# KVDrive: A Holistic Multi-Tier KV Cache Management System for Long-Context LLM Inference

Anonymous Author(s)

## Abstract

Supporting long-context LLMs is challenging due to the substantial memory demands of the key–value (KV) cache. Existing offloading systems store the full cache in host memory and selectively fetch critical entries during decoding, but this strategy quickly hits a ceiling: sparsity cannot be pushed further without degrading accuracy. As a result, when context length and batch size grow, the volume of KV transfers rises sharply and becomes the dominant source of decoding latency. We present KVDrive, a holistic multi-tier KV cache management system spanning GPU memory, host DRAM, and SSD. Unlike prior work that pursues greater sparsity through algorithmic refinements, KVDrive tackles the problem from a systems perspective—jointly orchestrating cache placement, pipeline scheduling, and cross-tier coordination to sustain high-throughput inference under tight GPU budgets. KVDrive advances three fundamental capabilities: it *adapts cache management to attention behavior* to maximize reuse and minimize redundant data movement; it *restructures the decoding pipeline* to overlap I/O- and CPU/GPU compute-bound stages, eliminating stalls across heterogeneous resources; and it *harmonizes data movement across memory tiers* to unlock scalable long-context inference far beyond GPU and DRAM limits. We have implemented a fully functional prototype of KVDrive and evaluated it on long-context benchmarks with popular LLMs. The system achieves up to 1.74× higher throughput compared to state-of-the-art works while preserving accuracy.

## Keywords

Large language model, long-context serving, KV cache offloading.

**ACM Reference Format:**

Anonymous Author(s). 2018. KVDrive: A Holistic Multi-Tier KV Cache Management System for Long-Context LLM Inference. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

As large language models (LLMs) continue to scale in both capability and adoption, supporting long-context inference [21] has become increasingly important for applications such as document understanding, complex agent workflows, software development [11], and reasoning over large knowledge bases.

A central obstacle to long-context inference is the memory footprint of the key-value (KV) cache. During autoregressive decoding, LLMs must retain the keys and values of all preceding tokens to enable attention computation. Unlike model weights, which remain fixed, the KV cache grows linearly with sequence length and batch size, and can easily surpass the size of the model itself. Efficient KV cache management has therefore emerged as a critical challenge for enabling long-context LLMs.

For example, Llama-3.1-8B-Instruct [23] supports sequences of up to 128k tokens—roughly 200 pages of text—requiring a KV cache exceeding 16 GB. Emerging models extend context windows to 1M tokens [34], further amplifying this demand. In contrast, commodity GPUs provide only tens of gigabytes of memory, which must also accommodate model weights, activations, and other runtime overheads. This memory gap makes it infeasible to store the entire KV cache in GPU memory for long-context or large-batch inference.

To bridge this gap, recent studies propose offloading KV caches to host memory [6, 7, 18, 28, 29, 40]. Before each attention computation, the needed entries are reloaded into GPU memory. While offloading alleviates GPU pressure, it introduces new inefficiencies across the system. Existing approaches fail to coordinate CPU and GPU computation, data transfer, and storage effectively, often leaving one or more resources underutilized—GPU memory idles while waiting for data, CPUs stall during GPU execution, or I/O bandwidth remains unused between transfers. As a result, they fail to achieve a holistic balance among GPU, CPU, and I/O subsystems, preventing them from reaching an optimal coordination point.

To address these challenges, we propose KVDrive, a holistic multi-tier KV cache management system spanning GPU memory, host DRAM, and SSD. Unlike prior work that relies on algorithmic sparsity refinements, KVDrive approaches the problem from a systems perspective—jointly optimizing cache management, pipeline scheduling, and storage tiering for efficient long-context inference under tight GPU budgets. It introduces three key techniques:

**(1) Attention-Based Cache Management.** KVDrive rethinks conventional cache management—traditionally guided by generic access frequency or recency—by making it attention-aware and tailored to the transformer architecture. Particularly, although the exact set of critical KV entries varies across tokens, they exhibit temporal locality within decoding windows. Leveraging this property, KVDrive maintains a sliding window of critical entries in GPU memory, incrementally updating only out-of-window differences. It depends on a 2D layer–head cache allocation and lookahead eviction to maximize reuse while keeping memory overhead bounded.

**(2) Elastic Pipeline Scheduling.** KVDrive decouples selection, fetching, and computation into independently scheduled stages through a new *SFC disaggregation* design. Leveraging the distinct characteristics of each stage, the system partitions decoding into micro-batches and executes these stages in parallel with minimal interference. Fine-grained micro-batching overlaps the memory-,



*Conference acronym 'XX, Woodstock, NY*



ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

transfer-, and compute-bound stages, while the index size, cache size, and micro-batch size are jointly tuned to balance latency, throughput, and accuracy. This design effectively eliminates pipeline stalls and sustains high utilization across GPU, CPU, and I/O subsystems under diverse workloads.

**(3) Coordinated Multi-Tier KV Storage.** Extending beyond DRAM-only offloading, KVDrive incorporates SSD as a third tier and coordinates data movement across HBM, DRAM, and SSD. It applies *importance-guided warm-up* to prioritize high-value KV entries during prefill, employs an *SSD-aware layout* to maximize sequential I/O locality, and performs *parallel sparse synchronization* to minimize cross-tier transfer overhead. Together, these mechanisms enable scalable long-context inference well beyond the memory capacity of GPU and DRAM alone.

In summary, this paper makes the following contributions:

- We present KVDrive, a holistic multi-tier KV cache management system that sustains efficient long-context LLM inference under tight GPU cache budgets.
- We introduce an attention-aware cache management mechanism that enables efficient reuse of KV entries in GPU memory, substantially reducing redundant data movement.
- We propose an elastic pipeline scheduling strategy that decouples selection, fetching, and computation, achieving fine-grained overlap and eliminating stalls.
- We design a coordinated multi-tier storage architecture that provides low-latency access and scalable support for long contexts beyond GPU and DRAM capacity.
- We implement and evaluate KVDrive on long-context benchmarks, demonstrating up to 1.74× higher throughput compared to existing systems, while preserving accuracy.

# 2 Background & Related Work

## 2.1 KV Cache Basics

Modern LLMs [25, 30] are typically built on decoder-only Transformers [9]. Inference is divided into two phases: *prefill* and *decoding*. During prefill, all input tokens (the prompt) are processed in parallel. This phase generates the first output token while storing intermediate key and value vectors in GPU memory, collectively referred to as the KV cache. In the subsequent decoding phase, tokens are generated autoregressively, with each step appending new key and value vectors to the cache. This design eliminates redundant computation and enables efficient token generation.

## 2.2 Sparse Attention

Not all tokens contribute equally to attention: tokens with higher attention scores typically play a more critical role [8, 15, 17, 22, 29, 35, 38]. Their corresponding key–value pairs are referred to as *critical* KV entries, and discarding non-critical entries generally leads to only minor accuracy loss [41]. Since token criticality depends on the current query, the system must dynamically identify which KV entries are relevant. For example, as shown in Figure 1, given a context "ABCDE", when decoding the next token, Head 1 in Layer 31 identifies the KV entries of tokens "A" and "E" as critical. As a representative system, Quest [29] partitions key entries into chunks and estimates their importance by multiplying the query vector with the channel-wise minimum and maximum of the keys.

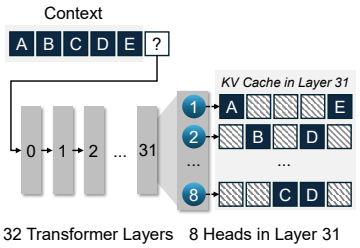



**Figure 1: An example of sparse attention in a 32-layer, 8-head model.**

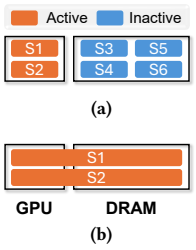



**Figure 2: Two types of offloading systems.**

By retrieving only the Top-K important chunks per query, Quest substantially reduces attention computation.

## 2.3 KV Cache Offloading

The KV cache size grows linearly with context length and batch size, quickly exceeding GPU memory capacity. To mitigate this bottleneck, recent studies propose offloading KV caches from GPU memory to slower but larger tiers such as host memory or SSDs.

Several systems [5, 10, 14, 32] focus on multi-session scenarios, where each session's KV cache fits in GPU memory but the aggregate demand across sessions exceeds capacity. Inactive-session caches are offloaded while the active session remains on GPU. For example, as shown in Figure 2a, the KV caches of four inactive sessions (S3–S6) are stored in DRAM while those of two active sessions (S1–S2) remain in GPU memory. These methods primarily aim to reduce time-to-first-token. However, they assume the active session's cache always fits in GPU memory—an assumption that fails in long-context inference.

More recent works [6, 7, 18, 28, 29, 40] instead offload the active sessions' KV caches to host memory, such as S1 and S2 in Figure 2b. During decoding, each token typically requires three steps: (1) *selecting* critical KV entries via the index stored in the GPU memory; (2) *fetching* these entries from host memory to GPU memory; (3) *computing* the new token with sparse attention and other layer operations (e.g., feed-forward networks).

The naive solution is to store all keys in GPU memory as the index and offload all values to host memory. Critical entries are then selected by multiplying each query with the full set of keys and retrieving the Top-K. To improve efficiency, three optimization strategies have been explored: (i) *Column selection*: InfiniGen [18] selects a subset of key columns with the largest magnitudes. (ii) *Spatial chunking*: Quest [29] and ShadowKV [28] partition adjacent keys into chunks and use their min/max/mean values as representatives. (iii) *Similarity grouping*: MagicPig [7] employs Locality-Sensitive Hashing to group similar keys, while RetrievalAttention [20], RetroInfer [6], and PQCache [40] leverage general Approximate Nearest Neighbor Search (ANNS) techniques, which use cluster centroids as index representatives for efficient key retrieval.

As summarized in Table 1, KVDrive advances KV cache offloading along three dimensions: (i) **Caching.** Existing systems such as ShadowKV [28], PQCache [40], and RetroInfer [6] employ generic cache management policies—typically LFU or LRU—that rely solely on past access frequency or recency. In contrast, KVDrive introduces an *attention-aware* cache manager that maintains its in-GPU

**Table 1: Comparison between several representative KV cache offloading systems and our KVDrive.**

| | Naive | Quest [29] | ShadowKV [28] | InfiniGen [18] | MagicPIG [7] | PQCache [40] | KVDrive (Ours) |
|---|---|---|---|---|---|---|---|
| Caching | × | × | LRU | × | × | LFU/LRU | **Attention-Based Cache Management** |
| Scheduling | Sequential | Sequential | Sequential | Partial Pipeline w/o Fetching Stalls | Sequential | Sequential | **Elastic Pipeline w/ Minimized Stalls** |
| Tiering | DRAM | DRAM | DRAM | DRAM | DRAM | DRAM | **Coordinated Multi-Tier KV Storage** |

cache based on real-time attention distributions and model-specific architectural patterns, enabling effective reuse aligned with the model's behavior. (ii) **Scheduling.** Most prior systems follow a sequential pipeline of selection, fetching, and computation, which leads to frequent GPU stalls. InfiniGen [18] attempts to mitigate this via speculative prefetching based on the previous layer's attention. However, this approximation often degrades accuracy in long-context tasks. KVDrive instead employs an *elastic pipeline scheduler* that overlaps these stages at fine granularity, effectively reducing stalls and sustaining high utilization under diverse workloads without compromising generation quality. (iii) **Tiering.** Prior systems like FlexGen [27] perform coarse-grained, layer-wise offloading. This is inefficient for sparse attention workloads, as fetching unused KV blocks causes severe I/O amplification. KVDrive instead adopts *parallel sparse synchronization*, fetching only the specific KV blocks required by each query. This fine-grained approach, coupled with our coordinated multi-tier design, allows KVDrive to extend the storage hierarchy to SSDs efficiently, mitigating the bandwidth bottleneck that limits coarse-grained offloading solutions.

These three mechanisms are not isolated optimizations but are integrated to fundamentally change how long-context KV cache management is performed: the elastic scheduler hides the latency of the sparse I/O, while the attention-guided cache minimizes the volume of that I/O, enabling scalable inference cross heterogeneous storage hierarchies.

## 3 Motivation

The memory footprint of the KV cache grows linearly with both context length and batch size, which quickly becomes prohibitive in practice. This linear growth makes directly storing all KV caches in GPU memory impractical under current hardware constraints. For instance, with a batch size of 8 and a context length of 100K, the KV cache of Llama-3-8B requires nearly 100GB of memory—beyond the capacity of most commodity GPUs.

As discussed in § 2.3, prior studies have proposed various KV cache offloading systems. We select Quest [29], RetroInfer [6], and ShadowKV [28] as representative systems, RULER [13] as a benchmark, and an L20 server (§ 9.1). All baselines are implemented following their original papers; however, to ensure a fair comparison, we disable several auxiliary optimizations. Details of these adjustments and the hyperparameter are provided in § 9.1. In the following, we identify three fundamental limitations that hinder their practicality and performance in large-scale deployments under realistic long-context and multi-batch settings.

First, most existing systems [18] load a fresh set of critical KV entries from host memory at every decoding step, discarding those previously fetched into GPU. Recent studies [28, 40] have observed short-range temporal locality—i.e., consecutive queries often attend to overlapping critical KV entries—yet this has not been systematically analyzed in the long range.

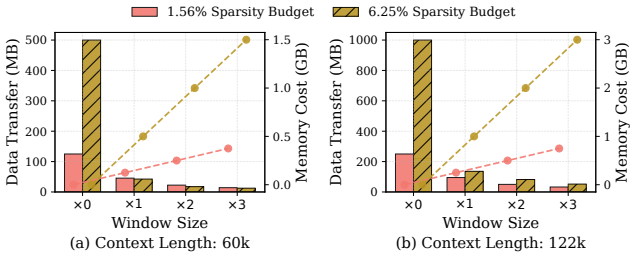



**Figure 3: Effect of critical KV windows with different window sizes for Llama-3-8B under 1.56% and 6.25% budgets. Expanding the critical KV window significantly reduces data transfer with minimal memory overhead.**

> Finding 1. *Critical KV entries exhibit strong temporal correlation not only between adjacent tokens but also across a broader local range. Maintaining a sliding window of recent critical KV entries therefore enables effective reuse.*

We define a *critical KV window* for each decoding step as the set of critical KV entries corresponding to multiple recent tokens. To evaluate temporal reuse, we measure (i) the additional GPU memory overhead and (ii) the amount of reloaded data, defined as the non-overlapping KV entries between adjacent windows.

We denote the window configuration as "×$N$", where $N$ represents how many times the window size exceeds the per-step sparsity budget. Here, "×0" indicates that no KV entries are retained after each decoding step (i.e., the cache is cleared entirely), "×1" retains one step's worth of critical entries, and larger values such as "×2" correspond to sliding windows that cover multiple recent steps.

Most existing offloading systems follow a per-step sparsity paradigm: each decoding step retrieves its corresponding critical KV entries, uses them once, and then discards them ("×0"). A few methods, such as ShadowKV [28] and PQCache [40], retain only the most recent step's entries ("×1"), achieving modest reuse. As illustrated in Figure 3, our analysis shows that maintaining a sliding window of multiple steps (e.g., "×2" and "×3") yields substantially higher benefits. With a sparsity budget of 6.25%, increasing the window size from ×0 to ×3 reduces host–GPU transfers from over

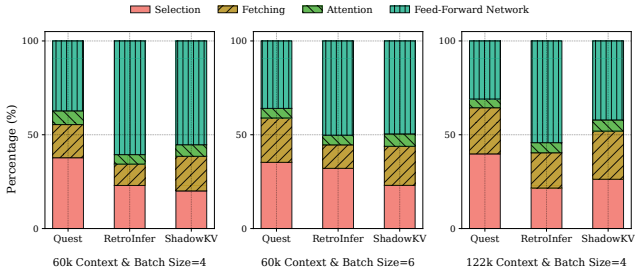



**Figure 4: Time breakdown of the three representative offloading systems under different context lengths and batch sizes under a sparsity budget of 1.56%. KV selection and fetching dominate decoding latency, creating significant GPU stalls that worsen with larger batches and contexts.**

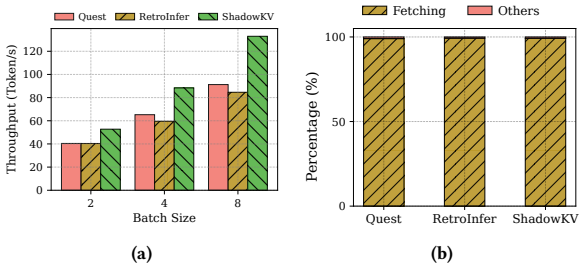



**Figure 5: Throughput scaling under DRAM-only and disk-backed offloading for a batch size of 8 and 122k context. (a) Throughput increases with batch size. (b) A strawman operates beyond DRAM limits but suffers from severe GPU–SSD bandwidth bottlenecks.**

500 MB to below 12.5 MB per step. This observation indicates that a small fraction of additional GPU memory, typically available after accounting for model parameters and intermediate activations, can be effectively repurposed to cache recently used KV entries.

> Finding 2. *KV selection and fetching together account for a substantial portion of the decoding latency. Their sequential execution creates pronounced GPU pipeline stalls, and their impact increases with larger batch sizes and longer contexts.*

As shown in Figure 4, both KV selection and fetching together account for nearly 50% of the total runtime under realistic context lengths and batch sizes. This bottleneck arises mainly from the high computational cost of selection and the data movement in fetching, and it becomes increasingly pronounced as the batch size or context length grows. During selection, different systems adopt distinct indexing schemes with varying computational overhead. Quest [29] evaluates each query against both the minimum and maximum keys within every chunk, resulting in the longest selection latency. In contrast, ShadowKV [28] compares queries only with the mean key of each chunk, while RetroInfer [6] uses cluster centroids, both of which reduce the selection cost. Following selection, the fetching stage must transfer the identified KV entries from host memory to GPU memory before attention computation can proceed. Because most existing systems execute these stages sequentially—select, then fetch, then compute—both phases block GPU execution and cause substantial stalls. Consequently, although offloading alleviates GPU memory pressure, these stalls dominate overall runtime at scale and prevent existing systems from efficiently handling long contexts or large batches.

> Finding 3. *Due to the limited bandwidth between GPU and disk compared to GPU–DRAM transfers, existing offloading systems struggle to efficiently migrate KV caches to disk, leading to severe throughput degradation.*

As shown in Figure 5a, when ample host memory is available, increasing the batch size significantly improves throughput, since wider batches better amortize memory accesses and parallelize computation. However, under realistic deployment constraints—where a typical data-center GPU node is provisioned with around 100 GB of host memory [12, 24] and edge devices often provide even less (tens of gigabytes)—the KV cache quickly exhausts available capacity once both long contexts and large batches are considered. Throughput then plateaus and eventually fails due to out-of-memory errors when host memory becomes saturated. Thus, systems that offload KV caches only to DRAM cannot fully exploit GPU compute power: memory pressure inevitably forces cache spillover to SSD, and without effective multi-tier management, decoding latency once again becomes dominated by data movement rather than computation. For example, as illustrated in Figure 5b, we implement a strawman design inspired by FlexGen [27], which stores the KV cache on disk using memory mapping. During inference, each layer loads its corresponding KV entries from SSD into GPU memory on demand, performs attention computation, and immediately evicts them back to disk after use. While this approach enables operation beyond DRAM capacity, it exposes the severe bandwidth gap between GPU–SSD and GPU–DRAM transfers, resulting in extremely limited throughput and frequent GPU stalls.

## 4 System Overview

As illustrated in Figure 6, KVDrive aims to support high-throughput long-context LLM inference despite tight GPU memory constraints. When the KV cache exceeds available GPU capacity, it is offloaded to host DRAM or SSDs and the system constructs an index in the GPU memory during the prefill phase (0). During decoding, each new token follows a three-stage workflow: selecting critical KV entries based on the index (1); fetching the selected entries from DRAM or SSD into GPU (2); and performing attention and feed-forward computation using the fetched KV entries (3).

KVDrive adopts indexing and sparse attention mechanisms following best practices from prior work (§ 2.3)—combining spatial chunking and similarity grouping through a new *hierarchical* design that organizes a lightweight index in a content-aware manner. Specifically, the KV cache is partitioned into chunks, and the mean key of each page is used as its representative [28, 29], forming higher-level centroids for similarity grouping. Unlike global

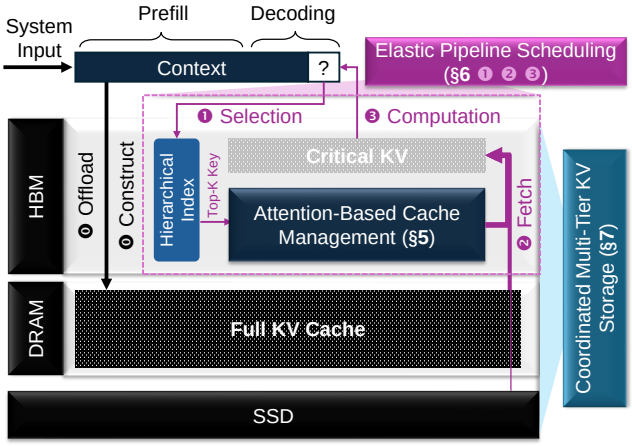



**Figure 6: System architecture. The prefill phase involves offloading the full KV cache to DRAM/SSD and constructing a hierarchical index. During decoding, the Elastic Pipeline Scheduling module orchestrates a three-stage pipeline (❶–❸). Specifically, upon identifying missing critical KV entries via Attention-Based Cache Management, the scheduling module initiates a fetch operation from the Coordinated Multi-Tier KV Storage to load required data from SSD to HBM.**

K-means–based ANNS approaches [20], this hierarchical structure preserves local semantic continuity among contiguous tokens. According to § 9, it achieves higher retrieval accuracy than similarity-grouping methods, while matching the precision of spatial chunking under only half the index size and delivering up to 2× faster lookup speed, effectively reducing GPU memory pressure and facilitating downstream scheduling and tiering optimizations.

KVDrive is structured around three core components:

**(1) Attention-Based Cache Management:** KVDrive goes beyond naive caching through two complementary mechanisms: (i) a *lookahead eviction policy* that leverages current attention signals to anticipate near-future reuse, ensuring that entries most likely to be needed remain resident; and (ii) a *2D layer–head scaling strategy* that allocates per-layer and per-head window sizes according to measured attention locality. Together, these mechanisms maximize reuse under tight GPU memory budgets while maintaining a fixed overall footprint (§ 5).

**(2) Elastic Pipeline Scheduling:** Redesigns the decoding pipeline through two complementary techniques: (i) *SFC disaggregation*, which decouples selection, fetching, and computation into independently scheduled stages, enabling fine-grained overlap between operations; and (ii) *pipeline optimization*, which optimizes index size, cache size, and micro-batch size to balance accuracy and throughput. Together, these mechanisms eliminate pipeline stalls and maintain high utilization across GPU, CPU, and I/O subsystems under varying batch and context configurations (§ 6).

**(3) Coordinated Multi-Tier KV Storage:** Extends beyond DRAM-only offloading by incorporating SSD as a third tier and coordinating data movement across HBM, DRAM, and SSD. KVDrive (i) applies *importance-guided warm-up* to prioritize high-value KV entries during prefill, (ii) employs an *SSD-aware layout* to maximize sequential I/O locality, and (iii) performs *parallel sparse synchronization* to minimize tier transfer overhead. Together, these mechanisms enable scalable long-context inference well beyond the memory capacity of GPU and DRAM alone (§ 7).

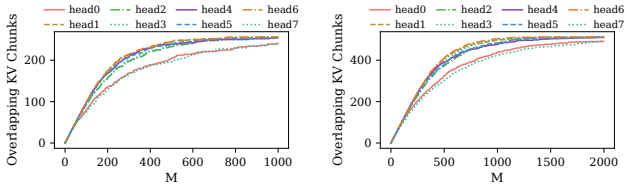



**Figure 7: Number of Top-M critical KV entries at one decoding step that also belong to the Top-K set at the next step.**

## 5 Attention-Based Cache Management

This section presents an in-GPU cache management scheme that departs from traditional usage-based policies (e.g., LRU, LFU) by leveraging the model's attention mechanism to infer the semantic importance of cached entries. By aligning cache residency with attention-derived importance rather than mere access recency or frequency, KVDrive captures temporal locality across nearby tokens and supports incremental updates instead of redundant reloads.

### 5.1 Sliding Window w/ Lookahead Eviction

KVDrive maintains a sliding window of critical KV entries covering several recent tokens in GPU memory. At initialization, offline profiling establishes a mapping between the window size and memory footprint for a given model. The system then selects the largest feasible window size that fits within the available GPU cache budget, where the budget is determined after accounting for the memory occupied by model parameters, activations, and intermediate buffers. This ensures that the cache fully utilizes the remaining GPU capacity without interfering with model computation.

During decoding, the window advances by one token at a time: new critical KV entries are fetched from host memory, while a subset of existing ones are evicted. Figure 7 shows that entries receiving high attention at one step are much more likely to remain critical in the next. As $M$ increases, the overlap with the subsequent step's top-$K$ set rises rapidly at first and then saturates, indicating that highly ranked entries are consistently reused while the marginal benefit of including lower-ranked ones quickly diminishes.

Motivated by this observation, instead of traditional LRU or LFU policies, KVDrive adopts a *lookahead eviction policy*: at each step, entries with the lowest current-step attention scores are discarded, as they are least likely to be reused in subsequent steps.

By maintaining a compact and incrementally updated working set of KV entries, KVDrive amortizes host–GPU transfers across decoding steps, improving bandwidth efficiency and reducing redundant data movement.

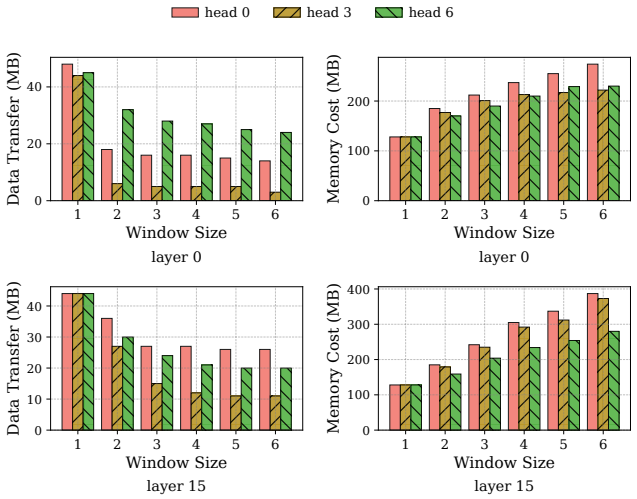



**Figure 8: Data transfer and memory overhead for different window sizes across layers and heads.**

## 5.2 2D Window Scaling

Our profiling in Figure 8 shows that different layers and attention heads exhibit heterogeneous trade-offs between memory overhead and transfer reduction when enlarging the window size. For most layer–head pairs, larger windows provide only modest savings, whereas certain layers and specific heads achieve disproportionately higher reductions in transfer volume for the same memory cost. This reflects the diverse roles of transformer layers and heads: some primarily capture localized dependencies, while others specialize in modeling long-range structure. We should allocate less space for the former one (e.g., Head 1 and 2 in Layer 31 in Figure 9) and more for the latter one (e.g., Head 8 in Layer 31 in Figure 9).

To systematically exploit this heterogeneity, we formulate *2D window scaling* as an offline optimization problem. For each layer $l$ and head $h$, profiling yields: $\text{Benefit}_{l,h}(w)$: transfer reduction achieved with window size $w$, $\text{Cost}_{l,h}(w)$: additional GPU memory consumed by that window size.

Given a total GPU cache budget $M$, the objective is:

$$\max_{\{w_{l,h}\}} \sum_{l,h} \text{Benefit}_{l,h}(w_{l,h}) \quad \text{s.t.} \quad \sum_{l,h} \text{Cost}_{l,h}(w_{l,h}) \le M.$$

This allocation problem is a variant of the *multiple-choice knapsack problem* (MCKP), which is NP-hard in general. Fortunately, the problem size in our setting is moderate (hundreds of layer–head pairs and a small set of candidate window sizes). In practice, KVDrive solves it offline: for small models, exhaustive search is feasible; for larger ones, we employ a greedy algorithm that starts from the smallest windows and iteratively enlarges the window with the highest benefit-to-cost ratio until the GPU cache budget is met. This approach produces near-optimal allocations within minutes and incurs no runtime overhead.

By tailoring window sizes across both layers and heads, KVDrive achieves finer-grained cache allocation, striking a better balance between memory efficiency and communication cost, thereby enabling high-quality inference under strict resource constraints.

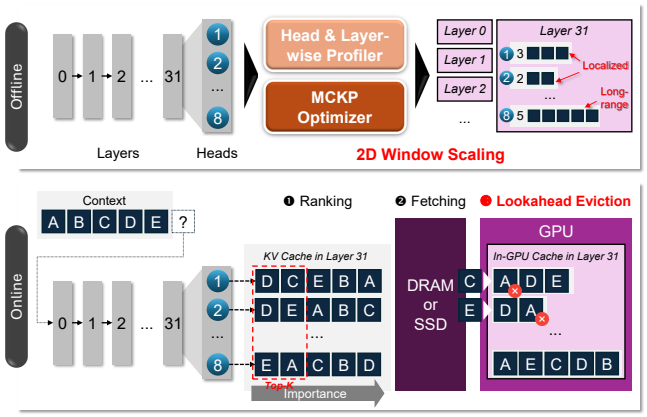



**Figure 9: The offline initialization and online running of the adaptive in-GPU cache management in KVDrive.**

## 6 Elastic Pipeline Scheduling

Most existing systems [7, 28, 29] adopt a sequential workflow (Figure 10(a)), where the computation of each layer during decoding proceeds in three stages: selecting critical KV entries, fetching them from host memory, and executing the layer's operations on the GPU. This design introduces substantial *stalls*, i.e., idle GPU cycles when computation must wait for selection or data transfer. To mitigate stalls, InfiniGen [18] adopts a pipelined design in which each layer prefetches critical KV entries using attention input from the previous layer (Figure 10(b)). This reduces fetching stalls by overlapping data transfer with computation, but selection stalls remain unresolved, and the approximation used for prefetching can lead to suboptimal KV selection, potentially degrading accuracy and stability in long-context scenarios. To eliminate both types of stalls without sacrificing accuracy, we propose an elastic pipeline scheduling strategy, as detailed below.

### 6.1 SFC Disaggregation

The three tightly coupled stages—selection, fetching, and computation—exhibit distinct performance bottlenecks. The selection stage executes on the GPU and is primarily I/O-bound, as it requires reading and scoring large index regions. The fetching stage is dominated by host–device data transfers, while computation (mainly feed-forward operations) is compute-bound on the GPU. Moreover, in-GPU cache hit/miss evaluation and metadata updates described in § 5 require CPU participation, introducing additional synchronization overhead. Executing these stages sequentially and applying an identical batching strategy across them leads to suboptimal utilization of GPU and CPU compute units as well as I/O bandwidth.

KVDrive addresses these inefficiencies through *SFC disaggregation*, which decouples the three stages for independent scheduling. Each batch is partitioned into multiple micro-batches to enable fine-grained parallelism between selection and fetching, while computation is executed for the entire batch as a single unit. During decoding, the GPU performs selection for the current micro-batch

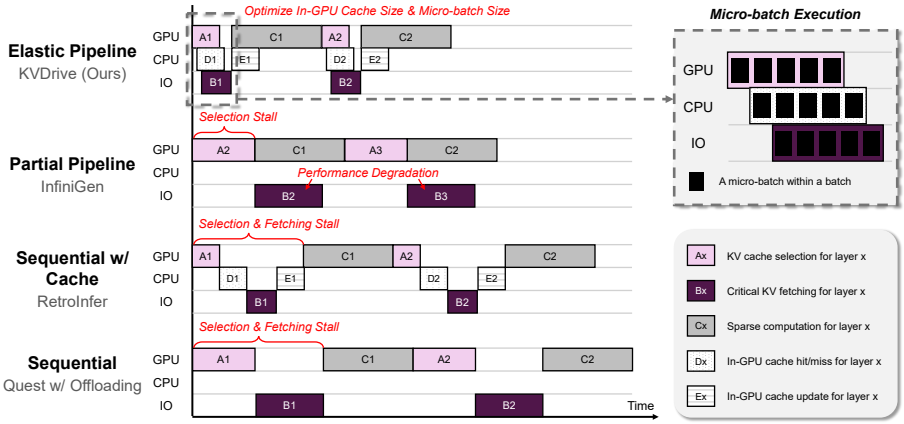



**Figure 10: Comparison between different KV cache offloading scheduling strategies.**

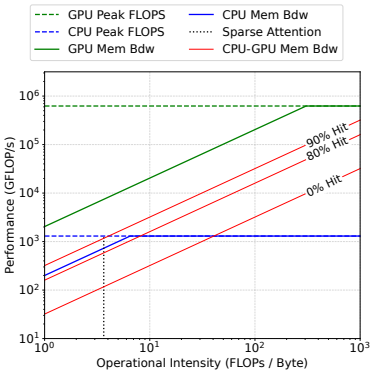



**Figure 11: A GPU-CPU roofline model of Llama 3 8B in a KV cache offloading system on an A100 instance.**

concurrently with the CPU evaluating cache hit/miss status for the previous one and fetching the KV entries of an earlier micro-batch from host memory. Computation proceeds once all micro-batches have completed fetching. Meanwhile, cache metadata updates are overlapped with computation, allowing the system to maintain cache consistency without interfering with the main decoding pipeline. This disaggregated design sustains high throughput by maximizing utilization across GPU, CPU, and I/O subsystems.

Unlike conventional pipeline overlap, SFC disaggregation explicitly separates the three stages into independently scheduled units coordinated through lightweight queues and asynchronous transfers. This design maintains balanced utilization across heterogeneous resources and delivers high throughput.

## 6.2 Pipeline Optimization

The performance of KVDrive's pipeline depends critically on three parameters: the number of centroids in the index, the GPU cache size, and the micro-batch size. These parameters jointly determine the balance between selection accuracy, CPU–GPU coordination cost, and overall pipeline efficiency.

**(1) Index.** A larger number of centroids improves selection granularity but increases selection cost, as each query must compare against more index representatives. Conversely, too few centroids reduce accuracy in identifying critical KV entries. Our experiments show that KVDrive achieves comparable accuracy to spatial-chunking methods [28, 29] using only half as many centroids, significantly reducing selection latency.

**(2) Cache size.** A larger in-GPU cache shortens fetching time by reducing host–device transfers, but increases CPU-side hit/miss evaluation time, which can become the new bottleneck. Before decoding begins, KVDrive performs a warm-up phase that incrementally increases the cache size until the CPU evaluation time and GPU fetching time reach equilibrium, maximizing end-to-end throughput under given memory constraints.

**(3) Micro-batch size.** The micro-batch size determines pipeline granularity. Excessively large micro-batches create long bubbles between overlapping stages, whereas very small ones lead to frequent kernel launches and synchronization overhead. Since the candidate range is limited, KVDrive performs a short pre-run calibration to empirically identify the optimal configuration before inference begins. Through this lightweight tuning process, the system adapts its scheduling and resource allocation to the underlying hardware characteristics, ensuring balanced utilization across compute and I/O components.

## 6.3 Performance Analysis via Roofline Model

Prior works [4, 7] advocate performing attention on the CPU in KV cache offloading systems, arguing that CPU–GPU transfer overhead dominates GPU execution. We instead analyze why GPU-based attention is preferable in our system, using the roofline model [39].

Figure 11 shows the roofline model for attention computation in Llama 3 8B when all KV caches reside in CPU memory. When the operational intensity falls below the threshold $P$, transferring data to the GPU yields no benefit, as performance is bounded by the CPU–GPU bandwidth roof. This is the regime assumed in prior studies. In contrast, our adaptive cache management in § 5 ensures that at each decoding step, only out-of-cache critical KV entries—those not already in the in-GPU cache—must be fetched from host memory. Empirically, over 90% of critical entries hit in the in-GPU cache (see § 9), leaving only a small fraction to be transferred. Consequently, the operational intensity remains above the threshold $P$, enabling the GPU to sustain higher effective throughput than the CPU and making it the preferable choice for attention computation.

# 7 Coordinated Multi-Tier KV Storage

While GPU and DRAM offer high-bandwidth access for KV caches, their combined capacity quickly becomes insufficient under long contexts or large batches. To extend capacity, prior work explores offloading KV caches to SSDs [26]; yet directly treating SSD as a slower extension of DRAM causes frequent high-latency transfers and stalls during decoding. To address this limitation, KVDrive designs a coordinated multi-tier KV storage system that enables efficient collaboration across HBM, DRAM, and SSD. It integrates three complementary techniques: importance-guided warm-up for tier placement initialization, an SSD-aware layout for sequential

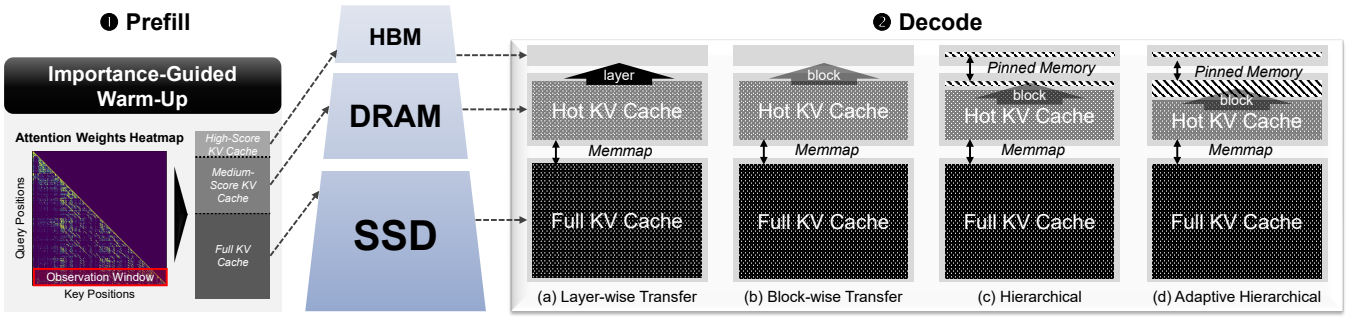



**Figure 12: The workflow of the coordinated multi-tier KV storage in KVDrive.**

I/O locality, and a parallel sparse synchronization pipeline for low-latency data movement during decoding.

## 7.1 Importance-Guided Warm-Up

At the end of the prefill phase, the model has already computed the attention between the prompt's final tokens and the entire prefix. KVDrive exploits this information to estimate the long-term importance of prefix KV entries before decoding begins. Inspired by SnapKV [19], we assign importance scores to prefix tokens based on the attention distribution produced by the queries in the prompt's final observation window.

The observation window refers to the last few tokens of the prompt (typically 16–64 tokens). For each query within this window, we compute its attention weights over all prefix keys and aggregate them across heads and layers to obtain an importance profile of prefix positions. This aggregated profile captures which KV entries are most likely to be reused during decoding and directly guides their initial placement across memory tiers.

According to this importance profile, all KV entries are first persisted to SSD as the full backing store, while higher-ranked entries are promoted to faster tiers: the entries with the highest scores are placed in GPU HBM for immediate access; the subsequent highest-scoring entries—those that do not fit in HBM—are offloaded to DRAM. This one-time, attention-informed warm-up establishes a balanced starting point for decoding, significantly reducing subsequent data migration across tiers. Unlike SnapKV, which leverages observation windows for KV compression, our approach repurposes the same insight for tiered placement, enabling coordinated HBM–DRAM–SSD management without compromising model accuracy.

## 7.2 SSD-Aware Layout Planning

When offloading KV entries to SSD, the data layout becomes a critical determinant of I/O efficiency. Random accesses on SSDs are costly, whereas sequential operations can achieve an order of magnitude higher throughput. To align storage organization with the temporal and structural access patterns observed during decoding, KVDrive employs a two-level packing strategy.

We first define an *extent* as a contiguous SSD block that groups multiple KV entries together. By packing entries into extents, the system can transform fine-grained, irregular accesses into coarse-grained sequential transfers, improving bandwidth utilization and reducing access latency.

**(1) Semantic-Contiguity Packing.** KV entries that are frequently attended together during decoding—such as tokens within the same semantic chunk or attention cluster—are placed sequentially within the same extent. This organization enables multiple related entries to be retrieved through a single large I/O, minimizing random-access overhead across non-contiguous regions.

**(2) Layer–Head Partitioning.** To further align with Transformer structure, extents are partitioned by layer and attention head. Each {layer, head} pair is assigned to a dedicated SSD segment, and its extents are stored contiguously within that segment to preserve structural locality.

By combining semantic-contiguity packing with layer–head partitioning, KVDrive reshapes inherently irregular KV access patterns into predictable, high-throughput sequential I/O. This SSD-aware layout design substantially reduces access latency and improves end-to-end decoding efficiency, making SSD-based tiering practical even for long-context and large-batch workloads.

## 7.3 Parallel Sparse Synchronization

Efficient data transfer across SSD, DRAM, and GPU HBM is essential for sustaining decoding throughput under multi-tier offloading. While SSD offers large capacity, its bandwidth is significantly lower than that of DRAM, and frequent migrations can easily become the new bottleneck. Figure 12 (right) compares four synchronization strategies that progressively improve data movement efficiency.

**(a) Naïve Layer-wise Transfer.** This follows FlexGen [27]: at each decoding step, the KV cache of an entire layer is transferred from SSD to HBM, using DRAM (via `memmap`) as a passive buffer. Although simple, this design provides almost no reuse: when the DRAM cache is smaller than the total KV footprint, subsequent layer transfers overwrite previously cached data, causing redundant I/O and frequent pipeline stalls as discussed in § 3.

**(b) Block-level Sparse Fetching.** Instead of migrating entire layers, KVDrive transfers only the blocks (clusters) of KV entries that are actually required by the current attention queries. This fine-grained selection reduces redundant transfers and better matches the sparsity patterns of attention. To avoid excessive random I/O, multiple block requests are coalesced into extent-sized sequential

reads with sufficient queue depth, ensuring high SSD throughput and efficient utilization of the I/O pipeline.

**(c) Hierarchical Transfer.** To further reduce SSD latency and overlap data preparation with computation, KVDrive adopts a hierarchical staging pipeline. KV entries fetched from SSD are first read into the DRAM-backed `memmap` region and then staged into preallocated page-pinned buffers before GPU execution. This design serves two purposes: (i) it avoids repeatedly allocating pinned memory by reusing a stable pool of pinned buffers, and (ii) it enables asynchronous prefetching—KV entries can be read ahead during the selection and pre-attention (QKV projection).

**(d) Balanced Coordination.** Finally, KVDrive balances the benefits of pinned buffers and `memmap` caching. Pinned memory provides high-bandwidth access, while the `memmap` region serves as a large passive cache in DRAM. The allocation between these two tiers is guided by offline profiling, prioritizing pinned memory for layer-head KV entries exhibiting frequent stalls—patterns empirically correlated with page-fault-heavy areas in the `memmap` tier—and keeping such regions permanently resident in pinned memory to ensure stable high-bandwidth access throughout decoding.

Overall, this parallel sparse synchronization enables efficient multi-tier coordination across HBM, DRAM, and SSD, sustaining high GPU utilization under long-context and large-batch settings.

# 8 Implementation

We have implemented a fully functional prototype of KVDrive comprising about 9,000 lines of Python, 1,000 lines of C++, and 3,000 lines of CUDA code. The system is built on PyTorch 2.3.0 and Python 3.12, running on Ubuntu 22.04 with CUDA 12.1. To support large KV cache storage, KVDrive employs `numpy.memmap` for memory-mapped arrays, enabling persistent and page-level access to KV tensors directly on disk. Sparse updates and retrievals are performed through `torch.Tensor.index_copy_()`. For clustering, we adopt the Triton kernels from RetroInfer [6], while the data movement primitives for gather and copy are derived from ShadowKV [28]. We also leverage FlashInfer [36] for high-performance GPU kernels (such as attention and normalization) optimized for LLM inference. Crucially, our implementation fully supports parallel sessions via continuous batching, a technique widely adopted in modern serving systems like Orca [37] and vLLM [16]. To ensure fair comparisons, all baseline systems mentioned in § 9 have been reimplemented and integrated into our unified evaluation framework to ensure consistent configurations and fair comparisons. In addition, all long-context models employ the official RoPE implementations provided by the huggingface's `transformers` framework (e.g., YaRN for Qwen models and LongRoPE for Phi models) to ensure compatibility and consistency across baselines.

# 9 Experiment

## 9.1 Experimental Setup

***Models.*** For evaluation, we select four widely used open-source LLMs with strong long-context capabilities: Llama-3-8B-1048k [2] (8B parameters, 1M-token context window), Qwen3-8B and Qwen3-14B [33] (128K-token context window), and Microsoft Phi-4-mini-instruct [1] (3.8B parameters, 128K-token context window). These models collectively cover a broad spectrum of parameter scales and context lengths, providing a representative basis for evaluating the effectiveness of KVDrive and the baselines.

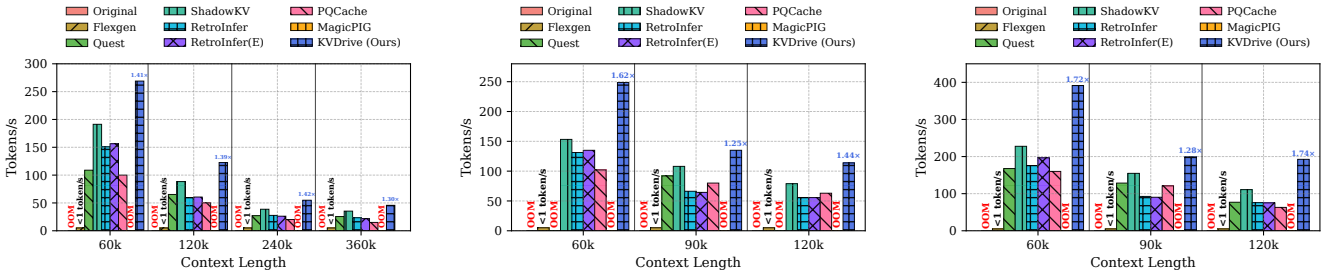
***Benchmarks.*** We evaluate KVDrive on two widely-used long-context benchmarks: LongBench [3], a bilingual dataset for long-context understanding, and RULER [13], which covers retrieval, multi-hop reasoning, aggregation, and QA tasks.

***Baselines.*** We compare KVDrive with eight KV cache offloading systems: Quest [29], ShadowKV [28], RetroInfer [6], PQCache [40], MagicPIG [7] and FlexGen [27], which adopts a full-cache strategy by repeatedly loading and evicting KV entries between GPU and host memory during attention computation. Additionally, we also include two reference configurations: RetroInfer(E), a variant of RetroInfer with attention estimation enabled, and Original, a baseline that maintains the entire KV cache in GPU memory without offloading. For all systems except FlexGen, we use exact prefilling followed by dynamic sparse attention in the decoding phase. For ShadowKV, we follow a setting from its original paper, i.e., the chunk size is 8 and the number of outliers is 48. For RetroInfer, for fair comparison, we disable its attention estimation mechanism. Following common practice in existing systems [7, 31], all systems retain sink tokens (the first 4 tokens) and 64 local tokens in the GPU memory for every sparse attention, as these tokens consistently carry high importance.

***Hardware Setting.*** We conduct experiments on three representative hardware environments covering different deployment tiers: 1) **Cost-efficient server:** equipped with an NVIDIA L20 (48 GB) GPU, Intel(R) Xeon(R) Platinum 8457C CPU, and 100 GB DDR5 host memory. This configuration represents a practical inference-oriented data-center node with moderate GPU memory and high PCIe bandwidth. 2) **High-end server:** featuring an NVIDIA H20 (96 GB) GPU, AMD EPYC 9K84 96-Core Processor, and 200 GB DDR5 host memory. It reflects a compute-rich environment typical of enterprise or cloud-scale deployments with larger GPU memory and stronger host-side compute. 3) **Workstation:** powered by an NVIDIA RTX 4090 (24 GB) GPU, Intel(R) Xeon(R) Gold 6430 CPU, and 120 GB DDR5 host memory. This setup models a resource-constrained local inference scenario representative of edge or on-premise deployments. The disks in the system are NVMe U.2 SSDs.

## 9.2 Overall Improvement

Figure 13 shows the generation throughput (tokens/s) of different systems under varying batch sizes and context lengths on Llama-3-8B-1048K, Qwen3-8B and Phi-4-Mini-128K in the L20 server. KVDrive consistently achieves the highest throughput across all settings. In particular, compared to ShadowKV—the strongest baseline—KVDrive delivers up to 70% higher throughput, thanks to its elastic pipeline scheduling and cache management. However, not all systems demonstrate the capability to handle the challenges posed by our experimental setup. For instance, RetroInfer fails to initialize due to its dependence on a massive pre-computed index structure. The memory requirements for storing this index in long-context models exceed the available GPU memory, leading to immediate out-of-memory (OOM) errors. Similarly, Quest also encounters OOM failures under heavy workloads. Its architecture, which combines an indexing mechanism with a large KV cache

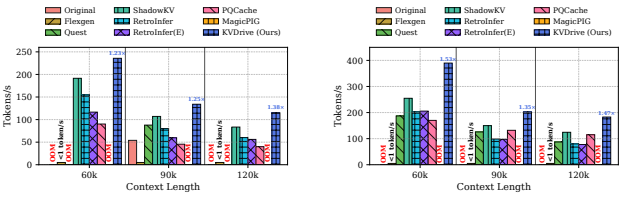



(a) Llama-3-8B-1048K (b) Qwen-3-8B (c) Phi-4-Mini-128K

**Figure 13: Generation throughput (tokens/s) under varying context lengths and batch sizes in the L20 server. The experiments use batch sizes of 8, 4, 4, 2 and 2 for context lengths of 60k, 90k, 120k, 240k and 360k, respectively.**

**Table 2: Performance comparison of different models and methods on RULER and LongBench.**

| | | RULER | | | | | | | | | | | | | LongBench | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Model | Method | S1 | S2 | S3 | MK1 | MK2 | MK3 | MQ | MV | QA-1 | QA-2 | VT | FWE | CWE | Avg. | NQA | MQA | HQA | MQue | DRead | Grep | SAM | PRer | LCC | Avg. |
| LLama-3-8B-1048K | Full | 100 | 100 | 100 | 98.95 | 97.91 | 43.75 | 98.95 | 96.35 | 75.00 | 48.95 | 78.12 | 72.56 | 0.20 | 77.74 | 18.98 | 41.84 | 36.79 | 21.47 | 31.93 | 34.18 | 35.96 | 81.5 | 56.07 | 39.85 |
| | Quest | 100 | 100 | 100 | 98.86 | 85.42 | 21.90 | 97.92 | 95.49 | 70.83 | 46.88 | 78.75 | 65.63 | 0.31 | 73.99 | 17.26 | 39.51 | 36.78 | 18.71 | 26.41 | 29.49 | 35.8 | 79.5 | 60.05 | 38.17 |
| | ShadowKV | 100 | 100 | 100 | 97.91 | 98.95 | 32.29 | 96.09 | 95.83 | 71.87 | 52.08 | 82.7 | 72.56 | 0.10 | 76.95 | 17.17 | 39.73 | 38.29 | 21.08 | 31.77 | 31.62 | 35.87 | 80 | 63.93 | 39.94 |
| | RetroInfer | 100 | 100 | 100 | 98.95 | 93.75 | 37.50 | 97.39 | 91.66 | 73.95 | 50.00 | 78.54 | 63.88 | 0.10 | 75.82 | 19.17 | 41.35 | 37.59 | 19.71 | 27.36 | 33.76 | 43.03 | 80.50 | 50.77 | 39.24 |
| | RetroInfer(E) | 100 | 100 | 100 | 98.95 | 96.87 | 41.66 | 97.91 | 90.62 | 73.95 | 48.95 | 76.25 | 73.61 | 0.10 | 76.83 | 18.56 | 42.55 | 36.89 | 20.29 | 27.53 | 33.25 | 42.99 | 80.00 | 45.38 | 38.60 |
| | PQCache | 85.40 | 94.80 | 94.80 | 92.70 | 85.41 | 34.37 | 95.83 | 95.05 | 59.37 | 43.80 | 58.12 | 63.54 | 0.22 | 69.49 | 19.13 | 40.99 | 36.30 | 20.65 | 26.21 | 31.77 | 41.56 | 80.50 | 44.92 | 38.00 |
| | MagicPIG | 100 | 95.83 | 94.79 | 86.46 | 87.50 | 23.96 | 78.65 | 73.96 | 61.46 | 44.79 | 61.46 | 69.10 | 0.10 | 67.54 | 17.83 | 40.95 | 37.09 | 21.13 | 23.81 | 28.98 | 41.76 | 80.00 | 43.55 | 37.23 |
| | KVDrive | 100 | 100 | 100 | 98.95 | 94.79 | 37.50 | 98.43 | 94.27 | 71.90 | 51.00 | 75.41 | 67.70 | 0.10 | 76.15 | 19.17 | 41.65 | 36.34 | 21.27 | 26.00 | 32.64 | 43.10 | 81.00 | 49.84 | 39.00 |
| Qwen-3-8B | Full | 100 | 100 | 100 | 79.16 | 62.50 | 13.54 | 95.57 | 96.09 | 59.37 | 32.29 | 95.83 | 83.33 | 43.54 | 73.94 | 3.53 | 24.60 | 11.39 | 6.39 | 19.97 | 25.82 | 44.45 | 85.08 | 65.86 | 31.89 |
| | Quest | 100 | 98.95 | 98.95 | 83.33 | 44.79 | 1.04 | 91.40 | 97.91 | 41.66 | 35.40 | 97.29 | 75.00 | 22.91 | 68.35 | 2.93 | 25.36 | 10.96 | 5.93 | 20.74 | 27.83 | 45.88 | 85.79 | 69.67 | 32.79 |
| | ShadowKV | 100 | 98.95 | 100 | 83.33 | 38.54 | 2.08 | 89.32 | 93.48 | 45.83 | 31.25 | 92.08 | 75.69 | 20.93 | 67.03 | 2.93 | 23.51 | 10.09 | 6.30 | 19.49 | 27.28 | 45.37 | 86.28 | 69.03 | 32.39 |
| | RetroInfer | 100 | 96.90 | 98.95 | 73.95 | 52.08 | 2.08 | 91.40 | 85.40 | 34.37 | 21.87 | 97.70 | 74.65 | 36.97 | 66.64 | 2.93 | 26.20 | 11.08 | 5.82 | 19.86 | 27.63 | 46.87 | 87.33 | 68.42 | 32.82 |
| | RetroInfer(E) | 100 | 94.79 | 98.95 | 75.00 | 51.04 | 3.12 | 90.62 | 85.67 | 66.66 | 46.87 | 87.91 | 77.08 | 16.35 | 68.77 | 3.57 | 25.43 | 11.11 | 6.27 | 19.85 | 27.49 | 45.81 | 86.29 | 67.98 | 32.64 |
| | PQCache | 65.62 | 79.16 | 84.37 | 77.08 | 30.20 | 1.04 | 88.28 | 93.22 | 38.54 | 31.25 | 82.70 | 72.56 | 20.62 | 58.81 | 3.51 | 25.56 | 11.42 | 5.85 | 22.57 | 27.21 | 46.06 | 76.45 | 70.04 | 32.08 |
| | MagicPIG | 100 | 83.33 | 79.16 | 68.75 | 34.37 | 0 | 66.14 | 80.20 | 27.08 | 21.87 | 29.58 | 75.00 | 4.37 | 51.52 | 2.88 | 18.63 | 10.23 | 5.30 | 17.81 | 18.75 | 39.06 | 96.73 | 56.97 | 29.59 |
| | KVDrive | 100 | 100 | 100 | 83.33 | 44.79 | 2.08 | 91.92 | 96.35 | 41.66 | 30.20 | 97.29 | 71.52 | 25.83 | 68.07 | 3.33 | 24.03 | 10.69 | 6.06 | 19.49 | 27.28 | 45.37 | 86.28 | 69.03 | 32.39 |
| Phi-4-Mini-128K | Full | 100 | 97.91 | 98.95 | 83.33 | 64.58 | 4.16 | 81.51 | 76.04 | 51.04 | 38.54 | 73.95 | 68.05 | 3.02 | 64.69 | 26.12 | 55.48 | 53.11 | 28.94 | 29.78 | 34.09 | 44.36 | 91.00 | 54.24 | 46.34 |
| | Quest | 100 | 93.75 | 79.16 | 75.00 | 55.20 | 0 | 76.30 | 72.65 | 45.83 | 38.54 | 81.66 | 54.16 | 2.08 | 59.57 | 25.69 | 52.88 | 52.83 | 29.24 | 29.31 | 33.94 | 45.13 | 91.50 | 56.19 | 46.30 |
| | ShadowKV | 100 | 95.83 | 68.75 | 79.16 | 53.12 | 1.04 | 69.53 | 62.50 | 46.87 | 37.50 | 79.16 | 61.80 | 1.56 | 58.21 | 25.69 | 52.63 | 52.89 | 29.49 | 29.92 | 33.06 | 45.41 | 92.00 | 56.99 | 46.45 |
| | RetroInfer | 100 | 94.79 | 93.75 | 78.12 | 54.16 | 0 | 72.39 | 54.42 | 50.00 | 36.45 | 81.45 | 54.51 | 1.66 | 59.36 | 25.69 | 53.18 | 52.93 | 28.54 | 31.45 | 33.43 | 45.86 | 91.50 | 56.13 | 46.52 |
| | RetroInfer(E) | 97.91 | 73.95 | 94.79 | 54.16 | 37.50 | 0 | 47.39 | 36.71 | 30.20 | 31.25 | 65.62 | 56.94 | 1.45 | 48.29 | 26.33 | 48.09 | 46.49 | 24.36 | 29.02 | 31.36 | 42.06 | 91.50 | 43.24 | 42.49 |
| | PQCache | 35.41 | 3.12 | 2.08 | 6.25 | 6.25 | 0 | 4.16 | 6.25 | 19.79 | 26.04 | 38.12 | 57.98 | 0.41 | 15.83 | 26.14 | 47.60 | 45.23 | 21.13 | 28.98 | 29.34 | 41.36 | 91.00 | 40.55 | 41.25 |
| | MagicPIG | 50 | 14.58 | 0 | 9.37 | 16.66 | 0 | 2.86 | 1.56 | 20.83 | 29.16 | 19.79 | 47.56 | 0.52 | 16.09 | 22.86 | 37.96 | 43.11 | 20.41 | 17.55 | 15.19 | 35.28 | 88.02 | 32.59 | 34.77 |
| | KVDrive | 100 | 94.79 | 90.62 | 79.16 | 52.10 | 1.04 | 71.61 | 60.67 | 48.95 | 36.45 | 82.70 | 59.72 | 1.14 | 59.91 | 25.69 | 52.24 | 53.16 | 28.95 | 29.69 | 32.84 | 45.05 | 91.50 | 53.79 | 45.87 |




(a) Qwen-3-14B in H20 (b) Phi-4-Mini-128K in RTX4090

**Figure 14: Generation throughput (tokens/s) under varying batch sizes and context lengths.**

footprint, imposes substantial memory pressure, saturating device memory and rendering it incapable of completing our test scenarios.

Among the systems that successfully initialized, FlexGen exhibits extremely poor performance in practical scenarios. While it avoids OOM errors by offloading data to host memory and storage, its design requires the entire KV cache to be loaded from off-chip storage at every generation step. This I/O-bound approach results in severe latency issues, with throughput dropping to less than 1 token/s. Such performance renders FlexGen impractical for real-world applications, particularly latency-sensitive tasks.

In contrast, KVDrive showcase robust and consistent performance across all evaluated hardware configurations. On the H20 and 4090 servers, it achieves throughput improvements ranging from 1.23× to 1.53× over the strongest baseline. These results underscore the efficiency and scalability of KVDrive, particularly in scenarios requiring high throughput for both long-context and large-batch inference. Its ability to maintain superior performance under diverse hardware and workload conditions further validates the versatility and robustness of its design.

## 9.3 Micro Benchmark

***Accuracy***. We first examine whether the system-level optimizations in KVDrive compromise model accuracy. Table 2 reports the results on both RULER and LongBench. Across all tasks, KVDrive

**Table 3: Effectiveness of the Lookahead (LA) eviction policy across different systems.**

| Method | Llama3-8B | | | Qwen-3-8B | | | Phi-4-mini-128K | | |
|---|---|---|---|---|---|---|---|---|---|
| | LRU(%) | LA(%) | Imp.(%) | LRU(%) | LA(%) | Imp.(%) | LRU(%) | LA(%) | Imp.(%) |
| Quest | 89.2 | 90.1 | +0.9 | 89.7 | 91.0 | +1.3 | 90.7 | 88.9 | -1.8 |
| Shadowkv | 84.2 | 87.3 | +2.9 | 81.0 | 83.2 | +2.2 | 86.1 | 84.6 | -1.5 |
| Retro | 80.7 | 82.0 | +1.3 | 70.0 | 73.9 | +3.9 | 78.3 | 80.1 | +1.9 |
| Kvdrive | 80.0 | 81.5 | +1.5 | 77.5 | 79.0 | +1.5 | 78.0 | 81.0 | +3.0 |

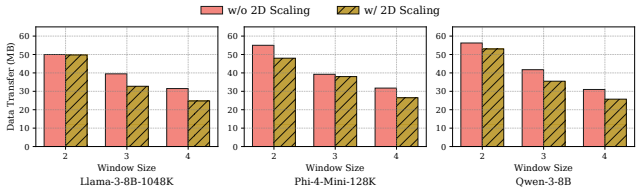



**Figure 15: Impact of 2D window scaling on data transfer volume across different window sizes and models.**

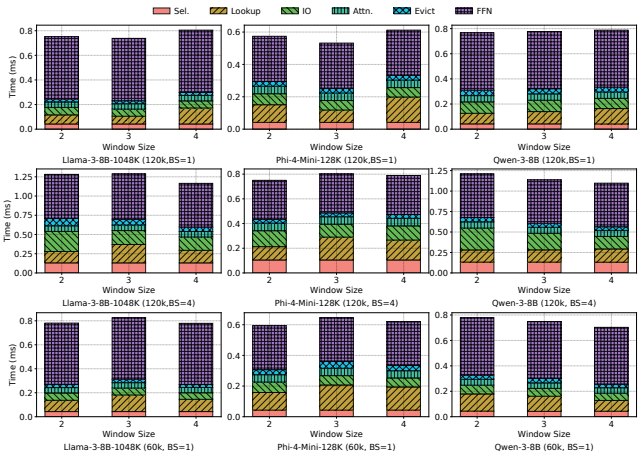



**Figure 16: Time breakdown of KVDrive under different window sizes and models under a sparsity budget of 1.56%.**

achieves accuracy that is on par with or even slightly better than state-of-the-art offloading systems such as Quest, ShadowKV, and InfiniGen. This demonstrates that our indexing and retrieval design in § 4 introduces negligible accuracy loss, even under long-context (up to 128K tokens). The results confirm that the gains of KVDrive primarily stem from improved cache management and scheduling, rather than trading accuracy for efficiency.

***Generality of Eviction Policy.*** Table 3 evaluates the generality of our Lookahead (LA) eviction policy by applying it to four different systems: Quest, ShadowKV, Retro, and KVDrive. The experiments are conducted with a 120K context length, window size of 2, sparsity budget of 2048, and a look-ahead candidate pool $M = 2560$. As shown in the table, the LA strategy demonstrates broad applicability, outperforming the traditional LRU baseline in the majority of configurations. Specifically, on Llama3-8B and Qwen-3-8B, the LA policy consistently boosts hit rates across all evaluated methods, achieving gains ranging from 0.9% to 3.9%. This indicates that identifying eviction candidates based on attention scores (as detailed in §5.1) is a robust approach for various system architectures. While Quest achieves marginally higher hit rates with both policies, it incurs significant overhead: its index size is over 4× larger than that of KVDrive and 2× larger than ShadowKV. In contrast, KVDrive with the LA policy achieves a comparable hit rate while maintaining a significantly smaller memory footprint.

***2D Window Scaling.*** Figure 15 reports the ablation results of our 2D window scaling strategy for Llama-3-8B-1048k, Phi-4-Mini-128K, and Qwen-3-8B in § 5.2. Under a constrained GPU memory budget, 2D scaling results in reduced data transfer compared to uniform window allocation. This indicates that heterogeneous reuse patterns across layers and attention heads can effectively improve the GPU cache utilization and avoid the waste of GPU cache.

***Window Size.*** Figure 16 depicts the latency breakdown of a single Transformer layer under a 1.56% sparsity budget. We evaluate the performance by sweeping window sizes through {2, 3, 4} across context lengths of 60k and 120k, and batch sizes of 1 and 4. As the window size increases, the reduction in data transfer volume alleviates I/O overhead; however, this benefit is offset by increased lookup latency. This trade-off necessitates a careful balance between I/O efficiency and lookup overhead. Specifically, with a batch size of 1, smaller window sizes (e.g., 2) generally yield lower latency. This is attributed to the surge in lookup time as cache capacity expands, while I/O bandwidth remains underutilized. Conversely, at a batch size of 4, larger window sizes (e.g., 4) prove more effective, primarily due to the reduced demand on I/O bandwidth. These results underscore the critical role of window size selection in maximizing throughput. Furthermore, as the batch size scales from 1 to 4, the latency contribution from non-FFN modules—including selection, lookup, attention, and I/O—increases significantly. This shift indicates that non-FFN components emerge as the dominant bottleneck at larger batch sizes, highlighting the imperative to optimize these modules for scalability.

***Chunk size.*** Figure 17 dispicts the latency breakdown of a single Transformer layer under a 1.56% sparsity budget. We evaluate performance across context lengths of 60k and 120k, sweeping chunk sizes through {1, 4, 8}. Notably, we observe a distinct U-shaped trend in both overall latency and specific I/O latency as the chunk size increases. This behavior reflects a fundamental trade-off between semantic continuity and data redundancy. With small chunk sizes (e.g., 1), data is processed in fragmented units, necessitating frequent memory accesses that inflate I/O overhead. Conversely, excessively large chunk sizes (e.g., 8) introduce redundant information, which increases the data transfer volume and negates the benefits of reduced access frequency. An intermediate chunk size (e.g., 4) strikes an optimal balance, preserving sufficient semantic continuity while minimizing redundancy, thereby achieving the lowest I/O cost.

***Number of Centroids.*** Figure 18 decomposes the latency of executing a single Transformer layer. We evaluate the performance across context lengths of 60k and 120k, sweeping the number of centroids through {2,048, 4,096, 8,192} under a fixed 1.56% sparsity budget. We observe a consistent relationship between context size and the

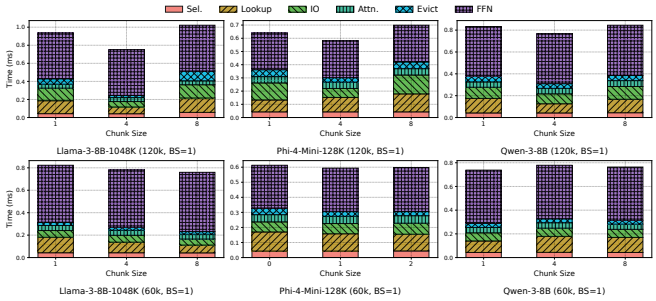



**Figure 17: Time breakdown of KVDrive under different chunk sizes under a sparsity budget of 1.56%.**

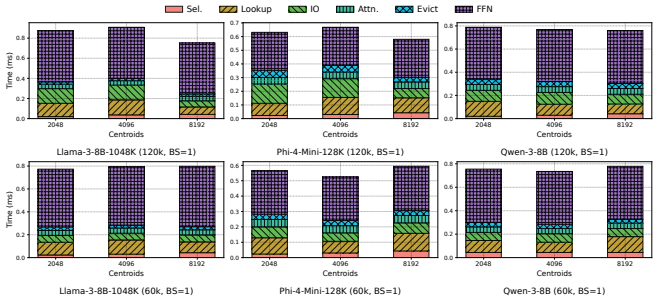



**Figure 18: Time breakdown of KVDrive under different centroids under a sparsity budget of 1.56%.**

optimal number of centroids: specifically, 8,192 centroids yield the best performance for the 120k context, while 4,096 centroids are optimal for 60k. This suggests that the optimal ratio of context length to centroids remains constant. While increasing the number of centroids incurs higher selection latency due to the added computational complexity of clustering, this overhead is effectively mitigated by significant reductions in I/O and lookup times. Higher centroids count facilitate finer-grained clustering, which distributes data more evenly across clusters. This improved load balance reduces synchronization overhead and enhances data locality, thereby lowering the latency of memory-bound components.

***Impact of Centroid Reduction.*** Figure 19 examines the accuracy of KVDrive under varying numbers of centroids for tasks such as QA-2, Variable Tracking, and Frequent Words Extraction. The results demonstrate that accuracy remains consistent regardless of the number of centroids, highlighting the robustness of the clustering approach. Notably, reducing the number of centroids from 8192 to 2048 achieves a 4× reduction in index size without any loss in precision. This observation underscores the effectiveness of the clustering mechanism in preserving the semantic structure required for accurate task performance, even with fewer centroids. The reduction in index size directly translates to lower memory and storage overhead, improving the overall efficiency of the method.

***Memory Layout.*** With a batch size of 8 and a context length of 120k, Figure 20 evaluates the GPU memory usage of various systems across three models: Llama-3-8B-1048K, Qwen-3-14B, and Phi-4-Mini-128K. Among the evaluated systems, KVDrive exhibits

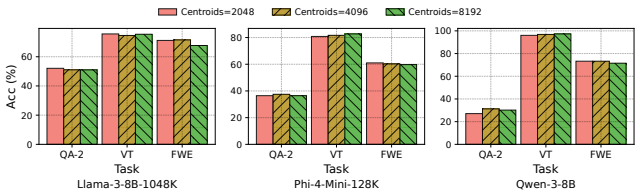



**Figure 19: Accuracy under varying centroids and tasks.**

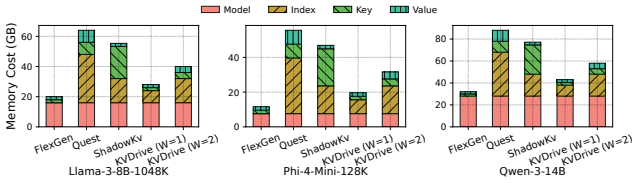
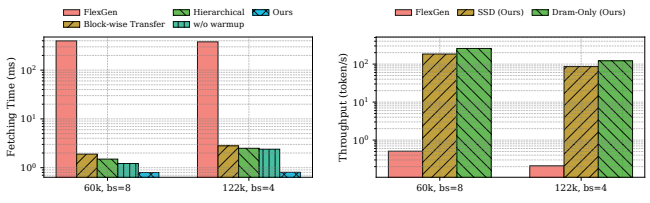



**Figure 20: Memory layout under varying models.**

significantly lower GPU memory usage compared to ShadowKV and Quest. This result primarily attributed to its sparse and efficient indexing mechanism. Conversely, ShadowKV incurs substantial memory overhead due to its reliance on compressed key storage, which is entirely resident in GPU memory and imposes significant cost. FlexGen achieves the lowest GPU memory utilization by avoiding in-memory storage of indexes. However, this design comes at the expense of performance: FlexGen must reload the entire KV cache for a layer at every generation step, introducing severe I/O bottlenecks and extreme latency. In contrast, KVDrive's memory footprint, while slightly higher than that of FlexGen, avoids these I/O penalties through its efficient caching and sparse indexing strategies. Moreover, as the window size increases (e.g., from 1 to 2), KVDrive maintains lower GPU memory usage compared to Quest and ShadowKV, while significantly alleviating I/O pressure.

***DRAM-Only VS DRAM + SSD.*** Figure 21 compares the performance of FlexGen and our approach under various configurations for the Llama-3-8B-1048K model, focusing on fetching time and throughput. As shown in Figure 21a, our Blockwise Transfer strategy delivers substantial performance improvements over FlexGen. By sparsely fetching only the required blocks on demand, our approach significantly reduces fetching overhead compared to FlexGen's full-layer loading mechanism. Moreover, the Hierarchical strategy with asynchronous fetching further minimizes latency by enabling more efficient lookups and data access. Additionally, the incorporation of a prefill warmup stage, which identifies critical KV cache entries during the final prefill phase, further optimizes fetching time by prioritizing high-value cache components. Figure 21b demonstrates that our approach achieves significantly higher throughput than FlexGen across all configurations. Notably, when using SSD as a component of the storage hierarchy, KVDrive sustains high throughput with only a 40% reduction compared to the DRAM-only configuration, while supporting larger batch sizes.

***Prefill Latency.*** Figure 22 evaluates the prefill latency of Llama-3-8B-1048K. KVDrive matches the performance of the full-attention baseline (Original) across all context lengths, indicating that its




(a) Llama-3-8B-1048K (b) Llama-3-8B-1048K

**Figure 21: Performance comparison in DRAM-Only and DRAM + SSD.**

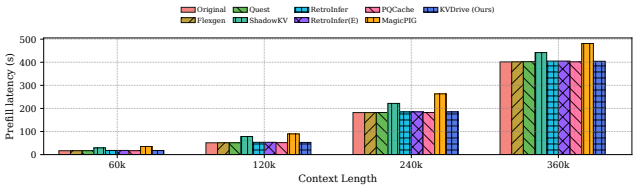
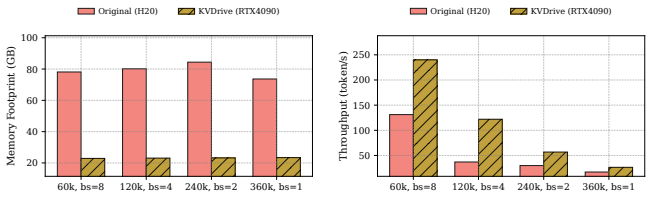



**Figure 22: Prefill latency (s) under different context lengths.**




(a) Llama-3-8B-1048K (b) Llama-3-8B-1048K

**Figure 23: Cost-efficiency analysis on Llama-3-8B-1048K.**

index construction and offloading incur negligible overhead. This efficiency is attributed to the low computational cost of our K-Means clustering—which is asymptotically superior to the quadratic self-attention—and the effective masking of offloading traffic via computation-communication overlapping. In comparison, MagicPIG and ShadowKV exhibit higher latencies due to specific bottlenecks: MagicPIG is limited by large LSH table construction, while ShadowKV incurs extra costs from low-rank key decomposition.

***Cost-Efficiency Analysis.*** To demonstrate the economic advantages of our design, Figure 23 compares the Llama-3-8B-1048K inference performance on two hardware tiers: a high-end NVIDIA H20 (96 GB HBM) using standard in-memory serving, and a consumer-grade NVIDIA RTX 4090 (24 GB HBM) powered by KVDrive. Figure 23a shows that KVDrive reduces the memory footprint by approximately 4× through effective sparse offloading, whereas the H20 baseline approaches memory saturation even with its larger capacity. By breaking the memory wall, KVDrive enables the RTX 4090 to achieve up to 3× higher throughput than the H20 baseline (Figure 23b). This result proves that with an optimized storage hierarchy, consumer hardware can effectively service long-context workloads that were previously limited to enterprise-grade GPUs.

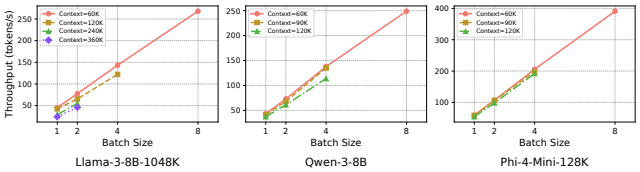



**Figure 24: Generation throughput (tokens/s) under different batch sizes.**

***Batch Size.*** Figure 24 presents the generation throughput of KVDrive with different batch sizes, evaluated across multiple models and context lengths. As shown, the system maintains a robust upward trend in throughput as batch size scales. This efficiency stems from our execution mechanism, which overlaps I/O and CPU-side pre-processing with GPU-side computation across micro-batches. By interleaving these distinct operations, KVDrive effectively hides data movement overheads and maximizes GPU utilization, thereby supporting larger batch sizes without significant performance degradation.

## 10 Conclusion

We present KVDrive, a holistic multi-tier KV cache management system for long-context LLM inference. KVDrive introduces three system-level techniques: attention-based in-GPU cache management with sliding-window reuse and lookahead eviction, elastic pipeline scheduling that overlaps selection, fetching, and computation with minimized stalls, and coordinated multi-tier KV storage. Our evaluation shows that KVDrive improves throughput by 1.74× over state-of-the-art offloading systems, while preserving accuracy. These results demonstrate the effectiveness of system-level cache and pipeline co-design in enabling efficient long-context inference under tight GPU budgets.

## 11 Future Work

Building on KVDrive, our future research will focus on three key directions. First, we aim to extend our holistic management to multimodal models, which present distinct KV cache access patterns compared to text-only LLMs. Second, we plan to investigate Processing-in-Memory hardware to offload selection and partial computation directly into storage tiers, further mitigating data movement bottlenecks. Finally, we plan to explore the synergy between KVDrive and compression techniques, such as quantization and pruning. We envision a *Tiered Mixed-Precision Storage* scheme that leverages KVDrive's ability to distinguish hot and cold KV blocks and assigns precision adaptively across tiers: maintaining high precision (e.g., FP16) for latency-critical hot blocks in HBM, while applying more aggressive quantization (e.g., INT4) to cold blocks spilled to SSD. This design trades modest (de)quantization overhead for higher effective I/O bandwidth and storage capacity, aiming to alleviate the I/O bottleneck in massive-context retrieval while preserving end-to-end generation quality.

# References


[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. 2024. Phi-4 Technical Report. arXiv:2412.08905 [cs.CL]

[2] Gradient AI. 2024. Llama 3-8B Instruct Gradient 1048k. https://huggingface.co/gradientai/Llama-3-8B-Instruct-Gradient-1048k. Accessed: 2025-05-15.

[3] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding. In *Annual Meeting of Association for Computational Linguistics*.

[4] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. 2025. MoE-Lightning: High-Throughput MoE Inference on Memory-constrained GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*.

[5] 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)*.

[6] Yaoqi Chen, Jinkai Zhang, Baotong Lu, Qianxi Zhang, Chengruidong Zhang, Jingjia Luo, Di Liu, Huiqiang Jiang, Qi Chen, Jing Liu, Bailu Ding, Xiao Yan, Jiawei Jiang, Chen Chen, Mingxing Zhang, Yuqing Yang, Fan Yang, and Mao Yang. 2025. RetroInfer: A Vector-Storage Approach for Scalable Long-Context LLM Inference. arXiv:2505.02922 [cs.LG]

[7] Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, and Beidi Chen. 2025. MagicPIG: LSH Sampling for Efficient LLM Generation. In *The Thirteenth International Conference on Learning Representations*.

[8] DeepSeek-AI. 2025. DeepSeek-V3.2-Exp: Boosting Long-Context Efficiency with DeepSeek Sparse Attention.

[9] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *PPoPP*.

[10] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *USENIX Annual Technical Conference (ATC)*.

[11] GitHub. 2025. Github copilot. https://github.com/features/copilot.

[12] Google. 2024. GPU machine types | Compute Engine Documentation. https://cloud.google.com/compute/docs/gpus.

[13] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. 2024. RULER: What's the Real Context Size of Your Long-Context Language Models?. In *First Conference on Language Modeling*.

[14] Jinwoo Jeong and Jeongseob Ahn. 2025. Accelerating LLM Serving for Multi-turn Dialogues with Efficient Resource Management. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[15] Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir 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 (NIPS)* (2024).

[16] 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 *SOSP*.

[17] Xunhao Lai, Jianqiao Lu, Yao Luo, Yiyuan Ma, and Xun Zhou. 2025. Flexprefill: A context-aware sparse attention mechanism for efficient long-sequence inference. *arXiv preprint arXiv:2502.20766* (2025).

[18] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *Operating Systems Design and Implementation*.

[19] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems (NIPS)* (2024).

[20] Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, Chen Chen, Fan Yang, Yuqing Yang, and Lili Qiu. 2024. RetrievalAttention: Accelerating Long-Context LLM Inference via Vector Retrieval. arXiv:2409.10516 [cs.LG]

[21] Jiaheng Liu, Dawei Zhu, Zhiqi Bai, Yancheng He, Huanxuan Liao, Haoran Que, Zekun Wang, Chenchen Zhang, Ge Zhang, Jiebin Zhang, Yuanxing Zhang, Zhuo Chen, Hangyu Guo, Shilong Li, Ziqiang Liu, Yong Shan, Yifan Song, Jiayi Tian, Wenhao Wu, Zhejian Zhou, Ruijie Zhu, Junlan Feng, Yang Gao, Shizhu He, Zhoujun Li, Tianyu Liu, Fanyu Meng, Wenbo Su, Yingshui Tan, Zili Wang, Jian Yang, Wei Ye, Bo Zheng, Wangchunshu Zhou, Wenhao Huang, Sujian Li, and Zhaoxiang Zhang. 2025. A Comprehensive Survey on Long Context Language Modeling. arXiv:2503.17407 [cs.CL]

[22] Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, Yuxin Wu, Yutao Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. 2025. MoBA: Mixture of Block Attention for Long-Context LLMs. *arXiv preprint arXiv:2502.13189* (2025).

[23] Meta AI. 2024. LLaMA 3.1 8B Instruct. https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct.

[24] Microsoft. 2024. NDasrA100_v4 sizes series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/ndasra100v4-series.

[25] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[26] Zebin Ren, Krijn Doekemeijer, Tiziano De Matteis, Christian Pinto, Radu Stoica, and Animesh Trivedi. 2025. An I/O Characterizing Study of Offloading LLM Models and KV Caches to NVMe SSD. In *Proceedings of the 5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. Association for Computing Machinery.

[27] 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 *Proceedings of the International Conference on Machine Learning (ICML)*.

[28] Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. 2025. ShadowKV: KV Cache in Shadows for High-Throughput Long-Context LLM Inference. In *ICML*.

[29] 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 International Conference on Machine Learning*.

[30] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[31] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. Efficient Streaming Language Models with Attention Sinks. In *ICLR*.

[32] Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. 2025. Strata: Hierarchical Context Caching for Long Context Language Model Serving. arXiv:2508.18572 [cs.DC]

[33] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388* (2025).

[34] An Yang, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoyan Huang, Jiandong Jiang, Jianhong Tu, Jianwei Zhang, Jingren Zhou, Junyang Lin, Kai Dang, Kexin Yang, Le Yu, Mei Li, Minmin Sun, Qin Zhu, Rui Men, Tao He, Weijia Xu, Wenbiao Yin, Wenyuan Yu, Xiafei Qiu, Xingzhang Ren, Xinlong Yang, Yong Li, Zhiying Xu, and Zipeng Zhang. 2025. Qwen2.5-1M Technical Report. arXiv:2501.15383 [cs.CL]

[35] Shang Yang, Junxian Guo, Haotian Tang, Qinghao Hu, Guangxuan Xiao, Jiaming Tang, Yujun Lin, Zhijian Liu, Yao Lu, and Song Han. 2025. LServe: Efficient Long-sequence LLM Serving with Unified Sparse Attention. In *ICML*.

[36] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. *arXiv preprint arXiv:2501.01005* (2025).

[37] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[38] 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)*. Association for Computational Linguistics, 23078–23097.

[39] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, Yan Yan, Beidi Chen, Guangyu Sun, and Kurt Keutzer. 2024. LLM Inference Unveiled: Survey and Roofline Model Insights. arXiv:2402.16363 [cs.CL]

[40] Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. 2025. PQCache: Product Quantization-based KVCache for Long Context LLM Inference. *Proc. ACM Manag. Data* (2025).

[41] 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 (NIPS)* (2023).