DiffKV: Differentiated Memory Management for Large Language Models with Parallel KV Compaction

Yanqi Zhang*
yanqizhang2015@gmail.com
Huawei

Yuwei Hu*
huyuwei1995@gmail.com
Huawei

Runyuan Zhao
zhaorunyuan23@gmail.com
Huawei

John C.S. Lui
cslui@cse.cuhk.edu.hk
The Chinese University of Hong Kong

Haibo Chen†
haibochen@sjtu.edu.cn
Shanghai Jiao Tong University

ABSTRACT

Large language models (LLMs) demonstrate remarkable capabilities but face substantial serving costs due to their high memory demands, with the key-value (KV) cache being a primary bottleneck. State-of-the-art KV cache compression techniques, such as quantization and pruning, apply uniform treatment to both keys and values, and discard unimportant tokens entirely, overlooking the fine-grained distinctions in the significance of individual KV cache components. To address such limitations, we introduce *DiffKV*, a novel framework for efficient KV cache compression that exploits three levels of differentiation in the KV cache: (1) the differing impact of keys and values on attention computation, (2) the varying importance of tokens, and (3) the diverse dynamic sparsity patterns across attention heads. These levels of differentiation introduce irregular memory usage patterns across different requests and attention heads, posing significant scalability challenges for memory management. To address these challenges, DiffKV proposes an on-GPU memory manager that compacts fragmented free memory list into contiguous regions in parallel, effectively translating sparsity in the KV cache into performance gains. We evaluate DiffKV on several mainstream LLMs, including the emerging thinking models that generate extended chains of thought. DiffKV is able to compress the KV cache by 2.7× to 5.7× with near-lossless accuracy on complex workloads requiring sophisticated reasoning and long-generation capabilities, and enhances throughput by 1.9× to 5.4×.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Computer systems organization** → *Parallel architectures*; • **Hardware** → Emerging architectures.

KEYWORDS

Machine Learning System, Large Language Model Serving

*Both authors contributed equally to this research.
†Haibo Chen is the corresponding author.

SOSP '25, October 13–16, 2025, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1870-0/2025/10
https://doi.org/10.1145/3731569.3764810

ACM Reference Format:
Yanqi Zhang, Yuwei Hu, Runyuan Zhao, John C.S. Lui, and Haibo Chen. 2025. *DiffKV*: Differentiated Memory Management for Large Language Models with Parallel KV Compaction. In *ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3731569.3764810

1 INTRODUCTION

Large language models (LLMs) like GPT [4, 10, 56] and Gemini [67] have demonstrated significant potential to impact our daily lives, offering promising applications in areas including chatbots [13, 14], programming [24, 27], mathematics [60, 79] and medical assistance [59, 73]. Despite their exceptional performance, hosting LLMs is costly due to their large model sizes, demanding extensive hardware resources. Given the pervasive adoption of LLMs, enhancing serving efficiency has become critically important [7, 21, 28, 38, 43, 57, 63, 76].

To avoid redundant computation, LLM inference frameworks typically cache intermediate key and value tensors in memory, commonly referred to as the KV cache [39]. The size of the KV cache scales linearly with both sequence length and the number of concurrent requests, often comprising over 90% of total memory consumption [22, 39, 74, 75]. As state-of-the-art models continue to support longer sequences [4, 20, 36, 46, 67, 69], and with the rise of recent thinking models [26, 35, 68] that generate extended reasoning processes, the KV cache has emerged as a critical bottleneck for LLM serving efficiency. It limits the number of concurrent requests and increases attention computation latency due to its memory bandwidth bound nature [16, 17].

Researchers have investigated various compression techniques, primarily focused on pruning [11, 23, 42, 72, 77] and quantization [2, 32, 45]. Pruning reduces the token sequence length by eliminating unimportant tokens based on attention scores. Quantization, on the other hand, reduces the size of the KV cache by converting floating-point feature values into lower-precision representations.

While effective, these methods overlook the nuanced variations in importance across different components of the KV cache. First, existing pruning approaches employ a one-size-fits-all strategy, statically allocating memory uniformly across attention heads despite the per-head dynamic attention sparsity patterns. Second, current quantization schemes apply uniform precision to all key and value vectors, ignoring both the distinct roles of keys versus values in the attention mechanism, and the varying significance of different tokens.

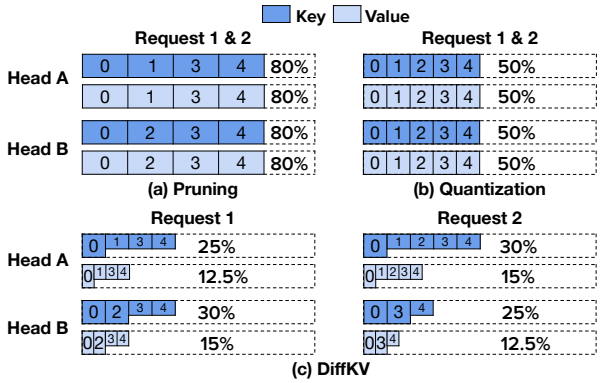


Figure 1: KV cache memory allocation patterns for (a) pruning, (b) uniform quantization, and (c) DiffKV across two attention heads in two 5-token requests. Boxes represent retained tokens (annotated with token IDs), with box size proportional to memory usage. Percentages indicate total memory usage relative to an uncompressed KV cache.

Finally, both pruning and quantization treat all requests uniformly, overlooking variability in information density across requests.

To address these limitations and advance KV cache compression, we propose *DiffKV*, a novel framework that exploits three levels of differentiation within the KV cache:

1. Differentiated Keys and Values. DiffKV assigns higher precision to key vectors than to value vectors, based on the observation that keys (critical for attention score computation) play a more significant role than values (used primarily for weighted aggregation) in attention computations.

2. Token Importance Differentiation. Guided by attention scores, DiffKV stores tokens at varying precision levels based on their significance. Most critical tokens are quantized at higher precision, less critical tokens are quantized at lower precision, and the least significant tokens are pruned.

3. Per-head Dynamic Sparsity. Attention score distributions exhibit dynamic sparsity across both heads and requests: the number of important tokens varies not only across different heads, but also for the same head under different requests. To address this variability, DiffKV dynamically identifies critical tokens per-head, per-request, adapting memory allocation accordingly.

Figure 1 compares the KV cache usage of pruning, quantization, and DiffKV across two heads in two 5-token requests. In both requests, pruning uniformly eliminates one token per head (reducing cache usage to 80%), failing to adapt to heads with naturally sparser attention patterns. Quantization, meanwhile, applies INT8 precision uniformly to all FP16 values, halving memory usage but missing opportunities for finger-grained savings by treating keys and values differently based on their relative importance. Moreover, both approaches treat two requests identically, overlooking their distinct sparsity patterns.

In contrast, DiffKV exhibits more adaptive memory usage. In Request 1, for Head A, DiffKV prunes one token and further distinguishes precision across the rest: Token 0, deemed highly important, is stored at high precision (key in INT8, value in INT4), while Tokens 1, 3, and 4 are stored at lower precision (keys in INT4, values in INT2). This reduces usage to 25% for keys and 12.5% for values. For Head B, DiffKV instead identifies two tokens as highly important, resulting in 30% and 15% KV cache usage for keys and values, respectively. Averaged across both heads, DiffKV uses 20.6% of memory for Request 1, outperforming pruning and quantization. Additionally, DiffKV also adapts flexibly across requests. In Request 2, Head A stores one token at high precision and the remaining four at lower precision without pruning, whereas Head B prunes two tokens and stores two others at high precision. These examples highlight DiffKV's ability to leverage both per-head dynamic sparsity and request-level variation, enabling more aggressive and flexible compression than prior methods.

Although DiffKV achieves a high KV cache compression ratio, it introduces significant memory usage irregularity. As depicted in Figure 1, pruning and quantization maintain uniform KV cache utilization across attention heads. In contrast, DiffKV exhibits substantial variability, not only across different heads but also between the key and value within individual heads. To mitigate fragmentation, the memory manager must precisely track and allocate memory per head and per request. However, this introduces a scalability challenge: even a modestly sized LLM like Llama3-8B [20] comprises hundreds of attention heads, and with hundreds of concurrent requests, the memory manager must handle tens of thousands of heterogeneous memory regions during each inference step. Considering that model execution time per step is on the order of tens of milliseconds, improper memory management could render the overhead of dynamic allocation prohibitive, negating the benefits of KV cache compression.

To overcome the scalability challenge, DiffKV introduces an on-GPU memory manager that efficiently handles the irregular memory usage patterns via *parallel KV compaction*, which optimizes memory allocation and recycling by packing fragmented free memory lists into contiguous regions directly on the GPU in parallel. Parallel KV compaction is enabled by three essential on-GPU data structures:

1. Unified Pages. GPU memory is partitioned into fixed-size pages, each storing tokens at a specific precision. For example, a page may hold tokens with keys in INT8 and values in INT4. Each page is dynamically configured and parsed according to the precision requirements of the head and request.

2. Circular Free Page List. All page IDs are managed in a centralized, GPU-resident circular list. Both free and used pages are kept in contiguous regions, enabling efficient page allocation and recycling via parallel prefix sum [29].

3. Bidirectional Page Table. Instead of maintaining separate page tables for high- and low-precision pages, DiffKV consolidates both into a single bi-directional page table to minimize metadata overhead. High-precision page IDs grow from the left side of the table, and low-precision page IDs grow from the right side. Additional precision levels can also be efficiently accommodated by employing multiple bi-directional page tables.

We have implemented DiffKV on vLLM [39] and evaluate it across multiple LLMs, including the emerging thinking models QwQ-32B [68], R1-Distill-Qwen-14B and R1-Distill-Llama-8B [26]. Our experiment results demonstrate that DiffKV offers a superior cost-accuracy tradeoff. Specifically, DiffKV compresses the KV cache by 2.7× to 5.7× with near-lossless accuracy, resulting in throughput improvement of 1.9× to 5.4×. Notably, DiffKV is, to

the best of our knowledge, the first KV cache compression framework evaluated on thinking models and complex reasoning tasks that require advanced chain-of-thought (CoT) capabilities, while achieving FP16-comparable generation quality.

2 BACKGROUND AND RELATED WORK

In this section, we describe Transformer-based large language models and review existing techniques for KV cache management.

2.1 Large Language Models

Large language models (LLMs) are predominantly built upon the Transformer architecture [70]. At the core of Transformer is the attention mechanism, which allows each token in a sequence to weigh the significance of other tokens when constructing its contextualized representation. During autoregressive inference, the attention mechanism operates in a causal manner, attending only to preceding tokens. Mathematically, the standard attention computation is defined as:

$$\begin{aligned}\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V})_i &= \sum_{j=1}^{i} \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)_{ij}\mathbf{v}_j \\ &= \sum_{j=1}^{i} \frac{\exp\left(\frac{\mathbf{q}_i\cdot\mathbf{k}_j}{\sqrt{d}}\right)}{\sum_{n=1}^{i}\exp\left(\frac{\mathbf{q}_i\cdot\mathbf{k}_n}{\sqrt{d}}\right)}\mathbf{v}_j\end{aligned} \tag{1}$$

$\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ are matrices of size $l \times d$, representing the queries, keys and values respectively, where l denotes the number of tokens processed so far in the sequence and d represents the feature dimensionality. Vectors $\mathbf{q}_i$, $\mathbf{k}_i$, and $\mathbf{v}_i$ correspond to the query, key, and value of the i^{th} token.

To capture a broader range of interactions between tokens, Transformer models employ multi-head attention (MHA), where each head independently computes attention using distinct projections of queries, keys, and values. Grouped-query attention (GQA) [6, 61] improves the efficiency of MHA by allowing a group of query heads to share the same projection of keys and values, referred to as a KV head.

LLM execution involves two phases: the *prompt phase*, where the model computes latent representations for all tokens in the prompt and generates the first new token, and the *generation phase*, where subsequent tokens are generated iteratively. To avoid redundant computations across generation steps, *KV cache* is introduced to store the keys and values of all previous tokens. However, the size of the KV cache grows linearly with both the sequence length and batch size, quickly becoming a bottleneck for inference throughput [39, 75]. Therefore, efficient KV cache management is critical for alleviating memory bottlenecks and improving LLM serving efficiency.

2.2 KV Cache Optimization

PagedAttention. Static KV cache management systems [1, 75] reserve memory for the maximum possible sequence length, leading to considerable memory waste when actual sequences are shorter. To mitigate this inefficiency, vLLM [39] introduces *PagedAttention*, which partitions the KV cache into pages, each containing a fixed number of tokens, and allocates the pages on-demand as the sequence length grows. By managing the KV cache at the granularity of pages, vLLM reduces memory waste and enables larger batch sizes to improve serving efficiency.

KV Cache Quantization. Quantization [18, 25, 32, 51] reduces the KV cache size by approximating high-precision floating points with discrete low-bit values. For a tensor $\mathbf{X}$, we first compute the scale s and zero point z based on $\mathbf{X}_{min}$ and $\mathbf{X}_{max}$, then apply quantization element-wise as: $\mathbf{Q} = \text{round}\left(\frac{\mathbf{X}-z}{s}\right)$. During inference, the original tensor is approximately reconstructed via dequantization: $\hat{\mathbf{X}} = s\cdot\mathbf{Q}+z$. Unlike the quantized tensor $\mathbf{Q}$, the metadata s and z are kept in higher precision (e.g., FP16) to ensure more accurate dequantization. State-of-the-art KV cache quantization methods such as Atom [78] and Qserve [45] apply this process to each key and value vector independently. However, these uniform quantization methods, which apply a fixed bit-width across keys and values of all tokens, may not be optimal. They overlook the varying importance of tokens as well as the different roles of keys and values in the attention calculation.

KV Cache Pruning. Several recent works [11, 23, 42, 48, 72, 77] have explored KV cache pruning to alleviate the memory bottleneck in LLM inference, which can be considered as an extreme case of quantization. These methods use attention scores to assess token importance and evict less important tokens, thereby reducing the KV cache size. The effectiveness of these methods, however, is limited by their static and inflexible allocation of memory resources. Specifically, H2O [77] and SnapKV [42] allocate the same memory budget uniformly across all attention heads and layers. PyramidKV [11] allocates more memory in lower layers and less in higher ones, based on the empirical observation that the number of important tokens shrinks as the model depth increases, but still relies on static heuristics and cannot dynamically adjust cache allocations to suit various workloads. Complementary to KV cache pruning, another line of work [40, 62] addresses GPU memory constraints by offloading the entire KV cache to CPU memory and selectively loading important tokens to the GPU for attention computations. However, these methods do not inherently reduce the KV cache size and incur additional latency due to data transfers between the CPU and GPU.

3 THE CASE FOR DIFFERENTIATED KV

We motivate the design of DiffKV by presenting the case for three levels of differentiation within the KV cache: (i) the differentiated impacts of keys and values (Section 3.1), (ii) differentiated token importance (Section 3.2), and (iii) per-head dynamic sparsity (Section 3.3). Building on these insights, Section 3.4 describes how DiffKV combines them together and highlights its contributions over prior systems.

3.1 Differentiated Impacts of Keys and Values

While the significance of different tokens can be directly inferred from attention scores, the roles of the key and value vectors within a token are less obvious. However, examining Equation 1 reveals a divergence in their impacts. Each token's contribution to the attention output depends on two factors: the attention score from softmax and the value vector. Key vectors, as part of the shared softmax denominator, influence the attention scores of all tokens, whereas value vectors only affect their respective token's contribution to the output.

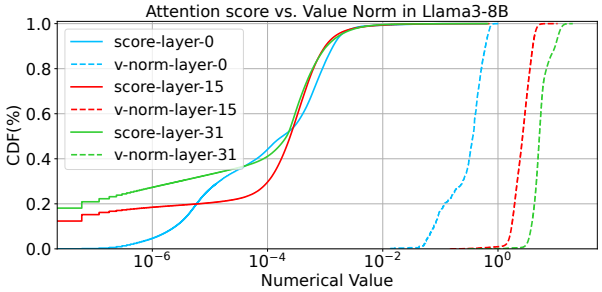


Figure 2: Distribution of attention score and value vector norm in Llama3-8B.

Beyond the broader impact of key vectors, it is crucial to establish the *relative significance* of attention scores, determined by key vectors, over value vectors. Without this, the significance of key vectors could be overshadowed by that of value vectors. To this end, we reformulate the attention mechanism as a weighted sum of unit vectors, as presented in Equation 2. This formulation decomposes each input token's contribution into two components: the unit vector $\frac{v_j}{|v_j|}$, obtained by dividing the token's value vector by its L2 norm to capture only its direction in feature space, and a coefficient, defined as the product of the attention score and the norm of the value vector, which determines the relative significance of the token's direction. This allows us to assess the relative importance of attention scores and value vectors by examining their contributions to these coefficients.

$$\text{Attn}(\mathbf{Q},\mathbf{K},\mathbf{V})_i = \sum_{j=1}^{i} \underbrace{\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)_{ij} |\mathbf{v}_j|}_{\text{Coefficient}} \underbrace{\frac{\mathbf{v}_j}{|\mathbf{v}_j|}}_{\text{Unit vector}} \tag{2}$$

Figure 2 presents the distributions of the average attention score per token and the norms of value vectors across three representative layers of the Llama3-8B model [20], evaluated on the first one thousand articles from the Wikitext dataset [50]. Notably, the attention scores span seven orders of magnitude, vastly exceeding the range of value vector norms, which only cover at most two orders of magnitude. Similar patterns are also observed in the larger Llama3-70B model. This pronounced disparity highlights the pivotal role of attention scores in determining each token's contribution to the attention output. We therefore conclude that key vectors exert a broader and more impactful influence than value vectors, motivating further exploration of processing key and value vectors at distinct precision levels. We validate this intuition in Section 7.2, showing that K8V4 and K4V2, which quantize keys to 8/4 bits and values to 4/2 bits respectively, outperform their mirror configurations K4V8 and K2V4.

3.2 Differentiated Token Importance

Tokens contribute to attention outputs with varying degrees of importance, as reflected by their attention scores. By exploiting these differences, we can apply finer-grained compression strategies that go beyond uniformly quantizing all tokens [45, 78] or exclusively pruning the least important ones [11, 77].

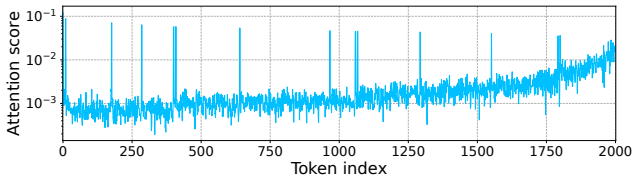


Figure 3: Per-token attention scores in the 8^{th} layer of Llama3-8B on a sequence randomly sampled from Wikitext.

Figure 3 shows the per-token attention scores in the 8^{th} layer, selected as a representative layer of Llama3-8B [20], on a sequence randomly sampled from the Wikitext dataset [50]. Similar sparsity patterns are observed across the other layers as well. This non-uniform distribution of token importance motivates a *hierarchical* compression strategy that allocates memory based on token significance: high-precision storage (e.g., K8V4) for the most important tokens, lower precision (e.g., K4V2) for moderately important tokens, and pruning for the least important. This hierarchical approach enables a smoother trade-off between memory savings and generation quality than uniform quantization methods (e.g., Atom [78], QServe [45]) and pruning-only approaches (e.g., H2O [77], SnapKV [42]).

3.3 Per-head Dynamic Sparsity Patterns

LLMs exhibit dynamic sparsity patterns that vary across both attention heads and requests: the number of critical tokens can differ not only between heads but also for the same head under different requests. To characterize the sparsity of attention, we analyze the minimum number of critical tokens required to retain the majority of information, specifically by preserving a target percentage (e.g., 95%) of the total attention score.

We begin by investigating dynamic sparsity patterns across layers, and evaluate the Llama3-8B model with the first thousand articles from the Wikitext dataset. Figure 4 illustrates the average number of critical tokens per layer, aggregated across all KV heads, required to retain 95% of the total attention score. Vertical bars represent the standard deviation, capturing variability across individual requests. Notably, the degree of sparsity varies considerably across layers, as indicated by differences in the number of critical tokens.

Next, we delve into the dynamic sparsity within individual layers. Figure 5 presents the average number of critical tokens per KV head for three representative layers, with horizontal bars indicating the standard deviation across requests. The sparsity pattern remains highly dynamic: within each layer, the number of critical tokens varies significantly across KV heads. Furthermore, within individual KV heads, the number of critical tokens can vary substantially across requests, as shown by the high standard deviation in some KV heads.

3.4 Main Insights and Implications

Our findings reveal three critical levels of differentiation within the KV cache:

- Keys exert a broader impact on attention computation than values, motivating differentiated precisions for keys and values.
- Tokens vary in importance as reflected by their attention scores, motivating a fine-grained, hierarchical compression strategy.

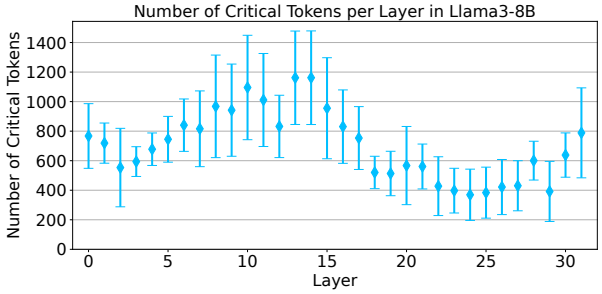


Figure 4: Number of critical tokens per layer in Llama3-8B to preserve 95% of the total attention score.

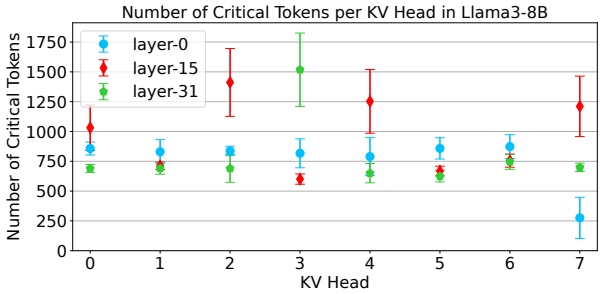


Figure 5: Number of critical tokens per KV head in Llama3-8B to preserve 95% of the total attention score.

- Attention sparsity patterns vary across requests and attention heads, requiring dynamically managing memory resources on a per-request and per-head basis.

Collectively, these highly dynamic sparsity patterns, across layers, attention heads, and individual requests, underscore the necessity of adaptive memory management on a per-head and per-request basis. They motivate the design of DiffKV, which integrates differentiated precisions for keys and values, hierarchical compression based on importance, and per-request and per-head memory management.

Contributions of DiffKV. DiffKV expands the design space for KV cache compression by *jointly exploiting the three levels of differentiation in KV cache*, enabling higher compression ratios with minimal quality degradation. Compared to prior work that explores importance-based KV cache pruning [5, 42, 66, 77] and mixed-precision quantization [19, 30, 78], DiffKV introduces the following key innovations:

- *Request-aware differentiated KV quantization.* DiffKV adaptively adjusts the mix of high- and low-precision tokens (e.g., K8V4, K4V2) on a per-request basis (Section 4), whereas prior work [5, 19, 78] typically applies a single static configuration across all requests.
- *Sequence length-adaptive importance estimation.* DiffKV modulates the proportion of important tokens based on sequence length, retaining more tokens in shorter sequences to preserve quality, while enabling more aggressive compression for longer ones (Section 4). Prior approaches [5, 42, 77] typically use a fixed token retention ratio, regardless of sequence length.
- *Per-head dynamic sparsity exploitation*. DiffKV dynamically adjusts memory allocation per head, based on observed sparsity patterns (Section 3.3 & 4), whereas existing methods allocate memory uniformly across attention heads, missing optimization opportunities.
- *Scalable on-GPU memory management*. DiffKV introduces an efficient on-GPU memory manager that handles irregular per-head memory allocation patterns, effectively translating memory savings into performance gains (Section 5).

4 KV COMPRESSION POLICY

We present the KV compression policy of DiffKV, designed to address the three levels of differentiation within the KV cache. First, to reflect the greater impact of keys on attention computation compared to values, we propose quantizing keys at a higher precision than values. For example, keys and values can be quantized to 8 and 4 bits (K8V4), or 4 and 2 bits (K4V2). Second, to account for the varying importance of tokens, we introduce a hierarchical compression strategy that classifies tokens into three significance levels: the most important tokens are quantized at high precision (e.g., K8V4), moderately important tokens at lower precision (e.g., K4V2), and the least important tokens are pruned entirely. This strategy is both request-aware and sequence-length adaptive. It dynamically adjusts the mix of high- and low-precision tokens per request to reflect their varying sparsity patterns. Moreover, it retains a larger fraction of tokens at high precision for short sequences to preserve quality, while applying low precision quantization and pruning more aggressively for long sequences to maximize memory savings. Finally, to address the dynamic attention sparsity patterns across requests and attention heads, we propose an adaptive memory management approach. Rather than imposing a fixed memory budget, our approach allows each attention head to determine its memory requirements dynamically based on its specific sparsity pattern.

Next, we describe the KV compression policy in greater detail, addressing the prompt phase and the generation phase separately. In both phases, compression is applied per-request and per-head, ensuring that memory usage is tailored to the specific sparsity pattern of each request and attention head.

Prompt Phase. In the prompt phase, key and value vectors for all tokens in the prompt are computed. The compression policy then determines the appropriate precision for storing each token based on its significance. The significance of the i^{th} token is calculated by averaging the $N - i$ attention scores it receives from subsequent tokens, where N denotes the prompt sequence length. In the case of GQA and MHA, scores from all attention heads associated with the KV head are aggregated using the maximum operation. To mitigate premature compression, the most recent W tokens are always quantized at high precision, where W is typically set to 64. For the remaining tokens, the policy determines the precision level of the i^{th} token in a sequence-length adaptive manner, by comparing its significance score to the theoretical average $\frac{1}{i}$, based on the intuition that tokens with scores below average are less critical. Specifically, the i^{th} token is quantized at high precision if its significance score exceeds $\frac{\alpha_h}{i}$, at low precision if its score lies within the interval $[\frac{\alpha_l}{i}, \frac{\alpha_h}{i}]$, and is pruned otherwise. The parameters α_l and α_h, which define the thresholds for high- and low-precision quantization, are determined offline through profiling on a calibration dataset. As a result of this hierarchical compression, the KV cache is conceptually divided into

two parts: a high-precision section KV_h and a low-precision section KV_l.

Generation Phase. In the generation phase, a single token is compressed at each step, aligning with the autoregressive nature of the generation process. The most recent token is added to the recent window to prevent premature compression, while the earliest token t_c in the window becomes a candidate for more aggressive compression. The compression procedure can be divided into two parts. First, token t_c is quantized at either a high or low precision and added to the corresponding section of the KV cache, or it may be completely pruned. Next, if t_c is quantized, the least significant token t_v in the corresponding precision section of the KV cache is considered for further downgrading: it may be re-quantized to lower precision or pruned, depending on its significance. Essentially, this policy establishes a smooth downgrading path for the less important token: rather than being pruned directly, it is first re-quantized to low precision, with pruning occurring only if it remains insignificant.

The detailed procedure is outlined in Algorithm 1, following the same intuition and parameters as the prompt phase. Given the sequence length N, t_c is quantized at high precision and added to KV_h, the high precision KV cache (line 6), if its significance exceeds $\frac{\alpha_h}{N}$. Subsequently, the least significant token in KV_h is identified as the victim token t_v for more aggressive compression (line 7). If t_v's significance exceeds $\frac{\alpha_h}{N}$, it remains in KV_h; if it falls within $[\frac{\alpha_l}{N}, \frac{\alpha_h}{N}]$, t_v is re-quantized to low precision and moved to KV_l, the low precision KV cache (line 9); otherwise, t_v is pruned. Similarly, if t_c's significance lies within $[\frac{\alpha_l}{N}, \frac{\alpha_h}{N}]$, it is quantized to low precision and added to KV_l (line 14). The least important token in KV_l is designated as the victim t_v (line 15), and is further pruned if its significance falls below $\frac{\alpha_l}{N}$ (line 17).

Discussion. The proposed KV compression policy is highly extensible, allowing for the potential use of more than two quantization precision levels. We adopt two quantization precision levels primarily to minimize metadata overhead and improve system efficiency, as will be detailed in the following section. In Section 7.2, we empirically show that the two-level scheme (K8V4-K4V2) achieves near-lossless generation quality across multiple models and benchmarks. We use a shared set of thresholds for all attention heads, as our empirical studies demonstrate that this approach is sufficient to capture the varying sparsity patterns across different heads. Notably, DiffKV can efficiently support more flexible compression policies, such as tuning thresholds for each head individually, which could potentially further optimize the balance between model accuracy and memory efficiency. We leave the exploration of such flexible policies for future work.

Algorithm 1: KV compression policy (generation)

1: **Input:** Parameters α_h, α_l; High & low precision P_h & P_l
2: **Input:** Candidate token t_c; Sequence length N
3: **Input:** High & low precision KV cache KV_h & KV_l
4: **Function:** Significance $Score$; Quantization $Quant$;
5: **if** Score(t_c) $\geq \frac{\alpha_h}{N}$ **then**
6: KV_h.add(Quant(t_c, P_h))
7: $t_v = argmin_{t \in KV_h}$(Score($t$))
8: **if** $\frac{\alpha_l}{N} \leq$ Score(t_v) $< \frac{\alpha_h}{N}$ **then**
9: KV_h.remove(t_v), KV_l.add(Quant(t_v, P_l))
10: **else if** Score(t_v) $< \frac{\alpha_l}{N}$ **then**
11: KV_h.remove(t_v)
12: **end if**
13: **else if** Score(t_c) $\geq \frac{\alpha_l}{N}$ **then**
14: KV_l.add(Quant(t_c, P_l))
15: $t_v = argmin_{t \in KV_l}$(Score($t$))
16: **if** Score (t_v) $< \frac{\alpha_l}{N}$ **then**
17: KV_l.remove(t_v)
18: **end if**
19: **end if**

5 MEMORY MANAGEMENT

In this section, we describe DiffKV's memory management mechanism which efficiently supports the three levels of differentiation in the KV cache. We first highlight the challenges posed by these differentiations in memory management, and then present parallel KV compaction, a novel memory management technique that effectively tackles the challenges.

5.1 Challenges

Flexible Paging. In PagedAttention [39], all tokens are stored at the same precision, allowing for a fixed page format. However, the differentiation of key and value vectors, along with varying token importance, introduces multiple precision levels both within and across tokens, making a fixed page format insufficient. Specifically, a fixed page format requires conservatively allocating high-precision slots for all tokens regardless of their actual precisions, leading to considerable memory wastage. Suppose memory is allocated to accommodate the highest precision K8V4, a token with K4V2 precision would waste 50% of the memory. Worse, such a fixed page format results in misaligned memory accesses, which hinder memory bandwidth utilization and reduce overall computational efficiency.

KV Compaction Scalability. At each LLM inference step, the memory management complexity is $O(\#requests)$ for PagedAttention, as memory is partitioned uniformly among attention heads. In contrast, DiffKV must handle varying numbers of high- and low-precision tokens across different heads, mapping these heterogeneous memory requirements to physical memory. We term this process *KV compaction*, whose complexity is $O(\#requests \times \#heads)$ per inference step. This poses significant scalability challenges for memory management. Given that model execution time per step is on the order of tens of milliseconds, improper memory management could render the overhead of dynamic allocation prohibitive, negating the benefits of KV cache compression. Moreover, managing both high-precision and low-precision tokens would require separate data structures for each precision, resulting in increased metadata overhead.

5.2 Parallel KV Compaction

A detailed analysis of the KV compaction process reveals opportunities for parallelization. KV compaction can be divided into two phases: *planning* and *coordination*. In the planning phase, each attention head independently determines its memory allocation requirements. The subsequent coordination phase synchronizes these per-head requirements and maps them to the GPU's physical memory. The primary source of increased memory management complexity lies in the planning phase, which is perfectly parallelizable and

well-suited to the GPU's parallel compute capabilities. Parallelizing the coordination phase is more challenging, as it requires synchronization across heads. However, we observe that this phase can be parallelized effectively via parallel prefix sum [52], provided that the free memory region is contiguous.

Thus, we propose *parallel KV compaction*, a novel management technique that efficiently performs per-head dynamic memory allocation and recycling in parallel directly on the GPU. Parallel KV compaction is enabled by three GPU-resident data structures collectively: the (1) *unified pages* to enable flexible paging, (2) the *circular free page list* to efficiently parallelize memory management, and (3) the *bidirectional page table* to minimize metadata overhead for tracking tokens of differentiated precisions.

Unified Pages. *Unified pages* abstract away the complexity of differentiated precisions both within individual tokens and across different tokens, thereby simplifying the implementation of parallel KV compaction. Specifically, we partition the available GPU memory into evenly sized pages, each configured *upon allocation* to store tokens at a given precision. Each unified page is organized into six segments: quantized keys, quantization metadata for keys, quantized values, quantization metadata for values, token scores, and positions. The quantization metadata includes scales and zero points for the key and value vectors. The number of tokens stored per page is adjusted according to the quantization configuration, ensuring compact memory usage. Furthermore, by consolidating keys, values, and their metadata into a single structure, unified pages enhance data locality and eliminate the need for scattered lookups, improving memory access efficiency during attention computation.

Circular Free Page List. The *circular free page list* serves as the cornerstone of parallel KV compaction, facilitating the parallelization of memory allocation and recycling by maintaining both free and used pages in contiguous regions.

This centralized, GPU-resident data structure contains all Page IDs and tracks free pages through a pair of pointers: a start pointer for allocations and an end pointer for recycling. These pointers wrap around to the beginning of the list upon reaching the end, forming a circular structure. When a page is allocated, the start pointer advances to the next available Page ID; when a page is freed, the end pointer advances to add the released Page ID back into the list. Both the available and unavailable regions in the list remain contiguous, enabling the coordination phase in memory management to be parallelized via parallel prefix sum.

In parallel KV compaction, after each head determines the number of pages to be allocated or freed, a parallel prefix sum operation computes a unique offset for each head relative to the start or end pointer. This step converts per-head memory demands into a globally consistent schedule, where each head is assigned a disjoint region of the free page list to read from (for allocation) or write to (for recycling), ensuring non-conflicting operations within the list. For memory allocation, each head concurrently retrieves its new page IDs from its designated region in the list, with the start pointer incremented by the cumulative number of pages required across all heads. Similarly, for memory recycling, each head writes freed page IDs to its designated region, with the end pointer incremented by the total number of pages released.

Bidirectional Page Table. Similar to PagedAttention, DiffKV maintains a page table for each attention head, mapping each request to its corresponding list of page IDs. To avoid the doubled metadata overhead associated with maintaining separate page tables for high- and low-precision pages, DiffKV introduces a unified, GPU-resident data structure called the *bidirectional page table*. This structure efficiently supports the use of two quantization precision levels for KV cache, as proposed in Section 4, minimizing the metadata overhead. In each entry of the bidirectional page table, high-precision page IDs grow from the left side of the list, while low-precision page IDs grow from the right, dynamically adapting to the precision requirements of the workload. The length of each page table entry is determined by the maximum sequence length divided by the tokens per high-precision page, ensuring no overflow since low-precision pages always contain more tokens than high-precision ones. This unified approach not only minimizes metadata overhead but also eliminates the need for separate lookups based on precision levels, thereby enhancing memory access efficiency during attention computations. The memory overhead of the bidirectional page table is minimal: for example, with a batch size of 128 on Llama3-8B, which has 32 layers and 8 KV heads per layer, the total size of all bidirectional page tables is only 32 MB. By contrast, the KV cache for a single request occupies 1 GB.

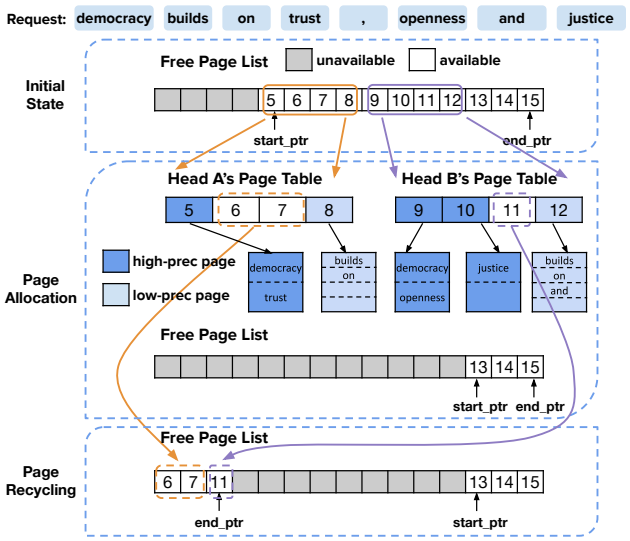


Figure 6: Memory management flow in the prompt phase.

5.3 KV Compaction Workflow

We delve into the KV compaction workflow during the prompt and generation phases, illustrating how the three data structures interact to enable differentiated KV cache compression. Notably, KV compaction is executed once per inference step for all requests in the batch and all layers in the LLM. This design ensures sufficient parallelism for efficient GPU execution, while amortizing the associated GPU kernel launch overheads.

Prompt Phase. Figure 6 illustrates the KV compaction workflow during the prompt phase for an example request with eight tokens. In this example, one high-precision page stores two tokens, and one low-precision page stores four. Since the exact numbers of high- and low-precision pages required by each head are unknown a priori, we conservatively allocate four unified pages per head, assuming

all tokens are stored at high precision. Pages 5-8 and 9-12 are thus assigned to Head A and B, respectively, as shown in their page tables. The end pointer of circular free page list advances to Page 13, marking Pages 5-12 as allocated.

Next comes the *planning phase*, during which each head independently applies the KV cache compression algorithm and calculates its specific memory requirements. In this example, Head A uses one high-precision page, while Head B uses two; both heads also require one low-precision page. We allocate high-precision pages left-to-right in the bidirectional page table and low-precision pages right-to-left: Head A uses Page 5 (high-precision) and Page 8 (low-precision), while Head B uses Pages 9–10 (high-precision) and Page 12 (low-precision).

Following the planning phase, the coordination phase finally reclaims unused pages (Pages 6-7 from Head A and Page 11 from Head B) via a parallel prefix-sum. These recycled pages are appended to the circular free page list; since the end pointer already points to the list's tail, it wraps around to the head to accommodate the recycled pages.

Generation Phase. At each generation step, a head allocates a new page only if either its high-precision or low-precision pages are full, requiring at most one additional page per step. Each head independently checks its page availability and, if needed, allocates a new page in parallel using the prefix-sum-based approach. Unlike in the prompt phase, page recycling is not performed during generation, as the total number of stored tokens either remains the same if an old token is evicted or increases by one if no eviction occurs. Once a request is finished, all pages allocated for that request are recycled, freeing up memory for incoming requests.

Supporting Additional Precision Levels. The proposed memory management system naturally extends to support more than two precision levels, by incorporating additional page tables. For example, accommodating three precision levels can be achieved by combining a unidirectional page table with a bidirectional one. Similarly, four precision levels can be supported using two bidirectional page tables. This composable design preserves memory efficiency while providing the flexibility needed to handle increasingly fine-grained precision differentiation.

6 IMPLEMENTATION

We implement DiffKV on top of vLLM, comprising 4.5K lines of CUDA/C++ code and 9K lines of Python. We first outline the architecture of DiffKV and then detail our custom GPU attention kernel, designed to efficiently support the differentiated KV cache compression.

6.1 DiffKV Architecture

The architecture of DiffKV is illustrated in Figure 7. At each inference step, the scheduler batches as many requests as possible within the available GPU memory to maximize throughput, sending the selected requests to all workers. Each GPU hosts one worker, responsible for executing a partition of the model [33, 53, 54, 64]. To facilitate efficient execution, each worker includes a dedicated memory manager that oversees the KV cache for its assigned attention heads, as outlined in Section 5, and an execution engine for model computation. To support differentiated KV cache compression, the execution engine integrates a KV compressor and a custom GPU attention kernel. After computing key and value vectors, the KV compressor is invoked to compress them following the policy described in Section 4, storing the results in the KV cache. The custom GPU attention kernel then efficiently computes the attention output using the compressed KV cache.

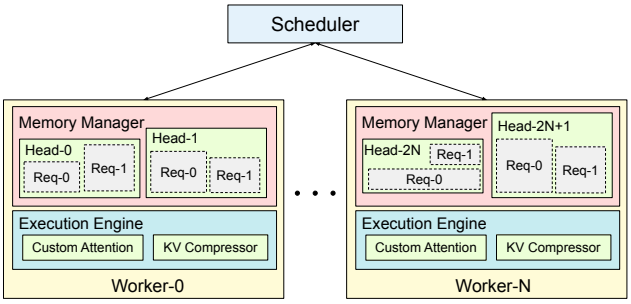


Figure 7: DiffKV architecture.

6.2 Efficient Attention Kernel

We develop a custom attention kernel that efficiently supports differentiated KV cache compression, leveraging reduced memory access volume to accelerate performance. Notably, the kernel design is portable across GPUs and other NPUs, as mainstream architectures support vectorized memory access and layout-aware tiling [37, 80]. We detail the GPU implementation in the following paragraphs.

Overall, we assign each thread block to process a single attention head per sequence. To mitigate the potential load imbalance caused by mixed-precision quantization, we let thread warps iterate over high-precision pages first, followed by low-precision pages. Each page is processed in two phases: dot product between the query and keys to derive attention scores, and weighted sum of values. These two phases follow different computation patterns: reduction over the feature dimension and token dimension, respectively. To ensure coalesced and vectorized memory accesses for both phases, we design tailored data layouts and parallelization strategies, elaborated as follows.

For key processing, each warp handles one page at a time, with threads divided into groups responsible for distinct key vectors. Within a group, each thread fetches its assigned key elements in a vectorized manner, performs dequantization on the fly, and computes a partial dot product between the key elements and corresponding query elements. Using a straightforward layout such as $[F, N_{\text{tokens}}]$ for keys, where F denotes the feature dimension length, would cause threads within a group to fetch non-contiguous elements strided by N_{tokens} when parallelizing across the feature dimension. Alternatively, using $[N_{\text{tokens}}, F]$ would cause thread groups to fetch non-contiguous elements strided by F when parallelizing across tokens. Both layouts suffer inefficient strided memory accesses, leading to low bandwidth utilization. Instead, we organize the layout of keys as $[\frac{F}{K_{\text{vec}} \times K_{\text{group}}}, N_{\text{tokens}}, K_{\text{group}}, K_{\text{vec}}]$, where K_{vec} denotes the vectorization factor and K_{group} denotes the number of threads per group. During each execution step, each thread fetches K_{vec} consecutive elements, threads within a group collectively fetch K_{group} adjacent chunks, and groups within the warp access contiguous chunks along the N_{tokens} dimension. As a result, the combined memory accesses

of all threads in a warp are contiguous, enabling memory coalescing and maximizing bandwidth utilization.

For value processing, tokens in a page are evenly distributed across thread groups in a warp; within each group, individual threads perform sum reductions over the feature dimension and save their accumulation results in registers. Once all groups finish, a tree reduction aggregates these partial results and produces the output. The number of registers required per thread is proportional to the feature dimension range assigned to it, while during key processing each thread only requires a single register to store the partial dot product. Consequently, vectorization along the feature dimension, as used in key processing, is not suitable for value processing, because it would significantly increase register pressure by forcing each thread to handle a larger portion of the feature dimension. To better align with the computation pattern of reduction across tokens, we apply vectorization to the token dimension instead, which reduces register pressure and ensures more effective parallelization. We organize the layout of values as $[\ \frac{F}{V_{\text{group}}}, \frac{N_{\text{tokens}}}{V_{\text{vec}}}, V_{\text{group}}, V_{\text{vec}}]$ accordingly.

Additionally, for ultra-long sequences, the attention kernel supports parallelization along the sequence dimension. It splits the sequence into multiple segments, processes them in parallel using the aforementioned method within separate thread blocks, and then merges the results with minimal computation overhead.

7 EVALUATION

We first evaluate the effectiveness of DiffKV's differentiated KV cache compression policy. Next, we evaluate the efficiency of the memory manager as well as the end-to-end throughput improvements achieved by DiffKV.

7.1 Experiment Setup

We evaluate DiffKV on several models spanning three major families: Llama3-8B and 70B [69], Qwen2.5-7B and 32B [34], and the recent thinking models QwQ-32B [68], R1-Distill-Qwen-14B and R1-Distill-Llama-8B [26], which generate extended chains of thought and exhibit strong capabilities on complex reasoning tasks.

We select benchmarks that align with those commonly used in recent LLM technical reports [20, 27, 34, 46, 68], covering a diverse set of task domains, including general knowledge (MMLU [31] and MMLU-Pro [65]), mathematics (GSM8K [15] and MATH [41]), code generation (HumanEval+ [12, 47] and MBPP+ [8, 47]), and long-context understanding (LongBench [9]). In addition, we assess DiffKV's generation quality on thinking models using two particularly challenging benchmarks: AIME24 [3], a mathematics competition dataset, and GPQA [58], which evaluates graduate-level science reasoning. We measure the throughput and latency mostly on NVIDIA L40 GPUs, each with 48 GB of memory [55]. We further evaluate DiffKV atop Ascend NPU [44] to demonstrate the portability of our approaches. In all experiments, model weights are stored in FP16 precision.

7.2 Differentiated KV Compression Policy

We first assess the effectiveness of differentiated KV quantization and dynamic sparsity individually, and then evaluate the combined benefits of the proposed differentiated KV cache compression. To ensure the reliability of our results and mitigate the influence of noise and randomness inherent in floating-point computations, each experiment is repeated five times with the dataset randomly shuffled, and the reported results represent the average across these runs.

Evaluating Differentiated KV Quantization. Figure 8 shows the model generation quality of differentiated KV quantization on GSM8K and HumanEval+, namely K8V4 and K4V2, compared to FP16 baselines, across Llama3-8B & 70B and Qwen2.5-7B. To validate our intuition that keys have a more significant impact than values (Section 3.1), we additionally evaluate the mirror configurations, namely K4V8 and K2V4, where values are stored with higher precision than keys, as well as more skewed variants, K8V2 and K4V1.

Results show that K8V4 matches the accuracy of the FP16 baseline across all models and benchmarks. In contrast, its mirror configuration, K4V8, exhibits noticeable accuracy degradation, particularly for Qwen2.5-7B, where accuracy drops to nearly zero on both tasks. This pronounced sensitivity to 4-bit key quantization in Qwen2.5-7B is likely due to its GQA architecture, which applies aggressive KV compression with a queries-per-KV ratio of 7, substantially higher than the ratio of 4 used in Llama3-8B. Similarly, K4V2 retains over 65% of FP16 accuracy across both benchmarks on Llama3-8B and 70B, while the mirror configuration K2V4 results in near-zero accuracy. The more skewed variant K8V2 retains at least 83% of FP16 accuracy across all models, even for Qwen2.5-7B, reinforcing the importance of assigning higher precision to keys. On the other hand, K4V1 yields almost zero accuracy, suggesting that 2 bits is the lower bound for effective value quantization. In summary, these results confirm that keys play a more critical role than values, and demonstrate the effectiveness of differentiated KV quantization in preserving accuracy while enabling aggressive compression.

The accuracy trends across various differentiated KV quantization configurations motivate the design of adopting two precision levels: K8V4 and K4V2. K8V4 matches the generation quality of the FP16 baseline and is applied to important tokens to preserve model accuracy. K8V2 and K4V2, when applied uniformly, both lead to noticeable accuracy degradation. Among them, K4V2 offers better efficiency and is thus selected for compressing less significant tokens, enabling opportunistic KV cache savings with minimal quality loss. As we demonstrate later, the K8V4–K4V2 scheme achieves near-lossless generation quality across multiple models and benchmarks, obviating the need for an intermediate precision level like K8V2. Likewise, introducing an even lower precision level such as K4V1 is suboptimal: not only does K4V1 significantly degrade accuracy, but the additional memory savings from a three-level configuration K8V4–K4V2–K4V1 are marginal compared to the K8V4–K4V2 design. That said, future models employing more aggressive training-time KV compression (e.g., architectures with higher queries-per-KV ratios in GQA) may benefit from an additional high-precision level (e.g., FP16–K8V4 or FP16–K8V4–K4V2), which we leave to future investigation.

Evaluating Dynamic Sparsity. We evaluate the effectiveness of per-head dynamic sparsity (Section 3.3) in identifying critical tokens, comparing it to the static sparsity method used in SnapKV [42], which allocates an equal memory budget to all attention heads. Figure 9 reports results on Qwen2.5-7B and Llama3-8B across GSM8K and HumanEval+. The x-axis indicates the percentage of tokens pruned, while the y-axis shows the resulting task accuracy. Dynamic

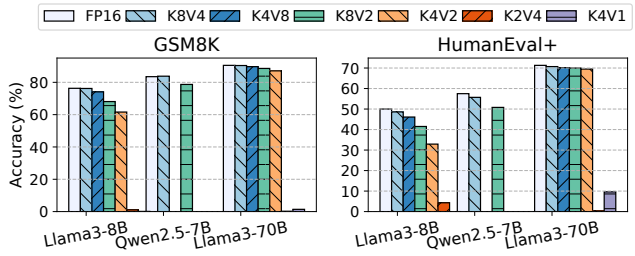


Figure 8: Accuracy of differentiated KV quantization.

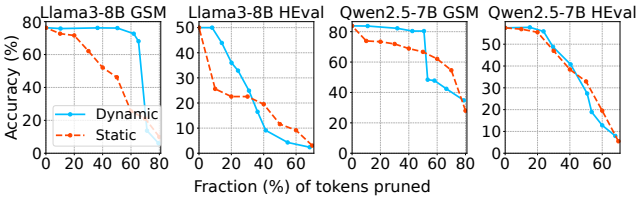


Figure 9: Accuracy of dynamic vs. static sparsity.

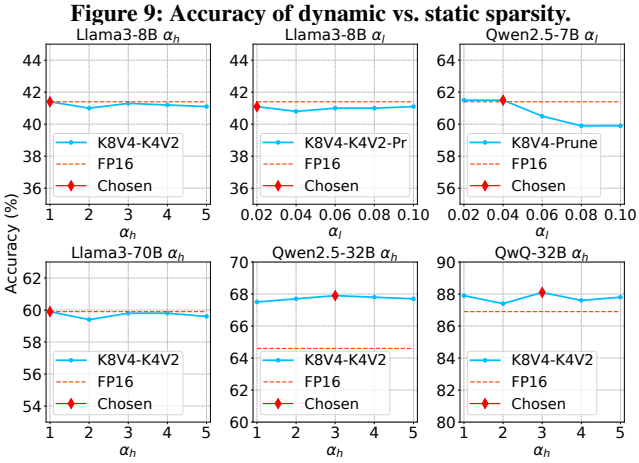


Figure 10: Calibrating the high- and low-precision thresholds α_h and α_l on the training split of the MATH dataset.

sparsity significantly outperforms static sparsity in Llama3-8B, maintaining full accuracy with 50% tokens pruned on GSM8K, and 10% tokens pruned for the more sensitive HumanEval+. For Qwen2.5-7B, dynamic sparsity also consistently surpasses static sparsity, achieving accuracy higher than 80% on GSM8K with 50% tokens pruned. While the improvement is less pronounced on HumanEval+, dynamic sparsity still yields higher accuracy at equivalent pruning ratios. In summary, per-head dynamic sparsity is superior to static sparsity by leveraging head-specific significance.

Parameter Calibration. As described in Section 4, DiffKV compares the significance score of a token with its theoretical average, $\frac{1}{N}$, where N denotes the sequence length. A token is quantized to high precision if its score exceeds $\frac{\alpha_h}{N}$, to low precision if its score falls within the interval $[\frac{\alpha_l}{N}, \frac{\alpha_h}{N}]$, and is pruned otherwise. To calibrate the appropriate parameters α_h and α_l, we utilize the training split of MATH [41]. We select MATH as the calibration dataset as it is comprised of complex mathematical reasoning tasks with high text information density, where even the removal of a single symbol can invalidate an argument. This makes MATH highly suitable for parameter fitting, as the resulting compression parameters are

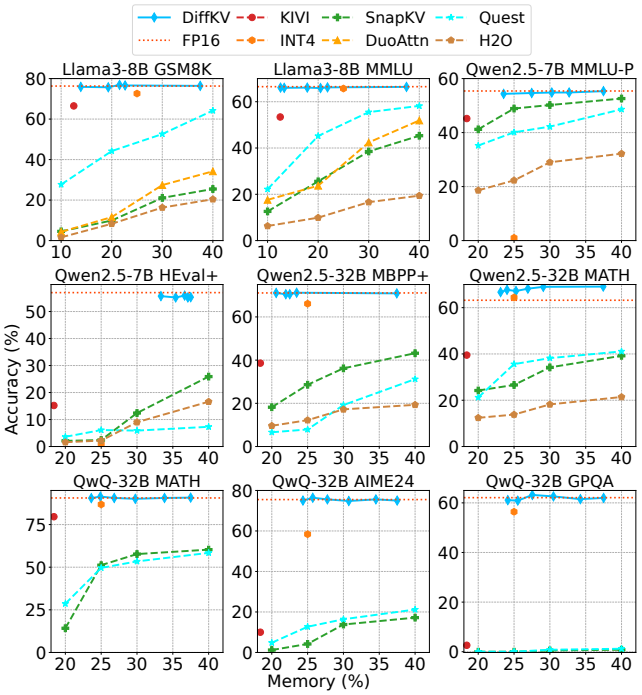


Figure 11: KV cache memory normalized to vLLM vs. benchmark accuracy tradeoff of DiffKV.

expected to generalize well across other tasks. Additionally, MATH provides a dedicated training split, ensuring that parameter fitting does not involve testing data and mitigating potential overfitting.

We profile a consistent range of parameters across all evaluated models, as depicted in Figure 10. The x-axis represents the profiled parameter values, while the y-axis shows the resulting accuracy on the calibration dataset. Specifically, we profile α_h within the integer range $[1, 5]$, progressively relaxing the high-precision threshold beyond the theoretical average. This adjustment accounts for aggregating scores from multiple attention heads by maximum for GQA, as mentioned in Section 4. The profiled values of α_h achieve FP16-equivalent accuracy across all models, except for Qwen2.5-7B, which exhibits heightened sensitivity to 4-bit key quantization. Accordingly, we select the values of α_h that yield the highest accuracy based on profiling results: $\alpha_h = 1$ for Llama3-8B and 70B, and $\alpha_h = 3$ for Qwen2.5-32B and QwQ-32B, and disable low-precision quantization for Qwen2.5-7B. Subsequently, we profile the low-precision threshold α_l, while keeping α_h fixed. We observe that setting $\alpha_l = 0.1$ induces at least 5% accuracy degradation on the larger models. As a result, we profile five equally spaced values within the range $[0, 0.1]$ and select the one yielding the highest accuracy for each model. The final selected values of α_l are 0.02 for Llama3-8B, 0.04 for Qwen2.5-7B, and 0 for the remaining models. While the profiled values cover a limited portion of the entire parameter space, as will be demonstrated in the following sections, DiffKV significantly outperforms existing baselines with these fitted parameters, highlighting its potential effectiveness.

Evaluating Differentiated Compression Policy. Finally, we evaluate the combined benefits of DiffKV's KV cache compression policy.

We compare DiffKV against several state-of-the-art baselines, including pruning-based methods such as H2O [77], SnapKV [42], and DuoAttention [71], quantization-based approaches such as 4-bit KV, KIVI [49], and QAQ [19], as well as KV partial loading techniques like Quest [66]. Table 1 compares the accuracy and KV cache memory usage of DiffKV and the best-performing baselines, both normalized to vLLM [39], for the evaluated non-thinking models. The parameters of DiffKV are tuned as described previously. For the pruning-based baselines and Quest, we tune their memory usage to 50% of the baseline. DiffKV achieves near-lossless accuracy relative to the FP16 baseline, with an average degradation of only 0.3%, while using 19.3% to 36.7% memory. In contrast, all other baselines incur more significant accuracy degradation at comparable or higher memory usage. Furthermore, DiffKV dynamically adapts its memory usage based on the task's information density. For example, DiffKV allocates less memory for the 5-shot MMLU benchmark compared to the 0-shot HumanEval+, where the prompt consists solely of the function name and comments.

DiffKV also outperforms baseline methods in long-context scenarios, as shown in Table 2, where we present results from one benchmark in each category of LongBench. For Llama3-8B and Qwen2.5-7B, DiffKV achieves memory usage of 14.9% and 27.0%, respectively, while baseline methods are tuned to 25% memory usage. Notably, DiffKV consistently achieves superior accuracy across all categories, with an average degradation of only 0.4%, while utilizing less or comparable memory compared to the baseline methods.

Furthermore, the generation quality gap between DiffKV and baseline methods becomes even more pronounced for the thinking models QwQ-32B, R1-Distill-Qwen-14B and R1-Distill-Llama-8B on the more challenging benchmarks, as demonstrated in Table 3. DiffKV achieves FP16-comparable generation quality using 27.4%, 29.4% and 23.5% of the memory on average for QwQ-32B, R1-Distill-Qwen-14B and R1-Distill-Llama-8B, respectively, while all other methods experience significant generation quality degradation. On the math competition benchmark AIME24, the most accurate baseline, 4-bit KV quantization incurs a 16.0% accuracy degradation. On the graduate-level science benchmark GPQA, the accuracy of pruning-based and KV partial loading methods drops to almost zero, while using 50% of the full KV cache. The larger gap between DiffKV and prior work in long CoT generation highlights the increased challenges such tasks pose for KV cache compression. In long CoT generation, tokens are generated autoregressively based on previous tokens, and errors introduced by compression are accumulated and propagated across generation steps. In contrast, in long prompt scenarios, the majority of the text is provided as ground truth in the prompt, and the model-generated portion is significantly shorter, limiting the impact of errors introduced by KV cache compression.

Lastly, Figure 11 depicts the accuracy-memory trade-offs of DiffKV under the profiled parameter values, namely $\alpha_h \in [0, 5]$ (step size of 1) for all models and $\alpha_l \in [0.02, 0.1]$ (step size of 0.02) for Llama3-8B and Qwen2.5-7B. We also include the accuracy of baseline methods within similar memory usage ranges for comparison. DiffKV consistently matches the FP16 baseline accuracy across different models and benchmarks within the profiled range of memory usage, demonstrating its robustness. In contrast, baseline methods experience significant accuracy degradation, particularly pruning-based and KV partial loading methods, which suffer rapid accuracy declines as memory usage decreases. In summary, DiffKV provides a superior accuracy-memory tradeoff compared to state-of-the-art methods.

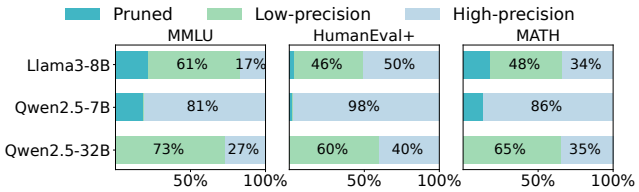


Figure 12: Fraction of tokens pruned, quantized to low precision, and quantized to high precision across different benchmarks and models.

KV Compression Breakdown. Figure 12 shows the fraction of tokens that are pruned, quantized to low precision, and quantized to high precision across three benchmarks (MMLU, HumanEval+, and MATH) for three different models. Two key observations emerge. First, sparsity levels vary substantially across workloads. The general knowledge benchmark MMLU exhibits the highest sparsity, likely due to the lower information density of natural language and the use of 5-shot prompts. In contrast, the reasoning-intensive benchmarks MATH and HumanEval+ display lower sparsity, with HumanEval+ being the least sparse, partly due to its 0-shot setting. Second, despite being calibrated on challenging reasoning-heavy workloads, DiffKV's KV compression policy adapts effectively to the varying sparsity patterns across different tasks, highlighting its generalizability to diverse scenarios.

7.3 System Performance

In this section, we provide a comprehensive evaluation of DiffKV's performance, comparing it against state-of-the-art systems including vLLM, Quest, and SnapKV across various models. For models that fit within the memory of a single GPU, such as Llama3-8B and Qwen2.5-7B, we use one GPU. For larger models, we parallelize the computation across multiple GPUs: four GPUs for Llama3-70B, and two GPUs for Qwen2.5-32B and QwQ-32B. Our evaluation focuses on latency breakdown, attention kernel speedup, and end-to-end throughput to demonstrate the efficiency and scalability of DiffKV.

Memory Management Overhead. Figure 13 compares DiffKV's on-GPU parallel KV compaction with an alternative implementation that performs multi-threaded memory management on the CPU. Evaluated with a sequence length of 1024 tokens across different batch sizes, DiffKV reduces memory management latency by up to three orders of magnitude relative to the on-CPU approach. For one entire inference step, DiffKV significantly outperforms the on-CPU approach, particularly in the generation phase, where the memory management overhead on the CPU far exceeds the model execution time. Therefore, DiffKV's on-GPU parallel memory management is essential for ensuring that the performance benefits of KV cache compression are fully realized without being overshadowed by memory management overhead.

Figure 14 further illustrates the latency breakdown of DiffKV during one inference step. The memory management overhead is remarkably low thanks to DiffKV's on-GPU parallel KV compaction, contributing less than 0.2% of the total latency in the prompt phase and under 0.9% in the generation phase. Model execution dominates

Table 1: Accuracy and memory usage of DiffKV and the best-performing baseline methods across models and benchmarks. Numbers in parentheses represent memory usage normalized to FP16. The memory usage is 50.0% for SnapKV, Quest and DuoAttn, 25.0% for INT4, 18.8% for QAQ, and 12.5% for KIVI. DiffKV achieves accuracy close to FP16, consistently outperforming prior quantization and pruning approaches.

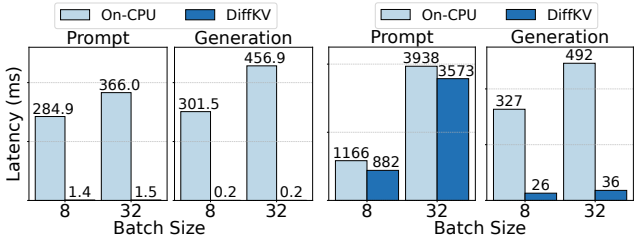
Benchmarks	Llama3-8B					Qwen2.5-7B					Qwen2.5-32B					Llama3-70B		
	FP16	DiffKV	INT4	QAQ	DuoAttn	FP16	DiffKV	Quest	SnapKV	KIVI	FP16	DiffKV	Quest	INT4	KIVI	FP16	DiffKV	INT4
GSM8K	76.3	75.7 (19.3%)	72.6	71.7	32.2	83.5	83.6 (26.8%)	76.7	66.7	72.2	90.4	90.2 (23.6%)	82.8	88.7	79.3	90.5	90.2 (21.6%)	89.3
MATH	28.1	27.9 (21.6%)	23.9	23.5	9.3	58.0	57.7 (32.3%)	41.7	45.1	39.5	63.2	67.2 (25.3%)	44.3	64.7	42.6	48.7	47.6 (21.3%)	45.3
MMLU	66.5	66.1 (17.8%)	65.2	60.9	42.4	75.1	74.7 (30.3%)	64.7	48.2	53.1	83.8	83.8 (23.1%)	68.5	83.4	70.5	81.0	80.9 (20.1%)	79.4
MMLU-Pro	41.5	41.0 (21.5%)	39.1	38.2	34.6	55.4	54.9 (30.2%)	51.6	49.7	45.2	67.8	67.4 (24.5%)	59.5	66.2	54.5	60.1	60.0 (21.5%)	59.2
HumanEval+	50.0	48.0 (27.6%)	45.1	7.9	4.8	57.5	55.9 (36.7%)	8.5	32.9	15.2	49.4	49.4 (26.2%)	11.0	45.8	15.3	71.3	71.5 (26.7%)	70.1
MBPP+	59.3	61.6 (17.6%)	58.8	56.3	35.6	64.3	62.8 (27.9%)	33.3	44.7	18.3	71.1	70.5 (21.8%)	35.2	67.3	38.6	68.6	69.7 (22.0%)	67.2

Table 2: Evaluating DiffKV on LongBench. DiffKV uses 14.9% memory for Llama3.1-8B and 27.0% for Qwen2.5-7B. For Quest and SnapKV, the memory usage is 25%.

		Qasper	HotpotQA	GovReport	TREC	PCount	Lcc
Llama3.1-8B	FP16	40.9	61.3	34.0	73.0	6.9	62.2
	DiffKV	42.8	61.3	32.9	73.0	7.2	61.9
	Quest	38.9	59.8	30.6	66.5	6.4	56.8
	SnapKV	39.2	59.6	30.1	65.9	6.2	57.2
Qwen2.5-7B	FP16	26.5	27.8	33.4	71.0	5.7	61.9
	DiffKV	26.4	28.2	32.2	70.0	5.3	62.3
	Quest	23.6	25.2	31.4	65.6	4.3	53.3
	SnapKV	22.6	25.5	30.8	64.6	4.5	52.6

Table 3: Evaluating DiffKV on thinking models.

		FP16	DiffKV	INT4	KIVI	Quest	SnapKV
QwQ-32B	MATH	90.6	90.6 (26.8%)	87.3	80.6	63.3	65.4
	GPQA	62.1	63.2 (27.6%)	57.8	2.6	1.2	0.8
	AIME24	75.5	75.3 (27.8%)	64.3	10.0	23.3	19.2
R1-Distill-Qwen-14B	MATH	94.2	94.1 (29.5%)	92.7	69.2	68.4	62.6
	GPQA	55.7	55.6 (29.2%)	52.0	0.0	0.5	0.0
	AIME24	67.0	67.0 (29.6%)	61.6	10.0	20.2	17.2
R1-Distill-Llama-8B	MATH	88.8	88.7 (23.2%)	86.1	48.2	58.9	54.2
	GPQA	47.4	46.5 (23.5%)	42.9	0.0	0.5	0.5
	AIME24	51.0	51.0 (23.8%)	42.3	6.6	18.2	15.4



(a) Memory management (b) One entire inference step

Figure 13: Latency comparison between parallel KV compaction and on-CPU multi-threaded memory management.

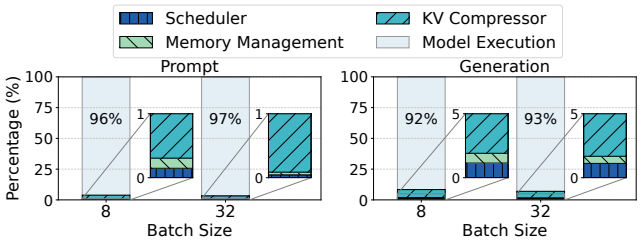
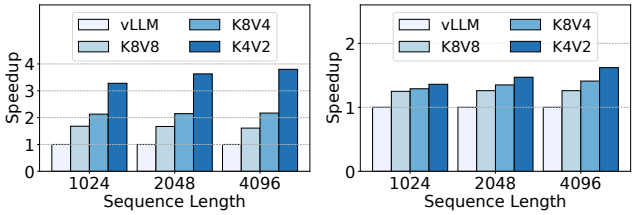


Figure 14: Latency breakdown of DiffKV.



(a) Attention kernel (b) One batch end-to-end

Figure 15: Latency speedup of DiffKV against vLLM.

the latency in both phases, accounting for 96–97% of the latency in the prompt phase and 92–93% in the generation phase. Notably, as the batch size increases, the percentage of time spent on model execution rises slightly, reflecting the scalability of DiffKV.

Latency Speedup. Figure 15a shows the speedup of DiffKV's custom attention kernel against vLLM under different quantization configurations. DiffKV achieves a near-linear speedup proportional to the reduction in KV cache size. For example, with K8V8, which halves the KV cache size relative to FP16 datatype, the theoretical speedup is 2×, and DiffKV achieves 1.7×. The slight gap is primarily due to the overhead of accessing quantization metadata and performing dequantization. Additionally, DiffKV achieves greater speedups on longer sequences, indicating that its bandwidth optimization techniques become increasingly effective as the sequence length grows. Figure 15b shows the end-to-end inference latency for a batch of 8 sequences. With sequence length 4096, DiffKV achieves a 1.4–1.6× speedup over vLLM. Note that while DiffKV does improve latency, its primary design focus is on enhancing throughput by supporting larger batch sizes.

Throughput Speedup. Figure 17 presents the end-to-end throughput and achieved batch sizes of DiffKV compared to vLLM, pruning-based approaches including Quest and SnapKV, as well as quantization-based methods including Atom (4-bit) and KIVI (2-bit). Note that these pruning- and quantization-based baselines incur larger accuracy degradation than DiffKV, as shown in Table 1. The maximum generation length is set to 16K tokens for QwQ-32B, 8K for Qwen2.5-32B, and 4K for the other models. We evaluate on 1000 sequences sampled from the MATH dataset [41], which elicits chain-of-thought reasoning and typically leads to long generations reaching the specified limit. Compression thresholds for DiffKV are adopted from Figure 10. Across all models, DiffKV consistently achieves higher throughput than prior systems. Notably, DiffKV achieves a remarkable 5.4× higher throughput over vLLM on the QwQ-32B thinking model, while Quest, SnapKV, Atom, and KIVI

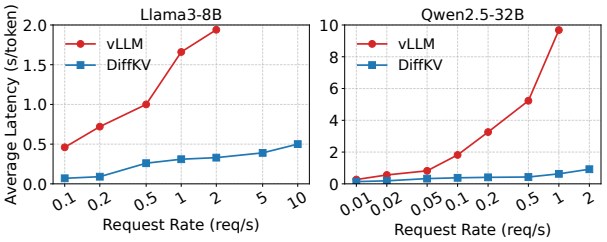


Figure 16: Comparison of average latency between DiffKV and vLLM under dynamic workloads.

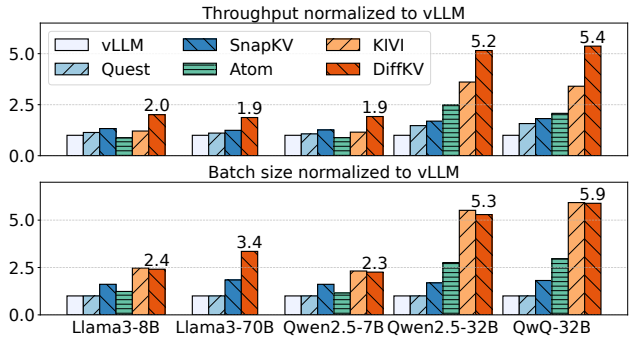


Figure 17: Throughput and achieved batch size.

achieve 1.6×, 1.8×, 2.1×, and 3.4× speedups, respectively. This result highlights the superior efficiency of DiffKV in processing long sequences, making it particularly well-suited for tasks that require extended reasoning.

The throughput improvements are directly correlated with the larger batch sizes supported by DiffKV, enabled by its KV cache compression techniques. For instance, with the QwQ-32B model, DiffKV sustains a batch size of 15.9, far exceeding vLLM's batch size of 2.7. The increased batch size enables DiffKV to maximize the utilization of GPU compute capacity, leading to higher throughput. In contrast, Quest supports the same batch size as vLLM because it retains the entire KV cache without reducing memory usage. Its speedup comes primarily from faster attention computation, as it processes only a subset of tokens deemed important. However, the process of estimating token importance incurs additional overhead, limiting the overall efficiency of Quest. Note that although Atom and KIVI substantially increase the batch size, approaching or even exceeding DiffKV, their throughput gains are not proportional. This is because both methods are built on the HuggingFace Transformers library, which incurs higher framework overhead than vLLM and DiffKV, and lacks high-performance GPU kernels that fuse dequantization with attention computation, as implemented in DiffKV. These results underscore that custom GPU kernels and an efficient runtime are essential for effectively translating KV cache compression into throughput gains.

Serving Dynamic Workloads. We evaluate DiffKV under dynamic workloads and compare it to vLLM [39]. Following the methodology of vLLM, we inject requests according to a Poisson arrival process and sweep the request rate. Figure 16 reports the average per-token latency (including both queuing and LLM processing time) for Llama3-8B and Qwen2.5-32B across a range of request rates. DiffKV consistently achieves lower latency and sustains higher loads before queuing delays grow sharply. These improvements stem from DiffKV's reduced KV cache footprint, which enables larger batch sizes, and its parallel on-GPU KV compaction, which keeps memory management overhead low even under bursty traffic.

Performance Speedup on Ascend NPU. Additionally, we ported DiffKV to Ascend NPU [80], integrating it into an in-house inference framework. On the Llama3-8B model, DiffKV increases the batch size by 2.3× over the baseline and improves end-to-end throughput by 1.7×. On the QwQ-32B model, it increases the batch size by 3.9× and throughput by 3.8×.

8 CONCLUSION

We proposed DiffKV, which enhances LLM serving efficiency by exploiting three levels of differentiation in KV cache: differentiated precision for keys and values, hierarchical compression based on token importance, and per-head dynamic sparsity. At the core of DiffKV is the parallel KV compaction technique that efficiently handles irregular memory requirements across requests and attention heads, effectively translating memory savings into performance gains. Our evaluation demonstrated that DiffKV is able to compress the KV cache by 2.7× to 5.7× with near-lossless accuracy, even on thinking models and complex workloads that require sophisticated reasoning and long-generation capabilities, and enhances throughput by 1.9× to 5.4×, outperforming prior KV cache compression methods.

9 ACKNOWLEDGMENT

We sincerely thank our shepherd, Tim Harris, for his thoughtful feedback and careful guidance throughout the paper shepherding process. We are also thankful to Mingzhe Hao, Shuang Chen, other friends, and the anonymous reviewers for their insightful comments on earlier versions of this manuscript.

REFERENCES

[1] FasterTransformer. https://github.com/NVIDIA/FasterTransformer.
[2] FlashInfer: Kernel Library for LLM Serving. https://github.com/flashinfer-ai/flashinfer.
[3] Aime 2024. https://huggingface.co/datasets/AI-MO/aimo-validation-aime, 2024.
[4] Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
[5] Adnan, M., Arunkumar, A., Jain, G., Nair, P. J., Soloveychik, I., and Kamath, P. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems 6* (2024), 114–127.
[6] Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245* (2023).
[7] Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis* (2022), IEEE, pp. 1–15.
[8] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
[9] 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. *arXiv preprint arXiv:2308.14508* (2023).
[10] Brown, T. B. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[11] Cai, Z., Zhang, Y., Gao, B., Liu, T., Lu, K., Xiong, W., Dong, Y., Chang, B., Hu, J., and Xiao, W. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069* (2024).

[12] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[13] Chiang, W.-L., Li, Z., Lin, Z., Sheng, Y., Wu, Z., Zhang, H., Zheng, L., Zhuang, S., Zhuang, Y., Gonzalez, J. E., Stoica, I., and Xing, E. P. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023.

[14] Chiang, W.-L., Zheng, L., Sheng, Y., Angelopoulos, A. N., Li, T., Li, D., Zhang, H., Zhu, B., Jordan, M., Gonzalez, J. E., et al. Chatbot arena: An open platform for evaluating llms by human preference. *arXiv preprint arXiv:2403.04132* (2024).

[15] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).

[16] Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).

[17] Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems 35* (2022), 16344–16359.

[18] Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems 35* (2022), 30318–30332.

[19] Dong, S., Cheng, W., Qin, J., and Wang, W. Qaq: Quality adaptive quantization for llm kv cache. *arXiv preprint arXiv:2403.04643* (2024).

[20] Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[21] Fang, J., Yu, Y., Zhao, C., and Zhou, J. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2021), pp. 389–402.

[22] Gao, P., Yu, L., Wu, Y., and Li, J. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–15.

[23] Ge, S., Zhang, Y., Liu, L., Zhang, M., Han, J., and Gao, J. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801* (2023).

[24] Github. Github copilot. https://github.com/features/copilot, 2023.

[25] Guo, C., Tang, J., Hu, W., Leng, J., Zhang, C., Yang, F., Liu, Y., Guo, M., and Zhu, Y. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (2023), pp. 1–15.

[26] 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* (2025).

[27] Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y., et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[28] Han, M., Zhang, H., Chen, R., and Chen, H. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (2022), pp. 539–558.

[29] Harris, M., Sengupta, S., and Owens, J. D. Parallel prefix sum (scan) with cuda. *GPU gems 3*, 39 (2007), 851–876.

[30] 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* (2024), 68287–68307.

[31] Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., and Steinhardt, J. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300* (2020).

[32] 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. *arXiv preprint arXiv:2401.18079* (2024).

[33] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems 32* (2019).

[34] Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[35] Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., Carney, A., et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720* (2024).

[36] Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. d. l., Hanna, E. B., Bressand, F., et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[37] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture* (2017), pp. 1–12.

[38] Kao, S.-C., Subramanian, S., Agrawal, G., Yazdanbakhsh, A., and Krishna, T. Flat: An optimized dataflow for mitigating attention bottlenecks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2023), pp. 295–310.

[39] 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* (2023), pp. 611–626.

[40] Lee, W., Lee, J., Seo, J., and Sim, J. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (2024), pp. 155–172.

[41] Lewkowycz, A., Andreassen, A., Dohan, D., Dyer, E., Michalewski, H., Ramasesh, V., Slone, A., Anil, C., Schlag, I., Gutman-Solo, T., et al. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems 35* (2022), 3843–3857.

[42] 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. *arXiv preprint arXiv:2404.14469* (2024).

[43] Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J. E., et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (2023), pp. 663–679.

[44] Liao, H., Tu, J., Xia, J., and Zhou, X. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)* (2019), IEEE Computer Society, pp. 1–44.

[45] Lin, Y., Tang, H., Yang, S., Zhang, Z., Xiao, G., Gan, C., and Han, S. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *arXiv preprint arXiv:2405.04532* (2024).

[46] 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* (2024).

[47] Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems* (2023).

[48] 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* (2024).

[49] 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* (2024).

[50] Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[51] Nagel, M., Fournarakis, M., Amjad, R. A., Bondarenko, Y., Van Baalen, M., and Blankevoort, T. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295* (2021).

[52] Nakano, K. An optimal parallel prefix-sums algorithm on the memory machine models for gpus. In *International Conference on Algorithms and Architectures for Parallel Processing* (2012), Springer, pp. 99–113.

[53] Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles* (2019), pp. 1–15.

[54] Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the international conference for high performance computing, networking, storage and analysis* (2021), pp. 1–15.

[55] Nvidia. Nvidia L40 datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/support-guide/NVIDIA-L40-Datasheet-January-2023.pdf, 2023.

[56] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems 35* (2022), 27730–27744.

[57] Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems 5* (2023), 606–624.

[58] Rein, D., Hou, B. L., Stickland, A. C., Petty, J., Pang, R. Y., Dirani, J., Michael, J., and Bowman, S. R. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling* (2024).

[59] Saab, K., Tu, T., Weng, W.-H., Tanno, R., Stutz, D., Wulczyn, E., Zhang, F., Strother, T., Park, C., Vedadi, E., et al. Capabilities of gemini models in medicine. *arXiv preprint arXiv:2404.18416* (2024).

[60] Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., Li, Y., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).

[61] Shazeer, N. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150* (2019).

[62] Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning* (2023), PMLR, pp. 31094–31116.

[63] Shi, Y., Yang, Z., Xue, J., Ma, L., Xia, Y., Miao, Z., Guo, Y., Yang, F., and Zhou, L. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (2023), pp. 701–718.

[64] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[65] Taghanaki, S. A., Khani, A., and Khasahmadi, A. Mmlu-pro+: Evaluating higher-order reasoning and shortcut learning in llms. *arXiv preprint arXiv:2409.02257* (2024).

[66] 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).

[67] Team, G., Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[68] The Qwen Team. Qwq-32b: Embracing the power of reinforcement learning, March 2025.

[69] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[70] Vaswani, A. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[71] Xiao, G., Tang, J., Zuo, J., Guo, J., Yang, S., Tang, H., Fu, Y., and Han, S. Duoattention: Efficient long-context llm inference with retrieval and streaming heads. *arXiv preprint arXiv:2410.10819* (2024).

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

[73] Xie, Y., Wu, J., Tu, H., Yang, S., Zhao, B., Zong, Y., Jin, Q., Xie, C., and Zhou, Y. A preliminary study of o1 in medicine: Are we closer to an ai doctor? *arXiv preprint arXiv:2409.15277* (2024).

[74] Yang, Y., Zhao, L., Li, Y., Zhang, H., Li, J., Zhao, M., Chen, X., and Li, K. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022), pp. 768–781.

[75] Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (2022), pp. 521–538.

[76] Zhang, H., Tang, Y., Khandelwal, A., and Stoica, I. {SHEPHERD}: Serving {DNNs} in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (2023), pp. 787–808.

[77] 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* (2024).

[78] Zhao, Y., Lin, C.-Y., Zhu, K., Ye, Z., Chen, L., Zheng, S., Ceze, L., Krishnamurthy, A., Chen, T., and Kasikci, B. Atom: Low-bit quantization for efficient and accurate llm serving. *Proceedings of Machine Learning and Systems 6* (2024), 196–209.

[79] Zhong, T., Liu, Z., Pan, Y., Zhang, Y., Zhou, Y., Liang, S., Wu, Z., Lyu, Y., Shu, P., Yu, X., et al. Evaluation of openai o1: Opportunities and challenges of agi. *arXiv preprint arXiv:2409.18486* (2024).

[80] Zuo, P., Lin, H., Deng, J., Zou, N., Yang, X., Diao, Y., Gao, W., Xu, K., Chen, Z., Lu, S., et al. Serving large language models on huawei cloudmatrix384. *arXiv preprint arXiv:2506.12708* (2025).