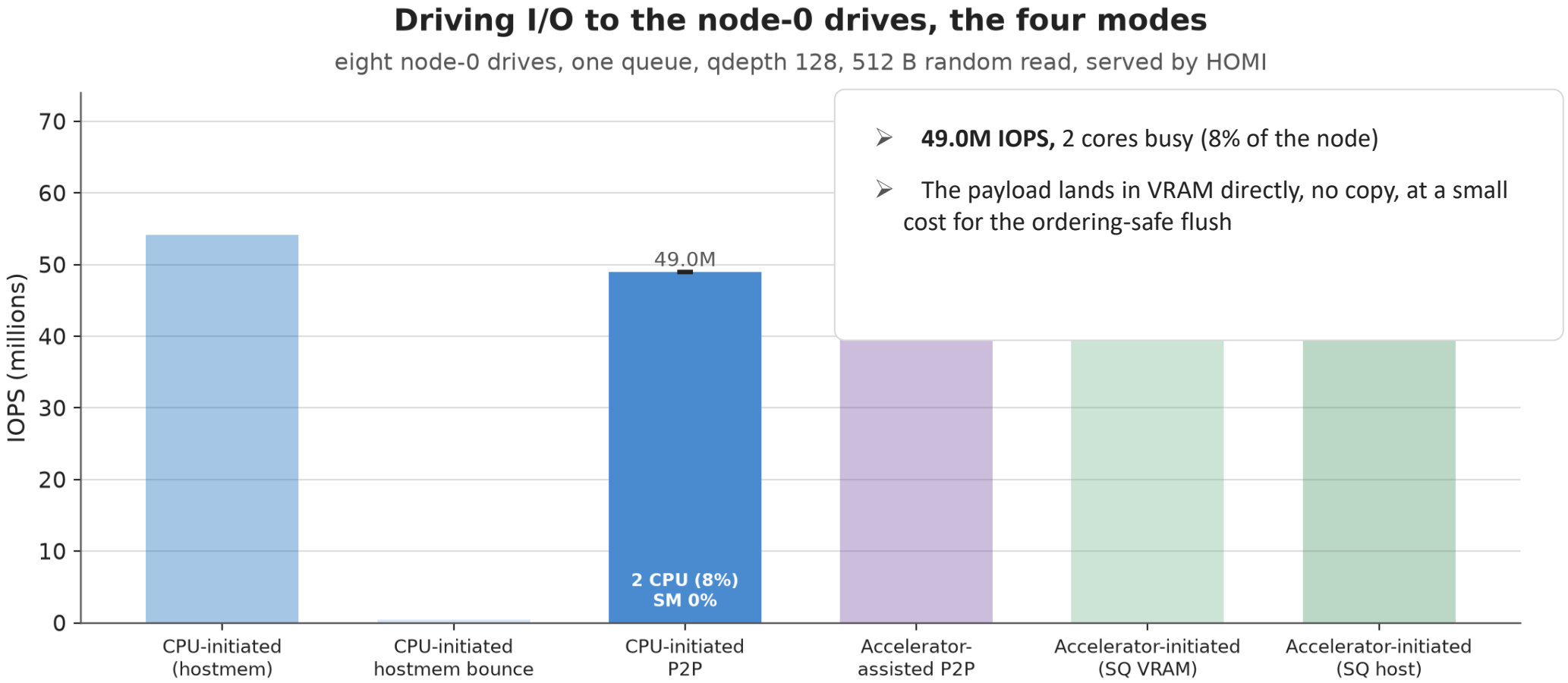
- **0.48M IOPS,** three orders of magnitude below hostmem alone
- An NVMe read completing into a pinned host buffer, then a CUDA HtoD copy on completion: the cost paid on every single record when there is no P2P path

median of three runs; annotations are CPUs busy (qublk's own idle-attached core excluded), as a share of the node's 24 cores, and GPU SM-active during the run

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.3: Accelerator-integrated Storage I/O modes

I/O Mode: CPU-initiated P2P



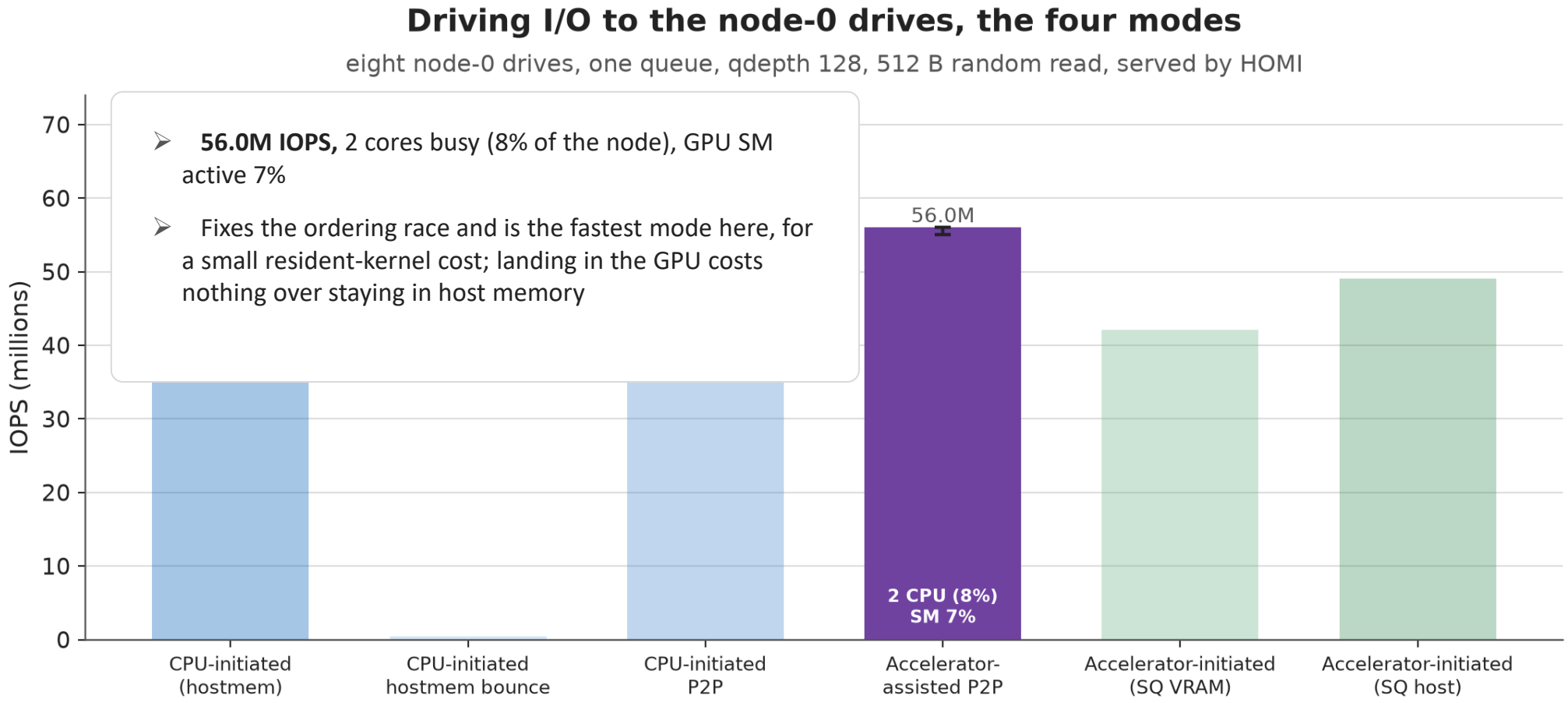
- **49.0M IOPS,** 2 cores busy (8% of the node)
- The payload lands in VRAM directly, no copy, at a small cost for the ordering-safe flush

median of three runs; annotations are CPUs busy (qublk's own idle-attached core excluded), as a share of the node's 24 cores, and GPU SM-active during the run

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.3: Accelerator-integrated Storage I/O modes

I/O Mode: Accelerator-assisted P2P

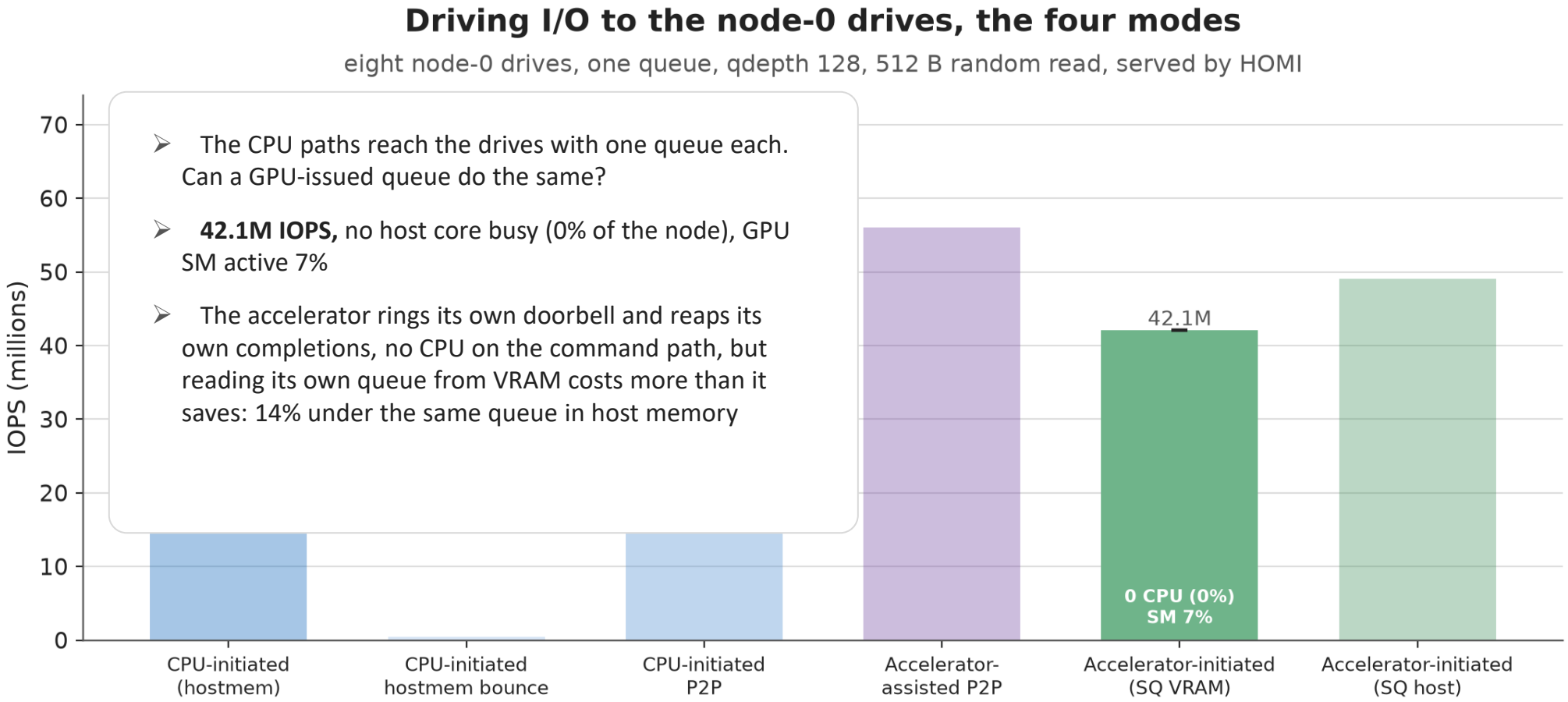


median of three runs; annotations are CPUs busy (qublk's own idle-attached core excluded), as a share of the node's 24 cores, and GPU SM-active during the run

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.3: Accelerator-integrated Storage I/O modes

I/O Mode: Accelerator-initiated P2P (SQ VRAM)



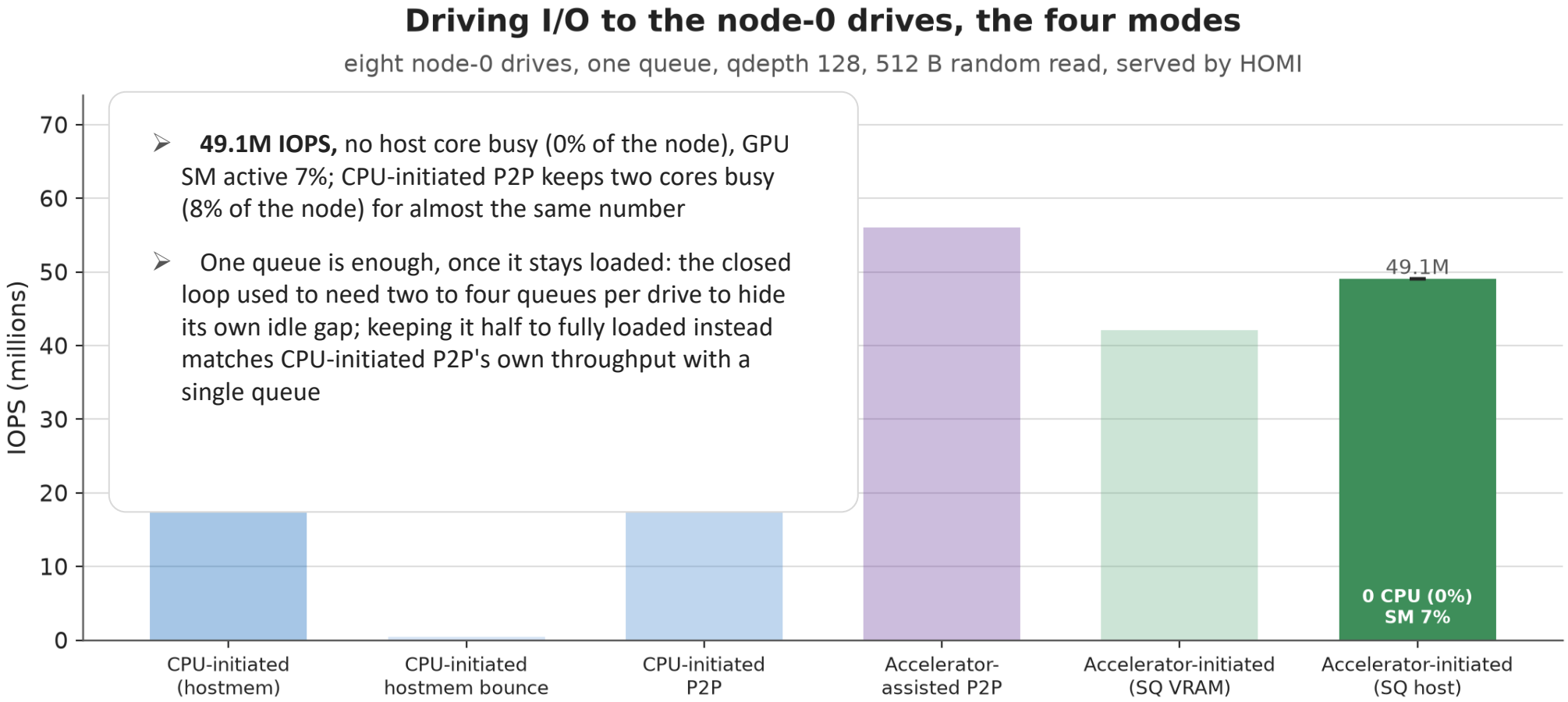
- The CPU paths reach the drives with one queue each. Can a GPU-issued queue do the same?
- **42.1M IOPS,** no host core busy (0% of the node), GPU SM active 7%
- The accelerator rings its own doorbell and reaps its own completions, no CPU on the command path, but reading its own queue from VRAM costs more than it saves: 14% under the same queue in host memory

median of three runs; annotations are CPUs busy (qublk's own idle-attached core excluded), as a share of the node's 24 cores, and GPU SM-active during the run

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.3: Accelerator-integrated Storage I/O modes

I/O Mode: Accelerator-initiated P2P (SQ hostmem)



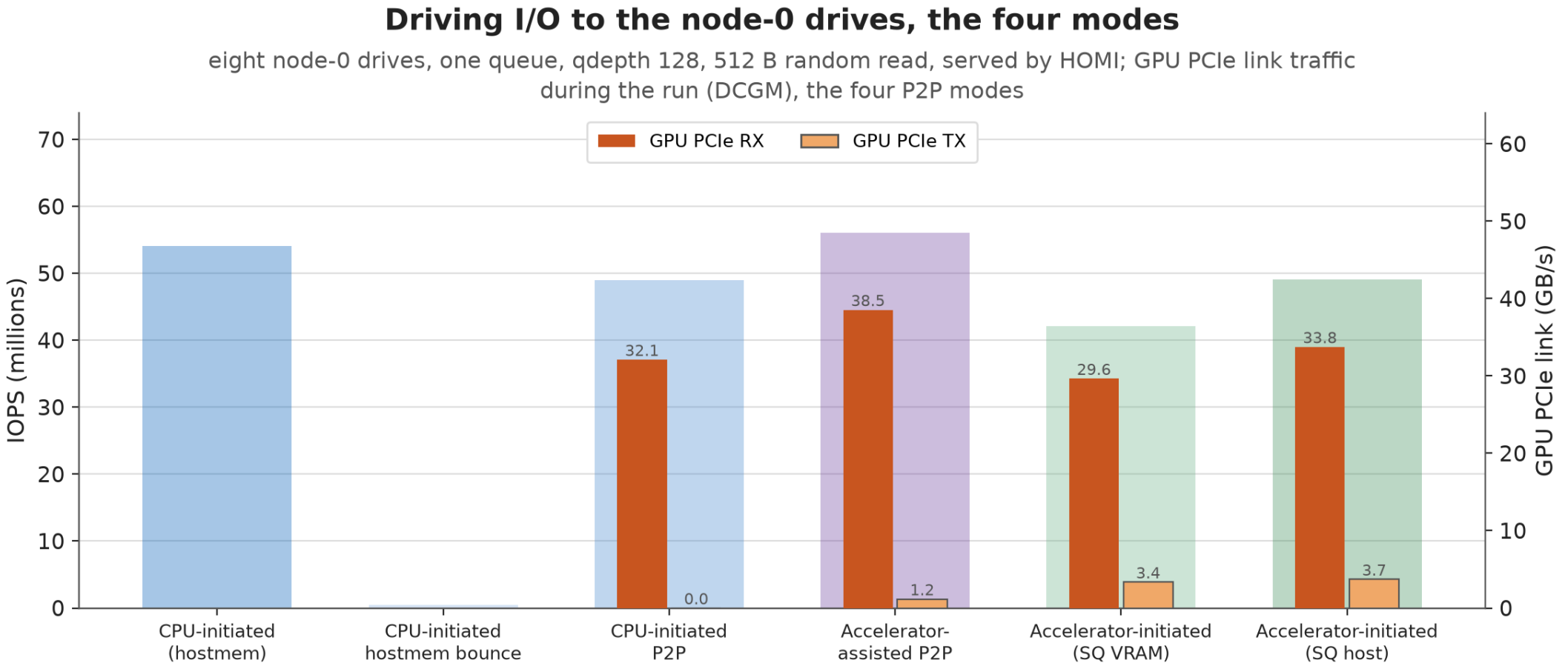
- **49.1M IOPS,** no host core busy (0% of the node), GPU SM active 7%; CPU-initiated P2P keeps two cores busy (8% of the node) for almost the same number
- One queue is enough, once it stays loaded: the closed loop used to need two to four queues per drive to hide its own idle gap; keeping it half to fully loaded instead matches CPU-initiated P2P's own throughput with a single queue

median of three runs; annotations are CPUs busy (qublk's own idle-attached core excluded), as a share of the node's 24 cores, and GPU SM-active during the run

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.3: Accelerator-integrated Storage I/O modes

GPU PCIe link traffic, in GB/s, during the execution of the P2P modes

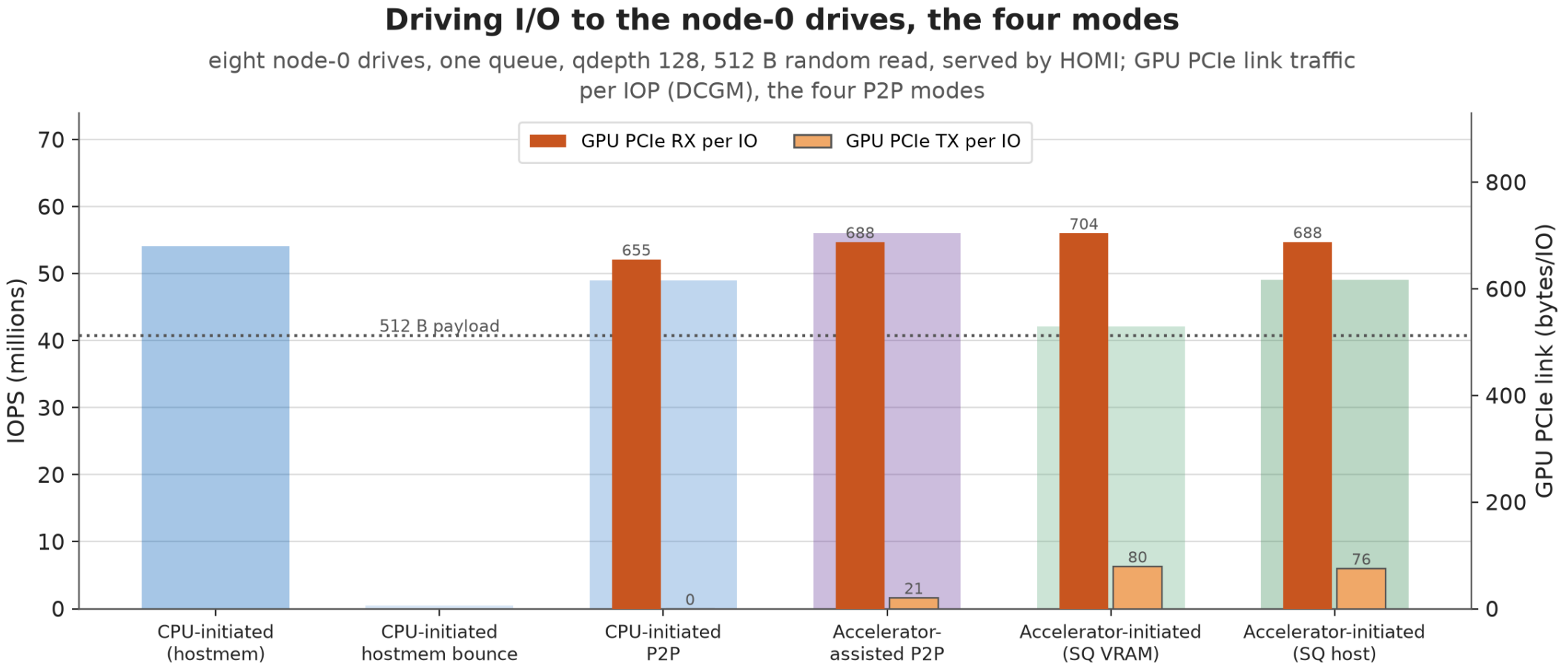


median of three runs; PCIe RX/TX are DCGM's own GPU link counters for the run

 ©2026 SNIA Developer Conference. All Rights Reserved.

Q3.3: Accelerator-integrated Storage I/O modes

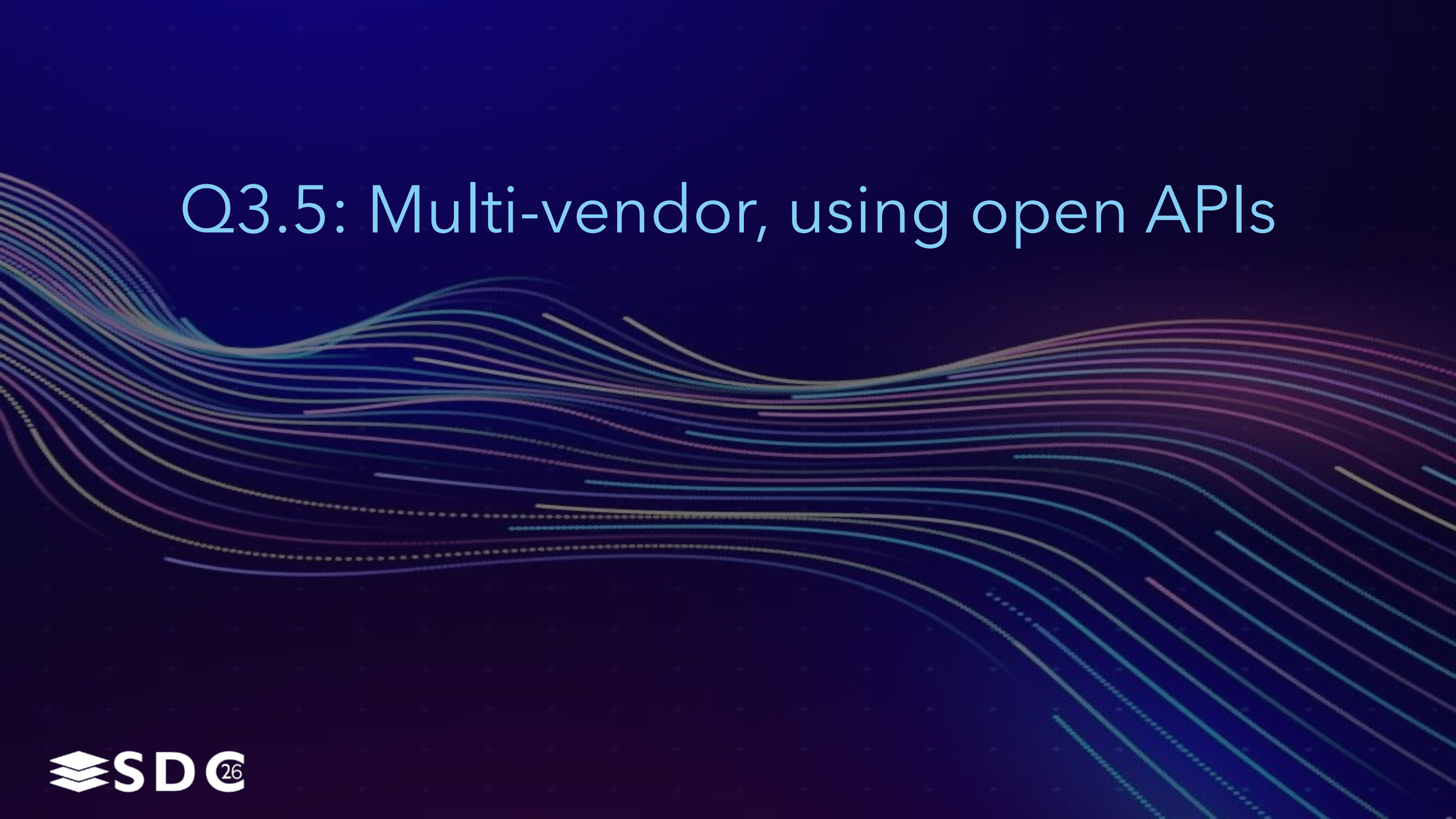
GPU PCIe link traffic per IOP, during the execution of the P2P modes



median of three runs; PCIe RX/TX are DCGM's own GPU link counters for the run, divided by the run's own IOPS

 ©2026 SNIA Developer Conference. All Rights Reserved.

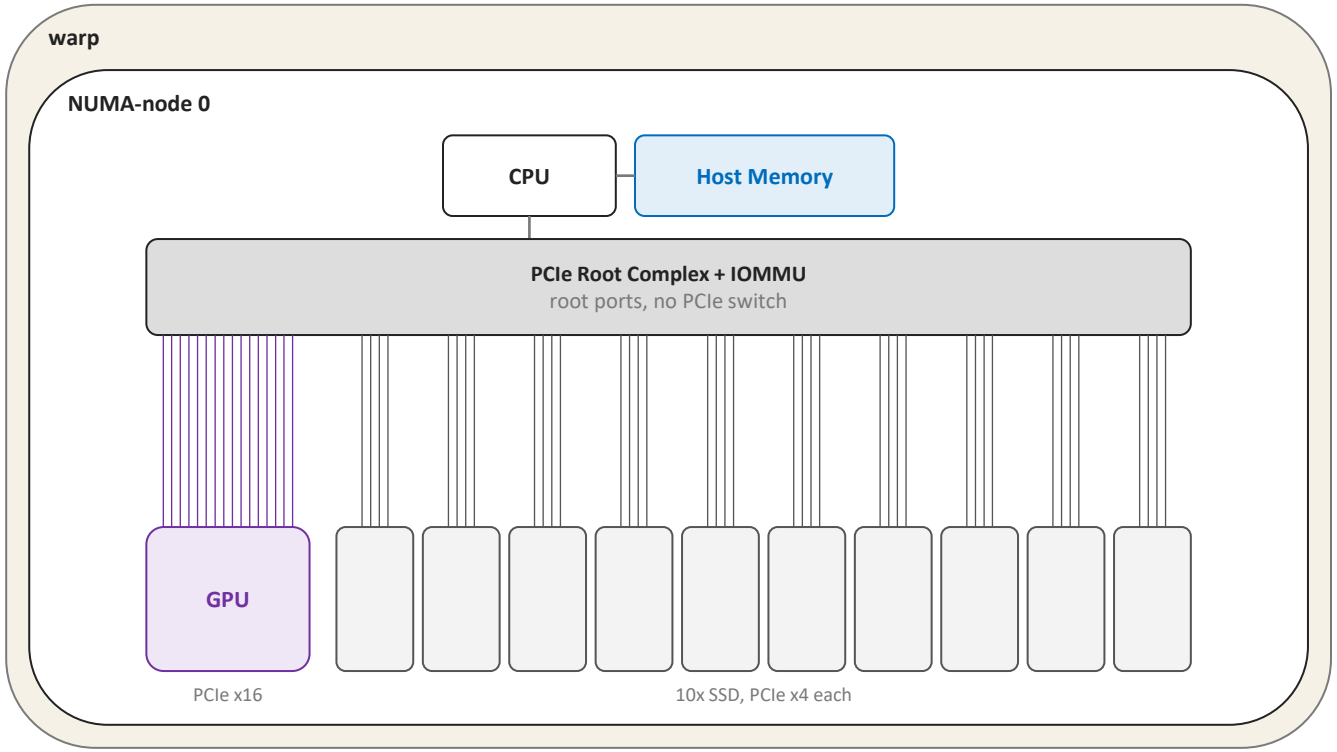
Q3.5: Multi-vendor, using open APIs



The system: warp

Same chassis, only the GPU changes: NVIDIA vs. AMD on identical hardware

	warp
CPU	1x AMD EPYC 7532 (32-core)
NUMA	1 domain; single socket, no cross-node placement
Drives	4x Gen4 @1.09M, 4x Gen4 @2.0M, 2x Gen5 @3.88M
Max aggregate IOPS	20.1M (10 drives)
GPU	NVIDIA RTX A6000 AMD RX 7800 XT Swapped per run
Clock, governor	2.4 GHz pinned, performance
IOMMU	translating, drives on vfio-pci
xNVMe build	homi_gpu tip, v0.7.5-547-g8581edac

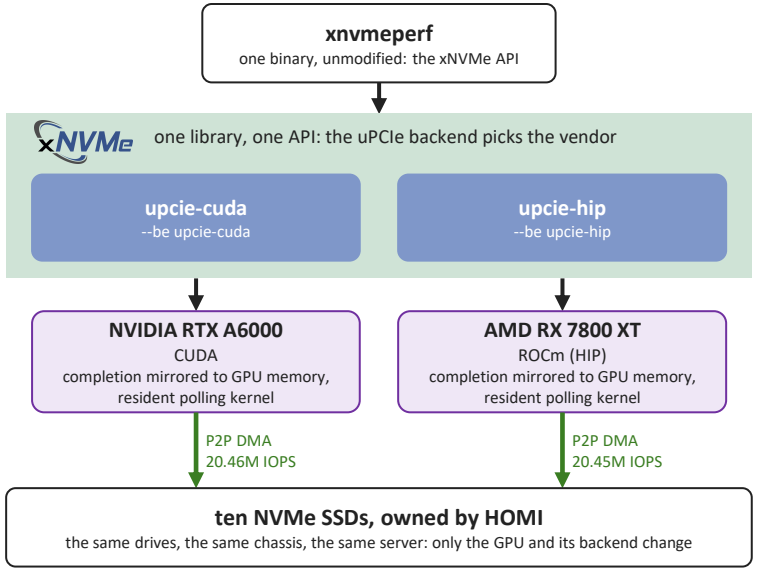


 ©2026 SNIA Developer Conference. All Rights Reserved.

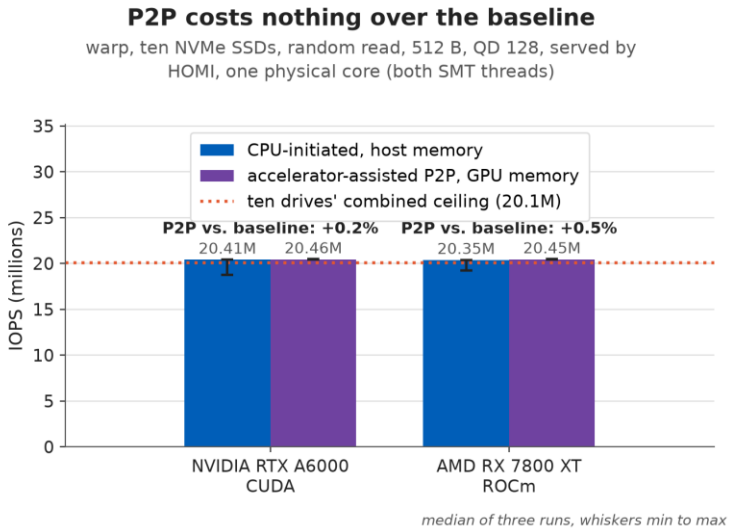
A genuinely open, multi-vendor path

one xnvmeperf binary, RTX A6000 and RX 7800 XT, same ten drives

- Performance Parity
  - Same 20M IOPS roofline on the system using accelerator-assisted P2P with both GPUs
  - Neither GPU is the limiting factor here
- **One client binary**, no proprietary API
  - The same xnvmeperf, unmodified, reaches both vendors' GPUs
  - Code written against the xNVMe CORE API runs unaltered on CUDA as well as ROCm
- cuFile compatible
  - The **OpenDS** project provides adoption facilitators for legacy code; the cuFile API is optional, and OpenDS provides APIs that better serve efficient I/O loops



Nothing on this path is vendor-owned: the vendor runtime is one backend among two, below an open API.



 ©2026 SNIA Developer Conference. All Rights Reserved.

Q4: Where do we go from here?



The checklist

What shipped, what's still open

shipped | works here, out of tree: waiting on upstream Linux | in progress, with a date | not yet, ours to do | the open challenge

SHIPPED

- libnvm/bam → Linux kernel Upstream Infrastructure
  libnvm/bam → uPCIe (dma-buf, iommufd, vfio-pci, uio_pci_generic, hugepages)
- Exclusive device ownership → Host-orchestrated Multipath I/O (HOMI)
- Single accelerator-vendor → Multi-vendor (Any with dma-buf exporter support)
  Single accelerator-vendor → Multi-vendor (Verified using CUDA and ROCm)
- Dropped abstractions → Block-layer enabled with xNVMe/qublk
- Dropped abstractions → File I/O (unmounted) with libxal / xal-server

OPEN

- custom kernel patches → dma-buf importer and iommu-map-pa, out-of-tree modules shipped with uPCIe; gone once IOMMU_IOAS_MAP_FILE takes more than vfio-pci dma-bufs (6.19)
- All four I/O modes on CUDA; ROCm accelerator-initiated still needs the controller BAR as a dma-buf the GPU can map
- NVMe/RDMA: in progress, expected functional by end of 2026
- NVMe/TCP: supported, not optimized
- qublk idle-attach: busy-waits on its own core; a fix is in progress, expected mid-October 2026

The challenge to conquer

Fast, efficient file leases and extent-info export to user space, in upstream Linux

XAL reads extents through its own XFS walk, or FIEMAP (mounted, debugging only), and keeps them in place with fsfreeze or reflink snapshots; nothing upstream lets user space fetch a file's extents cheaply and hold a lease that stops them moving while an accelerator DMAs against them

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O (AiSIO)

The vision: Accelerators as first-class storage citizens

Next release
1 Nov. 2026

- Accelerators: GPUs, NPUs, TPUs, and similar
  - Current focus is on a single class of accelerator: GPUs
- Preserve files, filesystems, and OS control
- Four I/O modes:
  - **CPU-initiated hostmem**: CPU drives I/O, data lands in host memory (possibly copied to accelerator)
  - **CPU-initiated P2P**: CPU drives I/O, data direct to device memory
  - **Accelerator-assisted P2P**: CPU drives I/O, completion mirrored beside the payload on the accelerator
  - **Accelerator-initiated P2P (aka GPU-initiated)**: Accelerator drives I/O directly into its own memory
- Interoperability over replacement
  - Extend the existing stack without sacrificing performance
- All open source; in or targeting upstream, with an open collaboration model

 ©2026 SNIA Developer Conference. All Rights Reserved.

Accelerator-integrated Storage I/O (AiSIO)

The vision: Accelerators as first-class storage citizens

Next release
1 Nov. 2026

- Accelerators: GPUs, NPUs, TPUs, and similar
  - Current focus is on a single class of accelerator: GPUs
- Preserve files, filesystems, and OS control
- Four I/O modes:
  - **CPU-initiated hostmem**: CPU drives I/O, data lands in host memory (possibly copied to accelerator)
  - **CPU-initiated P2P**: CPU drives I/O, data direct to device memory
  - **Accelerator-assisted P2P**: CPU drives I/O, completion mirrored beside the payload on the accelerator
  - **Accelerator-initiated P2P (aka GPU-initiated)**: Accelerator drives I/O directly into its own memory
- Interoperability over replacement
  - Extend the existing stack without sacrificing performance
- All open source; in or targeting upstream, with an open collaboration model



 ©2026 SNIA Developer Conference. All Rights Reserved.

Thank you for attending!

Join us at the BoF
SNIA xIO Technical Work Group
At this conference

Next xIO-TWG meeting
SNIA xIO Technical Work Group
October 19, 2026

Please remember to rate this session.

Access presentations at sniadeveloper.org/conference

Extra

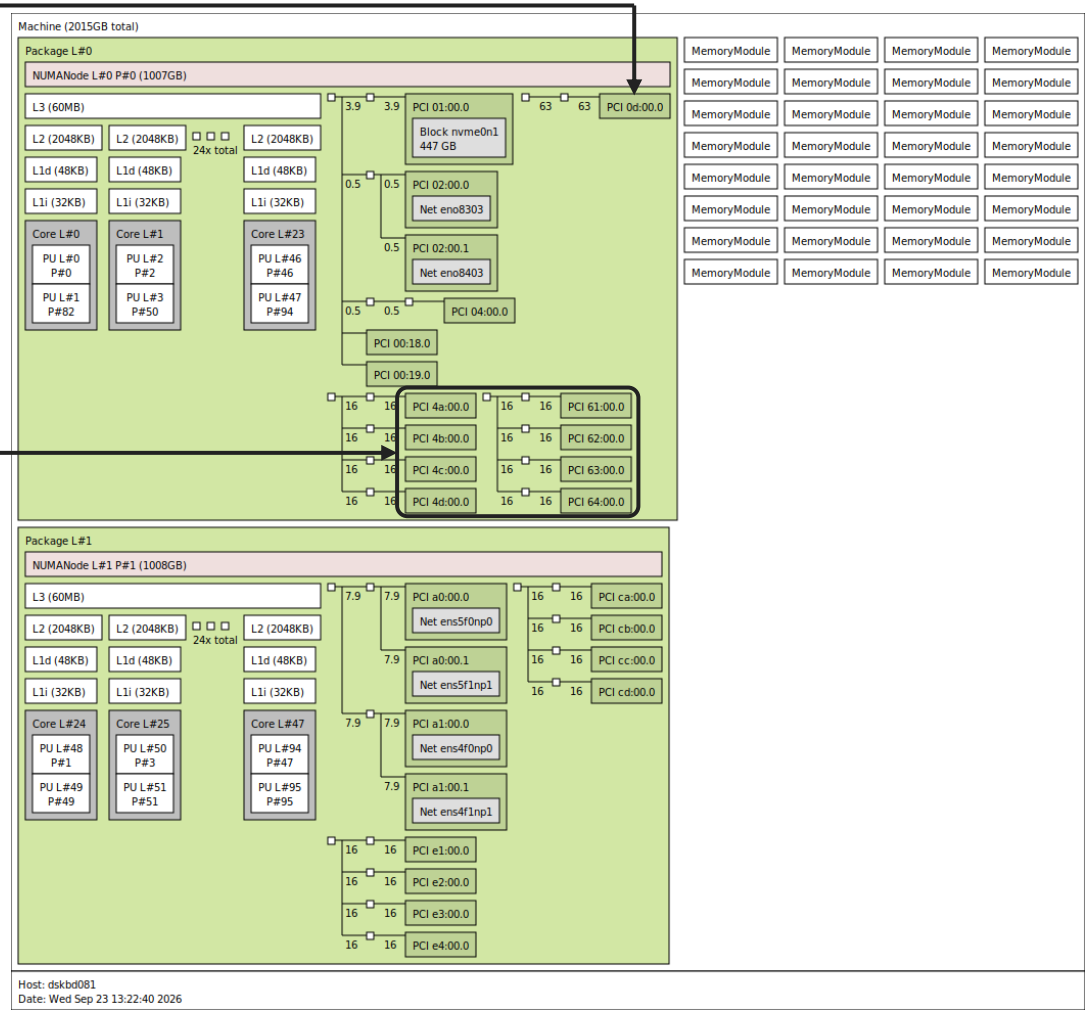


Extra: maxlab topology, as lstopo sees it

lstopo --of png on maxlab (R760); the system slide's diagram simplifies this

NVIDIA H100 GPU

Eight node-0 drives,
used for today's testing



 ©2026 SNIA Developer Conference. All Rights Reserved.

Extra: why four drives per core/thread?

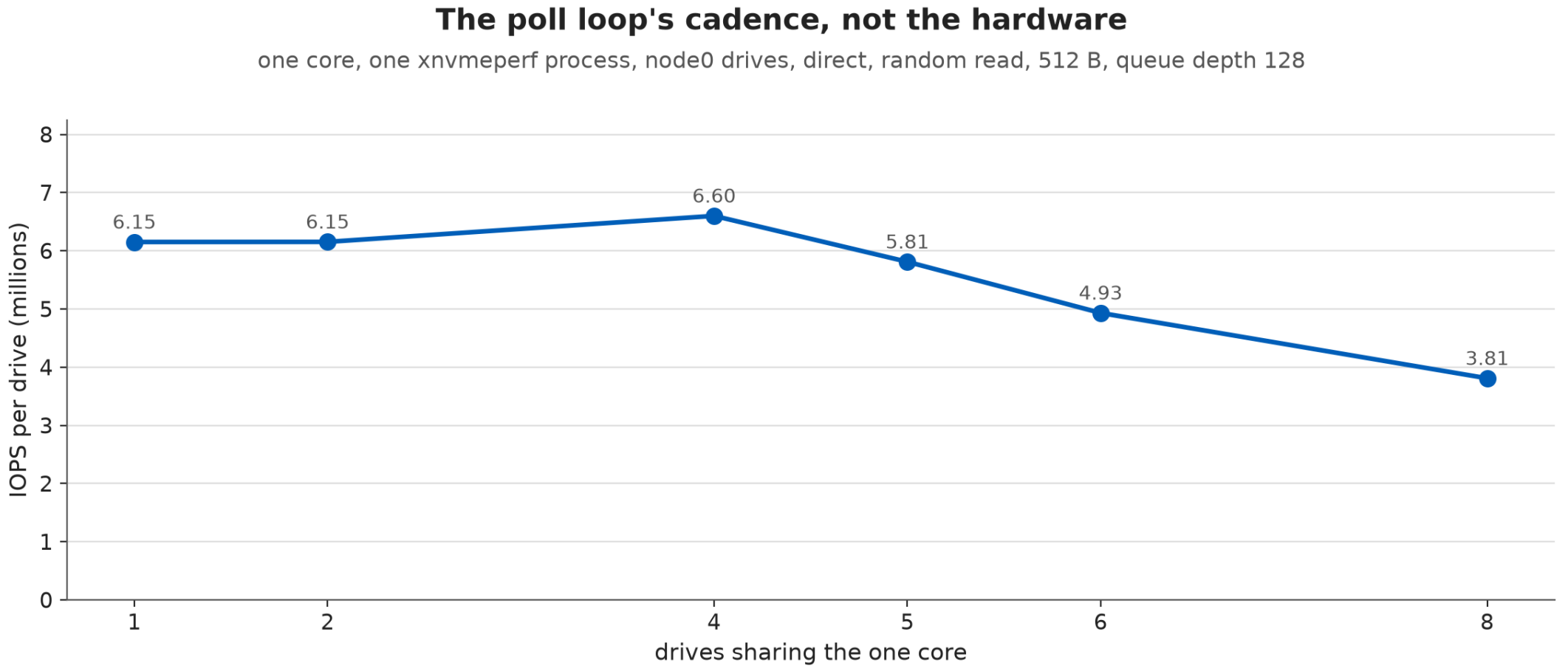
one core, node-0 drives, direct; drives per polling thread swept, 1 to 8

- **The shape was not arbitrary.** Question: is four drives per polling thread special, or would any shape do?
- **It is a real sweet spot:** one and two drives per core both land at 6.15M/drive; four climbs to 6.60M; beyond that it falls, 5.81M at five, 4.93M at six, 3.81M at eight
  - Isolated on one core, node-0 drives, direct; only how many drives share the poll loop changes
- **The mechanism:** a thread revisiting one drive's queue gets more time to accumulate completions between checks the more other drives it interleaves with; too few wastes polling overhead, too many overloads the one core driving them
- **SPDK does not share this:** bdevperf and spdk_nvme_perf only get worse per drive as more devices share a core, no sweet spot, in the same core-scaling sweep

 ©2026 SNIA Developer Conference. All Rights Reserved.

Extra: the climb mechanism, isolated

one core, node-0 drives, direct, devices per core swept



median of three runs per point; core, drives and everything else held fixed, only devices-per-core changes

 ©2026 SNIA Developer Conference. All Rights Reserved.

Extra: the kernel over cores, IOMMU on and off

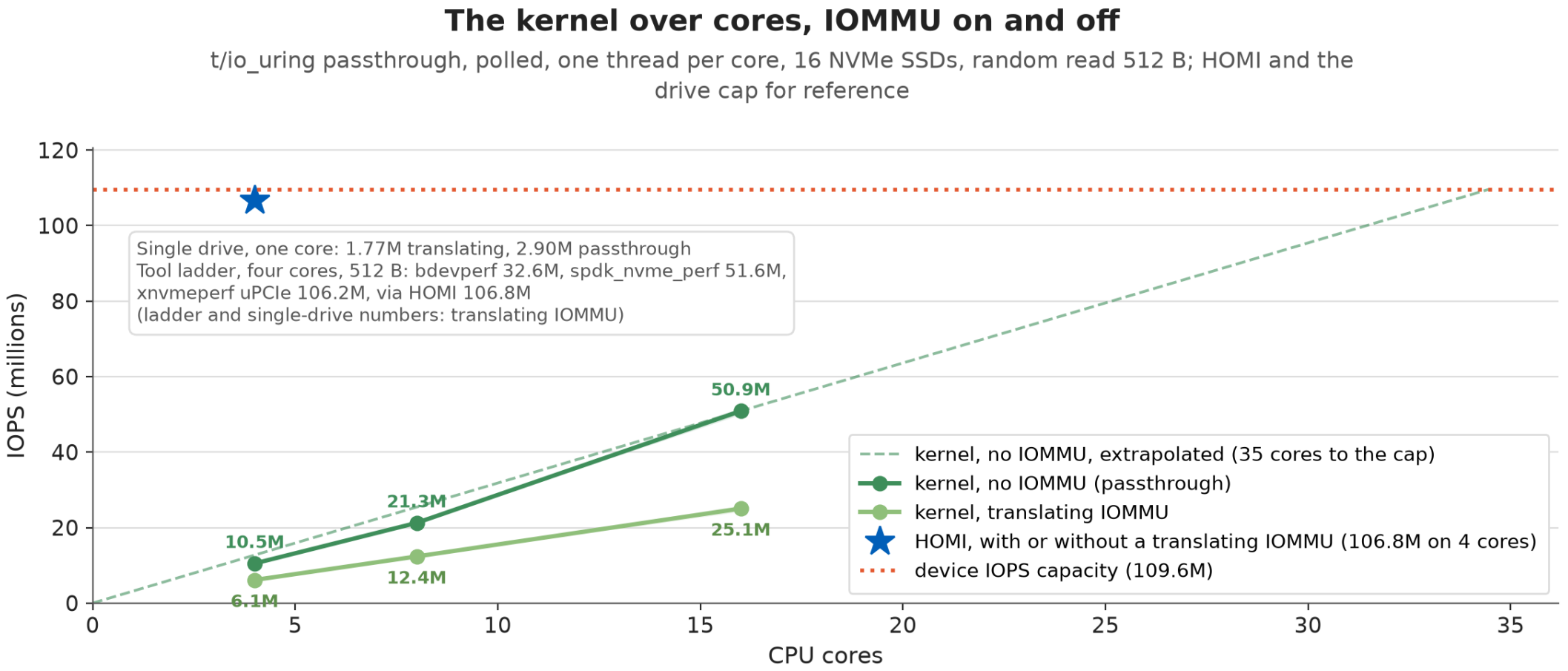
fio and t/io_uring on all sixteen drives, IOMMU translating vs. passthrough

- Not part of the talk; here for questions about the Linux kernel path on the same sixteen drives
  - io_uring / io_uring_cmd through fio and fio's t/io_uring; polled completions, one thread per core; the best form shown is t/io_uring with NVMe passthrough
- **The IOMMU is the kernel's tax:** it maps and unmaps every request; uPCIe and SPDK map their memory once and never pay it
  - Translating, the kernel does 1.6M IOPS per core; in passthrough 3.2M, about double
- **Cores to the cap:** the kernel is linear to 16 cores (50.9M in passthrough), so ~35 cores reach the 109.6M cap without the IOMMU; with it, more cores than this host has
- **HOMI** reaches 106.8M on four cores with or without a translating IOMMU

 ©2026 SNIA Developer Conference. All Rights Reserved.

Extra: kernel IOPS over cores

single-drive and four-core tool-ladder reference numbers on the chart itself

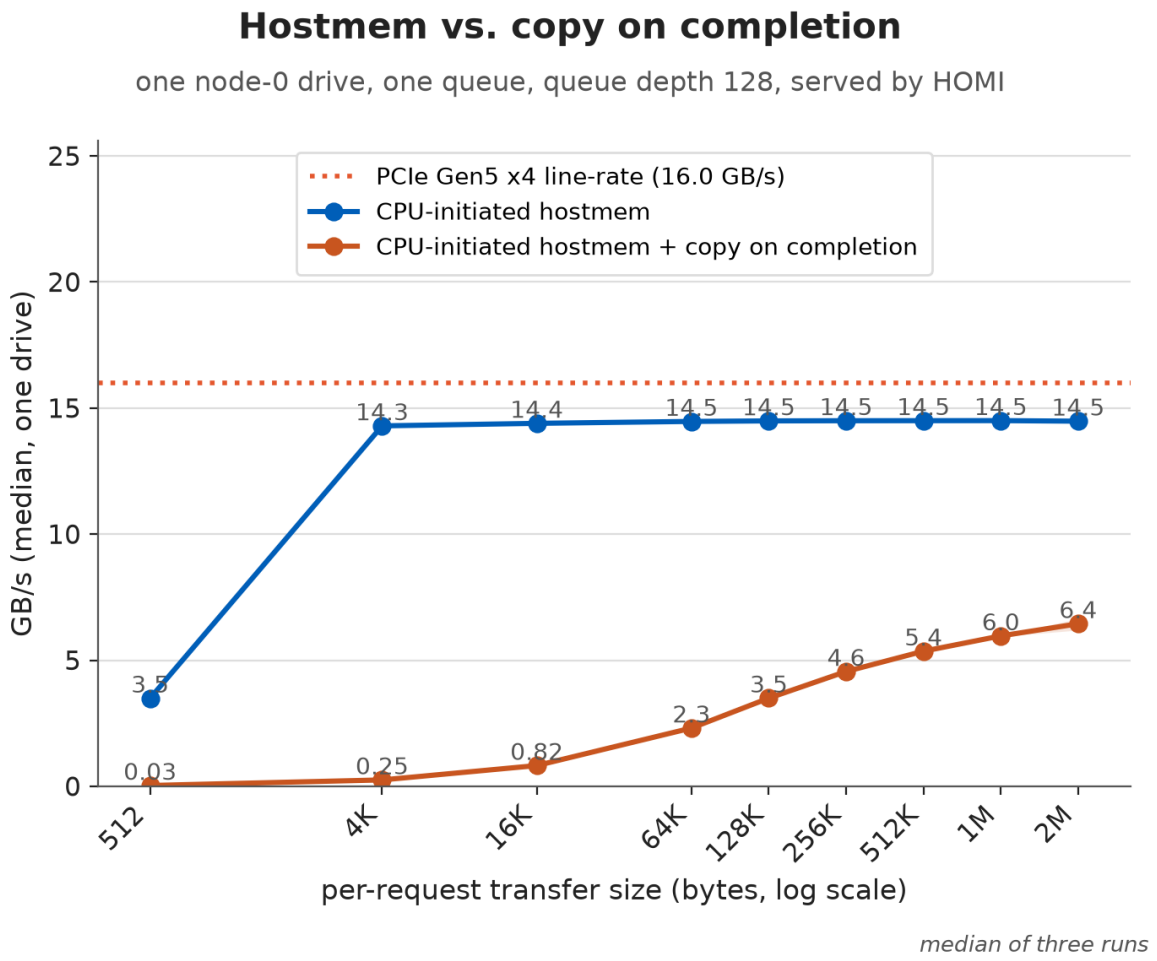


kernel measured to 16 cores, linear, extrapolated to the cap; SPDK v26.05, Dell R760, sixteen Gen6 NVMe SSDs, 3.0 GHz

 ©2026 SNIA Developer Conference. All Rights Reserved.

Extra: hostmem vs. copy on completion, per drive

one node-0 drive, one queue, queue depth 128, served by HOMI



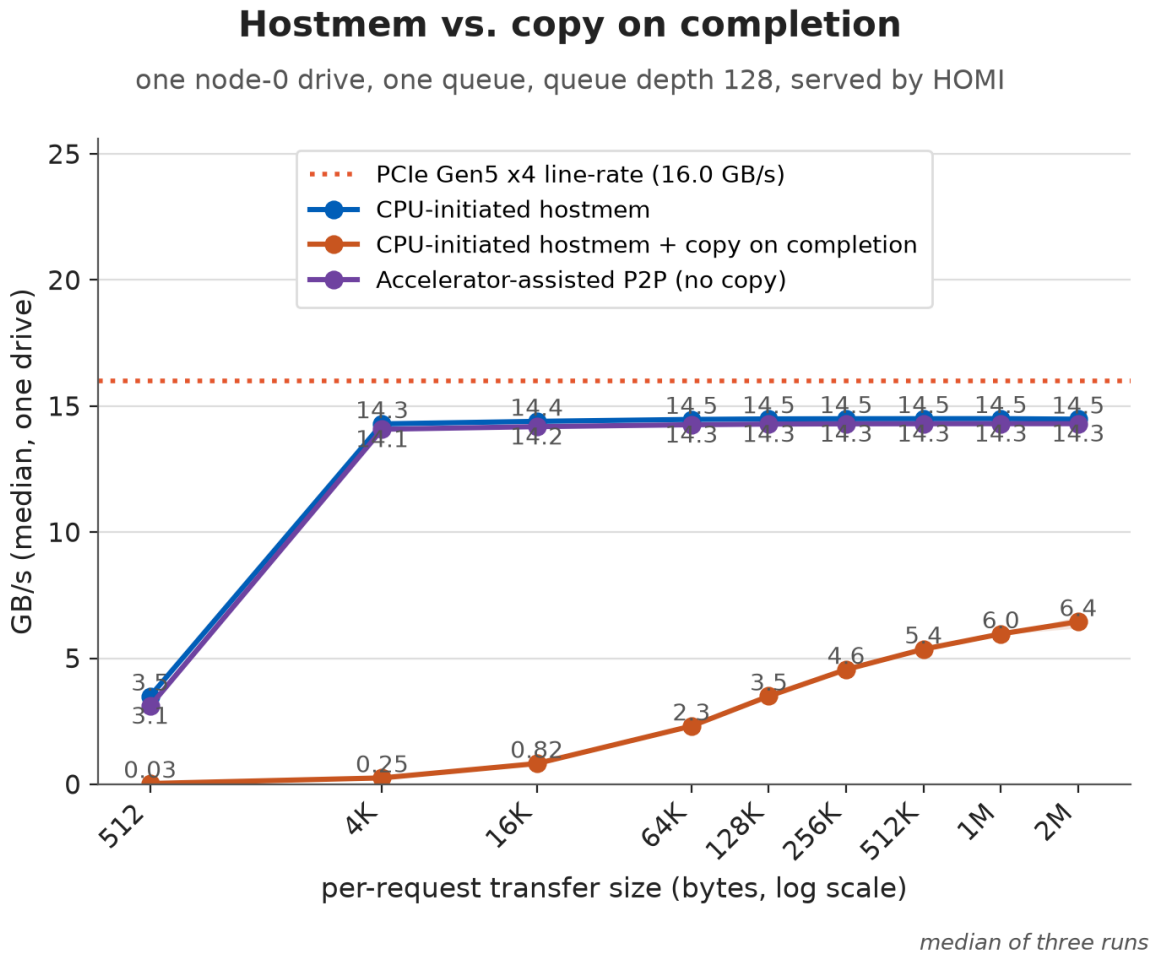
- **What this measures:** hostmem alone against the same path plus a CUDA HtoD copy on completion
- **Hostmem:** reaches the PCIe Gen5 x4 link's own ceiling almost immediately, ~14.5 GB/s from 4 KiB up, flat from there (Gen6 drives, Gen5 platform)
- **Copy on completion:** costs real bandwidth throughout, 6.4 GB/s at 2 MiB, under half of hostmem's own ceiling
- **What P2P avoids:** landing the payload straight in VRAM, nothing to batch up

Does P2P hold this ceiling past one drive? See "Extra: one drive, the per-drive ceiling" and the three slides after it.

 ©2026 SNIA Developer Conference. All Rights Reserved.

Extra: Accelerator-integrated Storage I/O modes

I/O Mode: Accelerator-assisted P2P



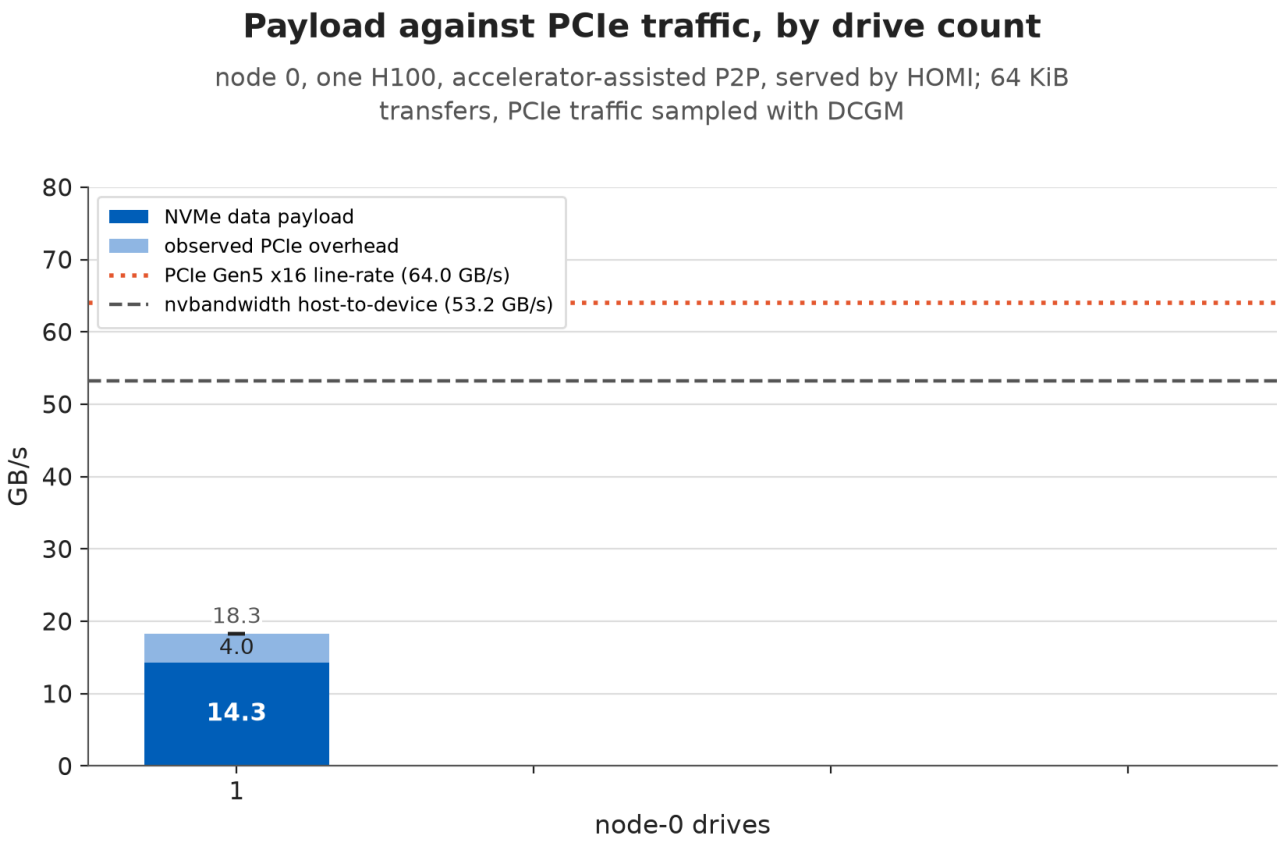
median of three runs

- **A closer look** at the host-bounce penalty
- **On hostmem's own line:** 14.3 GB/s from 4 KiB up, within 1-2% of hostmem at every size, no copy penalty
- **Genuinely single-drive:** measured alone, not sharing the GPU's own link with seven other devices
- **Answers the previous slide:** P2P does not just avoid the copy, it reaches the same ceiling hostmem itself is bound by

 ©2026 SNIA Developer Conference. All Rights Reserved.

Extra: one drive, the per-drive ceiling

one H100, accelerator-assisted P2P via HOMI, 64 KiB; PCIe traffic from DCGM



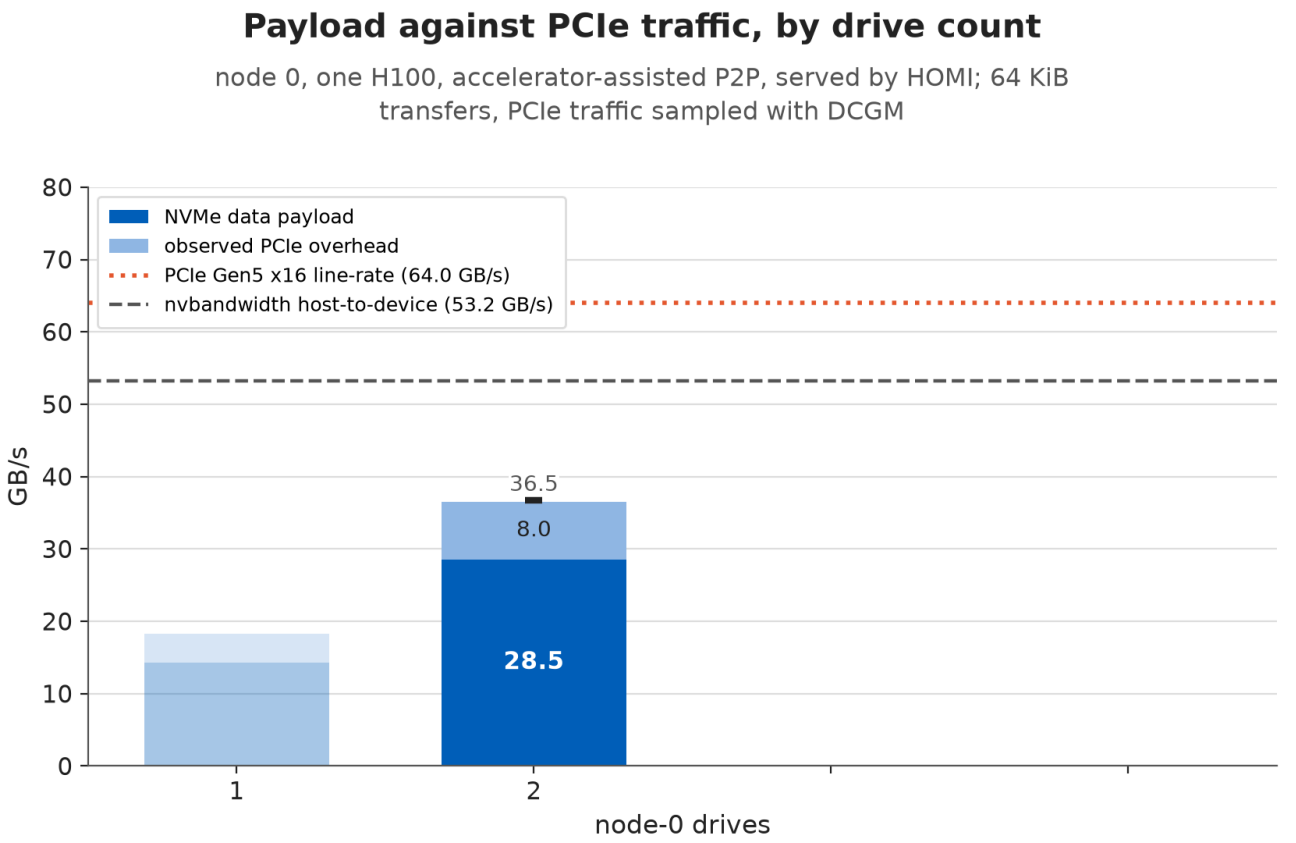
median of three runs; 64 KiB transfers

- **14.3 GB/s payload,** 18.3 GB/s total PCIe traffic
- Matches the per-drive ceiling the previous slide just showed

 ©2026 SNIA Developer Conference. All Rights Reserved.

Extra: two drives, linear so far

one H100, accelerator-assisted P2P via HOMI, 64 KiB; PCIe traffic from DCGM



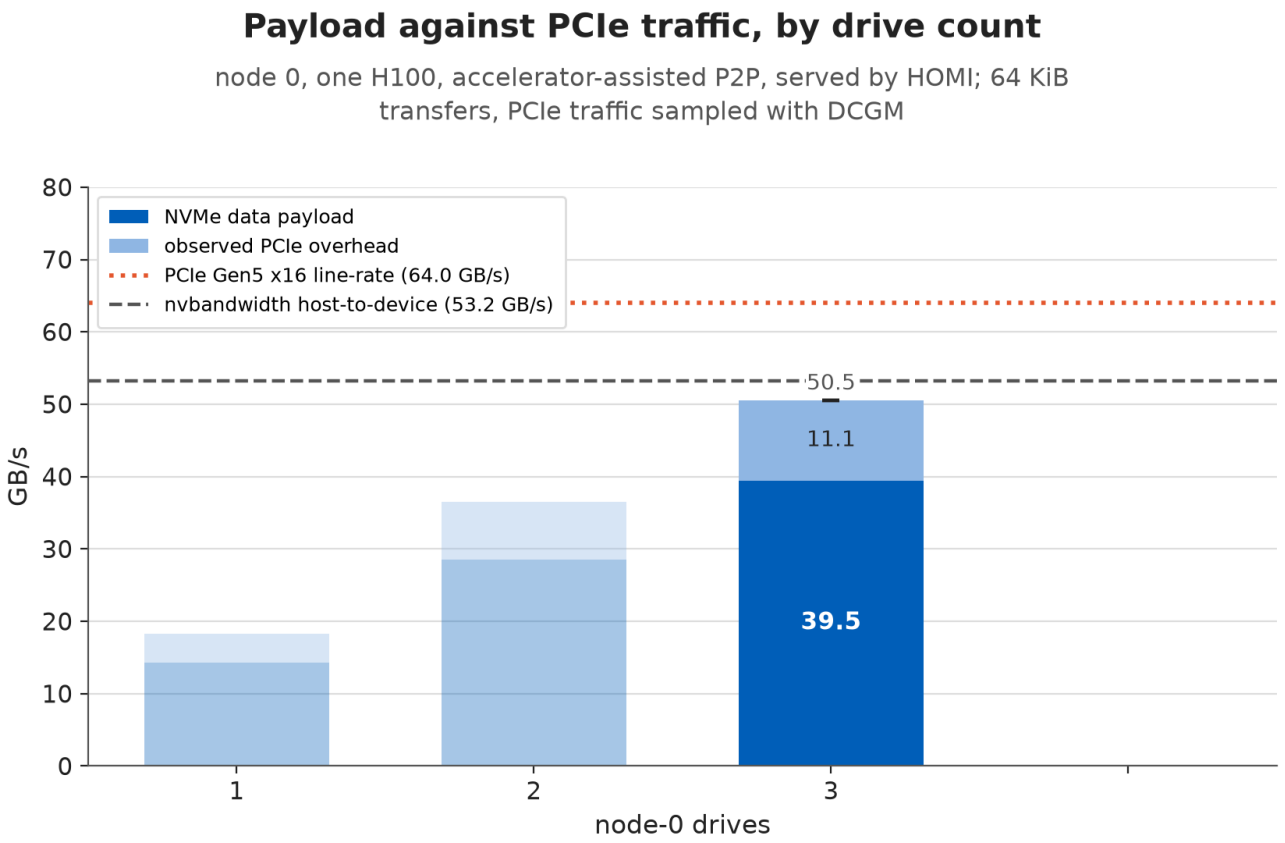
median of three runs; 64 KiB transfers

- **28.5 GB/s payload,** 36.5 GB/s total PCIe traffic; doubles cleanly

 ©2026 SNIA Developer Conference. All Rights Reserved.

Extra: three drives, hits the roof

one H100, accelerator-assisted P2P via HOMI, 64 KiB; PCIe traffic from DCGM

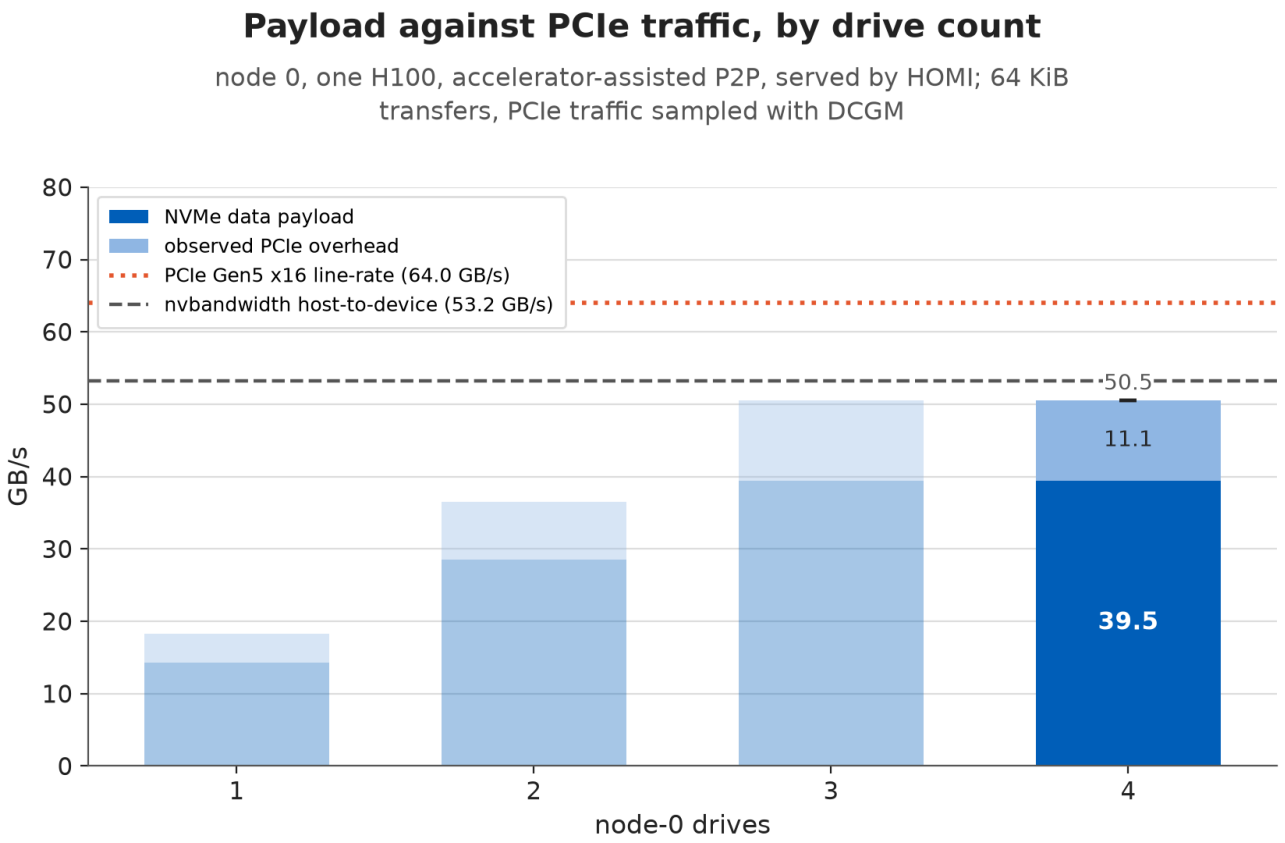


- **39.5 GB/s payload,** 50.5 GB/s total PCIe traffic; 95% of nvbandwidth's independently measured 53.2 GB/s host-to-device ceiling

 ©2026 SNIA Developer Conference. All Rights Reserved.

Extra: four drives, the theory was close, not exact

one H100, accelerator-assisted P2P via HOMI, 64 KiB; PCIe traffic from DCGM



median of three runs; 64 KiB transfers

- **No gain at four:** the roof was already hit at three; the gap under nvbandwidth's raw-transfer roofline is NVMe's own overhead

 ©2026 SNIA Developer Conference. All Rights Reserved.