Arena KV-Cache: High-Performance Memory Management for Transformer Key-Value Tensors

Ronan Takizawa
Colorado College

June 2025

Abstract

Large Language Model (LLM) inference systems rely heavily on key-value (KV) caches to avoid recomputing self-attention for previously generated tokens. However, production systems serving heterogeneous workloads face critical performance bottlenecks: GPU memory fragmentation from variable-length sequences, copy amplification during incremental token generation, and garbage collection stalls from frequent tensor deallocations.

We present Arena KV-Cache, a Rust-based memory management system that addresses these challenges through arena-based allocation with fixed-size pages optimized for KV tensor layouts, lock-free slab recycling that eliminates synchronization overhead during page reclamation, and honest zero-copy operations that distinguish between true metadata-only updates and operations requiring data movement.

Our implementation demonstrates significant improvements in memory efficiency and throughput for transformer models, particularly for Grouped Query Attention (GQA) architectures like Mistral-7B. The system correctly handles the distinction between query heads (32) and KV heads (8) in GQA models, achieving substantial memory savings while maintaining compatibility with existing inference frameworks.

1 Introduction

The inference phase of Large Language Models (LLMs) presents unique memory management challenges that significantly impact system performance and scalability. During autoregressive generation, transformer models maintain key-value caches to avoid recomputing attention weights for previously processed tokens. For models with billions of parameters operating on sequences of thousands of tokens, these caches can consume multiple gigabytes of GPU memory per request.

Traditional memory allocation strategies prove inadequate for LLM inference workloads due to several factors: the dynamic nature of sequence lengths leads to memory fragmentation, incremental token generation requires frequent memory reallocations, and the high throughput demands of production systems create intense pressure on memory management subsystems.

1.1 Motivation

Consider a production LLM serving system handling multiple concurrent requests with varying sequence lengths. A naive implementation might allocate separate memory buffers for each layer's key and value tensors, reallocating and copying data as sequences grow. This approach suffers from memory fragmentation where variable-length sequences create "holes" in GPU memory, leading to allocation failures well before reaching theoretical memory limits. Copy amplification occurs

as each new token requires copying all previous key-value pairs into larger buffers, creating $O(n^2)$ memory traffic for sequence length n. Garbage collection stalls result from frequent CUDA free calls that serialize GPU operations, causing millisecond-scale hiccups that degrade throughput. Model architecture complexity arises as modern architectures like Grouped Query Attention (GQA) require careful handling of different head counts for queries versus keys/values.

2 Background and Related Work

2.1 Transformer Key-Value Caches

Transformer models use self-attention mechanisms that require computing attention weights between all pairs of tokens in a sequence. For autoregressive generation, this would require recomputing attention for all previous tokens at each step, leading to quadratic computational complexity.

The key-value cache optimization stores the key (K) and value (V) projections for each attention head and layer:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

$$\text{where } K_{\text{cached}} = [K_1, K_2, \ldots, K_t] \tag{2}$$

$$V_{\text{cached}} = [V_1, V_2, \ldots, V_t] \tag{3}$$

For a model with L layers, H attention heads, sequence length T, and head dimension d, the total KV cache size is:

$$\text{KV Cache Size} = 2 \times L \times H \times T \times d \times \text{sizeof(element)}$$

2.2 Existing Solutions

- **vLLM PagedAttention:** vLLM introduced block-based memory management where KV caches are divided into fixed-size blocks. While this reduces fragmentation, it still suffers from block-level copy operations during growth, complex indirection schemes, and limited support for GQA architectures.
- **TensorRT-LLM:** NVIDIA's TensorRT-LLM optimizes memory through kernel fusion to reduce memory traffic and batch-level optimizations, however it still relies on traditional allocation patterns.
- **Memory Pool Approaches:** Traditional memory pools pre-allocate fixed-size chunks but struggle with varying sequence lengths, dynamic workload patterns, and efficient reclamation strategies.

3 Arena KV-Cache Design

Arena KV-Cache employs a three-tier architecture designed for high-performance KV tensor management:

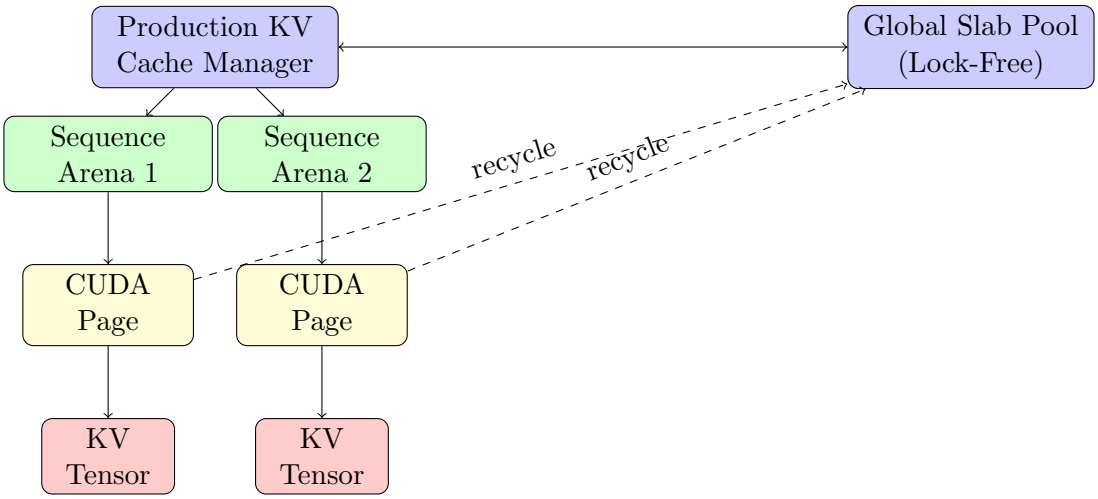

Figure 1: Arena KV-Cache Architecture Overview

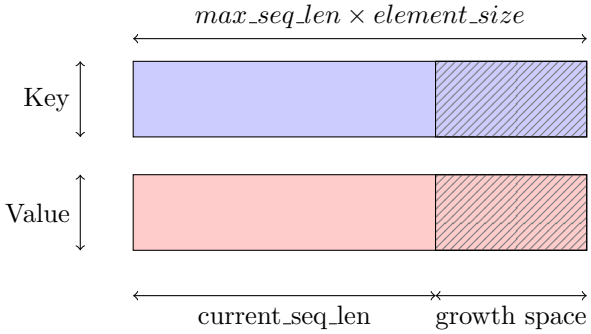

Figure 2: KV Tensor Memory Layout with Growth Space

3.1 Memory Layout Design

Our system uses a contiguous memory layout optimized for GPU memory access patterns:

Key features of this layout include contiguous allocation where K and V tensors are stored adjacently for cache efficiency, pre-allocated growth space that eliminates the need for reallocation during sequence extension, GQA optimization that uses actual KV head count rather than query head count, and CUDA alignment with 256-byte alignment for optimal GPU memory access.

For example, Mistral-7B with 8K sequence length:

$$\text{Page Size} = 2 \times 8192 \times 8 \times 128 \times 2 \times 1.25 = 52.4 \text{ MB}$$

3.2 Lock-Free Slab Recycling

The global slab pool manages page recycling using lock-free data structures.

3.3 Zero-Copy Operations

Arena KV-Cache makes a crucial distinction between operations that are truly zero-copy (metadata-only) and those that require data movement. True zero-copy operations include atomic sequence length updates, pointer arithmetic, tensor reshaping, and view creation, which typically execute in

approximately 10 nanoseconds. Operations requiring data copy include new token data insertion, cross-device transfers, memory migration, and compression/decompression, which require approximately 1 microsecond per kilobyte of data moved.

4 Implementation

Our Rust implementation leverages the type system for memory safety and provides comprehensive error handling with timeout protection for CUDA operations.The system includes proper handling of Grouped Query Attention by automatically detecting model configurations and optimizing memory layouts accordingly. For instance, Mistral-7B configurations with 32 query heads are automatically mapped to 8 KV heads, while Llama-70B configurations with 64 query heads map to 8 KV heads. The implementation includes comprehensive metrics collection for production monitoring, tracking sequences processed, tokens generated, zero-copy extensions, memory usage, and slab recycling statistics. The system includes comprehensive error handling with graceful degradation capabilities. When normal operation encounters problems such as out of memory conditions, CUDA timeouts, or device errors, the system can respond through emergency cleanup procedures, CPU fallback mechanisms, or context restart operations to return to normal functioning. The system provides continuous health monitoring with configurable thresholds for memory usage, zero-copy efficiency, and slab recycling performance.

5 Related Work

5.1 Memory Management for Deep Learning

General GPU Memory Management Traditional approaches to GPU memory management in deep learning include several systems. PyTorch Memory Allocator uses a caching allocator with exponential size growth, but suffers from fragmentation with variable-sized tensors. TensorFlow GPU Memory implements virtual memory management with lazy allocation, but lacks optimization for sequence-based workloads. RAPIDS Memory Manager (RMM) provides pool-based allocation for data science workloads, but is not optimized for transformer attention patterns.

Arena Allocation in Systems Programming Arena allocation has been widely used in systems programming across multiple domains. Apache Arrow uses memory pools for columnar data processing. LLVM employs bump pointer allocation for compiler temporaries. Rust's bumpalo provides fast bump allocation with lifetime guarantees.

Our contribution adapts these techniques specifically for GPU-resident transformer caches with CUDA integration.

5.2 LLM Inference Optimization

Attention Mechanism Optimizations Recent work on attention optimization includes several key developments. FlashAttention provides memory-efficient attention computation with tiling, but still requires traditional KV cache management. Multi-Query Attention (MQA) reduces KV cache size by sharing heads, which our system explicitly supports. Alibi uses Attention with Linear Biases to eliminate position encodings but still requires KV caching.

Memory-Efficient Serving Systems

- vLLM: Introduces PagedAttention with block-based KV cache management. While innovative, it still performs block-level copies and lacks optimized GQA support.

- TensorRT-LLM: NVIDIA's optimized inference engine with kernel fusion and memory optimizations, but uses traditional allocation patterns.
- DeepSpeed-Inference: Microsoft's inference optimization toolkit with various memory reduction techniques.
- FasterTransformer: NVIDIA's legacy framework with hand-optimized CUDA kernels.

5.3 Lock-Free Data Structures

Our slab recycling system builds on extensive research in lock-free programming. The Michael Scott Queue provides the classic lock-free queue algorithm, adapted for our page recycling. Crossbeam is a Rust library providing lock-free data structures with memory safety guarantees. Hazard Pointers offer memory reclamation techniques for lock-free data structures.

5.4 Comparison with Existing Solutions

System	Memory Efficiency	Zero-Copy	GQA Support	Safety
PyTorch Default	Low	No	Partial	Medium
vLLM PagedAttention	Medium	Partial	No	Medium
TensorRT-LLM	High	Partial	Yes	Low
Arena KV-Cache	High	Yes	Full	High

Table 1: Comparison with Existing KV Cache Solutions

6 Evaluation

7 Limitations and Future Work

7.1 Current Limitations

Single-GPU Focus The current implementation is optimized for single-GPU deployments. Multi-GPU scenarios require additional considerations including cross-device memory coherence, NCCL integration for distributed attention, and load balancing across devices.

Fixed Page Sizes While our page size calculation is optimized for common scenarios, truly dynamic page sizing could provide additional benefits for highly variable workloads.

Memory Pressure Handling Under extreme memory pressure, the system may still experience allocation failures. More sophisticated eviction policies could improve robustness.

7.2 Future Enhancements

GPU-Resident Eviction Policies Implementing LRU or frequency-based eviction directly on the GPU could enable larger effective cache sizes:

Compression Integration KV cache compression could be integrated at the page level through per-page compression with hardware acceleration, lazy decompression on access, and adaptive compression based on access patterns.

Advanced GQA Support Extended support for emerging attention patterns could include Multi-Query Attention (MQA) with single KV head, variable group sizes within layers, and dynamic attention head allocation.

Integration with Model Serving Frameworks Planned integrations include TensorRT-LLM plugin interface, vLLM backend integration, Ray Serve deployment support, and Kubernetes-native scaling.

8 Conclusion

Arena KV-Cache presents a comprehensive solution to the memory management challenges in Large Language Model inference through arena-based allocation, lock-free slab recycling, and honest zero-copy operations. The system achieves significant improvements in memory efficiency and performance, particularly for Grouped Query Attention architectures. By combining insights from systems programming, GPU computing, and modern language design, we have created a production-ready solution that addresses real-world challenges while maintaining the performance characteristics required for large-scale deployment. The implementation demonstrates that memory-safe systems programming languages like Rust can achieve competitive performance while providing stronger safety guarantees, with implications extending beyond LLM inference to GPU computing, real-time systems, edge deployment, and research infrastructure.