

Native Tiered KV Cache Offloading in vLLM

From storage offloading to disaggregated serving

Or Ozeri · IBM Research
PyTorch Conference 2026

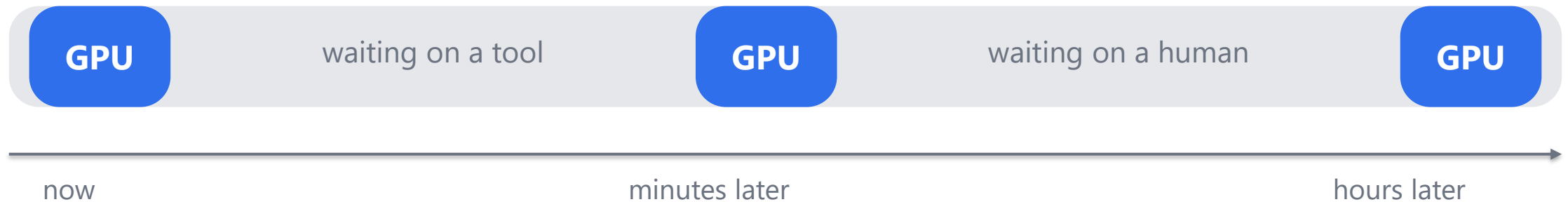


Connect on LinkedIn

Agentic sessions don't end. They pause.

- A session runs, then waits — minutes, hours, days
- Evict its KV, and the next turn pays a full prefill

One agentic session



GPU needed for seconds. KV needed for hours.

Two reasons to move KV out of the GPU

Capacity

offload

- More sessions than HBM can hold
- CPU, disk, object storage
- Reload instead of recompute

Mobility

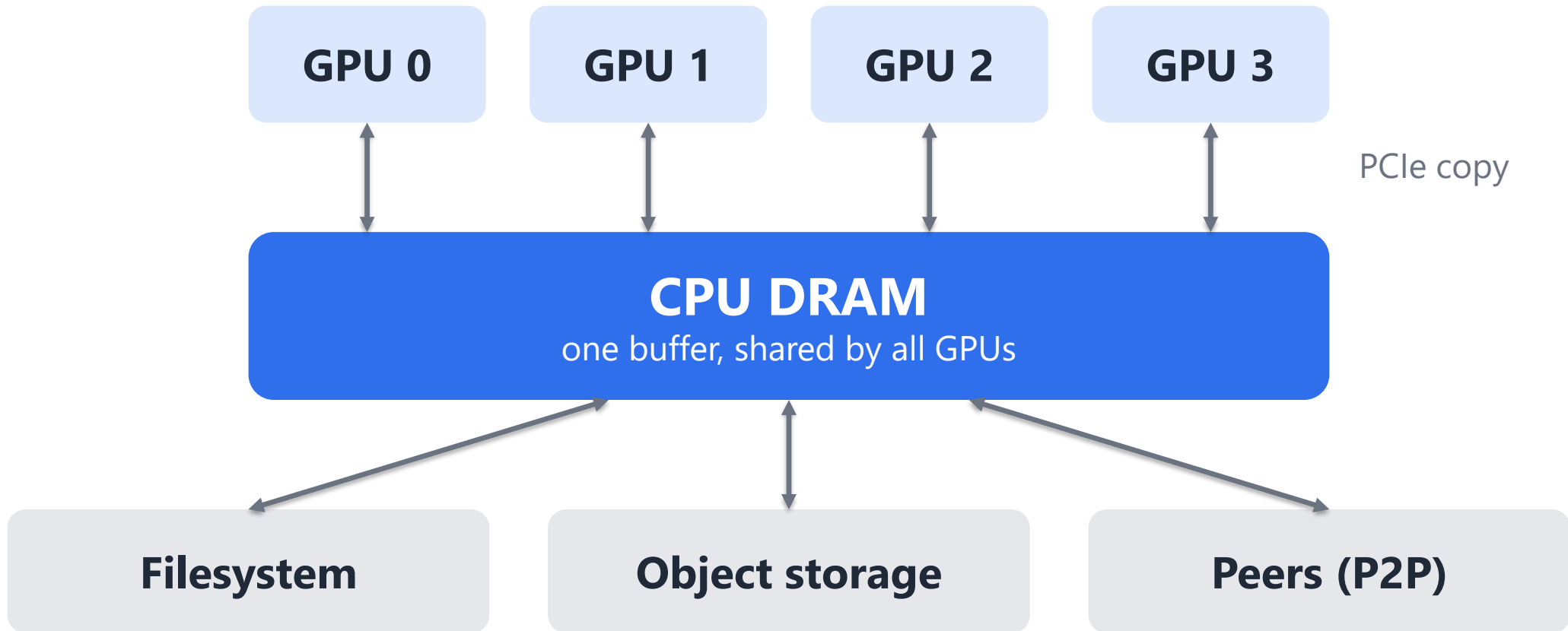
move between instances

- Prefill/decode disaggregation
- Load balancing

Same mechanism for both

The main design decision

All KV data flows through CPU memory



Why route everything through the CPU?

GPU $\rightleftarrows$ CPU

the only hop the GPU sees

- **Just a copy** to host memory
- **Minimal overhead** on the GPU

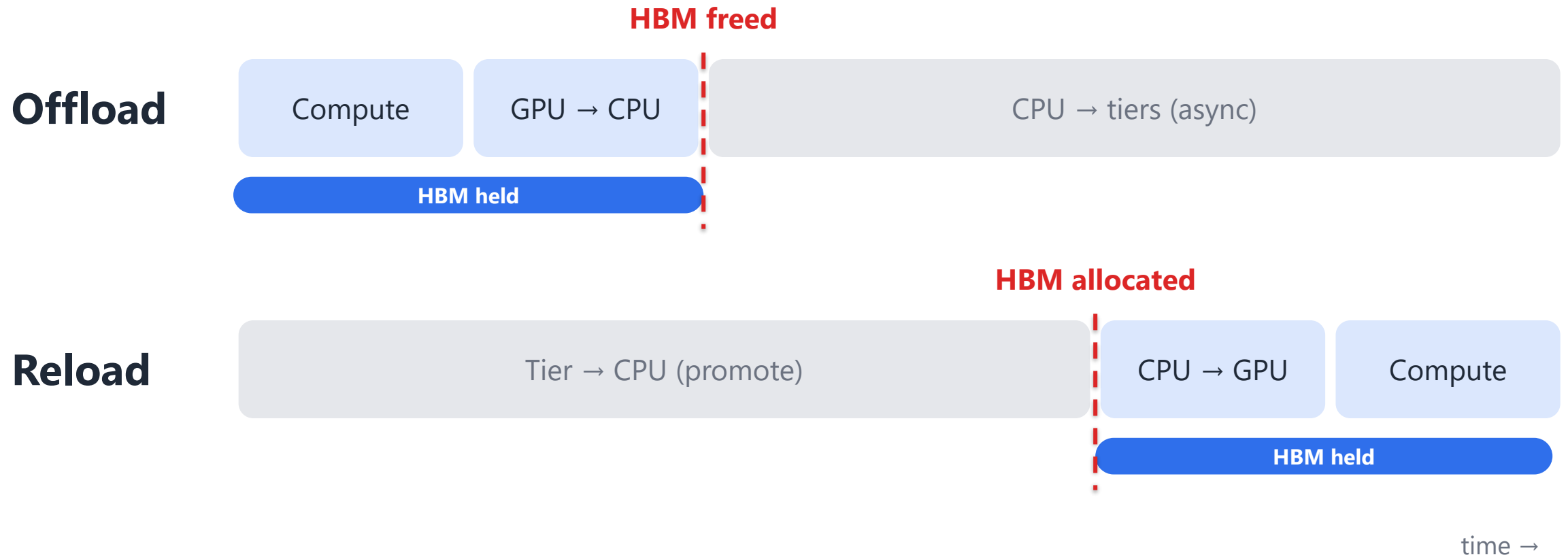
CPU $\rightleftarrows$ tiers

where secondary tiers plug in

- **Consolidated I/O** — fewer, bigger I/Os
- **Accelerator-agnostic** — CPU code
- **Canonical layout** — any node reads it

Simple for the GPU. Simple for the tiers.

GPU memory is held only for the fast hop



- Offload: free HBM as soon as CPU has the data
- Reload: allocate HBM only once CPU has the data

Three places to plug in

Connector

Your connector
scheduler + worker hooks

Model-aware code: **you**

Copy: **GPU blocks, per worker**

Offloading

Your OffloadingSpec
manager + worker

Model-aware code: **vLLM**

Copy: **GPU blocks, per worker**

Tiering

Your SecondaryTierManager
one class

TieringOffloadingSpec
is an OffloadingSpec

Model-aware code: **vLLM**

Copy: **CPU blocks, per instance**

OffloadingConnector
is a connector

Connector API
KVConnectorBase_V1

Offloading makes your code model-agnostic. Tiering also makes it accelerator-agnostic.

Tiers that ship today

- **Filesystem** — local or shared disk; shared mount = shared cache
- **Object storage** — S3-compatible
- **P2P** — CPU-to-CPU over RDMA
 - CPU-based prefill/decode disaggregation
 - Streams KV while prefill is still running
- **KVCR** — Dynamo's KV cache runner

used by **llm-d**

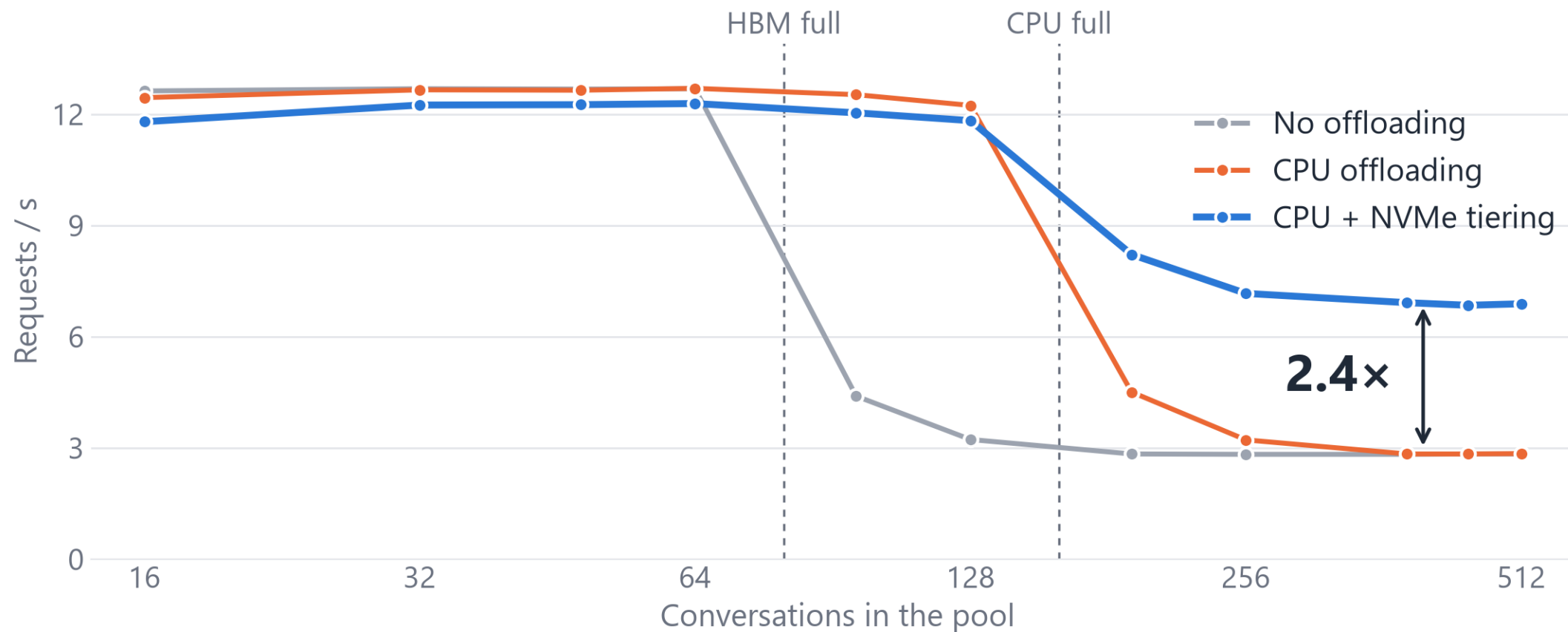


llm-d P2P blog

built by **Dynamo**

llm-d and Dynamo both build on vLLM tiering

What it buys you



Qwen3.6-35B-A3B · 2×H100 (TP=2) · NVMe filesystem tier · 12K prompt + 4K × 8 rounds · 64 concurrent

Turning it on

```
vllm serve Qwen/Qwen3.6-35B-A3B \
  --kv-transfer-config '{
    "kv_connector": "OffloadingConnector",
    "kv_role": "kv_both",
    "kv_connector_extra_config": {
      "spec_name": "TieringOffloadingSpec",
      "cpu_bytes_to_use": 107374182400,
      "secondary_tiers": [{"type": "fs", "root_dir": "/mnt/kv-cache"}]
    }
  }'
```

Connector → **spec** → **tier** — the three layers from before

Per-tier metrics on /metrics: hit rate, latency, bytes, CPU usage

Writing your own tier

```
class SecondaryTierManager(ABC):
    def lookup(self, key, req_context) -> LookupResult: ... # HIT/MISS/RETRY
    def submit_store(self, job_metadata: TransferJob) -> None: ...
    def submit_load(self, job_metadata: TransferJob) -> None: ...
    def get_finished_jobs(self) -> Iterable[JobResult]: ...
    # + on_new_request(), drain_jobs()
```

Each job points into the CPU buffer

```
class TransferJob:
    job_id: int
    keys: Collection[OffloadKey]
    chunk_ids: np.ndarray # CPU slots
```

Load it from your own package

```
"secondary_tiers": [{
    "type": "MyTier",
    "module_path": "my_pkg.my_tier"
}]
```

No GPU code. No fork. Reads and writes the CPU buffer directly.

Takeaways

- **CPU memory is the hub** for all KV movement
- **Tiering is the easiest way in** — model- and accelerator-agnostic
- **Already in use** — llm-d (P2P), Dynamo (KVCR)

Thank you! Questions?

vllm.ai/blog/2026-09-10-tiered-kv-offloading



Read the blog