

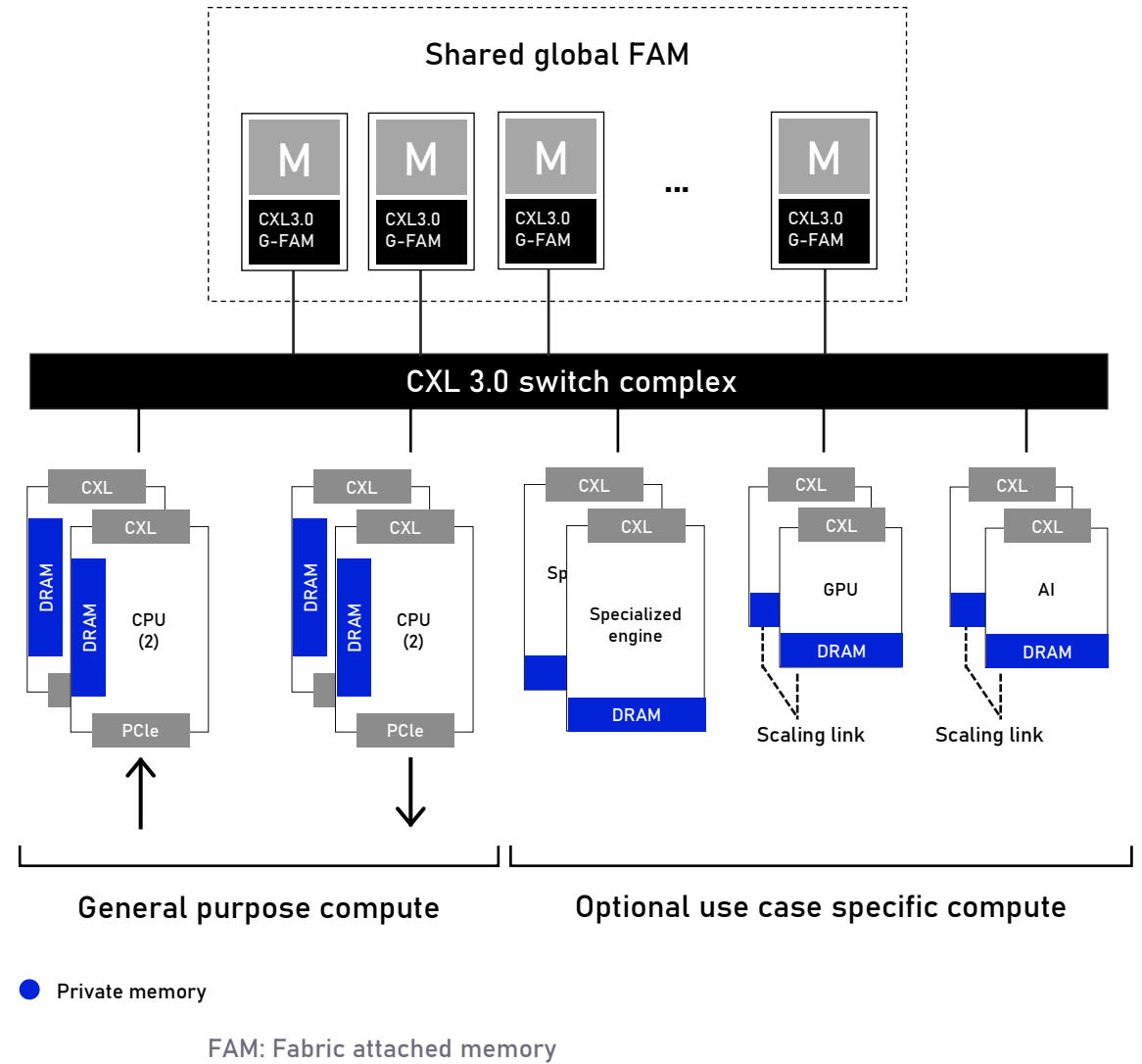
KV Cache Scaling using Dynamo and CXL Shared Memory

Dr. Skyler Windh, MTS Software Engineering, Micron

October 21, 2026

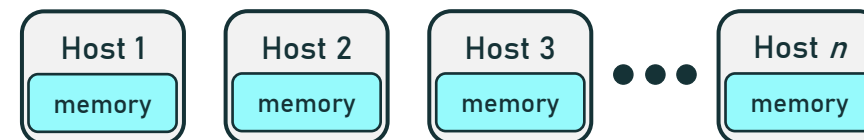
01	Motivations for Disaggregated Memory
02	Modeling KV Cache Sharing
03	Disaggregated Memory Enablement Considerations
04	KV Sharing implementation In Dynamo
05	KV Sharing results

Memory fabrics enable memory-centric system architectures



Addressing AI/ML scalability challenges with global fabric attached memory

- Problem
imbalanced memory:compute → stranded compute + excess data movement → poor performance & energy
- GFAM
scale compute and memory independently → efficient use for memory-bound workloads (LLMs)
- Bonus
less data movement — data stays put in zero-copy formats (NumPy, pandas, Arrow)
- Cost
strategic placement + some software-stack changes



Aggregate memory footprint

Large # of compute resources to achieve required total memory



CXL GFAM memory footprint

Memory size independent of # of compute resources

Abaco 3.0 Reference Architecture (DOE/PNNL program)

High-capacity shared memory system architecture for AI & HPC workloads

Goal: *Independently scale memory* like the industry already has for compute, storage, and networking. Micron is the preferred memory partner for this configuration.

Memory Footprint	Rack Capacity (4x 4U JBOMs)	JBOM Capacity (4U)	JBOM RDIMM Configuration	System Availability
Small	40TB	10TB	160x 64GB	CQ4-26
Medium	80TB	20TB	160x 128GB	CQ4-26
Large	160TB	40TB	160x 256GB	CQ1-27
Extra Large	320TB	80TB	160x 512GB	CQ4-27

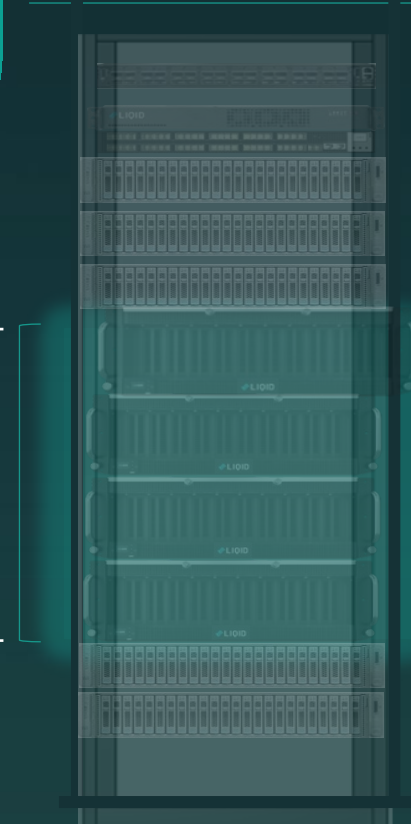
All configurations based on Micron® DDR5 RDIMMs, Primemas® PCIe add-in cards (10x AICs per system, 16x RDIMMs per AIC), and Liquid® EX-5410C Memory Expansion Chassis (4U) with 4 PCIe ports x16, dual-socket CPUs with 16 memory channels, and 6000W chassis power limit (no thermal or cooling concerns since Micron reference architecture only needs ~3000W)

The Abaco 3.0 is a rack-scale system architecture developed for the Department of Energy's Pacific Northwest National Laboratory (PNNL) to power next-generation, large-scale scientific AI, HPC and data-intensive workloads. This effort is supported by DOE's Office of Science, Advanced Scientific Computing Research program.

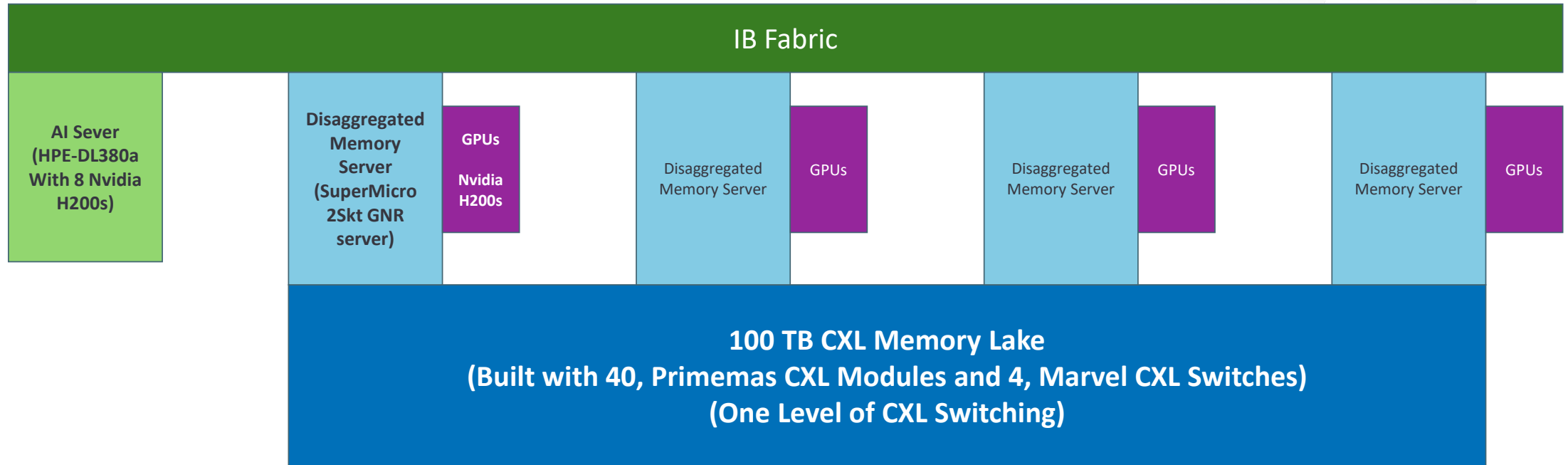
*Leveraging open-source [Famfs](#) (fabric-attached memory file system), contributed by Micron

Abaco 3.0 Rack Specs

- 1x Fabric manager server host
- 1x CXL switch
- 5x AI GPU compute servers
 - PCIe Gen5/6
 - Dual-socket CPU
 - CXL host adapter card
 - 4x GPUs
- 4x JBOMs
 - Liquid® EX-5410C Memory Expansion Chassis (4U)
 - 10x Primemas® PMA Memory Expansion CXL Add-In Cards
 - 160x Micron® DDR5 RDIMMs
- Target Applications*
 - KV cache for LLM inference
 - Large databases
 - Graph analytics
 - Vector RAG
 - Graph RAG



SMS's CXL Cluster Prototype



Micron's DDR5 128GB and 256 GB RDIMMs are used to build the Memory Lake

Great Abaco Rack

Front of Rack (Cold side)

Back of Rack (Warm side)

Head node Server

AI Server

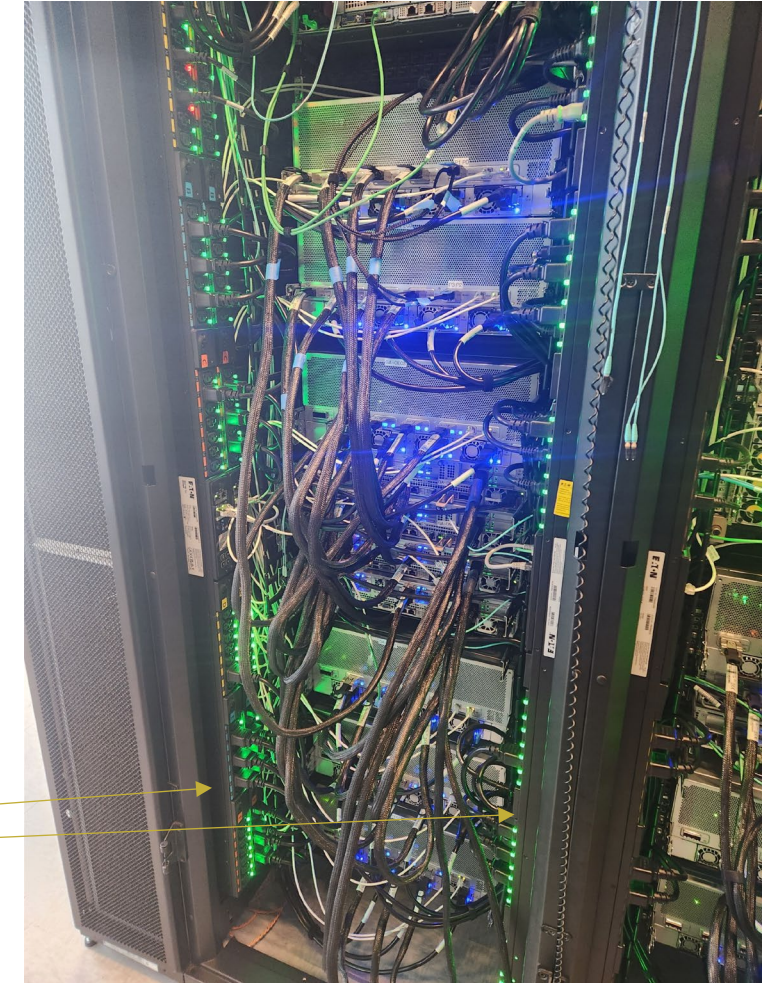
Liquid Director

PCIe Expansion

CXL Expansion

DM Servers

PDU



Modeling KV Cache Sharing

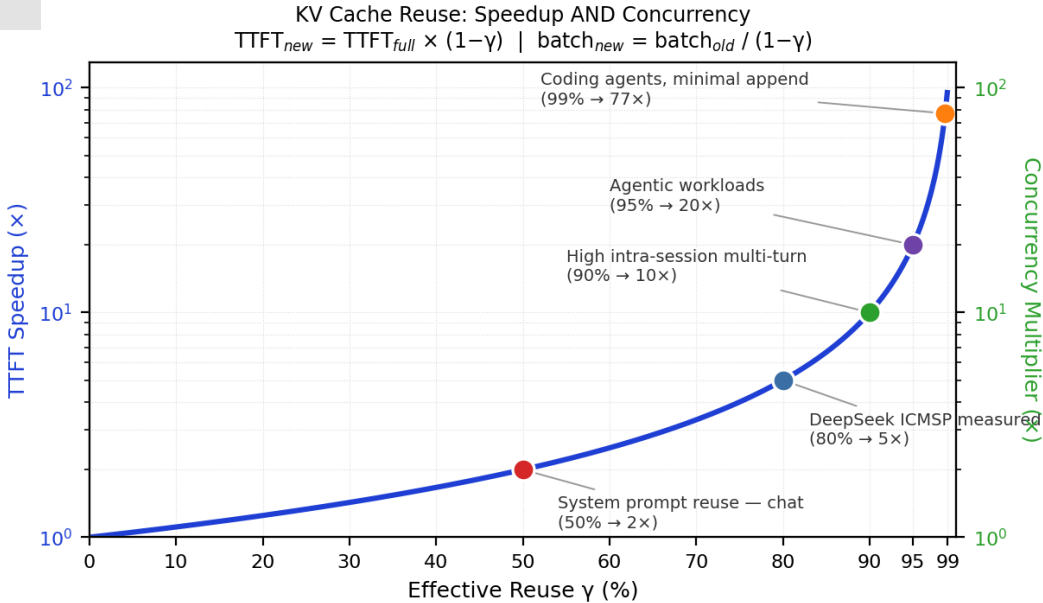
Theoretical Impact of KV Reuse on LLM Serving

KV cache reuse (γ) improves both latency and throughput simultaneously

KV reuse (γ) drives

$$\begin{aligned} \text{TTFT}_{\text{offload}} &= \text{TTFT}_{\text{re_compute}} \times (1 - \gamma_{\text{effective}}) \\ \text{batch}_{\text{offload}} &= \text{batch}_{\text{re_compute}} / (1 - \gamma_{\text{effective}}) \end{aligned}$$

Effective Reuse (γ)	TTFT Speedup	Concurrency Multiplier	Workload Example
50%	2×	2×	System prompt reuse — chat
80%	5×	5×	ICMSP DeepSeek measured
90%	10×	10×	High intra-session multi-turn
95%	20×	20×	Agentic workloads
98.7%	~77×	~77×	Coding agents, minimal append



$\gamma_{\text{effective}} = \gamma_{\text{workload}} \times \text{hit_rate}(S, W)$ — capacity and bandwidth determine how close you get to the max

Reuse simultaneously improves latency and throughput, making it a first-order lever for serving efficiency.

Additional Insights from Analytical Modeling

1. Reuse γ sets the ceiling
 - Maximum speed-up and concurrency scale as $1/(1-\gamma)$
2. Capacity is the first hard gate
 - Reuse cannot be realized unless the reusable working set fits in the tier.
 - Capacity $\rightarrow$ Hit Rate $\rightarrow \gamma_{\text{effective}} \rightarrow$ Performance
3. Bandwidth determines how closely the system approaches the theoretical reuse ceiling.
 - Dense models tend to remain compute-bound.
 - Large MoE models become increasingly bandwidth sensitive.
4. Different KV Classes Exhibit Different Reuse and Retention Characteristics
 - Shared system prompts $\rightarrow$ high reuse $\rightarrow$ storage-class tier
 - Session context $\rightarrow$ medium retention $\rightarrow$ CXL memory-class tier
 - Agentic workflows reveal that not all KV has the same reuse potential

Reuse defines the maximum benefit. Capacity and bandwidth determine how much of that benefit can be realized.

KV Cache Sharing

Software Considerations

Private Caching Is Straightforward. Cross-host Reuse Needs Software.

~90%

Max host KV capacity before host crashes

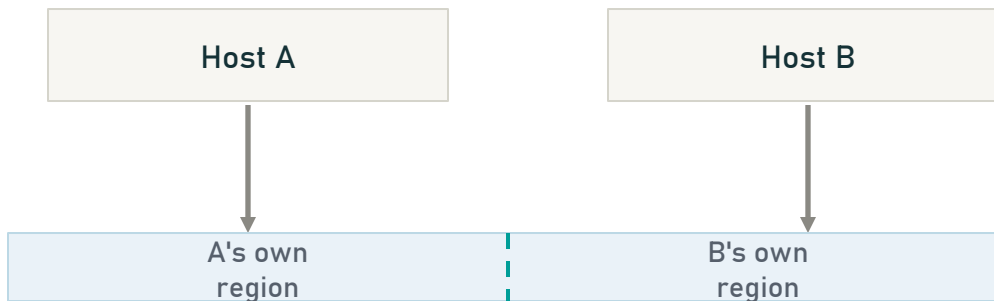
4 TB

KV cache reached over CXL

3× (with half the memory)

Throughput when hosts reuse each other's blocks

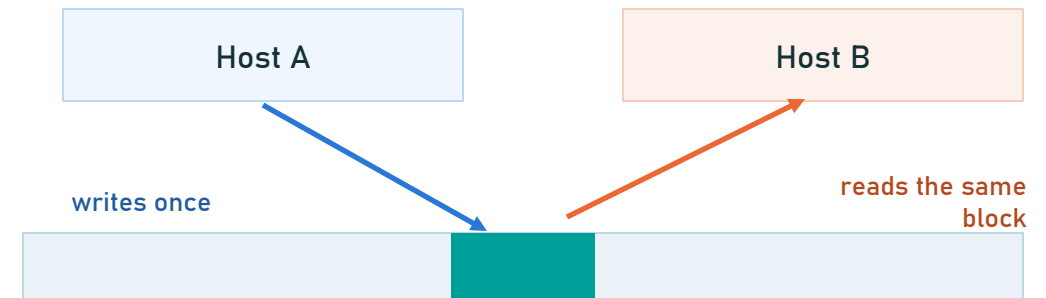
Pooling · PRIVATE REGIONS



Each host reads only what it wrote

Nothing to keep coherent — and nothing in the application changes

Sharing · SHARED BLOCKS

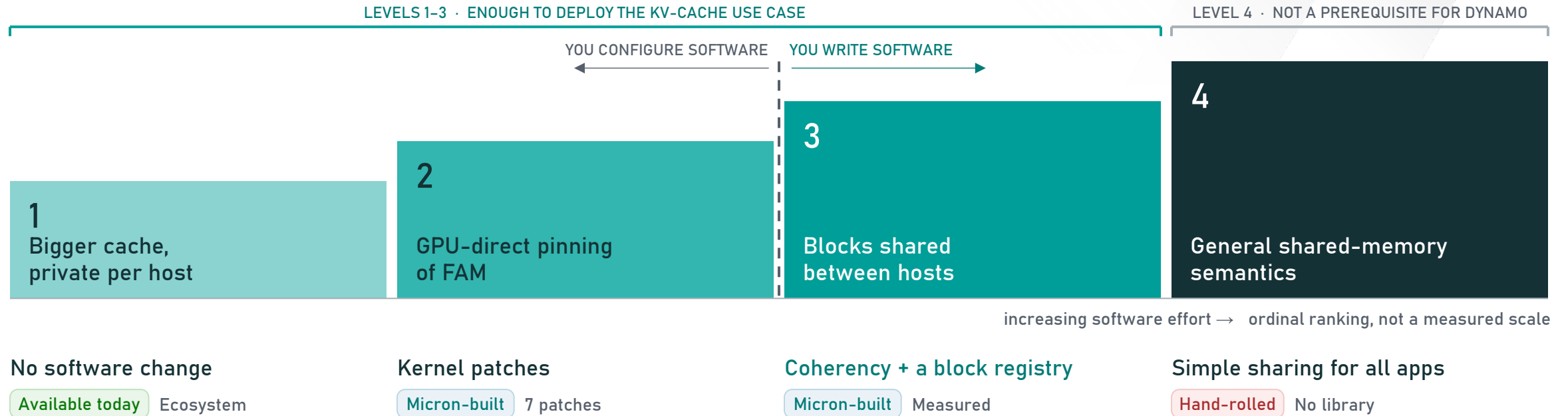


Host B reads a block Host A wrote

Coherency is now on you — and that is where every software cost starts

The line is **not one host versus many** — it is whether a host ever reads a block **another host wrote**.

Four Levels Of Enablement



Level 4 means general library support for many applications. Likely done through industry standards and libraries. As CXL 3.x matures, hardware coherency also solves some of level 4

What was Modified in Dynamo



CLIENT & SERVING	Dynamo frontend + KV-aware router · load generator Never learns that a CXL tier exists	UNCHANGED
INFERENCE ENGINE	vLLM · PyTorch / CUDA / NCCL · NVIDIA GPU driver Zero vLLM patches in any configuration	UNCHANGED
KV CACHE MANAGER	KVBM — the Dynamo KV Block Manager core (NVIDIA) Takes a base address and a length, so it does not care which tier	UNCHANGED
----- UNTOUCHED ABOVE THE LINE · BUILT OR CHANGED BELOW IT		
MICRON CXL SOFTWARE	CXL tier for KVBM + cross-host sharing Block pool, remote registry, descriptor exchange — and the coherency recipe	NEW · LEVEL 3
DMT MEMORY MANAGEMENT	famfs (<i>Apache-2.0, upstreaming</i>) Allocates fabric memory and gives every host the same named view of it	NEW · LEVEL 1-2
OS & KERNEL	famfs + <i>long-term-pin patches</i> Patches to enable GPU pinning of famfs files	CHANGED · LEVEL 2
HARDWARE	GPU server unchanged + CXL memory chassis & fabric switch — <i>the only hardware DMT adds</i>	

Three adjacent layers. Zero vLLM patches, and the router never learns a CXL tier exists.

The Ecosystem is Shipping Capacity but Not Sharing



WHAT COMES FREE / AND WHERE IT STOPS

Free today LMCache, since August 2026

- A byte-addressable tier between DRAM and storage, at 86–90% of pinned-DRAM bandwidth.
- No vLLM patch and no connector protocol
- Devices added, removed and resized at runtime.

Still open Not addressed yet

- Gap 1 · cache coherency — never mentioned.
- Gap 2 · sharing is between processes on one host, not across servers.
- Gap 3 · no per-TP partitions — slot-based accounting only.
- Gap 4 · volatile — restart recovery is not implemented.

Capacity on one host is solved, and free.

Gaps 1–3 are what we built — the rest of this deck, one gap at a time. Gap 4 we deliberately do not chase.

Source: LMCache blog, 2026-08-04, “Device-DAX in LMCache” (Qwen3-14B). Bandwidth and TTFT figures are the authors' own and setup-specific.

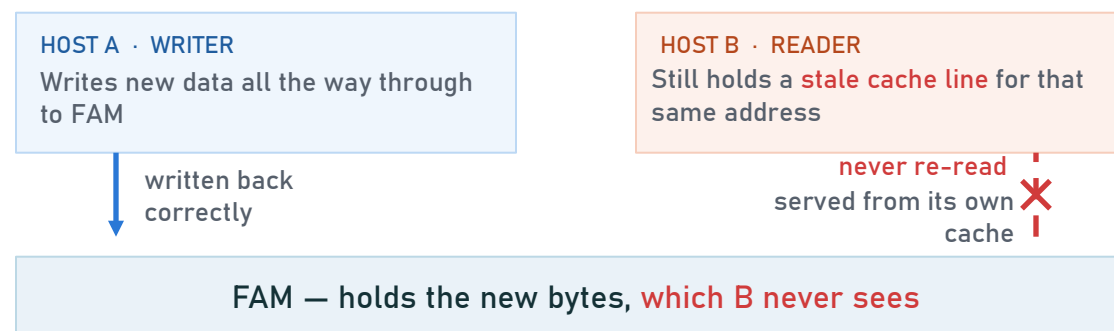
Two Major Failure Cases in Software Coherency

GAP 1 · COHERENCY / WHY CROSS-HOST REUSE IS HARD

1 · The writer's data never arrives



2 · The reader never sees the update



Neither failure crashes

The read succeeds and returns plausible bytes. In a KV cache those bytes are attention state, so the symptom is subtly wrong generated tokens — on a fraction of requests, only under cache pressure.

The fix must be **structural**: producers flush *before* the block is published, so an unflushed block is never discoverable. Consumers flush to invalidate cache lines before reading data.

KV Cache Sharing

Implementation in Dynamo

From a Cache Hit to a GPU Copy

DYNAMO SCHEDULING PIPELINE · OVERVIEW

SCHEDULER THREAD — must never block (feeds GPU decode)

1 · LOOKUP

acquire_local_matches

host -> disk -> CXL registry; stage matched
shard sets



2 · STAGE & FAN-OUT

trigger_onboarding

assign device blocks; fan out per-rank (shard,
dst)



3 · HAND OFF

xfer_tx.send(...)

enqueue CxlRemoteOnboard; returns
immediately

↓ **channel boundary** ↓

TRANSFER ENGINE / WORKER — off-thread, bounded concurrency

4 · ROUTE

LocalTransferEngine

-> serial cxl_onboard_task (backpressure)



5 · FENCE + COPY

cxl_remote_onboard_blocks

consumer flush -> vectorized_copy kernel: CXL -
> GPU



6 · COMPLETE

event-worker -> oneshot

off-thread cuEventSynchronize -> vLLM
get_finished

Steps 1-3 run on the scheduler thread but do only cheap bookkeeping and a channel send. All blocking work (CUDA copy, cache flush, event sync) lives past the channel boundary.

Core Implementation Concepts

How to enable CXL Sharing



CXL Backed Host Blocks

CXL FAM Address Space (20TB total)

ga-dm0 Host Block Pool

Region: cxl_host_b7a421c5-... • Addr: 0x200000000040 • Size: 30 GB (15,258 blocks)

B0 B1 B2 B3 ... B15257 ← each 2 MB

ga-dm1 Host Block Pool

Region: cxl_host_1097f9e8-... • Addr: 0x200773400080 • Size: 30 GB (15,258 blocks)

B0 B1 B2 B3 ... B15257 ← each 2 MB

Both pools are in the same physical FAM address space.
host0 CAN read host1's blocks (and vice versa) after coherency flushes.

Normal (no CXL)

PinnedStorage
backing: Cuda
(cudaMallocHost)

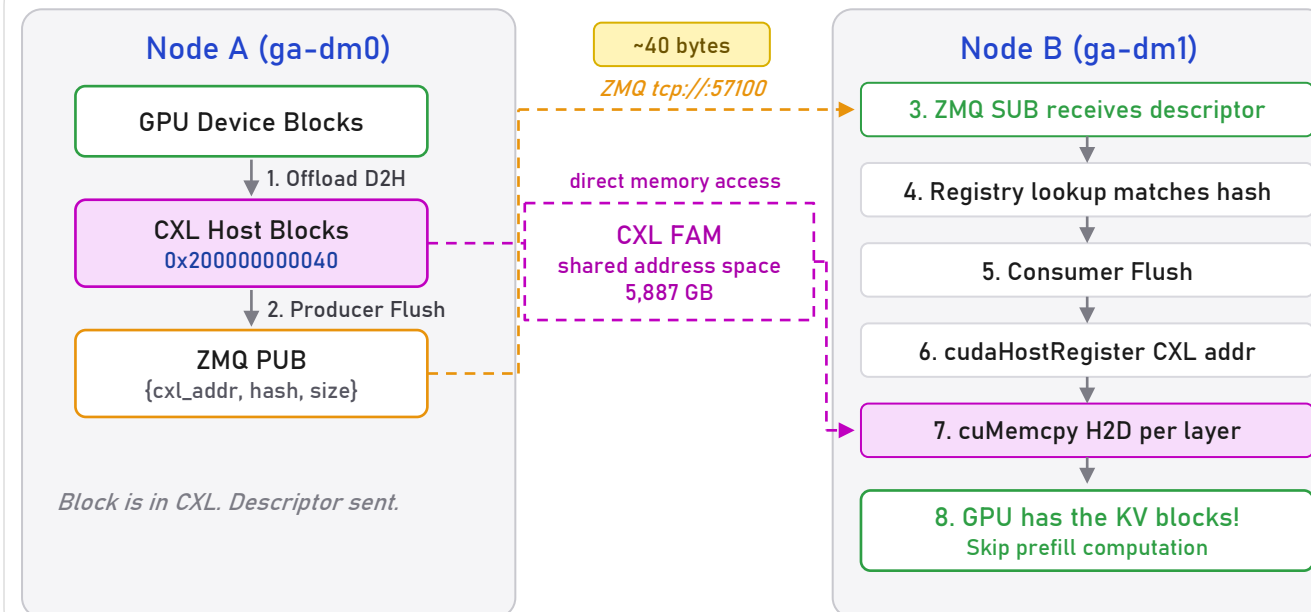
Local to one host

With CXL

PinnedStorage
backing: Cxl
(famfs file)

Shared across all hosts via DAX-mapped FAM

Cross-Node CXL Registry



Key Takeaway: a cluster with N XGB pools performs like a cluster with a single N*XGB pool

FAMFS — CXL Memory as a Shared-Memory Filesystem



STORAGE TIERING · FAMFS ON CXL MEMORY

famfs exposes fabric-attached CXL memory as a filesystem: pre-allocated, DAX-backed files, so mmap gives direct load/store to CXL with no copy. A drop-in offload-tier backend — the pool file is pre-created on the mount and each worker just opens it.

```
// Pool file is PRE-CREATED + pre-sized on the famfs mount (`famfs creat`).
// The worker only OPENS it -- never creates, never unlinks.
fn open_external(path: &str, size: usize) -> Result<Self, StorageError> {
    // famfs DAX is mmap/POSIX -> run with DYN_KVBM_DISK_DISABLE_O_DIRECT.
    let oflags = libc::O_RDWR | libc::O_CLOEXEC; // NO O_CREAT, NO fallocate
    let raw_fd = unsafe { libc::open(path_cstr.as_ptr(), oflags) };

    let file_len = fstat(raw_fd)?.st_size as usize; // must cover the pool
    if file_len < size {
        return Err(StorageError::InvalidConfig(/* too small */));
    }
    Ok(Self { /* fd, external: true -> unlink() is a no-op */ })
}
```

KVBM wiring

DYN_KVBM_DISK_CACHE_FILE=/mnt/famfs/pool -> each TP rank opens its own shard pool.0 / pool.1, size-checked, never created, never unlinked. Nothing else in the offload path changes.

Host coherency fences — producer and consumer

HOST COHERENCY · WRITE PATH & READ PATH

WRITE PATH · Producer fence — flush dirty lines, then publish

```
// Producer (writer) fence -- replicates the sw_coherency flush.
#[cfg(target_arch = "x86_64")]
pub(crate) unsafe fn flush_fence_producer_direct(ptr: *const u8, bytes: usize) {
    use core::arch::x86_64::{_mm_clflush, _mm_sfence};
    const CACHELINE: usize = 64;
    let base = ptr;
    _mm_sfence(); // order the prior GPU->host stores

    let mut offset = 0;
    while offset < bytes {
        _mm_clflush(base.add(offset)); // write each dirty line back to FAM
        offset += CACHELINE;
    }
    _mm_clflush(base.add(bytes - 1)); // tail (handles non-aligned end)
    _mm_sfence(); // barrier BEFORE the descriptor is published
}
```

Ordering guarantee

The flush's trailing sfence completes before any descriptor is published, so a peer that sees the descriptor is guaranteed to read committed FAM data — never a partially-flushed block.

READ PATH · Consumer fence — invalidate stale lines, then read

```
// Consumer (reader) fence -- invalidate local cache before reading FAM.
#[cfg(target_arch = "x86_64")]
pub(crate) unsafe fn flush_fence_consumer_direct(ptr: *const u8, bytes: usize) {
    use core::arch::x86_64::{_mm_clflush, _mm mfence};
    const CACHELINE: usize = 64;
    let base = ptr as *const u8;
    let mut offset = 0;
    while offset < bytes {
        _mm_clflush(base.add(offset)); // drop the stale cached copy
        _mm mfence(); // serialize the invalidation
        offset += CACHELINE;
    }
    _mm_clflush(base.add(bytes - 1));
    _mm mfence();
}
```

Flush where it's cheap

FAM is software-coherent: CPU caches are not kept in sync across hosts by hardware, so the reader must invalidate (clflush + mfence) before reading. We flush only the shards this rank actually onboards — moved off the ZMQ receive path (>98% of peer offloads are never onboarded) to per-onboard.

Flush a slot once, re-arm on eviction

OPTIMIZATION · FLUSH-ONCE

```
struct RegistryInner {
    blocks: HashMap<BlockHash, Vec<CxlBlockDescriptor>>,
    flushed: HashSet<u64>, // CXL slot addrs consumer-flushed at least once
} // ONE mutex -> "present?" and "flushed?" mutate atomically

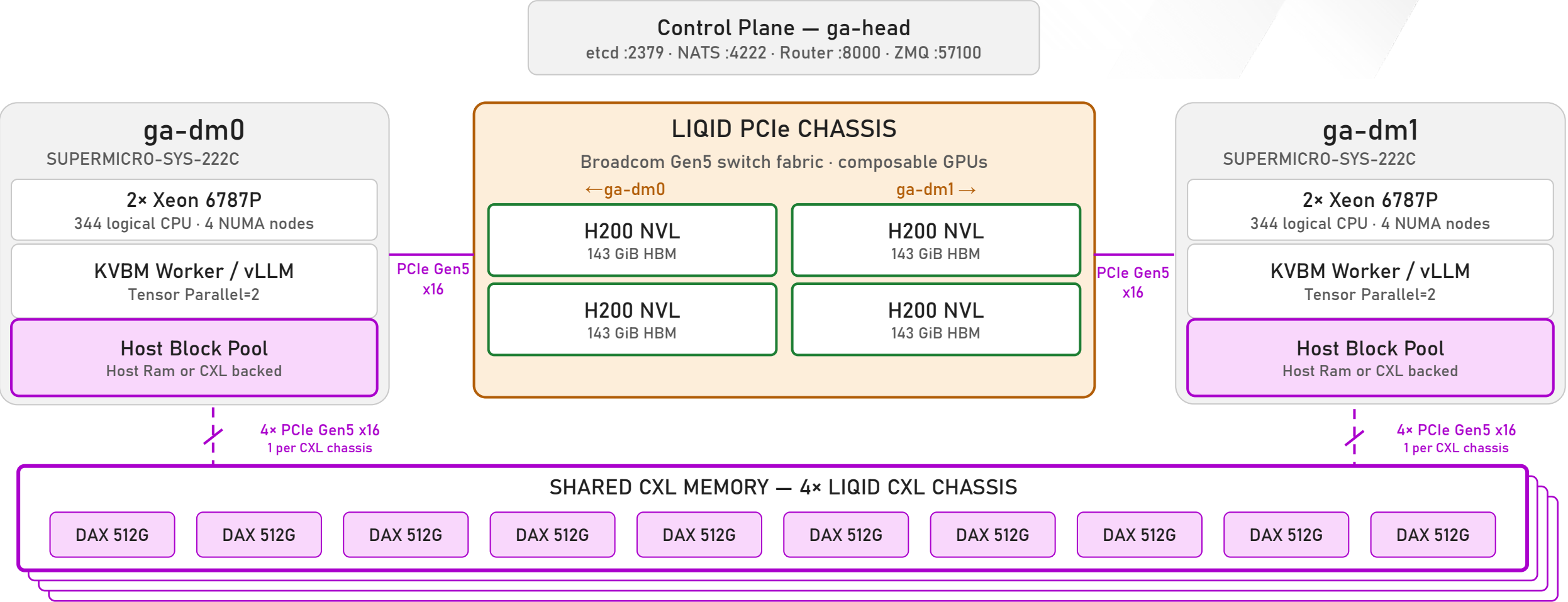
pub fn lookup_shards(&self, hashes: &[BlockHash], tp: usize) {
    let RegistryInner { blocks, flushed } = &mut *self.inner.lock().unwrap();
    // insert() is true when the addr was NOT already present (first onboard,
    // or first after an eviction re-armed it):
    let needs_flush = Self::flush_keys(d).into_iter()
        .fold(false, |acc, addr| flushed.insert(addr) || acc);
    let mut d = d.clone();
    d.skip_consumer_flush = !needs_flush; // set on the CLONE only
}
// Eviction re-arms the flush: flushed.remove(&addr);
```

Impact

KV is content-addressed, so an unchanged FAM slot stays coherent after the first flush; eviction is the only re-arm signal. Per-onboard flush cost fell from ~3.7 ms (~59% of onboard) to ~1.6-1.9 ms (~20-24%) — hot blocks were being re-flushed ~30x each before this.

Great Abaco Cluster

As configured for Dynamo workloads



Experimental Dataset Layout

Synthetic Dataset with >90% reuse rate

1. When solving coding problems, Claude Code generates prompts with 92–97% shared prefix¹

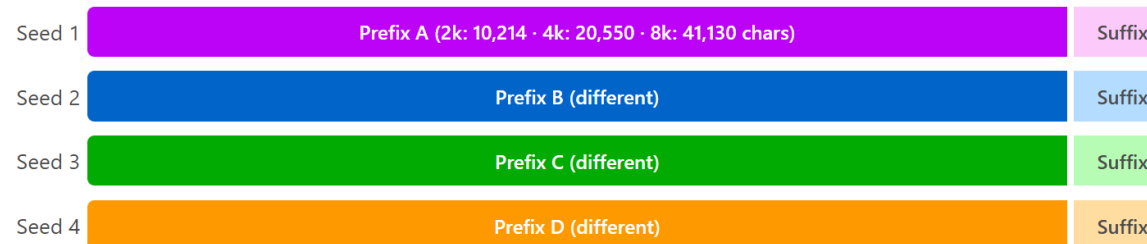
2. When multiple users interact with the same underlying data, there will be matching KV blocks

3. Larger prompts have more KV blocks

1

Generate 4 batches of 256 entries with different seeds

Each seed produces a unique shared prefix (~91–94%). Within each batch, 256 entries share that prefix with unique random suffixes (~6–9%).



0% prefix overlap between groups. Each seed generates completely independent random tokens for the shared prefix.

▼ interleave (round-robin)

2

Interleaved dataset: 1,024 entries

Entries from all 4 groups are interleaved in round-robin order, simulating mixed production traffic.

Entry order in file:



Every 4th entry shares a prefix: entries 0, 4, 8, ... share Prefix A; entries 1, 5, 9, ... share Prefix B; etc.

Interleaving creates **maximum cache pressure**: consecutive requests use different prefixes, forcing cache evictions on every request group switch. This is more realistic than contiguous blocks of 256.

- [\[1\] Context Engineering & Reuse Pattern Under the Hood of Claude Code](#)

KV Capacity Outruns Host DRAM

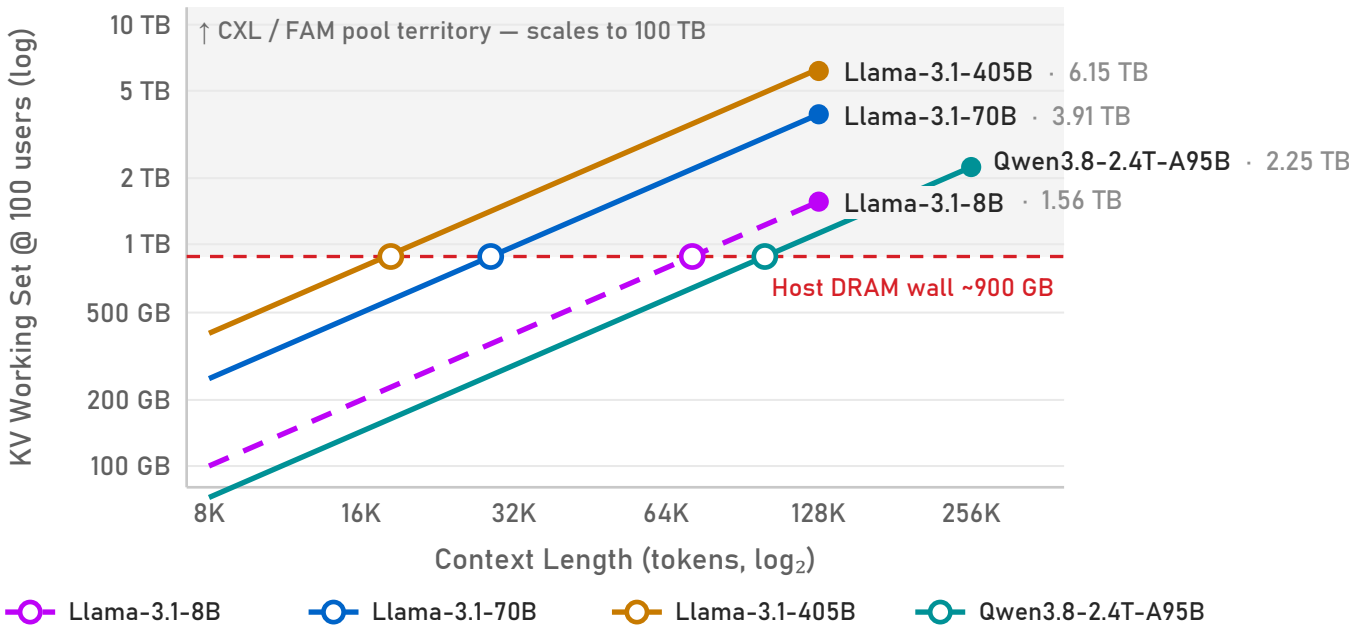


Every frontier model's native context window exceeds what ~900 GB of host DRAM can serve at 100 concurrent users

Per-token KV is *not* what grows — Qwen3.8 needs just 92 KB/token, under Llama-3.1-8B's 128 KB. The **working set** does: 8K → 256K contexts at 100+ concurrency clears the ~900 GB host-DRAM wall well inside every model's supported context.

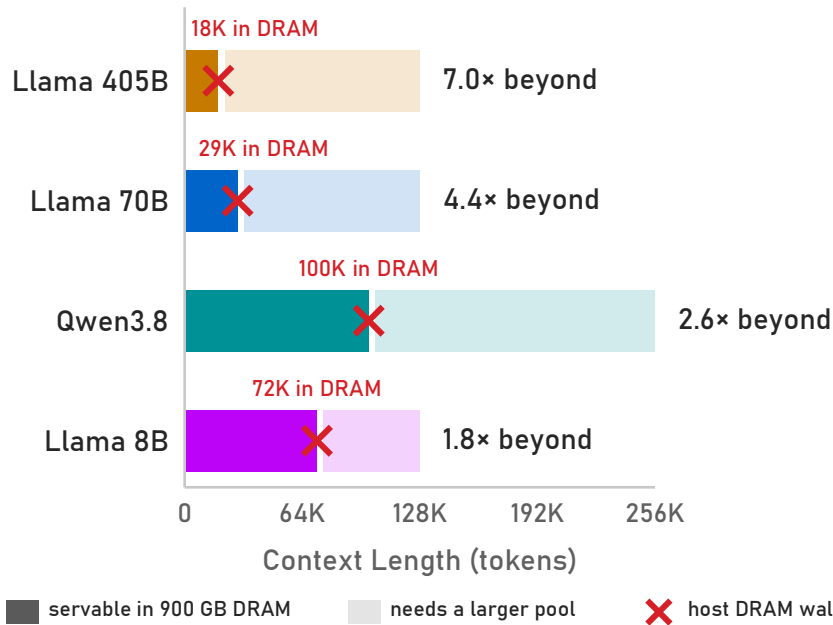
KV Working Set vs Context Length (100 concurrent users)

Log-log; each line ends at that model's native context window. Above the red line — the shaded region — only a pooled tier holds the working set. All four models end above it.



How Far Past the Wall Each Model Reaches

Solid = context servable inside a 900 GB DRAM cache at 100 users; faded = the rest of the native window. The X marks the wall.

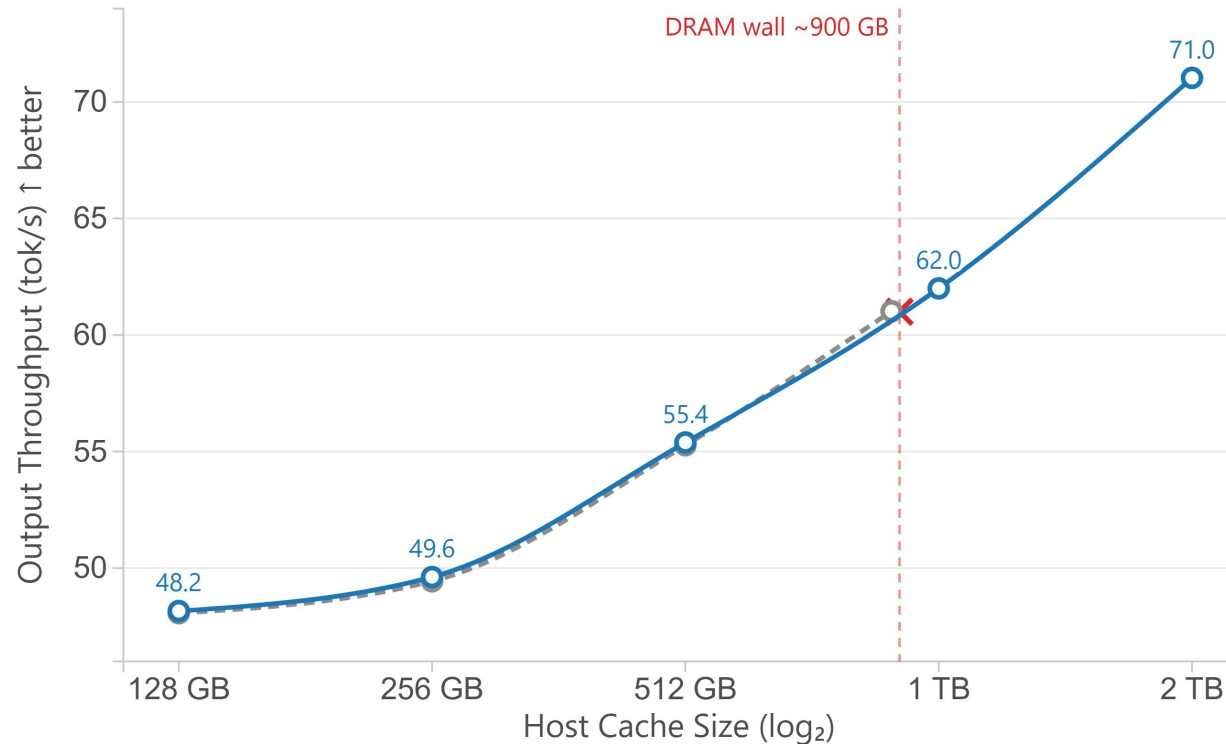


Dynamo KV Offload

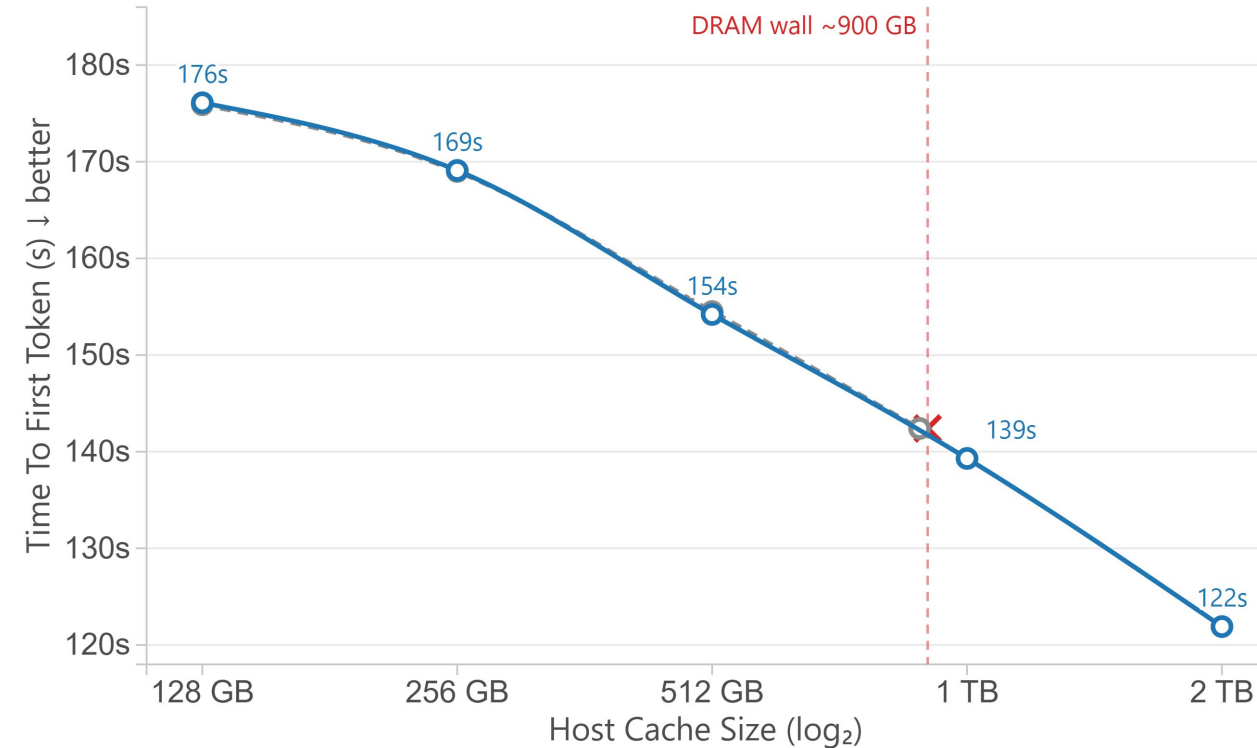
Larger offload pools improve performance - FAM memory within 99% of host DRAM



Output Throughput vs Host Cache Size



Time To First Token vs Host Cache Size



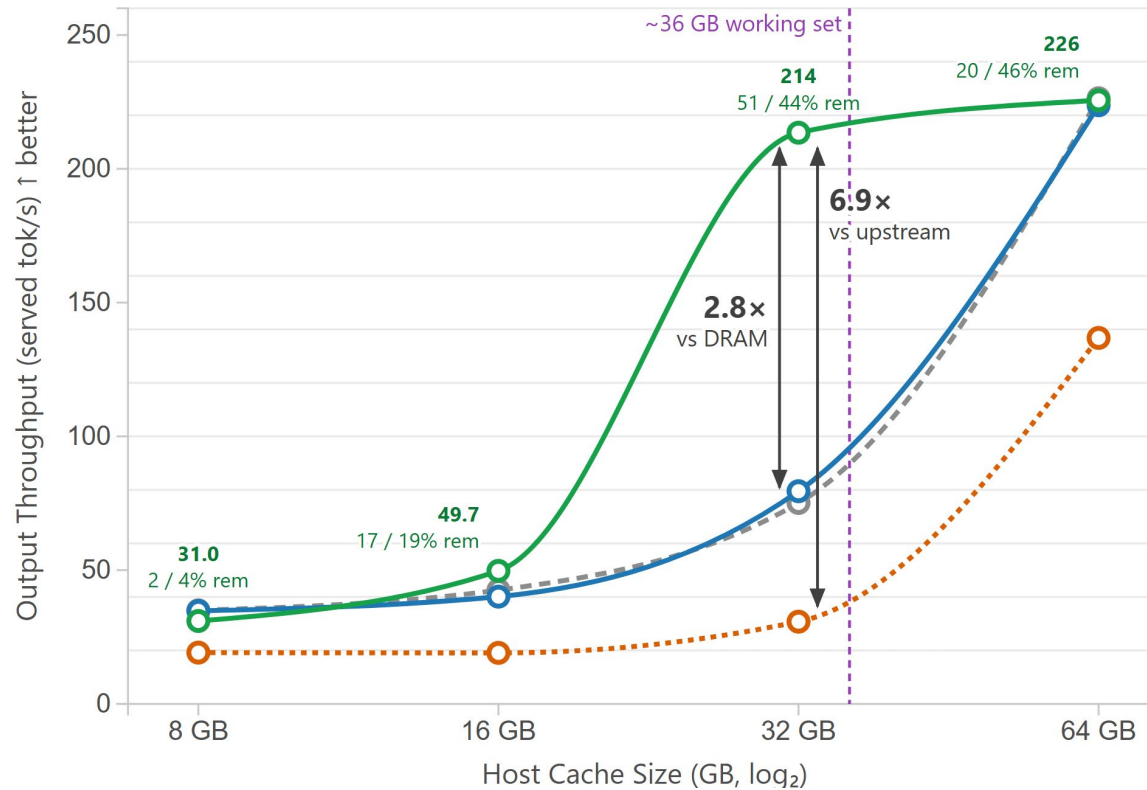
-○- DRAM host cache -○- CXL host cache × beyond DRAM capacity

- Profile A, Llama3-70B, 256 unique groups, concurrency 64, 2560 total requests, GPU cache on

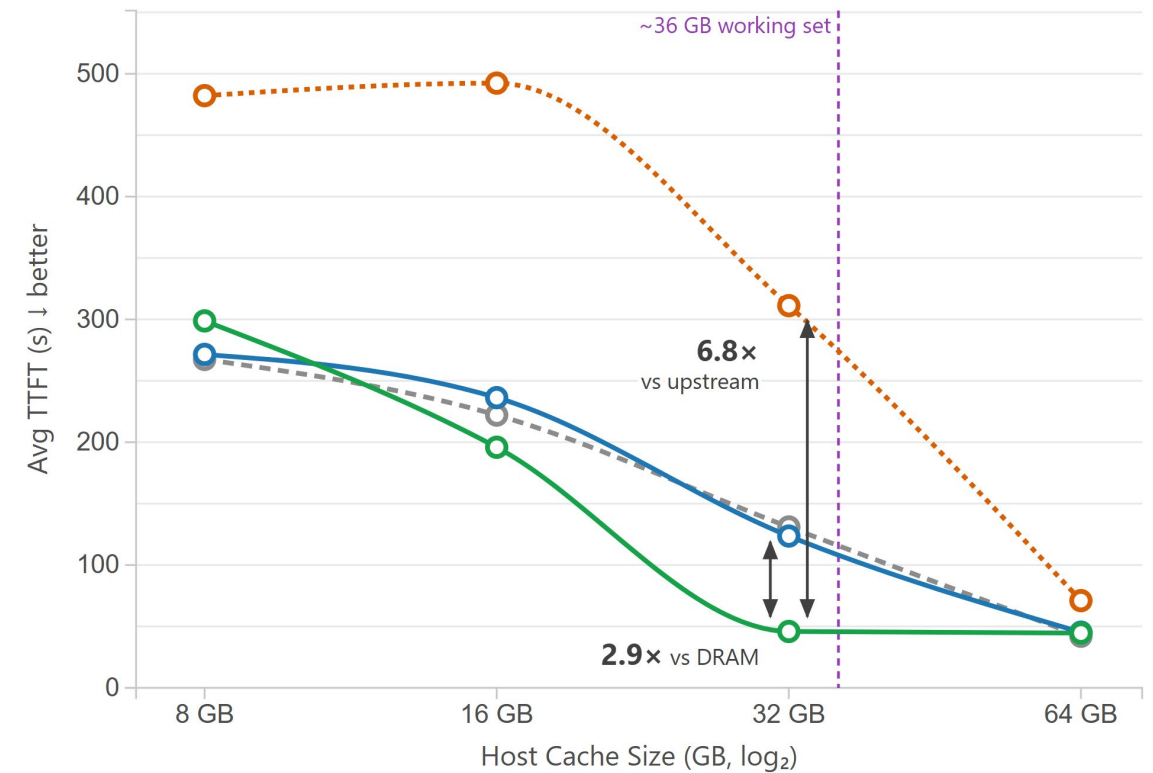
With Sharing, a Small Cache Behaves Like Large One

Time to first token (TTFT) and Throughput approach full cache performance early.
Shared, reused prefixes increase effective cache capacity and improve GPU utilization

Output Throughput vs Host Cache Size



Average TTFT vs Host Cache Size



---○--- DRAM, stock upstream -○- DRAM (no CXL) -○- CXL (no sharing) -○- CXL sharing

LLama3-70B • 4 Unique prefix groups • 90% shared • GPU cache off

Summary: KV Cache Scaling



1. Reuse sets the potential benefit
 - Capacity and bandwidth determine how much reuse improves latency and throughput.
2. CXL/FAM expands the KV cache beyond host DRAM
 - Shared memory lets compute and memory scale independently as KV working sets grow.
3. Cross-host reuse requires software support
 - Dynamo combines CXL-backed blocks, a shared registry, and producer/consumer coherency fences.
4. Experiments show the value of larger and shared pools
 - Llama3-70B tests show better TTFT and throughput, including a synthetic high-reuse workload.

Shared CXL memory expands effective KV cache capacity and reduces repeated prefill work.

Q&A

Please share your questions in the
Question Box



Thank You

www.ComputeExpressLink.org