DSC-204A: Scalable Data Systems, Fall 2025

Lecturer: Prof. Junchen Jiang *Scribe: Rohan Surana, Yuxin Wan, Tianxiang Wang, Ziwen Liu*

0 - 15mins, Rohan Surana

This talk was focused on **KV Cache as the New AI Memory Abstraction**. The speaker, an associate professor at the University of Chicago, works on networked systems and machine learning systems and is involved in the open source project **LMCache** and the startup **TensorMesh**. The core framing is that memory, especially KV cache, should be treated as a first class citizen in large language model serving.

1 LMCache and the Serving Stack.

In industry, both closed source and open source models co-exist. For open weight models, practitioners typically rely on an open source infrastructure organized into three layers: (1) distributed LLM serving and orchestration, such as Kubernetes, (2) LLM inference engines, such as vLLM and SGLang, and (3) optimizations centered on KV cache. In this view, KV cache is model memory, not merely a caching add on, and LMCache is an example of this optimization layer.

2 Why KV Cache Matters for Long Context.

KV cache is important because the real motivation behind it is long context inference. LLMs are popular in part due to their ability to read and understand long documents or repositories, but supporting long contexts is also one of the biggest serving challenges. As context length increases, response delay grows (higher time to first token), GPU cost rises roughly linearly with usage and context, and model quality can degrade on very long or tail contexts. These pressures make efficient KV cache handling essential for practical long context workloads.

3 What KV Cache Is.

KV cache captures the model's internal understanding of how the input tokens relate to the tokens generated so far. During decoding, to produce the next token the model would otherwise need to repeatedly look back over all prior tokens, which is inefficient for long sequences. KV cache avoids this by storing the intermediate attention results from prefill and reusing them at each step, rather than rescanning and recomputing every time. Despite the name, it is not a traditional key value store. In practice, it is simply two large tensors, the **K** (keys) tensor and the **V** (values) tensor, that together represent the cached attention state.

4 Redundant Prefill and Reuse.

In long context workloads, different requests over the same document are often unaware of one another, so they repeatedly prefill identical or overlapping prefixes. KV cache reuse avoids this duplication by allowing a prefix prefilling done once by one engine to be reused by others.

5 Toward Hierarchical KV Storage.

This leads to an important open question: how can we maximize KV cache reuse. Ideally, systems would store more KV caches for longer periods, and each cache would persist until its last reuse. The practical constraint is GPU memory capacity. Today, KV caches live primarily in GPU memory, kept there as much as possible and for as long as possible, but that approach limits how many caches can be retained.

15 mins - 30 mins, Tianxiang Wang

6 The Problem: KV Cache Makes LLM Inference Too Slow

To answer a new query, an LLM must "prefill" (process) the whole context. But many queries share the same long context (books, docs, histories).

If every query re-prefills the same long context:

- huge latency
- huge compute waste
- poor GPU utilization

So the core insight is: The KV cache of a shared prefix should be reused instead of recomputed.

7 Why Reusing KV Cache Is Hard

1. **KV cache is extremely large.** Modern LLMs require hundreds of GB of KV cache per million tokens (e.g., Llama-70B ∼300GB). Since a single H100 has only 80GB, KV caches cannot fit in GPU memory, preventing reuse.
2. **Many users share similar contexts.** A GPU cannot simultaneously store KV for multiple long contexts (A, B, C, . . .). KV eviction happens immediately, eliminating any benefit of reuse.
3. **Paged-memory engines introduce I/O bottlenecks.** Engines such as vLLM and SGLang store KV in small pages and frequently move them across GPU, CPU, and storage tiers. This leads to slow I/O and degraded decode performance, especially for long-context workloads.
4. **LLM engines evolve too quickly.** New open-weight models frequently change KV sizes, memory allocators, and scheduling patterns. Traditional caching layers cannot keep up and would require constant rewrites.

5. **No unified API for KV management.** Operators lack basic primitives (lookup, eviction, movement, pinning, compression), making intelligent caching policies impossible.

8 LMCache: A Unified KV Caching Layer

LMCache introduces an engine-agnostic, hierarchical memory layer for KV storage:

- GPU memory (L1)
- CPU RAM (L2)
- Local SSD (L3)
- Remote object storage (L4)

It automatically decides where each KV cache should live, when it should move, and how to efficiently fetch or persist it.

9 Key Contributions

1. **High-performance KV movement.** Layer-wise pipelining, asynchronous requests, prefetching, and a KV-proxy keep GPU compute overlapped with I/O.
2. **Batched operations.** LMCache merges small KV pages, uses customized CUDA kernels, and batches KV store/load operations to reduce overhead.
3. **Minimal-copy data transfer.** Zero-copy paths, eviction-based offloading, and compressed formats significantly lower I/O latency.
4. **Engine-agnostic KVConnector API.** A stable API allows LMCache to remain compatible even as vLLM/SGLang evolve.
5. **Comprehensive KV management API.** Primitives such as `Lookup`, `Clear`, `Pin`, `Move`, `Compress`, and `CheckFinish` enable real-time policy control.

10 Performance Highlights

- **Compute vs. storage cost.** Prefill computation is 10–100× more expensive than storing KV on SSD or S3, making KV reuse economically desirable.
- **Low latency under load.** Across multiple models (e.g., Llama-3.1 8B, Qwen-2.5 72B), LMCache maintains stable TTFT/ITL even at high QPS, while naive vLLM and commercial systems exhibit dramatic latency spikes.

30mins - 45 mins, Yuxin Wan

11 Enabler of KV caches reuses

KV caches are large, expensive to recompute, and slow to move across accelerators. Real workloads show an average reuse factor of about 1.21×, making KV reuse an important optimization.

11.1 Why KV Cache Reuse Matters

- KV tensors can reach hundreds of MB per request.
- Remote storage (e.g., S3) adds tens–hundreds of ms latency.
- KV size per token is decreasing, while CPU RAM (256GB–1TB) makes storage practical.

Thus, caching is far cheaper than recomputation.

11.2 What Is Actually Cached

Systems do *not* cache answers. They cache intermediate attention states (KV cache), allowing:

- skipping past-token computation,
- fresh output generation,
- reuse of shared prefixes or shared segments.

KV caches may be stored in CPU memory, local SSD, or remote storage via LMCache.

11.3 Limits of Classical KV Caching

Classical systems require:

- exact prefix matching,
- same model,
- no modification of KV content.

This severely restricts reuse.

11.4 Routing Cannot Replace Caching

Routing repeated requests to the same GPU is unreliable due to:

- load balancing,

- limited GPU memory residency,
- batching and scheduling variability.

Caching remains necessary.

11.5 Why Prefix Caching Is Not Enough

Prefix-only reuse reaches at most ∼57% hit rate. Relaxing to substring reuse can reach 90–95%, because real workloads involve:

- text insertion or deletion,
- document stitching (RAG),
- multi-agent prompts.

Most non-prefix contexts still share large reused segments.

11.6 RAG Example and Missing Cross-Attention

In RAG, Doc1 and Doc2 are concatenated. If their KV caches were created separately:

- cross-attention between them never forms,
- position encodings differ,
- simple KV concatenation yields wrong answers.

Real attention maps confirm that cross-attention disappears when KV caches are naively merged.

11.7 Computation vs. Quality

- Full prefill (no caching): highest quality, highest cost.
- Reusing all KV (non-prefix): lowest cost, but quality drops due to missing cross-attention.

11.8 Audience Questions

Question 1: Why does concatenating KV caches cause wrong answers? Answer: Because there is no cross-attention between separately generated KV caches. When Doc1 and Doc2 are processed independently, the model never attends across them. Even after position encoding adjustment, the missing cross-attention leads to incorrect reasoning.

Question 2: Can we reuse all KV caches to avoid recomputation? Answer: No. Reusing all KV caches eliminates computation but loses accuracy in non-prefix contexts. Missing cross-attention must be recomputed; otherwise the model fails on reasoning tasks that depend on interactions across document segments.

45min-60min, Ziwen Liu

12 Full Prefill vs. Selective Prefill (CacheBlend)

12.1 Typical KV Prefill

- Long-context inference normally recomputes the KV cache for **every token at every layer**.
- When multiple retrieved chunks (Doc 1, Doc 2) are concatenated, only the prefix (Doc 1) is reusable.
- Doc 2 always triggers a full prefill over all 24 layers.

12.2 Selective Prefill via CacheBlend

- Layer 0: perform full prefill for the new chunk.
- Compare KV from prefill vs. stored KV for that chunk.
- Identify tokens with the **largest deviation**.
- Recompute only these tokens on Layer 1.
- Repeat selectively for higher layers.
- On average, only $\sim 15\%$ of tokens per layer require recomputation.

12.3 Outcome

- Substantial compute savings while maintaining accuracy.
- Recomputes only "unstable" tokens whose KV meaningfully changes.

13 Faster Response (TTFT) and Accuracy Tradeoff

13.1 Methods Compared

- **Full KV recompute**: slowest, highest quality.
- **KV reuse only**: fastest, lowest quality.
- **Prefix-only caching**: minimal improvement.
- **CacheBlend**: selectively recompute only important tokens; best speed–quality balance.

13.2 Key Results

- CacheBlend achieves **3× faster** time-to-first-token compared to full prefill.
- Accuracy significantly higher than pure KV reuse.
- TTFT remains low even at rising request load.

14 Higher Throughput

14.1 Experimental Setup

- Dataset: 2WikiMQA (6 chunks, each 512 tokens).
- 1.5K sampled queries.
- Model: Llama-70B; Hardware: 2× A40 GPUs.
- KV cache stored on disk initially.

14.2 Findings

- Full recompute and prefix-only caching experience TTFT spikes early under load.
- CacheBlend's TTFT grows slowly, enabling **3× higher sustainable throughput**.

15 Surprising Result: Selective Recompute Can Increase Accuracy

15.1 Observed Phenomenon

- Across multiple models (20B, 8B, Qwen-30B) and datasets:
  - Accuracy peaks at a recompute ratio of 10–40%.
  - 100% recompute returns to full-prefill accuracy.
  - 0% recompute gives lowest quality.

15.2 Hypothesis

- RAG-retrieved chunks are often **not naturally coherent**.
- Full cross-attention between unrelated chunks may introduce noise.
- Selective recompute effectively **blocks harmful cross-attention**, retaining only beneficial interactions.

15.3 Status

- Reproducible empirically; theoretical explanation remains open.
- Considered an interesting open research direction.

16 Why KV Cache (AI-Native Data) Matters

16.1 KV Cache as Model-Native Representation

- All text, chats, and videos are converted internally into KV tensors.
- This is **AI-native data**: not human-readable, not token-level, but directly consumable by models.

16.2 Benefits

- **5–10× faster** inference by reusing KV.
- Lower GPU cost.
- Enables smarter retrieval, compression, and context blending.

16.3 Future Importance

- KV cache evolving from an "optimization trick" to a **core data layer** of the AI stack.

17 Conclusion

- KV cache is the **first major form of AI-native data**.
- CacheBlend demonstrates:
  - major compute reduction,
  - faster inference,
  - occasionally **higher-than-baseline accuracy**.
- LMCache is the widely used open-source KV caching library.
- Tensormesh aims to make KV caching the **central layer** of the AI stack.