DeltaKV: Residual-Based KV Cache Compression via Long-Range Similarity

Jitai Hao [1] **Qiang Huang** [1 †] **Yaowei Wang** [1] **Min Zhang** [1] **Jun Yu** [1 †]

Abstract

The deployment of efficient long-context LLMs in applications like autonomous agents, long-chain reasoning, and creative writing is fundamentally bottlenecked by the linear growth of KV cache memory. Existing compression and eviction methods often struggle to balance accuracy, compression ratio, and hardware efficiency. We propose **DeltaKV**, a residual-based KV cache compression framework motivated by two empirical findings: *long-range inter-token similarity* and *highly shared latent components* in KV representations. Instead of discarding tokens, DeltaKV encodes semantic residuals relative to retrieved historical references, preserving fidelity while substantially reducing storage. To translate compression gains into real system speedups, we further introduce **Sparse-vLLM**, a high-performance inference engine with decoupled memory management and kernels optimized for sparse and irregular KV layouts. Experiments show that DeltaKV reduces KV cache memory to **29%** of the original while maintaining near-lossless accuracy on LongBench, SCBench, and AIME. When integrated with Sparse-vLLM, it achieves up to **2**× throughput improvement over vLLM in long-context scenarios, demonstrating a practical path toward scalable long-context LLM deployment.

1. Introduction

Modern LLM applications–including autonomous agents, legal and financial document analysis, code understanding, scientific discovery, and multimodal understanding and generation–routinely operate over extremely long contexts (Yao et al., 2022; Jimenez et al., 2023; Alayrac et al., 2022; Guha et al., 2023; Hao et al., 2025b). However, the quadratic $\mathcal{O}(n^2)$ attention cost makes long-context inference prohibitively expensive: a 128k-token prompt can incur a Time to First Token (TTFT) of over 20 seconds. Meanwhile, KV cache memory consumption grows linearly with sequence length, quickly exceeding available GPU memory. Concretely, for Llama-3.1-8B-Instruct (Grattafiori et al., 2024), a context length of 128k with batch size 8 requires more than **130 GB** of KV cache storage, far beyond the capacity of a single accelerator.

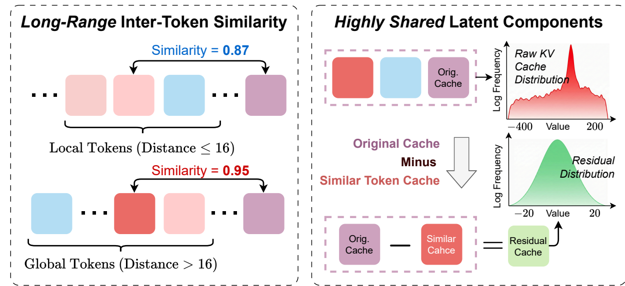


Figure 1. **Illustration of two empirical observations in KV caches:** *long-range* inter-token similarity beyond local context and *highly shared* latent components across KV representations.

To address this bottleneck, a major line of work focuses on reducing inference cost through **token eviction or selection** (Zhang et al., 2023; Li et al., 2024a; Oren et al., 2024; Liu et al., 2023). These methods aim to identify *important tokens* using attention scores or heuristic metrics. Existing approaches can be categorized into static eviction methods, such as SnapKV (Li et al., 2024a) and H2O (Zhang et al., 2023), and dynamic selection methods, including OmniKV (Hao et al., 2025c), Quest (Tang et al., 2024), and PQCache (Zhang et al., 2025), which adaptively select tokens conditioned on the current decoding step.

Nevertheless, in realistic multi-stage settings where model focus shifts (e.g., multi-turn dialogue (Li et al., 2024b) and complex reasoning (MAA)), static eviction methods like SnapKV often discard tokens that later become critical, causing large performance drops. Dynamic sparsity methods (e.g., OmniKV, Quest) mitigate this by retaining the full KV cache and applying sparse attention, but they do not fundamentally reduce GPU memory without offloading, which introduces latency and PCIe overhead.

Beyond token selection, **KV cache compression** offers a more principled path toward memory reduction. Nonetheless, existing compression approaches face substantial obstacles when deployed in real systems, particularly in bal-

[1]Harbin Institute of Technology (Shenzhen), China. Correspondence to: Qiang Huang <huangqiang@hit.edu.cn>, Jun Yu <yujun@hit.edu.cn>.

Preprint. February 7, 2026.

ancing compression effectiveness, hardware efficiency, and framework compatibility:

- **Local Similarity Bias:** Methods such as CacheGen (Liu et al., 2024c) and Chelsea (Hu et al., 2025) exploit similarity among nearby tokens, but implicitly assume locality. As we show in Figure 2b, this assumption overlooks substantial *global* similarity across distant tokens.
- **GPU-Unfriendly Pipeline:** Approaches like PQCache (Zhang et al., 2025) and Lexico (Kim et al., 2024) rely on multi-stage compression pipelines or complex codebooks, which introduce irregular memory access patterns, GPU underutilization, and reduced throughput.
- **Poor Integration with Inference Frameworks:** Methods that evict tokens unevenly across layers (e.g., SnapKV) or impose heterogeneous per-layer/per-head budgets (e.g., PyramidKV (Cai et al., 2024) and AdaKV (Feng et al., 2025)) are difficult to integrate into production-grade inference engines such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024). As a result, many promising sparsity and compression techniques remain impractical in real deployments.

Motivated by these limitations, we conduct an empirical analysis of KV cache representations and uncover two key observations, illustrated in Figure 1: **(1)** ***Long-Range*** **Inter-Token Similarity:** Contrary to locality-driven assumptions, semantically similar tokens are often distributed *globally* across the context rather than confined to nearby positions. **(2)** ***Highly Shared*** **Latent Components:** KV caches exhibit strong anisotropy, with a small number of high-norm latent directions capturing common linguistic and structural patterns shared across many tokens. These observations indicate that much of the KV cache is redundant shared structure, while the remaining token-specific information is comparatively low-magnitude and easier to compress.

Based on this insight, we propose **DeltaKV**, a *simple, residual-based, and GPU-friendly* KV cache compression framework that reduces KV cache memory to **29%** of its original size while maintaining near-lossless performance. DeltaKV partitions the KV cache into a small set of uncompressed reference tokens and a larger set of compressed tokens. For each compressed token, DeltaKV retrieves a few globally similar references, subtracts their shared components, and encodes only the resulting *residual* using a lightweight MLP or linear projection.

During inference, DeltaKV naturally complements sparse attention methods such as OmniKV: only a small subset of important tokens are reconstructed on demand, avoiding unnecessary decompression and memory I/O. Despite introducing compressor and decompressor modules, DeltaKV adds negligible parameter overhead (typically under 5% and can be trained efficiently in approximately 8 GPU hours.

Contributions Our contributions are threefold:

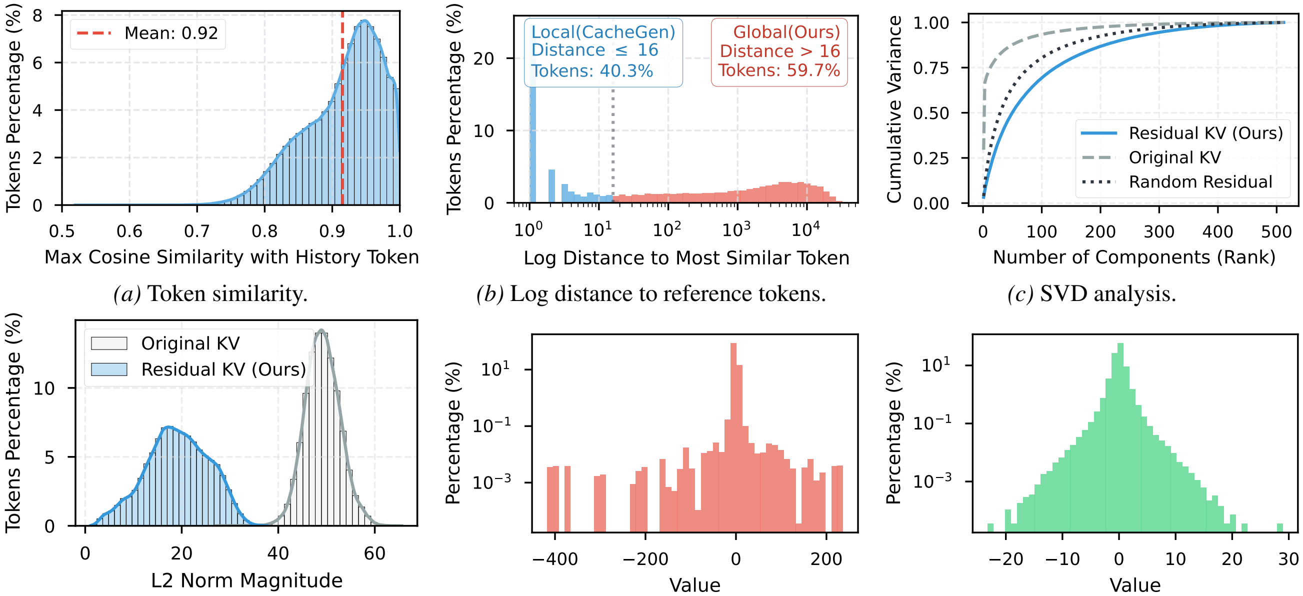
- **Global Redundancy in KV Caches:** We show that KV cache similarity is fundamentally *global*: over **60%** of similar tokens are separated by more than **16 positions**. We further identify dominant high-norm shared components whose removal yields low-magnitude residuals, exposing global redundancy (Figure 2).
- **DeltaKV: Residual-Based, GPU-Friendly KV Cache Compression:** We introduce DeltaKV, a residual-based KV cache compression framework that encodes only token-specific deviations from a small set of globally retrieved references. DeltaKV uses lightweight linear or MLP projections, is fully GPU-friendly, and reduces KV cache memory to **29%** with near-lossless accuracy.
- **Sparse-vLLM for Practical Deployment:** We present Sparse-vLLM, an inference framework for sparse and compressed KV caches with irregular memory layouts. It supports DeltaKV and related methods and achieves up to $\mathbf{2}\times$ higher throughput than vLLM, enabling practical deployment of KV cache compression.

2. Related Work

As foundation models continue to scale, substantial effort has been devoted to reducing inference cost, enabling deployment under memory and latency constraints, and improving throughput (Li et al., 2024a; Oren et al., 2024; Liu et al., 2024b; Lin et al., 2025; Qi et al., 2025; Liu et al., 2024a; Guo et al., 2025b; Hao et al., 2025a). Among these directions, the most relevant to our work are token selection, KV cache compression and quantization, token clustering, and inference frameworks for long-context LLMs.

Token Selection Token selection methods aim to reduce attention cost by retaining only a subset of tokens. They fall into *static eviction* (Zhang et al., 2023; Oren et al., 2024; Li et al., 2024a; Xiao et al., 2023b), which permanently remove tokens based on heuristics or early attention signals, and *dynamic selection* (Hao et al., 2025c; Tang et al., 2024; Zhang et al., 2025; Xiao et al., 2024; Liu et al., 2025a; Sun et al., 2024), which adaptively selects tokens during decoding. While dynamic methods achieve near-lossless accuracy in complex scenarios, they retain the full KV cache and rely on offloading, making performance bottlenecked by PCIe bandwidth. In contrast, DeltaKV performs GPU-resident, parallelizable compression that directly reduces KV cache footprint without offloading.

KV Cache Compression KV cache compression methods exploit low-rank or subspace structure to reduce memory footprint (Saxena et al., 2024; Zhang et al., 2024; Chang et al., 2025; Liu et al., 2024a), but typically require reconstructing the *entire* KV cache during inference, incurring substantial computational overhead. DeltaKV instead compresses token-wise residuals relative to retrieved references



(a) Token similarity. *(b)* Log distance to reference tokens. *(c)* SVD analysis.

(d) L2 Norm gap. *(e)* Original KV distribution. *(f)* Residual KV distribution.

Figure 2. **Experimental analysis of KV redundancy and residualization.**

and, when combined with sparse attention, reconstructs only $\leq \mathbf{10}\%$ tokens, achieving significantly higher efficiency.

KV Cache Quantization Quantization reduces memory and I/O cost by lowering numerical precision of KV caches, improving throughput in memory-bound decoding (Liu et al., 2024d; Xiao et al., 2023a; Hooper et al., 2024; He et al., 2024). Yet, it neither reduces attention computation nor integrates well with sparsification due to channel-wise operations (Liu et al., 2024d). In contrast, DeltaKV produces low-magnitude residuals that naturally support token-level quantization for further memory reduction.

Token Clustering Token clustering methods compress KV caches by reusing shared representations, but existing approaches either rely on CPU-hosted structures and incur PCIe bottlenecks (e.g., ClusterKV (Liu et al., 2025b)) or focus on local similarity, limiting reconstruction quality (e.g., Chelsea (Hu et al., 2025)). In contrast, DeltaKV performs global, GPU-resident residual compression using long-range similarity, avoiding external memory traffic.

Inference Frameworks Popular inference frameworks such as vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2024), and LightLLM (Gong et al., 2025) are optimized for full attention and page-based KV management, making sparsification and compression difficult to integrate. Recent adaptations either sacrifice batch scalability (e.g., Sparse Frontier (Nawrot et al., 2025)) or rely on less optimized backends (e.g., KVPress (Devoto et al., 2025)). In contrast, Sparse-vLLM abandons page-level assumptions and natively supports sparse, irregular KV layouts, enabling efficient deployment of DeltaKV in practice.

3. Observations

Background and Notation DeltaKV targets standard Transformer architectures such as Qwen (Yang et al., 2025) and Llama (Grattafiori et al., 2024). Each layer consists of self-attention and a feed-forward network (FFN). Let $\boldsymbol{W}_q \in \mathbb{R}^{d_q \times d}$, $\boldsymbol{W}_\mathrm{k} \in \mathbb{R}^{d_\mathrm{k} \times d}$, $\boldsymbol{W}_\mathrm{v} \in \mathbb{R}^{d_\mathrm{v} \times d}$, and $\boldsymbol{W}_\mathrm{o} \in \mathbb{R}^{d \times d}$ be the query, key, value, and output projections, where d is the hidden size, and $d_\mathrm{q}, d_\mathrm{k}, d_\mathrm{v}$ denote the total query/key/value dimensions across heads. Generally, $d_\mathrm{k} = d_\mathrm{v}$.

During inference, the KV cache stores key-value states (i.e., $\boldsymbol{K}, \boldsymbol{V}$) from previous tokens to avoid recomputation. Given token embeddings $\boldsymbol{H} \in \mathbb{R}^{B \times L \times d}$, the KV cache is computed as $\boldsymbol{K} = \boldsymbol{H}\boldsymbol{W}_\mathrm{k}$ and $\boldsymbol{V} = \boldsymbol{H}\boldsymbol{W}_\mathrm{v}$, both with shape $[B, L, d_\mathrm{k}]$ or equivalently $[B, L, N, D]$, where B denotes the batch size, N the number of key/value heads, L the sequence length, and D the key/value head dimension.

Inference consists of a *prefill* phase that constructs the KV cache and an autoregressive *decode* phase. For long contexts, *chunk prefill* reduces activation memory without altering the computation. To analyze KV cache redundancy, we define the **Residual KV** as:

$$\boldsymbol{KV}_\Delta = \boldsymbol{KV} - \overline{\boldsymbol{KV}}_R,$$

where $\boldsymbol{KV} = \mathrm{Concat}(\boldsymbol{K}, \boldsymbol{V})$ and $\overline{\boldsymbol{KV}}_R$ is the mean of the k nearest historical *reference* tokens. All analyses are conducted on Qwen2.5-7B-Instruct-1M (Yang et al., 2025). The results are presented in Figure 2.

Observation 1: *Long-Range* Inter-Token Similarity Token redundancy in natural language is not confined to local neighborhoods. While prior work like CacheGen (Liu et al.,

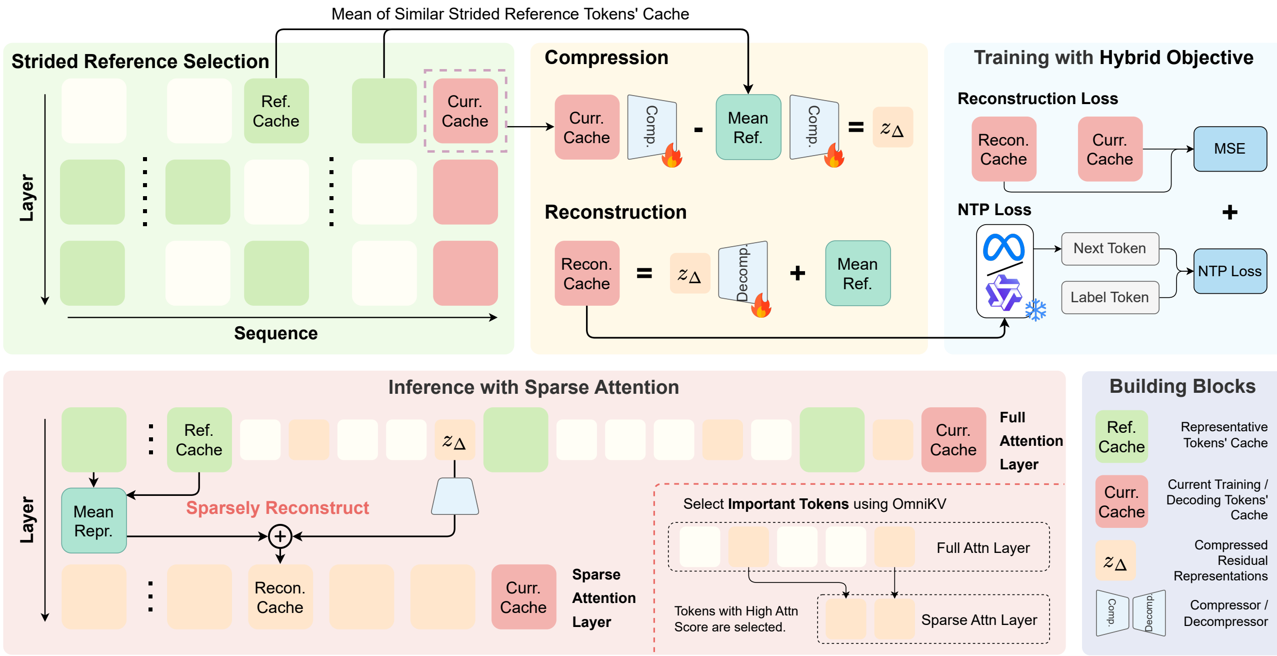


Figure 3. **Overview of DeltaKV.** DeltaKV compresses KV caches by encoding token-wise residuals relative to globally retrieved reference tokens, trained with a hybrid MSE+NTP objective (top). At inference, DeltaKV integrates with sparse attention (e.g., OmniKV), storing compressed residuals and reconstructing only selected tokens on demand, reducing memory usage while preserving accuracy (bottom).

2024c) emphasizes locality (similarity to immediate neighbors), we find that a token's closest semantic match often lies far back in the context. As shown in Figure 2a, KV representations frequently exhibit cosine similarity above **0.9** with historical tokens. More strikingly, Figure 2b shows that over **60%** of the most similar tokens occur at distances greater than 16. This pervasive *long-range* similarity indicates that effective KV compression must leverage global retrieval rather than local heuristics.

Observation 2: *Highly Shared* Latent Components We further observe that KV caches share dominant high-norm latent directions, reflecting common linguistic and structural patterns. SVD analysis (Figure 2c) reveals a steep spectral decay in original KV representations, whereas residual KV exhibits a significantly flatter spectrum. Subtracting retrieved references effectively removes these shared components. As a result, residual KVs collapse to low-magnitude, noise-like signals: their L2 norms shrink substantially (Figure 2d), and their value distribution becomes sharply concentrated around zero (Figure 2f). This suggests that most KV cache capacity is consumed by redundant shared structure, while token-specific information is both low-energy and inherently easier to compress.

4. The DeltaKV Framework

Based on the preceding observations, we propose **DeltaKV**, a residual-based and GPU-friendly KV cache compression framework (Figure 3). DeltaKV compresses KV states by encoding the residuals between the current token and a small set of selected strided references, enabling high compression efficiency while preserving attention fidelity.

4.1. Residual-based KV Cache Compression

Rather than compressing raw KV representations, DeltaKV exploits the *long-range inter-token similarity* identified in Section 3. The core idea is to subtract information already captured by similar historical tokens and compress only the remaining residual signal.

Strided Reference Selection Searching the entire token history is both computationally and memory-intensive. DeltaKV, therefore, maintains a strided reference set $\mathcal{T}$ by selecting tokens at a fixed interval (stride) s:

$$\mathcal{T} = \{\boldsymbol{kv}_t \mid t \mod s = 0, t < i\},$$

where $\boldsymbol{kv}_t$ denotes the concatenation of key and value states across all heads *for a single token*, with $\boldsymbol{kv}_t \in \mathbb{R}^{2d_\mathrm{k}}$ (typically $d_\mathrm{k} = d_\mathrm{v} = ND$, hence $\boldsymbol{kv}_t \in \mathbb{R}^{2ND}$).

It is important to note that all operations in DeltaKV are performed on the **pre-RoPE** (Su et al., 2024) key-value states to ensure position-invariant representations. For the current token i, we retrieve the top-k nearest tokens from $\mathcal{T}$ based on the L_2 distance, denoted as $\mathcal{R}_i$:

$$\mathcal{R}_i = \arg\operatorname{topk}_{\boldsymbol{kv}_j \in \mathcal{T}} \left(- \|\boldsymbol{kv}_i - \boldsymbol{kv}_j\|_2^2 \right).$$

The reference representation is computed as their means:

$$\overline{\boldsymbol{KV}}_R = \tfrac{1}{k} \textstyle\sum_{j \in \mathcal{R}_i} \boldsymbol{kv}_j \in \mathbb{R}^{2d_\mathrm{k}}.$$

Compression and Reconstruction DeltaKV computes residuals in only two steps: compressing the token itself and the mean of the reference tokens, which allows for efficient parallel compression on the GPU.

Compressor Both the current KV and the reference average are projected using an MLP $f_c : \mathbb{R}^{2d_k} \to \mathbb{R}^{d_c}$, and then compute the residual vector $\boldsymbol{z}_\Delta \in \mathbb{R}^{d_c}$:

$$\boldsymbol{z}_\Delta = f_c(\boldsymbol{KV}) - f_c(\overline{\boldsymbol{KV}}_R).$$

Here, the compressor $f_c(x) = \mathrm{GeLU}(x\boldsymbol{W}_{c1} + b_{c1})\boldsymbol{W}_{c2} + b_{c2}$, where $\boldsymbol{W}_{c1} \in \mathbb{R}^{2d_k \times d_h}$ and $\boldsymbol{W}_{c2} \in \mathbb{R}^{d_h \times d_c}$ with hidden width d_h. Correspondingly, the compressed residual codes over a batch/sequence have shape $\boldsymbol{Z}_\Delta \in \mathbb{R}^{B \times L \times d_c}$.

Reconstruction To reconstruct the KV cache, the residual $\boldsymbol{z}_\Delta$ is decoded through a decompressor $f_d : \mathbb{R}^{d_c} \to \mathbb{R}^{2d_k}$:

$$\widehat{\boldsymbol{KV}}_\Delta = f_d(\boldsymbol{z}_\Delta).$$

Here, $f_d(\boldsymbol{z}_\Delta) = \mathrm{GeLU}(\boldsymbol{z}_\Delta \boldsymbol{W}_{d1} + \boldsymbol{b}_{d1})\boldsymbol{W}_{d2} + \boldsymbol{b}_{d2}$, with $\boldsymbol{W}_{d1} \in \mathbb{R}^{d_c \times d'_h}$, $\boldsymbol{b}_{d1} \in \mathbb{R}^{d'_h}$, $\boldsymbol{W}_{d2} \in \mathbb{R}^{d'_h \times 2d_k}$, and $\boldsymbol{b}_{d2} \in \mathbb{R}^{2d_k}$ with hidden width d'_h.

We provide two variants of the decompressor f_d: an MLP for higher fidelity, and a linear decoder for latency-critical settings (Section 5.3). The final KV cache is recovered as:

$$\widehat{\boldsymbol{KV}}_i = \widehat{\boldsymbol{KV}}_\Delta + \overline{\boldsymbol{KV}}_R,$$

and reshaped for attention computation.

4.2. Training and Inference

Training with Hybrid Objective Minimizing reconstruction error alone can suppress low-magnitude but attention-critical features. Thus, DeltaKV adopts a hybrid objective combining MSE and next-token prediction (NTP) loss:

$$\mathcal{L} = \sum \|\boldsymbol{KV} - \widehat{\boldsymbol{KV}}\|^2 + \mathcal{L}_{\mathrm{ntp}}(\theta, \phi),$$

where θ denotes frozen LLM parameters and ϕ the learnable DeltaKV modules. The NTP loss ensures preservation of features essential for end-to-end generation. The training procedure is detailed in Appendix A.

Inference with Sparse Attention DeltaKV is designed to seamlessly complement sparse attention methods such as OmniKV (Hao et al., 2025c). OmniKV designates a small subset of *filter layers* that compute global attention scores using the full KV Cache. Subsequently, the computed KV cache is directly utilized in other layers (sparse layers), as illustrated in Figure 3 and detailed in Appendix C.

To avoid the overhead of reconstructing all tokens in the filter layers, DeltaKV does not apply compression within these layers. Crucially, these layers are not a heuristic tuning for specific datasets but are grounded in the intrinsic heterogeneity of Transformer layers (Hao et al., 2025c). Since compression is performed independently per token, DeltaKV allows for *selective decompression*: we only reconstruct the KV pairs required by the sparse attention mask.

4.3. Sparse-vLLM Implementation

To enable efficient deployment, we design and implement **Sparse-vLLM**, a modular inference framework optimized for sparse and compressed KV layouts. Unlike existing frameworks that tightly couple memory management with model execution (Kwon et al., 2023; Zheng et al., 2024), Sparse-vLLM cleanly decouples these concerns. Its core design introduces a pluggable CacheManager to support diverse storage structures and a centralized Sparse Controller to uniformly manage sparse view construction and KV lifecycle. Implementation details are provided in Appendix B.

Modular CacheManager Sparse attention algorithms differ substantially in how KV caches are allocated, updated, and reclaimed. Sparse-vLLM addresses this heterogeneity through a modular CacheManager abstraction that encapsulates physical memory allocation and logical–physical mapping, enabling flexible integration of diverse sparsification and compression strategies.

Sparse Controller To decouple sparse algorithms from model architectures, we introduce a Sparse Controller that orchestrates sparse execution throughout the forward pass. It consists of the following stages: (1) Pre-Forward (View Construction): Before entering the attention operator, the Controller computes the logical view based on the currently configured algorithm; (2) Post-Forward (Lifecycle Management): After the computation is complete, the Controller is responsible for triggering the KV Cache update logic.

Efficient Kernel Execution At the operator level, Sparse-vLLM reuses high-performance Triton operators from the open-source ecosystem and introduces optimized kernels tailored for DeltaKV. Notably, the token-level Triton attention operator from LightLLM (Gong et al., 2025) efficiently operates on non-contiguous memory, naturally matching the CacheManager's discrete storage layout and eliminating costly memory defragmentation. We further fuse attention score extraction into the Triton kernel, avoiding redundant PyTorch-level computation and improving throughput.

5. Experiments

5.1. Experimental Setup

Training Configuration We evaluate DeltaKV on **Llama3.1-8B-Instruct** (Grattafiori et al., 2024), **Qwen2.5-7B-Instruct-1M** (Yang et al., 2025), **Qwen2.5-32B-Instruct** (Team et al., 2024), and **DeepSeek-R1-Distill-Qwen-7B** (Guo et al., 2025a). DeltaKV is lightweight to train, requiring only **160M tokens** and can be fully trained in **8 GPU hours** for standard 7B/8B models on a single NVIDIA RTX PRO 6000. Full hyperparameters, hardware details, and training overhead are provided in Appendix D.4.

Baselines We compare against three categories of strong

Table 1. **Main results on the LongBench benchmark.** We compare DeltaKV against state-of-the-art baselines across varying model scales. **KR** and **CR** denote the KV Cache Keep Ratio and Compute Ratio, respectively. Detailed calculation methods are provided in Appendix D.5. DeltaKV marked with † indicates the variant utilizing a lightweight decompressor for enhanced inference efficiency. "4-bit" indicates that we further quantize only the compressed KV Cache to further reduce GPU memory consumption.

Method	KR ↓	CR ↓	Single-Doc ↑	Multi-Doc ↑	Summ. ↑	Few-Shot ↑	Synthetic ↑	Code ↑	Overall ↑
Llama-3.1-8B (Grattafiori et al., 2024)	100	100	45.3	46.2	28.7	69.4	52.7	57.9	50.0
SnapKV (Li et al., 2024a)	30	30	44.5	46.4	26.5	68.2	52.7	60.3	49.8
PyramidKV (Cai et al., 2024)	30	30	44.1	46.3	26.0	68.5	52.6	59.1	49.5
AdaKV (Feng et al., 2025)	30	30	44.7	46.5	26.6	69.3	52.7	58.3	49.7
Quest (Tang et al., 2024)	100	30	44.4	46.2	**29.3**	69.0	53.1	57.6	50.0
OmniKV (Hao et al., 2025c)	100	30	**44.8**	46.1	28.9	68.9	52.8	59.9	50.2
+DeltaKV†	45	30	43.2	**46.8**	27.7	69.5	**54.8**	59.7	50.3
+DeltaKV	45	30	44.4	45.9	27.6	69.7	53.4	60.2	50.2
+4-bit	29	30	43.3	46.6	27.3	**69.8**	54.4	**60.5**	**50.3**
SnapKV (Li et al., 2024a)	20	20	42.9	45.4	25.8	68.8	54.8	57.4	49.2
Quest (Tang et al., 2024)	100	20	42.2	**46.9**	**28.8**	**68.9**	54.9	56.5	49.7
OmniKV (Hao et al., 2025c)	100	20	**43.0**	45.7	27.5	68.6	**54.9**	**60.8**	**50.1**
+DeltaKV†	43	20	42.7	46.2	26.2	68.7	54.3	60.5	49.8
Qwen2.5-7B (Yang et al., 2025)	100	100	42.5	49.6	28.6	68.7	53.8	42.5	47.6
SnapKV (Li et al., 2024a)	30	30	41.7	48.8	26.6	68.3	**54.5**	**41.8**	47.0
PyramidKV (Cai et al., 2024)	30	30	40.6	48.7	24.4	67.7	54.5	40.6	46.1
Palu (Chang et al., 2025)	50	100	34.4	35.6	27.5	68.7	45.0	21.3	38.8
OmniKV (Hao et al., 2025c)	100	30	**41.9**	**49.4**	**28.4**	69.1	54.0	41.5	**47.4**
+DeltaKV	48	30	41.8	49.0	27.7	**69.1**	53.3	41.7	47.1
Qwen2.5-32B (Team et al., 2024)	100	100	42.7	54.3	27.3	67.6	56.0	42.6	48.4
SnapKV (Li et al., 2024a)	20	20	39.5	53.8	24.6	67.2	56.0	41.7	47.1
OmniKV (Hao et al., 2025c)	100	20	**42.4**	54.2	**26.9**	**67.2**	**56.1**	41.7	**48.1**
+DeltaKV	44	20	41.9	**54.2**	26.2	66.1	55.4	**42.1**	47.7

baselines: (1) *Static eviction* methods (**SnapKV** (Li et al., 2024a), **PyramidKV** (Cai et al., 2024), and **AdaKV** (Feng et al., 2025)), which permanently remove tokens identified as unimportant and may lose critical information in multi-turn dialogues or complex reasoning scenarios. (2) *Dynamic sparsity* methods (**Quest** (Tang et al., 2024) and **OmniKV** (Hao et al., 2025c)), which select tokens adaptively for sparse attention during computation but retain the full KV Cache. (3) *KV cache compression*, where **Palu** (Chang et al., 2025) is most related but only considers individual token information and ignores inter-token similarity.

Evaluation To comprehensively evaluate DeltaKV, we conduct experiments on **LongBench** (Bai et al., 2024) (general long-context understanding), **SCBench** (Li et al., 2024b) (multi-turn dialogue), and **AIME** (MAA) (complex reasoning). To quantify the efficiency gains in terms of memory and computation, we report the KV Cache Keep Ratio (**KR**) and KV Cache Compute Ratio (**CR**). Dataset details and metric definitions are provided in Appendix D.1.

5.2. Downstream Performance

Simple Single-step LongBench Table 1 shows that DeltaKV achieves competitive performance across various models and scales on LongBench, matching the results of OmniKV. Furthermore, compared to OmniKV, DeltaKV is more memory-efficient, typically reducing KR by about half. It can also be combined with KV quantization for further memory savings without noticeable performance loss.

Complex Multi-turn SCBench Results in Table 2 indicate that DeltaKV preserves most of its performance in multi-turn settings. In contrast, static eviction methods like SnapKV often discard tokens that may become critical later, leading to significant performance degradation, particularly on Retrieval KV (**R.KV**). Although DeltaKV shows slightly larger drops on this task, it still outperforms static eviction baselines. We attribute the gap mainly to distribution mismatch, as R.KV contains many complex SSID-like strings.

Complex Reasoning AIME Table 3 demonstrates that DeltaKV remains effective on mathematical reasoning benchmarks, maintaining strong performance on AIME and confirming its applicability to reasoning-oriented models.

5.3. Inference Efficiency

Heavy Compressor, Light Decompressor Since KV cache compression is performed once per token while reconstruction occurs repeatedly during decoding, we adopt an asymmetric design to minimize runtime overhead. The compres-

Table 2. **Evaluation on SCBench.** We report results across four representative tasks: Retrieval KV (**R.KV**), English QA (**En.QA**), Mixture of Summarization and NIAH (**S+N**), and Many-Shot In-Context Learning (**MS**).

Method	KR ↓	CR ↓	R.KV ↑	En.QA ↑	S+N ↑	MS ↑	Avg. ↑
Llama-3.1-8B	100	100	79.0	21.7	56.8	44.1	50.4
SnapKV	30	30	0.4	20.3	55.7	43.7	30.0
OmniKV	100	30	**72.2**	**21.3**	**57.0**	46.3	**49.2**
+DeltaKV	45	30	58.0	20.5	50.0	51.5	45.0
+DeltaKV†	45	30	60.4	19.5	<u>52.4</u>	<u>53.0</u>	46.3
+4-bit	29	30	<u>60.4</u>	<u>20.7</u>	52.2	**53.7**	<u>46.8</u>
Qwen2.5-7B	100	100	70.4	22.9	60.7	57.0	52.8
SnapKV	30	30	6.2	21.3	60.6	56.3	36.1
OmniKV	48	30	**69.2**	22.7	<u>61.2</u>	56.7	**52.4**
+DeltaKV	48	30	59.4	**24.0**	60.6	**58.5**	50.6
+DeltaKV†	48	30	<u>62.4</u>	<u>23.4</u>	**61.3**	<u>58.2</u>	<u>51.3</u>

Table 3. Performance on the AIME reasoning benchmark.

DeepSeek-Qwen-7B	Full	SnapKV	OmniKV	DeltaKV
AIME ↑	50.0	33.3	**46.7**	<u>43.3</u>

sor uses a SwiGLU block,

$$f_{\mathrm{c}}(x) = (\mathrm{Swish}(x\boldsymbol{W}_1) \otimes (x\boldsymbol{W}_2))\boldsymbol{W}_3,$$

whereas the decompressor is a bias-free linear projection,

$$f_{\mathrm{d}}(x) = x\boldsymbol{W}_d.$$

Our experiments demonstrate that this design improves inference efficiency with negligible performance impact (Tables 1 and 2) and yields a 1.26× decoding speedup (Table 4).

Sparse-vLLM Inference Performance We measure decoding throughput of DeltaKV within Sparse-vLLM on a single NVIDIA RTX PRO 6000 (Blackwell), summarized in Table 4. Sparse-vLLM introduces minimal overhead: under full attention with 128k context, it achieves 135.0 tokens/s versus 143.2 tokens/s for native vLLM.

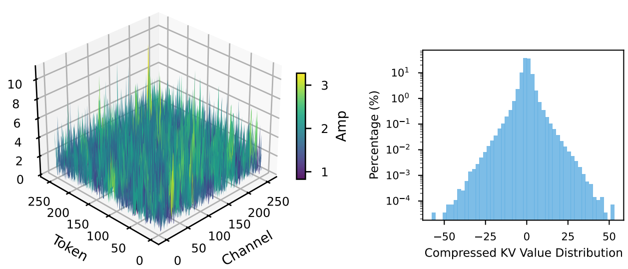
Enabling DeltaKV yields consistent throughput gains, especially at long contexts. At 128k, DeltaKV reaches 187.0 tokens/s, already exceeding vLLM. The advantage grows with sequence length: at 256k, DeltaKV provides a 1.7× speedup, and at 512k it achieves 67.7 tokens/s versus 33.1 tokens/s for vLLM (**2**× improvement).

Although static eviction methods (e.g., SnapKV) can achieve higher raw throughput by aggressively reducing computation, they incur significant accuracy degradation on complex tasks. DeltaKV instead offers a more favorable efficiency-accuracy trade-off.

Finally, our current implementation does not yet use a fully fused reconstruction–attention kernel. Fusing reconstruction into attention is a promising direction for further speedups. Sparse-vLLM also offers additional GPU memory savings, discussed in Appendix B.4.

Table 4. Sparse-vLLM inference decode throughput. "Avail. Max BS" refers to the **maximum batch size** supported by the current GPU memory. Throughput represents the number of tokens generated per second during the decoding phase.

Framework	Method	Avail. Max BS	Context Len.	Throughput
vLLM	Full Attn	8	128k	143.2
Sparse-vLLM	Full Attn	8	128k	135.0
Sparse-vLLM	SnapKV	8	128k	338.8
Sparse-vLLM	OmniKV	8	128k	216.7
Sparse-vLLM	DeltaKV	16	128k	148.4
Sparse-vLLM	DeltaKV†	16	128k	187.0
vLLM	Full Attn	4	256k	70.2
Sparse-vLLM	Full Attn	4	256k	69.5
Sparse-vLLM	SnapKV	4	256k	168.8
Sparse-vLLM	OmniKV	4	256k	115.9
Sparse-vLLM	DeltaKV†	8	256k	120.6
vLLM	Full Attn	2	512k	33.1
Sparse-vLLM	Full Attn	2	512k	32.1
Sparse-vLLM	DeltaKV†	4	512k	67.7
vLLM	Full Attn	1	900k	18.6
Sparse-vLLM	DeltaKV†	2	900k	38.9



(a) Visualization of absolute values. *(b)* Value distribution.

Figure 4. **Analysis of compressed KV cache values.** (a) The heatmap validates a highly uniform distribution across tokens and channels; (b) The histogram reveals concentration near 0.

5.4. Design Analysis and Ablations

Compatibility with Quantization We analyze the value distribution of compressed KV representations produced by Qwen2.5-7B (Figure 4). The distribution is highly uniform across channels and tokens, with no prominent spikes (Figure 4a). Most values concentrate near zero (Figure 4b), indicating strong suitability for quantization. Applying token-wise quantization from KIVI (Liu et al., 2024d) to z_Δ yields near-lossless performance (Tables 1, 2), confirming that DeltaKV's residual codes are quantization-friendly.

Train Short, Test Long Although compressors are trained only on sequences of length 8,192, they generalize well to contexts beyond 100K tokens. This suggests strong length generalization. We hypothesize that this arises from per-

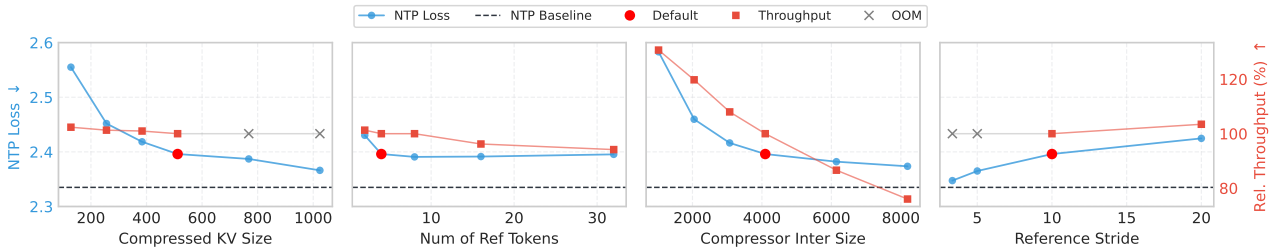


Figure 5. Ablation studies on model configurations.

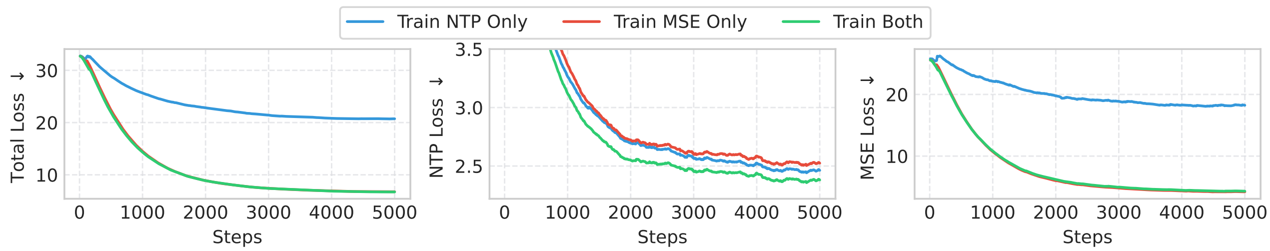


Figure 6. Ablation on NTP and MSE losses.

Table 5. Component-wise ablation study on Llama-3.1-8B.

Method	KR	CR	SD	MD	Sum	FS	Syn	Code	Avg.
DeltaKV	45	30	44.4	45.9	27.6	69.7	53.4	60.2	50.2
w/o f_c and f_d	45	30	34.4	43.2	24.5	65.9	55.1	57.2	46.7
w/o Ref. Tokens $\mathcal{T}$	47	30	39.3	45.3	22.1	62.3	50.8	55.4	45.9

forming compression before positional encoding, making the learned mapping largely position-invariant.

Ablation on Modules We conduct module ablations on Llama-3.1-8B-Inst ($2d_k = 2048$). DeltaKV has two core components: (1) residual construction using reference tokens, and (2) residual compression via a lightweight module. To ensure fair comparison, we increase d_c from 512 to 768 for a $\sim$35% KV keep ratio. Table 5 shows that removing either component causes substantial degradation, confirming that both residualization and compression are essential.

Hyperparameter Sensitivity We further investigate the sensitivity of DeltaKV to key hyperparameters, as illustrated in Figure 5. Specifically, we evaluate the throughput performance under various hyperparameters using a 128k sequence length and a batch size of 16, demonstrating the trade-off between precision and inference efficiency.

- **Compressed KV Size (d_c):** Increasing the dimension d_c consistently yields lower NTP loss; however, excessively large dimensions lead to diminishing returns and eventually trigger Out-Of-Memory (OOM) errors.
- **Number of Reference Tokens (k):** The performance gain saturates rapidly. While selecting a small number of reference tokens can lead to high average similarity, it often introduces more noise; conversely, selecting too many tokens reduces the average similarity. Thus, values of 4 or 8 serve as effective and efficient hyperparameters.
- **Compressor Intermediate Size (d_h):** While larger MLP hidden dimensions improve reconstruction quality by reducing loss, they introduce additional parameters and result in poorer inference efficiency.
- **Reference Stride (s):** A smaller stride s for reference tokens improves accuracy but linearly degrades inference throughput due to higher storage and retrieval overheads.

Ablation of Reconstruction and NTP Loss Figure 6 evaluates our dual-loss objective. Since NTP loss on Fineweb-Edu (Penedo et al., 2024) correlates with downstream performance, combining losses improves training effectiveness. Interestingly, MSE-only training reduces NTP loss, but NTP-only training does not reduce MSE. This suggests that while numerical reconstruction helps language modeling, exact KV reconstruction is not strictly required.

6. Conclusions

We introduced **DeltaKV**, a residual-based KV cache compression framework that exploits long-range inter-token similarity to remove redundant shared structure and retain only lightweight residuals. This design reduces KV memory usage to 29% of the original size while preserving the information most relevant for attention. When co-designed with our **Sparse-vLLM** inference engine, DeltaKV delivers up to $2\times$ higher decoding throughput and near-lossless accuracy on diverse long-context and reasoning benchmarks, including LongBench and AIME. Beyond empirical gains, we highlight two insights: (i) KV caches exhibit substantial global redundancy beyond local similarity, and (ii) residuals after reference subtraction have favorable statistics for compression and quantization. This suggests KV optimization should move beyond eviction and low-rank projection toward similarity-aware representations.

Impact Statement

This work contributes to the efficiency of long-context LLM serving by significantly reducing memory footprints and increasing inference throughput. These advancements promote energy-efficient computing and broaden the accessibility of powerful models on resource-constrained hardware.

While increasing the efficiency of foundational models is generally beneficial, it may also lower the barrier for indiscriminate large-scale deployment, potentially amplifying existing societal risks associated with LLMs. Additionally, although our compression method demonstrates robust performance, reliance on approximated representations could theoretically impact model reliability in unforeseen edge cases. We encourage practitioners to conduct thorough evaluations and adhere to responsible deployment guidelines when applying these techniques in real-world systems.

References

Alayrac, J.-B., Donahue, J., Luc, P., Miech, A., Barr, I., Hasson, Y., Lenc, K., Mensch, A., Millican, K., Reynolds, M., et al. Flamingo: a visual language model for few-shot learning. *Advances in neural information processing systems*, 35:23716–23736, 2022.

Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., et al. Longbench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 3119–3137, 2024.

Cai, Z., Zhang, Y., Gao, B., Liu, Y., Li, Y., Liu, T., Lu, K., Xiong, W., Dong, Y., Hu, J., et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2024.

Chang, C.-C., Lin, W.-C., Lin, C.-Y., Chen, C.-Y., Hu, Y.-F., Wang, P.-S., Huang, N.-C., Ceze, L., Abdelfattah, M. S., and Wu, K.-C. Palu: Kv-cache compression with low-rank projection. In *The Thirteenth International Conference on Learning Representations*, 2025.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Devoto, A., Jeblick, M., and Jégou, S. Expected attention: Kv cache compression by estimating attention from future queries distribution. *arXiv preprint arXiv:2510.00636*, 2025.

Feng, Y., Lv, J., Cao, Y., Xie, X., and Zhou, S. K. Ada-KV: Optimizing KV cache eviction by adaptive budget allocation for efficient LLM inference. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Gong, R., Bai, S., Wu, S., Fan, Y., Wang, Z., Li, X., Yang, H., and Liu, X. Past-future scheduler for llm serving under sla guarantees. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 798–813, 2025.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Guha, N., Nyarko, J., Ho, D., Ré, C., Chilton, A., Chohlas-Wood, A., Peters, A., Waldon, B., Rockmore, D., Zambrano, D., et al. Legalbench: A collaboratively built benchmark for measuring legal reasoning in large language models. *Advances in neural information processing systems*, 36:44123–44279, 2023.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025a.

Guo, S., Zhang, S., and Ren, Z. Enhancing rag efficiency with adaptive context compression. *arXiv preprint arXiv:2507.22931*, 2025b.

Hao, J., Huang, Q., Liu, H., Xiao, X., Ren, Z., and Yu, J. A token is worth over 1,000 tokens: Efficient knowledge distillation through low-rank clone. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025a.

Hao, J., Liu, H., Xiao, X., Huang, Q., and Yu, J. Uni-x: Mitigating modality conflict with a two-end-separated architecture for unified multimodal models. *arXiv preprint arXiv:2509.24365*, 2025b.

Hao, J., Zhu, Y., Wang, T., Yu, J., Xin, X., Zheng, B., Ren, Z., and Guo, S. Omnikv: Dynamic context selection for efficient long-context llms. In *The Thirteenth International Conference on Learning Representations*, 2025c.

He, Y., Zhang, L., Wu, W., Liu, J., Zhou, H., and Zhuang, B. Zipcache: Accurate and efficient kv cache quantization with salient token identification. *Advances in Neural Information Processing Systems*, 37:68287–68307, 2024.

Hooper, C., Kim, S., Mohammadzadeh, H., Mahoney, M. W., Shao, Y. S., Keutzer, K., and Gholami, A. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.

Hu, J., Wang, S., He, Y., Gong, P., Yi, J., Zhang, J., Bai, Y., Chen, R., Zhang, G., Li, C., et al. Efficient long-context llm inference via kv cache clustering. *arXiv preprint arXiv:2506.11418*, 2025.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Kim, J., Park, J., Cho, J., and Papailiopoulos, D. Lexico: Extreme kv cache compression via sparse coding over universal dictionaries. *arXiv preprint arXiv:2412.08890*, 2024.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Li, Y., Huang, Y., Yang, B., Venkitesh, B., Locatelli, A., Ye, H., Cai, T., Lewis, P., and Chen, D. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024a.

Li, Y., Jiang, H., Wu, Q., Luo, X., Ahn, S., Zhang, C., Abdi, A. H., Li, D., Gao, J., Yang, Y., et al. Scbench: A kv cache-centric analysis of long-context methods. *arXiv preprint arXiv:2412.10319*, 2024b.

Lin, H., Xu, H., Wu, Y., Guo, Z., Zhang, R., Lu, Z., Wei, Y., Zhang, Q., and Sun, Z. Quantization meets dllms: A systematic study of post-training quantization for diffusion llms. *arXiv preprint arXiv:2508.14896*, 2025.

Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024a.

Liu, A., Mei, A., Lin, B., Xue, B., Wang, B., Xu, B., Wu, B., Zhang, B., Lin, C., Dong, C., et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025a.

Liu, G., Li, C., Zhao, J., Zhang, C., and Guo, M. Clusterkv: Manipulating llm kv cache in semantic space for recallable compression. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pp. 1–7. IEEE, 2025b.

Liu, R., Bai, H., Lin, H., Li, Y., Gao, H., Xu, Z., Hou, L., Yao, J., and Yuan, C. Intactkv: Improving large language model quantization by keeping pivot tokens intact. *arXiv preprint arXiv:2403.01241*, 2024b.

Liu, Y., Li, H., Cheng, Y., Ray, S., Huang, Y., Zhang, Q., Du, K., Yao, J., Lu, S., Ananthanarayanan, G., et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 38–56, 2024c.

Liu, Z., Desai, A., Liao, F., Wang, W., Xie, V., Xu, Z., Kyrillidis, A., and Shrivastava, A. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36:52342–52364, 2023.

Liu, Z., Yuan, J., Jin, H., Zhong, S., Xu, Z., Braverman, V., Chen, B., and Hu, X. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024d.

MAA. Maa invitational competitions. `https://maa.org/maa-invitational-competitions/`. Accessed: 2026-01-28. Includes the American Invitational Mathematics Examination (AIME) section.

Nawrot, P., Li, R., Huang, R., Ruder, S., Marchisio, K., and Ponti, E. M. The sparse frontier: Sparse attention trade-offs in transformer llms. *arXiv:2504.17768*, 2025.

Oren, M., Hassid, M., Yarden, N., Adi, Y., and Schwartz, R. Transformers are multi-state rnns. *arXiv preprint arXiv:2401.06104*, 2024.

Penedo, G., Kydlíček, H., Lozhkov, A., Mitchell, M., Raffel, C. A., Von Werra, L., Wolf, T., et al. The fineweb datasets: Decanting the web for the finest text data at scale. *Advances in Neural Information Processing Systems*, 37: 30811–30849, 2024.

Qi, J., Gao, C., Ren, Z., and Chen, Q. Deltallm: A training-free framework exploiting temporal sparsity for efficient edge llm inference. *arXiv preprint arXiv:2507.19608*, 2025.

Saxena, U., Saha, G., Choudhary, S., and Roy, K. Eigen attention: Attention in low-rank space for kv cache compression. *arXiv preprint arXiv:2408.05646*, 2024.

Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Sun, H., Chang, L.-W., Bao, W., Zheng, S., Zheng, N., Liu, X., Dong, H., Chi, Y., and Chen, B. Shadowkv: Kv cache in shadows for high-throughput long-context llm inference. *arXiv preprint arXiv:2410.21465*, 2024.

Tang, J., Zhao, Y., Zhu, K., Xiao, G., Kasikci, B., and Han, S. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.

Team, Q. et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2(3), 2024.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pp. 38–45, 2020.

Xiao, C., Zhang, P., Han, X., Xiao, G., Lin, Y., Zhang, Z., Liu, Z., and Sun, M. Infllm: Training-free long-context extrapolation for llms with an efficient context memory. *Advances in Neural Information Processing Systems*, 37: 119638–119661, 2024.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International conference on machine learning*, pp. 38087–38099. PMLR, 2023a.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023b.

Yang, A., Yu, B., Li, C., Liu, D., Huang, F., Huang, H., Jiang, J., Tu, J., Zhang, J., Zhou, J., et al. Qwen2. 5-1m technical report. *arXiv preprint arXiv:2501.15383*, 2025.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Zhang, H., Ji, X., Chen, Y., Fu, F., Miao, X., Nie, X., Chen, W., and Cui, B. Pqcache: Product quantization-based kvcache for long context llm inference. *Proceedings of the ACM on Management of Data*, 3(3):1–30, 2025.

Zhang, R., Wang, K., Liu, L., Wang, S., Cheng, H., Zhang, C., and Shen, Y. Lorc: Low-rank compression for llms kv cache with a progressive compression strategy. *arXiv preprint arXiv:2410.03111*, 2024.

Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. SGLang: Efficient execution of structured language model programs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

A. Training Procedure of DeltaKV

Algorithm 1 details the training workflow of the DeltaKV framework, which optimizes the compression and decompression modules alongside the frozen LLM backbone. The procedure consists of three steps:

- **Ground Truth Forward:** The model first performs a standard forward pass using the original uncompressed KV cache to generate the target KV representations and logits.
- **DeltaKV Forward & Reconstruction:** In this step, the algorithm iterates through each layer and token sequence. For every current token, it retrieves the top-k most similar historical tokens from a strided reference set $\mathcal{T}_{ref}$ based on the L_2 distance. The mean of these references, $\overline{KV}_R$, is subtracted from the current token's representation to form a residual. This residual is compressed into a low-dimensional latent vector z_Δ via the compressor f_c and subsequently reconstructed by the decompressor f_d. The final approximated KV state, $\hat{kv}_i$, is obtained by adding the reconstructed residual back to the mean reference.
- **End-to-End Loss Calculation:** The training objective combines two loss functions: a reconstruction loss ($\mathcal{L}_{rec}$), which minimizes the Mean Squared Error (MSE) between the original and reconstructed KV states, and a Next Token Prediction loss ($\mathcal{L}_{ntp}$), which ensures the compressed cache maintains the model's generative capabilities. The total loss is computed as $\mathcal{L} = \mathcal{L}_{rec} + \mathcal{L}_{ntp}$.

Algorithm 1: Training Procedure of DeltaKV

Input: Input token sequence $\mathcal{T}$; number of layers L; frozen LLM parameters Θ; trainable compressor f_c; trainable decompressor f_d; stride s; number of references k;

Output: Total loss $\mathcal{L}$;

`// Step 1: Ground Truth Forward`

$\{\boldsymbol{KV}, \cdots\} \leftarrow \mathrm{Forward}(\mathcal{T}, \Theta)$; `// Cache original KV states`

`// Step 2: DeltaKV Forward & Reconstruction`

$\mathcal{L}_\mathrm{rec} \leftarrow 0$; $\boldsymbol{h} \leftarrow \mathrm{Embed}(\mathcal{T})$;

for $l = 1$ **to** L **do**

- $\mathcal{T}_\mathrm{ref} \leftarrow \emptyset$;
- $\boldsymbol{KV}^\mathrm{cur} \leftarrow \mathrm{Proj}(\boldsymbol{h}, \Theta_l)$; `// Compute current-layer KV for all tokens`
- **for** $i = 1$ **to** $|\mathcal{T}|$ **do**
  - $\boldsymbol{kv}_i \leftarrow \boldsymbol{KV}^\mathrm{cur}[i]$
  - $\mathcal{R}_i \leftarrow \arg\mathrm{top}k_{\boldsymbol{kv}_\mathrm{ref} \in \mathcal{T}_\mathrm{ref}} \left(-\|\boldsymbol{kv}_i - \boldsymbol{kv}_\mathrm{ref}\|_2^2\right)$;
  - $\overline{\boldsymbol{KV}}_R \leftarrow \mathrm{Mean}(\mathcal{R}_i)$; `// Mean of retrieved references`
  - $\boldsymbol{z}_\Delta \leftarrow f_\mathrm{c}(\boldsymbol{kv}_i) - f_\mathrm{c}(\overline{\boldsymbol{KV}}_R)$; `// Latent residual`
  - $\widehat{\boldsymbol{KV}}_\Delta \leftarrow f_\mathrm{d}(\boldsymbol{z}_\Delta)$;
  - $\widehat{\boldsymbol{kv}}_i \leftarrow \widehat{\boldsymbol{KV}}_\Delta + \overline{\boldsymbol{KV}}_R$; `// Reconstruct KV`
  - $\widehat{\boldsymbol{KV}}^\mathrm{cur}[i] \leftarrow \widehat{\boldsymbol{kv}}_i$;
  - $\mathcal{L}_\mathrm{rec} \leftarrow \mathcal{L}_\mathrm{rec} + \|\boldsymbol{KV}_l[i] - \widehat{\boldsymbol{kv}}_i\|^2$;
  - **if** $i \bmod s = 0$ **then**
    - $\mathcal{T}_\mathrm{ref} \leftarrow \mathcal{T}_\mathrm{ref} \cup \{\widehat{\boldsymbol{kv}}_i\}$; `// Reference set: only tokens with index` $i \bmod s = 0$
- $\boldsymbol{h} \leftarrow \mathrm{AttnFFN}(\boldsymbol{h}, \widehat{\boldsymbol{KV}}^\mathrm{cur}, \Theta_l)$;

`// Step 3: End-to-End Loss Calculation`

$\boldsymbol{O}^\mathrm{DeltaKV} \leftarrow \mathrm{LMHead}(\boldsymbol{h})$;

$\mathcal{L}_\mathrm{ntp} \leftarrow \mathrm{CrossEntropy}(\boldsymbol{O}^\mathrm{DeltaKV}, \mathcal{T}_\mathrm{next})$;

$\mathcal{L} \leftarrow \mathcal{L}_\mathrm{rec} + \mathcal{L}_\mathrm{ntp}$; `// Joint optimization`

return $\mathcal{L}$;

B. Implementation Details of Sparse-vLLM

This appendix provides the technical specifications of the Sparse-vLLM architecture, focusing on the data structures and algorithmic workflows that enable the modularity described in the main text.

B.1. CacheManager Data Structures

The `CacheManager` features an extensible architecture where internal data layouts can be customized to match the memory access patterns of emerging algorithms. To demonstrate this flexibility, we currently provide implementations for three representative storage backends catering to physical eviction, logical masking, and hybrid compression paradigms:

Per-Layer Independent Mapping (for Physical Eviction) Algorithms like SnapKV and PyramidKV diverge in their token retention across layers. To support this without approximation errors, the `CacheManager` instantiates L independent page tables (where L is the number of layers). Each table is a tensor `buffer_req_to_token_slots[layer_idx]`, mapping logical positions to physical slots. While this increases metadata memory usage by a factor of L, it is strictly necessary for algorithms where the "KV view" is physically discontinuous and unique per layer.

Global Shared Mapping (for Logical Masking) For Full Attention and OmniKV, where tokens are retained globally but masked logically, we maintain a unified `req_to_token_slots` table shared across all layers. This minimizes metadata overhead and maximizes cache locality for the mapping tables during kernel execution.

Heterogeneous DeltaKV Storage For DeltaKV, the `CacheManager` introduces a tiered storage system to handle the duality of raw and compressed data:

- **Dual Physical Pools**: It manages a `Full Pool` for high-precision tokens (Sink/Recent) and a separate `Latent Pool` for compressed vectors. The system dynamically allocates slots from these pools based on the token's lifecycle state.
- **Intra-Group Slot Sharing**: To optimize the Observation-Sparse layer groups, the manager implements a "Copy-on-Write" style mechanism for reconstruction. When an Observation Layer identifies Top-K tokens, the subsequent sparse layers share the underlying temporary slots allocated for reconstruction. This prevents redundant decompression operations within the same group, significantly reducing the memory bandwidth pressure during the pre-forward phase.

B.2. Sparse Controller Workflows

The Sparse Controller orchestrates the interaction between the model and the CacheManager. Below, we detail the specific workflow implemented for the DeltaKV mechanism:

DeltaKV View Construction (Pre-Forward) Unlike standard retrieval, DeltaKV requires on-the-fly reconstruction. The Controller executes the following pipeline before the attention operation:

(1) **Index Resolution**: Based on the reference tokens, the Controller identifies the logical indices of tokens requiring decompression.
(2) **Batch Reconstruction**: It instructs the `CacheManager` to fetch compressed vectors from the `Latent Pool` and their corresponding references.
(3) **Slot Virtualization**: The reconstructed KV pairs are written to a temporary physical buffer. The Controller then constructs a virtual `slot_mapping` that stitches together the static slots (Sink/Recent) and these temporary dynamic slots, presenting a contiguous logical view to the attention kernel.

DeltaKV Lifecycle Management (Post-Forward) To handle the transition from high-precision to compressed storage, the Controller monitors the `Recent Buffer` boundary. Upon buffer overflow, it triggers a specialized fused kernel that:

(1) Computes the residual between the overflowed token and its assigned reference tokens.
(2) Compresses the residual via the down-projection encoder.
(3) Writes the result to the `Latent Pool` and frees the original `Full Pool` slots immediately, ensuring constant memory complexity relative to sequence length.

B.3. Kernel Optimizations

While we leverage standard high-performance operators (e.g., FlashAttention with indirect addressing), DeltaKV necessitates specific kernel optimizations to minimize overhead:

- **Indirect Addressing via Slot Mapping**: We modified the Flash-Decoding kernels to accept a token-level `req_to_token_slots` index array. This allows the attention mechanism to read directly from non-contiguous physical memory locations without intermediate copy operations or block-table lookups.
- **Fused DeltaKV Kernels**: We implemented custom Triton kernels to accelerate the compression/decompression loop.

This includes a **Batch L2 Distance** kernel for rapid reference searching and a **Fused Reconstruction** kernel that combines the gathering of reference tokens, mean calculation, and residual addition into a single kernel launch to minimize GPU memory bandwidth consumption. In the main text, except for the configurations used in ablation studies, the reference stride is generally set to $s = 10$. This implies that even for a sequence of 1M tokens, there are only approximately 100k references, and performing efficient matrix multiplication on the GPU remains remarkably fast. Consequently, we did not consider employing approximate algorithms such as Approximate Nearest Neighbors (ANN).

B.4. Potential Memory Efficiency

Although DeltaKV already achieves significant memory reduction ($\approx 29\%$), there remains substantial headroom for further optimization. Currently, to guarantee near-lossless performance and simplify engineering implementation, we strictly limit quantization to the compressed residuals z_Δ, while keeping both the reference tokens and the full attention layers in high precision (e.g., BF16).

Full-Pipeline Quantization. Integrating DeltaKV with advanced quantization techniques for the uncompressed components could yield extreme compression rates. If we were to apply 4-bit quantization uniformly across the full attention layers and the reference tokens in the sparse layers—complementing the already quantized residuals—the theoretical memory footprint could plummet to approximately **7.2%** of the original size. This calculation assumes a global 4-bit representation reduces the storage requirement by $4\times$ relative to BF16, superimposed on DeltaKV's structural sparsity. While this poses challenges in maintaining accuracy and necessitates complex kernel fusion, it represents a promising frontier for deploying massive-context models on consumer-grade hardware.

Synergy with Offloading and Caching. DeltaKV is inherently compatible with system-level optimizations like memory offloading.

- **Reduced Bandwidth Overhead:** By compressing token-specific information into low-magnitude residuals, DeltaKV effectively reduces the "unit volume" of the KV cache. When combined with quantization, the data transfer requirement drops to nearly $1/16$ of the standard breakdown, significantly alleviating the PCIe bandwidth bottleneck that plagues traditional offloading schemes.
- **Fine-Grained Cache Management:** Furthermore, our proposed Sparse-vLLM framework manages memory at the token level rather than the page level. This granularity naturally aligns with sophisticated cache eviction policies (e.g., LRU or LFU). By keeping only the most frequently accessed compressed residuals and references in GPU memory while offloading the rest to host memory, DeltaKV could enable virtually infinite context lengths with minimal latency penalties.

B.5. Detailed Latency Profiling and Future Optimization

As illustrated in Figure 7, DeltaKV enables long-context inference under a tight memory budget, but the current prototype still incurs substantial *runtime* overhead. At $BS = 16$, the measured step latency is 91.0 ms, of which 37.3 ms is attributed to KV reconstruction and 24.7 ms to view/slot bookkeeping, leaving the remainder for model computation. To obtain stable and attributable timings, we insert CUDA synchronizations around key regions; this forces the host to wait for GPU completion and reduces overlap between kernel launches and other asynchronous work. As a result, the reported latencies should be interpreted as conservative (upper-bound) measurements of the current software stack, and can be higher than end-to-end throughput observed under fully asynchronous execution.

The bottleneck mainly stems from two factors: (1) *Python-level control overhead*: sequence-wise logic, slot mapping, and dynamic view construction are executed serially in Python, triggering many small kernel launches and occasional host-device synchronization, with the impact growing with batch size. (2) *Fragmented memory traffic*: reconstruction and bookkeeping are decomposed into multiple operators that materialize intermediate tensors in HBM; although some kernels are already fused, redundant global-memory reads/writes and launch overhead remain significant.

These costs indicate clear optimization opportunities via deeper operator fusion. A more integrated Triton/CUDA implementation could consolidate reconstruction (e.g., gather/de-RoPE, delta application, re-RoPE, and writeback) into fewer passes and move view/slot management onto GPU kernels, reducing both launch overhead and global-memory traffic by keeping temporaries in on-chip storage (registers/shared memory) when feasible. While the exact gain depends on hardware and workload, we expect that eliminating most Python-driven bookkeeping and further fusing reconstruction could reduce the $BS = 16$ step latency to the ~ 55–60 ms range (about $1.5\times$–$1.7\times$ higher throughput), making the system more practical for large-scale long-context deployment.

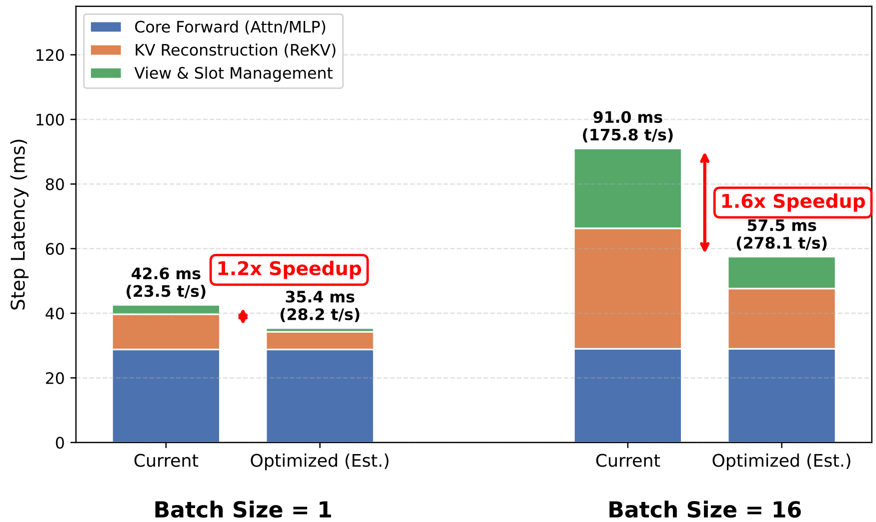


Figure 7. Detailed latency breakdown and estimated speedup of Sparse-vLLM following kernel fusion, evaluated at a 128k context length with batch sizes of 1 and 16. **CUDA synchronization** was enabled at points during testing for the purpose of latency analysis, which **incurs some performance overhead**.

C. Brief Introduction to OmniKV

OmniKV (Hao et al., 2025c) is a training-free framework designed to optimize KV cache memory usage and inference latency for long-context LLMs. It operates on the core insight of *Inter-Layer Attention Similarity*, which posits that the set of important tokens identified in a specific layer remains significant for subsequent layers. Specifically, OmniKV designates a small subset of layers as "filter layers", denoted as $\mathbb{L}_{filter}$. During the decoding phase, for any layer $l \in \mathbb{L}_{filter}$, the model computes the full attention scores using the current query and the entire historical KV cache. Based on these scores, a Context Selector identifies the indices of the top-k most relevant tokens, denoted as $\mathcal{I}_{topk}$. For the subsequent layers that are not in $\mathbb{L}_{filter}$, OmniKV avoids full attention computation; instead, it selectively retrieves only the subset of keys and values corresponding to $\mathcal{I}_{topk}$ (i.e., $K_{\mathcal{I}_{topk}}$ and $V_{\mathcal{I}_{topk}}$) from the CPU-offloaded Context Bank to the GPU. This dynamic context selection mechanism significantly reduces the GPU memory footprint and PCIe bandwidth overhead while maintaining model performance.

Prefill Acceleration via Chunking. Processing long prompts (e.g., >100k tokens) in a single forward pass often exceeds GPU memory limits. To address this, OmniKV employs *Chunk Prefill*, where the long input sequence is split into smaller segments (chunks) of length L_q that are processed sequentially. Consequently, during the prefill phase, the length of the queries (L_q) corresponds to the current chunk size, while the length of the Key/Value cache (L_{kv}) represents the accumulated history plus the current chunk, implying $L_q \leq L_{kv}$.

To accelerate this phase without retaining the full history, OmniKV identifies important tokens by aggregating attention scores. Formally, let H denote the number of attention heads. Given the attention score matrix $\mathbf{A} \in \mathbb{R}^{H \times L_q \times L_{kv}}$ computed in the filter layers for the current chunk, the importance score s_j for the j-th KV token is derived by first averaging over the query length L_q to capture temporal relevance, and then taking the maximum across the head dimension H to retain the strongest signal:

$$s_j = \max_{1 \leq h \leq H} \left(\frac{1}{L_q} \sum_{i=1}^{L_q} \mathbf{A}_{h,i,j} \right) \tag{1}$$

By utilizing this score, OmniKV identifies the top-k globally significant tokens from the current L_{kv} candidates. This strategy allows the model to discard less relevant tokens on-the-fly after processing each chunk, preventing memory explosion while preserving critical long-range context.

Table 6. Model checkpoints used in our experiments.

Model	Huggingface Model ID
Llama-3.1-8B-Instruct	`meta-llama/Llama-3.1-8B-Instruct`
Qwen2.5-7B-Instruct-1M	`Qwen/Qwen2.5-7B-Instruct-1M`
Qwen2.5-32B-Instruct	`Qwen/Qwen2.5-32B-Instruct`
DeepSeek-R1-Distill-Qwen-7B	`deepseek-ai/DeepSeek-R1-Distill-Qwen-7B`

D. Detailed Configurations

D.1. Benchmarks and Task Selection

LongBench Following the setting of AdaKV (Feng et al., 2025), we utilize 16 datasets from LongBench covering Single/Multiple Document QA, Summarization, Few-Shot learning, Synthetic tasks, and Code generation. This benchmark primarily assesses single-turn dialogue capabilities.

SCBench To evaluate the model's performance sustainability in realistic multi-turn dialogues, we employ SCBench. Due to computational resource constraints, we selected four representative tasks covering distinct categories: Retr.KV (String Retrieval), En.QA (Semantic Retrieval), ICL.ManyShot (Global Information), and Mix.Sum+NIAH (Multi-tasking).

AIME We further verify the effectiveness of DeltaKV on complex reasoning tasks using the AIME benchmark, with a maximum output length set to 32,768 tokens.

Budget Ratio Setting To ensure a fair comparison across methods, we enforce a budget ratio rather than a fixed token number. Consequently, the actual number of retained tokens is dynamically determined based on the prompt context length (Appendix D.5).

D.2. Efficiency Metrics

To quantify the trade-offs between memory saving and computational acceleration, we utilize two perspectives:

- **KV Cache Keep Ratio (KR):** Represents the percentage of KV cache stored in GPU memory relative to the full cache.
- **KV Cache Compute Ratio (CR):** Represents the percentage of tokens involved in the actual attention computation.

For specific calculation formulas regarding different baselines and DeltaKV, please refer to Appendix D.5.

D.3. Model Checkpoints

We provide the specific Hugging Face model identifiers for all models evaluated in this work in Table 6. The experiments were conducted using the official Instruct versions of Llama-3.1, the Qwen2.5 series (including the 1M context variant), and the DeepSeek-R1 distilled model.

D.4. Detailed Setting of DeltaKV

Training Configuration. We train the DeltaKV modules using the AdamW optimizer with a learning rate of 2e-4, a batch size of 1, and 1 gradient accumulation step. The learning rate schedule includes a linear warmup for the first 2% of the training steps, followed by a linear decay to zero.

Environment and Efficiency: All experiments are implemented with `PyTorch` and `transformers` (Wolf et al., 2020) on a single NVIDIA RTX PRO 6000 (Blackwell). For Qwen2.5-32B-Instruct, we utilized `bitsandbytes` and gradient checkpointing (Chen et al., 2016) to reduce memory consumption. Training is efficient: for standard models (Llama-3.1-8B and Qwen2.5-7B), we trained on packed sequences of length 8,192 (160M tokens total) from **Fineweb-Edu**, completing in just **8 GPU hours**. The 32B model required 14 GPU hours. For the reasoning model (DeepSeek-R1-Distill), we sampled 8,000 sequences from **AM-DeepSeek-R1-Distilled-1.4M**, taking only 3.5 GPU hours.

Architecture and Hyperparameters. For the compression architecture, we set the compressed residual dimension d_c to 25% of the original KV dimension (i.e., $d_c = 0.25 \times 2d_k$). The compressor utilizes a hidden dimension of $d_h = 4096$ for the standard version. For the latency-optimized "Light" variant, we reduce the hidden dimension to $d_h = 3072$ and employ a SwiGLU-based encoder paired with a bias-free linear decoder. During retrieval, we select the top-$k = 4$ nearest reference tokens from the history. The reference tokens are maintained with a stride of $s = 10$, corresponding to a reference keep ratio

Table 7. Configuration of Full Attention Layers for different models and budgets. Layers not listed are compressed.

Model & Setting	Full Attention Layer Indices
Llama-3.1-8B (30% Budget)	0, 1, 2, 8, 18
Llama-3.1-8B (20% Budget)	0, 1, 10, 18
Qwen2.5-7B (30% Budget)	0, 1, 2, 4, 7, 14
Qwen2.5-32B (20% Budget)	0, 1, 2, 3, 4, 5, 17, 29, 40

of approximately 10%.

Layer-wise Configuration. To maximize performance under strict memory budgets, we employ a hybrid strategy where a small subset of "Full Attention Layers" retain their complete KV cache, while the remaining layers are compressed using DeltaKV. The specific layer indices for full attention are manually selected based on importance profiling and are listed in Table 7.

D.5. KR and CR Calculation

To comprehensively evaluate the efficiency of DeltaKV against existing approaches, we utilize two key metrics: KV Cache Keep Ratio (**KR**) and KV Cache Compute Ratio (**CR**). Let r denote the target budget ratio (e.g., the percentage of tokens retained or selected for attention). The calculation methods for different categories of baselines and our proposed DeltaKV are formulated as follows:

Baselines Existing methods exhibit distinct trade-offs between memory footprint and computational overhead:

- **Static Eviction Methods (e.g., SnapKV, PyramidKV, AdaKV):** These methods permanently evict tokens to meet the budget constraint. Consequently, both the storage and the computation are reduced proportionally. Here, r represents the sparsity ratio, indicating the proportion of the KV Cache that participates in each attention computation.

$$\text{KR} = r, \quad \text{CR} = r \tag{2}$$

- **Dynamic Selection Methods (e.g., OmniKV, Quest):** These approaches retain the full KV cache in GPU memory to preserve history but dynamically select only the top-k important tokens for attention computation at each step.

$$\text{KR} = 100\%, \quad \text{CR} = r \tag{3}$$

- **Low-Rank Compression (e.g., Palu):** This method compresses the KV cache along the hidden dimension but typically requires reconstructing the full cache for attention computation, offering memory savings without computational acceleration.

$$\text{KR} = \frac{d_{\text{low}}}{d_{\text{orig}}}, \quad \text{CR} = 100\% \tag{4}$$

  where d_{low} and d_{orig} represent the compressed low-rank dimension and the original hidden dimension, respectively.

DeltaKV Our method employs a hybrid layer strategy, where L_{full} layers perform standard full attention and L_{sparse} layers utilize our residual compression with sparse attention.

- **KV Cache Keep Ratio (KR):** The memory footprint consists of the full cache for standard layers, and for sparse layers, the strided reference tokens (stride s) plus the compressed residuals (dimension d_c).

$$\text{KR}_{\text{DeltaKV}} = \frac{L_{\text{full}}}{L_{\text{sparse}} + L_{\text{full}}} + \frac{L_{\text{sparse}}}{L_{\text{sparse}} + L_{\text{full}}} \left(\frac{1}{s} + \frac{d_c}{2d_k} \right) \tag{5}$$

  where $L_{\text{sparse}} + L_{\text{full}}$ is the total number of layers and $2d_k$ is the dimension of the original KV Cache.
- **KV Cache Compute Ratio (CR):** DeltaKV adopts the same sparse attention mechanism as OmniKV for the compressed layers. Therefore, the computational cost is determined by the budget ratio r applied in the sparse layers.

$$\text{CR}_{\text{DeltaKV}} = \frac{L_{\text{full}}}{L_{\text{sparse}} + L_{\text{full}}} \times 100\% + \frac{L_{\text{sparse}}}{L_{\text{sparse}} + L_{\text{full}}} \times r \tag{6}$$

Table 8. KV Cache Keep Ratio (KR) and Compute Ratio (CR) Calculations for Llama-3.1-8B ($L = 32$). Comparison between DeltaKV and Baselines under 30% and 20% Budgets.

Budget	Method	Configuration Details	Formula Breakdown	KR (%)	CR (%)
30%	SnapKV/PyramidKV	Static Eviction ($r = 0.3$)	$\text{KR} = r$	30.0	30.0
	OmniKV/Quest	Dynamic Selection ($r = 0.3$)	$\text{KR} = 100\%$	100.0	30.0
	DeltaKV	$L_{\text{full}} = 5$ (Idx: 0, 1, 2, 8, 18) $L_{\text{sparse}} = 27$ $1/s = 0.10$ (Ref. Ratio) $d_c/2d_k = 0.25$ (Comp. Rate)	$\text{KR} = \frac{5}{32} + \frac{27}{32}\left(\underbrace{0.10}_{1/s} + \underbrace{0.25}_{d_c/2d_k}\right)$ $= 0.156 + 0.844 \times 0.35$	**45.2**	**30.0**
20%	SnapKV/PyramidKV	Static Eviction ($r = 0.2$)	$\text{KR} = r$	20.0	20.0
	OmniKV/Quest	Dynamic Selection ($r = 0.2$)	$\text{KR} = 100\%$	100.0	20.0
	DeltaKV	$L_{\text{full}} = 4$ (Idx: 0, 1, 10, 18) $L_{\text{sparse}} = 28$ $1/s = 0.10$ (Ref. Ratio) $d_c/2d_k = 0.25$ (Comp. Rate)	$\text{KR} = \frac{4}{32} + \frac{28}{32}\left(\underbrace{0.10}_{1/s} + \underbrace{0.25}_{d_c/2d_k}\right)$ $= 0.125 + 0.875 \times 0.35$	**43.1**	**20.0**

Settings Table 8 details the calculation of KV Cache Keep Ratio (KR) and Compute Ratio (CR) for Llama-3.1-8B ($L_{\text{sparse}} + L_{\text{full}} = 32$) under 30% and 20% computation budgets. We compare DeltaKV against static eviction (e.g., SnapKV) and dynamic selection (e.g., OmniKV) baselines. For DeltaKV, we adopt a hybrid layer strategy where specific layers retain full attention based on importance profiling. In the 30% budget setting, 5 layers (indices 0, 1, 2, 8, 18) are kept full ($L_{\text{full}} = 5$), while the remaining 27 layers are compressed. In the 20% budget setting, 4 layers (indices 0, 1, 8, 10) are kept full ($L_{\text{full}} = 4$). The compressed layers utilize a residual compression rate of 25% ($d_c/2d_k = 0.25$) and a reference token stride of $s = 10$ ($1/s = 0.10$). The analysis shows that DeltaKV achieves a significantly reduced memory footprint (KR ≈ 43–45%) compared to dynamic selection methods (100% KR), while matching the efficient Compute Ratio (CR) of static baselines.

E. Detailed and Other Experiments

E.1. LongBench

In this section, we present the detailed experimental results on the LongBench benchmark. Tables 9 and 10 provide a comprehensive performance breakdown across all 16 datasets, covering Single-Document QA, Multi-Document QA, Summarization, Few-Shot Learning, Synthetic tasks, and Code generation. These results cover varying model scales, including Llama-3.1-8B-Instruct, Qwen2.5-7B-Instruct-1M, and Qwen2.5-32B-Instruct, offering a granular view of the aggregated metrics reported in the main text.

E.2. SCBench

We report the detailed subtask results for SCBench in Tables 13 and 14. These tables illustrate the performance stability of DeltaKV across complex multi-turn scenarios, including Retrieval KV (R-1 to R-5), English QA (E-1 to E-7), Mixture of Summarization and NIAH (S-1 to S-10), and Many-Shot In-Context Learning (M-1 to M-5). The breakdown highlights the method's robustness in retaining critical information over long contexts compared to static eviction baselines.

Table 9. Performance on Single-Doc QA, Multi-Doc QA, and Summarization tasks. Comparison with state-of-the-art baselines.

Method	KR ↓	CR ↓	Single-Doc QA				Multi-Doc QA				Summarization			
			NQA ↑	Qasp ↑	MFQA ↑	Avg. ↑	HPQA ↑	2WQA ↑	Musq ↑	Avg. ↑	GovR ↑	QMSm ↑	MNew ↑	Avg. ↑
Llama-3.1-8B-Instruct														
Full Cache	100	100	32.2	46.6	56.9	45.3	58.1	48.0	32.4	46.2	34.3	24.9	27.0	28.7
SnapKV	30	30	31.0	45.3	57.2	44.5	57.4	49.0	32.9	46.4	30.0	25.1	24.3	26.5
PyramidKV	30	30	32.1	43.2	56.9	44.1	57.5	49.0	32.5	46.3	29.4	24.8	23.9	26.0
Quest	100	30	31.8	45.0	56.4	44.4	57.9	48.7	32.1	46.2	35.6	25.2	27.2	29.3
KIVI	25	100	31.1	46.5	56.7	44.8	58.0	49.2	31.4	46.2	34.3	25.5	27.2	29.0
AdaKV	30	30	31.6	45.2	57.1	44.7	58.7	48.3	32.5	46.5	30.6	24.7	24.3	26.6
OmniKV	100	30	31.6	46.0	56.8	44.8	58.1	48.2	32.1	46.1	34.3	25.2	27.0	28.9
+DeltaKV	45	30	32.1	43.9	57.0	44.4	58.8	47.7	31.4	45.9	31.5	25.3	25.9	27.6
+DeltaKV†	45	30	29.5	44.3	55.8	43.2	57.4	49.7	33.3	46.8	32.4	25.4	25.5	27.7
+4-bit	29	30	30.5	43.2	56.2	43.3	57.6	49.3	33.0	46.6	31.6	25.0	25.3	27.3
Qwen2.5-32B-Instruct														
Full Cache	100	100	29.5	46.3	52.4	42.7	63.1	60.7	39.2	54.3	32.6	24.3	24.9	27.3
SnapKV	20	20	30.8	38.7	48.8	39.5	62.8	59.6	38.9	53.8	29.1	22.8	21.9	24.6
OmniKV	100	20	29.7	46.4	51.0	42.4	62.1	60.5	40.0	54.2	32.2	23.9	24.6	26.9
+DeltaKV	44	20	30.0	44.3	51.4	41.9	62.3	61.2	39.2	54.2	30.4	23.7	24.4	26.2
Qwen2.5-7B-Instruct-1M														
Full Cache	100	100	29.4	47.7	50.3	42.5	60.4	54.8	33.4	49.6	35.5	24.4	25.9	28.6
SnapKV	30	30	28.8	46.1	50.3	41.7	59.3	53.5	33.5	48.8	32.8	24.0	23.0	26.6
PyramidKV	30	30	29.1	42.9	49.6	40.6	59.0	53.3	33.7	48.7	29.6	23.6	20.1	24.4
Palu	50	100	24.4	31.7	47.2	34.4	48.2	39.7	18.9	35.6	31.1	25.1	26.3	27.5
OmniKV	100	30	28.9	47.9	48.8	41.9	60.3	54.5	33.3	49.4	35.3	24.1	25.7	28.4
+DeltaKV	48.9	30	29.2	46.7	49.6	41.8	59.5	53.5	33.9	49.0	33.8	24.2	25.0	27.7

Table 10. Performance on Few-Shot, Synthetic, and Code tasks, with Overall Average. (Part 2 of Main Results).

Method	KR ↓	CR ↓	Few-Shot				Synthetic			Code			Avg. ↑
			Trec ↑	TQA ↑	SamS ↑	Avg. ↑	Cnt ↑	Retr ↑	Avg. ↑	LCC ↑	Repo ↑	Avg. ↑	
Llama-3.1-8B-Instruct													
Full Cache	100	100	73.0	92.0	43.2	69.4	6.0	99.5	52.7	63.4	52.3	57.9	50.0
SnapKV	30	30	70.0	91.9	42.6	68.2	6.0	99.5	52.7	63.1	57.4	60.3	49.8
PyramidKV	30	30	71.0	91.7	42.9	68.5	5.7	99.5	52.6	62.3	56.0	59.1	49.5
Quest	100	30	73.0	90.5	43.6	69.0	6.6	99.5	53.1	59.9	55.3	57.6	50.0
KIVI	25	100	72.5	91.7	44.2	69.5	7.6	100.0	53.8	63.0	56.6	59.8	50.5
AdaKV	30	30	73.0	92.4	42.4	69.3	6.0	99.5	52.7	63.7	52.8	58.3	49.7
OmniKV	100	30	72.5	92.1	42.1	68.9	6.1	99.5	52.8	63.1	56.6	59.9	50.2
+DeltaKV	45	30	73.0	92.3	43.6	69.7	6.8	100.0	53.4	63.5	57.0	60.2	50.2
+DeltaKV†	45	30	73.0	92.3	43.1	69.5	10.2	99.5	54.8	63.5	56.0	59.7	50.3
+4-bit	29	30	73.0	92.6	43.8	69.8	10.3	98.5	54.4	63.6	57.1	60.4	50.3
Qwen2.5-32B-Instruct													
Full Cache	100	100	72.0	84.5	46.5	67.6	12.0	100.0	56.0	50.9	34.3	42.6	48.4
SnapKV	20	20	71.0	84.1	46.4	67.2	12.0	100.0	56.0	49.2	34.1	41.7	47.1
OmniKV	100	20	71.5	83.7	46.4	67.2	12.3	100.0	56.1	49.2	34.2	41.7	48.1
+DeltaKV	44	20	71.5	81.7	45.2	66.1	10.9	100.0	55.4	49.6	34.6	42.1	47.7
Qwen2.5-7B-Instruct-1M													
Full Cache	100	100	77.0	84.1	45.1	68.7	7.5	100.0	53.8	47.9	37.1	42.5	47.6
SnapKV	30	30	75.0	85.0	45.0	68.3	9.0	100.0	54.5	46.7	36.8	41.8	47.0
PyramidKV	30	30	74.5	84.2	44.5	67.7	9.0	100.0	54.5	44.7	36.5	40.6	46.1
Palu	50	100	76.0	86.4	43.7	68.7	2.5	87.5	45.0	20.2	22.4	21.3	38.8
OmniKV	100	30	77.5	85.4	44.3	69.1	8.0	100.0	54.0	46.1	36.8	41.5	47.4
+DeltaKV	48.9	30	77.5	84.6	45.2	69.1	8.0	98.5	53.3	45.6	37.8	41.7	47.1

Table 11. Component-wise ablation study on Llama-3.1-8B-Instruct (Part 1).

Method	KR ↓	CR ↓	Single-Doc QA				Multi-Doc QA				Summarization			
			NQA ↑	Qasp ↑	MFQA ↑	Avg. ↑	HPQA ↑	2WQA ↑	Musq ↑	Avg. ↑	GovR ↑	QMSm ↑	MNew ↑	Avg. ↑
DeltaKV	45	30	32.1	43.9	57.0	44.4	58.8	47.7	31.4	45.9	31.5	25.3	25.9	27.6
w/o f_c and f_d	45	30	28.5	32.3	42.3	34.4	53.4	45.8	30.3	43.2	26.8	23.8	22.8	24.5
w/o Ref. Tokens $\mathcal{T}$	47	30	30.7	36.4	50.8	39.3	58.1	46.3	31.3	45.3	22.0	22.8	21.5	22.1

Table 12. Component-wise ablation study on Llama-3.1-8B-Instruct (Part 2).

Method	KR ↓	CR ↓	Few-Shot				Synthetic			Code			Avg. ↑
			Trec ↑	TQA ↑	SamS ↑	Avg. ↑	Cnt ↑	Retr ↑	Avg. ↑	LCC ↑	Repo ↑	Avg. ↑	
DeltaKV	45	30	73.0	92.3	43.6	69.7	6.8	100.0	53.4	63.5	57.0	60.2	50.2
w/o f_c and f_d	45	30	66.0	90.0	41.7	65.9	10.1	100.0	55.1	60.6	53.9	57.2	46.7
w/o Ref. Tokens $\mathcal{T}$	47	30	55.5	90.8	40.5	62.3	7.0	94.5	50.8	59.0	51.8	55.4	45.9

Table 13. SCBench Results Part 1: Performance on Retrieval KV and English QA tasks. (R-x: Retrieval subtasks, E-x: English QA subtasks).

Method	KR ↓	CR ↓	Avg. ↑	Retrieval KV						English QA							
				Avg. ↑	R-1 ↑	R-2 ↑	R-3 ↑	R-4 ↑	R-5 ↑	Avg. ↑	E-1 ↑	E-2 ↑	E-3 ↑	E-4 ↑	E-5 ↑	E-6 ↑	E-7 ↑
Llama-3.1-8B-Instruct																	
Full Cache	100	100	50.4	79.0	60.0	77.0	86.0	85.0	87.0	21.7	30.4	27.5	29.0	31.9	33.3	0.0	0.0
SnapKV	30	30	30.0	0.4	0.0	0.0	1.0	0.0	1.0	20.3	29.0	23.2	26.1	33.3	30.4	0.0	0.0
OmniKV	100	30	49.2	72.2	49.0	70.0	80.0	79.0	83.0	21.3	29.0	29.0	26.1	30.4	34.8	0.0	0.0
+DeltaKV	45	30	45.0	58.0	38.0	51.0	68.0	62.0	71.0	20.5	27.5	29.0	21.7	31.8	33.3	0.0	0.0
+DeltaKV†	45	30	46.3	60.4	37.0	54.0	72.0	63.0	76.0	19.5	27.5	23.2	23.2	30.4	31.9	0.0	0.0
+4-bit	29	30	46.8	60.4	34.0	51.0	73.0	66.0	78.0	20.7	29.0	27.5	24.6	31.9	31.9	0.0	0.0
Qwen2.5-7B-Instruct-1M																	
Full Cache	100	100	52.8	70.4	67.0	68.0	72.0	72.0	73.0	22.9	30.4	23.2	27.5	30.4	29.0	20.0	0.0
SnapKV	30	30	36.1	6.2	17.0	5.0	4.0	3.0	2.0	21.3	31.9	23.2	21.7	23.2	29.0	20.0	0.0
OmniKV	48	30	52.4	69.2	67.0	67.0	72.0	70.0	70.0	22.7	31.9	23.2	30.4	27.5	26.1	20.0	0.0
+DeltaKV	48	30	50.6	59.4	54.0	60.0	63.0	59.0	61.0	24.0	36.2	27.5	27.5	30.4	26.1	20.0	0.0
+DeltaKV†	48	30	51.3	62.4	57.0	63.0	65.0	65.0	62.0	23.4	30.4	29.0	26.1	30.4	27.5	20.0	0.0

Table 14. SCBench Results Part 2: Performance on Mixture of Summarization+NIAH and Many-Shot tasks. (S-x: Summ+NIAH subtasks, M-x: Many-Shot subtasks).

Method	KR ↓	CR ↓	Mix.Sum + NIAH											Many-Shot					
			Avg. ↑	S-1 ↑	S-2 ↑	S-3 ↑	S-4 ↑	S-5 ↑	S-6 ↑	S-7 ↑	S-8 ↑	S-9 ↑	S-10 ↑	Avg. ↑	M-1 ↑	M-2 ↑	M-3 ↑	M-4 ↑	M-5 ↑
Llama-3.1-8B-Instruct																			
Full Cache	100	100	56.8	39.0	42.9	39.2	45.7	45.0	82.9	43.8	91.4	49.6	88.6	44.1	44.4	70.4	53.7	25.9	25.9
SnapKV	30	30	55.7	40.1	45.7	35.4	54.3	43.9	77.1	43.4	91.4	48.9	77.1	43.7	38.9	81.5	44.4	29.6	24.1
OmniKV	100	30	57.0	39.0	45.7	39.5	42.9	44.7	82.9	45.4	91.4	49.5	88.8	46.3	42.6	68.5	57.4	33.3	29.6
+DeltaKV	45	30	50.0	37.3	28.6	35.7	8.6	42.9	74.3	44.2	91.4	48.0	88.6	51.5	46.3	74.1	59.3	35.2	42.6
+DeltaKV†	45	30	52.4	38.2	37.1	38.4	2.9	43.4	82.9	44.9	91.4	50.4	94.3	53.0	46.3	75.9	61.1	35.2	46.3
+4-bit	29	30	52.2	38.8	34.3	38.3	0.0	43.2	88.6	43.8	91.4	49.7	94.3	53.7	48.1	79.6	59.3	35.2	46.3
Qwen2.5-7B-Instruct-1M																			
Full Cache	100	100	60.7	34.9	8.6	44.7	91.4	40.0	97.1	42.8	100.0	47.9	100.0	57.0	51.9	48.1	57.4	64.8	63.0
SnapKV	30	30	60.6	36.8	8.6	43.4	94.3	40.5	94.3	41.7	100.0	46.9	100.0	56.3	50.0	53.7	61.1	63.0	53.7
OmniKV	48	30	61.2	36.9	8.6	45.6	91.4	39.7	97.1	43.8	100.0	48.9	100.0	56.7	51.9	48.1	59.3	63.0	61.1
+DeltaKV	48	30	60.6	35.2	8.6	41.4	94.3	40.7	100.0	41.8	100.0	47.3	97.1	58.5	53.7	48.1	59.3	63.0	68.5
+DeltaKV†	48	30	61.3	34.8	8.6	43.2	94.3	41.5	100.0	43.5	100.0	47.0	100.0	58.2	53.7	46.3	59.3	66.7	64.8