KVSwap: Disk-aware KV Cache Offloading for Long-Context On-device Inference

Huawei Zhang
schz@leeds.ac.uk
University of Leeds

Chunwei Xia
C.Xia@leeds.ac.uk
University of Leeds

Zheng Wang
Z.Wang5@leeds.ac.uk
University of Leeds

Abstract

Language models (LMs) underpin emerging mobile and embedded AI applications like meeting and video summarization and document analysis, which often require processing multiple long-context inputs. Running an LM locally on-device improves privacy, enables offline use, and reduces cost, but long-context inference quickly hits a *memory capacity wall* as the key-value (KV) cache grows linearly with context length and batch size. Existing KV-cache offloading schemes are designed to transfer cache data from GPU memory to CPU memory; however, they are not suitable for embedded and mobile systems, where the CPU and GPU (or NPU) typically share a unified memory and the non-volatile secondary storage (disk) offers limited I/O bandwidth. We present KVSwap, a software framework tailored for local devices that achieves high memory efficiency while effectively leveraging disk storage. KVSwap stores the full cache on disk, uses highly compact in-memory metadata to predict which entries to preload, overlaps computation with hardware-aware disk access, and orchestrates read patterns to match storage device characteristics. Our evaluation shows that across representative LMs and storage types, KVSwap delivers higher throughput under tight memory budgets while maintaining generation quality over existing KV cache offloading schemes.

1 Introduction

Language models (LMs) have demonstrated strong capabilities in long-form document analysis and multimedia understanding [35, 68, 73]. These underpin emerging mobile and embedded applications such as smart note-taking for meetings [22, 46], document analysis [8, 24], voice assistants for transcription and summarization [12, 26], and video search with natural language queries [12, 25]. Increasingly, users expect these capabilities to run directly on their devices.

On-device deployment offers clear benefits: enhanced privacy by keeping data on-device, offline functionality under poor connectivity, and avoiding LM API costs. With the growing availability of powerful GPUs and NPUs on modern systems on chips [3, 7, 48], local deployment is becoming feasible.

However, running LMs efficiently on resource-constrained devices like mobile and embedded AI systems remains challenging due to limited memory capacity and I/O bandwidth. While quantization [23, 37] and parameter offloading [10, 63] reduce model footprint, a critical bottleneck lies in managing the key-value (KV) cache. Unlike fixed model weights, the KV cache grows linearly with sequence length and batch size, often surpassing model size (as shown in Fig. 1). On a 4B-parameter LM, batched inputs over 16K tokens already push the KV cache beyond device memory, causing latency spikes or out-of-memory failures. As LMs are increasingly applied to long-context tasks, practical on-device inference requires efficient KV cache management.

Prior work reduces KV cache overhead by offloading from GPU to CPU memory [36, 52], alleviating GPU pressure in server settings with abundant CPU RAM. These techniques, however, are ill-suite for mobile and embedded systems, where GPUs/NPUs typically share only 8-32 GB of RAM with the CPU. A natural alternative is to offload the KV cache to non-volatile storage[1], such as NVMe-, UFS-, or eMMC-based devices. However, without careful optimization, this approach severely degrades performance: mobile memory bandwidth is roughly 100 GB/s, while disk bandwidth is often just 1 GB/s - two to three orders of magnitude lower than their server counterparts.

Can we offload the KV cache to disk for long-context, on-device inference without sacrificing throughput or generation quality? In answer, we propose KVSwap, a framework for extended-context LM inference on resource-constrained devices. Built on FlexGen [50], KVSwap supports diverse LM architectures and storage types, with simple APIs to use. KVSwap offloads KV cache entries to disk and partially reloads them during generation. It stores the complete KV cache on disk but maintains a highly compact in-memory K cache representation to predict which entries on the disk are critical for each layer. These entries are prefetched into a buffer ahead of computation, reducing disk stalls without compromising generation quality. The KVSwap runtime automatically overlaps disk transfers with compute and synchronizes memory-disk copies.

The design of KVSwap needs to overcome two major challenges introduced by on-device disk offloading. First, the

[1]We use *disk* for non-volatile storage and *memory* for main memory.

ever-increasing KV-cache size introduced by long-context is a challenge for resource-constrained mobile and embedded devices. The limited memory capacity and strict per-application memory quotas result in a *tight memory budget*, far beyond what existing KV-cache offloading methods, which were primarily designed for server-side deployments, can accommodate. To address this, KVSwap proposes a compression and reconstruction algorithm that enables aggressive memory compression while still achieving information recovery with adequate quality.

Second, embedded storage devices based on NVMe, UFS, and eMMC often exhibit *read amplification*, where the controller reads larger physical units (e.g., entire NAND pages) than requested [27, 45], resulting in extra data movement and unnecessary read operations. To mitigate this, KVSwap groups KV entries at appropriate granularities and optimizes access patterns. It also integrates a software cache (reuse buffer) to retain recently accessed entries across generations. This reduces redundant I/O, which is particularly important for low-bandwidth devices (< 200 MB/s) and batched inference, where data movement dominates latency. Together, these specified designs and optimizations enable KVSwap for efficient long-context on-device inference with negligible loss of generation quality.

Recent work explores storing the full KV cache on disk with selective retrieval [17, 39, 66]. These methods, designed for cloud-serving scenarios, primarily optimize time-to-first-token (TTFT) to reduce inference start-up delay. However, they are ineffective on resource-constrained devices during iterative decoding, as they still incur high memory overhead and assume static rather than dynamically evolving KV caches. KVSwap addresses this gap by *targeting the decoding stage* of LM inference, efficiently managing dynamically changing KV caches to reduce memory footprint while preserving generation quality. It is the first disk-based KV cache offloading method that enables high-quality, long-context LM generation on resource-constrained platforms.

We evaluate KVSwap on text, reasoning and video LMs across model sizes, memory budgets, disk types, batch sizes, and context lengths. Compared with the strongest KV cache offloading baseline, KVSwap delivers up to 1.8× (NVMe) and 4.1× (eMMC) throughput improvements at 32K context length under the same tight memory budget, while also providing *substantially higher* generation quality. Against the industry-grade vLLM [34] - in an idealized setting where all available memory is dedicated to the KV cache - KVSwap achieves comparable or higher throughput while using 11.0× less KV cache memory. These results demonstrate that KVSwap offers high memory efficiency and throughput with preserved generation quality for on-device long-context LM inference.

This paper makes the following contributions. It presents:

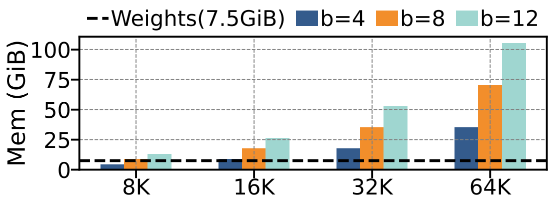


Figure 1: KV cache memory footprint across batch sizes (b) and sequence lengths (x-axis) of Qwen3-4B.

- the first disk-based KV cache offloading framework for efficient long-context, on-device LM decoding;
- a memory-efficient algorithm and system co-design that accurately identifies and accesses groups of critical KV entries, achieving substantial memory compression with preserved quality while mitigating disk read amplification;
- a runtime system for managing and caching KV entries, with optimized computation pipeline and reduced I/O transfers.

2 Background and Motivation

2.1 Language Models

Language models (LMs) process queries through a two-stage pipeline: *prefilling* and *decoding*. During prefilling, the model reads the entire input prompt to generate the first output token. During decoding, the model produces subsequent tokens one at a time, each conditioned on the previously generated tokens. Prefilling happens only once, whereas decoding is repeated for every output token. For open-ended tasks like document summarization or dialogue, decoding can involve *hundreds or even thousands* of steps, and this becomes even more common when using reasoning-focused models with chain-of-thought (CoT) [59].

Attention is central to modern LMs, where each token is represented by a *Query* (Q), *Key* (K), and *Value* (V) vector. Scores are computed by comparing queries with keys, and the resulting weights are applied to values to form contextualized representations. *Multi-head attention (MHA)* [57] extends standard attention by computing multiple sets of QKV vectors in parallel, but increases memory use since each head stores its own keys and values. *Grouped-query attention (GQA)*[9] addresses this by sharing keys and values across heads, reducing memory overhead while maintaining accuracy. GQA is widely adopted in SOTA models such as LLaMA3 and Qwen3 [20, 64].

During generation, LMs use a *key-value (KV) cache* to store keys and values from past steps, enabling the reuse of computations and reducing inference latency [47].

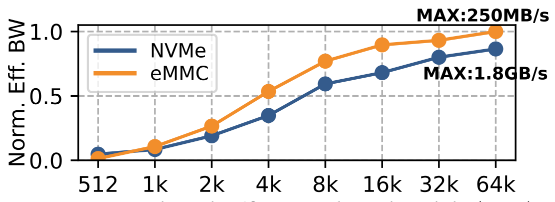


Figure 2: Normalized effective bandwidth (BW) with varying block sizes (bytes).

2.2 On-device KV Cache Memory Wall

The KV cache serves as a critical component of program state in on-device applications of LMs and is therefore usually required to be memory-resident. Fig. 1 shows the KV cache memory footprint for the Qwen3-4B model (W16A16) under varying batch sizes and context lengths. The model weights alone occupy 7.5 GiB of RAM. At a context length of 16 K and batch size 4, the KV cache reaches 9 GiB, already exceeding the weight size. Because KV cache size scales linearly with both batch size and context length, it grows rapidly: at 32 K context and batch size 12, it reaches 54 GiB.

On embedded and mobile systems with limited memory capacity (e.g., 8 GiB) and strict per-application quotas (to avoid disrupting other applications and OS services), retaining the full KV cache in memory becomes infeasible. This results in a *KV cache memory capacity wall*, where the KV cache overtakes model weights as the dominant memory consumer, limiting the scalability of long-context on-device inference.

2.3 KV Cache Disk Offloading

Prior work shows that a small fraction of tokens substantially influence attention score computation, and that these influential tokens change dynamically with the context and query [36, 52, 53, 74]. This insight enables selective KV cache loading by predicting the most relevant tokens at each generation step. Existing techniques exploit this by offloading KV cache from GPU to CPU memory [36, 52], where both memory hierarchies are high-bandwidth and optimized for low-latency access. In contrast, our work targets embedded and mobile platforms, where RAM is scarce and subject to strict quotas. We propose offloading KV cache from memory to disk, targeting a different storage hierarchy with far more severe bandwidth and latency constraints.

To assess the feasibility of disk offloading, we profile the random read bandwidth of two representative types of disks - NVMe[2] and eMMC - under varying block sizes. We focus on random, rather than sequential access patterns because our setting requires *selectively* retrieving critical KV entries,

[2]UFS-based storage is not supported on our platform, but it exhibits I/O bandwidth and characteristics similar to NVMe.

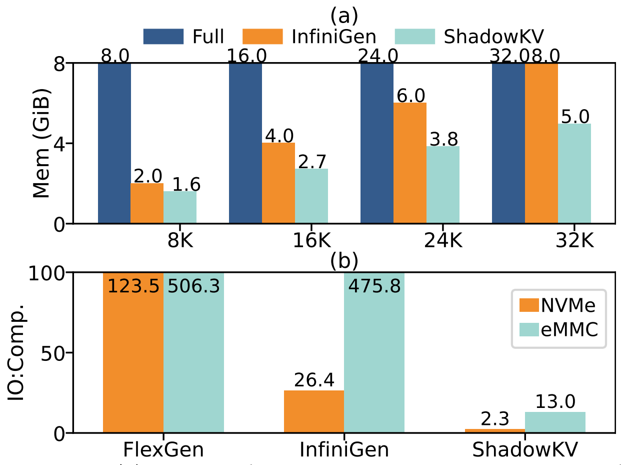


Figure 3: (a) KV cache management memory with batch=8 and varying context lengths (x-axis); (b) Decoding latency ratios of I/O to compute under 32K context length and batch=8.

operations that inherently produce predominantly random accesses. From Fig. 2, we observe:

Diverse disk I/O bandwidths. Disk types vary significantly in I/O bandwidth: NVMe can reach up to 1.8 GB/s, while eMMC can be limited to around 250 MB/s.

Poor I/O utilization for small KV entries. All storage types show severe bandwidth under-utilization for small transfer sizes. At 512 bytes, the typical size of a KV entry[3], the effective bandwidth drops below 6% of peak bandwidth for both devices.

We observe that bandwidth utilization improves substantially with larger block sizes. KVSwap addresses this by adopting a quality-aware, *group-wise* strategy that packs multiple KV entries into a single transfer, thereby amortizing I/O overhead and reducing fragmentation while incurring negligible model accuracy degradation (Sec. 3.3).

2.4 Drawbacks of Prior Offloading Schemes

High memory overhead. Recent KV cache offloading methods [36, 50, 52] are designed for CPU-GPU transfer on servers and are ill-suited for disk-based on-device offloading. They do not explicitly consider extremely tight memory constraints (e.g., <500 MiB), a regime that closely matches the practical application memory budget in common on-device deployment scenarios. Figure 3a compares the KV cache management memory of InfiniGen [36] and ShadowKV [52] against storing the full cache in memory for LLaMA3-8B. Their designs introduce a significant memory footprint, particularly for long contexts. For example, with a 16K context length and batch size 8, memory consumption already reaches 4 GiB

[3]128 (head dimension) × 2 (key and value) × 2 Bytes (float16) = 512 Bytes.

```
import KVSWAP
KVSWAP.Parameter_Tuning(
    model=model_obj, max_context_len=32768,
    max_batch_size=8, max_kv_mem=2200 # in MiB
).save("config.json")
```

(a) Offline parameter tuning.

```
engine = KVSWAP.Init( model=model_obj,
↪  kv_offload_path="...",
↪  config_file="config.json" )
outputs = engine.generate( inputs=[...],
↪  sampling_params=[...] )
```

(b) Model serving.

Figure 4: Simplified examples for using KVSwap for offline parameter tuning (a) and model serving (b).

and 2.7 GiB. Here, we attempted to reconfigure their parameters for tighter memory budgets and extend them to disk-based offloading, but this severely degrades generation quality (Sec. 5.1). We attribute this to the inability of their memory compression algorithms to tolerate high compression ratios. A detailed discussion is provided in Sec. 3.2 and Sec. 3.3.

Low I/O efficiency. The fine-grained head- or token-level operations create fragmented disk access patterns, causing excessive small disk reads and degrading I/O efficiency. As an example, Figure 3b shows the decoding latency ratio of I/O to compute for FlexGen [50], InfiniGen [36], and ShadowKV [52]. Note that FlexGen does not select KV entries; instead, it loads the full KV cache into memory. All these methods exhibit ratios far exceeding 1, with some cases reaching over 100. While ShadowKV achieves the lowest ratios, it still reaches 13.0 on eMMC and 2.3 on NVMe. These high ratios indicate a severely imbalanced I/O–compute pipeline caused by inefficient I/O access. End-to-end results in Sec. 5.2 further validate the resulting low system throughput for these methods.

Lessons learnt. For on-device disk offloading, existing KV cache offloading methods suffer from *high memory overhead*, which limits scalability to longer contexts or larger batch sizes. In addition, their *low I/O efficiency* significantly limits system decoding, resulting in low generation speed and energy waste. KVSwap is designed to avoid these pitfalls.

3 Our Approach

KVSwap is a Python framework and can support PyTorch-based Transformer LMs, providing simple APIs for model serving and parameter tuning. Figure 4 shows example usage for LM inference: an offline parameter tuning API (Fig. 4a) generates configurations used by the KVSwap runtime (Fig. 4b). Using KVSwap is straightforward and typically requires only a few dozen lines of Python code.

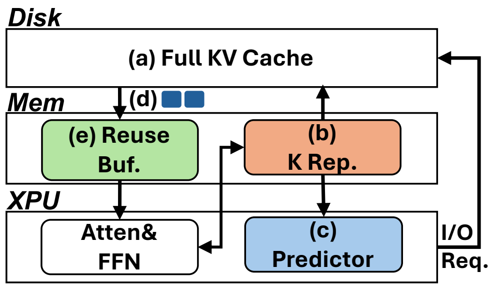


Figure 5: High-level overview of KVSwap.

Fig. 5 gives a high-level overview of KVSwap. KVSwap reduces memory pressure and improves system efficiency by: (a) offloading the full KV cache to disk; (b) maintaining a compact in-memory key cache representation to identify important KV entries using a predictor (c), which will be preloaded from the disk for decoding; (d) minimizing disk I/O overhead by structuring disk access patterns; and (e) effectively caching and reusing previously accessed KV entries through a reuse buffer. The KVSwap runtime coordinates I/O and compute, tracks reusable entries across layers, fetches missing ones from disk for the attention kernel, and periodically updates the on-disk full cache and the in-memory representation with newly generated data. The KV predictor is detailed in Sec. 3.1 and Sec. 3.3, the in-memory compression scheme in Sec. 3.2, and the runtime scheduling and reuse buffer in Sec. 3.4.

3.1 Predicting Important KV Cache Entries

The KVSwap KV predictor identifies, *at runtime*, a subset of KV entries needed for each attention computation. Like InfiniGen [36], it estimates the next layer's attention scores from the current layer's input, treating it as an approximate query (Q). These scores quantify the importance of each key and its associated value. Unlike [36], our setting assumes the entire KV cache resides on disk, where memory and I/O bandwidth are limited. To make prediction feasible, KVSwap maintains a *compact* in-memory representation of the K cache. This compressed cache is used to compute approximate attention scores and select a configurable number of top-ranked KV entries *as groups* for preloading.

Thus, the predictor balances two objectives: (1) minimizing memory footprint, through K cache compression and reconstruction, and (2) improving disk efficiency, through grouped critical KV prediction. These two techniques are detailed in the following subsections.

3.2 Compressing K Cache

To enable prediction without an in-memory full K cache, KVSwap maintains a *compressed* K cache in memory using low-rank approximation. Instead of storing every detail of the K matrix, we keep a smaller "summary" that captures its most important patterns. The compressed K cache is automatically managed and updated by the KVSwap runtime (Sec. 3.4.1). Since the low-rank K cache is used only for estimating attention scores - not for actual attention computation - precision can be flexibly traded for memory efficiency.

We consider two designs: (a) *per-head compression*, compressing each head with a separate projection matrix, and (b) *joint-head compression*, merging head and embedding dimensions with a single projection. We adopt the latter for its unified representation and lower memory cost, reconstructing the multi-head structure during prediction (Sec. 3.3).

We first reshape the K cache into a matrix of size $N \times (H_k \cdot d)$, where N is the token count, H_k is the number of heads, and d is the head dimension. Each vector is then projected into a lower dimension r using a *pre-computed low-rank adapter* $A \in \mathbb{R}^{(H_k \cdot d) \times r}$, with $r \ll H_k \cdot d$. To obtain A, KVSwap applies *Singular Value Decomposition (SVD)* [21] to a flattened K cache ($K_{\text{ftn}} \in \mathbb{R}^{N \times (H_k \cdot d)}$) collected during offline parameter tuning. By default, samples of this K cache are drawn from general-purpose datasets [44, 49], though users may provide their own through the KVSwap API. Formally, $\text{SVD}(K_{\text{ftn}}) = U\,\text{diag}(S)\,V^{\top}$, where $V \in \mathbb{R}^{(H_k \cdot d) \times m}$ and $m = \min(N, H_k \cdot d)$. The top-r right singular vectors of V form A, and the compressed K cache is $K_{\text{lr}} = Flatten(K)A$, with compression level controlled by the compression ratio, $\sigma = \frac{H_k \cdot d}{r}$. We found that the resulting low-rank adapter generalizes well to different datasets.

Compared with prior K cache compression scheme. While ShadowKV [52] also employs a low-rank K cache, its design and purpose differ from ours. ShadowKV compresses the K cache to reduce its memory footprint, but must reconstruct the full K cache for computation. Hence, it has to use a conservative compression ratio to limit accuracy loss. In contrast, KVSwap uses the low-rank K cache only to predict critical KV entry indices, making it far more resistant to aggressive compression. Combined with our prediction algorithm (Sec. 3.3), this enables KVSwap to maintain generation quality even under high compression ratios, whereas ShadowKV degrades significantly (Sec. 5.1). The construction method also differs. ShadowKV performs an online SVD at runtime, adding substantial prefill latency (e.g., 4.9× slower at 8K context length).[4] KVSwap instead precomputes the low-rank adapter offline, incurring no extra prefill cost.

3.3 Grouped Critical KV Entries Prediction

To minimize disk I/O, KVSwap estimates which KV entries are most important for attention computing in the next layers and only preloads these. A unique feature of KVSwap is that it first groups G consecutive key-value (KV) entries to align with the block-read characteristics of devices like NVMe, UFS, and eMMC (see also Fig. 2), then predicts and loads the most important groups. This is different from prior work [36, 51, 52] that predicts on individual heads or tokens. The group size is configurable and can be automatically determined during offline processing (Sec. 3.5).

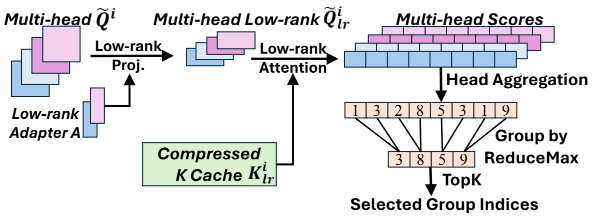


Figure 6: Grouped critical KV entries prediction.

Fig. 6 depicts how we predict which *groups* of KV cache entries to be loaded from disk. The idea is to use the low-rank K cache, K_{lr}, to estimate the attention scores ($QK^{\top}$) without accessing the full K cache. K cache elements (and their associated values, V) that contribute to a high attention score will be prefetched into memory. Specifically, for query head h, we approximate the attention score as:

$$Q_h K_{g(h)}^{\top} \approx Q_h \left(K_{\text{lr}} A_{g(h)}^{\top}\right)^{\top} = \left(Q_h A_{g(h)}\right) K_{\text{lr}}^{\top} \qquad (1)$$

Here, $Q_h \in \mathbb{R}^{1 \times d}$ is the query vector for head h; $A_{g(h)} \in \mathbb{R}^{d \times r}$ is the low-rank adapter for the K cache assigned to h; and $g(h)$ maps each query head to its shared K head. We refer to the term $Q_h A_{g(h)}$ as a low-rank query vector. This enables the reconstruction of multi-head attention from a compressed, head-unified K_{lr}, thereby reducing computation and allowing us to decide which *groups* of KV entries are worth loading.

Online prediction. We predict the critical KV groups ahead of time to issue disk load concurrently with the computation. Specifically, while computing layer $i{-}1$, KVSwap predicts in advance the critical KV groups needed for layer i. To do this, we approximate the upcoming input X_i using X_{i-1}, taking advantage of the observation that the input (embeddings) to a Transformer layer i is often similar to that at layer $i{-}1$ [36, 57]. Using this approximation, we apply layer i's projections to obtain a low-rank multi-head query vector $\tilde{Q}^i_{\text{lr}}$. Together with K^i_{lr}, this vector is used in a low-rank attention computation to estimate the attention scores for layer i.

Scoring and selection. To select critical KV cache entries, we first compute a single importance score for each token by summing the scores across all heads. We then form groups of G consecutive tokens and represent each group by the maximum score among its members. Finally, we select the top-ranked groups with the highest representative scores. A `ReduceMax` operation within each group captures the most salient token score, and a `TopK` step selects the indices of M most relevant KV entry groups to load for layer i.

[4]From 7.6s to 37.2s on LLaMA3-8B with NVIDIA Orin AGX, batch size = 1.

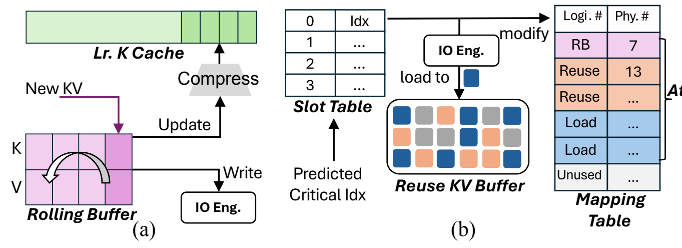


Figure 7: KVSwap runtime: (a) a compressed K cache and (b) reuse buffer for KV cache management.

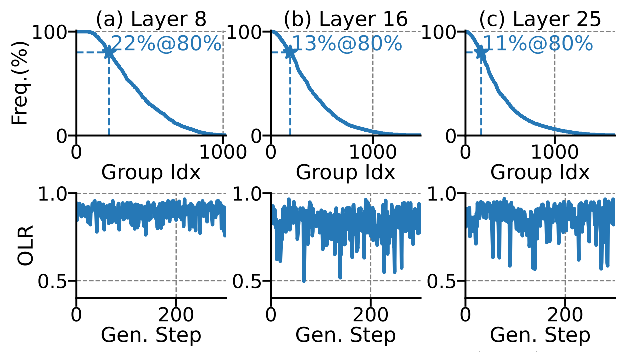


Figure 8: Frequency and overlap ratio (OLR) of predicted critical KV groups over 300 decoding steps. Group indices are reordered by frequency for clarity.

Compared with prior prediction method. Unlike prior work [36], our approach introduces prediction at the group granularity, rather than at individual KV entries. Beyond this, while InfiniGen [36], a representative prior work, also employs approximated attention as its prediction mechanism, our algorithm differs in two fundamental aspects. (1) InfiniGen adopts an index-selecting strategy that pre-determines which embedding dimensions of the K cache to retain and directly selects them via indices. Under stringent K cache compression, this strategy struggles to preserve fidelity as the number of selected indices decreases. (2) InfiniGen applies multi-head attention directly to the index-selected K cache, whereas our method constructs a head-unified low-rank K cache that aggregates information across heads, enabling more aggressive compression. We then reconstruct multi-head attention from this representation using Eq.1.

3.4 KVSwap Runtime System

We now introduce the overall KVSwap runtime system, which operates during the prefilling and decoding stages (Sec. 2.1). During the *prefilling* phase, KVSwap captures the LM-generated KV cache for the input prompts and writes it to disk in a layer-by-layer fashion. At the same time, it computes a low-rank K cache from the full K cache to create an initial compact K cache (Sec. 3.2) that is stored in memory. When *decoding* begins, the runtime coordinates the grouped critical KV predictor (Sec. 3.3) and the KV cache manager (Sec. 3.4.4) to efficiently support disk-based KV cache loading and attention computation. It identifies, loads, and reuses important KV entry groups while keeping the most recent KV entries in memory. To maximize efficiency and overlap I/O with computation, these operations for the next layer are performed concurrently with the attention and feed-forward network (FFN) computations of the current layer (Sec. 3.3).

3.4.1 New KV entries management. As shown in Fig. 7a, our in-memory KV cache has two components: a low-rank compressed K cache (Sec. 3.2) and a rolling buffer (RB). The compressed K cache serves as the input to our KV predictor (Sec. 3.3). During decoding, new entries are appended to the RB. Since critical entries are predicted in groups, the importance of recent entries cannot be assessed until a group is completed. Thus, the RB temporarily stores recently generated KV entries until they accumulate to a full group of size G for offloading to disk.

3.4.2 Locality in grouped KV predictions. We observe that a subset of critical KV entries exhibits temporal locality across layers. For example, applying our grouped KV predictor (with group size=4) to Qwen3-8B on QMSum [14] samples, we define the *overlap ratio* at generation step j as the fraction of important groups at step j that also appeared at step $j-1$. Fig. 8 reports frequency and overlap ratios over 300 decoding steps[5]. Fewer than 22% of groups account for 80% of occurrences, showing that the set of important groups changes substantially over time. However, adjacent steps exhibit strong overlap, revealing temporal redundancy. This motivates our reuse strategy: retaining recently accessed critical groups in memory allows them to be efficiently reused in subsequent steps without reloading from disk.

3.4.3 Reuse buffer for caching grouped KV entries. We introduce a *KV reuse buffer* (not to be confused with the rolling buffer described in Sec. 3.4.1 that stores the recently generated KV entries) to store recently accessed critical KV entries (loaded from disk) for potential reuse across prediction steps. By reusing overlapping groups of critical KV entries rather than reloading them, the reuse buffer reduces I/O overhead - especially on low-bandwidth disks or when handling large KV volumes. We implement the reuse buffer as a fixed set of dedicated *memory slots*, each capable of storing one KV group. A slot table records the group ID currently stored in each slot. When the predictor identifies the next set of critical groups (Fig. 7b), the system first checks the slot table to see whether a group is already in the reuse buffer. If present, the corresponding slot from the reuse buffer is passed to the attention kernel for computation directly; if not, the KV group

[5]Results from layers 8, 16, 25 are shown; other layers have similar patterns.

is loaded from disk and stored in an available slot, with the slot table updated accordingly. We use a simple FIFO policy as the replacement strategy for the reuse buffer.

3.4.4 KV cache manager. With KVSwap, attention computation consumes KV entries from: hit slots in the reuse buffer, important KV groups that are not in the reuse buffer and need to be loaded from disk, and recent newly generated entries from the rolling buffer. The KVSwap KV cache manager uses a mapping table to logically address these heterogeneous memory regions - similar to OS virtual memory. This abstraction provides direct compatibility with PagedAttention [34], which expects a contiguous logical view of the KV cache. During generation, reuse patterns change over time, causing the valid memory regions and the positions of usable data in the reuse buffer to shift. To handle this, the runtime updates the mapping table before each attention computation to reflect the current KV entry layout (Fig. 7b).

3.5 Offline Parameter Tuning

KVSwap provides an API (Fig. 4a) for *one-off* offline parameter tuning, which selects optimal runtime settings based on user constraints (maximum memory $\mathcal{B}_{\max}$, context length $S_{\max}$, and batch size $b_{\max}$), the target LM, and the hardware platform. In our evaluation, tuning completes in under 30 minutes and outputs a JSON file that records the best-found parameters for runtime scheduling. These include group size for KV prediction G, K-cache compression ratio σ, number of selected groups M, and reuse buffer capacity C (Sec. 3.4).

Parameter search tries to balance three tightly coupled factors: *generation quality*, *inference throughput*, and *memory usage*. For example, G, M, and C affect both I/O cost and computation, directly influencing throughput. G and σ shape generation quality, while σ, M, and C determine memory footprint. These interdependencies create a three-way trade-off where improving one dimension often degrades another. Since accurate quality evaluation is also costly, finding a global optimum is impractical. To address this, we use a *greedy-based parameter solver* to efficiently search offline for good local configurations. More details of our parameter search method can be found in Appendix A.

4 Experimental Setup

4.1 Platform and Workloads

Evaluation platform. We evaluate our approach on the NVIDIA Jetson Orin AGX [1] embedded platform with a 12-core Cortex-A78 CPU at 1.3 GHz, an Ampere GPU with 2048 CUDA cores and 64 GB of unified RAM. We apply KVSwap to two widely used disk types on mobile and embedded systems: NVMe (I/O bandwidth 1.8 GB/s) and eMMC (I/O bandwidth 250 MB/s). The system runs JetPack-6.2 with Linux kernel 5.15.148-tegra, CUDA 12.6 and PyTorch 2.7. KVSwap is designed for scenarios with a tight memory budget. However, as some competing baselines require considerable memory, we leverage the 64 GB of RAM available on the platform to ensure that all methods can be evaluated fairly. Sec. 4.3 reports the actual KV-cache memory used in each experimental setting.

Language models. We evaluate KVSwap on both text and video understanding tasks using eight LMs of varying sizes. For text tasks, we use LLaMA-3.1-8B-Instruct (LLaMA3-8B), LLaMA-3.2-3B-Instruct (LLaMA3-3B) [20], and Qwen3 [64] models (4B, 8B, 14B). We enable the thinking mode of Qwen3 to evaluate it on CoT reasoning scenarios. For video understanding, we select Qwen2.5-VL-3B, 7B [13] and InternVL3-14B [76].

Tasks. We test on long-context tasks using all English tasks from LongBench [14], Needle-in-a-Haystack (NIAH) [32], and eight tasks from RULER [28]. Test context length is fixed at 32K for RULER, and ranges from 4K to 32K for LongBench and NIAH. For video understanding, we use open-ended generation tasks from MLVU [75], including Sub-Scene Captioning (SSC) and Video Summarization (VS), with context lengths of 22K–32K per video.

4.2 Competing Baselines

We compare KVSwap to the following baselines:

InfiniGen [36]: This closely related work offloads KV cache between GPU and CPU memory by predicting and selecting important KV entries *per* head and token position, unlike our group-based prediction. We extend InfiniGen to the modern GQA attention scheme and adapt its runtime for disk-based offloading. We also evaluate two enhanced variants: *InfiniGen** adds head aggregation from our prediction strategy (Fig. 6), and *InfiniGen*+ru* further incorporates our KV reuse strategy to improve disk I/O.

FlexGen [50]: It is designed to offload the entire KV cache to disk. During decoding, the full KV cache is restored layer by layer into memory for full attention computation.

ShadowKV [52]: This CPU-GPU KV offloading framework stores a low-rank K cache on the GPU and offloads the V cache to CPU memory (see also Sec. 3.2). During generation, it loads only important parts of the V cache and reconstructs the K cache on-the-fly. Like InfiniGen, we also adapt its runtime to support disk offloading.

Loki [51]: It is a sparse attention method originally designed to accelerate attention computation. We modify its core approximate attention formulation to function as a predictor for identifying critical KV entries, and embed it into our modified version of InfiniGen with disk-offloading support, where the core prediction components are replaced.

vLLM [34]: This industry-grade LM serving system stores the full KV cache in memory by default. We use vLLM to *approximate the throughput upper bound*, measuring how closely an offloading strategy approaches the ideal case without disk overhead. All remaining device memory - after storing model weights and allocating workspace - is dedicated to the KV cache. We tune vLLM parameters and report the best throughput achieved.

4.3 Evaluation Methodology

Perspective. We assess KVSwap under two settings. **(A)** We align the *per-batch* memory consumption of each method to a consistent, low budget and compare their performance with varying conditions under this constraint. **(B)** We fix the *total* memory budget and evaluate each method under its best-case configuration. Notably, for baselines at their optimal settings, per-batch memory consumption varies, which directly impacts the maximum achievable batch size for each. We evaluate setting A in Sec. 5.1, 5.2, and 5.3, and setting B in Sec. 5.4.

Setting A. In setting A, for all offloading methods, we constrain the *per-batch* KV memory budget of all evaluated LMs to fixed fractions of their corresponding full cache. We employ two budgets: a relaxed budget (1/13 of the full cache) and a tight budget (1/34 of the full cache). The latter is denoted with the suffix "-t" (e.g., KVSwap-t). Specifically, we reconfigure the baselines as follows: InfiniGen by adjusting the partial weight ratio [36], Loki by tuning key dimensionality, and ShadowKV by adjusting chunk size, outliers, and K-cache rank. For KVSwap, we set $S_{\max} = 32\text{K}$, $b_{\max} = 16$, $MG = 400$, and maximum K-cache compression ratio $\sigma_{\max} = 32$.

Setting B. In setting B, we apply the same configuration for KVSwap as in setting A. And for all offloading baselines, we apply their optimal configurations. Specifically, for ShadowKV and LoKi, we directly adopt the configurations reported in their source publications. For InfiniGen, since no reference configurations are readily available, we experimentally set the partial weight ratio to 0.5, resulting in a conservative 4× KV cache memory reduction. Here, their per-batch memory consumption varies and is generally higher than that of KVSwap. To ensure a fair comparison, we instead fix the *total* memory budget and evaluate each method at the *maximum batch size* it can support within this budget. We evaluate under a relaxed total budget of 2000 MiB and a tight total budget of 800 MiB.

Example test memory configuration. Tab. 1 shows the tested memory budgets for LLaMA3-8B. For vLLM, it is an ideal baseline that uses all available device memory for KV cache (31 GiB) and is not affected by the evaluation settings. Qwen3-8B has a memory budget similar to that of LLaMA3-8B.

Table 1: Tested KV memory budgets for LLaMA3-8B. For KVSwap in both settings (and offloading baselines in setting A), we constrain the *per-batch* KV memory budget of all evaluated LMs to 1/13 (relaxed) and 1/34 (tight) of their full KV cache. For setting B, we fix the *total* KV memory budget to 2000 MiB (relaxed) and 800 MiB (tight) for all offloading methods.

Settings	KV Memory Configuration (MiB)		vLLM (GiB)
	Relaxed	Tight	
A (Sec. 5.1, 5.2, 5.3)	310/batch@32K	120/batch@32K	31
B (Sec. 5.4)	2000 in total	800 in total	

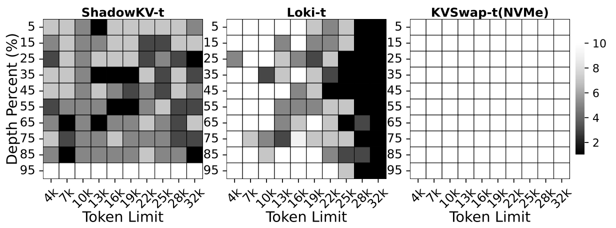


Figure 9: Qwen3-8B generation quality on NIAH. x-axis is the tested context length and the y-axis is the relative position to test within a context length.

4.4 Performance Metrics

We report *generation accuracy* and *throughput* (tokens per second) during decoding. Accuracy is reported under varying KV cache budgets and context lengths. Throughput is measured as the average decoding rate over 1000 continuously generated tokens. Each result is the average of five runs with different randomly selected test inputs. Unless otherwise stated, throughput is calculated on LLaMA3-8B, with Qwen3-8B showing similar trends.

5 Experimental Results

In this section, we show that KVSwap maintains generation quality while delivering strong throughput, outperforming all offloading baselines across diverse settings.

5.1 LM Generation Quality

5.1.1 Text processing. Tab. 2 reports the generation accuracy of LLaMA3-8B on RULER and LongBench with maximum context length 32K. *Full-KV* shows raw accuracy using the entire KV cache, while other results indicate accuracy loss relative to Full-KV. KVSwap occasionally exceeds Full-KV due to the probabilistic nature of LMs. Results for Qwen3-8B are similar and given in Tab.1 of Appendix B.

KVSwap and KVSwap-t (with a tighter memory budget - see Sec. 4.3) outperform all offloading baselines with acceptable accuracy loss. InfiniGen has the largest degradation, as it fails to capture critical KV entries under tight budgets.

Table 2: Relative accuracy loss (%) of LLaMA3-8B compared to Full-KV on RULER (left) and LongBench (right).

Methods	S1	S2	MK1	MQ	MV	QA1	QA2	VT	Avg.	SQA	MQA	SUM	FSL	SYN	COD	Avg.
Full-KV (raw %)	100.0	100.0	100.0	98.8	99.5	82.0	56.0	96.6	91.6	39.3	41.4	28.0	68.0	99.5	49.1	54.2
InfiniGen	−100.0	−100.0	−100.0	−98.8	−99.5	−61.0	−40.0	−96.6	−87.0	−36.0	−37.8	−26.8	−50.8	−99.0	−44.7	−49.2
InfiniGen*	−86.0	−97.0	−97.0	−98.3	−99.5	−39.0	−20.0	−93.0	−78.7	−14.4	−14.8	−10.7	−14.6	−5.3	−25.4	−14.2
Loki	−3.0	−14.0	−18.0	−67.5	−61.5	−28.0	−15.0	−65.4	−34.0	−7.4	−4.9	−10.8	−8.0	−0.3	−21.6	−8.8
ShadowKV	**0.0**	−86.0	−84.0	−93.0	−94.5	−13.0	−7.0	−41.0	−52.3	−9.7	−2.8	−4.1	−5.1	−1.5	−6.8	−5.0
KVSwap $^{\text{NVMe}}$	**0.0**	−1.0	**0.0**	**−3.3**	**−7.0**	−1.0	**+2.0**	**−10.4**	**−2.6**	**+0.9**	−0.3	**−2.1**	**−0.7**	**0.0**	**−1.8**	**−0.6**
KVSwap $^{\text{eMMC}}$	**0.0**	**0.0**	**0.0**	−9.0	−12.0	**0.0**	−3.0	−11.8	−4.4	−0.5	**−0.1**	−2.3	−1.4	−0.5	**−1.8**	−1.1
Loki-t	−99.0	−100.0	−100.0	−98.8	−99.5	−57.0	−32.0	−96.6	−85.4	−24.5	−25.7	−20.5	−43.2	−7.6	−38.2	−26.6
ShadowKV-t	−21.0	−89.0	−93.0	−95.8	−95.0	−16.0	−6.0	−79.2	−61.9	−15.5	−7.0	−7.1	−10.2	−4.5	−8.0	−8.7
KVSwap-t$^{\text{NVMe}}$	**0.0**	**−1.0**	**−3.0**	**−10.8**	**−13.3**	**−4.0**	**0.0**	**−13.0**	**−5.6**	**−2.5**	**−0.4**	**−3.0**	−2.7	**0.0**	−5.8	**−2.4**
KVSwap-t$^{\text{eMMC}}$	**0.0**	−13.0	−16.0	−54.3	−55.8	−13.0	−7.0	−24.6	−22.9	−4.2	−1.1	−3.9	**−1.5**	−0.5	**−5.2**	−2.7

Table 3: Relative accuracy (%) and score loss over Full-KV on reasoning (left) and video understanding (right).

Methods	Q4B	Q8B	Q14B	QVL3B	QVL7B	IVL14B
Full-KV (raw %)	60.5	62.5	66.0	4.5	4.9	4.8
Loki	−10.0	−4.1	−9.6	−1.4	−1.0	−2.0
ShadowKV	−6.6	−5.4	−7.3	−0.7	−0.6	−0.5
KVSwap $^{\text{NVMe}}$	**−1.8**	**−2.4**	**−2.6**	**−0.4**	**−0.3**	**−0.3**
KVSwap $^{\text{eMMC}}$	−4.0	−3.0	−4.6	**−0.4**	**−0.3**	**−0.3**
Loki-t	−57.2	−47.2	−58.6	−2.5	−2.9	−2.7
ShadowKV-t	−50.1	−47.5	−45.0	−2.1	−2.4	−2.4
KVSwap-t$^{\text{NVMe}}$	**−3.6**	**−4.8**	**−6.7**	**−1.7**	**−0.6**	**−0.4**
KVSwap-t$^{\text{eMMC}}$	−7.2	−6.9	−10.0	−1.8	−0.7	−0.5

Table 4: Throughput (tokens/s) comparison of LLaMA3-8B across batch sizes and context lengths (CLs).

	Methods	CL=16K					CL=32K				
		b=1	b=2	b=4	b=8	b=16	b=1	b=2	b=4	b=8	b=16
eMMC	FlexGen	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
eMMC	Loki/InfiGen	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
eMMC	InfiGen*	1.7	1.7	1.5	1.4	1.4	1.6	1.5	1.4	1.4	1.4
eMMC	InfiGen*+ru	4.0	4.7	5.2	4.2	3.7	4.2	4.4	4.9	4.5	3.5
eMMC	ShadowKV	3.0	3.8	4.1	4.4	3.4	3.0	3.7	4.2	3.8	3.4
eMMC	KVSwap	**5.9**	**10.5**	**16.2**	**15.8**	**11.2**	**6.0**	**10.3**	**15.2**	**15.7**	**10.9**
NVMe	FlexGen	0.8	0.8	0.8	0.8	0.8	0.4	0.4	0.4	0.4	0.4
NVMe	Loki/InfiGen	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.8	1.8
NVMe	InfiGen*	5.0	8.4	10.8	11.3	13.1	5.1	7.5	8.9	10.7	12.0
NVMe	InfiGen*+ru	5.2	9.2	14.9	22.5	26.6	4.9	8.5	11.9	14.3	17.2
NVMe	ShadowKV	6.4	10.4	16.3	21.9	26.7	6.4	10.0	16.3	21.5	26.2
NVMe	KVSwap	**6.9**	**11.9**	**20.8**	**35.1**	**46.1**	**6.9**	**11.9**	**21.4**	**35.6**	**46.8**
-	vLLM	9.7	17.1	28.4	41.2	39.5	9.2	14.1	21.0	20.8	23.2

Although InfiniGen* reduces accuracy loss by incorporating our head aggregation to reduce prediction noise, it still falls far behind KVSwap, with average accuracy losses of 78.7% on RULER and 14.2% on LongBench. By contrast, KVSwap keeps average accuracy loss within 4.4% and 1.1%. ShadowKV leads to 52.3% and 5.0% accuracy loss. Loki also suffers high losses of 34.0% and 8.8%. Furthermore, under a highly constrained budget, only KVSwap-t maintains usable accuracy: ≤5.6% and 2.4% loss for NVMe. Note that KVSwap $^{\text{NVMe}}$ and KVSwap $^{\text{eMMC}}$ differ because the best group size is G=4 for NVMe and G=8 for eMMC.

Fig. 9 compares KVSwap-t with Loki-t and ShadowKV-t on NIAH with Qwen3-8B under a tight 130 MiB per-batch memory budget (using NVMe). The x-axis shows the context length (or token limit) to be tested, and the y-axis shows the relative position to be tested within a context; lower scores with dark regions indicate areas where the model exhibits degraded capability. Only KVSwap-t maintains full processing capability at all sequence locations, while Loki-t and ShadowKV-t suffer substantial generation performance degradation under the same memory budget.

5.1.2 Reasoning and video understanding. Tab. 3 reports generation quality on CoT reasoning and video understanding LMs. For reasoning, we evaluate on three multi-document QA tasks from LongBench (HotpotQA, 2WikiMultihopQA, MuSiQue) [14], reporting averaged accuracy across tasks. For video understanding, we evaluate Qwen2.5-VL-3B (QVL3B), 7B (QVL7B), and InternVL3-14B (IVL14B), reporting averaged scores on MLVU-SSC and MLVU-VS (range: 2-10). Recalling evaluation setting A (Sec. 4.3), we configure all offloading methods with per-batch memory budgets set to 1/13 and 1/34 of the full KV cache, applied across all models. We omit InfiniGen and InfiniGen* since Tab. 2 and Tab. 1 of Appendix B already confirm their poor performance.

Like with text processing, KVSwap outperforms Loki and ShadowKV on NVMe and eMMC disks, with minor accuracy losses ≤4.6%, and score losses ≤0.4. KVSwap-t also strikes a good balance between KV cache memory footprint and generation quality, while all other methods incur severe degradation in output quality (≥45.0% accuracy loss on reasoning models and ≥2.1 points, or 46.7%, on video models). In contrast, KVSwap-t constraints losses to 3.6-10.0% on reasoning models and 0.4-1.8 points on video models. Its advantage grows with model size: ≥0.3 points on QVL3B, ≥1.7 on QVL7B, and ≥1.9 on IVL14B.

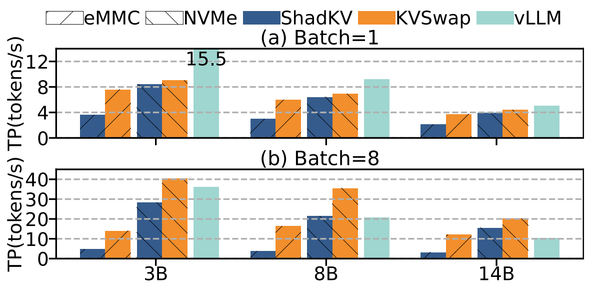


Figure 10: Throughput (TP) comparison across model sizes at a 32K context with batch sizes of (a) 1 and (b) 8.

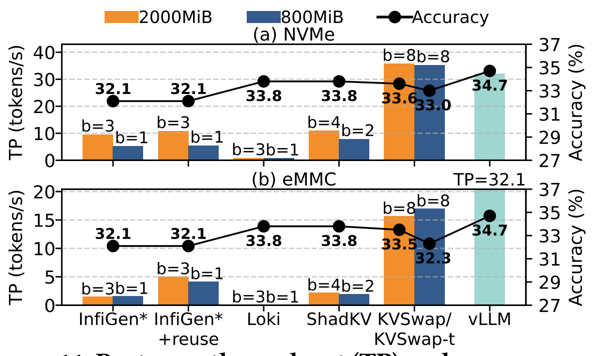


Figure 11: Best-case throughput (TP) and accuracy comparison using (a) NVMe and (b) eMMC. *b* = largest possible batch size.

Summary. KVSwap maintains accuracy within an acceptable range across tasks, model sizes, modalities, context lengths, and disk types, consistently outperforming KV cache offloading baselines under the same low-level memory budget. Moreover, KVSwap-t remains robust under a very tight memory budget, while most baselines fail and become impractical for use.

5.2 LM Generation Throughput

Tab. 4 presents generation throughput with batch sizes ranging from 1 to 16 and context lengths of 16K and 32K. All, except FlexGen and vLLM, use a fixed per-batch KV memory budget of 1/13 of the full KV cache (see setting A in Tab. 1). FlexGen is unconstrained, as it offloads the full KV cache, while vLLM serves a throughput baseline under idealized memory. Loki and InfiniGen perform similarly due to their fine-grained per-token and per-head design, which yields poor I/O efficiency; their results are averaged for brevity. KVSwap outperforms all other offloading methods across storage devices while maintaining the best generation quality (Sec. 5.1). Throughput remains stable across context lengths from 16K to 32K since the number of selected KV entries is fixed, making I/O latency largely independent of sequence length.

FlexGen, Loki, and InfiniGen achieve only up to 1.9 and 0.1 tokens/s on NVMe and eMMC respectively due to heavy disk traffic (FlexGen) or poor bandwidth utilization (Loki and InfiniGen). KVSwap sustains 6.9 tokens/s (NVMe) and 5.9 tokens/s (eMMC) with batch size 1 at 16K context, rising to 35.1 and 15.8 tokens/s at batch size 8. On NVMe, throughput continues to scale, reaching 46.1 tokens/s at batch size 16, while eMMC saturates due to its limited bandwidth. Even under saturation, KVSwap maintains significant speedups over competing baselines. Variants of InfiniGen (InfiniGen* and InfiniGen*+ru) offer moderate improvements but remain far behind KVSwap. For instance, at batch size 4 and 16K context on eMMC, InfiniGen*+ru achieves 5.2 tokens/s versus 16.2 tokens/s for KVSwap, a 3.1× gain. These results confirm the effectiveness of the grouped KV access pattern we designed.

Finally, KVSwap can sometimes surpass vLLM on NVMe at large batch sizes or contexts by carefully trading LM generation accuracy for throughput. At context length 16K and batch size 16, it is 1.2× faster while using 11.0× less KV memory. Whereas vLLM saturates once its 31 GiB cache limit is exceeded, KVSwap delivers up to 2.0× higher throughput with substantially less KV cache memory at an increased 32K context length, a benefit that grows with workload scale.

A complete throughput comparison with additional 8K and 24K context lengths is given in Tab. 2 of Appendix B.

5.3 Impact of LM Model Sizes

Fig. 10 compares throughput for LLaMA3 (3B and 8B) and Qwen3-14B at a 32K context length with batch sizes of 1 and 8, given available KV cache memory for vLLM of around 43, 31, and 16 GiB. Across model sizes, KVSwap outperforms ShadowKV on eMMC, achieving speedups of over 1.8× (up to 2.1×) at batch size 1, and over 2.9× (up to 4.1×) at batch size 8. On NVMe, KVSwap achieves speedups of at least 1.3× (up to 1.7×) at batch size 8. Although throughput is comparable to ShadowKV on NVMe with batch size 1, KVSwap delivers much better generation quality, as discussed in Sec. 5.1. When compared with vLLM at batch size 8, KVSwap (NVMe) achieves throughput improvements of 1.1×, 1.7×, and 1.9× on the 3B, 8B, and 14B models, respectively. Notably, on the 14B model, KVSwap with the low-end eMMC disk even surpasses vLLM by 1.2× at batch size 8. These results demonstrate that KVSwap scales well as the model grows larger.

5.4 Best-case Configurations for Baselines

So far, we have set a consistent per-batch memory budget for each KV cache offloading scheme. We now compare KVSwap against offloading baselines using their recommended best-case configurations. Given total memory budgets of 2000 MiB and 800 MiB, we compare throughput using the largest possible batch size supported by each method, as described in

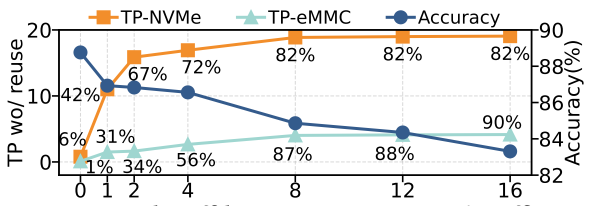


Figure 12: Trade-off between accuracy, I/O efficiency, and throughput (TP) under varying group sizes (x-axis).

Table 5: Reuse ratio and throughput (TP) statistics. "No Reu." = TP wo/ reuse. "↑" = TP improvement of w/ over wo/ reuse.

Disk	Dataset	Metric	Min	Max	Std	Avg	No Reu.	↑
NVMe	QMSum	Reuse(%)	76.2	79.1	0.7	77.3	-	2.1×
		TP(toks/s)	33.4	38.7	2.0	35.8	17.3	
	MSQue	Reuse(%)	75.3	81.2	1.1	76.2	-	2.0×
		TP(toks/s)	32.4	37.6	1.5	35.4	17.4	
eMMC	QMSum	Reuse(%)	76.1	79.2	0.7	77.2	-	4.0×
		TP(toks/s)	14.4	17.5	0.6	15.7	3.9	
	MSQue	Reuse(%)	76.3	79.4	0.7	77.6	-	3.8×
		TP(toks/s)	14.5	16.0	0.4	15.3	4.0	

Sec. 4.3-setting B. We also report the average accuracy on two task types of multi-document QA (MQA) and summarization (SUM) from LongBench, six tasks in total, for each method as a reference.

Fig. 11 presents the results, with vLLM numbers taken from Sec. 5.2 for direct comparison. KVSwap proves highly effective at trading a small loss in accuracy for substantially higher throughput and much lower memory usage. While vLLM, ShadowKV, and Loki achieve the top-3 accuracy, they do so at the cost of either large memory demand or low throughput. In contrast, KVSwap delivers clear advantages in both throughput and memory efficiency, with only marginal accuracy degradation. Compared with vLLM, KVSwap on NVMe reduces KV cache memory by 15.9×–39.7× while still providing 1.1× higher throughput, at a maximum accuracy drop of just 2.4%. Relative to ShadowKV, KVSwap achieves large throughput gains - 7.1× and 8.6× on eMMC, and 3.3× and 4.5× on NVMe - with accuracy drops ≤1.5%.

6 Analysis of Design Choices

To evaluate how our key design choices, including group size, KV reuse buffer and number of selected KV entries, affect the overall throughput and accuracy, we apply KVSwap to LLaMA3-8B under a 310 MiB per-batch KV memory budget. Accuracy is reported as the average over MV, QA1, and VT tasks from RULER, while throughput and latency are measured at a 32K context length with a batch size of 8.

Group size and system efficiency. Figure 12 shows how throughput and accuracy vary with KV prediction group

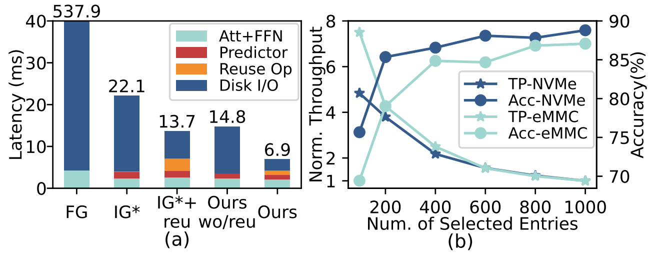


Figure 13: (a) Decoding latency breakdown on NVMe. (b) Trade-off between accuracy and throughput across KV selection sizes and disk types.

sizes (G). The number beside each point indicates I/O utilization. A group size of 0 disables our head aggregation strategy. This ablation study reports throughput *without* KV reuse to isolate our optimizations. As G increases, accuracy drops gradually from 88.8% to 83.3%, while throughput (without reuse) improves from 1.8 to 19.1 tokens/s on NVMe and 0.1 to 4.2 tokens/s on eMMC. When $G = 0$ or 1, both throughput and I/O utilization are low. This highlights that our grouped critical KV prediction is important for improving the throughput while maintaining generation accuracy. KVSwap provides an API to automatically obtain the best group size, but users can configure KVSwap to prioritize generation accuracy or throughput.

KV reuse and throughput. To analyse the characteristics and benefits of KV reuse, we evaluate 100 randomly sampled inputs: 50 from the QMSum meeting summarization dataset and 50 from the MuSiQue multi-document QA dataset, covering different use scenarios. Tab. 5 reports reuse rates and throughput across disk types and datasets. Reuse rates remain high across all cases, ranging from 75.3% to 81.2%. The throughput (in tokens/s) achieves 2.0×–2.1× speedups on NVMe and 3.8×–4.0× on eMMC compared to the no-reuse baseline. This indicates that our reuse buffer can effectively improve the throughput, especially for slower persistent storage. Similar trends hold across the evaluated workloads: reuse rates vary little across inputs (standard deviation ≤ 1.1%), and throughput variation remains bounded within 2.0 standard deviations on NVMe and 0.6 on eMMC, contributing to no more than 5.6% of the average throughput. These results justify using the average reuse rate during offline tuning (Appendix A.1).

Latency breakdown. Figure 13a reports the decoding latency of a single Transformer block. In FlexGen (FG), loading all KV entries from disk makes I/O the bottleneck. InfiniGen* (IG*) alleviates this with selective loading and our head aggregation scheme, but I/O latency still dominates. KVSwap without reuse (Ours wo/ reu) better utilizes disk bandwidth, reducing latency by 1.5×, already comparable to InfiniGen*+reuse (IG*+reu). With reuse enabled, I/O latency of KVSwap further drops by 4.3×, incurring only 1.0 ms reuse overhead, and achieves the lowest total latency of 6.9 ms.

Number of selected entries. Figure 13b shows normalized throughput versus the number of selected KV entries MG (omitting the number of KV heads H_{kv}). Increasing MG improves accuracy on NVMe and eMMC but reduces throughput. Beyond $MG = 400$, accuracy gains become marginal while throughput continues to drop, making $MG = 400$ the balanced trade-off and our default setting.

Rolling buffer. We compare generation accuracy with and without the rolling buffer (RB) on KV entry group sizes (G) 2, 4, 8 and 12 (see also Tab. 3 of Appendix B). Disabling RB causes a sharp accuracy drop of ≥29%, since new KV entries are generated token by token during decoding and often cannot immediately form a group for prediction. The RB allows newly generated entries to participate in computation in a timely manner, helping to preserve accuracy.

7 Discussion

Model weight optimization. KVSwap optimizes KV cache memory footprint. Weight-centric methods such as quantization [23, 37] and parameter offloading [10, 63] are orthogonal and complementary.

Low-bit KV. KVSwap is designed to preserve generation quality, but can be combined with low-bit KV compression [38, 41]. For comparison, 2-bit quantization yields an 8× reduction, while KVSwap achieves more than 30×.

Prefilling optimization. KVSwap can extend to prefilling by (1) computing attention only for important tokens, and (2) storing only critical KV entries to disk. It also complements prefill-oriented frameworks [29].

OS paging. Using OS-level paging is inefficient, as it is workload-agnostic and misses optimizations like selective KV loading, reduced transfer, and overlapped prefetching.

ShadowKV+reuse. While employing reuse to ShadowKV [52] may improve throughput, it cannot meet the tight on-device memory limits. By algorithm and system co-design, KVSwap improves both memory efficiency and system throughput.

8 Related Work

KVSwap builds on the following past foundations.

Sparse attention. Sparse attention [18] reduces the memory and compute overhead of attention and KV cache. Static methods apply fixed patterns to select KV entries [15, 19, 31, 70], reducing computation but risking loss of long-range dependencies and accuracy. Dynamic methods adapt token selection to the input, focusing on tokens that dominate attention scores [11, 33, 51, 54, 58, 61, 65, 69, 71, 72, 74]. KVSwap builds on this insight by identifying important KV entries at each step, but adapts it for on-device disk offloading. Unlike sparse attention, which assumes KV cache is fully in memory, KVSwap selectively preloads critical KV groups from disk into memory using two techniques: a compressed K cache (Sec. 3.2) to reduce memory overhead, and a grouped KV prediction algorithm (Sec. 3.3) to improve disk I/O.

KV-cache offloading and optimization. InfiniGen [36] and ShadowKV [52] move KV entries between GPU and CPU, while SpeCache [30] applies low-bit compression and speculative prefetching. llm-offload [60] and ArkVale [16] adopt page-based KV managers to recall evicted tokens. These *GPU-CPU offloading* methods assume high-bandwidth, low-latency PCIe connections, whereas mobile and embedded devices rely on NVMe, UFS, eMMC, or SD storages with orders-of-magnitude lower bandwidth and higher latency. This makes disk-based offloading a fundamentally harder problem. KVSwap is the first framework explicitly designed to tackle this challenge. Optimizations are also proposed to *time-to-first-token (TTFT)* to reduce startup delay. CacheBlend fuses cached knowledge for faster retrieval-augmented generation [66], IMPRESS [17] accelerates prefix KV loading on server-class SSDs, and CacheGen [39] compresses KV for faster transmission. In contrast, KVSwap targets iterative decoding on resource-constrained devices, where decoding throughput - not TTFT - is the primary bottleneck.

On-device inference. General-purpose LM serving frameworks such as llama.cpp [2] and MLC-LLM [4] enable inference on consumer hardware. Beyond these, recent work has explored memory-efficient serving through quantization and weight optimization. Other approaches exploit contextual sparsity [40], weight offloading [10, 63], or model specialization such as EdgeMoE [67]. System-level advances include LLMCad [62] for speculative decoding and architectural redesigns for mobile platforms [42, 43, 55, 56]. Most of these efforts focus on optimizing *model weights*. In contrast, KVSwap tackles the growing overhead of the *KV cache*, the main bottleneck for long-context inference and largely overlooked in on-device scenarios. It complements weight-centric optimizations to enable long-context LM inference on memory-limited devices.

9 Conclusion

We have presented KVSwap, a KV cache offloading scheme to support long-context on-device LM inference on mobile and embedded devices. KVSwap leverages non-volatile secondary storage as a swapping medium to offload KV cache data. Through the co-design of algorithms and the runtime system, it delivers substantial system efficiency improvements over existing KV cache offloading solutions while maintaining generation quality. KVSwap delivers good scalability under different context lengths, batch sizes, model sizes, and limited memory budgets.

References

[1] NVIDIA 2024. *Jetson-Orin*. NVIDIA. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/

[2] Georgi Gerganov 2024. *llama.cpp*. Georgi Gerganov. https://github.com/ggerganov/llama.cpp

[3] MediaTek 2025. *MediaTek Dimensity 9400 Plus*. MediaTek. https://www.mediatek.com/products/smartphones/mediatek-dimensity-9400-plus

[4] MLC Team 2024. *MLC-LLM*. MLC Team. https://github.com/mlc-ai/mlc-llm

[5] NVIDIA 2025. *NVIDIA Nsight Systems*. NVIDIA. https://developer.nvidia.com/nsight-systems

[6] NVIDIA 2025. *NVIDIA Tools Extension Library*. NVIDIA. https://github.com/NVIDIA/NVTX

[7] Orange Pi 2025. *Orange Pi 5 Pro*. Orange Pi. http://www.orangepi.org/html/hardWare/computerAndMicrocontrollers/details/Orange-Pi-5-Pro.html

[8] Adobe Inc. 2025. Adobe Acrobat AI Assistant: Generative AI Tool for PDF Summarization and Smart Q&A. https://www.adobe.com/acrobat/generative-ai-pdf.html.

[9] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245* (2023).

[10] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2023. Llm in a flash: Efficient large language model inference with limited memory. *arXiv preprint arXiv:2312.11514* (2023).

[11] Sotiris Anagnostidis, Dario Pavllo, Luca Biggio, Lorenzo Noci, Aurelien Lucchi, and Thomas Hofmann. 2023. Dynamic context pruning for efficient and interpretable autoregressive transformers. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2845, 22 pages.

[12] Apple Inc. 2025. Apple Intelligence. https://apple.com/apple-intelligence.

[13] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, et al. 2025. Qwen2.5-vl technical report. *arXiv preprint arXiv:2502.13923* (2025).

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

[15] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *CoRR* abs/2004.05150 (2020). arXiv:2004.05150 https://arxiv.org/abs/2004.05150

[16] Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. 2024. ArkVale: Efficient Generative LLM Inference with Recallable Key-Value Eviction. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/cd4b49379efac6e84186a3ffce108c37-Abstract-Conference.html

[17] Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. 2025. {IMPRESS}: An {Importance-Informed}{Multi-Tier} Prefix {KV} Storage System for Large Language Model Inference. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 187–201.

[18] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. *CoRR* abs/1904.10509 (2019). arXiv:1904.10509 http://arxiv.org/abs/1904.10509

[19] Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, and Furu Wei. 2023. LongNet: Scaling Transformers to 1,000,000,000 Tokens. arXiv:2307.02486 [cs.CL] https://arxiv.org/abs/2307.02486

[20] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[21] Carl Eckart and Gale Young. 1936. The approximation of one matrix by another of lower rank. *Psychometrika* 1, 3 (1936), 211–218.

[22] Fireflies.ai Corp. 2025. Fireflies.ai: AI Meeting Assistant for transcription, summaries, highlights and search. https://fireflies.ai/.

[23] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323* (2022).

[24] Google. 2025. Google Gemini in Gmail: AI-powered Email Summaries and Natural-Language Q&A. https://blog.google/products/gmail/.

[25] Google. 2025. Google Lens: Natural-language video and voice search. https://lens.google/.

[26] Google. 2025. Google Recorder: On-device audio transcription and summarization. https://recorder.google.com/.

[27] Gabriel Haas and Viktor Leis. 2023. What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2090–2102.

[28] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. RULER: What's the Real Context Size of Your Long-Context Language Models? *arXiv preprint arXiv:2404.06654* (2024).

[29] Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, et al. 2024. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Advances in Neural Information Processing Systems* 37 (2024), 52481–52515.

[30] Shibo Jie, Yehui Tang, Kai Han, Zhi-Hong Deng, and Jing Han. 2025. SpeCache: Speculative Key-Value Caching for Efficient Generation of LLMs. arXiv:2503.16163 [cs.CL] https://arxiv.org/abs/2503.16163

[31] Hongye Jin, Xiaotian Han, Jingfeng Yang, Zhimeng Jiang, Zirui Liu, Chia-Yuan Chang, Huiyuan Chen, and Xia Hu. 2024. LLM Maybe LongLM: SelfExtend LLM context window without tuning. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 888, 16 pages.

[32] Greg Kamradt. 2023. *Needle in a Haystack - Pressure Testing LLMs*. https://github.com/gkamradt/LLMTest_NeedleInAHaystack

[33] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The Efficient Transformer. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=rkgNKkHtvB

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

[35] Md Tahmid Rahman Laskar, Xue-Yong Fu, Cheng Chen, and Shashi Bhushan Tn. 2023. Building real-world meeting summarization systems using large language models: A practical perspective. *arXiv preprint arXiv:2310.19233* (2023).

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

[37] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. *Proceedings of Machine Learning and Systems* 6 (2024), 87–100.

[38] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2024. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *arXiv preprint arXiv:2405.04532* (2024).

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

[40] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. 2023. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*. PMLR, 22137–22176.

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

[42] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, et al. 2024. Mobilellm: Optimizing sub-billion parameter language models for on-device use cases. *arXiv preprint arXiv:2402.14905* (2024).

[43] Sachin Mehta, Mohammad Hossein Sekhavat, Qingqing Cao, Maxwell Horton, Yanzi Jin, Chenfan Sun, Seyed Iman Mirzadeh, Mahyar Najibi, Dmitry Belenko, Peter Zatloukal, et al. 2024. Openelm: An efficient language model family with open training and inference framework. In *Workshop on Efficient Systems for Foundation Models II@ ICML2024*.

[44] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[45] Jayashree Mohan, Rohan Kadekodi, and Vijay Chidambaram. 2017. Analyzing IO amplification in Linux file systems. *arXiv preprint arXiv:1707.08514* (2017).

[46] Otter.ai, Inc. 2025. Otter.ai: AI Meeting Agent for real-time transcription, summarization, and action-item extraction. https://otter.ai/.

[47] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of machine learning and systems* 5 (2023), 606–624.

[48] Qualcomm Technologies, Inc. 2024. Snapdragon 8 Elite Mobile Platform. https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-8-elite-mobile-platform.

[49] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

[50] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. PMLR, 31094–31116.

[51] Prajwal Singhania, Siddharth Singh, Shwai He, Soheil Feizi, and Abhinav Bhatele. 2024. Loki: Low-rank keys for efficient sparse attention. *arXiv preprint arXiv:2406.02542* (2024).

[52] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. 2024. Shadowkv: Kv cache in shadows for high-throughput long-context llm inference. *arXiv preprint arXiv:2410.21465* (2024).

[53] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774* (2024).

[54] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. 2024. QUEST: query-aware sparsity for efficient long-context LLM inference. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 1955, 11 pages.

[55] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118* (2024).

[56] Omkar Thawakar, Ashmal Vayani, Salman Khan, Hisham Cholakal, Rao M Anwer, Michael Felsberg, Tim Baldwin, Eric P Xing, and Fahad Shahbaz Khan. 2024. Mobillama: Towards accurate and lightweight fully transparent gpt. *arXiv preprint arXiv:2402.16840* (2024).

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[58] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 97–110. doi:10.1109/HPCA51647.2021.00018

[59] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[60] Jianbo Wu, Jie Ren, Shuangyan Yang, Konstantinos Parasyris, Giorgis Georgakoudis, Ignacio Laguna, and Dong Li. 2025. LM-Offload: Performance Model-Guided Generative Inference of Large Language Models with Parallelism Control. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 840–849. doi:10.1109/IPDPSW66978.2025.00134

[61] Wei Wu, Zhuoshi Pan, Chao Wang, Liyi Chen, Yunchu Bai, Tianfu Wang, Kun Fu, Zheng Wang, and Hui Xiong. 2025. TokenSelect: Efficient Long-Context Inference and Length Extrapolation for LLMs via Dynamic Token-Level KV Cache Selection. arXiv:2411.02886 [cs.CL] https://arxiv.org/abs/2411.02886

[62] Daliang Xu, Wangsong Yin, Xin Jin, Ying Zhang, Shiyun Wei, Mengwei Xu, and Xuanzhe Liu. 2023. Llmcad: Fast and scalable on-device large language model inference. *arXiv preprint arXiv:2309.04255* (2023).

[63] Zhenliang Xue, Yixin Song, Zeyu Mi, Le Chen, Yubin Xia, and Haibo Chen. 2024. PowerInfer-2: Fast Large Language Model Inference on a Smartphone. *arXiv preprint arXiv:2406.06282* (2024).

[64] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[65] Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. 2024. Gated linear attention transformers with hardware-efficient training. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 2333, 23 pages.

[66] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*. 94–109.

[67] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. 2023. Edgemoe: Fast on-device inference of moe-based large language models. *arXiv preprint arXiv:2308.14352* (2023).

[68] Shukang Yin, Chaoyou Fu, Sirui Zhao, Ke Li, Xing Sun, Tong Xu, and Enhong Chen. 2024. A survey on multimodal large language models. *National Science Review* 11, 12 (2024), nwae403.

[69] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. 2025. Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 23078–23097. doi:10.18653/v1/2025.acl-long.1126

[70] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big bird: transformers for longer sequences. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1450, 15 pages.

[71] Biao Zhang, Ivan Titov, and Rico Sennrich. 2021. Sparse Attention with Linear Units. arXiv:2104.07012 [cs.CL]

[72] Haojie Zhang, Zhenyu Wu, Zhenyu Zhang, Shixing Liu, Zujie Ren, Jingji Chen, Mingjie Zhan, Zirui Liu, Chen-Yu Lee, Yuchen Zhang, Haichuan Yang, Yuxiang Liu, Yafan He, and Zhaofeng He. 2024. SemSA: Semantic Sparse Attention is hidden in Large Language Models.. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=eG9AkHtYYH

[73] Haopeng Zhang, Philip S Yu, and Jiawei Zhang. 2025. A systematic survey of text summarization: From statistical methods to large language models. *Comput. Surveys* 57, 11 (2025), 1–41.

[74] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2023. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems* 36 (2023), 34661–34710.

[75] Junjie Zhou, Yan Shu, Bo Zhao, Boya Wu, Zhengyang Liang, Shitao Xiao, Minghao Qin, Xi Yang, Yongping Xiong, Bo Zhang, et al. 2025. Mlvu: Benchmarking multi-task long video understanding. In *Proceedings of the Computer Vision and Pattern Recognition Conference*. 13691–13701.

[76] Jinguo Zhu, Weiyun Wang, Zhe Chen, Zhaoyang Liu, Shenglong Ye, Lixin Gu, Hao Tian, Yuchen Duan, Weijie Su, Jie Shao, et al. 2025. Internvl3: Exploring advanced training and test-time recipes for open-source multimodal models. *arXiv preprint arXiv:2504.10479* (2025).

A Details of KVSwap's Parameter Search

As described in the main text, KVSwap provides an API to search for a set of parameters to be used by the KVSwap runtime. Search is achieved through a combination of heuristics and a greedy-based search parameter solver using profiling informaiton.

A.1 Precomputed parameter lookup tables

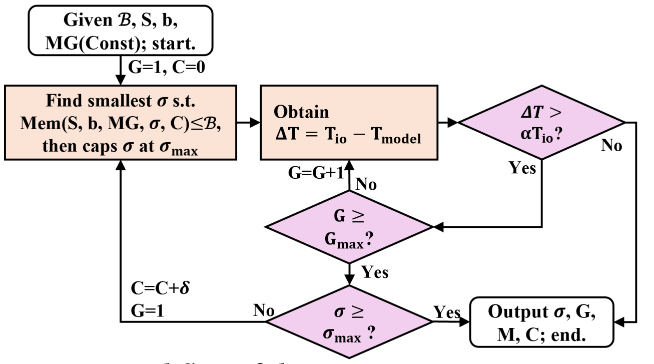
To reduce tuning cost, the KVSwap API allows precomputing hardware-independent parameters on a high-performance server. This is done by profiling the LM on selected datasets, with results stored in two *lookup tables*: (1) reuse buffer capacity C to reuse rate, and (2) compression ratio σ to the low-rank adapter for building the compressed K cache. As shown in Sec.6, reuse rates for a given C are largely input-invariant, so we store the average value to accelerate tuning. By default, precomputation uses the C4 dataset [49] with 20 sampled batches, though both dataset and sample size are user-configurable via the API.

A.2 Number of selected entries

Recall that KVSwap predicts and loads important KV cache entries in groups. For a group size G and a selected group number M, the size of the KV preloading buffer is $H_{kv}MG$, where H_{kv} is the number of KV heads. While $H_{kv}MG$ affects the I/O latency, its impact on memory is negligible because: (a) our layerwise preloading strategy allows this buffer to be shared across all layers, and with KV reuse enabled, the buffer is merged into the reuse buffer; and (b) prior work [52, 74] shows that critical KV entries are sparse, typically comprising less than 5% of the long context. This allows us to preset $H_{kv}MG$ to reduce the search space with minimal accuracy loss. As H_{kv} is a model constant, we control the number of selected entries by fixing $MG = \texttt{Const}$ (e.g., $\texttt{Const} = 400$).

A.3 Metric measurement

We use sampled profiling to obtain two metrics, *I/O delay* and *model delay*, which depend on the underlying hardware and the target LM. In this work, we configure KVSwap to use NVIDIA NVTX [6] and Nsight Systems [5] to collect timing information for the core functions (instrumentation is enabled by turning on an KVSwap option) of the KVSwap runtime, but other profiling tools can be used by overwriting a KVSwap method. Profiling is performed by sampling different combinations, (b, S), of batch sizes b, and context lengths S. Specifically, we construct the sets $\{b\}$ and $\{S\}$ by sweeping $b \in [1, b_{\max}]$ and $S \in [S_{\min}, S_{\max}]$, where $S_{\min}$ is a predefined lower bound context length (e.g., 4K). By default, the steps for b and S are set to 1 and 2K, respectively, but may be increased when $b_{\max}$ or $S_{\max}$ is excessively large to



Given $\mathcal{B}$, S, b, MG(Const); start.
G=1, C=0
Find smallest σ s.t. Mem(S, b, MG, σ, C)$\leq\mathcal{B}$, then caps σ at $\sigma_{\max}$
Obtain $\Delta T = T_{io} - T_{model}$
$\Delta T > \alpha T_{io}$?
No
Yes
G=G+1
No
$G \geq G_{\max}$?
Yes
C=C+δ
G=1
No
$\sigma \geq \sigma_{\max}$?
Yes
Output σ, G, M, C; end.

Figure 1: Workflow of the KVSwap runtime parameter solver for parameter tuning.

reduce the time cost of exhaustive profiling. We apply interpolation on the measured configurations to estimate the values of missing points. In addition to (b, S), we sweep the compression ratio σ and reuse buffer capacity C over the keys of lookup tables (Sec. A.1), which are designed to span a sufficiently large range and fine granularity.

Specifically, for given b, S, σ, C, the number of selected groups M, group size G, and $MG = \texttt{Const}$, we profile KVSwap when serving the target LM on a sample data:

The I/O delay: $T_{\text{io}}(b, \texttt{Const}, G, C)$ measures disk load time for KV entries. The effect of C is modeled by averaging the latency using multiple randomly sampled reuse patterns according to the lookup reuse rate. We omit incremental disk updates because their latency is small and hidden within the compute–I/O pipeline.

The model delay: $T_{\text{model}}(b, \texttt{Const}, C, S, \sigma)$ covers the time spent on standard transformer blocks (i.e., attention and FFN), and the overhead introduced by KVSwap's prediction and reuse-buffer management. Attention time depends on b and $\texttt{Const}$; prediction time on b, σ, and S; and reuse management cost on b, C, and $\texttt{Const}$. Compact K cache and rolling buffer updates are negligible.

As a Transformer-based LM typically consists of multiple stacked Transformer blocks, we only need to profile a single block as a representative, which takes only a few seconds per parameter combination.

A.4 Parameter solver

Figure 1 shows the workflow of the KVSwap parameter solver. It first determines the compression ratio σ so that the memory budget $\mathcal{B}$ is not exceeded. It then finds the smallest possible G that can hide I/O latency within computation.

Overlapping I/O and computation. I/O latency can be substantial on slower disks (e.g., eMMC) or under batched processing, and fully hiding I/O may severely degrade generation quality. To balance this trade-off, we introduce a relaxation factor α, permitting only a $(1 - \alpha)$ fraction of I/O

Table 1: Relative accuracy loss (%) of Qwen3-8B compared to Full-KV on RULER (left) and LongBench (right).

Methods	S1	S2	MK1	MQ	MV	QA1	QA2	VT	Avg.	SQA	MQA	SUM	FSL	SYN	COD	Avg.
Full-KV (raw %)	100.0	100.0	100.0	99.8	98.8	79.0	67.0	100.0	93.1	41.6	44.6	26.0	69.8	100.0	56.9	56.5
InfiniGen	−100.0	−100.0	−100.0	−99.8	−98.8	−67.0	−48.0	−99.8	−89.2	−29.7	−33.4	−18.7	−32.5	−91.3	−46.7	−42.1
InfiniGen*	−56.0	−100.0	−99.0	−99.8	−98.8	−34.0	−18.0	−38.2	−68.0	−11.1	−6.6	−4.7	−8.3	−41.0	−13.6	−14.2
Loki	**0.0**	**0.0**	**0.0**	−1.0	−6.3	−5.0	**−5.0**	−3.0	−2.5	−2.5	**+0.5**	**+1.2**	−0.9	**0.0**	−3.7	**−0.9**
ShadowKV	−2.0	−2.0	−6.0	−4.5	−17.5	−13.0	−12.0	−19.6	−9.6	−4.6	−1.3	−0.7	−0.8	−2.0	−4.6	−2.3
KVSwap $^{\text{NVMe}}$	**0.0**	**0.0**	**0.0**	−0.5	**+0.5**	**−2.0**	**−5.0**	**0.0**	**−0.9**	−1.3	−0.7	−1.1	−1.3	**0.0**	**−1.2**	**−0.9**
KVSwap $^{\text{eMMC}}$	**0.0**	**0.0**	**0.0**	**−0.3**	+0.2	−4.0	−9.0	−0.4	−1.7	**−0.9**	−1.3	−1.1	**−0.6**	**0.0**	−2.4	−1.1
Loki-t	−100.0	−100.0	−100.0	−99.8	−98.8	−59.0	−44.0	−93.8	−86.9	−14.1	−10.9	−4.8	−8.3	−25.5	−27.1	−15.1
ShadowKV-t	−100.0	−98.0	−96.0	−97.0	−96.3	−38.0	−24.0	−100.0	−81.2	−15.7	−15.3	−8.4	−10.3	−44.0	−16.0	−18.3
KVSwap-t$^{\text{NVMe}}$	**0.0**	**0.0**	**0.0**	**0.0**	−1.3	**−15.0**	−12.0	**−0.2**	**−3.6**	**−1.8**	−2.7	**−1.4**	**−1.7**	**0.0**	−2.9	**−1.8**
KVSwap-t$^{\text{eMMC}}$	**0.0**	**0.0**	**0.0**	−0.5	**−1.0**	−21.0	**−11.0**	−1.6	−4.4	−2.3	**−1.6**	**−1.4**	−2.6	**0.0**	**−2.7**	**−1.8**

Table 2: Full throughput (tokens/s) comparison of LLaMA3-8B across batch sizes and context lengths (CLs).

Disks	Methods	CL=8K					CL=16K					CL=24K					CL=32K				
		b=1	**b=2**	**b=4**	**b=8**	**b=16**	**b=1**	**b=2**	**b=4**	**b=8**	**b=16**	**b=1**	**b=2**	**b=4**	**b=8**	**b=16**	**b=1**	**b=2**	**b=4**	**b=8**	**b=16**
eMMC	FlexGen	0.2	0.2	0.2	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
	Loki/InfiniGen	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
	InfiniGen*	1.7	1.6	1.5	1.5	1.5	1.7	1.7	1.5	1.4	1.4	1.6	1.5	1.4	1.4	1.4	1.6	1.5	1.4	1.4	1.4
	InfiniGen*+ru	4.3	5.2	5.7	4.6	3.5	4.0	4.7	5.2	4.2	3.7	4.2	4.5	5.1	4.4	3.5	4.2	4.4	4.9	4.5	3.5
	ShadowKV	2.7	3.7	4.2	4.5	4.4	3.0	3.8	4.1	4.4	3.4	2.8	3.5	4.1	4.1	3.4	3.0	3.7	4.2	3.8	3.4
	KVSwap	**5.9**	**10.8**	**16.1**	**18.7**	**15.0**	**5.9**	**10.5**	**16.2**	**15.8**	**11.2**	**6.0**	**10.4**	**15.3**	**15.7**	**11.1**	**6.0**	**10.3**	**15.2**	**15.7**	**10.9**
NVMe	FlexGen	1.6	1.6	1.6	1.6	1.6	0.8	0.8	0.8	0.8	0.8	0.5	0.5	0.5	0.5	0.5	0.4	0.4	0.4	0.4	0.4
	Loki/InfiniGen	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.8	1.9	1.9	1.9	1.8	1.8
	InfiniGen*	5.2	8.8	10.9	12.4	13.9	5.0	8.4	10.8	11.3	13.1	5.3	8.2	11.0	11.8	13.5	5.1	7.5	8.9	10.7	12.0
	InfiniGen*+ru	5.5	9.6	16.8	25.0	30.8	5.2	9.2	14.9	22.5	26.6	5.2	8.7	13.1	19.4	23.1	4.9	8.5	11.9	14.3	17.2
	ShadowKV	6.5	11.1	16.9	22.2	26.7	6.4	10.4	16.3	21.9	26.7	6.4	10.0	16.1	20.4	25.8	6.4	10.0	16.3	21.5	26.2
	KVSwap	**6.8**	**12.0**	**20.0**	**35.7**	**48.2**	**6.9**	**11.9**	**20.8**	**35.1**	**46.1**	**7.0**	**11.8**	**21.3**	**35.8**	**45.2**	**6.9**	**11.9**	**21.4**	**35.6**	**46.8**
None	vLLM	10.0	19.3	35.4	55.2	81.7	9.7	17.1	28.4	41.2	39.5	9.2	14.8	24.0	32.1	30.0	9.2	14.1	21.0	20.8	23.2

Table 3: Accuracy (%) w/ and w/o rolling buffer (RB) for different KV prediction/selection group sizes (G).

	G=2	G=4	G=8	G=12
With RB	86.9	86.6	84.9	84.4
No RB	57.9	41.1	34.9	30.6

to be hidden within computation. If hiding fails at $G_{\max}$, the solver reallocates part of the memory budget to the reuse buffer (increasing C by δ) to reduce I/O volume, while adjusting σ to stay within budget and restarting the search from $G = 1$. The search stops once $(1-\alpha)$ of I/O is overlapped with computation or the limits $\sigma_{\max}$ and $G_{\max}$ are reached. Generation quality is preserved by keeping σ and G as small as possible, with $\sigma_{\max}$ and $G_{\max}$ preventing excessive accuracy degradation under tight memory budgets.

Record solutions. KVSwap records the optimal parameter setting for each batch size b and context length S pair (b, S).Given $\mathcal{B}$ and MG = Const, the solver sweeps over (b, S) pairs and stores every solution. At runtime, parameters are retrieved by exact match; if unavailable, the nearest pair is chosen. Since users only specify the maximum memory budget $\mathcal{B}$max and the maximum batch size $b$max, actual memory usage scales with b. To account for this, we allocate a uniform per-batch budget of $\mathcal{B}$max$/b$max.

B More Evaluation Results

Tab. 1 compares the generation accuracy of Qwen3-8B on RULER and LongBench. Tab. 2 presents the generation throughput on NVMe and eMMC with batch sizes of 1, 2, 4, 8, 16, and context lengths of 8K, 16K, 24K and 32K for LLaMA3-8B. Tab. 3 compares the average generation accuracy with and without the rolling buffer on KV entry group sizes (G) of 2, 4, 8, and 12, for LLaMA3-8B over MV, QA1, and VT datasets from RULER.