JOINT ENCODING OF KV-CACHE BLOCKS FOR SCALABLE LLM SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern large language models (LLMs) drive interactive AI systems but are bottlenecked by the memory-heavy growth of key-value (KV) caches, which limits real-time throughput under concurrent loads. Existing KV-cache compression methods rely on rigid heuristics, disrupt tensor layouts, or require specialized compute, hindering scalability and deployment.

We propose joint encoding of KV-cache blocks, which fuses similar blocks across requests and input chunks into shared representations while preserving standard cache structure. This alleviates the KV-cache memory bottleneck, supporting high-concurrency serving without specialized hardware. Theoretically, we analyze the rate-distortion tradeoff of fused cache blocks under a Poisson process model. Empirically, our method achieves up to 4.38× KV-cache compression with negligible accuracy loss across diverse LLMs and benchmarks, outperforming recent structured and adaptive compression baselines. In real LLM serving, joint encoding improves the token throughput by ∼40% on a single-machine vLLM benchmark, demonstrating substantial gains in inference throughput. Code is available at `kv_joint_encoding-55B0`.

1 INTRODUCTION

Large Language Model (LLM) services have recently gained tremendous popularity, but their serving pipeline faces distinct challenges across two stages. In the *prefill stage*, the model processes the input sequence. This stage is compute-bound but executed only once per request. In contrast, the *decode stage* generates the subsequent output tokens, is repeated for every token, and is primarily constrained by memory bandwidth. The bottleneck arises because each decoding step requires fetching large model parameters from memory to maintain fast computation (Xie et al., 2025). This implies that batch size can be increased to serve more concurrent users without increasing generation latency, up to the point where computation time aligns with memory access time and becomes the new bottleneck.

Scaling LLM services for high concurrency rapidly encounters memory constraints. On-chip memory is consumed by three main components: the *model weights*, the *inference activations*, and the *key–value (KV) cache*. The weights, though large, are static and do not grow with users concurrency. Activations are lightweight, scale linearly with the number of users, and vanish after each step. The KV-cache, however, is both persistent and expansive: it scales with the number of active sessions, and requires reserving memory proportional to the maximum sequence length per user. In practice, the KV-cache rapidly becomes the dominant memory consumer, and hence, it is the primary obstacle to scaling users concurrency.

To alleviate these memory constraints, recent serving frameworks borrow ideas from virtual memory and paging in operating systems. In particular, vLLM introduces *Paged-Attention (PA)* Kwon et al. (2023a), which partitions the KV-cache of each request into fixed-size blocks, each holding a predefined number of key-value tokens. These blocks do not need to be contiguous in memory; instead, a *block table* tracks their locations, allowing the attention mechanism to retrieve relevant entries efficiently (Kwon et al., 2023b). This design removes the need of reserving per-user memory proportional to the maximum sequence length. Instead, memory consumption scales with the actual demand, which is typically far smaller on average.

Although Paged-Attention mitigates memory fragmentation, it does not address the fundamental asymmetry between the compute-bound prefill stage and the memory-bound decode stage. To address

this, the vLLM scheduler prioritizes *prefill requests*, reducing the Time-To-First-Token (TTFT) and allowing larger decode batches. Yet, this prioritization comes at a cost of delaying *decode requests*, increasing the Time-Between-Tokens (TBT). In long-context scenarios, this elevated TBT can become severe, stalling generation for several seconds (Agrawal et al., 2023). Recently, studies suggested to separate the two stages: requests are first processed on prefill servers, and their KV-cache is then migrated to decode servers. This separation enables independent optimization of the stages, reducing TTFT during prefill and TBT during decoding (Qin et al., 2024; Zhong et al., 2024; Hu et al., 2024).

In the prefill phase, strategies such as *Tensor Parallelism* and *chunked-prefilling* have been shown to reduce TTFT and increase compute utilization (Agrawal et al., 2023; 2024). In the decode phase, however, compute utilization remains low because each step requires fetching the model parameters to generate only a single token per request. Increasing the batch size can mitigate this inefficiency, but larger batches demand proportionally more KV-cache and are limited by the memory.

To reduce this overhead, prefix sharing has been proposed: grouping requests with identical prefixes so they can share the cached blocks and hence minimize memory (Zhu et al., 2024; Juravsky et al., 2024). Yet, exact prefix matches are rare in heterogeneous workloads, limiting its practical benefit. This limitation motivates the need for more general block-sharing mechanisms that extend the advantages of prefix sharing without relying on exact prefix matches.

In this paper, we address the KV-cache bottleneck in LLM serving by introducing a *joint-encoding scheme* that compresses and reuses similar cache blocks across requests. We develop two complementary methods: *Batch Fast-Fusion (BFF)*, which fuses blocks *across different requests* prior to decoding, and *Chunks Fast-Fusion (CFF)*, which fuses blocks *across input chunks* during prefilling. Both schemes reduce KV-cache size, enable larger batch sizes, and lower network bandwidth demands. At the core of our approach is a *tree-structured fusion strategy* that efficiently identifies encoding opportunities while preserving model accuracy. By applying joint encoding in both prefill and decode phases, our scheme delivers high throughput even under heterogeneous workloads with diverse input/output lengths and arrival patterns. These new fusion strategies substantially extend the generality and efficiency of cache sharing, delivering both memory and bandwidth savings in heterogeneous, real-world workloads.

The remainder of this paper is organized as follows: Section 2 discusses the most relevant related work. Section 3 outlines the motivation behind block fusion in LLM serving systems. Section 4 describes the proposed scheme in detail. Section 5 analyzes the scheme in lens of the point processes, Section 6 presents the experimental results and evaluation, and Section 7 concludes the paper.

2 Related Work

A major line of research has focused on *reducing the memory footprint of the KV-cache*, primarily through compression. Key approaches include quantization, low-rank approximation, and selective eviction. *Quantization-based methods*, such as Hooper et al. (2024); Liu et al. (2024d), compress key and value embeddings to 2–3 bits with minimal accuracy loss, substantially increasing batch size and throughput over standard precision. Low-rank and latent representations, for example, Multi-Head Latent Attention (MLA) Liu et al. (2024a) and ReCalKV Yan et al. (2025) stored KV matrices in compact subspaces that are later reconstructed as needed. This approach achieves a significant reduction in the KV-cache size while maintaining attention accuracy. Recently, Meng et al. (2025) suggested a method for transforming the attention of pretrained models to MLA architectures.

Given the redundancy of KV states *across adjacent layers*, other works interpolate vector directions or use SVD to compress cross-layer states (e.g., Liu et al. (2024b); Chang et al. (2025)), often retaining only a subset of tokens for quality. Combining quantization with these techniques achieves KV-cache compression ratios up to 5×. Recent work leverages adaptive arithmetic encoding (Liu et al., 2024c), head-level token importance with residual merging (Liu et al., 2025), and sensitivity-driven dynamic sparsity (Zhang et al., 2024; Yang et al., 2025) to further boost cache efficiency.

A complementary research direction explores *memory sharing across requests* by reusing computation for common input segments. Frameworks like HydraGen (Juravsky et al., 2024), RelayAttention (Zhu et al., 2024), and vLLM/SGLang (Wu et al., 2025; Zheng et al., 2023) exploit prefix sharing, where requests with identical prefixes share cached states, reducing both computation and memory requirements. However, such methods are limited by the rarity of exact prefix matches in practical

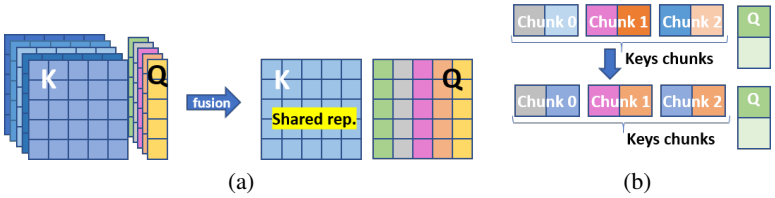


Figure 1: Attention computation (a) batch of requests with similar keys blocks. After the fusion, the KV=cache footprint is smaller and batch decoding is optimized by performing matrix-matrix multiplication. (b) After fusing chunks into a unified representation, the blocks computation of chunks 0 and 1 can be reused in chunk 2 (same color).

workloads. Fragmentation and fine-grained reuse of overlapping segments are considered in (Zhang et al., 2025; Prabhu et al., 2025; Yang et al., 2025).

Despite significant progress, most prior methods rely on rigid heuristics, disrupt native tensor layouts, or require exact prefix matches, limiting scalability and flexible integration into modern LLM serving pipelines. Our work directly addresses these limitations by introducing joint compression of KV-cache blocks, allowing finer-grained, layout-preserving, and general block sharing that extends the benefits of prefix reuse to arbitrarily similar segments. This flexibility is critical for memory- and bandwidth-efficient serving in real-world settings.

3 Motivation

Improving the throughput of LLM serving is often achieved by increasing batch size. Since the decoding phase is primarily memory-bound, dominated by repeated fetching of large weight matrices, larger batch sizes do not substantially increase per-request decoding latency. However, serving scalability remains constrained by the memory footprint of the KV-cache, which must be reserved per user session. Apart from the common compression methods like quantization and sparsification (by eviction), prefix-sharing techniques like Juravsky et al. (2024) and Zheng et al. (2024) demonstrate that overlapping prompt segments across requests enable KV-cache sharing. In addition, reuse of computed blocks as well as improved computation can improve throughput by up to 36% (Huijong Jeong & Kim, 2024). Nonetheless, these approaches rely on *exact* prefix matches, which severely limits applicability in real-world settings where variations in phrasing or task-specific inputs are common (Wu et al., 2025; Zheng et al., 2024). For example, two translation requests with slightly different introductory phrases (“Help me translate” vs. “Translate this”) would fail to share any KV blocks under strict prefix matching (Wu et al., 2025).

To overcome the strict requirement of an exact match, we propose a KV-cache blocks joint-encoding scheme that fuses blocks based on similarity threshold. This approach extends the benefits of prefix sharing to non-identical contexts. Figure 1 highlights the memory and computational benefits of jointly encoding blocks in Softmax Attention (SA). Specifically, Figure 1(a) illustrates the benefits in the decoding phase, where keys with similar representations (indicated by color) can be combined into a unified representation. Fusing these keys (and values) into a unified representation not only allows increasing the batch size but also facilitates optimized matrix-matrix multiplication, instead of multiple matrix-vector multiplications. In the prefill phase, Figure 1(b) illustrates how chunking input requests allows joint encoding of blocks across chunks, which reduces KV blocks memory, and thus, facilitates handling longer inputs and further reusing computation in SA. Consequently, any SA computation involving a fused block can be reused in subsequent chunks that include instances of this block, maximizing efficiency and throughput.

In both cases, after encoding, some blocks share the same representation and the block table points to the shared representation. Thus, any SA computation that involves a fused block can be reused in later chunks that comprise an instance of this block. The final SA result is derived by merging the SA of the fused components with that of the unique components (and rescaling). The recent version of vLLM can take advantage of the computation of SA with shared blocks.

4 KV BLOCKS JOINT-ENCODING

In this section, we introduce our KV blocks joint-encoding scheme for the chunk-prefilling setting, to further facilitate the joint encoding of the chunks while they are processed in the LLM (Qin et al., 2024; Agrawal et al., 2023). The main objective of our scheme is to maximize the batch size by jointly encoding KV blocks. The impact is threefold. First, in the decode phase, benefit from a higher arithmetic efficiency due to a larger batch size, and further, minimize KV-cache prefetching, mitigating the memory bandwidth limit. Second, in the prefill phase, it also enables the reuse of fused blocks computations, alleviating the compute limitation (see Figure 1). Third, reduce the usage of network bandwidth, which often becomes a bottleneck, especially when considering a high degree of parallelism (Agrawal et al., 2024). Certainly, it is crucial to achieve this goal while maintaining the model's accuracy.

To fulfill this goal, we propose Fast-Fusion (FF) method that jointly encodes similar blocks into a unified representation if the cosine similarity between contexts of different blocks is above predefined threshold. The similarity threshold is substantial for both the resulting *compression ratio* and *model's accuracy*. To avoid excessive blocks comparisons overhead, we design an efficient *tree-like fusion strategy* that scales as $O(B \log B)$ and $O(C \log C)$, where B is the batch size and C is the number of chunks, respectively. This strategy allows the parallelization of the fusion process at each level of the tree. Accordingly, the fusion can be done over blocks of different requests (i.e., BFF) or blocks of different chunks (i.e., CFF).

In practice, the KV-cache layout in vLLM is (B, p, t, h, d), where p, t, h, and d are the number of blocks, tokens per block, number of heads, and the head embedding size, respectively. Before encoding, the KV-cache is unfolded into a convenient layout of blocks per request or chunk. Specifically, let $r = t \cdot h \cdot d$, and note that $C = \lfloor p \cdot t/(\text{size of chunk}) \rfloor$, we use a layout of (B, p, r) and $(C, p/C, r)$ for the BFF and the CFF, respectively. The norm of each r is stored to allow a proper rescaling of the fused blocks. The algorithm is applied iteratively for every N chunks or requests in each layer.

A pseudo-code of the FF algorithm is given in Algorithm 1. Roughly speaking, after unfolding the KV-cache into a convenient layout and storing the norms, we recursively call the FF method in Algorithm 1, which fuses blocks of different requests or chunks if their similarity level is above threshold. Intuitively, blocks represented by r-dimensional vectors can also be expressed in terms of their norm and corresponding r-dimensional unit (direction) vectors. In this view, fusion can be understood as aligning multiple unit vectors into a single unified direction, while *preserving the distinctiveness of the original blocks through their norms*. This allows representing multiple blocks using a single unit vector and a norm (scalar) per block. To further enhance the compute, the fused blocks are taken into account in the attention computation, allowing low-level kernels to leverage jointly encoded blocks in the decoding phase, and reuse computations in the prefill phase. Only one copy of the fused blocks is needed, and redundant copies can be evicted to reduce memory usage. Further, different number of blocks may be encoded in each layer, which requires running each layer with its own block table, as implemented in vLLM. Note that our scheme attains a compression that is at least the compression attained by shared prefix methods, since we further compress inputs that do not share prefixes.

Consequently, the joint encoding allows us to reuse computations to address the limitations of compute, memory bandwidth, and network bandwidth. The tree structure is beneficial for detecting fusion opportunities at a reasonable cost in large-scale prefill and decoding serving systems.

Remark 1. *The order of requests impact the resulting compression. Specifically, it is beneficial to place shorter requests (in terms of number of blocks) on the left tree to gain higher diversity per block. At the best case, all the blocks in the right tree will be unified into the left tree blocks representation, yielding the maximal compression. However, the responsibility of ordering the requests belongs to the scheduler, which is out of the scope of this paper.*

Remark 2. *To avoid manual tuning, the threshold can be self-adjusted using a simple feedback. For example, percentile-based or target-compression strategies monitor fusion statistics during serving: if too many blocks are fused (or compression exceeds a desired CR), the threshold is increased, and vice versa. Alternatively, a small calibration set can be used to refine the threshold based on acceptable distortion levels, providing a lightweight and fully automated adaptation mechanism.*

Remark 3. *Prior work (e.g., Wu et al. (2025)) demonstrated that prefix-sharing may leak information through timing side-channels. In particular, by examining the TTFT, an attacker can determine*

Algorithm 1 Fast Fusion

Require: KV-cache (k,v), threshold (thr)
  if len(k) == 1 **then**
    return normalize(k), normalize(v)
  end if
  (*tree left*)
  $\mathrm{k}_l, \mathrm{v}_l \leftarrow$ FastFusion(k[: len(k)//2], v[: len(v)//2], thr)
  (*tree right*)
  $\mathrm{k}_r, \mathrm{v}_r \leftarrow$ FastFusion(k[len(k)//2 :], v[len(v)//2 :], thr)
  sim $\leftarrow (\mathrm{k}_l \mathrm{k}_r^\top)$ (*similarity per block of k_l*)
  for each row in sim **do**
    $idx_{\text{fused}} \leftarrow$ indices where $\mathrm{sim}(row) > thr$
    if $|idx_{\text{fused}}| > 0$ **then**
      (*block fusion*)
      $\mathrm{k}_l(row) \leftarrow \mathrm{normalize}(\mathrm{k}_l(row) + \sum_{i \in idx_{\text{fused}}} \mathrm{k}_r(i))$
      $\mathrm{v}_l(row) \leftarrow \mathrm{normalize}(\mathrm{v}_l(row) + \sum_{i \in idx_{\text{fused}}} \mathrm{v}_r(i))$
      evict fused blocks at $\mathrm{k}_r(idx_{\text{fused}})$ and $\mathrm{v}_r(idx_{\text{fused}})$
      (*for matrix-matrix efficiency*)
      mark shared blocks at $\mathrm{k}_l(row)$, and $\mathrm{v}_l(row)$
    end if
  end for
  update block-table bt
  return cat([k_l | k_r non-shared blocks]),
    cat([v_l | v_r non-shared blocks])

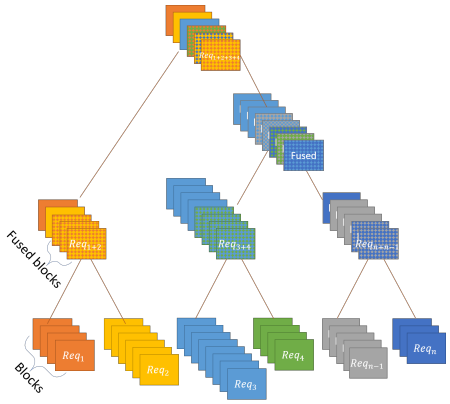


Figure 2: BFF example for 6 request, where each request has a different number of blocks (and a different color). The Fast Fusion is recursively called, fusing every pair of requests (fused blocks are depicted by a mixed color). Then, fusing pairs of pairs, and so on, until all the requests' blocks are jointly described in the KV-cache. Clearly, some blocks are not fused (remain their original color), and some are fused more than once while traversing the tree.

whether a particular prefix has already been cached by the vLLM. Interestingly, joint encoding mitigates such risks: because multiple distinct KV blocks *are mapped to a shared latent direction, any reduction in compute time is no longer attributable to a single prefix or token, but to a many-to-one cluster. This substantially weakens the leakage signal compared to prefix matching.*

5 Analysis

The similarity threshold is substantial to the algorithm performance as it dictates the resulting accuracy (or distortion) on the one hand, and the compression (or rate) on the other. In terms of rate-distortion, high threshold ensures low KV cache distortion, yet may result in low exceedance rate, and consequently, moderate compression. Similarly, low threshold value allows unifying many blocks into a single representation (high compression), yet may degrade the performance due to a highly distorted KV cache. Hence, it is crucial to thoroughly study the impact of the threshold.

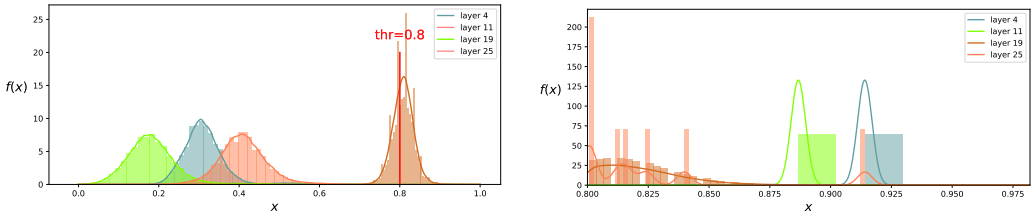
As mentioned, Algorithm 1 recursively fuses pairs of requests (see Figure 2 for illustration). Assume a pair of requests with m_1 and m_2 blocks, respectively, for which there are $n = m_1 \cdot m_2$ similarity samples $\{x_i\}_{i=1}^n$. Then, using Kernel Density Estimation (KDE) with a Gaussian kernel (Wand & Jones, 1994, Ch. 2), the similarity density in each layer is approximately $f_h(x) = \frac{1}{nh} \sum_{i=1}^n \phi\left(\frac{x - x_i}{h}\right)$, where h is the kernel bandwidth (standard deviation). Therefore, the similarity distribution is approximately

$$F_h(x) = \frac{1}{n} \sum_{i=1}^{n} \Phi\left(\frac{x - x_i}{h}\right), \tag{1}$$

where ϕ and Φ are the Gaussian density and distribution functions, respectively. In other words, each sample x_i contributes a Gaussian kernel function with mean value x_i to the overall estimated probability density function, thus contributing a different probability of exceeding the threshold.

Proposition 1. *For a sufficiently high similarity threshold u, the number of above-threshold similarities is asymptotically a Poisson variable with rate*

$$\Lambda(u) = \frac{1}{n} \sum_{i=1}^{n} \exp\left\{-\frac{u - (hb_n + x_i)}{ha_n}\right\}, \tag{2}$$



(a) (b)

Figure 3: Empirical similarity and analysis for DeepSeek-R1-Distill-Qwen-7B (2025) on nVidia HelpSteer (2024) dataset for several layers. Bars represents the empirical similarity and solid line the KDE approximation. (a) Blocks similarity density. (b) Above-threshold density for threshold=0.8.

where $a_n = (2\log n)^{-1/2}$ *and* $b_n = (2\log n)^{1/2} - \frac{1}{2}(2\log n)^{-1/2}(\log\log n + \log(4\pi))$ *are normalization constants.*

Proof. We wish to analyze the asymptotic threshold exceedance rate of eq. (1). Since the indices of high threshold exceedance are random and are relatively rare, the number of above-threshold observations can be modeled as a Poisson random variable when the threshold is sufficiently high (Leadbetter et al., 2012, Ch. 5). In particular, let u be a threshold such that the kernel of each sample i satisfies $\Pr(\mathbf{x}_i > u) = \left(1 - \Phi\left(\frac{u - x_i}{h}\right)\right) = \Theta(1/n)$. Then, according to the *uniformly asymptotically negligible* condition, the number of threshold arrivals is approximately a Poisson variable with rate $\Lambda(u)$ when the samples are independent (Falk et al., 2010, Ch. 8.3). Similar treatment for dependent samples is given in (Coles et al., 2001, Ch. 5).

For the Gaussian case, the threshold exceedance rate is $\Lambda(u)$ given in eq. (2), where $a_n = (2\log n)^{-1/2}$ and $b_n = (2\log n)^{1/2} - \frac{1}{2}(2\log n)^{-1/2}(\log\log n + \log(4\pi))$ are normalizing constants (Kampeas et al., 2014, Theorem 5). □

Figure 3(a) depicts the empirical similarity and the above-threshold distribution together with the KDE approximation for DeepSeek-R1-Distill-Qwen-7B (2025) on nVidia HelpSteer (2024) dataset in several layers. Evidently, for this model and dataset, the similarity appears Gaussian, yet with different mean and variance in each layer. Interestingly, letting the similarity in layer ℓ be approximately Gaussian with mean $\mu_\ell = \frac{1}{n}\sum_{i=1}^{n} x_i$ and variance $\sigma_\ell^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \mu_\ell^2$, by the Poisson point process, the threshold exceedance rate is approximately $\Lambda_\ell(u) = n \cdot \left(1 - \Phi\left(\frac{u-\mu_\ell}{\sigma_\ell}\right)\right)$. Moreover, even though each layer follows a different distribution and, therefore, has a different threshold exceedance rate, by Proposition 1, the overall exceedance rate for all model layers can be evaluated.

The Poisson formulation facilitates analyzing the probability of observing non-compressible layer, and the expected compression ratio over the entire model. From the properties of the Poisson, we have the following corollary.

Corollary 1. *After the fusion, the compression ratio over L layers is*

$$\textit{compression ratio} = L(m_1 + m_2) / \left((L(m_1 + m_2) - \sum_{\ell=1}^{L} \Lambda_\ell(u) \right).$$

The probability of no fusion in layer ℓ *is*

$$\Pr(\textit{no fusion in layer } \ell) = \exp(-\Lambda_\ell(u))$$

Next, let us examine the attention distribution drift that stems from fusing blocks. Let $\mathbf{z}_t = \mathbf{q} \cdot \mathbf{k}_t / \sqrt{d}, \forall t \in 1, \ldots, p$ be the logits vector of a block with p tokens, and let let $\mathbf{s} = \exp(\mathbf{z}) / \sum_j \exp(z_j)$ be the softmax distribution before fusion. Similarly, let $\mathbf{z}' = \mathbf{z} + \delta$ and $\mathbf{s}' = \exp(\mathbf{z}') / \sum_j \exp(z'_j)$ be the perturbed (fused) logits and softmax distribution. Then, the attention distribution drift satisfies the following bound.

Lemma 1. *After fusion, the attention distribution drift is bounded by*

$$\|\mathbf{s}' - \mathbf{s}\|_1 \leq 2\epsilon + O(\epsilon^2)$$

where $\epsilon = \max_i \frac{\|\mathbf{k}_i\|\|\mathbf{q}\|}{\sqrt{d}} \sqrt{2(1-u)}$.

Proof is deferred to Appendix A.1. The analysis provides a theoretical foundation for understanding the trade-offs inherent in block fusion for LLM serving. The Poisson point process modeling indeed shows that the similarity threshold directly governs the balance between compression and distortion in the KV cache. Our per-layer analysis predicts the expected compression for a given threshold, allowing skipping non-compressible layers in probability.

6 Results

In this section, we present the results of our context fusion scheme for CFF and BFF. Measuring the accuracy, compression, and latency together requires a dedicated kernel, which we keep in mind for future work. Sections 6.1-6.3 assess compression and accuracy, by fuse and replicate the rescaled cache across all fused blocks (without freeing blocks). Section 6.4 examines the end-to-end latency by fusing and freeing blocks (without rescaling).

We evaluate the performance of our approach using a commodity GPU on various benchmarks and models and compare it to the baseline performance. The results demonstrate the effectiveness of our scheduling algorithm in terms of compute efficiency, memory reduction, and network bandwidth consumption.

6.1 Block Diversity and Rate-Distortion

Both the number of requests (or chunks) and the number of blocks in each request (or chunk) influence the resulting Compression Ratio (CR). The larger the batch, the more requests to fuse, and the more blocks per request, the greater diversity of blocks for fusion, which yields a better CR. To assess the significance of each, we examine the BFF CR on the random data benchmark in vLLM API (2023), using a fixed similarity threshold for a various number of requests and blocks per request using Llama2-7B with 16 tokens per block. Interestingly, Figure 4(a) reveals that the diversity of blocks per request is more significant than the diversity of requests, since the CR grows faster when more blocks per request are used.

Figure 4(b) and 4(c) depict the the rate distortion trade-off for BFF and CFF, respectively. The rate distortion curve is given for Llama2-7B on Das (2024) conversational dataset, where the number of requests (chunks) is fixed, and the similarity threshold varies. The baseline result is given for reference. Specifically, in Figure 4(b) the BFF achieves a CR of $\sim 2.15\times$, without degrading accuracy for batch size 256 with a block size of 16 tokens. In Figure 4(c), the CFF achieves a CR of $\sim 1.25\times$ without losing accuracy for 4 chunks with a block size of 16 tokens.

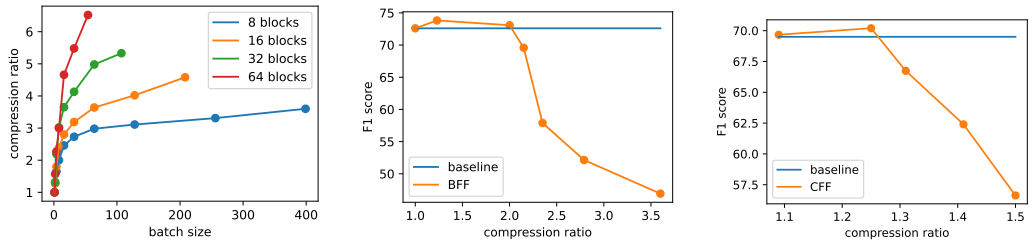
Remarkably, this rate distortion formulation indicates that a higher CR can be achieved without sacrificing accuracy when fixing the similarity threshold value. In particular, since only the threshold determines the resulting accuracy (distortion), once the threshold is fixed, increasing the batch size or the number of blocks per request yields a better CR due to diversity in requests and blocks.

6.2 Batch Fast-Fusion During Decoding

The impact of diversity on the CR and the resulting F1 score is of great practical interest, as it indicates the gain of a larger batch using our enhancement. Specifically, in this section, we investigate the CR and the F1 score when increasing the number of requests in the BFF scheme, where the similarity threshold is set to a fixed value, using the nVidia OpenMathInstruct-2 (2024) dataset.

Figure 5(a) depicts the CR versus batch size for the Llama3.1-8B model and Qwen2.5-72B. Notably, in both cases, the CR grows logarithmically with the batch size, reaching CR of $\sim 3.11\times$ and $\sim 4.38\times$ for batch sizes of 128 and 64, respectively.

Using the same settings, we further evaluate the behavior of the F1 score when increasing the number of requests in a batch in Figure 5(b). Interestingly, the F1 score is a bit higher on average for both



(a) (b) (c)

Figure 4: CR and F1 score of BFF and CFF for Llama-2 7B. (a) CR vs. batch size for diverse number of blocks on vLLM random-data benchmark. (b) BFF F1 score vs. CR for batch size 256 on conversational dataset. (c) CFF F1 score vs. CR, for 4 chunks on conversational dataset.

models, and especially for the Qwen2.5 72B model. The phenomenon where averaging similar blocks improves the model accuracy can be interpreted through the lens of the *crowd wisdom effect*. Each block representation, akin to an individual expert, contributes unique insights about the block context over the layers. When focusing on relatively similar representations, averaging these representations reduces individual biases and errors, much as a group makes more accurate decisions (Trott, 2024).

Table 1 describes the F1 score and the CR behavior when applying BFF to every 8 requests (i.e., batch size 8) for various thresholds in a variety of MMLU (2020) tasks and GSM8k (2022). These results highlight the ability of the BFF to significantly reduce the KV cache size in many cases, without compromising accuracy, showcasing its value in improving decoding efficiency.

Table 1: F1 score and CR (in parenthesis) achieved by BFF for Llama3.1-8B.

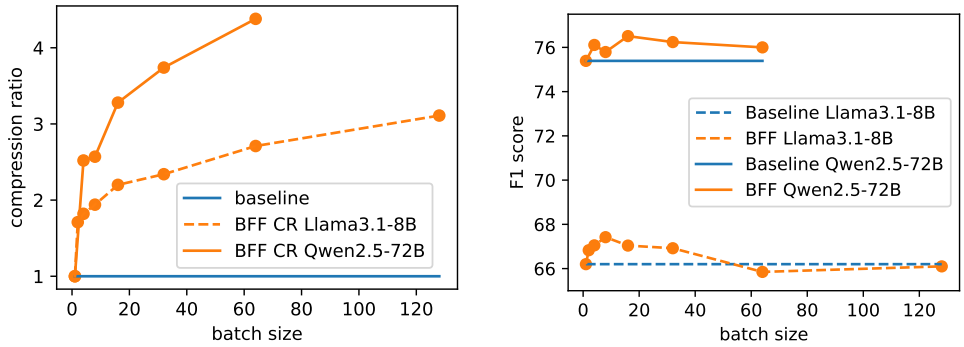
Model	Method	GSM8K	Con.Phy.	E.Eng.	F.Logic	HS.Bio	misc.	sociology	Average
Llama3.1-8B	BFF thr=0.7	41.47 (×3.1)	37.87 (×3.29)	**40.69** (×3.21)	35.2 (×2.69)	44.19 (×2.51)	51.72 (×2.93)	**40.8** (×3.11)	41.71 (×2.98)
	BFF thr=0.74	51.91 (×2.52)	40.85 (×2.52)	38.62 (×2.48)	40 (×2.21)	46.13 (×2.04)	51.34 (×2.32)	39.3 (×2.4)	44.02 (×2.36)
	BFF thr=0.78	**62.86** (×2.09)	40.43 (×1.63)	40 (×1.59)	**41.27** (×1.53)	45.48 (×1.37)	**54.79** (×1.52)	38.31 (×1.61)	46.16 (×1.62)
	Baseline	62.46	**40.85**	40.69	40	**48.39**	54.79	40.8	**46.85**

Practically, these results indicate up to $\sim 3.11\times$ and $\sim 4.38\times$ reduction of KV cache blocks, for Llama3.1-8B and Qwen2.5-72B, respectively. Of course, this also mitigates the memory fetching and the network bottleneck which is significant, especially when considering a high level of parallelism. Furthermore, using fused blocks is beneficial for performing hardware-optimized matrix-matrix multiplications (Juravsky et al., 2024).

6.3 Chunks Fast Fusion During Prefill

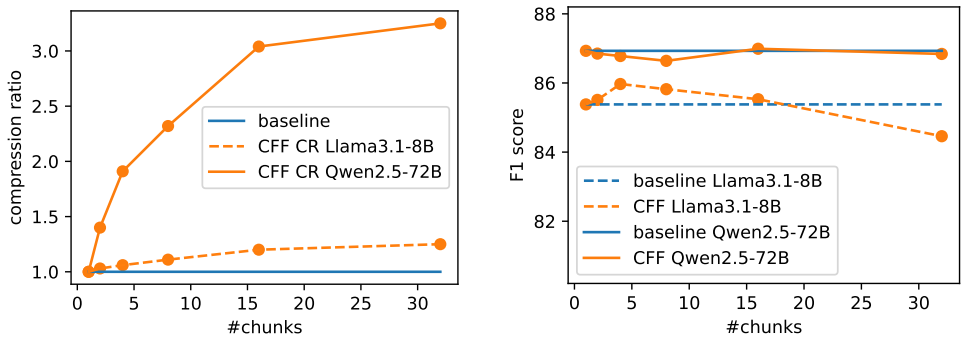
In this section, we examine the CR and the resulting F1 score for the CFF. Even though eliminating the distance that stems from RoPE can yield a higher CR, it disables reusing computations, which is substantial for the prefill phase. Thus, the CFF is applied to blocks within chunks, together with their positioning. In addition to compression, the results also indicate the computation reuse factor.

To characterize the impart of CFF on the CR and F1 score, the similarity threshold is set to a fixed value, and the number of chunks to fuse is varied. The evaluation is performed on Llama3.1-8B and Qwen2.5-72B models with 16 tokens per block on the Longbench qmsum dataset Bai et al. (2023). This dataset contains relatively long inputs, which allows characterizing the CR and accuracy of CFF when applied to increasing number of chunks. Figure 6(a) depicts the CR and when applying CFF to the chunks. As we see, the CFF uses up to $\sim 3.25\times$ fewer blocks, for which their computation can be reused, thus mitigating computational and network bottlenecks. Figure 6(b) depicts the F1 score of the CFF when scaling the number of chunks to fuse on the same task. The CFF manages to keep the accuracy in most cases, and experiences only a negligible accuracy loss.



(a) (b)

Figure 5: CR and F1 score of BFF vs. baseline for Llama-3.1 8B and Qwen2.5 72B on nVidia OpenMathInstruct-2 (2024) dataset. (a) CR vs. batch size. (b) F1 score vs. batch size.



(a) (b)

Figure 6: CR (a) and F1 score (b) of CFF vs. number of chunks for Llama3.1-8B and Qwen2.5-72B using the Longbench qmsum dataset.

Table 2 describes the F1 score and the CR behavior when applying CFF to every 8 chunks for various thresholds in a variety of Longbench tasks (Bai et al., 2023). The table highlights consistent gains in compression with minimal or no loss in accuracy. Notably, even at lower thresholds, the F1 scores remain comparable to the baseline, while achieving compression ratios of up to $\times 1.87$.

Table 2: F1 score and CR (in parenthesis) achieved by CFF for Llama3.1-8B.

Model	Method	LCC	RepoBench-P	PR-en	TREC	2wikimqa	GovReport	MQA-zh	Average
Llama3.1-8B	CFF thr=0.62	75.77 ($\times$1.87)	74.07 ($\times$1.59)	**21.04** ($\times$1.32)	40.48 ($\times$1.43)	**45.9** ($\times$1.38)	83.8 ($\times$1.59)	27.34 ($\times$1.53)	52.63 ($\times$1.53)
	CFF thr=0.64	77.12 ($\times$1.49)	74.8 ($\times$1.3)	18.3 ($\times$1.15)	40.3 ($\times$1.16)	45.88 ($\times$1.17)	81.39 ($\times$1.25)	38.08 ($\times$1.24)	53.7 ($\times$1.25)
	CFF thr=0.66	77.24 ($\times$1.26)	**75.99** ($\times$1.15)	18.38 ($\times$1.06)	**41.15** ($\times$1.06)	44.94 ($\times$1.06)	**83.83** ($\times$1.09)	**40.31** ($\times$1.1)	**54.55** ($\times$1.11)
	Baseline	**77.73**	75.57	18.74	40.95	44.12	82.81	39.18	54.16

‘

6.4 End-to-End Serving Perfromance

We evaluate BFF on a single-machine vLLM serving benchmark over a random dataset of 100 requests with input size 1024 and varying number of output tokens (with random-ratio 0.5) using Llama-3.1-8B. After the fusing blocks (with similarity above threshold 0.7), the model runner sends

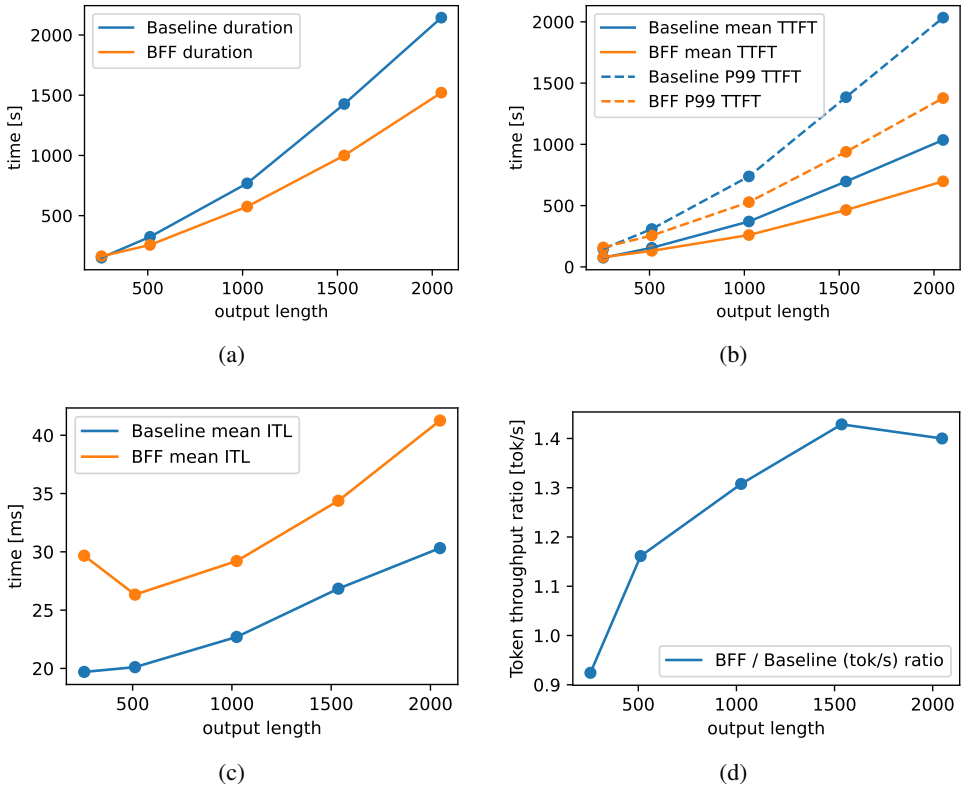


Figure 7: Throughput and end-to-end performance comparison on Llama3.1-8B (a) benchmark duration (b) TTFT mean and P99 (c) mean ITL (d) throughput ratio (tok/sec)

the scheduler the updated block-tables, allowing the scheduler to free unused blocks, and increase reference counts for shared blocks for further reuse across requests.

Even with our naive implementation, where the entire fusion overhead is reflected in the decode phase, we see that BFF yields measurable system-level benefits: lower prefill latency (TTFT), higher effective batching, and increased throughput. The Decode-phase inter-token latency (ITL) increased due to larger batch GEMMs, and the fusion overhead. However, the overall serving throughput and request completion time reduced substantially.

Overall, the results presented in this section demonstrate the effectiveness of our context-sharing scheduling scheme for the KV cache-centric disaggregated architecture. Our approach significantly improves compute efficiency, reduces memory and network bandwidth consumption, and scales well with increasing system size, making it a promising solution to accelerate LLM serving in resource-constrained scenarios.

7 Conclusion

In this paper, we presented *Fast Fusion*, a novel context-sharing enhancement that can improve LLM serving efficiency by introducing BFF and CFF. These techniques enable fine-grained fusion of similar KV cache blocks across requests or chunks, achieving up to $\times 4.38$ compression without compromising accuracy. By significantly reducing KV cache transfers, our method reduces significantly the average number of blocks, and thus, allows to scale the serving capacity effectively under heterogeneous workloads. Theoretical analysis based on Poisson point processes provides insight into the rate-distortion trade-offs, and extensive empirical evaluations across multiple benchmarks and model sizes validate the practical benefits. Looking forward, we plan to explore the impact on serving throughput, using adaptive threshold tuning, integration with quantization, and pruning methods.

REFERENCES

Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient LLM inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

Amey Agrawal, Junda Chen, Íñigo Goiri, Ramachandran Ramjee, Chaojie Zhang, Alexey Tumanov, and Esha Choukse. Mnemosyne: Parallelization strategies for efficiently serving multi-million context length LLM inference requests without approximations. *arXiv preprint arXiv:2409.17264*, 2024.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Chi-Chih Chang, Chien-Yu Lin, Yash Akhauri, Wei-Cheng Lin, Kai-Chiang Wu, Luis Ceze, and Mohamed S Abdelfattah. xkv: Cross-layer svd for kv-cache compression. *arXiv preprint arXiv:2503.18893*, 2025.

Stuart Coles, Joanna Bawa, Lesley Trenner, and Pat Dorazio. *An introduction to statistical modeling of extreme values*, volume 208. Springer, 2001.

Bhaskar Lal Das. Synthetic therapy conversations, 2024. URL `https://huggingface.co/datasets/Mr-Bhaskar/Synthetic_Therapy_Conversations`. Accessed: 2025-05-6.

DeepSeek-R1-Distill-Qwen-7B. Deepseek-r1-distill-qwen-7b, 2025. URL `deepseek-ai/DeepSeek-R1-Distill-Qwen-7B`. Accessed: 2025-05-6.

Michael Falk, Jürg Hüsler, and Rolf-Dieter Reiss. *Laws of small numbers: extremes and rare events*. Springer Science & Business Media, 2010.

GSM8k. Open-ai gsm8k benchmark, 2022. URL `https://huggingface.co/datasets/openai/gsm8k`. Accessed: 2025-05-6.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.

Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate LLM inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

Yeonjoon Jung Huijong Jeong, Daehyun Ahn and Taesu Kim. [vllm vs tensorrt-llm] #12. automatic prefix caching, 2024. URL `https://blog.squeezebits.com/vllm-vs-tensorrtllm-12-automatic-prefix-caching-38189`. Accessed: 2025-05-11.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput LLM inference with shared prefixes. *arXiv preprint arXiv:2402.05099*, 2024.

Joseph Kampeas, Asaf Cohen, and Omer Gurewitz. Capacity of distributed opportunistic scheduling in nonhomogeneous networks. *IEEE Transactions on Information Theory*, 60(11):7231–7247, 2014.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Yu, Joey Gonzalez, Hao Zhang, and Ion Stoica. vllm: Easy, fast, and cheap llm serving with pagedattention. *See https://vllm. ai/(accessed 9 August 2023)*, 2023a. URL `https://github.com/vllm-project/vllm`.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023b.

Malcolm R Leadbetter, Georg Lindgren, and Holger Rootzén. *Extremes and related properties of random sequences and processes*. Springer Science & Business Media, 2012.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024a.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Reza Haffari, and Bohan Zhuang. Minicache: Kv cache compression in depth dimension for large language models. *Advances in Neural Information Processing Systems*, 37:139997–140031, 2024b.

Xin Liu, Pei Liu, and Guoming Tang. Zsmerge: Zero-shot kv cache compression for memory-efficient long-context llms. *arXiv preprint arXiv:2503.10714*, 2025.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, et al. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 38–56, 2024c.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024d.

Fanxu Meng, Pingzhi Tang, Xiaojuan Tang, Zengwei Yao, Xing Sun, and Muhan Zhang. Transmla: Multi-head latent attention is all you need. *arXiv preprint arXiv:2502.07864*, 2025.

MMLU. Massive multitask language understanding (MMLU) benchmark, 2020. URL `https://huggingface.co/datasets/cais/mmlu`. Accessed: 2025-05-6.

nVidia HelpSteer. nvidia helpsteer, 2024. URL `https://huggingface.co/datasets/nvidia/HelpSteer`. Accessed: 2025-05-6.

nVidia OpenMathInstruct-2. nvidia openmathinstruct-2, 2024. URL `https://huggingface.co/datasets/nvidia/OpenMathInstruct-2`. Accessed: 2025-05-6.

Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vattention: Dynamic memory management for serving llms without pagedattention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 1133–1150, 2025.

Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for LLM serving. *arXiv preprint arXiv:2407.00079*, 2024.

Sean Trott. Large language models and the wisdom of small crowds. *Open Mind*, 8:723–738, 05 2024. ISSN 2470-2986. doi: 10.1162/opmi_a_00144. URL `https://doi.org/10.1162/opmi_a_00144`.

vLLM API. Randomdataset, 2023. URL `https://docs.vllm.ai/en/latest/api/vllm/vllm.benchmarks.datasets.html#vllm.benchmarks.datasets.RandomDataset`. Accessed: 2025-05-11.

Matt P Wand and M Chris Jones. *Kernel smoothing*. CRC press, 1994.

Guanlong Wu, Zheng Zhang, Yao Zhang, Weili Wang, Jianyu Niu, Ye Wu, and Yinqian Zhang. I know what you asked: Prompt leakage via kv-cache sharing in multi-tenant llm serving. In *Proceedings of the 2025 Network and Distributed System Security (NDSS) Symposium. San Diego, CA, USA*, 2025.

Rui Xie, Asad Ul Haq, Linsen Ma, Yunhua Fang, Zirak Burzin Engineer, Liu Liu, and Tong Zhang. Reimagining memory access for llm inference: Compression-aware memory controller design. *arXiv preprint arXiv:2503.18869*, 2025.

Xianglong Yan, Zhiteng Li, Tianao Zhang, Linghe Kong, Yulun Zhang, and Xiaokang Yang. Recalkv: Low-rank kv cache compression via head reordering and offline calibration. *arXiv preprint arXiv:2505.24357*, 2025.

Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. Kvlink: Accelerating large language models via efficient kv cache reuse. *arXiv preprint arXiv:2502.16002*, 2025.

Chen Zhang, Kuntai Du, Shu Liu, Woosuk Kwon, Xiangxi Mo, Yufeng Wang, Xiaoxuan Liu, Kaichao You, Zhuohan Li, Mingsheng Long, et al. Jenga: Effective memory management for serving llm with heterogeneity. *arXiv preprint arXiv:2503.18292*, 2025.

Yanqi Zhang, Yuwei Hu, Runyuan Zhao, John Lui, and Haibo Chen. Unifying kv cache compression for large language models with LeanKV. *arXiv preprint arXiv:2412.03131*, 2024.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *CoRR*, 2023.

Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. Batchllm: Optimizing large batched llm inference with global prefix sharing and throughput-oriented token batching. *arXiv preprint arXiv:2412.03594*, 2024.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, 2024.

Lei Zhu, Xinjiang Wang, Wayne Zhang, and Rynson WH Lau. Relayattention for efficient large language model serving with long system prompts. *arXiv preprint arXiv:2402.14808*, 2024.

A Appendix

A.1 Proof to Lemma 1

Proof. First, since $\mathbf{k}$ and $\mathbf{k}'$ have the same norm, we can represent then as $\mathbf{k} = \|\mathbf{k}\| \cdot \hat{\mathbf{k}}$ and $\mathbf{k}' = \|\mathbf{k}\| \cdot \hat{\mathbf{k}}'$. Let $\delta_t = \mathbf{z}'_t - \mathbf{z}_t = \|\mathbf{k}\| \cdot \mathbf{q} \cdot (\hat{\mathbf{k}}' - \hat{\mathbf{k}})/\sqrt{d}$. Thus,

$$\|\delta_t\| = \|\mathbf{z}'_t - \mathbf{z}_t\| \tag{3}$$

$$\leq \|\mathbf{k}\| \cdot \|\mathbf{q}\| \cdot \|\hat{\mathbf{k}}' - \hat{\mathbf{k}}\|/\sqrt{d} \tag{4}$$

$$= \|\mathbf{k}\| \cdot \|\mathbf{q}\| \cdot \sqrt{2(1 - \hat{\mathbf{k}}' \cdot \hat{\mathbf{k}})}/\sqrt{d} \tag{5}$$

$$\leq \|\mathbf{k}\| \cdot \|\mathbf{q}\| \cdot \sqrt{2(1-u)}/\sqrt{d} \tag{6}$$

Since $\epsilon = \max_i |\delta_i|$, the denominator of $\mathbf{s}$ satisfies

$$\exp(-\epsilon) \sum_j \exp(z_j) \leq \sum_j \exp(z_j + \delta_j) \leq \exp(\epsilon) \sum_j \exp(z_j)$$

Similarly, for the numerator of $\mathbf{s}$ satisfies

$$\exp(-\epsilon) \exp(\mathbf{z}) \leq \exp(\mathbf{z} + \delta) \leq \exp(\epsilon) \exp(\mathbf{z})$$

From these bounds we obtain that each index i satisfies $\exp(-2\epsilon)s_i \leq s'_i \leq \exp(2\epsilon)s$, as

$$\mathbf{s}' \leq \exp(\epsilon) \exp(\mathbf{z}) / \exp(-\epsilon) \sum_j \exp(z_j) = \exp(2\epsilon)\mathbf{s}.$$

Now, let us divide into the case. When $s'_i \geq s_i$, then $s'_i - s_i \leq s_i(\exp(2\epsilon) - 1)$. Similarly, when $s'_i < s_i$, then $s_i - s'_i \leq s_i(1 - \exp(-2\epsilon)) \leq s_i(\exp(2\epsilon) - 1)$. Thus, $|s'_i - s_i| \leq (\exp(2\epsilon) - 1)s_i$.

Finally, sum up the index i, since $\sum_i s_i = 1$, we have $\sum_i |s'_i - s_i| = \|\mathbf{s}' - \mathbf{s}\|_1 \leq (\exp(2\epsilon) - 1) \leq 2\epsilon + O(\epsilon^2)$ from Taylor expansion. Since $\epsilon \leq \max_i \|\mathbf{k}_i\| \|\mathbf{q}\| \sqrt{2(1-u)}/\sqrt{d}$, the lemma follows. □