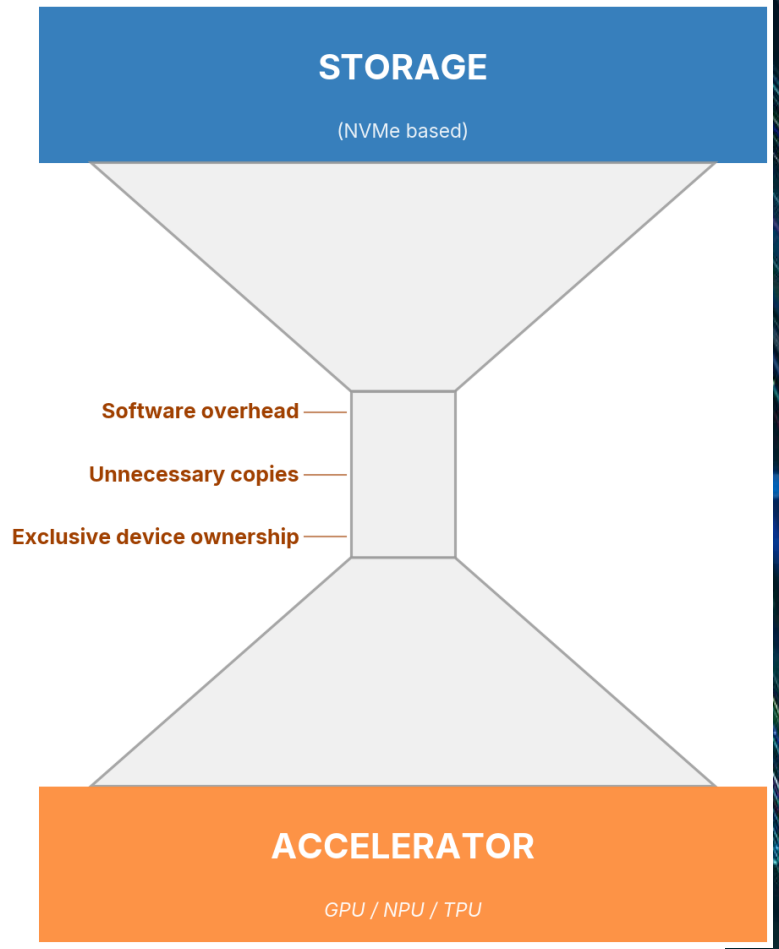

SDC StorageAI™
A SNIA Event
April 29, 2026 • Denver, Colorado
AiSIO
Orchestrating Storage I/O
Across CPUs and Accelerators
Simon A. F. Lund / Samsung

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.



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
- Three I/O modes:
  - **CPU-initiated**: 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
  - **Device-initiated P2P**: 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

Four Questions This Talk Answers

Roadmap for the presentation

- Q1: Where did we start?
  - The AiSIO proof-of-concept (PoC) and its limitations
- Q2: What are we building instead?
  - The AiSIO open source implementation, Host-orchestrated Multipath I/O, and the component stack
- Q3: Does it actually perform?
  - CPU-initiated baseline, software-overhead evaluation, P2P bandwidth, device-initiated I/O
- Q4: Where do we go from here?
  - From PoC to open platform with SR-IOV, ublk, and beyond

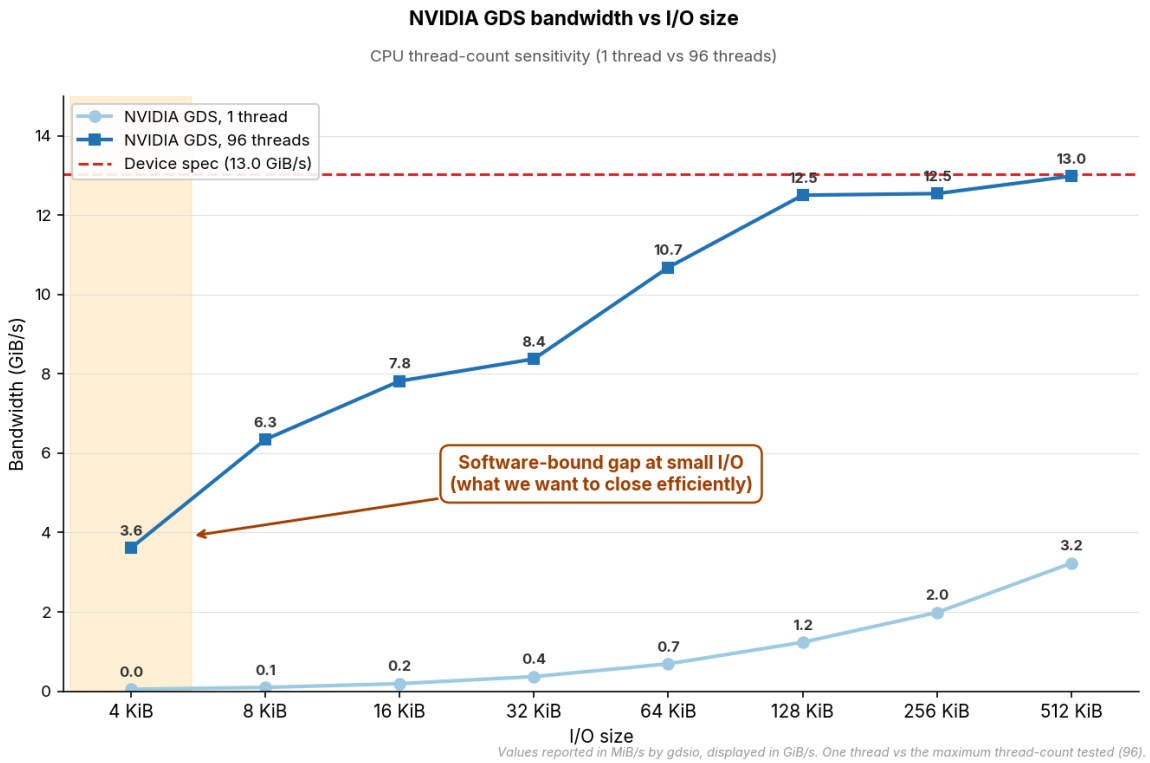
Existing and Emerging Approaches

Each approach reflects different design priorities

- NVIDIA GPUDirect Storage (GDS)
  - CPU-initiated I/O with P2P DMA to GPU memory; removes host-DRAM bounce
  - Vendor-maintained and widely deployed in industry
- BaM (Big Accelerator Memory) - NVIDIA Research
  - Device-initiated I/O from GPU
  - Operates with exclusive device ownership and out-of-tree kernel modules
- NVIDIA SCADA (SCaled Accelerated Data Access)
  - Announced direction for device-initiated storage access
  - Specification and implementation not yet publicly available
- AMD ROCm-XIO (Accelerator-initiated I/O)
  - Device-initiated I/O from GPU; covers NVMe, RDMA, and SDMA paths
  - Open source on GitHub; early-access stage (released April 2026)
- **AiSIO** (this talk)
  - CPU-initiated, CPU-initiated P2P, and Device-initiated P2P
  - Interoperates with the existing stack; open source, targeting upstream

PoC Goal: Close the Performance Gap

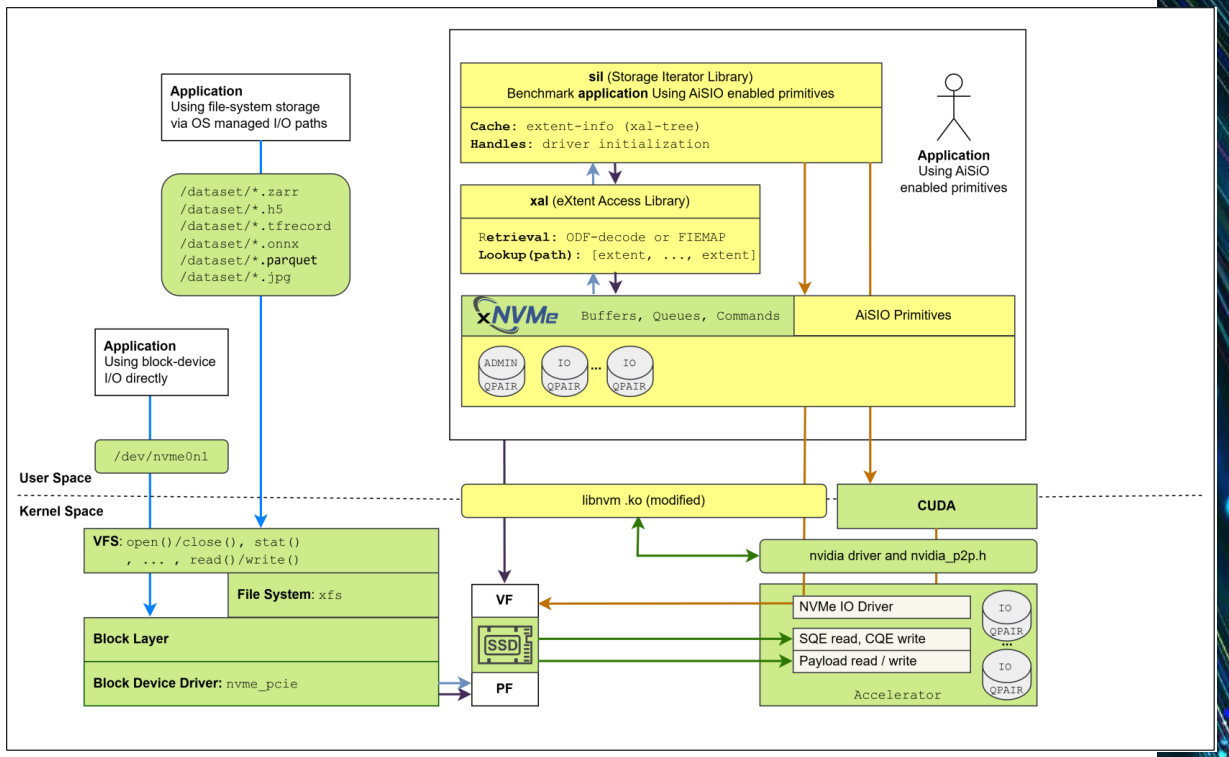
- Observed that at the maximum of 96 threads, small I/O leaves device bandwidth utilization far below capability
  - 4 KiB: ~3.6 GiB/s vs ~13 GiB/s device spec (~27% utilization)
  - Throwing more CPU threads at it stops scaling well before the roofline
- Scaling I/O size **closes the gap**, but it is not the fix we want
  - Workloads do not always have large, sequential I/O to give
  - Larger I/O hides software cost; it does not remove it



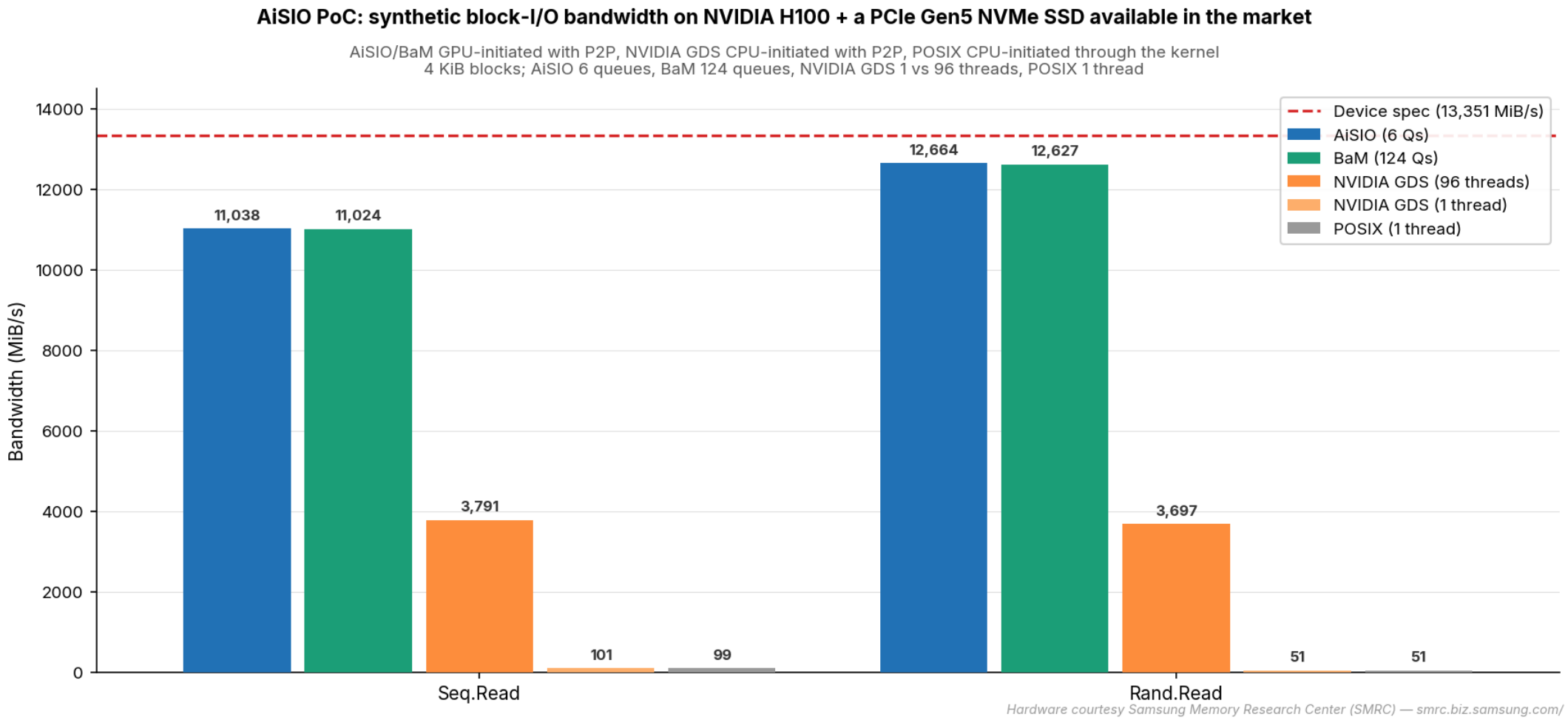
The gap on the small-I/O side is software overhead, not hardware.
Close it by removing per-I/O software cost rather than by enlarging requests.

PoC Stack: Built on BaM and libnvm

- Built on **BaM** (paper | github) a fork of **libnvm** (paper | github) and prework described in paper
- Reproduced from BaM:
  - **Device-initiated I/O**: GPU drives storage commands directly, not the CPU
  - **Small-I/O performance**: dramatically higher bandwidth and IOPS than the conventional path
- Extending BaM with two contributions:
  - **File-backed access**: GPU performs I/O on the data backing files
  - **Coexistence**: conventional I/O path and applications relying on it remain available, demonstrating interoperability
- Demo uses NVIDIA V100/H100 and Samsung SSDs
  - Presented at FMS'25, demoed at OCP'25
  - Hardware courtesy of Samsung Memory Research Center (SMRC)

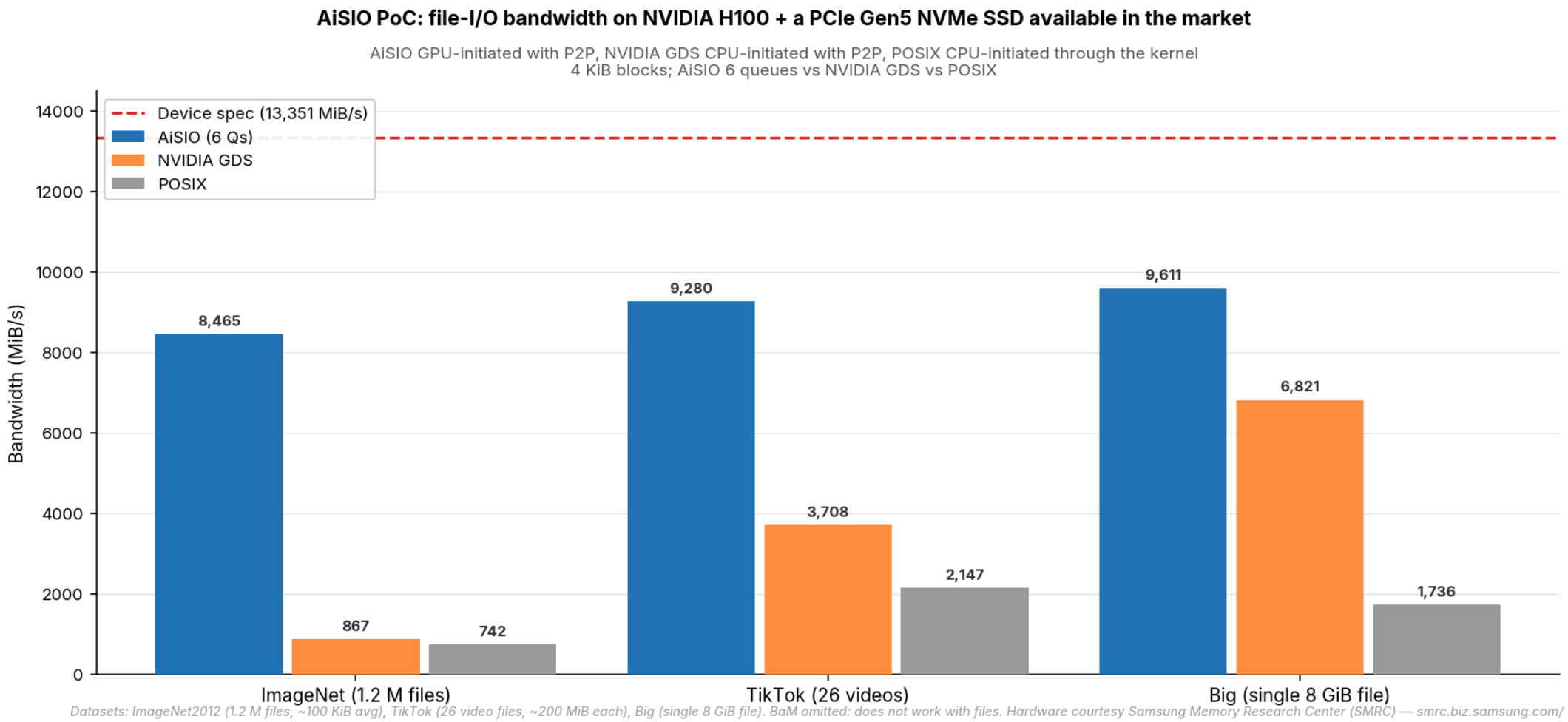


PoC Results: Synthetic Block I/O



SDC | StorageAI™

PoC Results: File I/O



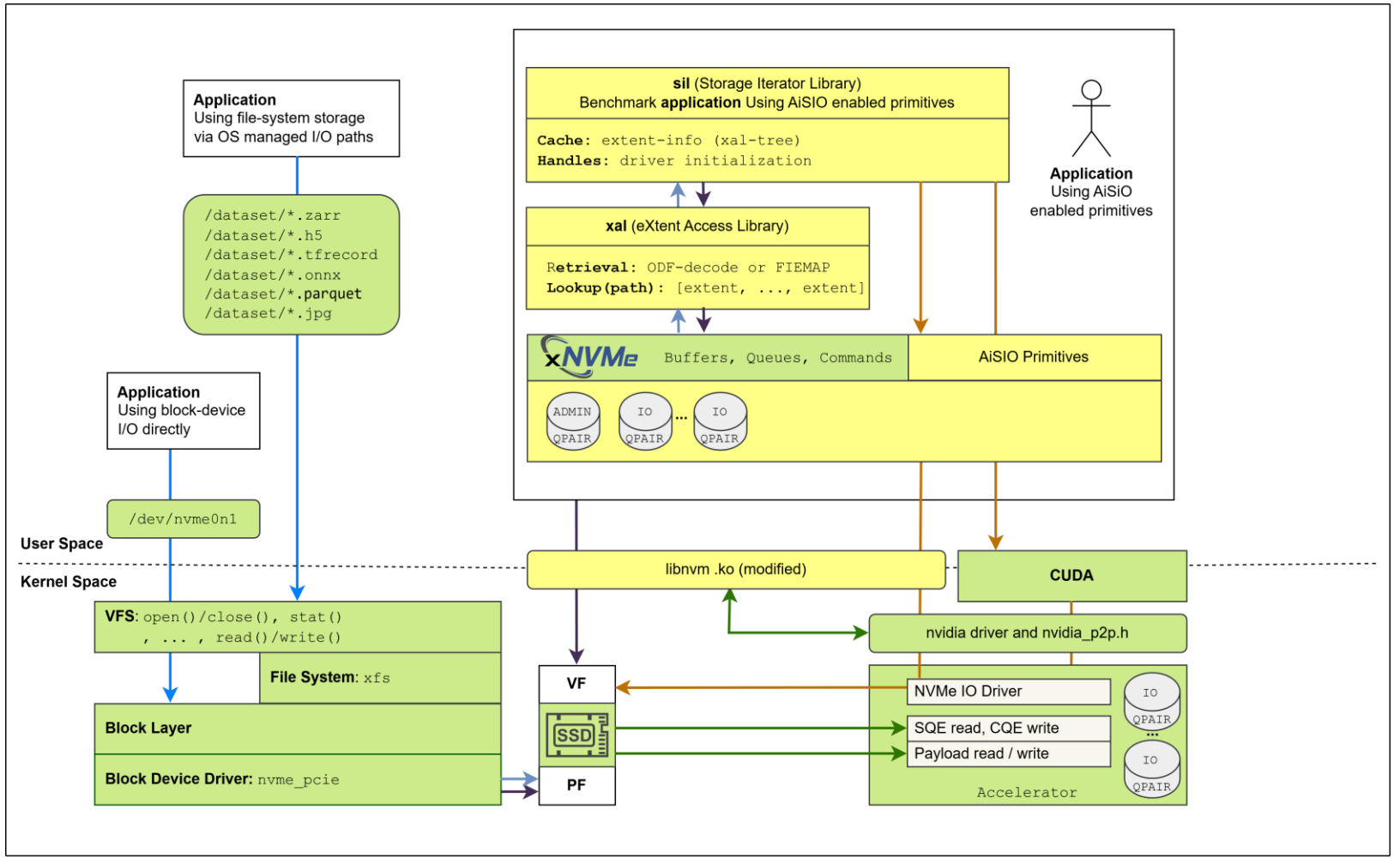
PoC Stack: Limitations of the implementation

- libnvm: predates the current xNVMe / uPCIe stack; not maintained for the configurations we target
- libnvm/BaM: research-stage codebase, not under active development
- Relies on non-upstream, NVIDIA-only interfaces
- Requires custom out-of-tree kernel modules; not on a path to upstream
- Exclusive device ownership: no OS or multipath coexistence
- Limited to the device-initiated path only
- xal/sil: per-process file-extent cache, no cross-process sharing
- xal: XFS-only; FIEMAP ioctl would cover all file systems with one implementation
- xal: no notification path for file-system changes (create/resize/truncate)

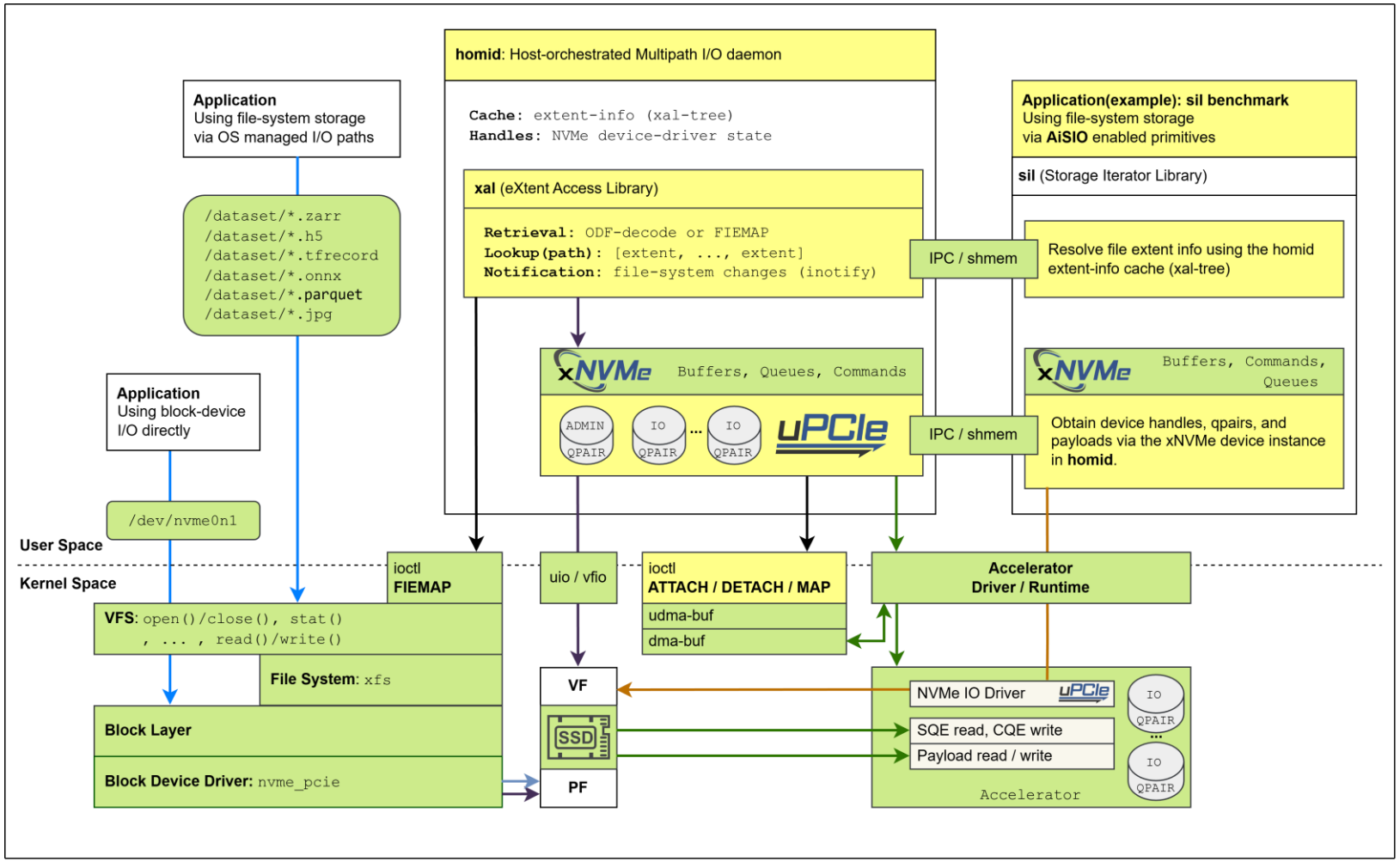
AiSIO: From PoC to Open Source Components

- Replace proprietary components with upstreamable infrastructure:
  - libnvm → **uPCIe**
  - custom kernel patches → **udmabuf-import**
  - BaM exclusive ownership → **HOMI** (host-orchestrated multipath I/O)
- Two paths to NVMe sharing, both managed by **HOMI**:
  - **SR-IOV** (hardware-based) – requires a VF-capable controller
  - **ublk** (software-based) – works with any NVMe device
- Expands from device-initiated only to all three I/O modes
  - Existing file-system software continues to function unmodified

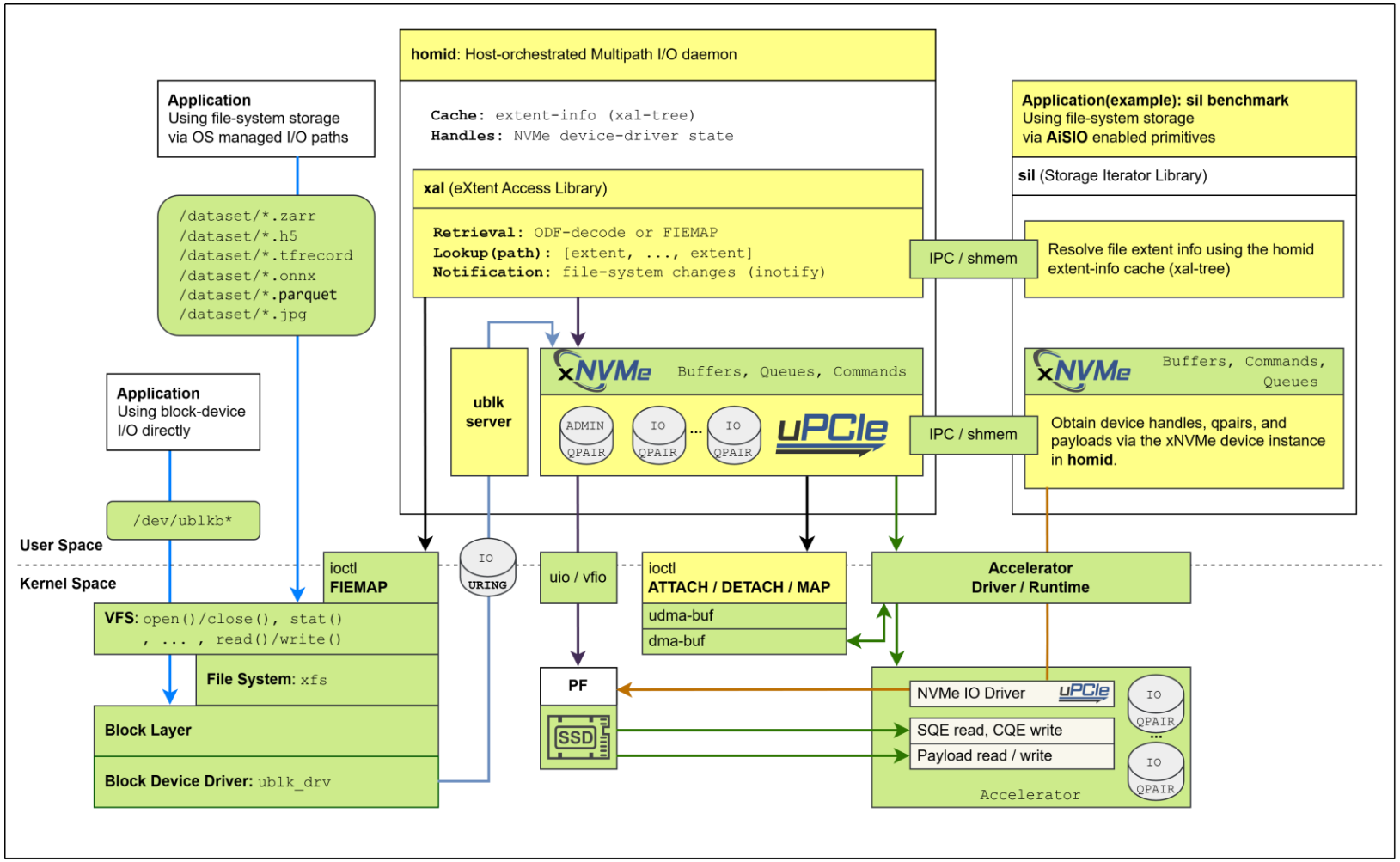
PoC Architecture: Recap



HOMI daemon using SR-IOV



HOMI daemon using ublk



The Component Stack

- **xNVMe** - user-space NVMe abstraction layer
- **xnvmeperf** - NVMe command-level benchmark across xNVMe backends
- **uPCIe** - header-only, zero-dependency PCIe driver library
- **udmabuf-import** - kernel module enabling P2P DMA buffer import (targeting upstream)
- **XAL** - eXtent access Library for efficient file-to-block mapping with/without the kernel
- **fil / filperf** - unified file-I/O library and benchmark across backends
- **HOMI** - host-orchestrated multipath I/O daemon
- **cijoe** - reproducible workflow automation
- **ublk server** - software NVMe multipath for kernel block-device coexistence

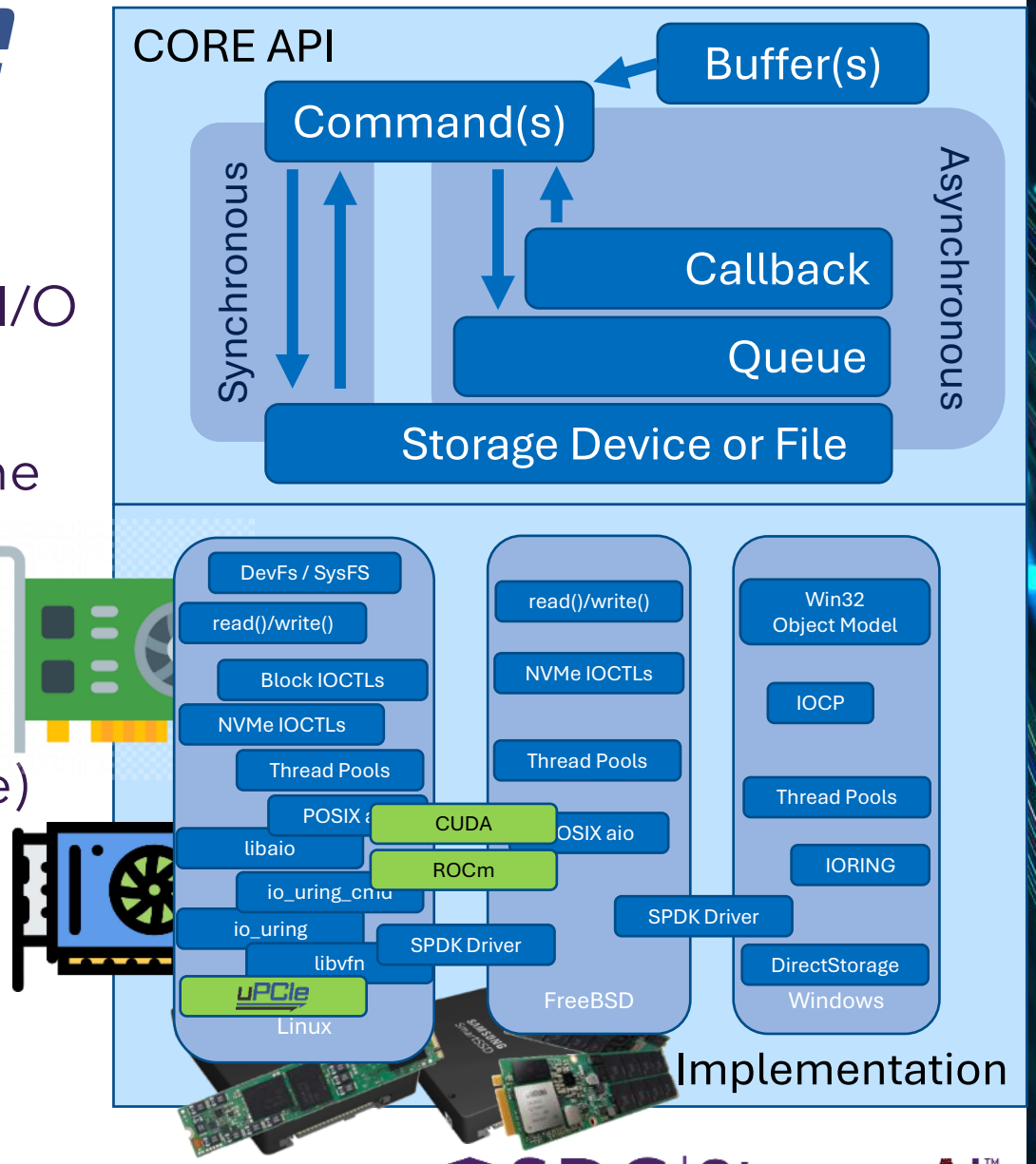
Components:

User space abstraction layer and unified API for I/O

Allowing application IO to be re-routed at runtime with minimal overhead.

- Expanded with **AiSIO** primitives (API)
- Expanded with **AiSIO** I/O paths (driver/runtime)

- Website: xnvme.io
- Source: github.com/xnvme/xnvme



SDC | StorageAI™

Component: xnvmeperf

- Multi-threaded async I/O benchmark for NVMe devices, built on xNVMe
- **xnvmeperf**: exercise I/O for a fixed duration, reports IOPS and throughput
  - --iopattern: read, write, randread, randwrite
  - --nqueues: queues per device (optional)
  - --qdepth: queue max capacity
  - --iosize: payload size in bytes
  - --runtime: duration in seconds
  - --cpumask: CPU affinity for worker threads
  - trailing positional arguments: one or more device paths
- Subcommands
  - **run** / **verify** (CPU-initiated)
  - **cuda-run** / **cuda-verify** (GPU device-initiated)
- Docs: https://xnvme.io/en/next/tools/xnvmeperf
- Source: https://github.com/xnvme/xnvme/blob/next/tools

`xnvmeperf` demonstrated on aPCIe Gen4 Development box, for CPU-initiated I/O it uses 1HWT on 1 CPU Core of a 32 Core / 64HWT EPYC 7532. Device-initiated I/O using P2P memory access to an RTX A6000. Total of 8 NVMe storage devices with combined 12.5M IOPS @ 512 using 4 capable 1.15M IOPS @ 512 and 4 capable of 1.95M IOPS. The single HWT still has cycles to spare.

Components: uPCIe - User-space PCIe Primitives

Problem

There is no minimal / clean user-space library layer for interacting with GPU drivers and runtimes to perform **mmio** and **dma**

What it is now

- C **Header-only** Library
- Driver binding and hugepage **tools**

- UIO-based device access
- **MMIO** + **DMA** using host-resident memory
- Explicit BAR mapping
- DMA memory management (via hugepages)

- **Minimalistic** CPU-initiated NVMe driver
- Open source at https://github.com/safl/upcie

No framework semantics, no policy,
no hidden execution model

Direction (Must Have)

- Extend MMIO + DMA to accelerators (GPUs, DPUs, others)
- Support device-resident memory + device-initiated I/O
- Keep CPU + host-memory path as a reference baseline

Direction (Nice to Have)

- Transition UIO → vfio-pci
- Adopt iommu-fd for explicit, fine-grained IOMMU control

Component: udmabuf-import

- Patch to the Linux **udmabuf** driver that adds a dma-buf **importer** role
  - Resolves physical addresses of device memory (e.g. GPU BAR1) from user space
  - Not specific to CUDA or NVIDIA; works with any dma-buf exporter
- Three new ioctls: **UDMABUF_ATTACH**, **UDMABUF_GET_MAP**, **UDMABUF_DETACH**
  - ATTACH imports the dma-buf fd and returns a page count
  - GET_MAP returns an array of (dma_addr, len) tuples
- Used by uPCIe to map GPU device memory for NVMe P2P I/O
  - Enables NVMe Payloads in GPU device memory (CPU-initiated P2P and Device-initiated P2P)
  - Plus payload support structures (PRP/SGL lists) and queue pairs in GPU device memory (Device-initiated P2P)
- Out-of-tree today; targeting upstream to remove custom-kernel requirement
- Source: https://github.com/xnvme/udmabuf-import

Component: XAL

- **eXtent Access Library**: backend-agnostic file-to-block mapping in C
- Same API (**xal_open / xal_index / xal_get_extents**) across backends:
  - **XFS** backend: parses the XFS on-disk format directly from the block device, no mount required
  - **FIEMAP** backend: kernel **FS_IOC_FIEMAP** ioctl on a mounted file system
  - **FIEMAP + eBPF** backend: bundling notification and the updated extent information
  - Sane defaults, backend is selected based on **FS** and whether the device appears in /proc/mounts
- FIEMAP backend supports inotify-driven watch modes
  - DIRTY_DETECTION marks the index stale; EXTENT_UPDATE refreshes extents in place via seq_lock
- Lookup modes: tree traversal (binary search per level) or O(1) hash map
- Enables file-backed I/O on all three AiSIO paths – including from user-space drivers and accelerators with no OS on the data path
- Source: https://github.com/xnvme/xal

Component: fil / filperf

- **fil** (File Iterator Library): unified file-I/O abstraction across backends
  - Originally **SIL** (Storage Iterator Library) in the PoC
  - Iterates over file-based datasets using XAL for extent resolution
- Pluggable backends: **posix**, **aisio-cpu**, **aisio-gpu** (WIP), **gds**
  - Same application code runs across kernel, CPU-initiated P2P, and GPU-initiated paths
- **filperf**: benchmark harness built on fil
  - Drives synthetic and realistic workloads (imagenetish, tiktokish, filesize8gib)
  - Used to compare AiSIO paths against POSIX and GPUDirect Storage
- Source: https://github.com/xnvme/fil

Component: HOMI

- **Host-Orchestrated Multipath I/O**
  reference implementation of the AiSIO control plane
- Host-resident daemon (**homid**) centralizes control-plane responsibilities:
  - Device discovery, bring-up, hotplug, and NVMe admin operations
  - I/O queue-pair provisioning for kernel, user-space, and accelerator consumers
  - File-extent extraction and caching via XAL
- Two realizations of multipath coexistence:
  - **ublk** (software-mediated) – commodity hardware, all logic in user space
  - **SR-IOV** (hardware-assisted) – VF per initiator, isolation in the controller
- Static queue ownership – no software arbitration on the data path
- Scope is deliberate: expose and coordinate paths, not re-invent the file system
- Source: https://github.com/xnvme/aisio/tree/main/homi

Component: ublk server

- Will be an integral part of the **HOMI** daemon
- Linux **ublk**: kernel block-device interface backed by a user-space server
  - Kernel exposes /dev/ublkbN; I/O requests forwarded to user space over io_uring
- AiSIO ublk server is backed by **xNVMe** / **uPCIe**
  - User-space driver owns the NVMe controller end-to-end
  - Provisions separate queue pairs for kernel, user-space, and accelerator consumers
- Enables HOMI coexistence on any NVMe device — no SR-IOV required
  - OS-managed file systems and applications keep working unmodified
  - User-space and device-initiated paths run concurrently on the same controller
- Trade-off vs SR-IOV: more host software, broader hardware compatibility
- **Status: not yet implemented** — architecture defined, work in progress
- Source (in HOMI): https://github.com/xnvme/aisio/tree/main/homi

Components:

- Reproducibility via **CIJOE**
- Seven steps from bare Ubuntu to bare-metal benchmark results:
  - setup_ubuntu - OS preparation
  - setup_udmabuf_import - kernel patch
  - setup_nvstack - NVIDIA drivers, CUDA, MLNX OFED
  - setup_aisio - fetch and build all components
  - bench_io - CPU-initiated I/O characterization
  - bench_tools - tool overhead analysis
  - bench_pcie - P2P bandwidth measurements
- Docs: cijoe.readthedocs.io
  - Tasks: github.com/xnvme/aisio/tree/main/tasks
  - Scripts: github.com/xnvme/aisio/tree/main/scripts
  - Configs: github.com/xnvme/aisio/tree/main/configs

AiSIO as a Project

- A reproducible research platform: open source, automated, deterministic
  - Bare-metal to results in version-controlled workflows
- White-paper: https://xnvme.io/aisio (online version)
- SDK: https://github.com/xnvme/aisio/

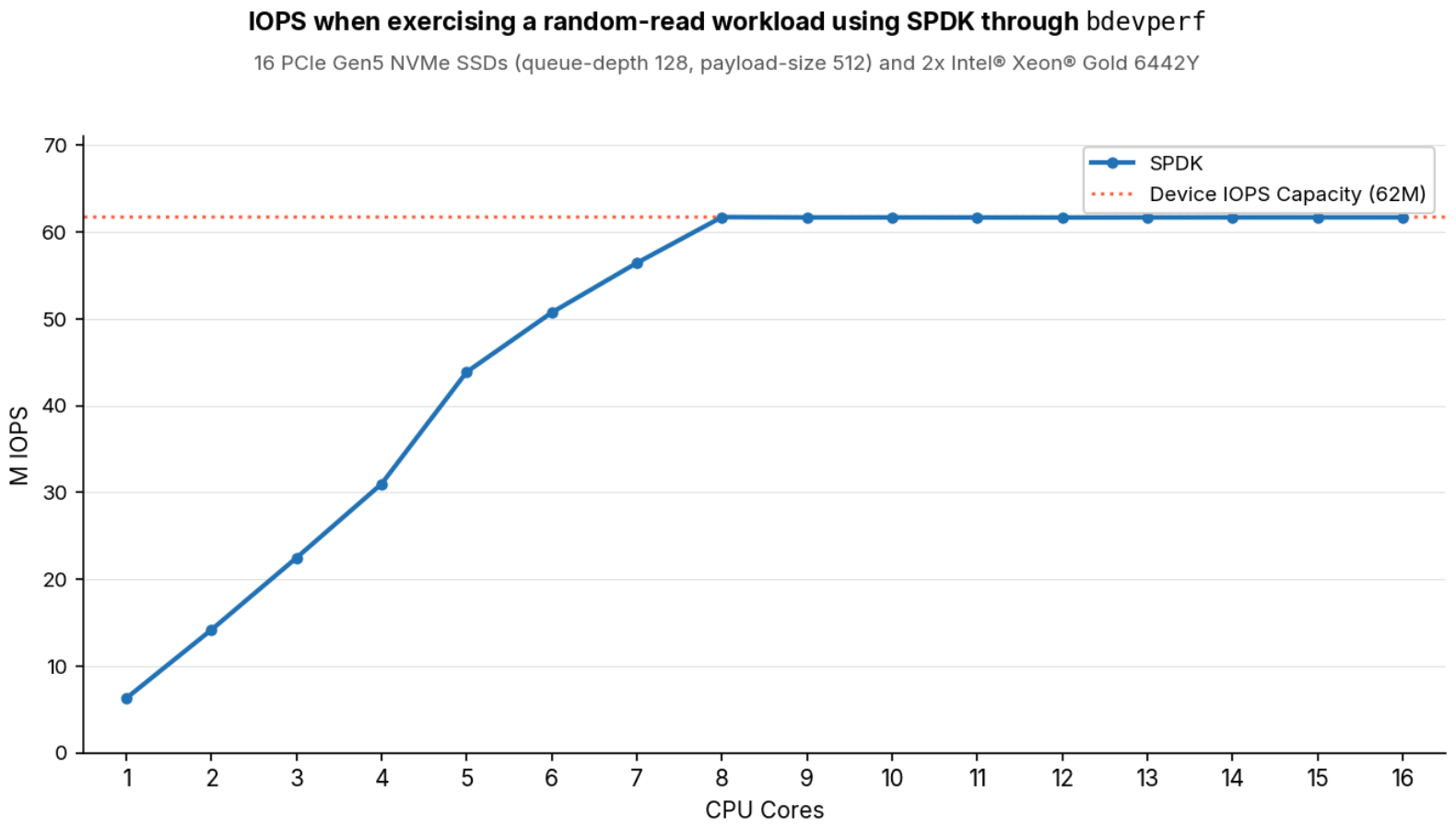
Four experiments evaluating the AiSIO open source implementation end-to-end

- **Experiment 1** – CPU-initiated baseline: what does the hardware allow?
- **Experiment 2** – Minimizing software overhead
- **Experiment 3** – Reaching the accelerator (CPU-initiated P2P, Device-initiated P2P)
- **Experiment 4** – P2P bandwidth scaling across NVMe devices

Experiment 1: CPU-Initiated I/O Characterization

- Setup:
  - 16 NVMe PCIe Gen5 SSDs (per-drive: 3.85M IOPS on unallocated LBAs, 3.3M on allocated; using unallocated)
  - Dell PowerEdge R760
  - 2x Intel® Xeon® Gold 6442Y (48 cores total, 96 HW threads with SMT)
  - SPDK bdevperf, randread 512B
  - Hardware courtesy of Samsung Memory Research Center (SMRC)
- Tuning knobs:
  - CPU governor, turbo boost, SMT
  - Queue depth, core count, device count

SPDK Core Scaling



One HW thread per core, turbo-boost enabled reaching 3.3GHz. NVMe LBAs are unallocated to minimize I/O processing and thereby increase I/O rate.
Hardware courtesy Samsung Memory Research Center (SMRC) — smrc.biz.samsung.com/

Baseline Results & Takeaways

- Near device-rated aggregate capacity: 62M IOPS with 16 devices, 8 cores, QD 128
  - On average; 7.75M IOPS per CPU core
  - Per-core variation visible in the plot, attributed to suboptimal placement / NUMA effects
- Scales linearly with devices and with cores until saturation (**1:2** core-to-device ratio)
- Tuning knobs that matter:
  - CPU governor: 60–70% drop with powersave vs. performance
  - Turbo boost: 3–7% impact, compensated by adding cores
  - Queue depth: sweet spot at QD 128
- CPU-initiated I/O is powerful but CPU-bound: every IOPS costs CPU cycles
- We can saturate **16** NVMe devices with **8** CPU cores,
  but those cores are then unavailable for anything else.

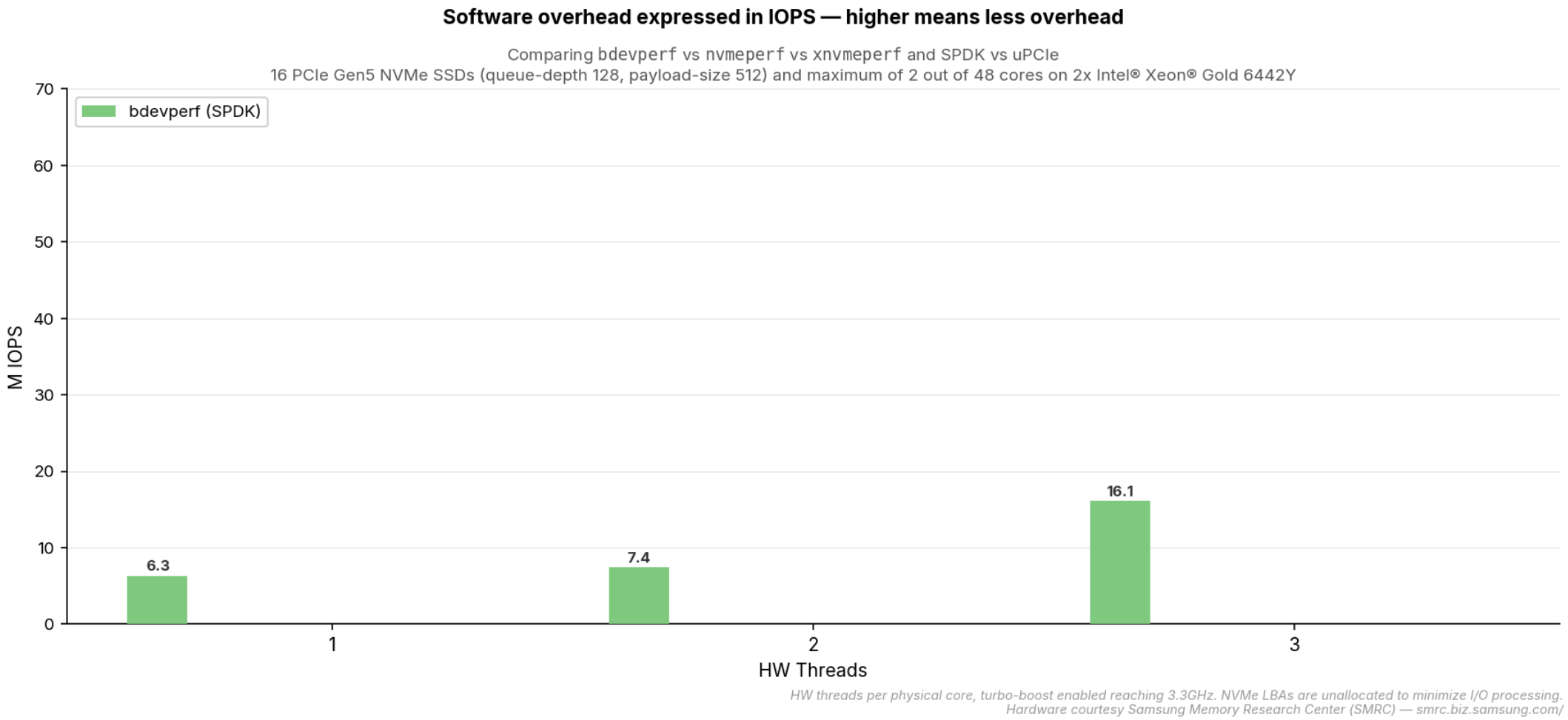
What if we could free them?

Experiment 2: Minimizing Software Overhead

- Goal: isolate software overhead from hardware limits for the CPU-initiated path
- Same hardware as Experiment 1, constrained to a fraction of CPU resources
- Compare four configurations:
  - bdevperf (SPDK)
  - nvmeperf (SPDK)
  - xnvmeperf with SPDK backend
  - xnvmeperf with uPCIe backend
- Metric: IOPS at payload-size 512, queue-depth 128, 16 NVMe SSDs

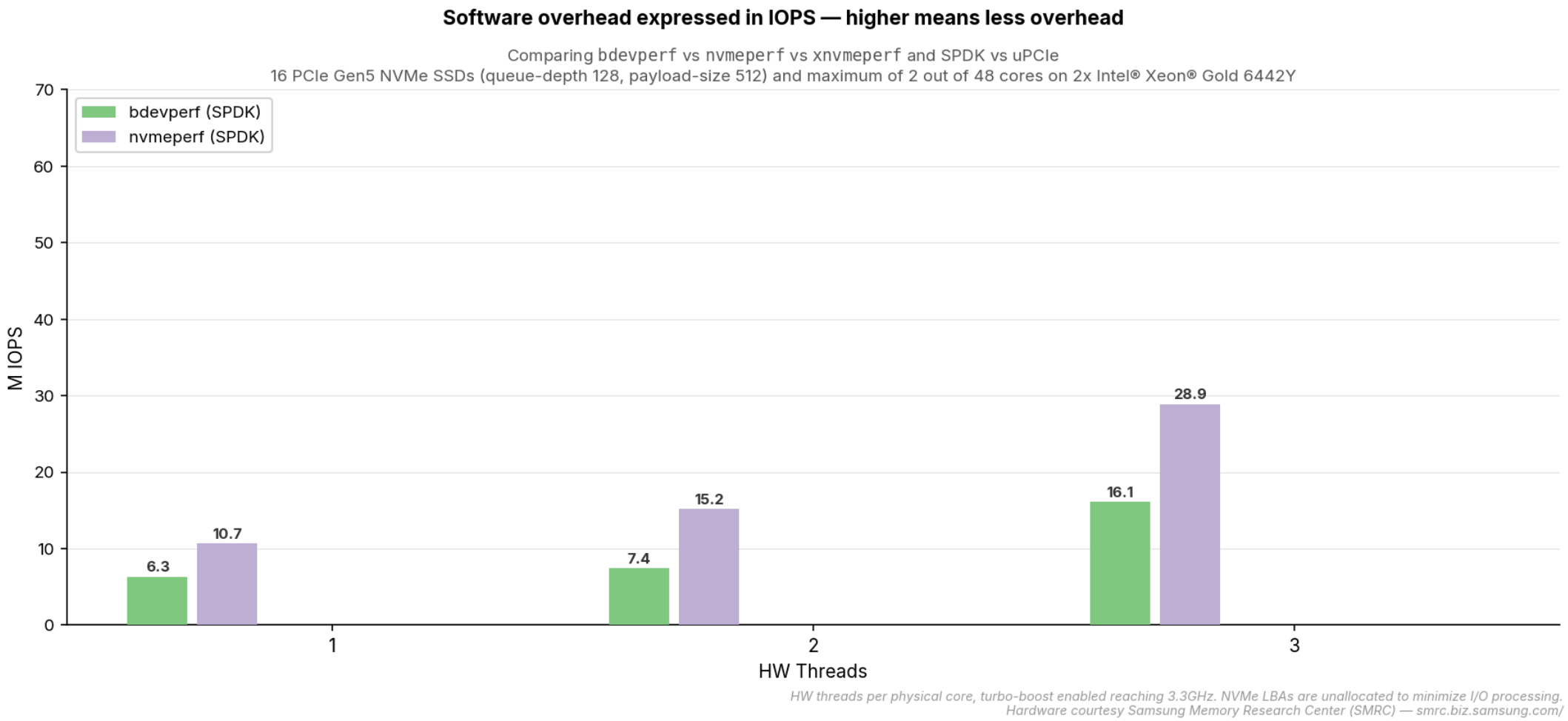
Minimizing Software Overhead (1/4)

bdevperf baseline



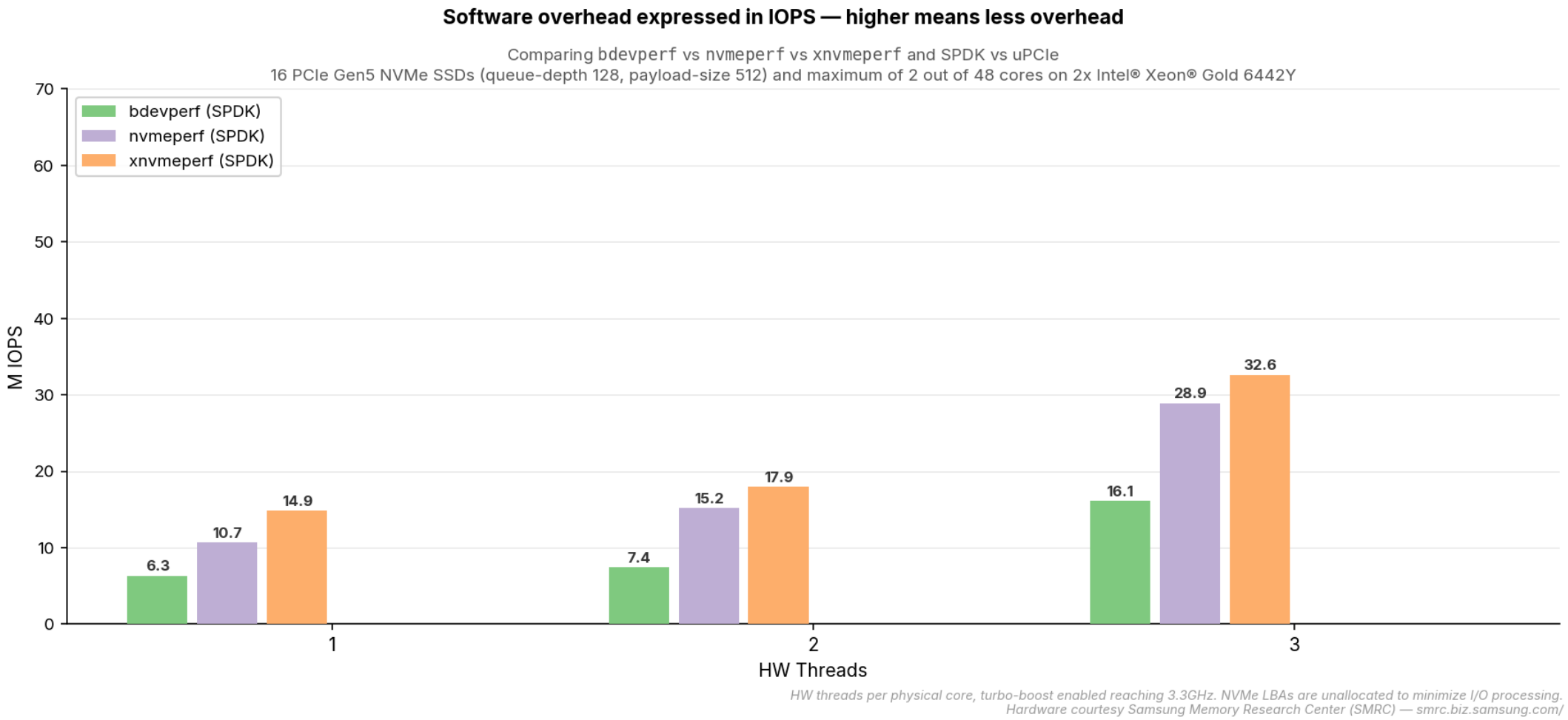
Minimizing Software Overhead (2/4)

+ nvmeperf



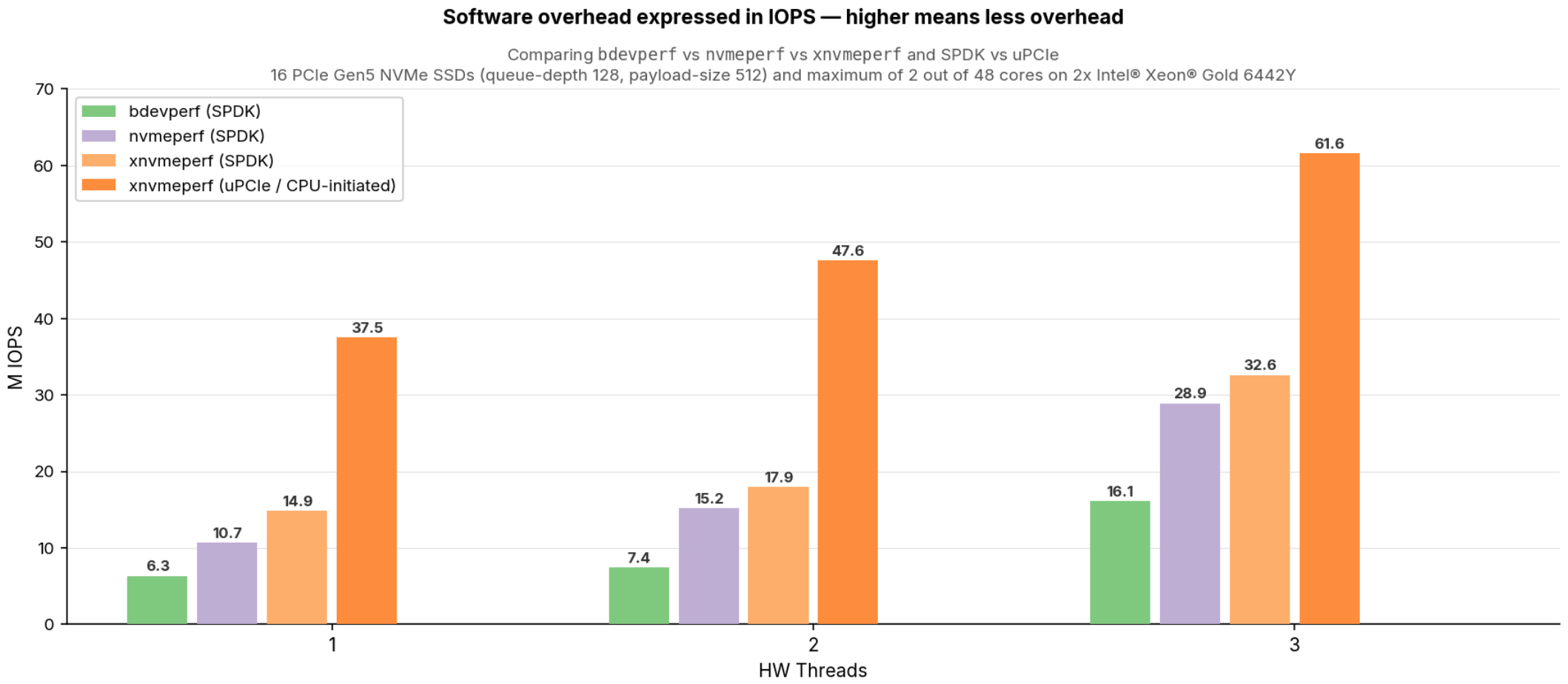
Minimizing Software Overhead (3/4)

+ xnvmeperf (SPDK backend)



Minimizing Software Overhead (4/4)

+ xnvmeperf (uPCIe, CPU-initiated)



HW threads per physical core, turbo-boost enabled reaching 3.3GHz. NVMe LBAs are unallocated to minimize I/O processing.
Hardware courtesy Samsung Memory Research Center (SMRC) — smrc.biz.samsung.com/

Experiment 2: Takeaways

- Single HW thread, 16 devices:
  - bdevperf (SPDK baseline): **6.3M IOPS**
  - xnvmeperf/uPCIe: **37.5M IOPS** – **~6×** improvement
- Scaling on a single CPU:
  - 2 HW threads (using 1 of 24 cores on one CPU): **47.6M IOPS**
  - 3 HW threads (using 1.5 of 24 cores on one CPU): **~61.5M IOPS**
    Saturates all 16 drives; backplane has no room for more
    Expecting linear scaling with more drives: 4 HW threads (2 of 24 cores) should reach **~96M IOPS**

Overhead is incurred at every layer, affecting the latency of every I/O.
With AiSIO we see the effect of reducing it at the application and driver layers.

Experiment 1: 8 cores for 62M IOPS
Experiment 2: 1.5 cores for 62M IOPS
6.5 cores freed for application work

Experiment 3: Reaching the Accelerator

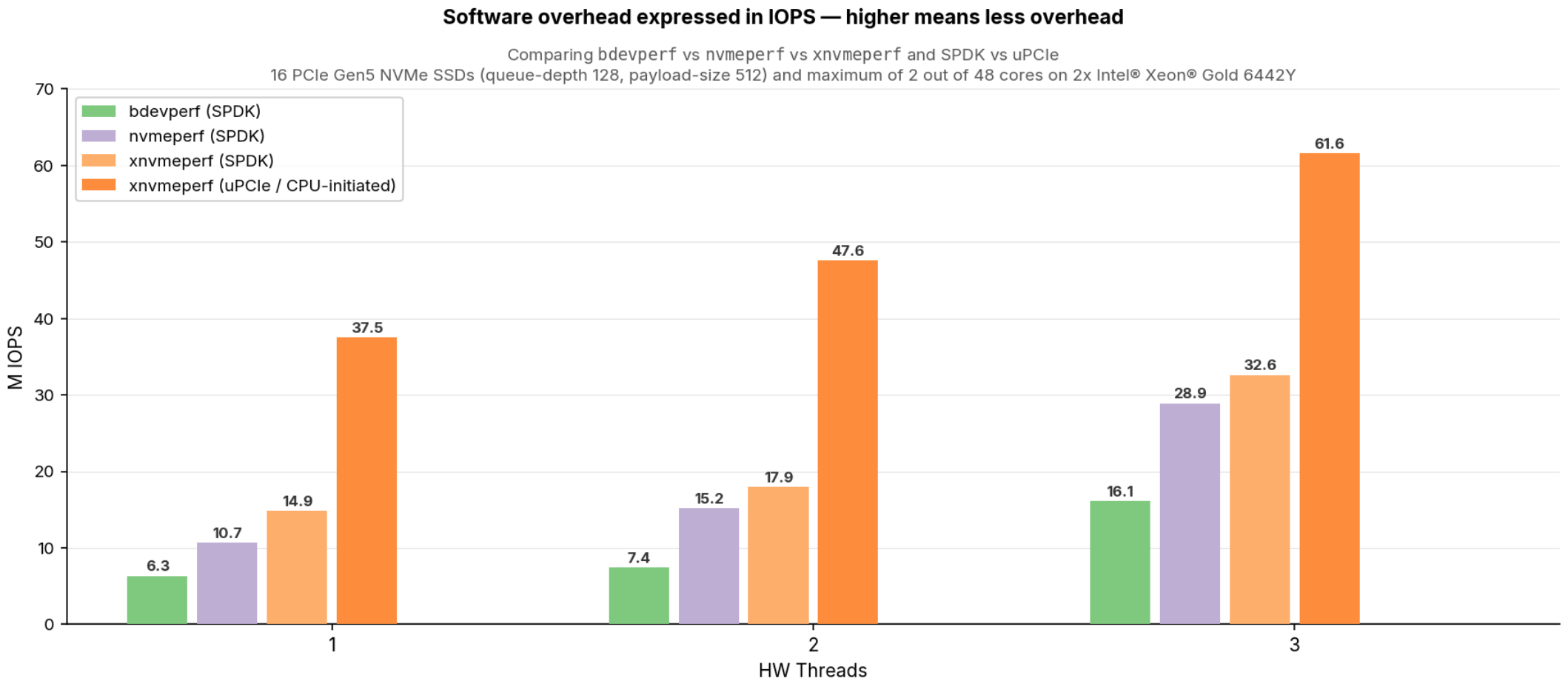
- Software overhead of CPU-initiated I/O is now minimized
- But data lands in host DRAM, not in device memory

How do we reach the accelerator without bouncing through host DRAM?

- Two new paths added to AiSIO and evaluated with xnvmeperf:
  - **CPU-initiated P2P** – CPU initiates, data routes directly to GPU memory via PCIe peer-to-peer DMA
  - **Device-initiated P2P** – accelerator issues NVMe commands itself; data lands directly in its own memory

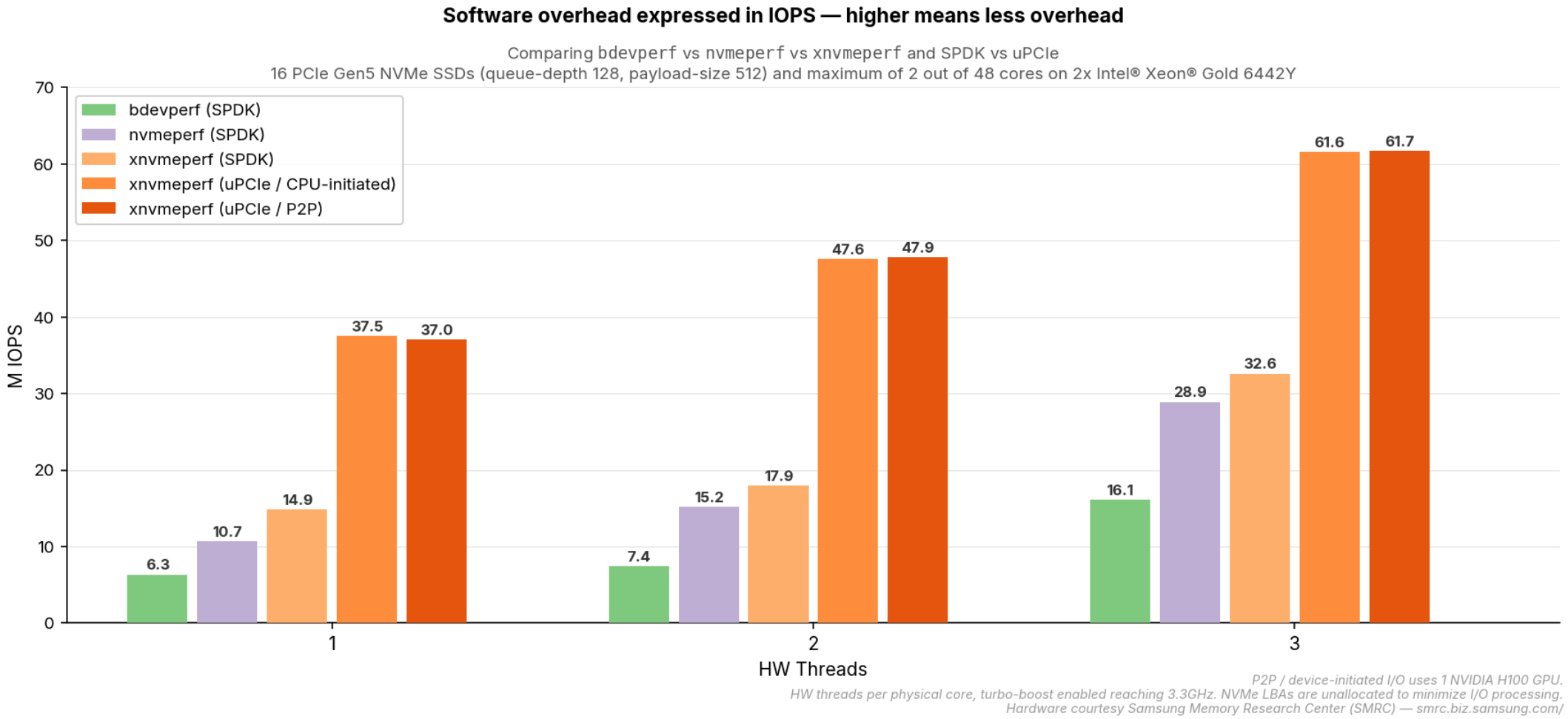
Reaching the Accelerator

Recap: where Experiment 2 left off



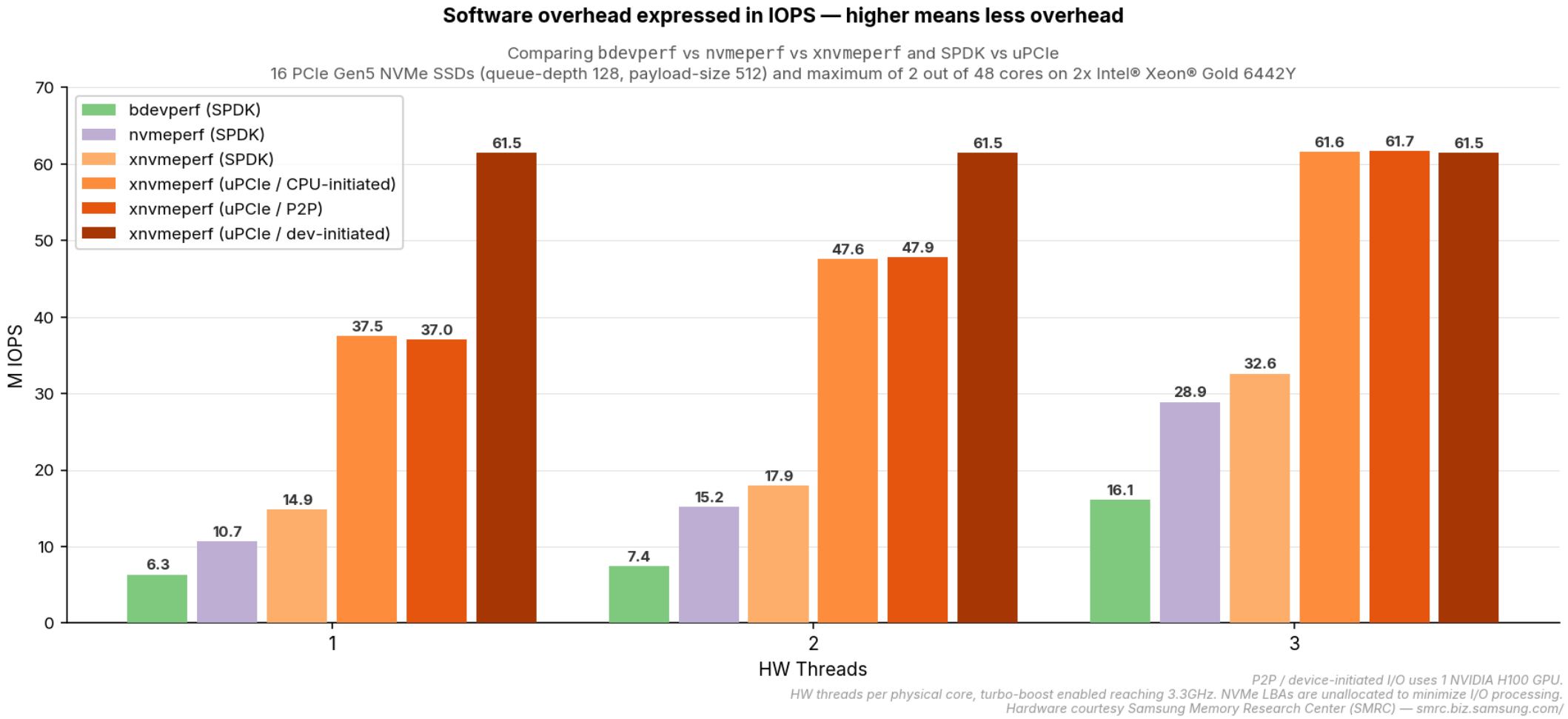
Reaching the Accelerator (1/2)

+ xnvmeperf (uPCIe, CPU-initiated P2P)



Reaching the Accelerator (2/2)

+ xnvmeperf (uPCIe, Device-initiated P2P)



Experiment 3: Takeaways

- Both paths reach the device ceiling of **~61.5M IOPS**
- The difference is what they avoid

- **CPU-initiated P2P**
  - Same CPU cost as routing to host memory
  - No host-memory bounce, no host-bandwidth bottleneck

- **Device-initiated P2P**
  - Removes the CPU from the per-I/O hot path entirely

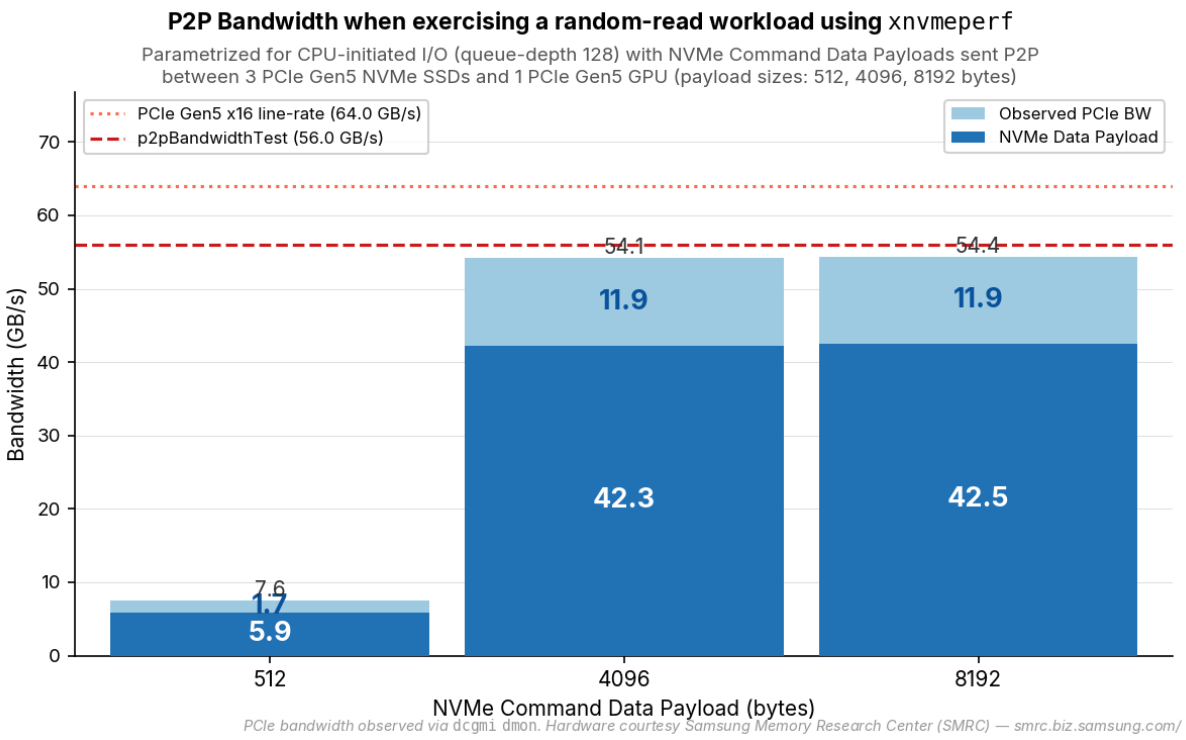
AiSIO delivers data to device memory at no extra CPU cost.
Device-initiated P2P frees the CPU from the per-I/O path entirely.

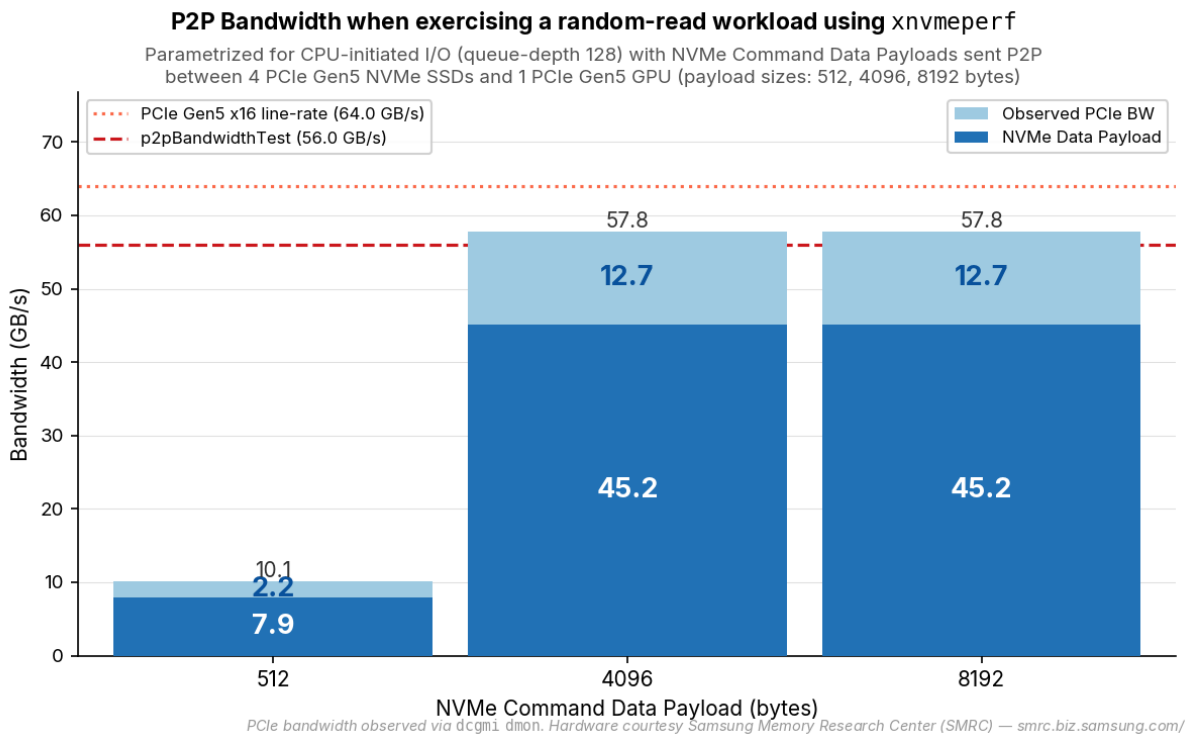
Experiment 4: P2P Bandwidth to GPU

- So far: focus on per-I/O latency and IOPS
- Bandwidth is the primary concern for AI-centric workloads
- Brief look: when can the GPU's PCIe bandwidth be saturated?
  - GPU is PCIe x16; one might assume 4 x4 NVMe SSDs is the right ratio
  - We compare 3 vs 4 NVMe SSDs aggregated to one GPU
- Metric: aggregate bandwidth (GB/s), stacked per device contribution

P2P Bandwidth: 3 vs 4 NVMe + GPU

Left: 3 NVMe SSDs | Right: 4 NVMe SSDs





SDC | StorageAI™

Experiment 4: Takeaways

- Bandwidth scales with the number of NVMe devices aggregated per GPU
- The intuitive 4 x4 ratio reaches the GPU's bandwidth ceiling
- Larger block sizes drop the required I/O rate dramatically: 4 KiB needs 1/8 the IOPS of 512B for the same bandwidth
  - Extrapolating: a single CPU core could feed ~**8 GPUs at 4 KiB**
- Surfaces a fourth challenge beyond the three challenges introduced earlier: transport protocol overhead (PCIe + NVMe encapsulation)
  - Significant impact on the bandwidth ceiling; not addressed by AiSIO

Due to overhead, 3 x4 NVMe SSDs is the cost-effective sweet spot
Nearly saturates the GPU; the 4th device adds very little

Accelerator-integrated Storage I/O (AiSIO)

The vision: delivered

- Accelerators integrated into the storage stack
  - Demonstrated using NVIDIA H100 and 16 Samsung PCIe Gen5 NVMe SSDs
- Three I/O modes (CPU-initiated, CPU-initiated P2P, Device-initiated P2P)
  - All three saturate the 16-drive system with ~61.5M IOPS, the CPU-paths consuming using 1.5 CPU cores
  - Expected to scale linearly to ~96M IOPS at 2 cores (4 HW threads) if more drives could fit
- Preserve files, filesystems, and OS control; interoperability over replacement
- All open source; in or targeting upstream, open collaboration model

Software always incurs overhead, but AiSIO reduces it dramatically.
Readies the Storage for AI software ecosystem for PCIe Gen6 and Gen7.

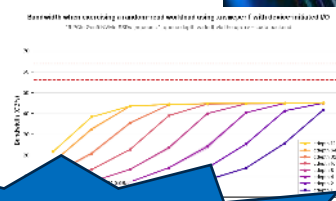
Accelerator-integrated Storage I/O (AiSIO)

The vision: delivered

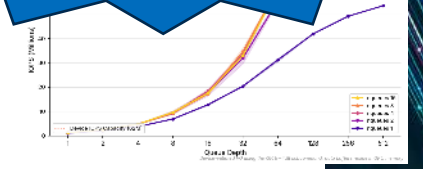
- Accelerators integrated into the storage stack
  - Demonstrated using NVIDIA H100 and 16 Samsung PCIe Gen5 NVMe SSDs
- Three I/O modes (CPU-initiated, CPU-initiated P2P, Device-initiated P2P)
  - All three saturate the 16-drive system with ~61.5M IOPS, the CPU-paths consuming using 1.5 CPU cores
  - Expected to scale linearly to ~96M IOPS at 2 cores (4 HW threads) if more drives could fit
- Preserve files, filesystems, and OS control; interoperability over replacement
- All open source; in or targeting upstream, open collaboration model

Software always incurs overhead, but AiSIO reduces it dramatically.
Readies the Storage for AI software ecosystem for PCIe Gen6 and Gen7.

Coming **next**: multipath I/O across all paths, with/without SR-IOV



New Device-initiated P2P experiments



Try it out!

- Source and reproducible experiments at github.com/xnvme/aisio
- Dive into details in the online edition of the white-paper at https://xnvme.io/aisio


SDC | StorageAI™
A SNIA Event
Thank You