KV Cache Optimization

Memory Management Strategies for Efficient LLM Serving

February 2026 • Technical Report

Table of Contents

1. Introduction

The key-value (KV) cache is one of the most critical data structures in autoregressive large language model inference. During text generation, the model computes key and value projections for every attention head at every layer for each token it processes. Rather than recomputing these projections at every decoding step, the system caches them for reuse. While this eliminates redundant computation, the KV cache quickly becomes the dominant consumer of GPU memory, often exceeding the memory footprint of the model weights themselves for long sequences. Optimizing KV cache memory usage directly determines how many concurrent sequences a serving system can handle, the maximum context length it can support, and ultimately the cost per generated token.

This report examines the primary strategies for managing and optimizing KV cache memory in production LLM inference, covering architectural innovations, compression techniques, and system-level memory management approaches.

2. The KV Cache Memory Problem

2.1 Memory Footprint Analysis

The memory required for a KV cache scales with several factors: the number of layers, the number of attention heads, the head dimension, the sequence length, and the batch size. For a typical 70-billion-parameter model with 80 layers, 64 heads, and a head dimension of 128 using FP16, a single sequence of 4,096 tokens requires approximately 5 GB of KV cache memory. At a batch size of 32, this grows to 160 GB, far exceeding the memory of any single GPU. As context windows extend to 128K or even 1M tokens, the problem becomes even more acute.

2.2 Internal Fragmentation

Traditional KV cache implementations pre-allocate contiguous memory blocks for each sequence based on the maximum possible sequence length. Since most sequences terminate well before the maximum length, this leads to severe internal fragmentation. Studies have shown that naive pre-allocation wastes 60 to 80 percent of allocated KV cache memory on average, drastically reducing the effective batch size the system can support.

3. PagedAttention and Virtual Memory

PagedAttention, introduced in the vLLM system, applies operating system virtual memory concepts to KV cache management. Instead of allocating contiguous memory for each sequence, the KV cache is divided into fixed-size pages (typically storing 16 tokens each). A page table maps each sequence's logical token positions to physical

memory pages, which need not be contiguous. Pages are allocated on demand as new tokens are generated and freed when sequences complete.

This approach virtually eliminates internal fragmentation and enables several powerful capabilities. Memory sharing becomes trivial: when multiple sequences share a common prefix (such as a system prompt), they can reference the same physical pages through copy-on-write semantics, reducing memory usage proportionally to the number of shared tokens. PagedAttention has been widely adopted and is now implemented in most major inference frameworks including vLLM, TensorRT-LLM, and SGLang.

4. KV Cache Compression Techniques

4.1 Quantization

KV cache quantization reduces the precision of cached key and value tensors from FP16 or BF16 to lower-bit formats such as INT8, INT4, or FP8. Since the KV cache is read far more frequently than it is written, the amortized cost of quantization and dequantization is low. INT8 KV cache quantization typically has negligible impact on output quality while halving memory usage. More aggressive INT4 quantization can reduce memory by 75 percent but may introduce noticeable quality degradation for certain tasks, particularly those requiring precise factual recall or mathematical reasoning.

4.2 Token Eviction and Pruning

Not all tokens in the KV cache contribute equally to generation quality. Token eviction strategies identify and remove less important tokens from the cache to free memory. Methods such as H2O (Heavy Hitter Oracle) observe that a small subset of tokens consistently receive high attention weights and evict tokens with low cumulative attention scores. Scissorhands and similar approaches use attention pattern analysis to prune the cache while preserving the tokens most likely to be attended to in future steps. These methods can reduce KV cache size by 50 to 80 percent with minimal quality impact on many tasks.

4.3 Grouped Query Attention and Multi-Query Attention

Architectural modifications at the model design stage can dramatically reduce KV cache requirements. **Multi-Query Attention (MQA)** shares a single key-value head across all query heads, reducing KV cache memory by a factor equal to the number of query heads (often 32x to 128x). **Grouped Query Attention (GQA)** is a compromise that groups query heads into clusters sharing key-value heads, achieving substantial memory savings (typically 4x to 8x) with less quality degradation than MQA. Most modern models, including Llama 3, Mistral, and Gemma 2, use GQA as the default attention mechanism.

4.4 Cross-Layer KV Cache Sharing

Recent research has explored sharing KV caches across adjacent layers, exploiting the observation that key and value representations are often highly similar between neighboring layers. Methods like YOCO (You Only Cache Once) and CLA (Cross-Layer Attention) assign KV caches to only a subset of layers and have other layers reference them, reducing total KV cache memory by 2x or more with modest quality trade-offs.

5. Prefix Caching and Radix Trees

Many LLM workloads involve repeated prefixes, such as system prompts, few-shot examples, or shared document contexts. Prefix caching stores the KV cache for common prefixes and reuses it across requests, eliminating redundant prefill computation and memory allocation. Advanced implementations use radix tree data structures to efficiently match incoming requests against cached prefixes, supporting partial prefix matches and hierarchical sharing. SGLang's RadixAttention system demonstrated that prefix caching can reduce time-to-first-token by 3 to 5x for workloads with high prefix overlap.

6. Offloading and Hierarchical Storage

When GPU memory is insufficient to hold the full KV cache for all active sequences, offloading strategies spill KV cache pages to CPU memory or even to NVMe storage, fetching them back on demand. This extends the effective KV cache capacity well beyond GPU memory limits, enabling support for very long contexts or very large batches. The key challenge is managing the latency of data transfers between memory tiers. Prefetching heuristics that predict which pages will be needed in upcoming decoding steps can hide much of this latency by overlapping data transfer with computation.

7. Comparison of Optimization Strategies

The following table summarizes the key KV cache optimization approaches and their characteristics.

Strategy	Memory Reduction	Quality Impact	Implementation
PagedAttention	~60–80% waste eliminated	None	System-level
INT8 Quantization	2x	Negligible	Runtime
INT4 Quantization	4x	Minor–Moderate	Runtime

Token Eviction (H2O)	2–5x	Minor (task-dependent)	Runtime
GQA (at training)	4–8x	Minimal	Architecture
MQA (at training)	32–128x	Minor–Moderate	Architecture
Cross-Layer Sharing	2x	Minor	Architecture
Prefix Caching	Variable (prefix-dependent)	None	System-level
CPU/NVMe Offloading	Extends beyond GPU capacity	None (latency cost)	System-level

8. Practical Considerations

In production systems, KV cache optimization strategies are rarely used in isolation. A typical high-performance serving stack might combine GQA at the model architecture level, PagedAttention for memory management, INT8 quantization for cache compression, prefix caching for common prompts, and offloading as a fallback for overflow. The interactions between these techniques require careful engineering: for example, quantized caches change memory access patterns that may affect the optimal page size for PagedAttention, and token eviction policies must be aware of prefix caching boundaries.

Monitoring KV cache utilization is essential for capacity planning. Metrics such as cache hit rate (for prefix caching), average occupancy per page, eviction frequency, and offload bandwidth utilization help operators tune system parameters and right-size their GPU fleet for the workload.

9. Future Directions

Several active research directions promise further improvements in KV cache efficiency. Learned compression methods that train lightweight neural networks to compress and decompress KV cache entries are showing early promise. Hardware-aware approaches that co-design cache formats with GPU memory hierarchy (exploiting L2 cache, shared memory, and register files) can improve access latency. The emergence of linear attention variants and state-space models (such as Mamba) that replace the KV cache entirely with fixed-size recurrent states represents a more radical architectural direction that could fundamentally change the memory scaling story for long-context inference.

10. Conclusion

KV cache optimization is a multi-layered challenge that spans model architecture, runtime systems, and hardware utilization. The combination of PagedAttention for fragmentation elimination, quantization and eviction for compression, architectural innovations like GQA for structural reduction, and prefix caching for workload-specific reuse has collectively enabled LLM serving systems to handle much larger batch sizes and context lengths than would otherwise be possible. As models and context windows continue to grow, continued innovation in KV cache management will remain essential for practical and cost-effective LLM deployment.