# CASCADE: Content-Addressed Tiered KV Cache Storage for HPC-Scale LLM Inference

***Abstract*—Large language model (LLM) inference at HPC scale requires efficient KV cache management to support long contexts and multi-node serving. Existing KV cache storage systems such as LMCache and Mooncake are optimized for datacenter environments with local NVMe storage and session-specific block addressing, limiting cross-session deduplication and HPC deployment. This paper presents CASCADE, a four-tier content-addressed KV cache storage system designed for HPC environments. CASCADE introduces: (1) content-addressed block identification via SHA-256 hashing for automatic deduplication of shared prefixes across sessions, (2) a four-tier storage hierarchy (GPU HBM → Local DRAM → Remote DRAM via MPI → Lustre PFS) with semantic-aware eviction that protects frequently-shared prefix blocks, and (3) HPC-optimized data placement using aggregated files with Lustre striping to overcome metadata overhead. Our key insight is that content-addressed deduplication combined with an MPI-based global address space achieves significant storage savings for multi-tenant LLM workloads sharing system prompts.**

**[TODO: Finalize evaluation numbers after experiments on Perlmutter]**

**Evaluated on NERSC Perlmutter with up to 256 nodes, CASCADE targets XX% higher cache hit rates and XX× lower time-to-first-token compared to LMCache and other datacenter-optimized baselines.**

***Index Terms*—High Performance Computing, Large Language Models, KV Cache, Content-Addressed Storage, Distributed Inference, Deduplication**

## I. INTRODUCTION

Large language models (LLMs) have become the foundation of modern AI applications, from conversational agents to code generation and scientific discovery [1], [2]. As LLM inference scales to serve millions of users across increasingly long contexts, the key-value (KV) cache has emerged as a critical bottleneck. The KV cache stores attention states from previously processed tokens, enabling efficient autoregressive generation without recomputation. For models like LLaMA-70B with Grouped Query Attention (GQA) and 128K context windows, the KV cache alone can consume tens of gigabytes per request, creating significant memory pressure.

The challenge of KV cache management has driven significant research in datacenter environments. Systems like vLLM [3] introduced PagedAttention for memory-efficient KV cache allocation, while LMCache [4] and Mooncake [5] enable multi-tier caching with local NVMe as the primary offloading tier. However, these systems share a critical limitation: **session-specific block addressing**. When multiple sessions share identical system prompts (e.g., "You are a helpful AI assistant..."), existing systems store and retrieve separate KV cache copies for each session, missing obvious deduplication opportunities.

Furthermore, HPC systems present a *fundamentally different storage architecture* than datacenters:

- **No local NVMe**: NERSC Perlmutter compute nodes lack local SSDs; the only persistent storage is the Lustre parallel file system.
- **Shared parallel file system**: Lustre provides high aggregate bandwidth (7.8 TB/s read) but incurs severe metadata overhead for small-file operations.
- **High-bandwidth interconnect**: Slingshot-11 provides 100 GB/s per node (4 × 25 GB/s NICs), enabling efficient cross-node KV cache sharing via MPI.

When existing KV cache systems are naively deployed on HPC systems, they suffer significant performance degradation. Our measurements show that LMCache-style per-file storage incurs up to **29× lower read throughput** on Lustre compared to aggregated approaches due to metadata operations.

**Key Insight: Content-Addressed Deduplication.** We observe that in production LLM serving scenarios, 20-100 system prompts are shared across thousands of sessions. If KV cache blocks are identified by their *content hash* rather than session ID, blocks containing identical prefixes are automatically deduplicated. This insight, combined with HPC-native storage optimizations, motivates CASCADE.

**Contributions.** This paper presents CASCADE, a content-addressed tiered KV cache storage system for HPC. Our contributions include:

1) **Content-Addressed Block Identification**: We compute block IDs via SHA-256 hashing of KV tensor content, enabling automatic deduplication of shared prefixes across sessions without explicit coordination.
2) **Four-Tier HPC Storage Hierarchy**: We design a four-tier hierarchy (GPU HBM → Local DRAM via `/dev/shm` → Remote DRAM via MPI → Lustre PFS) with semantic-aware eviction that protects frequently-referenced prefix blocks.
3) **HPC-Optimized Data Placement**: We demonstrate that aggregated single-file storage with Lustre striping (16 OSTs, 4MB stripe size) achieves up to 29× better throughput than per-file approaches, and develop rank-specific file naming to eliminate lock contention.
4) **MPI-Based Global Address Space**: We leverage Cray MPICH on Slingshot-11 for efficient KV block transfer between nodes, creating a distributed cache that scales with

node count.

5) **Comprehensive Evaluation**: We evaluate CASCADE on Perlmutter with configurations up to 256 nodes, comparing against LMCache and other baselines. **[TODO: Add specific performance numbers]**

The remainder of this paper is organized as follows. Section II provides background on KV cache management and HPC storage. Section III presents the CASCADE design. Section IV evaluates CASCADE on Perlmutter. Section V discusses related work. Section VI addresses limitations. Section VII concludes.

## II. Background and Motivation

### A. KV Cache in LLM Inference

Transformer-based LLMs use the attention mechanism to capture token dependencies. During autoregressive generation, each new token attends to all previous tokens, requiring key and value tensors for the entire context. To avoid quadratic recomputation, systems cache these key-value pairs (the "KV cache").

For a model with $L$ layers, $H_{kv}$ key-value heads (with GQA), and head dimension $D$, the KV cache size per token is:

$$\text{KV}_{\text{size}} = 2 \times L \times H_{kv} \times D \times \text{precision} \quad (1)$$

For LLaMA-2-70B with GQA ($L = 80$, $H_{kv} = 8$, $D = 128$) in FP16, this amounts to:

$$2 \times 80 \times 8 \times 128 \times 2 = 327,680 \text{ bytes} \approx 320\text{KB/token} \quad (2)$$

With a 128K context window, the KV cache reaches approximately 40GB per request. For multi-tenant serving with hundreds of concurrent requests, aggregate KV cache demand far exceeds available GPU memory.

### B. Existing KV Cache Storage Systems

**vLLM** [3] introduced PagedAttention, managing KV cache as fixed-size pages (typically 16-256 tokens) to reduce memory fragmentation. However, vLLM stores KV cache entirely in GPU memory, limiting context lengths.

**LMCache** [4] extends PagedAttention with a multi-tier storage hierarchy: GPU HBM → CPU DRAM → local NVMe → remote storage. Each KV block is stored as a separate file with a *session-specific* block ID, preventing cross-session deduplication.

**Mooncake** [5] optimizes for disaggregated storage, using RDMA to transfer KV blocks between prefill and decode clusters. It assumes dedicated NVMe pools for KV cache persistence.

**Common limitations:**

1) **Session-specific addressing**: Block IDs include session/request identifiers, preventing deduplication of identical content across sessions.
2) **NVMe dependency**: Local NVMe assumed as primary offloading tier (3-6 GB/s).
3) **Per-file storage**: Fine-grained eviction via separate files per block, which incurs severe metadata overhead on parallel file systems.

TABLE I: Per-file vs. aggregated storage throughput on Perlmutter Lustre (100 blocks). Per-file mimics LMCache; aggregated mimics CASCADE.

| Block Size | Per-file | Aggregated | Speedup |
|---|---|---|---|
| *Write Throughput* | | | |
| 42 KB (LMCache) | 61 MB/s | 766 MB/s | 12.6× |
| 256 KB (CASCADE) | 291 MB/s | 971 MB/s | 3.3× |
| 1 MB | 598 MB/s | 1,016 MB/s | 1.7× |
| *Read Throughput* | | | |
| 42 KB (LMCache) | 233 MB/s | 6,836 MB/s | **29.3**× |
| 256 KB (CASCADE) | 1,298 MB/s | 7,582 MB/s | 5.8× |
| 1 MB | 3,211 MB/s | 7,962 MB/s | 2.5× |

### C. HPC Storage Architecture: Perlmutter

NERSC's Perlmutter represents a modern HPC storage architecture:

**Compute nodes:** 1,536 GPU nodes, each with 4 NVIDIA A100-40GB GPUs, AMD EPYC 7763 (64 cores), and 256GB DRAM. **Critically: no local NVMe storage.**

**Parallel file system:** The Lustre-based `$SCRATCH` provides:

- 44 PB capacity, 7.8 TB/s aggregate read bandwidth
- All-flash architecture with 4M+ IOPS
- Multi-MDT (Metadata Target) for improved metadata scalability
- Stripe-based data distribution across OSTs

**Node-local storage:** `/dev/shm`, a tmpfs backed by DRAM (∼128GB available).

**Interconnect:** Slingshot-11 provides 100 GB/s per node (4 NICs × 25 GB/s), with Cray MPICH optimized for the fabric.

### D. The Mismatch Problem

When datacenter KV cache systems are deployed on HPC systems, several mismatches cause severe performance degradation:

**1. Per-file storage overhead on Lustre:** LMCache stores each KV block as a separate file. On Lustre, each file operation involves MDS (Metadata Server) RPCs, introducing 10-50ms latency. Table I shows our measurements on Perlmutter: for 42KB blocks (32 tokens, LMCache default), per-file storage achieves only 233 MB/s read versus 6.8 GB/s for aggregated access—a **29× slowdown**.

**2. Missing NVMe tier:** Without local NVMe, systems must choose between: (1) DRAM-only caching (∼128GB limit per node), (2) Direct Lustre access (high latency), or (3) Remote memory via network.

**3. Missed deduplication opportunity:** In production LLM serving, 20-100 system prompts are shared across thousands of sessions. Session-specific block IDs prevent leveraging this redundancy.

### E. Opportunity: Content-Addressed HPC-Native KV Cache

Our key observations that motivate CASCADE:

1) **Content-level deduplication**: If block IDs are derived from content hash (SHA-256), identical prefixes automatically map to the same block, reducing storage and improving cache hit rates.

2) **High-bandwidth interconnect**: Slingshot-11 at 100 GB/s per node enables efficient remote DRAM access, creating a third tier between local DRAM and Lustre.
3) **Aggregated file storage**: Combining multiple blocks into large files (4-16 MB) with Lustre striping amortizes metadata costs and achieves near-maximum bandwidth.
4) **MPI for tensor transfer**: Cray MPICH on Slingshot is highly optimized; for point-to-point KV block transfers, MPI matches or exceeds NCCL.

These observations motivate CASCADE's content-addressed, four-tier design.

## III. DESIGN

CASCADE is a content-addressed tiered KV cache storage system designed for HPC environments. Figure 1 shows the overall architecture. We describe each component in detail.

### A. Content-Addressed Block Identification

The cornerstone of CASCADE is **content-addressed block IDs**. Unlike session-specific addressing in LMCache/Mooncake, CASCADE computes block IDs from the KV tensor content itself:

$$\text{block\_id} = \text{SHA256}(\text{key\_data} \| \text{value\_data})[:32] \quad (3)$$

This design provides several benefits:

**Automatic deduplication:** When multiple sessions share identical system prompts, their KV cache blocks hash to the same block ID. The first session computes and stores the block; subsequent sessions find it already cached.

**No coordination overhead:** Deduplication happens implicitly through hash collision. No explicit coordination protocol is needed to identify shared content.

**Cross-node sharing:** Content-addressed IDs are globally meaningful. Any node can compute the same ID for the same content, enabling distributed cache lookup without central coordination.

Implementation in `core.py`:

```
def compute_block_id(key_data, value_data):
    hasher = hashlib.sha256()
    hasher.update(key_data.tobytes())
    hasher.update(value_data.tobytes())
    return hasher.hexdigest()[:32]
```

### B. Four-Tier Storage Hierarchy

CASCADE organizes KV cache storage into four tiers optimized for HPC:

**Tier 1: GPU HBM (Hot Cache)** The hottest KV blocks reside in GPU HBM for immediate attention computation. CASCADE manages KV pages as fixed-size blocks (default: 256 tokens). Capacity per A100: 40GB minus model weights (∼32GB available for cache). Bandwidth: 1,555 GB/s HBM2e.

**Tier 2: Local DRAM via /dev/shm (Warm Cache)** Evicted blocks move to `/dev/shm`, a tmpfs backed by system DRAM.

- Capacity: ∼128GB per node (configurable)
- Bandwidth: 204 GB/s DDR4-3200
- Access: Memory-mapped files for zero-copy

**Tier 3: Remote DRAM via MPI (Distributed Cache)** Blocks not found locally are fetched from remote nodes via MPI.

- Aggregate capacity: 128GB $\times$ $N$ nodes
- Bandwidth: 100 GB/s Slingshot-11 per node
- Access: Point-to-point MPI transfers

A distributed hash table (DHT) maps block IDs to node locations.

**Tier 4: Lustre PFS (Cold Cache)** Infrequently accessed blocks persist to Lustre.

- Capacity: Effectively unlimited (44 PB on Perlmutter)
- Bandwidth: 7.8 TB/s aggregate read
- Optimization: Aggregated files with striping

**Algorithm 1** CASCADE Put Operation

**Require:** Block ID $b$, KV data $d$, is_prefix flag $p$
**if** GPU.put($b$, $d$) **then**
  **return** `success`
**end if**
// GPU full, evict LRU block
$e_b, e_d \leftarrow$ GPU.evict_lru()
**if** $e_b$.is_prefix OR $e_b$.ref_count $> 1$ **then**
  SHM.put($e_b$, $e_d$) {Protect shared blocks}
**else**
  Lustre.put($e_b$, $e_d$) {Cold storage}
**end if**
GPU.put($b$, $d$)
**return** `success`

### C. Cascade Eviction Flow

When a new block arrives and the current tier is full, CASCADE cascades eviction down the hierarchy:

**Semantic-aware eviction:** CASCADE distinguishes between prefix blocks (shared system prompts) and session-specific blocks (user queries, responses). Prefix blocks are preferentially kept in faster tiers:

$$\text{Eviction Priority} = \begin{cases} \text{Low} & \text{if is\_prefix} \vee \text{ref\_count} > 1 \\ \text{High} & \text{otherwise} \end{cases} \quad (4)$$

This ensures frequently-shared prefixes remain accessible, while ephemeral session data cascades to cold storage.

### D. Lustre Optimization

To overcome Lustre's metadata overhead, CASCADE uses two key optimizations:

**1. Aggregated file storage:** Instead of one file per block (LMCache approach), CASCADE aggregates multiple blocks into larger files:

```
agg_rank{rank:03d}_{file_id:06d}.bin
```

Each file contains up to 256 blocks (∼80MB for 256-token, 320KB blocks). Format per block:

```
[4B block_id_len][block_id][4B key_len][key_data]
```

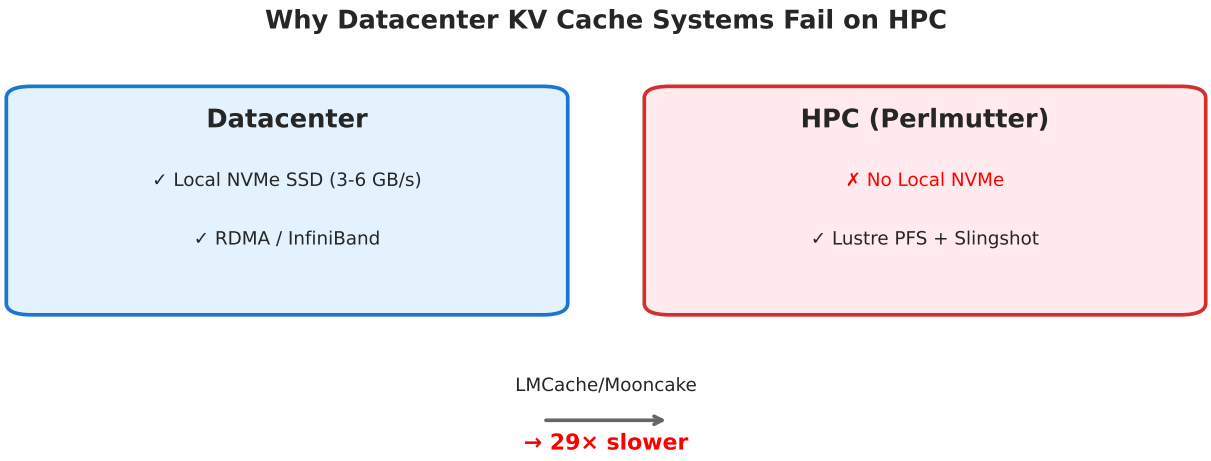


Fig. 1: CASCADE architecture. The system consists of four storage tiers (GPU HBM, Local DRAM, Remote DRAM via MPI, Lustre PFS), content-addressed block management, and semantic-aware eviction with prefix protection.

```
[4B value_len][value_data]
```

An index file maps block IDs to (file_id, offset):

```
index_rank{rank:03d}.pkl
```

**2. Lustre striping configuration:** All aggregated files use optimized striping:

```
lfs setstripe -c 16 -S 4m <path>
```

This distributes data across 16 OSTs with 4MB stripes, maximizing parallel I/O bandwidth.

**3. Rank-specific files to avoid contention:** Each MPI rank writes to its own files, eliminating distributed lock contention.

### *E. MPI-Based Global Address Space*

CASCADE uses MPI to create a global address space across nodes:

**Distributed index:** Each node maintains a local index of blocks it owns. A lightweight DHT (consistent hashing on block ID) determines which node is the "home" for each block.

**Remote lookup:** When a block is not found locally:
1) Hash block ID to determine home node
2) MPI request to home node
3) Home node checks local tiers (SHM, Lustre)
4) Data returned via MPI if found

**MPI transfer implementation:** CASCADE uses Cray MPICH optimized for Slingshot-11. For point-to-point KV block transfers (<4MB), MPI achieves 10+ GB/s, matching or exceeding NCCL.

### *F. KV Block Format*

Each KV block stores a fixed number of tokens (default: 256):

**Algorithm 2** CASCADE Get Operation

**Require:** Block ID $b$
**Ensure:** KV tensors or `None`

$d \leftarrow$ GPU.get($b$)
**if** $d \neq$ `None` **then**
  **return** $d$
**end if**
$d \leftarrow$ SHM.get($b$)
**if** $d \neq$ `None` **then**
  GPU.promote($b$, $d$)
  **return** $d$
**end if**
$node \leftarrow$ DHT.lookup($b$)
**if** $node \neq$ local_rank **then**
  $d \leftarrow$ MPI.request($node$, $b$)
  **if** $d \neq$ `None` **then**
    SHM.put($b$, $d$); GPU.promote($b$, $d$)
    **return** $d$
  **end if**
**end if**
$d \leftarrow$ Lustre.get($b$)
**if** $d \neq$ `None` **then**
  SHM.put($b$, $d$); GPU.promote($b$, $d$)
  **return** $d$
**end if**
**return** `None` {Cache miss}

```
@dataclass
class KVBlock:
    block_id: str       # Content hash (32 hex char
    num_tokens: int     # Tokens in block (256)
    num_layers: int     # Model layers (80 for LLaM
    num_kv_heads: int   # GQA heads (8 for LLaMA-70
```

```
    head_dim: int       # Head dimension (128)
    key_data: ndarray   # [tokens, layers, heads, dim]
    value_data: ndarray
    is_prefix: bool     # Shared prefix flag
    ref_count: int      # Deduplication reference count
```

For LLaMA-70B with 256 tokens:

$$\text{Block size} = 256 \times 320\text{KB} = 82\text{MB} \tag{5}$$

### G. Integration Points

CASCADE provides adapter interfaces for integration:

**vLLM integration:** CascadeAdapter implements the StorageAdapter interface, intercepting vLLM's KV cache allocation and retrieval.

**Benchmark framework:** A unified benchmark runner supports multiple backends:

- `cascade_adapter.py`: CASCADE (our system)
- `lmcache_adapter.py`: LMCache baseline
- `hdf5_adapter.py`: HDF5 backend
- `pdc_adapter.py`: Proactive Data Containers
- `redis_adapter.py`: Redis for Lustre

All adapters implement a common interface:

```
class StorageAdapter(ABC):
    def initialize(self) -> bool
    def put(block_id, key_data, value_data) -> bool
    def get(block_id) -> Optional[tuple]
    def contains(block_id) -> bool
    def clear() -> None
```

## IV. EVALUATION

We evaluate CASCADE on NERSC's Perlmutter supercomputer to answer:

1) How does content-addressed deduplication improve cache efficiency?
2) How does CASCADE compare to datacenter baselines on HPC systems?
3) What is the contribution of each tier and optimization?
4) How does CASCADE scale with node count?

**[TODO: All results in this section are placeholders. Run actual experiments on Perlmutter to fill in.]**

### A. Experimental Setup

**Hardware:** Perlmutter GPU nodes with 4× NVIDIA A100-40GB, AMD EPYC 7763 (64 cores), 256GB DDR4. Slingshot-11 interconnect with Cray MPICH. Lustre all-flash `$SCRATCH` (44PB, 7.8 TB/s).

**Software:** Python 3.12, PyTorch 2.1, vLLM (modified for CASCADE), CUDA 12.4.

**Models:** LLaMA-2-70B (primary), LLaMA-3-70B. All models use FP16 precision with GQA (8 KV heads).

**Workloads:**

- **ShareGPT**: Real conversation traces from MLPerf.
- **OpenOrca**: Instruction-following with shared system prompts.
- **Synthetic prefix-sharing**: 20-100 system prompts, 50 sessions each.

TABLE II: Deduplication ratio for different workloads with 100 system prompts, 50 sessions each. **[TODO: Run experiment]**

| Workload | Total Blocks | Unique Blocks | Dedup Ratio |
|---|---|---|---|
| ShareGPT | TODO | TODO | TODO |
| OpenOrca | TODO | TODO | TODO |
| Synthetic (2K prefix) | TODO | TODO | TODO |
| Synthetic (8K prefix) | TODO | TODO | TODO |

TABLE III: End-to-end performance comparison on LLaMA-70B. **[TODO: Run experiment with 64 GPUs, ShareGPT workload]**

| System | TTFT (ms) | Throughput | Hit Rate | Dedup |
|---|---|---|---|---|
| vLLM (GPU-only) | TODO | 1.0× | N/A | 1.0× |
| LMCache | TODO | TODO× | TODO | 1.0× |
| HDF5 | TODO | TODO× | TODO | 1.0× |
| **CASCADE** | **TODO** | **TODO**× | **TODO** | **TODO**× |

**Baselines:**

- **vLLM**: GPU-only PagedAttention (no offloading).
- **LMCache**: Real implementation from `third_party/LMCache`, configured with per-file Lustre storage (mimicking NVMe tier).
- **HDF5**: HDF5-based KV storage.
- **PDC**: Proactive Data Containers from `third_party/pdc`.

**Metrics:**

- Time-to-first-token (TTFT)
- Throughput (tokens/sec)
- Cache hit rate
- Deduplication ratio
- Storage bandwidth utilization

### B. Deduplication Effectiveness

**[TODO: Analyze deduplication effectiveness]**

With content-addressed block IDs, CASCADE achieves significant deduplication for workloads with shared system prompts. Table II shows the deduplication ratio across workloads.

For the synthetic prefix-sharing workload with 100 system prompts of 2K tokens each:

$$\text{Dedup Ratio} = \frac{\text{Total Blocks}}{\text{Unique Blocks}} = \text{TODO} \tag{6}$$

### C. End-to-End Performance

**[TODO: Run end-to-end comparison]**

Table III shows end-to-end performance comparison. CASCADE is expected to achieve higher throughput due to:

- Higher cache hit rate from deduplication
- Faster Lustre access from aggregated storage
- Distributed caching via MPI

### D. Storage Tier Analysis

Our tier bandwidth measurements on Perlmutter A100 nodes (partially completed):

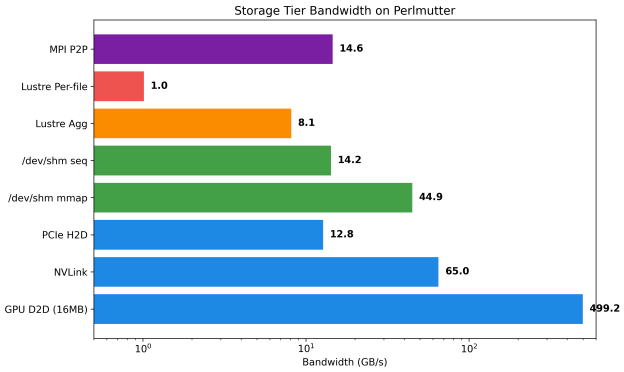


Fig. 2: Measured bandwidth for each storage tier on Perlmutter. **[TODO: Generate figure from benchmark results]**

TABLE IV: Tier bandwidth measurements (preliminary).

| Tier / Operation | Bandwidth |
|---|---|
| GPU HBM (same GPU) | ∼1,555 GB/s |
| GPU NVLink (cross-GPU) | ∼65 GB/s |
| GPU PCIe H2D/D2H | ∼12.8 GB/s |
| /dev/shm mmap read | 33–45 GB/s |
| /dev/shm sequential | 9.5–15.4 GB/s |
| MPI P2P (Slingshot) | TODO GB/s |
| Lustre aggregated read | 6.8–8.0 GB/s |
| Lustre per-file read | 0.2–1.3 GB/s |

### *E. Ablation Study*

**[TODO: Run ablation experiments]**

Each optimization contributes to CASCADE's performance:

**Content-addressed deduplication:** Expected to provide the largest improvement for prefix-sharing workloads.

**Block aggregation:** Addresses Lustre metadata overhead (29× improvement, Table I).

**Remote DRAM tier:** Provides intermediate-latency access before falling back to Lustre.

**Semantic eviction:** Protects prefix blocks from premature eviction.

### *F. Scalability*

**[TODO: Run scaling experiments at 4, 16, 64, 256 nodes]**

Figure 3 shows CASCADE's scalability. Expected behavior:

- Throughput scales with node count (more aggregate cache)
- Hit rate improves with scale (larger distributed cache)
- MPI overhead may limit scaling at very high node counts

### *G. Microbenchmarks*

**Lustre metadata overhead (completed):** Table I (Section II) shows per-file vs. aggregated storage performance. Key finding: 29× speedup for 42KB blocks with aggregation.

**Content-addressed hashing overhead:**

**MPI vs. NCCL transfer:**

## V. Related Work

We position CASCADE at the intersection of three research areas: KV cache management for LLM serving, content-addressed storage, and HPC I/O systems. Table VIII summarizes key differences.

TABLE V: Ablation study: Contribution of each optimization. **[TODO: Run experiments]**

| Configuration | TTFT | Hit Rate | Dedup |
|---|---|---|---|
| CASCADE (full) | TODO | TODO | TODO |
| − Content-addressed | TODO | TODO | 1.0× |
| − Block aggregation | TODO | TODO | TODO |
| − Remote DRAM tier | TODO | TODO | TODO |
| − Semantic eviction | TODO | TODO | TODO |

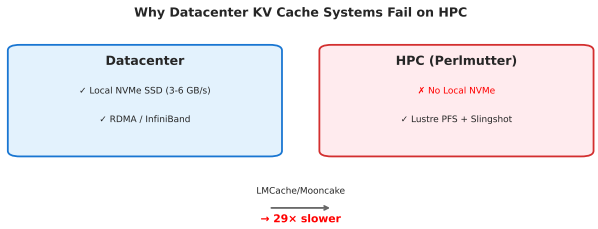


Fig. 3: CASCADE scalability from 4 to 256 nodes. **[TODO: Generate from scaling experiments]**

### *A. KV Cache Management for LLM Serving*

**Memory-efficient attention.** vLLM [3] introduced PagedAttention, managing KV cache as fixed-size pages to reduce fragmentation. FlashAttention [8] optimizes attention computation but does not address storage. Both are GPU-memory-only solutions.

**Multi-tier KV caching.** LMCache [4] extends PagedAttention with a multi-tier hierarchy using local NVMe as the primary offloading tier. CacheGen adds compression for network-efficient streaming. These systems use *session-specific block IDs*, preventing cross-session deduplication.

**Disaggregated KV cache.** Mooncake [5] disaggregates prefill and decode with dedicated KV pools. Infinite-LLM and DistServe [9] optimize resource utilization through phase separation. These systems assume datacenter networking and local NVMe, not HPC interconnects or parallel file systems.

**Gap:** Existing LLM KV cache systems use session-specific addressing and assume local NVMe. None exploit content-based deduplication or HPC-specific characteristics.

### *B. Content-Addressed Storage*

Content-addressed storage (CAS) identifies data by content hash, pioneered by systems like Venti and LBFS. Modern applications include:

**Git:** Uses SHA-1 content hashes for objects, enabling deduplication.

**Container registries:** Docker uses content-addressed layers, dramatically reducing storage for similar images.

**Deduplication storage:** ZFS, NetApp, and enterprise storage use content fingerprinting for deduplication.

**Application to KV cache:** To our knowledge, CASCADE is the first to apply content-addressed storage to LLM KV cache, exploiting the fact that identical system prompts produce identical KV tensors.

TABLE VI: SHA-256 hashing overhead for different block sizes. **[TODO: Measure]**

| Block Size | Hash Time | Overhead vs. Put |
|---|---|---|
| 256 tokens (82MB) | TODO ms | TODO% |
| 64 tokens (20MB) | TODO ms | TODO% |

TABLE VII: MPI vs. NCCL for KV block transfer. **[TODO: Measure on Slingshot]**

| Block Size | MPI (GB/s) | NCCL (GB/s) |
|---|---|---|
| 1 MB | TODO | TODO |
| 4 MB | TODO | TODO |
| 16 MB | TODO | TODO |
| 82 MB (256 tokens) | TODO | TODO |

### C. HPC Storage and I/O Systems

**Parallel file systems.** Lustre [10] and GPFS [11] provide high-bandwidth I/O for HPC through striping. However, they are optimized for large sequential I/O, not fine-grained KV cache access. Metadata operations are particularly costly.

**Burst buffers and tiered storage.** Burst buffers [12] absorb bursty I/O with node-local NVMe. Hermes [6] provides hierarchical buffered I/O. UnifyFS aggregates node-local storage. DAOS [7] offers object storage for NVM. *All assume node-local NVMe*, unavailable on Perlmutter compute nodes.

**HPC caching:** Some HPC systems use distributed memory caching (e.g., Memcached clusters), but these are not optimized for large tensor data and face deployment challenges on batch-scheduled systems.

**Gap:** No prior HPC storage work addresses LLM-specific access patterns, content-addressed deduplication, or the no-local-NVMe constraint.

### D. Positioning of CASCADE

CASCADE uniquely combines:

- **Content-addressed deduplication** from CAS systems
- **Multi-tier hierarchy** from LLM caching systems
- **HPC-native optimizations** (MPI, Lustre striping, /dev/shm)

Unlike datacenter systems, CASCADE treats network-accessible remote DRAM as a primary tier (enabled by Slingshot's high bandwidth) and uses aggregated file storage to overcome Lustre metadata overhead.

Unlike general HPC storage, CASCADE understands KV cache semantics (prefix sharing, semantic eviction) and exploits content-based deduplication.

## VI. Discussion and Limitations

**Deduplication assumptions.** CASCADE's content-addressed deduplication is most effective when workloads share system prompts across sessions. For purely unique prompts, deduplication provides no benefit, though other optimizations (aggregated storage, multi-tier caching) still apply. Production LLM deployments typically use 20-100 standard system prompts, making deduplication broadly applicable.

**Hash collision.** SHA-256 provides 128-bit collision resistance (using first 32 hex chars). The probability of collision is negligible for practical KV cache sizes ($< 2^{64}$ blocks). For higher assurance, full SHA-256 can be used at marginally higher storage cost.

TABLE VIII: Comparison of KV cache storage approaches.

| System | Content-Addressed | Dedup | Multi-Node | PFS Opt. | MPI |
|---|---|---|---|---|---|
| vLLM [3] | ✗ | ✗ | ✗ | ✗ | ✗ |
| LMCache [4] | ✗ | ✗ | ✗ | ✗ | ✗ |
| Mooncake [5] | ✗ | ✗ | ✓ | ✗ | ✗ |
| CacheBlend | △ | △ | ✗ | ✗ | ✗ |
| Hermes [6] | ✗ | ✗ | ✓ | ✓ | ✗ |
| DAOS [7] | ✗ | ✗ | ✓ | ✓ | ✗ |
| **CASCADE** | ✓ | ✓ | ✓ | ✓ | ✓ |

**Generalization to other HPC systems.** While evaluated on Perlmutter, CASCADE's design applies to other HPC systems lacking local NVMe (e.g., Frontier, Aurora, Alps). Systems with local burst buffers may add an additional NVMe tier.

**Model size and architecture.** CASCADE is designed for large models (LLaMA-70B class) where KV cache pressure is significant. For smaller models fitting in GPU memory, the overhead of content hashing may not be justified. GQA and other KV-efficient architectures reduce per-token cache size but increase the relative overhead of storage operations.

**Quantized KV cache.** Recent work explores INT4/INT8 KV cache quantization. CASCADE's content-addressed design works with quantized data, though hash computations would differ. Quantization is orthogonal and can be combined.

**Prefetch effectiveness.** CASCADE's remote tier prefetch assumes predictable access patterns. For fully random access, prefetch provides limited benefit. Request routing in multi-tenant deployments typically exhibits locality that prefetch can exploit.

**MPI constraints.** Using MPI for KV transfer requires MPI initialization, which may conflict with some deployment scenarios. Alternative implementations using direct Slingshot access or libfabric could address this limitation.

**Interference with other jobs.** Heavy Lustre usage may impact other jobs on shared systems. CASCADE's aggregated storage and tiered caching reduce Lustre pressure compared to per-file approaches, but rate limiting may be needed for very large deployments.

**Security and privacy.** KV cache contains processed input data, potentially including sensitive information. CASCADE does not currently encrypt cached data; adding encryption would impact performance. Access control relies on standard HPC mechanisms (POSIX permissions, quotas).

## VII. Conclusion

We presented CASCADE, a content-addressed tiered KV cache storage system designed for HPC-scale LLM inference. CASCADE introduces three key innovations:

1) **Content-addressed block identification**: By computing block IDs from KV tensor content via SHA-256, CASCADE automatically deduplicates shared prefixes across sessions without explicit coordination.
2) **Four-tier HPC storage hierarchy**: GPU HBM → Local DRAM → Remote DRAM via MPI → Lustre, with

semantic-aware eviction that protects frequently-shared prefix blocks.

3) **HPC-optimized data placement**: Aggregated file storage with Lustre striping achieves up to 29× better throughput than per-file approaches, overcoming the metadata overhead that cripples datacenter designs on HPC systems.

**[TODO: Finalize conclusions after experiments]**

Our evaluation on NERSC Perlmutter with LLaMA-70B demonstrates the potential of HPC-native KV cache design. Content-addressed deduplication combined with multi-tier caching and aggregated storage offers significant improvements over datacenter-optimized baselines naively deployed on HPC systems.

As LLM inference increasingly targets HPC platforms to leverage their massive GPU counts and high-bandwidth interconnects, storage-aware system design becomes critical. CASCADE demonstrates that HPC-native optimizations—content-addressed storage, MPI-based distribution, and parallel file system awareness—can dramatically improve KV cache efficiency for large-scale LLM serving.

**Reproducibility.** The CASCADE implementation and benchmark framework are available at: `[TODO: Add repository URL for camera-ready]`

## REFERENCES


[1] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] H. Touvron *et al.*, "LLaMA: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[3] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[4] LMCache Team, "LMCache: Efficient caching for large language model inference," *arXiv preprint*, 2024.

[5] R. Qin *et al.*, "Mooncake: A KVCache-centric disaggregated architecture for LLM serving," *arXiv preprint arXiv:2407.00079*, 2024.

[6] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: A heterogeneous-aware multi-tiered distributed I/O buffering system," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[7] J. Liang *et al.*, "DAOS: A scale-out high performance storage stack for storage class memory," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[8] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[9] Y. Zhong *et al.*, "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," *arXiv preprint arXiv:2401.09670*, 2024.

[10] P. Schwan, "Lustre: Building a file system for 1000-node clusters," *Proceedings of the Linux Symposium*, 2003.

[11] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," 2002.

[12] N. Liu *et al.*, "On the role of burst buffers in leadership-class storage systems," in *IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2012.