

# DYNAKV: Enabling Accurate and Efficient Long-Sequence LLM Decoding on Smartphones

Tuowei Wang\*  
Tsinghua University  
Beijing, China

Minxing Huang\*  
Tsinghua University  
Beijing, China

Fengzu Li  
Tsinghua University  
Beijing, China

Ligeng Chen  
Honor Device Co., Ltd.  
Shenzhen, China

Jinrui Zhang  
Tsinghua University  
Beijing, China

Ju Ren  
Tsinghua University  
Beijing, China

## Abstract

As the demand for human-like reasoning, multi-turn dialogues, and long-form responses grows, large language models (LLMs) are increasingly expected to support efficient and effective long-sequence decoding. However, due to limited DRAM capacity, long-sequence LLM decoding on smartphones is constrained by the key-value cache (KVCache), whose memory footprint increases linearly with sequence length. Retrieval-based methods mitigate DRAM pressure by offloading KVCache to flash and retrieving query-relevant entries through cluster-based indexing. Unfortunately, as decoding progresses, KVCache distribution shifts render static or local cluster updates progressively misaligned, excluding essential entries or fetching redundant ones. These issues are further exacerbated by smartphone-specific limitations in bandwidth, IOPS, and memory capacity.

We propose DYNAKV, the first adaptive KVCache management approach that jointly addresses accuracy and efficiency for long-sequence decoding on smartphones. DYNAKV integrates three key techniques: (1) Migration-Free Cluster Adaptation, which adaptively splits clusters during retrieval without incurring additional transfers; (2) Continuity-Centric Flash Management, which co-locates correlated entries and clusters and employs a dual-head layout for efficient updates; and (3) Memory-Efficient Cache Design, which virtualizes cache space across DRAM and flash and extends replacement policies to align with cluster-level access patterns. Evaluations demonstrate that DYNAKV improves retrieval accuracy and reduces end-to-end latency compared to state-of-the-art solutions, achieving average gains of 1.38 $\times$  in accuracy and 1.47 $\times$  speedups. Furthermore, the insights of DYNAKV naturally extend to other long-context workloads and multi-tier memory hierarchies, underscoring its broader applicability.

## 1 Introduction

The emergence of large language models (LLMs) has fundamentally transformed the processing of long-context data. Beyond understanding extensive inputs such as documents [7, 11] or codebases [16, 38], LLMs are increasingly expected to generate coherent and long-form outputs, particularly on

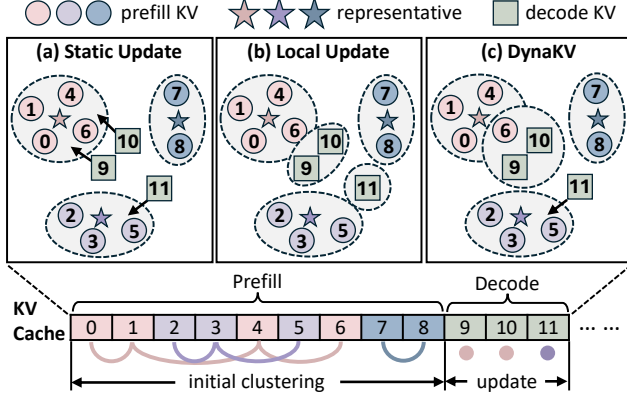
mobile devices like smartphones. On one hand, state-of-the-art LLMs are moving toward reasoning-oriented designs [9, 15, 21], leveraging structured thought processes to strengthen problem-solving capabilities. On the other hand, mobile applications such as advanced conversational agents demand both detailed response generation [27, 40, 59] and sustaining multi-turn dialogues [5, 46, 63]. These trends together underscore the critical need to enable efficient and effective long-sequence LLM decoding on smartphones.

Similar to other long-context scenarios, the limited memory capacity on smartphones poses a major challenge for long-sequence LLM decoding. Since the memory usage of the **key-value cache (KVCache)** scales linearly with sequence length, it quickly becomes the dominant bottleneck at longer contexts. To mitigate this issue, prior studies have investigated the contextual sparsity inherent in the attention mechanism [10, 34, 50, 52]. In practice, each query vector interacts significantly with only a small, query-dependent subset of key and value vectors. Building on this property, a variety of retrieval-based methods [30, 31, 55, 60] have been proposed for memory-constrained settings, offloading the entire KVCache to external storage and retrieving only the relevant entries into main memory on demand.

To balance retrieval efficiency and accuracy, the KVCache is typically partitioned into clusters based on embedding similarity, which then serve as the fundamental retrieval units. Since global clustering is both computationally and memory-intensive, it is performed only once during the prefill phase. Newly generated KV entries are either greedily appended to the nearest existing clusters (*Static Update*) [18, 60] or periodically re-clustered within a local context window (*Local Update*) [31, 45], as illustrated in Figure 1. These methods remain effective when the prefill proportion is relatively large. However, the situation changes in long-sequence decoding, where generated tokens dominate the inference context.

Specifically, we observe that as the decode length increases, the distribution of the KVCache undergoes continual transformation, exhibiting clustering patterns that diverge substantially from the initial ones, a phenomenon we term **KV-Cache Distribution Shift**. Unfortunately, existing clustering methods fail to effectively accommodate this dynamic

\*Both authors contributed equally to this research.



**Figure 1.** Illustration of three KVCache cluster management strategies. (a) Static Update: new entries are greedily appended to their nearest existing clusters. (b) Local Update: new entries are re-clustered independently of existing clusters. (c) DYNKV: adaptively appends new entries or splits existing clusters to maintain accuracy and efficiency.

shift, resulting in suboptimal KVCache retrieval. In particular, excluding essential entries degrades inference accuracy, while transferring redundant entries wastes I/O bandwidth. This observation motivates a critical question: *Can we develop an adaptive KVCache management that preserves both accuracy and efficiency of KVCache retrieval on smartphones?*

However, this is not a low-hanging fruit, as it stems from the inherent misalignment between the adaptive demands of clustering mechanisms and the stringent constraints of smartphone memory hierarchies. The presence of KVCache clustering shift necessitates access to global information for adaptive clustering, which unavoidably incurs additional data transfers from external storage to main memory. Given the hardware characteristics of smartphones, such I/O operations are readily bottlenecked by both limited **bandwidth**, restricted **Input/Output Operations Per Second (IOPS)**, and constrained **memory capacity**. Our analysis highlights three critical technical challenges that must be tackled:

- (1) **Bandwidth-Limited Adaptation.** Adaptive clustering requires continuous updates to existing clusters as new KV entries are generated. Since external storage lacks compute capability, the relevant clusters must first be transferred into main memory before being updated. The resulting frequent transfers can quickly lead to I/O performance bottlenecks.
- (2) **IOPS-Bounded Flash Access.** Retrieved KV entries exhibit scattered and dynamic access patterns across different queries. Placing the KVCache naively in strict sequence order results in fragmented flash accesses, exacerbating the IOPS bottleneck. Alternatively, organizing it by clusters improves continuous reads but raises challenges for frequent updates.
- (3) **Memory-Constrained Caching.** Retaining frequently retrieved KV entries in main memory is essential for reducing

transfer overhead. However, the distinctive access patterns of KVCache render standard replacement policies ineffective. Moreover, cluster-based organization inflates cache demands, which is incompatible with the limited memory capacity.

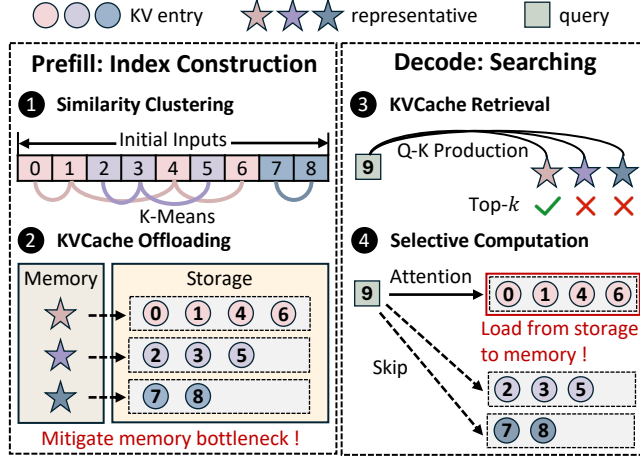
In this paper, we propose **DYNKV**, an efficient approach that enables long-sequence LLM decoding on smartphones with adaptive KVCache management. DYNKV employs three key techniques to tackle these challenges, respectively:

- (1) **Migration-Free Cluster Adaptation.** DYNKV design a variance-based self-adaptive clustering algorithm that preserves clustering effectiveness while maintaining cluster granularity as coarse as possible. To avoid extra I/O overhead, DYNKV adopts a delayed-split strategy that integrates cluster splitting naturally into the existing retrieval workflow.
- (2) **Continuity-Centric Flash Management.** To facilitate continuous read access and mitigate IOPS bottlenecks, DYNKV places all entries within a cluster as well as highly-correlated clusters continuously in flash. To support frequent adaptations, DYNKV employs a dual-head layout that allows cluster updates without interfering with each other.
- (3) **Memory-Efficient Cache Design.** DYNKV constructs a virtual cache space that unifies DRAM and flash. By leveraging idle bandwidth during computation, cache clusters are seamlessly swapped between the two tiers. To further improve cache hit rate, DYNKV augments standard replacement policies to align with cluster-based retrieval.

We evaluate DYNKV on three smartphones with distinct hardware configurations, benchmarking a diverse range of LLMs varying in structures and scales. The results demonstrate that DYNKV outperforms the state-of-the-art solutions in retrieval accuracy during long-sequence decoding, achieving improvements of 1.38 $\times$  on average. Moreover, the system-level optimizations guarantee retrieval efficiency, delivering average speedups of 1.47 $\times$  in end-to-end latency.

To the best of our knowledge, DYNKV is the first to simultaneously achieve both accuracy and efficiency of long-sequence LLM decoding on smartphones. Our main contributions can be summarized as follows:

- We identify the phenomenon of KVCache distribution shift in long-sequence LLM decoding, which degrades both the accuracy and efficiency of KVCache retrieval.
- We develop three core techniques for adaptive KVCache management that integrate algorithmic innovations with system-level optimizations, overcoming the constraints of smartphone memory hierarchies.
- We conduct extensive evaluations across various LLMs and mobile hardware, demonstrating substantial improvements over state-of-the-art solutions.



**Figure 2.** Two-phase KVCache retrieval process. Offloading KVCache from memory to storage mitigates memory bottlenecks, but shifts the critical bottleneck to I/O operations.

## 2 Background and Motivation

### 2.1 KVCache Retrieval in LLM Inference

Almost all state-of-the-art LLMs adopt a transformer-based architecture that generates outputs in an autoregressive manner. During LLM inference, tokens are generated sequentially, with each new token attending not only to the most recent hidden states but also to all preceding tokens in the sequence. To avoid recomputing the hidden states of earlier tokens, the model typically stores the intermediate key and value vectors from each attention layer in a dedicated KVCache.

While the KVCache greatly reduces computation during inference, it incurs a substantial memory footprint, particularly in long-context scenarios where the cache must grow linearly with sequence length. As shown in Table 1, the KVCache exceeds even the model weights in size at longer context, contributing 64%-70% of the total memory footprint. To address this, recent studies [10, 34, 50] explore the inherent **Contextual Sparsity** of attention: for computing the next token, only a subset of the stored keys and values significantly contribute to the attention output, and the relevant subset varies dynamically with individual tokens.

This sparsity property enables **Targeted Retrieval** on the KVCache [30, 31, 55, 60], where only the most relevant

KV entries for a given query are accessed for processing, analogous to information retrieval in database systems [3, 23, 36]. Specifically, the KVCache can be offloaded to external storage rather than maintained entirely in main memory, with relevant entries transferred back on demand. Figure 2 illustrates a typical two-phase KVCache retrieval process:

- (1) **Prefill Phase:** The KVCache is partitioned into clusters based on embedding similarity and stored in external storage. Each cluster is assigned a representative to facilitate indexing. These clusters then serve as the fundamental retrieval units, balancing accuracy and efficiency in subsequent retrieval.
- (2) **Decode Phase:** Given a newly generated query, the top- $k$  relevant KV clusters are identified by comparing the query against the cluster representatives. This paradigm effectively mitigates the KVCache memory bottleneck, particularly on resource-constrained devices such as smartphones.

### 2.2 Two-Tier Memory Hierarchy on Smartphones

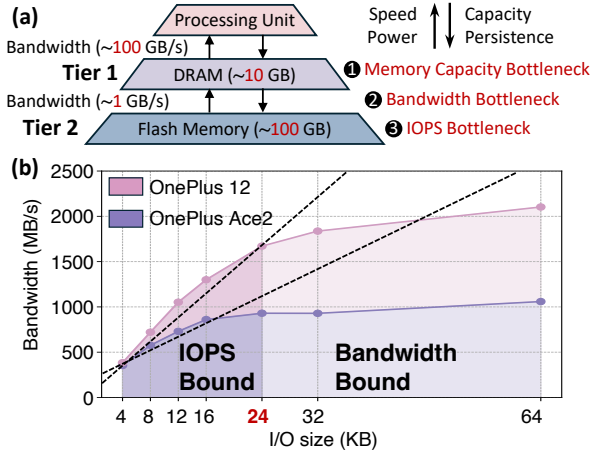
Mobile Devices such as smartphones typically employ a two-tier memory hierarchy, as illustrated in Figure 3(a). The first tier is **Dynamic Random-Access Memory (DRAM)**, which provides fast access and supports direct computation. Due to the cost and power constraints, the size of DRAM is relatively small, typically only a few gigabytes (GB). The second tier is **Flash Memory**, primarily used for external storage and commonly accessed through universal flash storage (UFS) [22]. Leveraging NAND flash provides much larger capacity than DRAM at a lower cost, with scalability reaching terabyte (TB) levels. However, since flash memory lacks compute capability, data must first be transferred into DRAM before computation can be performed. Our comprehensive evaluation shows that this asymmetric two-tier hierarchy introduces three key challenges for KVCache retrieval on smartphones:

- (1) **Bandwidth.** Retrieval-based paradigm requires transferring relevant KVCache from external storage into memory at each inference step. Consequently, the I/O bandwidth between flash and DRAM becomes a fundamental constraint. While DRAM can sustain hundreds of GB/s of throughput, the bandwidth of flash memory remains relatively low (e.g., 2.9 GB/s per lane in the latest UFS 4.0). Consequently, I/O transfers often dominate inference latency, with data transfer overhead emerging as a primary performance bottleneck.
- (2) **IOPS.** Unlike server-grade storage such as NVMe, UFS on smartphones typically supports only a shallow command queue (e.g., 32 entries). This constraint reduces the achievable IOPS for flash reads and can even prevent full utilization of the available I/O bandwidth. As illustrated in Figure 3(b), the read bandwidth increases with the size of continuous I/O, since larger continuous reads can be issued by a single read operation. Specifically, when the continuous I/O size is below 24 KB, the bandwidth scales almost linearly with the I/O size, indicating that these reads are primarily IOPS-bound.
- (3) **Memory Capacity.** The limited capacity of DRAM further exacerbates the above challenges. Since the KVCache

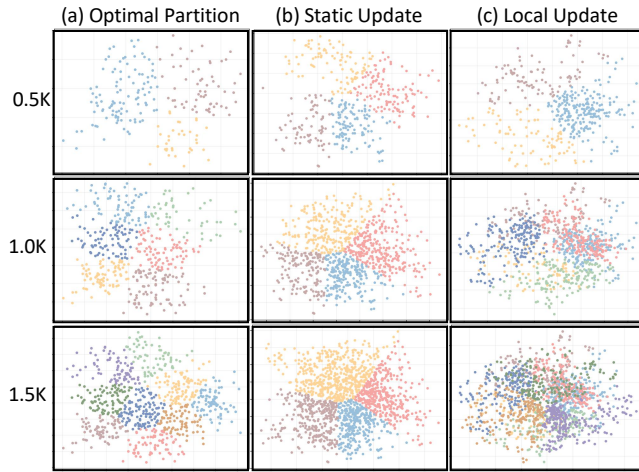
**Table 1.** KVCache memory footprint (relative to 8-bit model weights) of Llama3.2 [37] across sequence lengths  $s$ .

| Model           | Weights | $s = 8K$ | $s = 16K$ | $s = 32K$ | $s = 64K$ |
|-----------------|---------|----------|-----------|-----------|-----------|
| Llama3.2-1B     | 1.2 GB  | 0.3 GB   | 0.5 GB    | 1.1 GB    | 2.1 GB    |
| KVCache/Weights | -       | 25.0%    | 41.7%     | 91.7%     | 175.0%    |
| Llama3.2-3B     | 3.2 GB  | 0.9 GB   | 1.9 GB    | 3.8 GB    | 7.5 GB    |
| KVCache/Weights | -       | 28.1%    | 59.4%     | 118.8%    | 234.4%    |





**Figure 3.** (a) Two-tier memory hierarchy on smartphones (DRAM and flash), highlighting three key bottlenecks. (b) UFS bandwidth under various continuous I/O sizes, where the limiting factor shifts from IOPS to bandwidth capacity.



**Figure 4.** Visualization of KVCache distribution shift during decoding using PCA [54], with Qwen2.5-3B evaluated on the MMLU dataset [17]. Different colors denote distinct clusters.

grows linearly with sequence length, it can quickly exceed available memory even for moderate contexts. Cluster-based KVCache organization amplifies memory demand, as entire clusters must be managed as retrieval units. On one hand, this constraint forces frequent offloading to flash and intensifies dependence on bandwidth- and IOPS-constrained I/O. On the other hand, it underscores the need for efficient cache design to maximize the utility of scarce DRAM resources.

### 2.3 Motivation: KVCache Distribution Shift

The cluster-based organization of the KVCache improves retrieval efficiency by enabling queries to be compared only against cluster representatives rather than individual entries.

However, the effectiveness of this approach critically depends on clustering quality. As the similarity among entries within a cluster decreases, its representative becomes less informative. This misalignment can lead to degraded inference accuracy when essential entries are excluded, and wasted I/O bandwidth when redundant entries are retrieved.

Theoretically, the optimal partition of the KVCache can be achieved through global clustering over the entire cache. However, because global clustering is both computationally and memory-intensive, it is typically performed only once during the prefill phase for initial partition. During decoding, newly generated KV entries are then managed using one of two strategies: (1) **Static Update**, which greedily assigns each new entry to its nearest existing cluster [18, 60], or (2) **Local Update**, which re-clusters new entries separately from the existing KVCache [31, 45]. Guided by the principle of minimally modifying the initial partition, both strategies introduce little update overhead and remain effective when the prefill proportion dominates the inference context.

In this paper, we observe an unexpected shift in KVCache distribution during decoding. As shown in Figure 4(a), the optimal partition (i.e., global clustering over the entire KVCache) progressively diverges from the initial partition as the decode length grows. Furthermore, existing strategies fail to adapt effectively to this shift. (1) Figure 4(b) shows that static update enlarges cluster sizes and raises intra-cluster variance. Under the KVCache distribution shift, newly generated entries may be distant from all existing clusters, making greedy appending increasingly ineffective. (2) Figure 4(c) shows that local update fragments the KVCache by generating many small clusters. Without global information, these clusters may fail to align with the overall KVCache distribution. This fragmentation also results in fine-grained read accesses, which are more likely to hit IOPS limitations.

This phenomenon is particularly pronounced in **Long-Sequence Decoding** scenarios, where the decode length far exceeds the prefill length. With the growing demand for human-like reasoning [9, 15, 21], multi-turn conversations [5, 46, 63], and long-form response generation [27, 40, 59], the need for more adaptive and efficient KVCache management has become critical, yet remains largely underexplored.

## 3 System Overview

We propose DYNKV, an efficient approach that enables long-sequence LLM decoding on smartphones with adaptive KVCache management. Unlike prior work that primarily targets KVCache memory bottlenecks, DYNKV highlights the dynamic shifts in KVCache distribution during decoding. By combining algorithmic innovations with system-level design, DYNKV achieves an effective balance between accuracy and efficiency. Figure 5 shows an overview of DYNKV.

**Migration-Free Cluster Adaptation (§ 4).** DYNKV begins with an algorithm that adaptively updates KVCache clusters

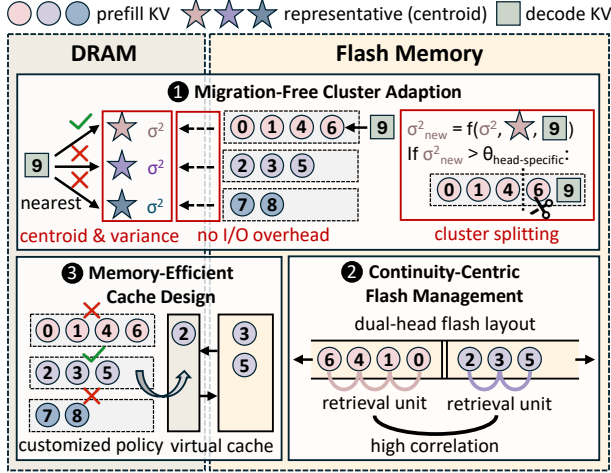


Figure 5. Overview of DYNAKV.

with minimal data migration. Based on initial partition, DYNAKV tracks the intra-cluster variance as an effectiveness metric and triggers a split once it exceeds a head-specific threshold. Particularly, splits are applied only to in-memory clusters and deferred for disk-resident clusters, eliminating extra I/O overhead while preserving correctness.

**Continuity-Centric Flash Management (§ 5).** Given the IOPS bottlenecks, DYNAKV optimizes the KVCache placement on flash to facilitate continuous read access. Specifically, DYNAKV leverages the co-retrieval property of KVCache by storing all entries within a cluster as well as highly-correlated clusters contiguously. DYNAKV further adopts a dual-head layout that places a pair of clusters back-to-back, enabling frequent cluster updates without costly data permutation.

**Memory-Efficient Cache Design (§ 6).** DYNAKV designs a virtual cache space to mitigate the limited memory capacity. By leveraging idle bandwidth during computation, a portion of cached entries is offloaded to flash and seamlessly fetched back on demand. DYNAKV also customizes cache replacement policies to align with cluster-based retrieval. By exploiting the structure and dynamics of retrieval patterns, small and recently updated ones are assigned higher priority.

## 4 Migration-Free Cluster Adaptation

### 4.1 Problem Formulation

**Long-Sequence Decoding.** We consider a typical two-phase process for LLM inference. In the prefill phase, the model processes an input prompt of length  $L_0$  and generates the initial KVCache  $\mathcal{K}_0$ :

$$\mathcal{K}_0 = \{\mathbf{k}_{\text{init}}^{(1)}, \mathbf{k}_{\text{init}}^{(2)}, \dots, \mathbf{k}_{\text{init}}^{(L_0)}\} \quad (1)$$

where each  $\mathbf{k}_i$  denotes a key-value (KV) vector pair.

Following the prefill phase, the model enters the decoding phase, where tokens are generated autoregressively. At each step  $t$ , the model produces a new KV pair  $\mathbf{k}_{\text{new}}^{(t)}$ . The

accumulated KVCache up to step  $t$  can thus be expressed as:

$$\mathcal{K}_t = \mathcal{K}_0 \cup \{\mathbf{k}_{\text{new}}^{(1)}, \mathbf{k}_{\text{new}}^{(2)}, \dots, \mathbf{k}_{\text{new}}^{(t)}\} \quad (2)$$

In this paper, we focus on long-sequence decoding, which is prevalent in scenarios such as enhanced reasoning and multi-turn dialogues. In these cases, the total number of decoded tokens substantially exceeds the initial prefill length. **Cluster-Based Retrieval.** As discussed in Section 2.1, KV-Cache  $\mathcal{K}_t$  is partitioned into  $M_t$  disjoint clusters using algorithms such as K-Means and stored on flash:

$$\mathcal{P}_t = \{C_1^{(t)}, C_2^{(t)}, \dots, C_{M_t}^{(t)}\}, \quad \bigcup_{j=1}^{M_t} C_j^{(t)} = \mathcal{K}_t \quad (3)$$

where each  $C_j^{(t)}$  denotes a KVCache cluster.

For each cluster  $C_j^{(t)}$ , a representative  $r_j^{(t)}$  is selected, typically chosen as the centroid of the cluster:

$$r_j^{(t)} = \frac{1}{|C_j^{(t)}|} \sum_{\mathbf{k} \in C_j^{(t)}} \mathbf{k}, \quad (4)$$

where  $|C_j^{(t)}|$  denotes the number of KV pairs in cluster  $C_j^{(t)}$ .

These representatives are stored in DRAM and serve as compact proxies for their respective clusters during retrieval. For a given query  $q$ , the similarity scores between  $q$  and all representatives are computed. The top- $k$  clusters are then selected, referred to as the active set  $\mathcal{P}_{\text{req}}^{(t)} \subseteq \mathcal{P}_t$ , transferred back from flash to DRAM for further computation.

**Optimization Objectives.** The presence of KVCache distribution shift indicates that the optimal partition  $\mathcal{P}_t$  at decoding step  $t$  diverges increasingly from the initial partition  $\mathcal{P}_0$ . The goal of DYNAKV is to design an adaptive clustering algorithm that jointly optimizes the following three objectives:

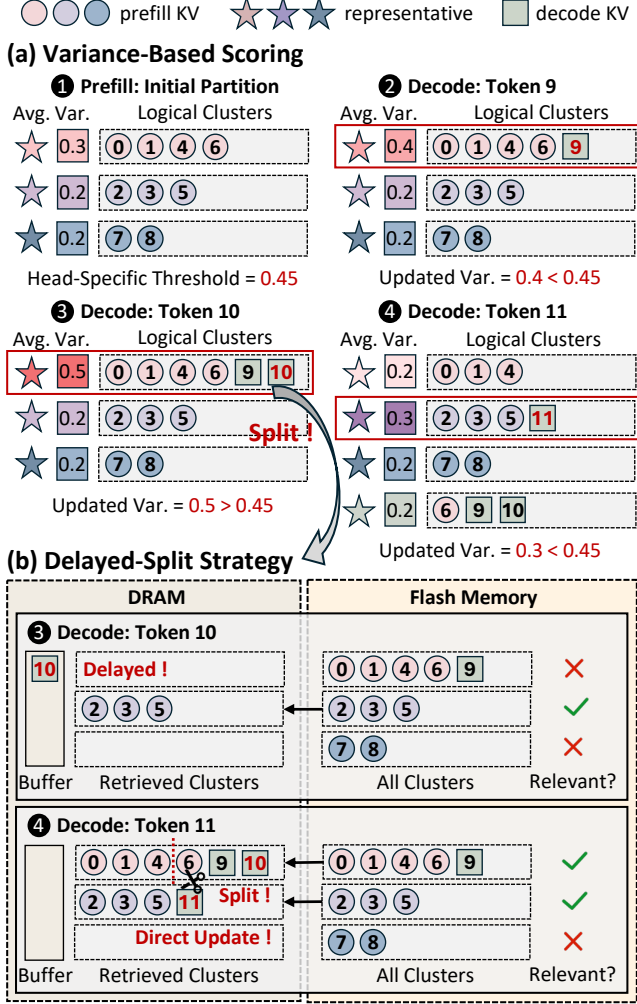
(1) **Accurate Coverage of Required Entries.** Let  $\mathcal{K}_{\text{req}}^{(t)} \subseteq \mathcal{K}_t$  denote the exact set of KV entries required at decoding step  $t$ . Excess entries waste I/O bandwidth, while insufficient entries degrade inference accuracy. Therefore, the selected clusters  $\mathcal{P}_{\text{req}}^{(t)}$  should approximate  $\mathcal{K}_{\text{req}}^{(t)}$  as closely as possible:

$$\bigcup_{C \in \mathcal{P}_{\text{req}}^{(t)}} C \approx \mathcal{K}_{\text{req}}^{(t)} \quad (5)$$

(2) **Coarse Granularity of Cluster Partition.** Subject to accuracy, the cluster partition should be as coarse as possible. Fewer clusters reduce the number of representatives, improving retrieval efficiency. Also, overly fragmented clusters leave the system IOPS-bound, leading to I/O inefficiency:

$$\text{minimize } |\mathcal{P}_{\text{req}}^{(t)}| \quad (6)$$

(3) **Low Overhead from Cluster Updates.** For efficiency, runtime adaptation of cluster partitions should introduce only minimal latency. Since I/O is the primary bottleneck



**Figure 6.** Example of adaptive KVCache management in DYNKV. (a) Intra-cluster variance is monitored to maintain clustering effectiveness. (b) Cluster splitting is integrated into the retrieval workflow, avoiding extra I/O overhead.

on smartphones, the crucial requirement is to keep the additional I/O load from cluster updates as low as possible:

$$\text{minimize Load}_{\text{io}}(\mathcal{P}_t, \mathcal{P}_{\text{req}}^{(t)}) \quad (7)$$

## 4.2 Self-Adaptive Clustering Algorithm

As illustrated in Figure 6, DYNKV employs a self-adaptive clustering algorithm to jointly satisfy these objectives. The algorithm is composed of two key components: a variance-based scoring mechanism that preserves clustering effectiveness, and a delayed-split strategy that ensures efficient cluster updates with minimal I/O overhead.

**Variance-Based Scoring.** As outlined in Algorithm 1, the process begins with the initial cluster partition  $\mathcal{P}_0$  during the prefill phase. For each newly generated KV entry  $\mathbf{k}_{\text{new}}^{(t)}$ ,

### Algorithm 1: Adaptive KVCache Management

**Input:** Prefill KVCache  $\mathcal{K}_0$ , head-specific thresholds  $\{\tau_h\}$ , buffer budget  $B_{\text{max}}$ .

// Prefill: build the initial partition

Initial partition  $\mathcal{P}_0 \leftarrow \text{NEWCLUSTER}(\mathcal{K}_0)$

**foreach** cluster  $C_j \in \mathcal{P}_0$  **do**

    Initialize cluster size  $n_j$ , centroid  $r_j$ , variance  $\sigma_j^2$

    Set buffer  $\text{Buf}_j \leftarrow \emptyset$  and flag  $f_j \leftarrow \emptyset$

// Decode: adaptively update partition

**for** decoding step  $t = 1, 2, \dots$  **do**

    Load top- $k$  clusters  $\mathcal{P}_{\text{req}}^{(t)}$  by query-centroid similarity

    Assign new KV entry  $\mathbf{k}_{\text{new}}^{(t)}$  to its nearest cluster index  $j$

    Update  $\sigma_j^2 \leftarrow \text{UPDATEVAR}(n_j, r_j, \sigma_j^2, \mathbf{k}_{\text{new}}^{(t)})$

**if**  $\sigma_j^2 \leq \tau_h$  **then**

        Update  $C_j \leftarrow \text{ADDEENTRY}(C_j, \mathbf{k}_{\text{new}}^{(t)})$

        Update  $n_j, r_j \leftarrow \text{UPDATESTATS}(n_j, r_j, C_j)$

**else if**  $C_j \in \mathcal{P}_{\text{req}}^{(t)}$  **then**

        Update  $\mathcal{P}_t \leftarrow \text{SPLITCLUSTER}(\mathcal{P}_t, C_j, \mathbf{k}_{\text{new}}^{(t)})$

        Update  $n_j, r_j \leftarrow \text{UPDATESTATS}(n_j, r_j, C_j)$

**else**

        Buffer  $\mathbf{k}_{\text{new}}^{(t)}$  in  $\text{Buf}_j$  and mark  $f_j \leftarrow \text{SPLIT}$

    // Delayed cluster splitting

**foreach**  $C_j \in \mathcal{P}_{\text{req}}^{(t)}$  with  $f_j = \text{SPLIT}$  **do**

        Update  $\mathcal{P}_t \leftarrow \text{SPLITCLUSTER}(\mathcal{P}_t, C_j, \text{Buf}_j)$

        Reset  $\text{Buf}_j, f_j \leftarrow \emptyset$

**if**  $\sum_j |\text{Buf}_j| \geq B_{\text{max}}$  **then**

        Load cluster  $C_{j^\dagger}$  with  $j^\dagger \leftarrow \arg \max_j |\text{Buf}_j|$

        Update  $\mathcal{P}_t \leftarrow \text{SPLITCLUSTER}(\mathcal{P}_t, C_{j^\dagger}, \text{Buf}_{j^\dagger})$

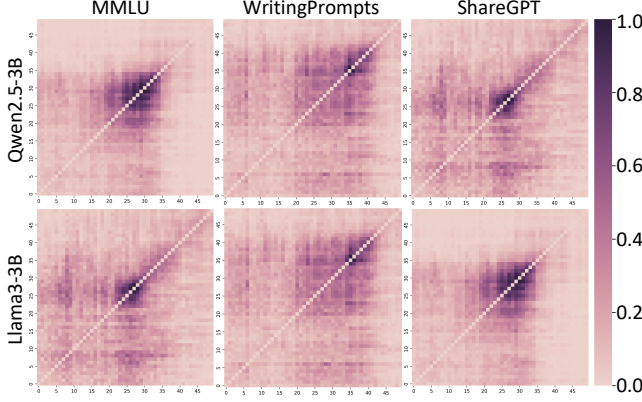
        Reset  $\text{Buf}_{j^\dagger}, f_{j^\dagger} \leftarrow \emptyset$

DYNKV compares it against existing cluster representatives and assigns it to the nearest cluster based on feature similarity. Importantly, DYNKV maintains and updates the variance of each cluster, which can be quickly computed from the centroid, the variance, and the new entry.

These variances serve as effectiveness scores for the clusters, capturing the similarity among KV entries within each cluster. If the variance remains below an acceptable threshold, the cluster is updated to incorporate the new entry. When the variance exceeds the threshold, indicating that the cluster is no longer cohesive, the cluster is flagged for splitting. Since cluster sizes are relatively small (naturally bounded by the variance), the computational cost of splitting is negligible. To account for variations in KVCache distributions across attention heads, the thresholds are defined in a head-specific manner. This scoring mechanism preserves clustering effectiveness (*Objective 1*) while keeping cluster granularity as coarse as possible (*Objective 2*).

**Delayed-Split Strategy.** Because flash memory lacks compute capability, cluster splitting must be performed in DRAM,





**Figure 7.** Visualization of inter-cluster correlation across models and datasets. Each matrix element  $(i, j)$  denotes the frequency with which cluster  $C_i$  and  $C_j$  are co-retrieved.

where a local clustering algorithm is executed. In other words, if a cluster resides in flash memory, it must first be transferred to DRAM. To reduce the I/O overhead caused by frequent splits, DYNKV adopts a delayed-split strategy.

Specifically, DYNKV performs actual cluster splitting only when the flagged cluster has been retrieved into DRAM. If the flagged cluster is not yet in memory, the split is deferred. Instead, newly added KV entries belonging to the flagged cluster are stored in a memory buffer until the cluster is eventually loaded. To control memory complexity, a maximum buffer size is pre-defined. Once the buffer is full, DYNKV prioritizes uploading the cluster with the largest number of buffered entries and performs the split operation. Because all deferred entries are guaranteed to be incorporated once the cluster is loaded, this strategy preserves correctness while greatly reducing I/O overhead (*Objective 3*).

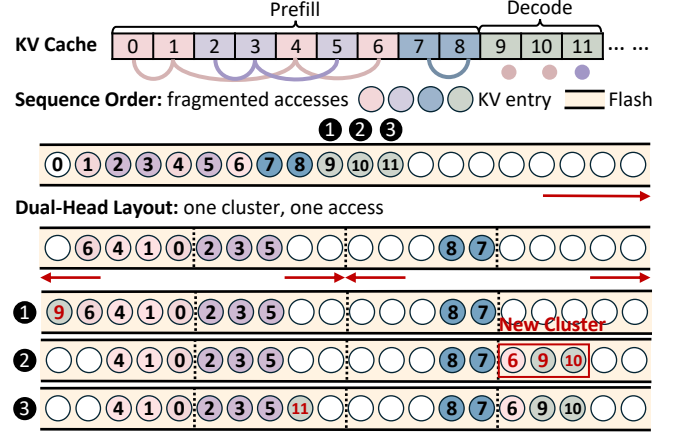
## 5 Continuity-Centric Flash Management

### 5.1 Correlation-Aware Placement

Given the sparse nature of KVCache retrieval, the KV entries within a cluster are scattered according to strict sequence order. Therefore, transferring a KVCache cluster from flash incurs numerous small-grained read accesses, leaving the I/O operations heavily IOPS-bound on smartphones. To address this issue, DYNKV jointly exploits intra- and inter-cluster correlations to guide data placement on flash, consolidating fragmented accesses and improving I/O efficiency.

**Intra-Cluster Correlation.** Since clusters serve as the fundamental retrieval units, all KV entries within a cluster are fetched from flash to DRAM together. To exploit this property, DYNKV adopts a cluster-based placement strategy, arranging KV entries within each cluster continuously on flash rather than preserving their strict sequence order.

**Inter-Cluster Correlation.** DYNKV further exploits the frequent co-retrieved patterns observed across clusters. As



**Figure 8.** Dual-Head flash layout in DYNKV. Unlike strict sequence order, this layout supports continuous flash access and frequent cluster updates with minimal storage overhead.

shown in Figure 7, certain clusters are repeatedly accessed together during inference, exhibiting strong inter-cluster correlation. To take advantage of this property, DYNKV constructs an adjacency matrix that records retrieval frequencies between clusters. This step is performed only once before decoding, using the initial input as a reference. By estimating probability using frequency  $f$ , DYNKV computes the co-retrieval probability of cluster  $C_i$  and  $C_j$  as follows:

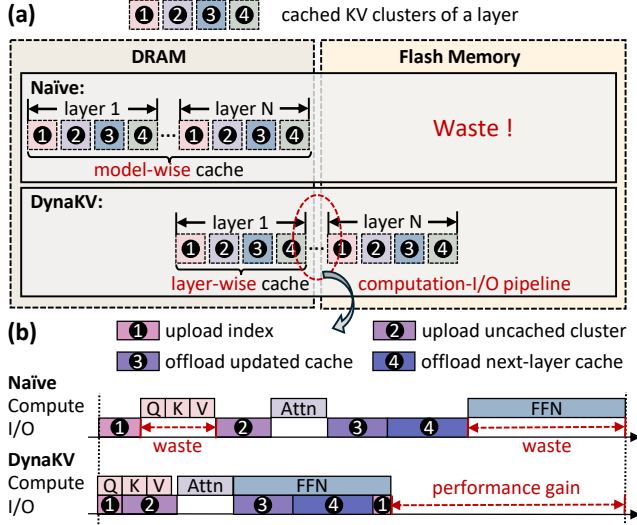
$$P(ij) = \frac{f(C_i, C_j)}{\sum_{k=1}^{M_0} \sum_{l=1}^{M_0} f(C_k, C_l)} \quad (8)$$

where  $M_0$  denotes the number of initial clusters. Guided by these patterns, DYNKV co-locates highly correlated clusters adjacently in flash. This placement not only preserves intra-cluster locality but also enables multiple clusters to be retrieved through a single sequential access.

### 5.2 Dual-Head Cluster Layout

While correlation-aware placement substantially alleviates the IOPS bottleneck of cluster transfers on smartphones, it introduces a new challenge for efficient cluster updates. Due to the adaptive clustering algorithm in DYNKV, a cluster may either append new entries or be split into two new clusters. Under a strictly contiguous layout, these updates would require costly data movement on flash, significantly degrading inference performance. To overcome this, DYNKV introduces a dual-head cluster layout, placing two correlated clusters in a back-to-back manner. In addition, a page-aligned buffer is employed to eliminate write amplification during updates, further improving the efficiency of DYNKV.

**Back-to-Back Manner.** As illustrated in Figure 8, DYNKV allocates a dedicated memory pool on flash for each pair of co-retrieved clusters, with pool size set to twice the maximum cluster size. The two clusters are placed back-to-back,



**Figure 9.** (a) A large virtual cache space is abstracted across DRAM and flash. (b) The additional data transfer overhead is effectively hidden by pipelining with computation.

growing inward from opposite ends of the pool. This layout enables new entries to be appended seamlessly to either cluster without requiring expensive data permutations. When a cluster is split, one of the resulting sub-clusters continues to grow from the original head, while the other is migrated to a newly allocated pool, paired with another cluster (or left with an empty partner slot). In this way, DYNKv preserves co-retrieval benefits while avoiding costly data relocation.

**Page-Aligned Buffer.** Flash writes must align to page boundaries, and small updates often trigger partial-page rewrites, amplifying write traffic. To improve the efficiency of cluster updates, DYNKv employs a page-aligned buffer in memory that aggregates new entries until a full page can be written. Buffers are maintained at the cluster level to isolate updates across different clusters. To conserve memory, buffers are allocated only for hot clusters with frequent updates, while cold clusters issue direct writes when necessary. This design eliminates write amplification during appends and splits while preserving sequential write patterns.

## 6 Memory-Efficient Cache Design

### 6.1 Pipeline-Driven Space Virtualization

Caching frequently accessed KV entries (including cluster representatives) in memory is critical for reducing I/O overhead, but the limited DRAM capacity on smartphones makes it infeasible to store the full KVCache. To address this limitation, DYNKv introduces a pipeline-driven virtualization mechanism that leverages both DRAM and flash, overlapping cache transfers with computation to hide I/O latency. By effectively virtualizing DRAM capacity, DYNKv provides

**Table 2.** Probability of updated clusters being retrieved on MMLU [17] across LLMs and subsequent decoding steps  $t$ .

| Step (tokens)     | $t = 1$ | $t = 2$ | $t = 4$ | $t = 8$ | $t = 16$ |
|-------------------|---------|---------|---------|---------|----------|
| Qwen2.5-1.5B [41] | 42.4%   | 47.9%   | 53.7%   | 60.5%   | 67.0%    |
| Qwen2.5-3B [41]   | 59.4%   | 65.5%   | 70.9%   | 76.3%   | 80.1%    |
| Llama3.2-1B [37]  | 36.5%   | 42.3%   | 48.7%   | 54.1%   | 61.4%    |
| Llama3.2-3B [37]  | 30.1%   | 36.3%   | 42.8%   | 49.6%   | 54.9%    |

the abstraction of a large cache space while physically occupying a small memory footprint. As shown in Figure 9, this virtualization is realized with two complementary pipelines: **Pipeline 1: Projection (QKV).** In KVCache retrieval, cluster representatives are necessary for determining which clusters to access and are typically cached persistently in memory. DYNKv eliminates this static memory cost by storing representatives on flash and pipelining their transfer with query (Q) projection. Since representatives are much smaller than full KV entries, most of the transfer time is effectively hidden. Once the representatives are resident in memory, the subsequent KV retrieval can proceed without delay. The transfer of retrieved KV entries is further overlapped with the computation of key-value (KV) projection and  $Q$ -K similarity.

**Pipeline 2: Attention and FFN.** Beyond cluster representatives, frequently retrieved KV clusters are also typically cached. Because their size is much larger, overlapping their transfer requires more computation time to fully hide I/O latency. DYNKv addresses this by pipelining cache transfers with computation during the attention and feed-forward network (FFN) stages. While the current layer executes, its KVCache is offloaded to flash, and the KVCache of the next layer is prefetched into DRAM. This layer-wise scheduling ensures that the cache of each layer is available exactly when needed, while unused layers remain virtualized on flash. As a result, memory consumption is bounded at the granularity of a single layer rather than the entire model, greatly improving efficiency under tight DRAM budgets.

### 6.2 Cluster-Aligned Replacement Policy

Beyond expanding cache space, DYNKv further optimizes its cache replacement strategy to maximize utilization under strict memory constraints. Conventional cache replacement policies [26, 28, 39] (e.g., LRU, LFU) are designed for entry-level access patterns and do not align well with the semantics of cluster-based retrieval. When applied directly to DYNKv, such policies often lead to excessive memory consumption or degraded hit rates. To overcome these limitations, DYNKv introduces a cluster-aligned cache replacement policy that explicitly incorporates the structure and dynamics of cluster-based KV retrieval. The design is guided by two principles: **Principle 1: Prioritize Small Clusters.** Caching large clusters is inefficient for two reasons. First, large clusters occupy disproportionate memory capacity, yet their performance



benefit is limited since they already benefit from sequential placement on flash. Second, large clusters are more likely to be split under the adaptive clustering algorithm, which undermines cache stability and can quickly invalidate cached data. In contrast, small clusters consume far less memory and yield higher marginal gains in access efficiency. By prioritizing small clusters, DYNKV reduces memory overhead and maximizes the performance benefit of caching.

**Principle 2: Retain Updated Clusters.** DYNKV leverages the unique access patterns of KVCache clusters to further reduce I/O transfers. As presented in Table 2, clusters that have been recently updated—either by appending new entries or by being split into sub-clusters—tend to exhibit stronger short-term relevance. To exploit this property, DYNKV reserves dedicated cache space for updated clusters and retains them in memory regardless of general replacement decisions. This targeted retention capitalizes on the unique temporal locality of KVCache, preventing premature eviction of high-utility clusters and increasing cache hit rates.

## 7 Implementation and Discussion

**System Implementation.** We implement DYNKV as an end-to-end system built on *llama.cpp* [14], the most widely used inference framework for mobile devices. Following prior studies that demonstrate the better performance of CPUs over NPUs for decoding on smartphones [49, 53], we adopt the CPU as the primary backend. To support the core techniques of DYNKV, we extend the runtime by rewriting the KVCache swapping, flash loading, and cache management modules. For high-performance asynchronous data transfers, we leverage the Linux kernel interface *io\_uring* [4]. To decouple I/O from computation, we create a dedicated data-loading thread using *ggml\_thread\_create*, and synchronize it with computation through atomic semaphores. On the computation side, we employ an optimized K-Means algorithm for multi-threaded execution and are robust to empty-cluster cases. We also modify the original *flash\_attention* kernel to handle potential numerical overflow errors introduced by reordered KV computations. Overall, DYNKV introduces 17 new operators and over 5,500 lines of new C++ code.

**Discussion 1: Long-Sequence Prefill.** DYNKV is fully compatible with long-sequence prefill workloads. Similar to long-sequence decoding, prefill also involves long-context processing and therefore requires efficient KVCache retrieval on smartphones. However, since the prefill phase dominates the overall sequence length, the impact of KVCache distribution shifts is relatively limited. In such cases, DYNKV behaves comparably to existing cluster-based approaches, with its additional mechanisms introducing no extra overhead. Importantly, although the KVCache generated during prefill is eventually offloaded, the prefill stage itself can incur significant peak memory usage, an issue largely overlooked

**Table 3.** Smartphone hardware configurations.

| Device           | SoC                 | DRAM | Flash | Storage |
|------------------|---------------------|------|-------|---------|
| OnePlus Ace5 Pro | Snapdragon 8 Elite  | 16GB | 512GB | UFS4.0  |
| OnePlus 12       | Snapdragon 8 Gen 3  | 24GB | 1TB   | UFS4.0  |
| OnePlus Ace3     | Snapdragon 8 Gen 2  | 16GB | 512GB | UFS4.0  |
| OnePlus Ace2     | Snapdragon 8+ Gen 1 | 16GB | 512GB | UFS3.1  |

**Table 4.** Model configurations.

| Model             | Size | Attention | Hidden Size | Group Size |
|-------------------|------|-----------|-------------|------------|
| Qwen(2.5)-S [41]  | 1.5B | GQA       | 1,536       | 12/2=6     |
| Qwen(2.5)-M [41]  | 3.1B | GQA       | 2,048       | 16/2=8     |
| Llama(3.2)-S [37] | 1.2B | GQA       | 2,048       | 32/8=4     |
| Llama(3.2)-M [37] | 3.2B | GQA       | 3,072       | 24/8=3     |
| Mistral [25]      | 7.8B | GQA       | 4,096       | 32/8=4     |

in prior work. In contrast, the adaptive KVCache management of DYNKV can be applied continuously throughout prefill, reducing memory peak while keeping accuracy.

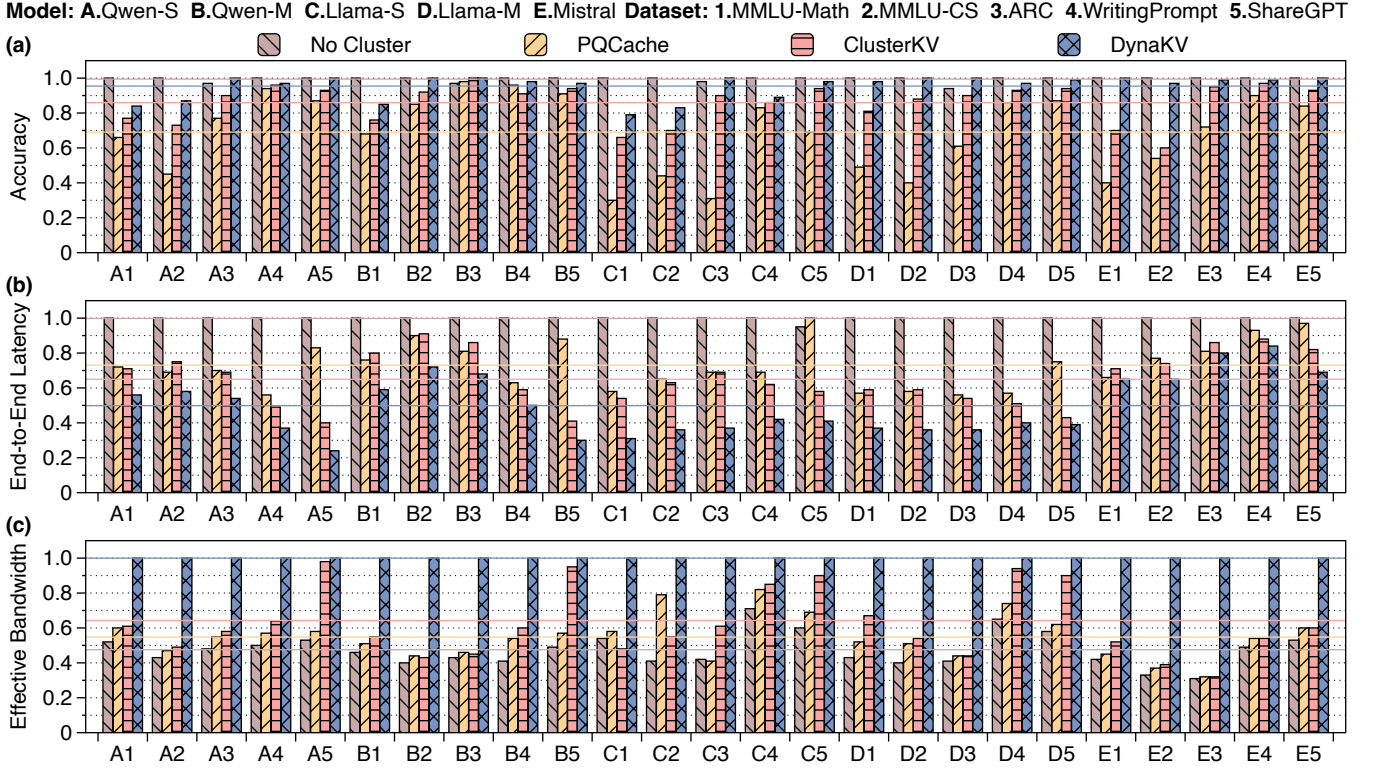
**Discussion 2: CPU-GPU Platform.** Although DYNKV primarily targets CPU-only execution on smartphones, the design naturally extends to heterogeneous CPU-GPU platforms. In this setting, GPU memory acts as the active working space for inference, while CPU memory serves as a staging buffer for KVCache offloading. Similarly, the bandwidth between CPU and GPU memory can become a critical bottleneck, making adaptive KVCache management in DYNKV equally important. In addition, lightweight operations (e.g., clustering updates or variance tracking) can be executed directly on the CPU without I/O communication. This division of workloads makes the integration of DYNKV into heterogeneous CPU-GPU platforms even more straightforward. When CPU storage is also incorporated, DYNKV can finally generalize into a three-tier memory hierarchy, efficiently spanning CPU storage, CPU memory, and GPU memory.

## 8 Evaluation

### 8.1 Experimental Setup

**Hardware.** We conduct evaluations on four representative smartphones, as detailed in Table 3. These devices feature varying SoC configurations, DRAM and flash capacities, and UFS versions, spanning from low-tier to high-tier hardware. All experiments are conducted on Termux [48], an Android shell application, to ensure compatibility across platforms.

**Models.** We choose five widely adopted on-device LLMs for evaluation, as listed in Table 4. These models differ in both architecture and parameter scale. Following common practice, all models adopt group-query attention (GQA) [1] with varying group sizes, where larger groups imply greater KVCache reuse. All experiments are conducted in FP8 precision.



**Figure 10.** Overall performance (normalized) across various models and datasets on OnePlus 12. Lines indicate average values.

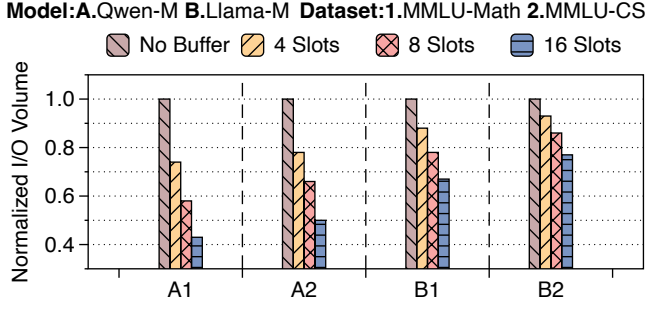
**Datasets.** We evaluate DYNKV across a diverse set of datasets covering three representative long-sequence decoding scenarios (1) *Reasoning*. We use mathematics and computer science tasks from MMLU[17], along with the AI2 Reasoning Challenge (ARC)[8], a benchmark for model reasoning ability. (2) *Story Generation*. We use Writing Prompts [12], a large-scale dataset for long-form creative story generation. (3) *Multi-turn Conversation*. We use ShareGPT [43], which contains real multi-turn conversations collected from ChatGPT. **Baselines.** We compare DYNKV against two state-of-the-art KVCache retrieval methods: PQCache [60] and ClusterKV [31]. These represent the two dominant directions for cluster adaptation: *static update* and *local update*, respectively. In addition, we include a naive baseline that performs KVCache retrieval without clustering, to demonstrate the necessity and effectiveness of cluster-based organization. **Metrics.** We evaluate DYNKV using two primary metrics: accuracy and end-to-end latency. For reasoning tasks, accuracy is measured directly on labels. For story generation and multi-turn conversation, we follow common practice [64] and use LLM-based evaluation from multiple perspectives. We normalize the values when large discrepancies occur for clarity. All reported results are averaged over 10 trials.

## 8.2 Overall Performance

**Accuracy Evaluation.** Figure 10(a) shows that DYNKV yields the highest accuracy over both cluster-based methods across all models and datasets, approaching the no-cluster baseline (the theoretical upper bound). This demonstrates that DYNKV effectively mitigates the KVCache distribution shift. In contrast, PQCache relies on static updates that fail to adapt to the shift, yielding only 0.69 accuracy on average.

**End-to-End Latency.** Figure 10(b) shows that DYNKV yields average speedup improvements of 2.00 $\times$ , 1.47 $\times$ , 1.31 $\times$  over the no-cluster baseline, PQCache, and ClusterKV, respectively. The no-cluster baseline exhibits the highest latency, underscoring the necessity of clustering. Moreover, the results demonstrate that adaptive KVCache clustering in DYNKV introduces negligible overhead, while system-level I/O optimizations play a dominant role in reducing latency.

**Effective Bandwidth.** Figure 10(c) shows that DYNKV delivers average effective bandwidth improvements of 2.10 $\times$ , 1.83 $\times$ , 1.56 $\times$  over the three baselines. These gains stem from two factors: (1) DYNKV forms more effective clusters that contain fewer redundant KV entries. (2) It alleviates the IOPS bottleneck by reducing fine-grained flash accesses.



**Figure 11.** KVCache transfer volume under varying buffer size (for delayed splitting) across models and datasets.

### 8.3 Ablation Study

**Technique 1: Cluster Adaptation.** Table 5 shows that clusters in DYNKQV exhibit significantly lower variance compared to the baselines. This demonstrates that the variance-based scoring mechanism in DYNKQV effectively preserves clustering quality, which is essential for retrieval accuracy.

Figure 11 further evaluates the KVCache transfer reduction achieved by the delayed-split strategy. By temporarily retaining a small number of KV entries (up to 16) in memory and deferring splitting until the corresponding cluster is retrieved, DYNKQV reduces I/O volume by up to 2.32 $\times$ .

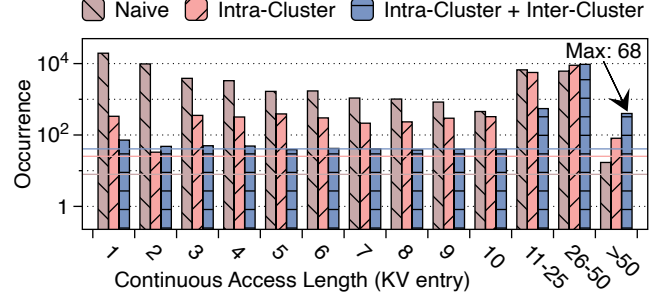
**Technique 2: Flash Management.** Figure 12 demonstrates the effectiveness of both intra-cluster and inter-cluster continuity designs in DYNKQV. Under strict sequence order, the average flash access length is limited to below 7.9 entries. By exploiting cluster correlations, DYNKQV increases continuous access lengths to 25.3 and 40.8, respectively. Notably, the maximum continuous access length reaches 68 entries.

Figure 13 shows the effectiveness of our dual-head flash layout. With only minor overhead in abundant storage, this design drastically reduces costly data permutation. Compared to the naive cluster-by-cluster layout, DYNKQV achieves over  $10^4$  reduction in data movement across both models.

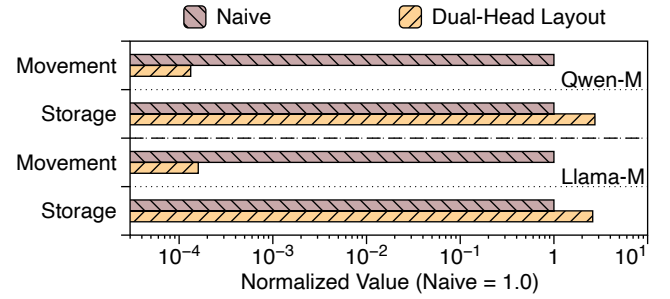
**Technique 3: Cache Design.** Figure 14(a) demonstrates that DYNKQV reduces cache memory usage across different cache ratios. By pipelining data transfer for the next layer with ongoing computation, cache virtualization shifts memory demand from the model level to the layer level, yielding increasing savings as the cache ratio grows (up to 2.22 $\times$ ).

**Table 5.** Cluster variance across models and datasets (A-D: Qwen-S and -M, Llama-S and -M; 1-2: MMLU-Math and -CS).

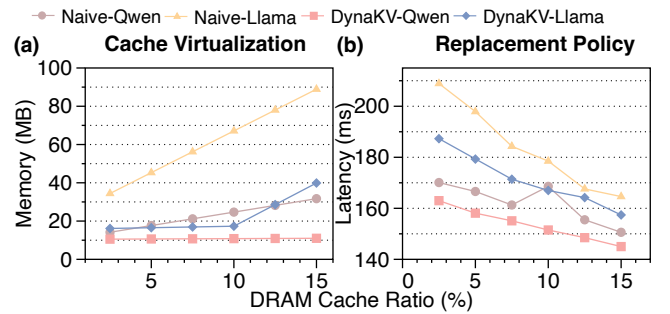
| Cases     | A1    | A2    | B1    | B2    | C1    | C2    | D1    | D2    |
|-----------|-------|-------|-------|-------|-------|-------|-------|-------|
| PQCache   | 206.7 | 198.2 | 211.3 | 214.2 | 146.1 | 154.4 | 218.1 | 201.6 |
| ClusterKV | 122.4 | 125.5 | 124.3 | 127.7 | 95.1  | 94.7  | 129.0 | 132.1 |
| DynaKV    | 119.6 | 122.5 | 118.3 | 125.2 | 94.6  | 90.8  | 123.3 | 123.8 |



**Figure 12.** Statistical information on flash access lengths of Qwen-M on MMLU. Lines indicate average values.



**Figure 13.** Data movement and storage usage with and without the dual-head layout across different models.



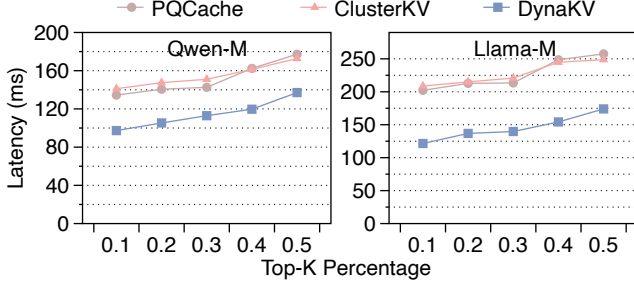
**Figure 14.** (a) Memory usage and (b) End-to-end latency under varying cache ratios with and without specific design.

Figure 14(b) shows that DYNKQV also outperforms the baselines in latency across different cache ratios. Compared with standard LRU, its cluster-aligned replacement policy achieves a higher cache hit rate under the same cache size, reducing I/O transfers and delivering up to 1.12 $\times$  speedups.

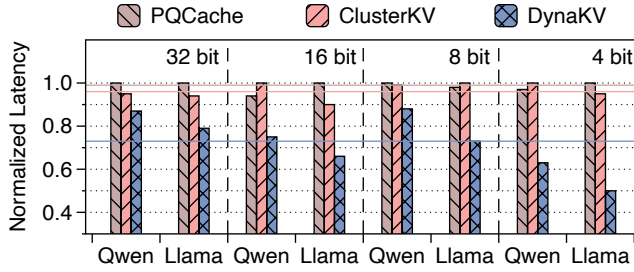
### 8.4 Sensitivity Analysis

**Impact of Decoding Lengths.** We first evaluate the performance of DYNKQV across decoding lengths in long-sequence decoding, as listed in Table 6. The results show that performance scales nearly linearly with sequence length up to 16K, indicating the strong scalability of DYNKQV to long contexts without incurring notable overhead from cluster adaptation.





**Figure 15.** End-to-end latency under varying top- $k$  percentages of KVCache retrieval on MMLU using OnePlus 12.

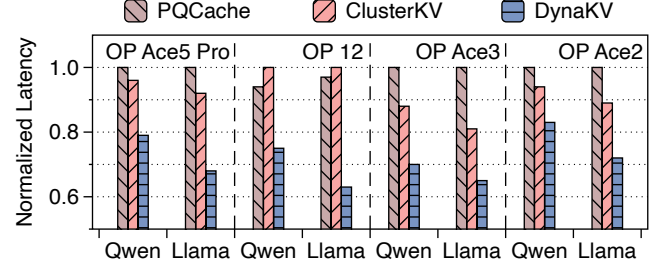


**Figure 16.** End-to-end latency of Qwen-M and Llama-M on MMLU across parameter precisions using OnePlus 12.

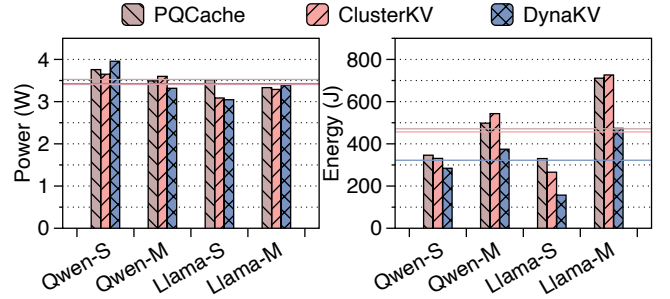
**Impact of Sparsity Levels.** We next evaluate how variations in KVCache sparsity levels affect the performance of DYNKV, as shown in Figure 15. The results show that DYNKV consistently outperforms the baselines across different top- $k$  retrieval ratios. The performance gains are more pronounced at higher sparsity levels, where accurate KVCache clustering and IOPS optimizations become more critical.

**Impact of Precision.** Figure 16 shows that DYNKV scales efficiently across different precisions, sustaining an average speedup of 1.36 $\times$  over two models. Notably, the benefits of DYNKV are even more pronounced when LLMs are quantized from 8-bit to 4-bit precision, since smaller neuron sizes exacerbate the negative impact of scattered flash access.

**Impact of Hardware.** Figure 17 shows that DYNKV consistently outperforms both baselines across diverse hardware configurations, achieving average 1.28 $\times$  and 1.48 $\times$  speedups



**Figure 17.** End-to-end latency of Qwen-M and Llama-M on MMLU across diverse hardware configurations.



**Figure 18.** Power and energy consumption across models on MMLU using OnePlus 12. Lines indicate average values.

across the two models. Among the tested devices, the OnePlus Ace2 exhibits the lowest performance due to its weaker UFS, making it easier to reach bandwidth limitations.

**Impact of Energy Consumption.** Figure 18 shows that DYNKV draws nearly the same power as the two baselines, since cluster splitting incurs minimal computation overhead. However, thanks to its latency improvements, DYNKV lowers the overall energy consumption by 1.57 $\times$  on average.

## 9 Related Works

**On-Device LLM Inference.** With growing concerns over privacy regulations and the need for real-time response, on-device LLM inference has become increasingly important. Prior efforts have largely focused on two directions: improving computational efficiency [6, 53, 57] and alleviating memory bottlenecks by offloading model weights to flash storage [2, 24, 51, 58]. DYNKV extends this second line of work to long-sequence decoding, where the KVCache, rather than model weights, emerges as the dominant memory bottleneck.

**KVCache Compression.** A promising direction for alleviating the KVCache memory bottleneck is compression. Some methods [13, 33, 56, 61] selectively retain only the most important tokens while evicting less important ones, thereby reducing memory usage. Others [19, 32, 35, 44, 62] focus on quantizing KVCache activations to lower precision without sacrificing accuracy. DYNKV is orthogonal to these methods and can be combined with them for further gains.

**Table 6.** End-to-end latency under varying decoding lengths on MMLU across different models using OnePlus 12.

| Length  | 1K       | 2K       | 4K       | 8K        | 16K       |
|---------|----------|----------|----------|-----------|-----------|
| Qwen-S  | 84.6 ms  | 96.5 ms  | 123.5 ms | 209.2 ms  | 376.3 ms  |
| Qwen-M  | 152.5 ms | 167.5 ms | 215.1 ms | 326.3 ms  | 636.8 ms  |
| Llama-S | 76.7 ms  | 97.6 ms  | 189.4 ms | 346.7 ms  | 677.3 ms  |
| Llama-M | 189.3 ms | 289.6 ms | 499.6 ms | 1044.0 ms | 1802.3 ms |

**KVCache Retrieval.** Recent works [29, 30, 42, 47, 55] exploit the inherent sparsity of the attention mechanism to reduce KVCache memory by retrieving only query-relevant tokens. To further improve retrieval efficiency, other approaches [18, 20, 31, 45, 60] organize the KVCache into clusters for indexing. However, all these methods overlook the distribution shift of the KVCache during decoding, leading to potential accuracy degradation. Moreover, they lack optimizations tailored for mobile devices, where stringent memory hierarchy constraints make sustaining efficiency especially challenging. DYNKV explicitly tackles both issues.

## 10 Conclusion

This paper introduces DYNKV, the first adaptive KVCache management approach designed to jointly address accuracy and efficiency for long-sequence LLM decoding on smartphones. Building on algorithmic innovations in adaptive KVCache clustering, DYNKV incorporates system-level optimizations that span the entire memory hierarchy. We envision DYNKV bringing us closer to realizing day-long intelligent agents on mobile devices, capable of continuously assisting people in their daily lives.

## References

- [1] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.
- [2] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12562–12584, 2024.
- [3] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.
- [4] Jens Axboe. liburing. <https://github.com/axboe/liburing>, 2025. Accessed: 2025-02-25.
- [5] Ge Bai, Jie Liu, Xingyuan Bu, Yancheng He, Jiaheng Liu, Zhanhui Zhou, Zhuoran Lin, Wenbo Su, Tiezheng Ge, Bo Zheng, et al. Mt-bench-101: A fine-grained benchmark for evaluating large language models in multi-turn dialogues. *arXiv preprint arXiv:2402.14762*, 2024.
- [6] Le Chen, Dahu Feng, Erhu Feng, Rong Zhao, Yingrui Wang, Yubin Xia, Haibo Chen, and Pinjie Xu. Heterollm: Accelerating large language model inference on mobile socs platform with heterogeneous ai accelerators. *arXiv preprint arXiv:2501.14794*, 2025.
- [7] Robert Chew, John Bollenbacher, Michael Wenger, Jessica Speer, and Annice Kim. Llm-assisted content analysis: Using large language models to support deductive coding. *arXiv preprint arXiv:2306.14924*, 2023.
- [8] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.
- [9] Gheorghe Comanici, Eric Bieher, Mike Schaekermann, Ice Pasupat, Naveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.
- [10] Yichuan Deng, Zhao Song, and Chiwon Yang. Attention is naturally sparse with gaussian distributed input. *CoRR*, 2024.
- [11] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitanansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.
- [12] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. *arXiv preprint arXiv:1805.04833*, 2018.
- [13] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801*, 2023.
- [14] Georgi Gerganov. ggerganov/llama.cpp: Port of facebook’s llama model in c/c++. <https://github.com/ggerganov/llama.cpp>, 2024.
- [15] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- [16] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.
- [17] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.
- [18] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Monishwaran Maheswaran, Sebastian Zhao, June Paik, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. Squeezed attention: Accelerating long context length llm inference. *arXiv preprint arXiv:2411.09688*, 2024.
- [19] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.
- [20] Jie Hu, Shengnan Wang, Yutong He, Ping Gong, Jiawei Yi, Juncheng Zhang, Youhui Bai, Renhai Chen, Gong Zhang, Cheng Li, et al. Efficient long-context llm inference via kv cache clustering. *arXiv preprint arXiv:2506.11418*, 2025.
- [21] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.
- [22] JEDEC. Jedec announces publication of universal flash storage (ufs) standard. <https://www.jedec.org>, February 2021. Accessed: 2024-10-02.
- [23] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.
- [24] Fucheng Jia, Zewen Wu, Shiqi Jiang, Huiqiang Jiang, Qianxi Zhang, Yuqing Yang, Yunxin Liu, Ju Ren, Deyu Zhang, and Ting Cao. Scaling up on-device llms via active-weight swapping between dram and flash. *arXiv preprint arXiv:2504.08378*, 2025.
- [25] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Léo Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.
- [26] Theodore Johnson and Dennis Shasha. X3: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th VLDB Conference*, pages 439–450, 1994.
- [27] Ishita Kumar, Snigdha Viswanathan, Sushrita Yerra, Alireza Salemi, Ryan A Rossi, Franck Dernoncourt, Hanieh Deilamsalehy, Xiang Chen,

- Ruiyi Zhang, Shubham Agarwal, et al. Longlamp: A benchmark for personalized long-form text generation. *arXiv preprint arXiv:2407.11016*, 2024.
- [28] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361, 2001.
- [29] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024.
- [30] Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, et al. Retrievalattention: Accelerating long-context llm inference via vector retrieval. *arXiv preprint arXiv:2409.10516*, 2024.
- [31] Guangda Liu, Chengwei Li, Jieru Zhao, Chenqi Zhang, and Minyi Guo. Clusterkv: Manipulating llm kv cache in semantic space for recallable compression. *arXiv preprint arXiv:2412.03213*, 2024.
- [32] Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. Llm-qat: Data-free quantization aware training for large language models. *arXiv preprint arXiv:2305.17888*, 2023.
- [33] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhao Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36:52342–52364, 2023.
- [34] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.
- [35] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhao Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.
- [36] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.
- [37] Meta AI. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models for edge & mobile devices. Blog post, Meta AI, September 2024.
- [38] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.
- [39] Elizabeth J O’neil, Patrick E O’neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.
- [40] Haoran Que, Feiyu Duan, Liquan He, Yutao Mou, Wangchunshu Zhou, Jiaheng Liu, Wenge Rong, Zekun Moore Wang, Jian Yang, Ge Zhang, et al. Hellobench: Evaluating long text generation capabilities of large language models. *arXiv preprint arXiv:2409.16191*, 2024.
- [41] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.
- [42] Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. Sparq attention: Bandwidth-efficient llm inference. *arXiv preprint arXiv:2312.04985*, 2023.
- [43] RyokoAI. RyokoAI/ShareGPT52K: Sharegpt human–ai conversations dataset. <https://huggingface.co/datasets/RyokoAI/ShareGPT52K>, 2023. CC0 1.0 license. Approx. 52 000 (legacy) to 90 000 conversations from ShareGPT API.
- [44] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.
- [45] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. Shadowkv: Kv cache in shadows for high-throughput long-context llm inference. *arXiv preprint arXiv:2410.21465*, 2024.
- [46] Yuchong Sun, Che Liu, Kun Zhou, Jinwen Huang, Ruihua Song, Wayne Xin Zhao, Fuzheng Zhang, Di Zhang, and Kun Gai. Parrot: Enhancing multi-turn instruction following for large language models. *arXiv preprint arXiv:2310.07301*, 2023.
- [47] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.
- [48] Termux. Termux app. <https://github.com/termux/termux-app>, 2025. Accessed: 2025-02-25.
- [49] Jinheng Wang, Hansong Zhou, Ting Song, Shijie Cao, Yan Xia, Ting Cao, Jianyu Wei, Shuming Ma, Hongyu Wang, and Furu Wei. Bit-net. cpp: Efficient edge inference for ternary llms. *arXiv preprint arXiv:2502.11880*, 2025.
- [50] Tuowei Wang, Xingyu Chen, Kun Li, Ting Cao, Ju Ren, and Yaoxue Zhang. {JENGA}: Enhancing {LLM}{Long-Context} fine-tuning with contextual token sparsity. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 123–141, 2025.
- [51] Tuowei Wang, Ruwen Fan, Minxing Huang, Zixu Hao, Kun Li, Ting Cao, Youyou Lu, Yaoxue Zhang, and Ju Ren. Neuralink: Fast on-device llm inference with neuron co-activation linking. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 147–162, 2025.
- [52] Tuowei Wang, Kun Li, Zixu Hao, Donglin Bai, Ju Ren, Yaoxue Zhang, Ting Cao, and Mao Yang. Long exposure: Accelerating parameter-efficient fine-tuning for llms under shadowy sparsity. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–18. IEEE, 2024.
- [53] Jianyu Wei, Shijie Cao, Ting Cao, Lingxiao Ma, Lei Wang, Yanyong Zhang, and Mao Yang. T-mac: Cpu renaissance via table lookup for low-bit llm deployment on edge. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 278–292, 2025.
- [54] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.
- [55] Chaojun Xiao, Pengleng Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. Infillm: Training-free long-context extrapolation for llms with an efficient context memory. *Advances in Neural Information Processing Systems*, 37:119638–119661, 2024.
- [56] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.
- [57] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuancheng Liu. Fast on-device llm inference with npus. 2025.
- [58] Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. Powerinfer-2: Fast large language model inference on a



- smartphone. *arXiv preprint arXiv:2406.06282*, 2024.
- [59] Shuai Yang, Yuying Ge, Yang Li, Yukang Chen, Yixiao Ge, Ying Shan, and Yingcong Chen. Seed-story: Multimodal long story generation with large language model. *arXiv preprint arXiv:2407.08683*, 2024.
- [60] Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. Pqcache: Product quantization-based kvcache for long context llm inference. *Proceedings of the ACM on Management of Data*, 3(3):1–30, 2025.
- [61] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.
- [62] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. *Proceedings of Machine Learning and Systems*, 6:196–209, 2024.
- [63] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.
- [64] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.