CXL Memory for LLM Inference Staging

via an On-Device DMA Controller

Alok Prasad · Veerasenareddy Burru · Pradeep Kumar Nalla · Satananda Burla

MARVELL™

HotStorage 2026

18th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '26) · September 28–29, 2026 · Prague, Czech Republic

Talk Outline

01 **LLM Serving & the KV Cache**
how models stay fast — and why memory grows

02 **The Memory Problem**
the KV cache fills scarce GPU memory

03 **Today's Options & Limits**
host DRAM, NVMe, RDMA — and why they fall short

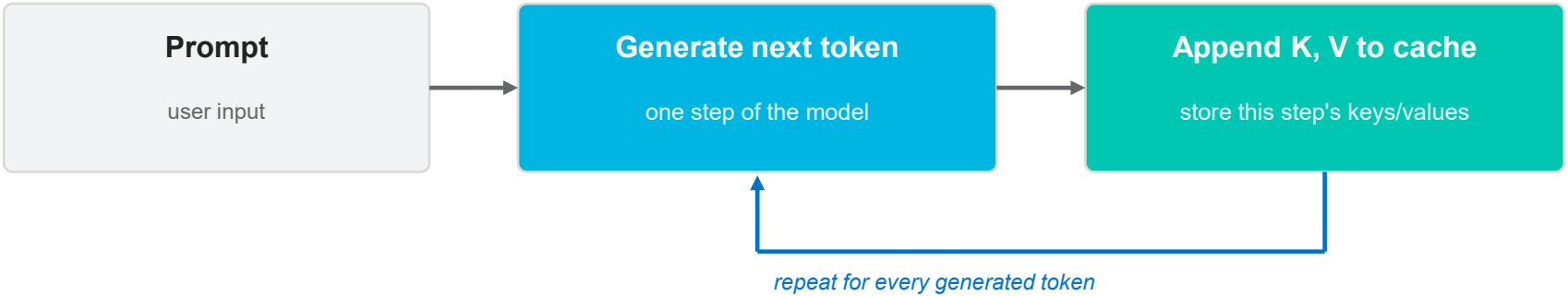
04 **Our Approach**
CXL + on-device DMA, and where it fits in NIXL

05 **Results**
bandwidth & latency vs. NVMe, across block sizes

06 **Conclusion & Future Work**
takeaways, plus the write-path & multi-GPU roadmap

How an LLM Serves a Request

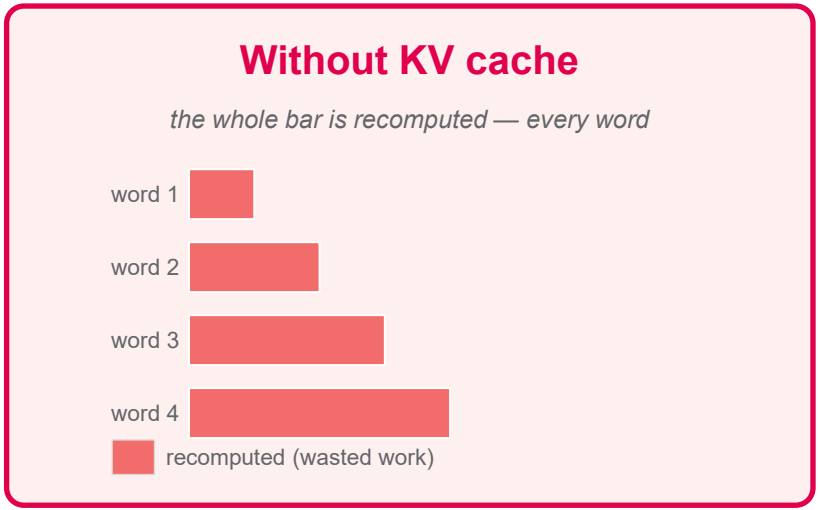
A language model generates text one token at a time — reusing earlier work through a KV cache.

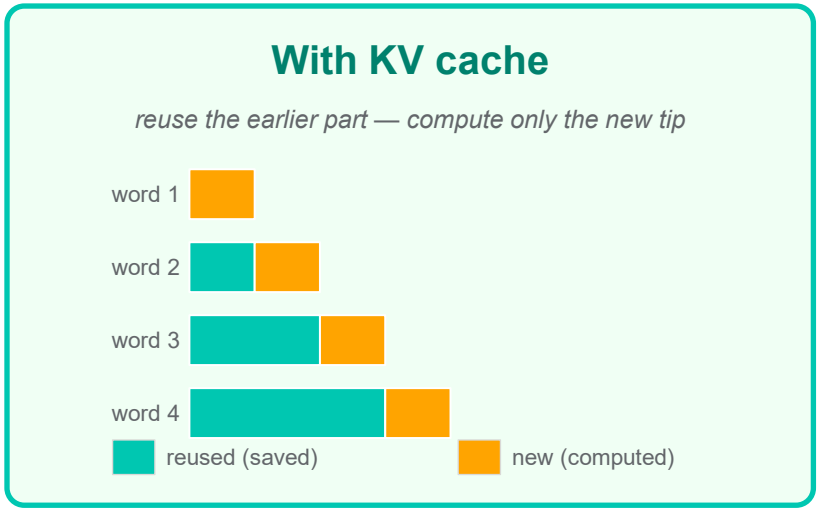


- Each new token attends to all earlier tokens' keys & values.
- The KV cache stores them, so they're computed once — not re-derived every step.
- More users and longer outputs mean a larger, continuously growing KV cache.

The KV Cache — Reuse Instead of Redo

Each bar is the work to produce one word. Without the cache, the whole bar is recomputed every step; with it, the earlier part is reused and only the new tip is computed.

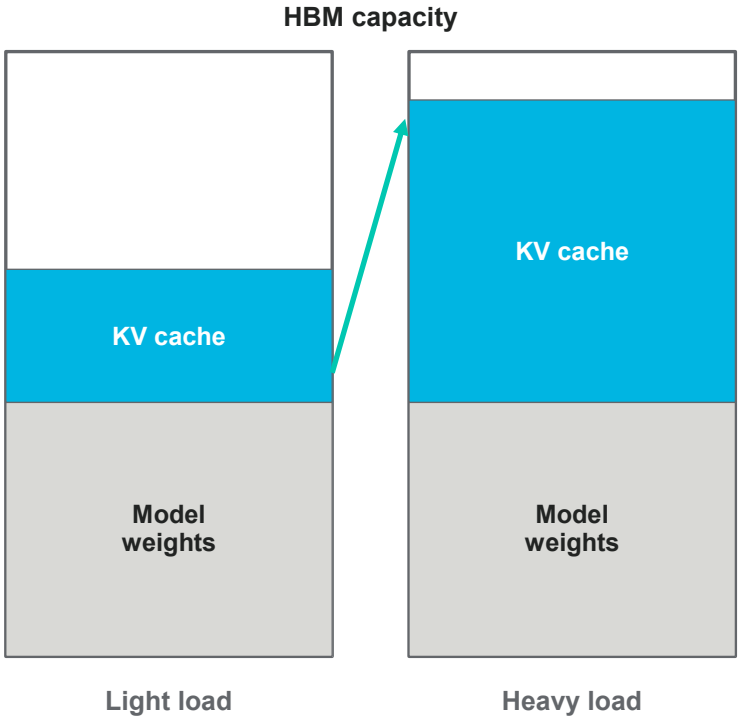




Reusing - makes AI Fast

Growing KV Cache Memory

*GPU memory (HBM) holds the **model weights** and the **KV cache** — and only the **KV cache** keeps growing.*



- Weights are fixed; the KV cache scales no. of user and with context.
- It lives in HBM — the fastest memory, but also expensive.
- Under load, the KV cache can take over most of HBM.

So HBM capacity — not compute — often decides how many requests you can serve at once.

When HBM Fills Up

Once weights + KV cache exceed HBM, inference has only three options

Shrink the workload

Limit user & Context

→ lower throughput

Evict & recompute

Drop cached keys/values(Context) and recompute them again later.

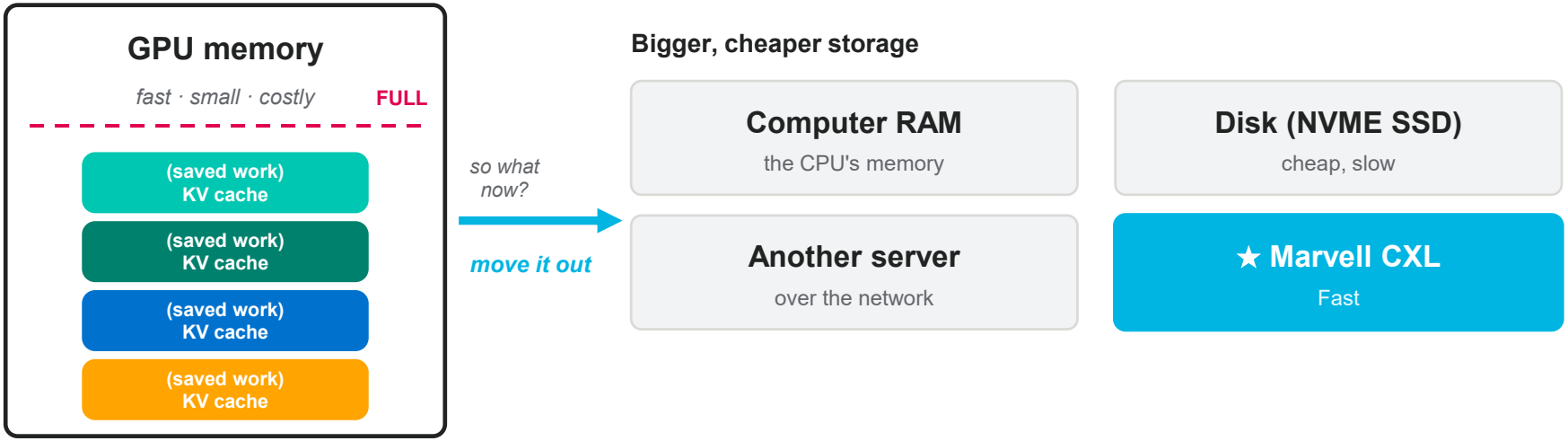
→ wasted GPU work, more latency

Spill off-GPU

Move the KV cache to another memory tier and fetch it back.

→ keeps throughput

When the GPU Runs Out of Room



Q1 Where do you put it?

Q2 How fast can you write and get it back?

Where Does the KV Cache Go Today?

KV-cache traffic is small, frequent and continuous — the access pattern today's off-GPU tiers handle worst

Each spill option falls short

Host DRAM

CPU Usage + shares the bus with other traffic – **Bus Contention**.

NVMe via GDS

Built for bulk block I/O — **slow and high-latency on small**, frequent transfers.

RDMA

Network-oriented and coarse-grained; throughput gated by **network latency**.

What KV-cache staging needs

High bandwidth — near the PCIe link speed

Low latency — host-bypass, no CPU in the path

Fine-grained access at **4 KB–128 KB** granularity

Our Approach: Direct GPU Access to CXL Memory

GPU-facing traffic runs on PCIe P2P DMA.

1 **No GPU Compute Utilization**

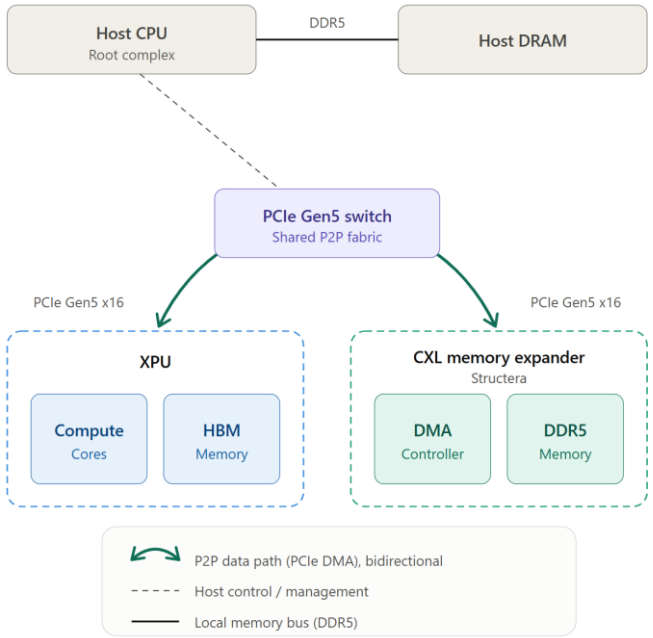
The expander's DMA controller drives every transfer directly against GPU HBM — the GPU is passive.

2 **Full host bypass (P2P DMA)**

Data never touches host DRAM - a pure PCIe switch-domain transfer.

3 **Framework-compatible**

No changes to LLM Inference Ecosystem.
Added as a plugin works along with existing other plugins.

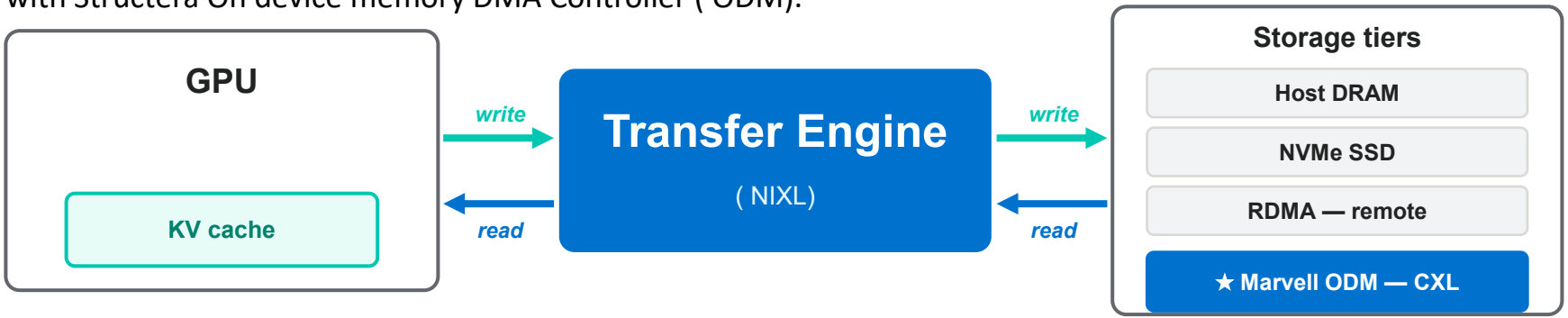


NIXL — Writing & Reading the KV Cache

Transfer Engine is a generic term for technologies designed to move KV cache efficiently across GPUs, nodes, or clusters with minimal latency.

NIXL (NVIDIA Inference Transfer Library): Emerging industry-standard transfer engine for KV-cache movement.

Our contribution: Extends and integrates with the NIXL-based transfer ecosystem through NIXL Plugin to Interact with Structera On device memory DMA Controller (ODM).



Write —save new KV cache out (and to free HBM as it fills)

Read — bring cached KV back to the GPU when it's needed again

Performance Characterization with the NIXL Framework

NIXL

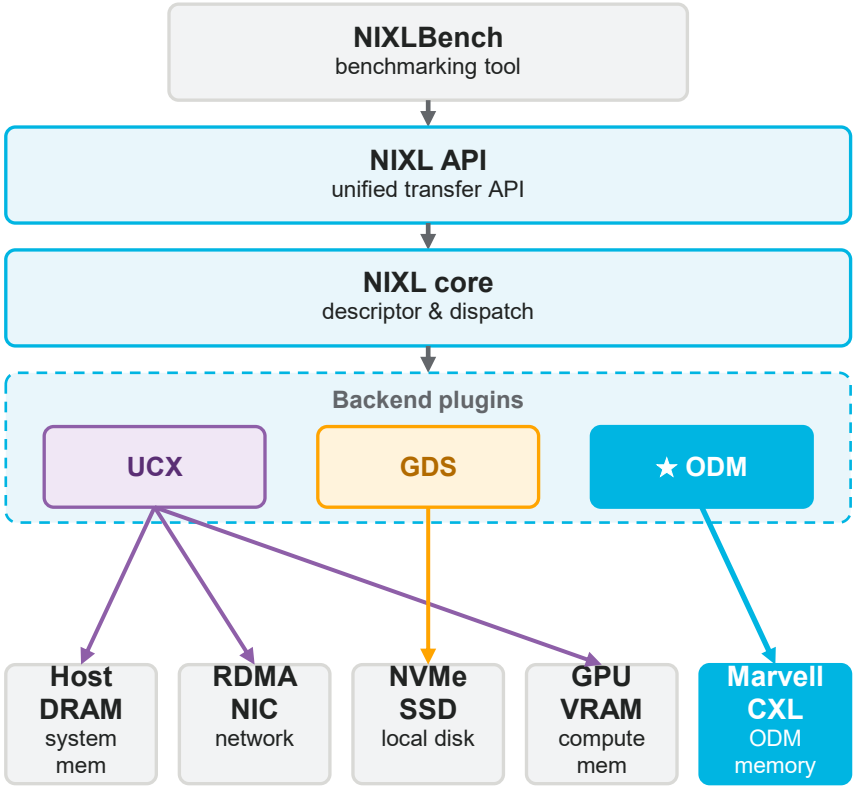
NVIDIA's library for fast point-to-point transfers in AI inference (e.g., Dynamo) — a plugin-based abstraction over many memory & storage types.

Backend plugins

- Do the actual data movement for specific hardware.
- Each plugin — UCX, GDS, or our ODM — executes the real memory transfer.

NIXLBench — benchmarking tool

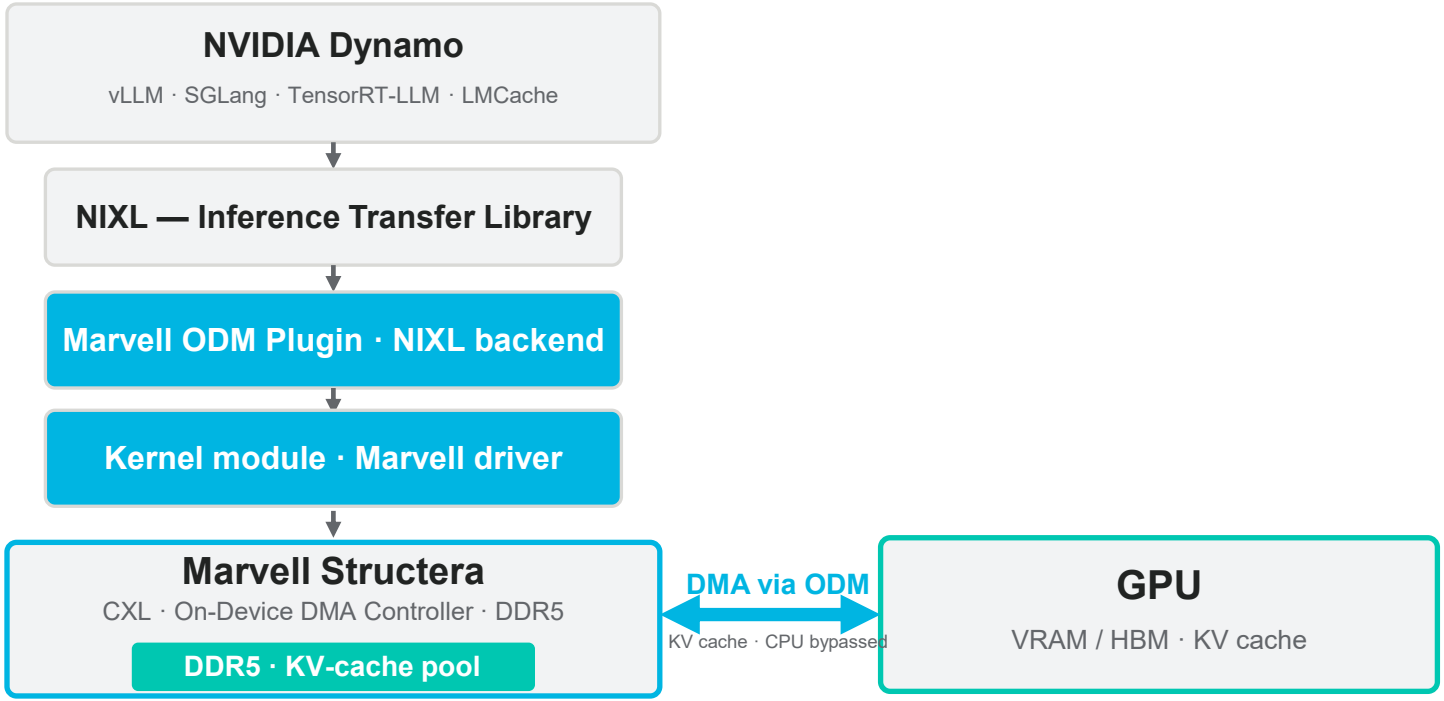
- Measures NIXL transfer performance across backend plugins.
- Reports latency & bandwidth for GPU, storage, and network paths.



Our ODM plugin adds Marvell CXL memory as a NIXL backend — and NIXLBench measures it head-to-head with UCX and GDS.

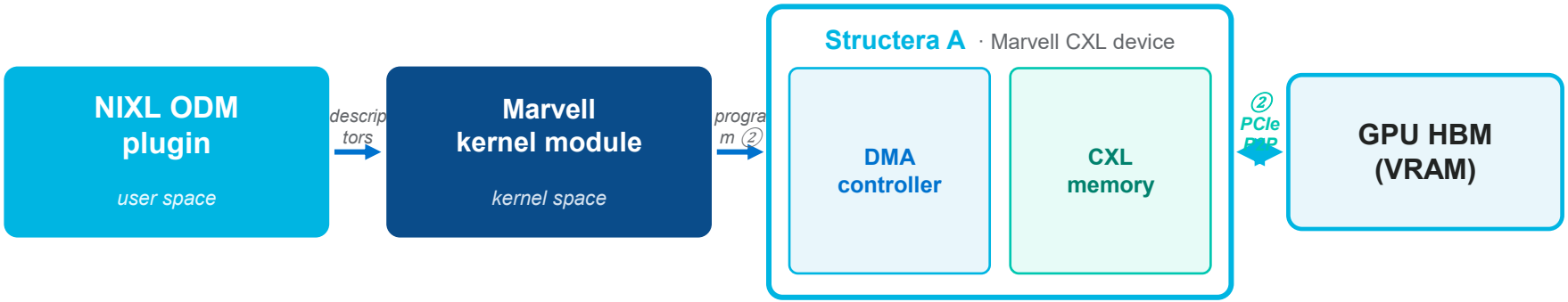
ODM Plugin

One top-to-bottom stack from orchestration down to the device; GPU exchanges KV cache with Structera by DMA.



Data Path: Anatomy of a Transfer

Structera A's on-device DMA controller is the sole master of every GPU-facing transfer.



① Submission
post descriptors (io_uring), spray to 16 queues, ring doorbell

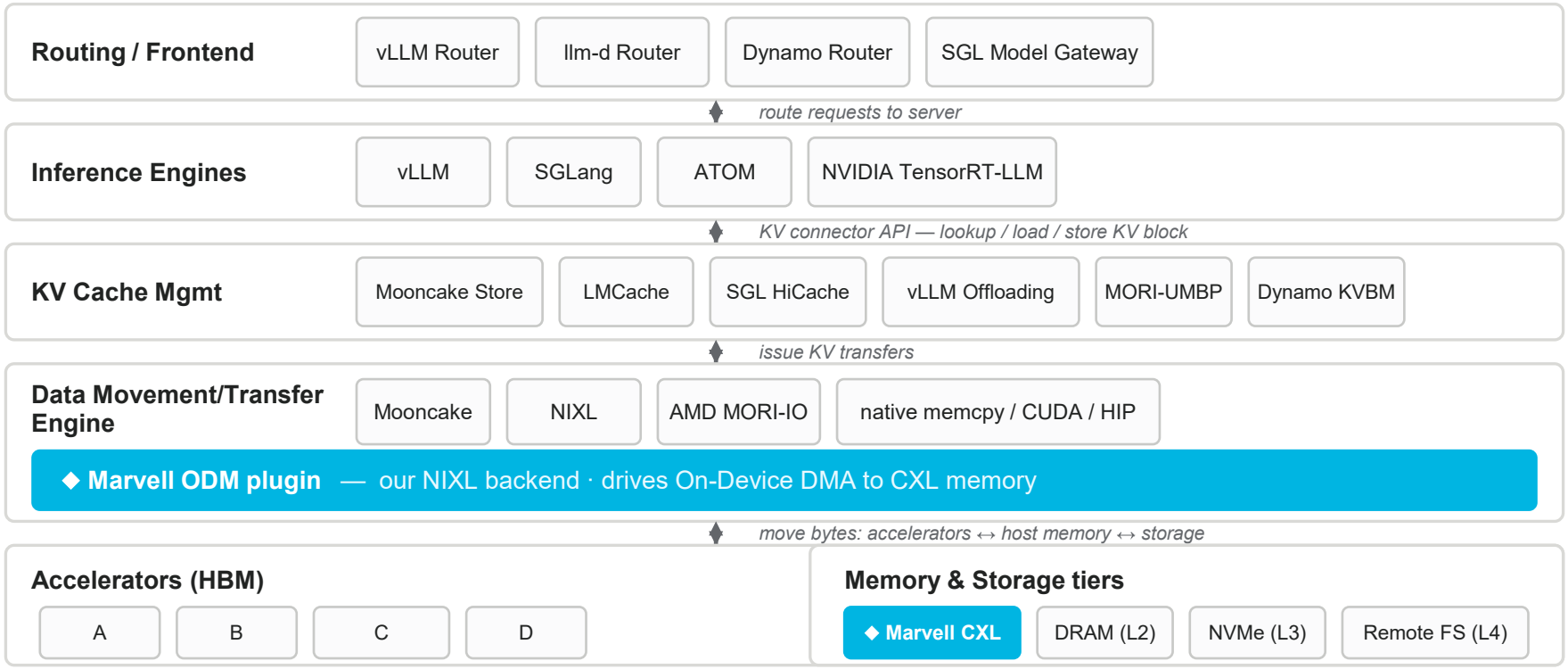
② Execution
PCIe read/write direct to GPU HBM — no host DRAM / CPU / IOMMU

③ Completion
SQ↔CQ pairs; polled counter favors low latency

④ Pipelining
back-to-back; a core per queue keeps the link busy

Placing Our ODM Plugin in the LLM Inference Ecosystem

Our contribution slots into two layers: an ODM plugin in the data-movement layer, and Marvell CXL memory as a new fast tier alongside DRAM, NVMe and remote storage.

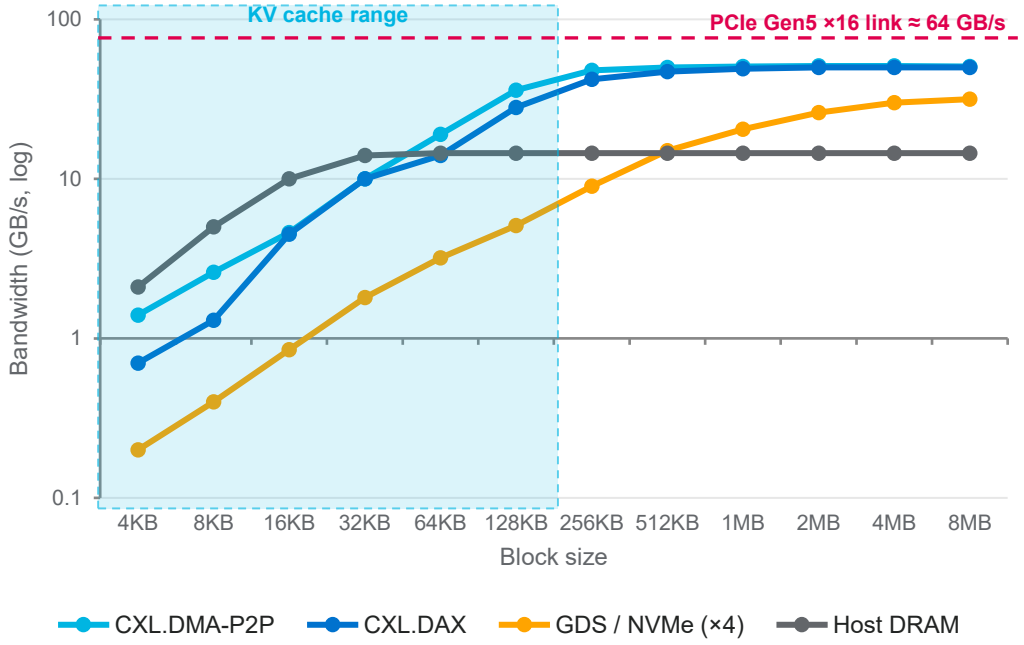


Experimental Setup

H200 GPU and a Gen5 ×16 CXL expander on one shared PCIe switch

GPU	NVIDIA H200 NVL
CXL expander	Marvell Structera A — on-device DMA controller
Expander memory	128 GB DDR5
PCIe	Gen5 ×16 — the shared link is the real bottleneck
Topology	GPU, expander & NVMe under one shared PCIe switch
Host CPU / DRAM	AMD EPYC 9555 64-core / 768 GB
OS / Kernel	Ubuntu 24.04 / 6.17.0 · IOMMU disabled (baseline)
Software	CUDA 13.1 · NIXLBench + custom NIXL backend & DAX plugins
SSD	4x NVME SSD Samsung 9100 Pro

Read Bandwidth vs. Block Size (NIXLBENCH)



4 KB–128 KB

KV cache sweet spot

Biggest bandwidth lead over NVMe/GDS here

51 GB/s

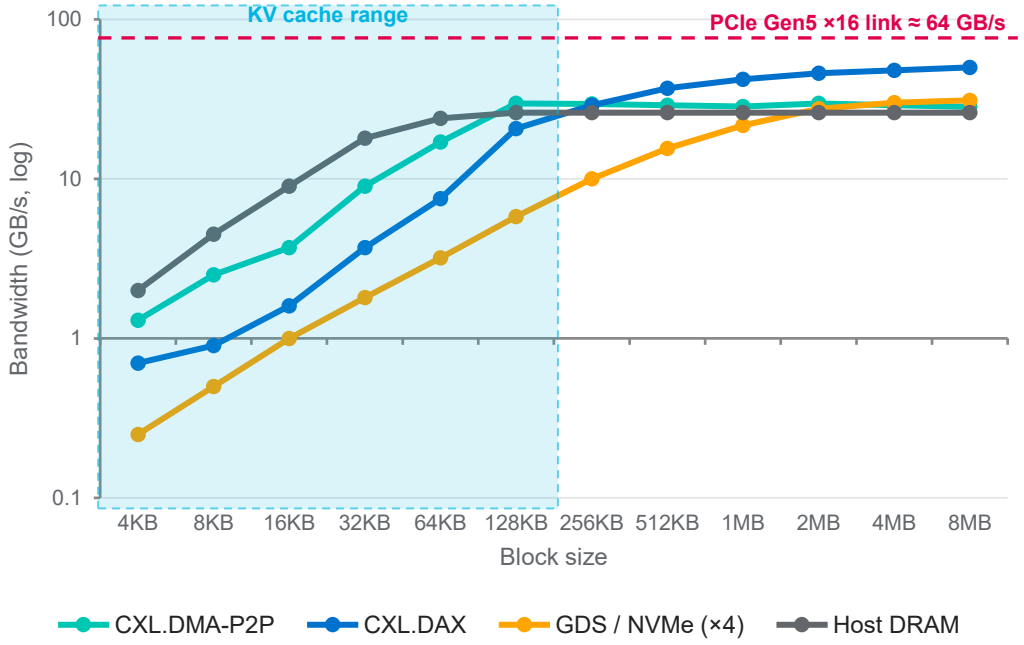
Peak read bandwidth

close to the Gen5 link limit

CXL.DMA-P2P stays ahead of NVMe/GDS at every block size.

NVMe baseline: 4× PCIe Gen5 ×4 SSDs (aggregate)
Host DRAM: shared PCIe bus w/ concurrent traffic

Write Bandwidth vs. Block Size



4 KB–128 KB

KV cache sweet spot

CXL leads across the KV range, up to ~1 MB

29.7 GB/s

Peak write bandwidth

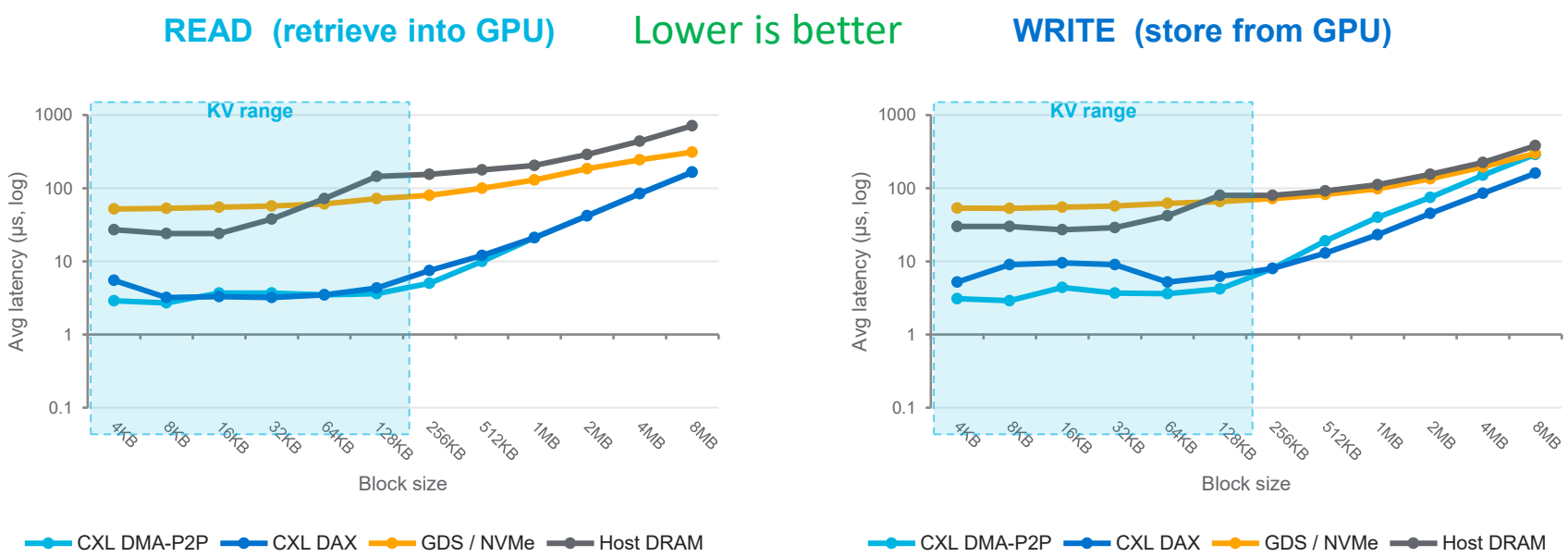
posted-write, fully pipelined

NVMe only pulls ahead on bulk, multi-MB writes.

NVMe baseline: 4× PCIe Gen5 ×4 SSDs (aggregate)
Host DRAM: shared PCIe bus w/ concurrent traffic

End-to-End Latency vs. Block Size — Read & Write

Same four paths, split by direction. In the KV-cache range, CXL DMA-P2P sits well below NVMe-GDS on both read and write.



nixlbench · H200 · Gen5 ×16 · avg per-transfer latency (log). CXL-P2P under ~50% DRAM contention; DRAM & NVMe uncontended — not like-for-like.

Conclusion & Future Work

Key Takeaways

- ✓ Reaches CXL memory over PCIe P2P DMA — no GPU changes
- ✓ High bandwidth — close to the PCIe link on reads
- ✓ Order-of-magnitude lower latency than NVMe/GDS/RDMA
- ✓ Biggest wins at KV-cache block sizes (4 KB–128 KB)
- ✓ Pluggable with NIXL.

Future work/Roadmap

- → Write-path optimization: pipelining ,queue depth.
- → Share one expander across multiple GPUs / tenants
- → On-expander KV compression using its own compute
- → Hot/Cold KV Transfer Intelligence.
- → End-to-end evaluation in vLLM / TensorRT-LLM.
- → Upstreaming NIXL Plugin.
- → Support other transfer Engines - Mooncake TE, AMD MORI-IO.

CXL memory expanders are a production-ready, high-bandwidth, low-latency alternative to NVMe staging for LLM inference — today.

MARVELL

Thank You

Questions & discussion welcome

Alok Prasad
palok@marvell.com
Bengaluru, India

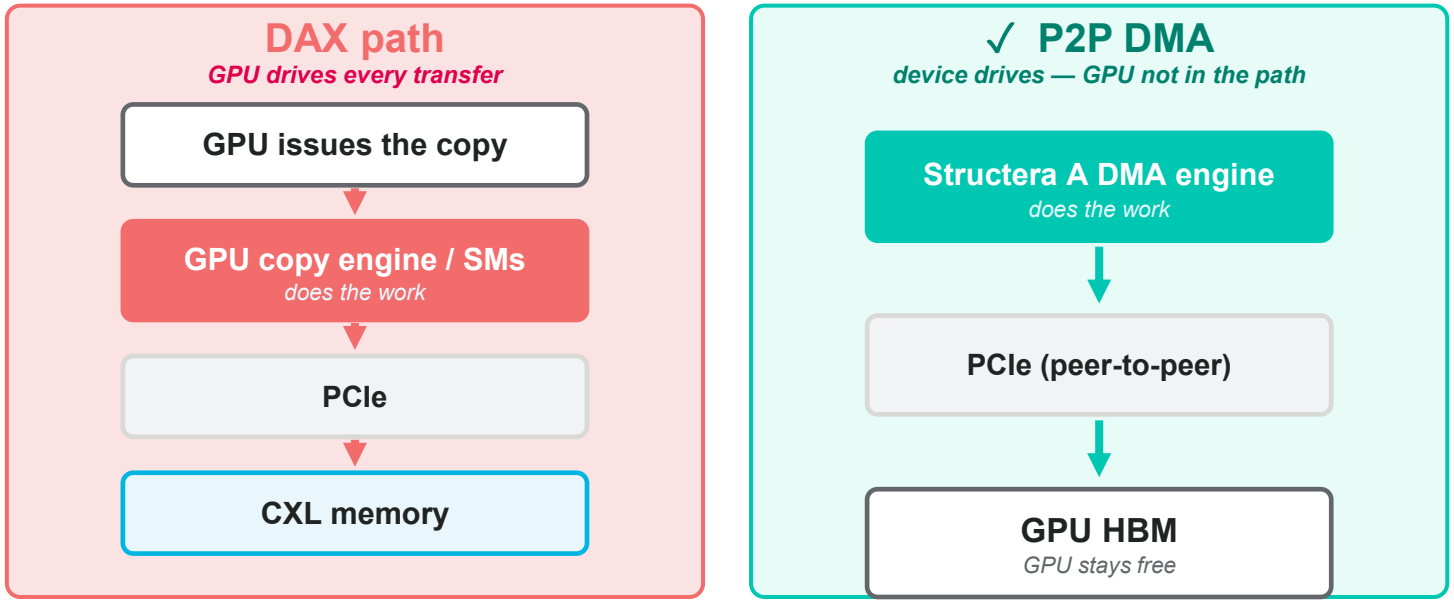
Paper: https://doi.org/10.1145/3837053.3837356

Backup

Paper: https://doi.org/10.1145/3837053.3837356

P2P DMA vs DAX Path — The Flow

Follow the blocks: the DAX path routes the transfer through the GPU's engines; P2P DMA hands it to the device.



✓ GPU stays free

✓ lower latency

✓ no CXL in GPU

✓ less contention

Understanding the Read / Write Asymmetry

Reads run faster than writes at peak

Read · expander → GPU HBM · posted, fire-and-forget



Write · GPU HBM → expander · non-posted, must wait



Write-path optimization is active future work — deeper pipelining, write-combining, request batching, completion-handling tuning.