Understanding and Optimizing KV-cache Management for Long-Context LLM Inference

A RESEARCH PROFICIENCY EXAM PRESENTED

BY
AMIR HOSSEIN NAJAFI ZADEH
STONY BROOK UNIVERSITY

Technical Report FSL-26-02

August 2026

Abstract

Key-Value (KV)-cache makes Large Language Model (LLM) inference practical. By storing the key and value vectors that models produce for already-processed tokens, inference serving engines reduce the time of processing a context from quadratic to linear. KV-cache growth rate is high, and it cannot fit in the GPU memory. A single 128K-token context on Llama-3.1-70B requires roughly 30 GB of KV-cache, so with a number of concurrent clients, KV-cache exhausts the High Bandwidth Memory (HBM) of a single GPU. Inference engines therefore offload KV-cache data from HBM to external storage systems such as CPU DRAM, local SSDs, or remote storage, and restore it when needed by the model. Since the model reads KV-cache data in place, all data required by the model must be resident in GPU memory before processing. The time required to move this data is governed by the storage tier in which it resides, the available interconnect bandwidth, and input traffic patterns. When restoration is slow enough, recomputing the key and value vectors on the GPU is faster and produces identical output. Which of the two is better is not trivial to determine, as it depends on the GPU, the storage hierarchy, the context length, the cache-hit ratio, and the applicable service-level objectives (SLOs). This work characterizes that trade-off as an I/O problem. By studying where I/O becomes a bottleneck in the system, it identifies restoration as the only KV-cache operation on the critical path. It then measures restoration cost against recomputation on GPUs spanning the performance range. Finally, it reduces the KV-cache storage stack to a performance model based on per-tier capacity, per-tier and interconnect bandwidth, and GPU arithmetic throughput to identify optimization opportunities for KV-cache management. Using our performance model, evaluated across 1,764 hardware configurations, our study shows that hardware selection alone can change restoration time by two to three orders of magnitude at a fixed context length. Moreover, the recomputation curves of nearly every GPU examined intersect the restoration-time distribution, meaning that the restoration-versus-recomputation trade-off is nontrivial and becomes even more complex under different SLOs. To optimize KV-cache management in these complex scenarios, our model reveals one key opportunity: dividing a restore request proportionally between the storage path and the GPU can improve inference performance while still meeting SLOs. Across six GPUs and a sweep of cache-hit ratios it raises the fraction of configurations meeting objectives by 2–10$\times$ over always-restore and always-recompute. Future work implements this division as a KV-cache management mechanism inside a production-grade serving system and extends the model beyond its assumptions to cover contention and admission control, hierarchies deeper than limited paths, and distributed deployments in which KV-cache data is shared across nodes.

Keywords: LLM inference, KV-cache management, storage hierarchy, tiered caching, I/O characterization, performance modeling, restore-versus-recompute.

To My Family.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Large Language Model (LLM) inference is a rapidly growing workload [35]. By *inference* we mean the act of serving a trained model, turning a user's input into generated output. As of 2026, LLMs have been deployed in a wide range of applications, including search engines, chatbots, coding assistants, and multi-step agents that call tools and each other [57, 63, 71]. The increasing demand for LLMs has led to the development of new models with longer *context windows*, which is the maximum amount of input text a model can take into account at once. As a result, models are now able to process and generate output based on much larger input sequences [9, 21]. Longer contexts enable classes of application that shorter ones cannot support. Without them, a coding assistant could not see a whole repository, a retrieval-augmented application could not inline the documents it retrieves as evidence, and an agent could not carry a conversation across hours of interaction.

1.1 LLM Inference

The unit of computation in LLM inference is called a *token*, which is a small piece of text, such as a word or a subword. Internally, LLM inference relies on large matrix multiplications to generate new tokens based on the input tokens. The inference process involves two main phases. In the first phase, also known as the *prefill* phase, the model takes the input tokens and identifies their meaning and context, using a mechanism called *attention*. To compute attention, the model converts the input tokens into a vector representation and multiplies those vectors by the model's *weights* (*i.e.*, the parameters learned during training) to generate a set of vectors called *key* and *value* vectors. These vectors represent the attention information of each token for the model. Therefore, processing takes longer for models with more weights and more layers, and for requests with longer contexts. After the prefill phase, the model enters the second phase, known as the *decoding* phase. In this phase, the model generates output tokens one at a time, based on the input tokens and the previously generated output tokens. Because the model needs access to the attention information of every previous token to generate the next one, the naïve algorithm scales quadratically with the number of tokens in the context [62]. These computations are therefore time-consuming and are typically executed on GPU accelerators. Despite the high computational throughput of GPUs, when LLMs are used for long-context inference, the time to process a request remains high. For example, the NVIDIA H200 GPU requires 18 seconds to start generating output for a 128K-token context using Llama-3.1-70B [9, 21]. Beyond the high latency, long-context inference also consumes a significant amount of GPU time and memory, which leads to resource contention and increased costs when many users share an accelerator.

1.2 KV-cache

The standard remedy for this quadratic cost is to cache the intermediate results rather than recompute them [8, 11,51,67,68,70,71,81]. These results are stored as *Key-Value (KV)-cache* data. Despite its name, a KV-cache is not a traditional key-value store. Instead, it holds the key and value vectors produced by the transformer attention layers for tokens the model has already processed [45, 60]. The key and value vectors are stored together and reused by later attention computations, which avoids redundant recomputation. A KV-cache thus reduces the computational cost of extending a context from $O(N^2)$ to $O(N)$, because the vectors for prior tokens are retrieved from the cache rather than recomputed. With KV-cache reuse enabled, an H200 produces the first token for a 128K-token context in under 2 seconds [25], roughly a $9\times$ reduction from the 18 seconds it needs without reuse. Traditionally, KV-cache data resides in the GPU's High Bandwidth Memory (HBM) [2, 32, 49, 78], which is the fast memory physically attached to the accelerator. With many users and longer context lengths, KV-cache data quickly exceeds GPU memory. For instance, a 128K-token context on Llama-3.1-70B requires roughly 30GB of KV-cache; with only 10 concurrent users, demand exceeds 300GB, far beyond the HBM capacity of a single GPU (*i.e.*, an H200 GPU has 141GB of HBM). Moreover, a significant portion of GPU memory must be reserved for model weights, which grow with every model generation, further reducing the space available for KV-cache data. To address this limitation, KV-cache offloading to multi-tier storage has emerged as a key solution [7, 10, 13, 39, 52, 55, 67, 76]. A storage *tier* is one level of a memory or storage hierarchy, characterized by its capacity, its bandwidth (*i.e.*, the rate at which it can deliver data), and its cost per byte. Tiers are ordered so that each successive level is larger, slower, and cheaper than the one above it. For an inference server they run from GPU HBM, through CPU DRAM, to local storage devices such as SSDs, and finally to remote storage such as an object store. *KV-cache Offloading* is the practice of moving KV-cache data that does not fit in HBM down this hierarchy, and bringing it back when the model needs it again.

Offloading is effective because of the way inference services are used. An inference engine serves many users, or agents, concurrently. Each user creates a session, which is a conversation whose accumulated tokens form the context of every subsequent request within it. A session that has gone idle still owns KV-cache data, but nothing is reading that data, and the HBM it occupies is worth more to an active user. The cache system therefore evicts it, copying the data to a lower tier and freeing the space. When the idle user eventually returns, that data has to be brought back into HBM before the request can be processed. We call this operation restoration.

Mechanically, the inference engine and the cache system are separate components joined by a connector, a plug-in interface through which the engine requests KV-cache data without needing to know which tier currently holds it [52, 61]. The cache system maintains metadata recording, for each piece of KV-cache data, which tier it resides in. That data is not managed as one contiguous buffer per request, but in fixed-size blocks, each holding the key and value vectors for a fixed number of consecutive tokens [32].

Two properties of this arrangement determine the cost of restoration. First, every tier below HBM is reached across a shared interconnect (*e.g.*, a PCIe link between the CPU and the GPU), so the bandwidth a tier can actually deliver is the lesser of the device's own bandwidth and that of the link. Second, restoration grows both more frequent and more expensive as a deployment becomes busier: as the number of users and the length of their sessions increase, more KV-cache data is evicted, it is evicted further down the hierarchy, and a larger fraction of a returning session's context must be fetched from the slowest tier. Restoration is therefore not a cold-start cost that amortizes away. In a loaded system it is the common case, and its cost increases with load. Restoring evicted data is not, however, the only way to make it available again. Because a token's key and value vectors are a deterministic function of the tokens that precede it, the GPU can recompute them from the request's input, exactly as it did during the original prefill. The cache system therefore has two ways to obtain the same vectors, and the choice between them is the subject of the remainder of this report.

1.3 To Restore or To Recompute

Restoration and recomputation are interchangeable in one important respect: they produce the same key and value vectors, and therefore the same model output. They differ only in what resource they consume and how long they take. Restoration consumes storage bandwidth and interconnect bandwidth; recomputation consumes GPU compute cycles. Depending on which tier holds the required data and on the hardware characteristics of the deployment, recomputing keys and values for a request can be faster than retrieving them from KV-cache storage. The cache system must therefore decide *when to restore and when to recompute*.

Fixing that decision once, at deployment time, is insufficient for three reasons. First, KV-cache offloading involves many hardware components, including interconnect links, storage devices, and memory tiers, whose combinations form a large configuration space. Choosing restoration and recomputation thresholds generally requires deployment-specific benchmarking [5, 27, 28, 53], which is time-consuming, expensive, and easily invalidated when the workload or the hardware changes [1]. The size of that space makes it hard to design a single static solution that generalizes across deployments [15,16]. Second, the correct decision changes while the system runs: access patterns evolve, data distributions are unpredictable, and service-level objectives (SLOs) vary between tenants and over time. An SLO is a target that a service commits to meeting, such as a bound on request latency. Finally, neither extreme policy performs well. Always restoring makes the storage device the bottleneck and inflates tail latency once the retained KV-cache data spills to disk. Tail latency is the latency seen by the slowest requests, which we measure at the 95^{th} percentile. Always recomputing wastes a cache that was expensive to populate, and saturates the GPU, the most expensive resource in the deployment.

This work treats the decision as an I/O problem and characterizes it in three steps. It first measures where I/O time is spent during long-context inference and which cache operation gates request latency. It then reduces the KV-cache storage stack to a performance model, which allows the restoration and recomputation costs to be compared over a catalog of hardware rather than over the machines available for measurement. The model makes one improvement visible: because the two paths use independent resources and can run concurrently, a restore request can be divided between them rather than assigned to one, and the division that equalizes their completion times is computable at runtime from observed throughputs and checkable against explicit objectives. That division is derived and evaluated in simulation in this report; implementing it inside a production-grade serving system is stated as future work.

Chapter 2

Background

This chapter collects the material the rest of the report assumes. We begin with the attention mechanism, which produces KV-cache data and determines its structure (Section 2.1). We then describe the two phases of inference and the metrics used to talk about them (Section 2.2), explain how inference engines manage KV-cache data in fixed-size pages (Section 2.3), and use that unit to quantify how large the data becomes (Section 2.4). Sections 2.5 and 2.6 close the chapter with KV-cache reuse and KV-cache offloading.

2.1 Origin of KV-cache

An LLM is built from a stack of identical *layers*, each of which computes *attention*, a weighted combination in which every token draws information from the other tokens in the context. Each layer divides this work among several *attention heads* that run in parallel, each looking at a different aspect of the input. Within a head, every token is multiplied by three sets of learned weights, producing three vectors for that token: a *query*, a *key*, and a *value*. For a request of N tokens, the head then compares every query against every key, which is an $N \times N$ comparison:

$$\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\top}}{\sqrt{d_h}} + \mathcal{M}\right)\mathbf{V}, \tag{2.1}$$

where $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ stack the query, key, and value vectors of all N tokens, d_h is the *head dimension*, and $\mathcal{M}$ is a *causal mask* that prevents a token from drawing on tokens that follow it. Equation 2.1 has two consequences. First, the $N \times N$ comparison means that a full pass over an N-token context costs $O(N^2)$ work. This is the source of the quadratic growth in prefill time noted in Chapter 1. Second, the causal mask means that a token's key and value vectors depend only on that token and the ones before it, so they do not change when later tokens are appended. This is what makes them cacheable: once computed, they remain valid for the rest of the request, and for any other request that begins with the same tokens. Query vectors, by contrast, are consumed immediately and never stored, which is why the cache is a *KV*-cache and not a *QKV*-cache [21, 45, 60].

2.2 Inference Metrics

Serving a request proceeds in two phases with different resource profiles. **Prefill** processes the entire input context in one pass. All N input tokens are available simultaneously, so a layer can process them together as one large multiplication. Prefill is compute-bound, meaning its speed is limited by how fast the GPU can perform arithmetic. It is also where the $O(N^2)$ term lives, and it is the phase whose latency the user experiences as delay before the first word appears. **Decode** generates output tokens one at a time. Each step

processes a single new token, which attends over every key and value produced so far. Decode is memory-bandwidth bound, meaning its speed is limited by how fast the model weights and the KV-cache can be streamed to the compute units rather than by the arithmetic itself. The two phases are usually described with three metrics. *Time-to-first-token* (TTFT) is the interval between request arrival and the first output token, and is therefore essentially prefill latency plus queueing. *Time-per-output-token* (TPOT), sometimes reported as inter-token latency, is the steady-state decode step time. *Time-to-last-token* (TTLT) is end-to-end completion time, roughly $\mathrm{TTFT} + (\text{output length} - 1) \times \mathrm{TPOT}$. Service throughput, the rate of requests or tokens completed per second, is the aggregate counterpart to these per-request measures. The split matters because restoration and recomputation affect mostly prefill.

2.3 Paged Attention

Early inference engines allocated one contiguous buffer per request, sized for the maximum possible output length. This wastes space through internal fragmentation, the unused remainder of an allocation that is larger than what is actually stored in it, and it makes sharing between requests impossible. PagedAttention [32], introduced by the vLLM inference engine, replaced this with the virtual-memory abstraction that every modern engine now uses. KV-cache data is stored in fixed-size blocks, each holding the key and value vectors for a fixed number of consecutive tokens (commonly 16 or 32), and a per-request block table maps a block's logical index within the request to its physical location. SGLang [78] and others adopt the same structure. A block is the unit in which KV-cache data is allocated, evicted, offloaded, and restored. Paging imposes one requirement on the data path. Because the attention kernel reads blocks in place through the block table, rather than gathering them into a contiguous staging buffer, every block a request needs must be resident in GPU memory before the kernel runs [32]. There is no way to stream a block from disk as the kernel touches it. The transfer is therefore an explicit step that precedes computation, and its cost is exactly what a restore-versus-recompute policy is choosing between. Paging also makes sharing cheap: two requests that begin with the same tokens can point their block tables at the same physical blocks, which is the mechanism behind the reuse techniques of Section 2.5.

2.4 Memory Footprint

The per-token KV-cache footprint follows directly from the architecture. For a model with L layers, H_{kv} key/value heads, head dimension d_h, and b bytes per element,

$$S_{\text{token}} = 2 \cdot L \cdot H_{kv} \cdot d_h \cdot b, \tag{2.2}$$

where the leading factor of two accounts for storing both a key and a value. A request holding N tokens occupies $N \cdot S_{\text{token}}$ bytes, and a block holding T_{block} tokens occupies

$$S_{\text{block}} = T_{\text{block}} \cdot S_{\text{token}}. \tag{2.3}$$

Table 2.1 evaluates Equation 2.2 for the two Llama-3.1 models used in this report. For Llama-3.1-8B with 16-token blocks, a block occupies 2 MB, which is the block size we assume throughout. The 70B row explains the pressure described in Chapter 1: at 128K tokens, one session's cache is measured in tens of gigabytes, and ten such sessions exceed the HBM capacity of any single GPU on the market. The 30 GB figure quoted in Chapter 1 falls between the `bf16` and 8-bit rows; the exact footprint a deployment sees depends on the numeric precision it chooses for KV-cache data [31,36,37]. The footprint is linear in context length with a large constant, so capacity is the binding resource rather than any property of the access pattern. The constant is set by the model architecture and the chosen precision, both of which are fixed at deployment time. The only remaining lever at runtime is *where* the data lives.

Model	L	H_{kv}	d_h	Per token	128K context
Llama-3.1-8B (`bf16`)	32	8	128	128 KiB	16 GiB
Llama-3.1-8B (8-bit)	32	8	128	64 KiB	8 GiB
Llama-3.1-70B (`bf16`)	80	8	128	320 KiB	40 GiB
Llama-3.1-70B (8-bit)	80	8	128	160 KiB	20 GiB

Table 2.1: KV-cache footprint per token and for a full 128K-token context, computed from Equation 2.2 with the Llama-3.1 architecture parameters [21]. Here `bf16` denotes 16-bit floating point, so $b = 2$ bytes; the 8-bit rows assume the KV-cache is stored at half that precision while the weights are not.

2.5 KV-cache Reuse

Retaining KV-cache data beyond the lifetime of a single request pays off because real workloads are repetitive: a multi-turn conversation re-sends the history of earlier turns as its prefix, a retrieval-augmented application inlines the same documents into many different prompts, and an agent pipeline prefixes every call with the same instructions and tool descriptions [26, 70, 71]. Measurements from a large cloud provider confirm that reuse opportunities across requests are both frequent and skewed toward a small set of popular prefixes [64]. Systems exploit this in several ways. The basic one is *prefix caching*: the blocks of a completed request are retained so that a later request beginning with the same tokens can skip prefill for the shared part. Chapter 6 surveys the schemes built on top of it. Reuse creates the demand for capacity. A cache that is discarded at the end of each request never needs a second tier; a cache that is retained across sessions rapidly outgrows HBM.

2.6 KV-cache Offloading

To retain more KV-cache data than HBM can hold, modern cache systems [39,41] offload blocks to external storage tiers: CPU DRAM, local storage devices such as SSDs, and remote storage systems such as object stores, which serve data over a network rather than over a local bus. Inference engines integrate with these cache systems through pluggable connectors [52,61], which coordinate data placement and retrieval without the engine having to know which tier a block came from. The resulting hierarchy has four levels. At the top, the GPU package holds the compute cores and the HBM attached to them; KV-cache blocks are produced in HBM and are readable by the attention kernel only while they remain there. Below it, and reachable only across the host interconnect, sit CPU DRAM, one or more local storage devices, and a remote tier reached over a network. Capacity increases and bandwidth decreases at each step down, so higher tiers provide greater throughput at higher cost per byte and lower capacity, and lower tiers provide capacity at lower throughput. Table 2.2 gives representative figures. HBM delivers roughly 3,360GB/s on an H200 [48], while terabytes of NVMe storage, the flash devices that attach directly to the PCIe bus, deliver on the order of 10GB/s. That is more than two orders of magnitude less. Table 2.2 also states a constraint that recurs in our model. Every tier below HBM is reached across the host interconnect, so its effective bandwidth is $\min(\beta_{\text{tier}}, \beta_{\text{link}})$ rather than β_{tier} alone. An NVMe array whose device bandwidth exceeds that of a Gen3 link therefore delivers no more than the link permits. In this work we consider a hierarchy consisting of GPU HBM, single-node CPU DRAM, and a local SSD connected through a single interconnect such as PCIe. More complex hierarchies (*e.g.*, multiple GPUs, disaggregated memory pools, or remote tiers with their own queueing behavior) are left to future work.

Tier	Typical capacity	Typical bandwidth	Notes
GPU HBM	24–141 GB	900–3,360 GB/s	shared with model weights
CPU DRAM	0.1–2 TB	50–400 GB/s	reachable only over PCIe
PCIe Gen4/5 $\times$16	n/a	32–64 GB/s	caps every lower tier
Local NVMe	1–60 TB	3–14 GB/s	Gen4/Gen5 devices
Local SATA SSD	0.5–8 TB	$\approx$0.55 GB/s	interface-limited
Remote object	petabytes	0.1–12.5 GB/s	network- and protocol-limited

Table 2.2: Representative capacity and bandwidth of the tiers in a KV-cache hierarchy. Gen4 and Gen5 denote successive PCIe generations, each roughly doubling the per-lane rate of the last. Figures are drawn from vendor specifications and public benchmark databases [3, 19, 48, 59]; exact values vary by part. Note that the PCIe row is not a storage tier but a ceiling: no tier below it can deliver more than the link allows.

Chapter 3

I/O Study of LLM Inference

Chapter 2 established that KV-cache data outgrows HBM and that inference engines offload it to external storage tiers. This chapter characterizes the I/O behavior that follows from offloading and its effect on inference performance. Section 3.1 identifies which cache operation lies on the critical path. Section 3.2 describes the path a KV-cache block takes through the storage hierarchy. Finally, Section 3.3 reports measured behavior on two GPUs.

3.1 Where the I/O Time Goes

A cache system exposes three operations to the inference engine for KV-cache offloading [39, 61]. *Lookup* resolves which of a request's blocks are cached and in which tier; *store* writes newly-produced or evicted blocks down the hierarchy; and *restore* brings blocks back into GPU memory. Their contributions to request latency differ by orders of magnitude. **Lookup** touches metadata only. A block's metadata is on the order of tens of bytes (*e.g.*, an identifier, a tier, and an offset) against a 2 MB payload, so the metadata for a full hierarchy resides in CPU DRAM and never reaches a storage device. Lookup contributes microseconds. **Store** moves data, but not on the critical path. When a session goes idle and its blocks are evicted, the request that produced them has already completed, and store completion gates nothing. A store backlog competes for the device bandwidth that restores need, so its cost appears as reduced effective restoration bandwidth rather than as latency attributable to any single request. **Restore** blocks. Since the attention kernel reads blocks in place through the block table (Section 2.3), a request cannot enter prefill until every block it needs is resident in HBM. A restore is asynchronous from the cache system's perspective but synchronous from the inference engine's, which must wait on complete restore before scheduling the request. The result is queueing delay directly attributable to I/O [20]. The magnitudes follow from Table 2.1. A single 1,000-token request on Llama-3.1-8B requires $1000 \times 128\,\mathrm{KiB} \approx 125$ MB of KV-cache data, or roughly 227 ms of transfer from a SATA SSD at 0.55 GB/s, before queueing and before competing traffic. A 128K-token session on the same model requires 16 GB, or about 30 seconds of transfer. At the lower tiers, restoration latency dominates prefill latency.

3.2 KV-cache Offloading Flow

A block traverses the hierarchy described in Section 2.6 in a fixed pattern: produced in HBM, demoted downward under capacity pressure, and promoted back to HBM on demand. The position it occupies at the time of a restore determines the cost of that restore. Blocks are produced in HBM during both prefill and decode phases. They remain in GPU memory while the request is running. Usually, one GPU worker does the prefill step and passes the results to another GPU worker for decode. The data is transferred through

interconnects such as PCIe or NVLink. During execution of a single request, if GPU memory pressure is hit, blocks may be offloaded to CPU DRAM and still be used instantly. The request life cycle is shorter than a session; therefore, the first top tiers will handle that case. The cache system records each block's tier in its metadata, so residency is a property tracked per block rather than per session. When GPU memory pressure rises, blocks belonging to idle sessions are copied to CPU DRAM and their HBM space is freed. When DRAM fills, the same mechanism spills them to the local storage device, and in deployments with a remote tier, from there to the network. Eviction is a background write, so its latency is hidden, but each level of descent multiplies the cost of the eventual restore by the bandwidth ratio between adjacent tiers, which Table 2.2 shows to span more than two orders of magnitude. When an idle session returns, a lookup classifies its blocks into four categories: still resident in HBM, resident in DRAM, resident on disk, and absent from the hierarchy entirely. The first category costs nothing. The second and third require a transfer whose duration is governed by the effective bandwidth of the tier, which is the lesser of the device bandwidth and the interconnect bandwidth (Section 2.6). The fourth must be recomputed, because the data no longer exists anywhere.

The rate at which requests reach the lower categories is set by capacity. Consider a single H200 with 141 GB of HBM serving Llama-3.1-70B. Weights in `bf16` consume $2P \approx 140$ GB, so the model must be sharded across GPUs or quantized. In a two-GPU parallel configuration, roughly 70 GB of HBM per device remains available for KV-cache data. From Table 2.1, one 128K-token session requires 40 GB of `bf16` KV-cache spread across the two devices, so a single GPU's free HBM holds on the order of three such sessions. Inference services retain conversational state across requests. A chat deployment with a few hundred concurrent sessions, an agent framework whose workers carry long tool histories, or a coding assistant holding a repository in context each produce more retained KV-cache data than a GPU can hold, and Section 2.5 established that the data is worth retaining because every session resends its history on the following turn. As the number of users and the length of their sessions grow, the fraction of a returning session's context that must be fetched from the slowest occupied tier grows with them. Restoration from disk is therefore the common case in a loaded system rather than a cold-start cost that amortizes away.

3.3 Measured Behavior

To observe this behavior end to end, we ran a chatbot workload against a cache hierarchy sized to force offloading, on two GPUs bracketing the performance range: an NVIDIA H200 and an NVIDIA RTX A5000, backed in both cases by a SATA SSD. Table 3.1 gives the configuration. As shown in the table, the HBM and DRAM cache capacities are constrained far below what either machine provides. This compresses the transition from HBM-resident to DRAM-resident to disk-resident into a session count reachable on a single node; the shape of the transition is set by the ratio of working set to capacity rather than by absolute capacity. The workload executes twice. The first pass populates the hierarchy and triggers evictions, and the second replays the same sessions so that each request has cache data to restore; without the first pass every request is a cold miss and the measurement reduces to prefill. The system is sampled periodically as sessions accumulate, increasing the aggregate context length from 1.6×10^3 to 1.6×10^6 tokens. Reported context lengths are the total KV-cache footprint across all live sessions rather than the length of an individual conversation, since tier residency is determined by the total footprint competing for each tier. The workload is drawn from ShareGPT [54], a corpus of real human-assistant dialogues whose multi-turn structure reproduces the reuse pattern of Section 2.5 without synthesis.

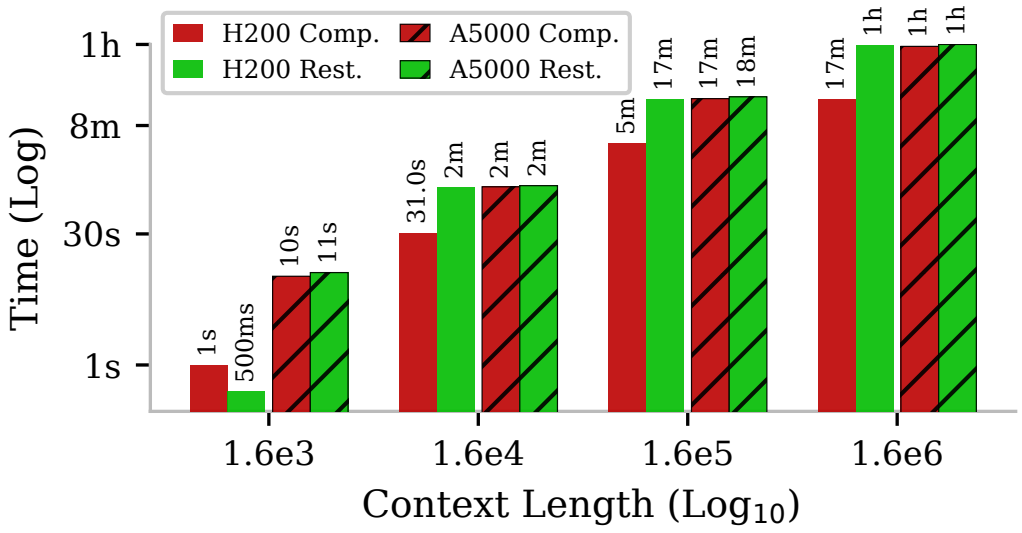


Figure 3.1: Processing time for different aggregate context lengths when the KV-cache data is computed on the GPU versus restored from SSD, for an H200 and an RTX A5000. Both axes are logarithmic.

Parameter	**Value**
Model	Llama-3.1-8B-Instruct [21]
GPUs	NVIDIA H200; NVIDIA RTX A5000
Storage tier	SATA SSD
HBM cache capacity	2 GB
DRAM cache capacity	1 GB
KV block size	16 tokens ($\approx$2 MB)
Workload	ShareGPT-derived chatbot [54]
Sessions	200
Requests / session	$\approx$5
Input tokens / request	$\approx$1,000
Passes	2 (populate, then replay)
Aggregate context swept	1.6×10^3 to 1.6×10^6 tokens

Table 3.1: Configuration of the I/O case study. Cache capacities are constrained far below what the hardware provides so that the working set spills to disk at session counts drivable on a single node.

Figure 3.1 reports the time to make a request's context available in HBM by recomputation and by restoration. As shown in the figure, for the H200 at the smallest aggregate context, restoration takes 500 ms against 1 s for recomputation, because the KV-cache data still fits within the constrained HBM cache and no device I/O occurs. As the aggregate context grows past the HBM and DRAM caps, restoration time rises to two minutes, then seventeen minutes, then to the order of an hour, inheriting the piecewise structure of the hierarchy with a distinct step at each tier boundary. Recomputation time depends on GPU compute capability and model architecture, not on data placement. On the H200 it grows from 1 s to about 17 minutes across three orders of magnitude of context, remaining below restoration at every point past the smallest. Restoration inflates end-to-end inference latency by up to 10$\times$ on high-performance GPUs. The A5000 exhibits the opposite behavior. Its recomputation is slow enough (10 s where the H200 requires 1 s) that restoration is already competitive at small contexts, and at larger contexts the two track closely: two minutes against two minutes, seventeen against eighteen. On this GPU the two mechanisms are near-substitutes across most of the range.

Chapter 4

KV-cache Tier Simulator

Chapter 3 established the restore-versus-recompute trade-off on two GPUs and one storage device. Determining how far that result generalizes by measurement is impractical, since it would require assembling on the order of a thousand physical machines. We therefore built a performance-modeling simulator of the KV-cache storage stack and swept the space analytically. Section 4.1 describes how the inference stack of Chapter 2 is reduced to a model and what that reduction discards. Section 4.2 gives the formulas and the configuration schema.

4.1 From the Inference Stack to a Simulator

The simulator computes a single quantity: for a given hardware configuration and context length, the time required to make that context's KV-cache data available in GPU memory, by restoration and by recomputation. It is a performance model in the tradition of roofline analysis for LLM inference [50, 73], not a discrete-event simulator of an inference engine. The reduction proceeds by discarding every component of the stack that does not affect that quantity. The model and the inference engine are represented only by the parameters that determine bytes and FLOPs: layer count, key/value head count, head dimension, element width, block size, and parameter count. The cache system is represented by per-tier capacities and the placement rule that restoration draws from higher tiers until their capacity is exhausted before falling back to lower ones, which approximates the behavior of deployed systems [39]. The hardware is represented by per-tier bandwidths, one interconnect bandwidth, and the GPU's peak arithmetic rate. What remains is a set of closed-form expressions that can be evaluated over the full cross-product of a hardware catalog. The simulator is implemented in approximately 1,000 lines of Python and Go and is publicly available.[1] The Python component expresses the model of Section 4.2 and drives the parameter sweep; the Go component evaluates configurations and scheduling decisions at a rate that makes the full cross-product tractable.

Scope and Assumptions

The hierarchy modeled is one GPU HBM, one CPU DRAM, and one disk, connected by a single interconnect. Besides KV-cache data, only model weights are assumed to occupy HBM. The following mechanisms are omitted.

- **Prefetching.** Blocks are fetched when requested, never in anticipation.
- **Inter-tier data movement.** Promotion and demotion between DRAM and disk while a request is in flight are not modeled.

[1] https://github.com/sbu-fsl/KVC-TierSim

- **Random access patterns.** A request's blocks are treated as a dense sequential range, which matches how attention consumes them (Section 2.1).
- **Contention.** Each configuration is evaluated in isolation, with no competing store traffic and no queueing across concurrent requests.
- **Model sharding.** A single GPU holds the whole model.

Two consequences bound the conclusions the model supports. First, absolute times are optimistic for both mechanisms: peak bandwidths and peak arithmetic rates are used, and hardware utilization is set to unity. The comparison between the two mechanisms is more robust than either absolute curve, though the residual bias favors recomputation, since perfect utilization helps the GPU while the storage side remains capped by the interconnect. Second, contention would reduce both the achievable restoration bandwidth and the achievable recomputation rate in a loaded system, which makes the online problem harder rather than easier.

4.2 Formulas and Schemas

Table 4.1 collects the notation used here and in Chapter 5.

Symbol	Meaning
P	model parameter count
L, H_{kv}, d_h, b	layers, KV heads, head dimension, bytes per element
T_{block}	tokens per KV-cache block
B, S_{block}	bytes per KV-cache block
N	blocks required by a request
M	of those, blocks missing from the hierarchy (cache misses)
k	cache-hit blocks reassigned from restoration to recomputation
N_c, N_s	blocks recomputed ($M + k$) and restored ($N - M - k$)
C_{tier}	capacity of a tier, in blocks
β_{tier}	raw bandwidth of a tier
β_{link}	interconnect (*e.g.*, PCIe) bandwidth
Φ_{peak}	peak FLOP/s of the GPU
η	hardware utilization efficiency (MFU)
F_{token}	FLOPs to process one token ($\approx 2P$)
$\hat{t}$	per-token compute time
X, Y	GPU recomputation and storage restoration throughput (bytes/s)
r	target request rate (requests/s)
p	target P95 prefill latency (s)
T_c, T_s, T_{req}	recomputation, restoration, and request completion time

Table 4.1: Notation used in the performance model (Chapter 4) and in the findings derived from it (Chapter 5).

Capacity

How much KV-cache data a tier holds determines where a request's blocks are found. HBM must also hold the model weights, so for a model with P parameters stored in `bf16`,

$$\mathit{EffectiveCapacity}_{\text{HBM}} = \mathit{Capacity}_{\text{HBM}} - 2P, \tag{4.1}$$

and the number of blocks that fit is

$$\mathit{KVCSpace}_{\text{HBM}} = \left\lfloor \frac{\mathit{EffectiveCapacity}_{\text{HBM}}}{S_{\text{block}}} \right\rfloor, \tag{4.2}$$

with S_{block} given by Equation 2.3. The subtrahend $2P$ grows with each model generation while $Capacity_{\text{HBM}}$ grows more slowly, so KV-cache space on a given GPU shrinks as models grow. DRAM and disk carry no such reservation: $C_{\text{tier}} = \lfloor Capacity_{\text{tier}} / S_{\text{block}} \rfloor$.

Restoration

Given a request for N blocks, blocks are assigned to tiers top-down, filling each tier's capacity before spilling to the next:

$$\begin{aligned} n_{\text{HBM}} &= \min(N,\ C_{\text{HBM}}), \\ n_{\text{DRAM}} &= \min(N - n_{\text{HBM}},\ C_{\text{DRAM}}), \\ n_{\text{disk}} &= N - n_{\text{HBM}} - n_{\text{DRAM}}. \end{aligned} \tag{4.3}$$

Blocks already in HBM require no restoration. For DRAM and disk, throughput is limited by the lesser of the tier bandwidth and the interconnect bandwidth, since every byte from those tiers crosses the same link:

$$T_s = \frac{n_{\text{DRAM}} \cdot S_{\text{block}}}{\min(\beta_{\text{dram}}, \beta_{\text{link}})} + \frac{n_{\text{disk}} \cdot S_{\text{block}}}{\min(\beta_{\text{disk}}, \beta_{\text{link}})}. \tag{4.4}$$

Equation 4.4 is piecewise linear in N with breakpoints at the tier capacities, which is the step structure observed in Figure 3.1. Summing the two terms treats transfers from the two tiers as serialized on the shared link; a system with independent paths would take the maximum instead. Writing the aggregate restoration path as a single effective bandwidth Y gives the form used in Chapter 5:

$$T_s = \frac{NB}{Y}, \qquad Y = \min(\beta_{\text{tier}}, \beta_{\text{link}}). \tag{4.5}$$

Recomputation

Compute time uses the roofline approximation, which divides the work to be done by the rate the hardware achieves. With F_{token} the floating-point operations (FLOPs) required per token, Φ_{peak} the peak FLOPs per second of the GPU, and η the sustained fraction of that peak, known as model FLOPs utilization (MFU) [9], the per-token compute time is

$$\hat{t} = \frac{F_{\text{token}}}{\Phi_{\text{peak}} \cdot \eta}, \tag{4.6}$$

with $F_{\text{token}} \approx 2P$ for a model that uses all P of its parameters on every token. Because processing each block requires attending to all preceding blocks, processing N blocks costs

$$t_{\text{compute}} = N^2 \cdot T_{\text{block}} \cdot \hat{t}. \tag{4.7}$$

The N^2 term is the causal-attention cost of Equation 2.1; summing i over $i = 1 \ldots N$ gives $N(N+1)/2$, and the model absorbs the constant factor. Expressing recomputation in the units of Equation 4.5 defines the GPU's effective recomputation throughput X, the number of KV-cache bytes it produces per second:

$$X = \frac{S_{\text{token}}}{\hat{t}} = \frac{2 \cdot L \cdot H_{kv} \cdot d_h \cdot b \cdot \Phi_{\text{peak}} \cdot \eta}{F_{\text{token}}}, \tag{4.8}$$

combining Equations 2.2 and 4.6. With X and Y in the same units, the choice between the two mechanisms reduces to their ordering: recomputation is faster when $X > Y$, restoration when $X < Y$. Neither Equation 4.6 nor Equation 4.7 involves storage, and Equation 4.4 involves no arithmetic rate, so the two costs move independently across the configuration space.

Configuration schema

A configuration is a tuple of one GPU, one interconnect, one DRAM specification, and one storage device, each drawn from a catalog of published parts. Table 4.2 gives the catalog: six GPUs, six PCIe generations, seven DRAM configurations, and seven storage devices, or $6 \times 6 \times 7 \times 7 = 1{,}764$ combinations. Device bandwidths and capacities were collected from NVIDIA [48], FS [19], Bizon-Tech [3], and UserBenchmark [59].

Dimension	**Levels**	**Range covered**
GPU	6	Tesla V100, RTX A5000, RTX 6000 Ada, A100, H100, H200
Interconnect	6	PCIe generations, spanning roughly $32\times$ in bandwidth
CPU DRAM	7	capacity/bandwidth pairs across generations
Storage device	7	SATA SSD through Gen5 NVMe
Total configurations		$6 \times 6 \times 7 \times 7 = 1{,}764$

Table 4.2: The hardware configuration space swept in Chapter 4. Device specifications were collected from NVIDIA [48], FS [19], Bizon-Tech [3], and UserBenchmark [59].

Chapter 5

Findings

This chapter reports what the simulator of Chapter 4 establishes. Section 5.1 quantifies the size of the solution domain, showing that hardware selection alone moves restoration time by two to three orders of magnitude and that no single mechanism dominates across it. Section 5.2 identifies the improvement the model makes visible: an allocation that divides a restore request between the two paths rather than choosing one of them.

5.1 A Large Domain of Solutions

Two results establish that the restore-versus-recompute ordering is not a property that can be fixed at design time. Figure 5.1 evaluates Equations 4.4 and 4.7 over the 1,764 configurations of Table 4.2. Each violin summarizes restoration times across all configurations at a given session context length; the dashed curves give recomputation time per GPU, which is independent of the storage configuration.

Three properties of the distribution follow. **Hardware choice alone spans two to three orders of magnitude.** At a fixed context length, with the same model and the same volume of data to move, varying only the storage device, the DRAM configuration, and the interconnect changes restoration time by a factor of 10^2 to 10^3. No property of the workload accounts for that spread. **The spread is widest where the hierarchy is in play.** At smaller context lengths the violins are tall because the outcome differs qualitatively across configurations: on a large-HBM GPU the data still fits in HBM and restoration costs nothing, while on a small-HBM GPU with a modest DRAM tier the same request already reaches disk. As context length grows past the effective HBM capacity of every configuration, the distribution narrows and shifts upward, becoming constrained by disk bandwidth and capacity. **Recomputation curves intersect the restoration distributions.** For nearly all GPUs the recomputation curve passes through the violins rather than above or below them, so for a given context length there exist storage configurations under which restoration is faster and others under which recomputation is faster. The exception is the H200, whose curve lies below the distribution across the range shown; on that GPU recomputation is the better choice almost everywhere. The domain of viable configurations is therefore large, and membership in it depends jointly on the accelerator, the storage device, the interconnect, and the context length. Enumerating it by benchmarking is expensive and is invalidated by any change to hardware or workload [5,27,28,53], a difficulty documented for multi-tier storage caches generally [15, 16, 79].

5.2 Potential Improvements

Because restoration and recomputation consume disjoint resources, they can proceed concurrently, and a request need not use a single mechanism for all of its blocks. The simulator makes the consequences of that observation measurable. The choice arises at a single point in a request's path. A restore request carries

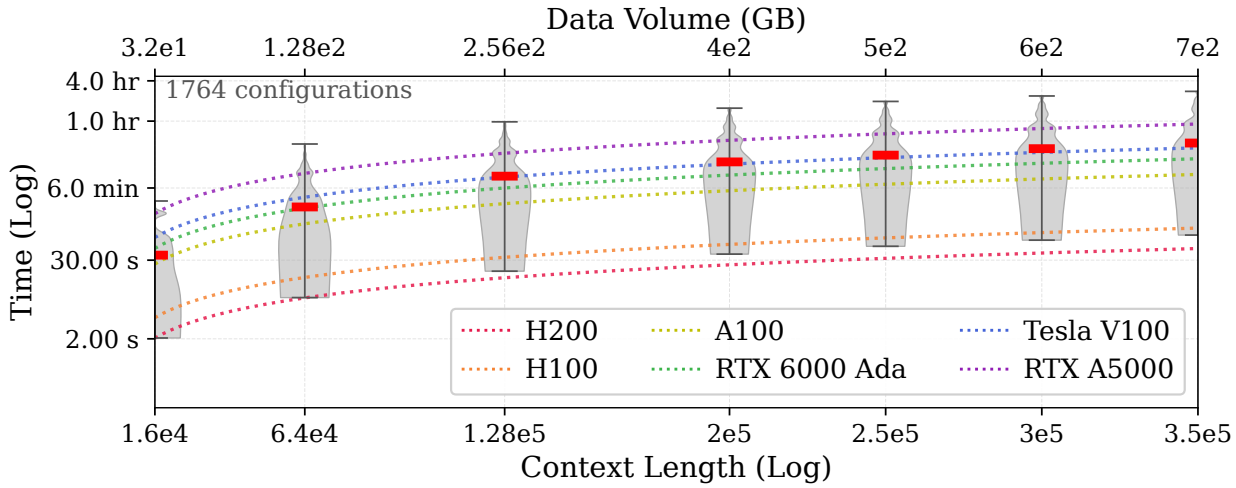


Figure 5.1: Performance modeling results for recomputation and restoration across 1,764 hardware configurations. Dashed lines show recomputation times for different GPUs; violin plots show restoration-time distributions, with red lines denoting medians. The x-axes give session context length and the corresponding KV-cache data volume; the y-axis is completion time on a logarithmic scale.

N blocks, and the cache lookup of Section 3.1 classifies them by residency into four groups. Blocks still resident in HBM require no action, since the attention kernel can already read them. The M blocks absent from the hierarchy must be recomputed, as no copy of them exists. Blocks resident in CPU DRAM are treated as a transient staging point rather than as a decision point and restoration from it is therefore always preferable. The remaining blocks, evicted to the disk tier, are the ones for which both mechanisms are viable, and they are what the allocation divides. The two resulting groups are issued concurrently: one set of block requests to the storage path and one recomputation batch to the GPU. Because the paths are independent, the request completes when the slower of the two finishes, at $T_{req} = \max(T_c, T_s)$, and the allocation problem is to choose the division that minimizes that maximum.

Balancing the two paths

Let X be the observed GPU recomputation throughput in bytes/s, Y the observed storage restoration throughput, N the number of blocks of size B in a restore request, and M the number of cache misses. Letting k cache-hit blocks be reassigned from restoration to recomputation,

$$N_c = M + k \qquad \text{and} \qquad N_s = N - M - k, \tag{5.1}$$

with $k \in [0,\ N - M]$ the single decision variable. Because the two paths run in parallel, completion time is set by the slower of them:

$$T_c = \frac{N_c B}{X}, \qquad T_s = \frac{N_s B}{Y}, \qquad T_{req} = \max(T_c, T_s). \tag{5.2}$$

The maximum is minimized when the two paths finish together. Setting $T_c = T_s$ and solving for k,

$$\begin{aligned} \frac{(M+k)B}{X} &= \frac{(N-M-k)B}{Y} \\ Y(M+k) &= X(N-M-k) \\ k(X+Y) &= XN - M(X+Y) \\ k^\star &= \frac{NX}{X+Y} - M, \end{aligned} \tag{5.3}$$

clamped to the feasible range:

$$k = \max\left(0,\ \min\left(N-M,\ \frac{NX}{X+Y} - M\right)\right). \tag{5.4}$$

When the clamp is inactive, substituting into Equation 5.1 gives

$$N_c = \frac{NX}{X+Y}, \qquad N_s = \frac{NY}{X+Y}, \tag{5.5}$$

so each path receives blocks in proportion to its own throughput, and

$$T_{req} = \frac{NB}{X+Y}. \tag{5.6}$$

The system behaves as a single delivery path of bandwidth $X + Y$. Always-restore achieves NB/Y and always-recompute achieves NB/X, so the gain over the better fixed mechanism is largest when $X \approx Y$, where it approaches a factor of two, and this is also the regime in which a static threshold is least reliable: small measurement errors reverse the ranking.

> **Proposition 1.** k as defined in Equation 5.4 minimizes $T_{req}(k) = \max(T_c(k), T_s(k))$ over $k \in [0, N-M]$.
>
> *Proof.* $T_c(k) = (M+k)B/X$ is continuous and strictly increasing in k; $T_s(k) = (N-M-k)B/Y$ is continuous and strictly decreasing. The pointwise maximum of an increasing and a decreasing function is quasi-convex, and is minimized at their crossing point if it lies in the interval, and otherwise at the nearer endpoint, which is what clamping $k^\star$ computes. □

Since the M cache misses admit no alternative, $T_{req} \geq MB/X$ for every k, and the achievable completion time is

$$T_{req} = \max\left(\frac{MB}{X},\ \frac{NB}{X+Y}\right), \tag{5.7}$$

the first term binding when the request is mostly misses and the second when it is mostly hits. The cache-hit ratio is therefore the axis along which the allocation must be evaluated.

Limiting cases

Equation 5.5 reduces to each fixed mechanism at the extremes. For $X \gg Y$, $N_c \to N$ and the allocation converges on always-recompute while still routing a $Y/(X+Y)$ share to a path that would otherwise be idle. For $X \ll Y$, the A5000 case, $N_s \to N$. For $M = N$ every block is a miss, k clamps to zero, and the request is an ordinary prefill. For $M > NX/(X+Y)$ the clamp binds at $k = 0$: mandatory recomputation already exceeds the GPU's balanced share, $T_{req} = MB/X$ by Equation 5.7, and no allocation improves on it.

Algorithm 1: I/O-aware restore/recompute allocation

Input : N blocks requested; M cache-miss blocks; block size B
Parameters: observed throughputs X, Y; SLO targets p, r
Output : blocks to recompute N_c, blocks to restore N_s, feasibility verdict

1 $\tau \leftarrow \min(p,\ 1/r)$; // binding objective
2 **if** $Y = 0$ **then return** $(N,\ 0,\ NB/X \leq \tau)$;
3 $k^\star \leftarrow NX/(X+Y) - M$; // balance point, Eq. 5.3
4 $k \leftarrow \max(0,\ \min(N-M,\ k^\star))$; // clamp to feasible range
5 $N_c \leftarrow M + k$;
6 $N_s \leftarrow N - M - k$;
7 $T_c \leftarrow N_c B/X$;
8 $T_s \leftarrow N_s B/Y$;
9 $T_{req} \leftarrow \max(T_c,\ T_s)$;
10 **if** $T_{req} \leq \tau$ **then**
11 | **return** $(N_c,\ N_s,$ FEASIBLE$)$;
12 **else**
13 | **return** $(N_c,\ N_s,$ BEST-EFFORT$)$; // minimizes the violation
14 **end**

Cost of the decision

Algorithm 1 states the allocation. The algorithm performs a fixed number of arithmetic operations with no loops, no search, and no state beyond the four parameters it reads, so its cost is $O(1)$ per restore request, independent of N, of the number of tiers, and of the number of concurrent sessions. The guard on line 2 covers the case in which the storage path has stalled to zero observed throughput, where Equation 5.3 would divide by zero. Because $k^\star$ is real-valued while blocks are discrete, rounding perturbs T_{req} by at most $B/\min(X,Y)$, the transfer time of one block, which for a 2 MB block on the slowest tier of Table 2.2 is under 4 ms. Measured over 20.2 million timed scheduling decisions in the simulator, the allocation averages 5.07 μs per decision and sustains approximately 2×10^5 decisions/s; scheduling 10^6 requests costs about 5.1 s against 0.23 s for the fixed mechanisms. Against the operation it governs the cost is negligible: the 227 ms of transfer computed in Section 3.1 for a single 1,000-token request from a SATA SSD exceeds it by five orders of magnitude. The parameters it reads already exist in a deployed system, since the cache subsystem measures its own transfer rates and the inference engine tracks its achieved throughput, so no additional state is required.

Minimizing completion time is not equivalent to meeting a service-level objective. The model permits both objectives introduced in Section 2.2 to be written as constraints on the same quantity and checked across the configuration space. The **tail-latency constraint** bounds request completion time by the target P95 prefill latency p:

$$T_{req} \leq p. \tag{5.8}$$

The **throughput constraint** is a resource-utilization condition. At a target request rate of r requests/s, neither path may be overloaded:

$$r\,T_c \leq 1 \qquad \text{and} \qquad r\,T_s \leq 1, \tag{5.9}$$

which states that per-request service time on a resource must not exceed the inter-arrival time, or the corresponding queue grows without bound. The three constraints collapse into one. Since $rT_c \leq 1 \wedge rT_s \leq$

$1 \iff \max(T_c, T_s) \le 1/r \iff T_{req} \le 1/r$, the feasibility condition is

$$T_{req} \le \tau, \qquad \tau \triangleq \min\left(p,\ \frac{1}{r}\right). \tag{5.10}$$

The two objectives are two bounds on the same quantity, and whichever is tighter binds: a latency-sensitive deployment is limited by p, a throughput-oriented one by $1/r$.

Proposition 2. The allocation of Equation 5.4 satisfies both objectives whenever any allocation satisfies them.

Proof. By Equation 5.10 an allocation is feasible iff $T_{req}(k) \le \tau$. By Proposition 1 the chosen k minimizes T_{req}. Hence if T_{req} at the chosen k exceeds τ, no k achieves feasibility. □

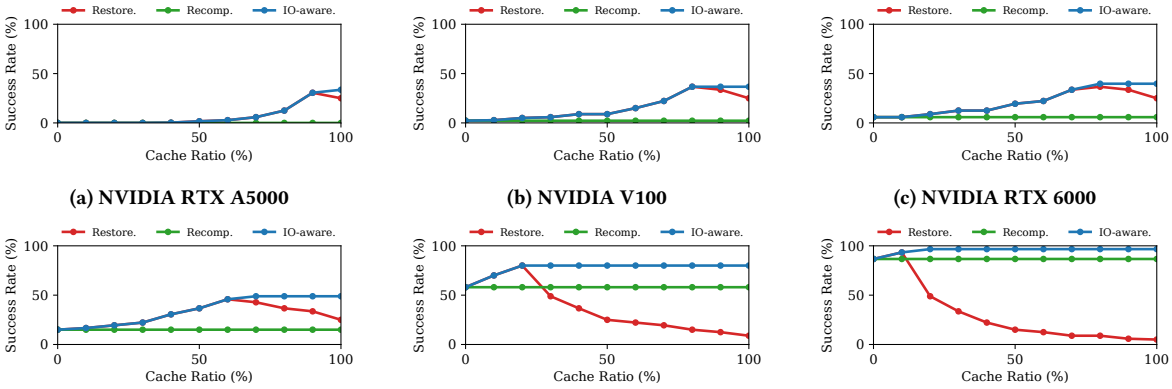
When no allocation is feasible, the request cannot meet both objectives under the current storage and compute capabilities, and additional resources, a relaxed objective, or a different placement strategy is required. The same allocation minimizes the maximum constraint violation, so a best-effort mode requires no separate computation.

Success rate across the configuration space

Equation 5.10 makes objective satisfaction a predicate that the simulator can evaluate for every configuration. We define the success rate as the fraction of evaluated configurations that satisfy both objectives, and sweep the 1,764 configurations of Table 4.2 against the full range of cache-hit ratios. A cache-hit ratio of c means that a fraction c of a request's blocks are available in the hierarchy and the remainder are misses, so $M = (1 - c)N$. Three allocations are compared: always-restore ($k = 0$), always-recompute ($k = N - M$), and the allocation of Equation 5.4.

A threshold metric is appropriate because an objective is itself a threshold: a configuration that misses its P95 target by 10% has missed it, and averaging that outcome with one that satisfies its target describes neither. Figure 5.2 reports the result across six GPUs, ordered from slowest to fastest. The I/O-aware allocation lies at or above both fixed mechanisms at every cache-hit ratio, which follows from Proposition 2, since both fixed mechanisms are particular values of k. The magnitude of the margin is the empirical result.

Slow GPUs are bounded by hardware. On the RTX A5000, V100, RTX 6000, and A100 all three allocations remain at low success rates: recomputation below 20% and restoration below 40%, because these GPUs lack the recomputation throughput to absorb work taken off the storage path. The I/O-aware allocation is highest in every case and its margin grows with the cache-hit ratio, but the ceiling is set by the accelerator. **Fast GPUs show the largest margin.** On the H200 the I/O-aware allocation holds a success rate near the top of the range across the entire sweep, while always-restore falls from roughly 88% at a 0% cache-hit ratio to a few percent at 100%. At the right-hand edge the gap exceeds an order of magnitude. Across all configurations the allocation yields a 2–10× improvement in success rate over the fixed mechanisms. **Restoration degrades as the cache improves.** Below a 25% cache-hit ratio, restoration outperforms recomputation by up to 25 percentage points, since little traffic reaches the storage path. As the cache-hit ratio rises, an always-restore allocation fetches more blocks, saturates the storage path, and fails an increasing fraction of configurations, approaching total failure at a 100% cache-hit ratio. A high cache-hit ratio is therefore both the condition under which the cache is most effective and the condition under which unconditional restoration performs worst. The transition occurs at lower cache-hit ratios on faster GPUs, which is why the margin in Figure 5.2 widens from panel (a) to panel (f).



(a) NVIDIA RTX A5000 **(b) NVIDIA V100** **(c) NVIDIA RTX 6000**

(d) NVIDIA A100 **(e) NVIDIA H100** **(f) NVIDIA H200**

Figure 5.2: Effectiveness of the I/O-aware allocation compared with always-restore and always-recompute across cache-ratio configurations, ordered from slower to faster GPUs. Success rate is the fraction of scenarios that satisfy both the throughput and tail-latency objectives; higher values are better.

Chapter 6

Related Work

Work on KV-cache management is extensive and growing quickly. This chapter organizes it by what each line of work does about the fundamental problem, that KV-cache data is too large for HBM and too expensive to recompute.

Reusing and sharing KV-cache data. The first response to the cost of prefill is to avoid repeating it. Prefix caching retains a completed request's blocks so that a later request sharing the prefix can skip recomputation, and a substantial body of work extends the idea beyond exact prefixes. CacheBlend [70] fuses cached knowledge for retrieval-augmented generation by selectively recomputing only the tokens whose cached vectors would be invalidated by a different surrounding context, and KVLink [68] pursues a similar goal for concatenated documents. Both are partial-recomputation schemes related to the one this report develops, though they recompute to preserve output quality rather than to manage I/O. KVShare [67] targets multi-tenant reuse, and SafeKV [11] studies the safety implications of sharing cache state across users. Content-delivery-style framings ask whether LLM serving needs a network for cache data at all [8], and modular frameworks make the reuse layer pluggable [51]. Others target specific structures: ChunkAttention exploits prefix-aware chunking within the kernel [72], multi-turn dialogue systems exploit the turn structure explicitly [26], and multi-agent systems reuse cache state across agent boundaries [71]. Surveys of efficient inference [35, 77] and of data management for LLMs [81] place these in context. These techniques create the demand this work addresses: every additional reuse opportunity is another block worth retaining, and retention is what drives data out of HBM and into the tiers where restoration becomes expensive.

Multi-tier KV-cache storage. A second line of work builds the storage hierarchy itself. LMCache [39] provides an enterprise-scale KV-cache layer spanning DRAM and disk, and the llm-d project offers a comparable open implementation [41]; both integrate with inference engines through connector interfaces [61]. Mooncake [52] states the trade in its subtitle, trading more storage for less computation, and builds a KVCache-centric serving architecture around it. FlexGen [55] introduced aggressive offloading for high-throughput single-GPU inference. IMPRESS [7] adds importance-informed placement to a multi-tier prefix store, MCAM [10] manages multi-tier KV-cache for distributed inference, and MemServe [23] provides context caching for disaggregated serving. Others address placement from the model side, offloading model weights to free HBM for KV-cache data [74], or use context importance to decide what to keep on a single commodity GPU [58]. ScoutAttention pre-computes on the CPU a layer ahead to hide offloading latency [76], and JENGA revisits memory management inside the engine [75]. Other work refines the paged-attention data path itself for long-context deployments [29]. This work operates above the tiering layer: it assumes such a system exists, exposes the block metadata a lookup produces, and reports its own achieved throughput; the policy then decides what to ask it for. Any improvement to the tiering layer raises Y and shifts the crossover, but does not remove it.

Making the data smaller. If restoration is bandwidth-bound, one remedy is to move fewer bytes. CacheGen [40] compresses and streams KV-cache data for fast serving, and HeteroCache [56] applies het-

erogeneous compression tuned to long-context behavior. Quantization reduces the footprint at the source: Oaken uses an online–offline hybrid scheme for serving [31], and a body of work studies the accuracy cost of quantized LLMs and their caches [36, 37]. Compression and quantization change the constant in Equation 2.2 and therefore both S_{block} and the effective Y, but they do not change the structure of the decision. Halving the KV precision halves the bytes to restore, but it also halves X in Equation 4.8, since the GPU then produces half as many cache bytes per unit of the same compute, so the ratio that decides the trade-off does not move in an obvious direction. Quantifying how these techniques move the crossover is left to future work.

Making the data path faster. A third approach addresses β_{link} directly. NVLink and its successors raise inter-device bandwidth by an order of magnitude over PCIe [34, 48], and characterizations of modern GPU interconnects quantify the resulting gain [24]. GPUDirect Storage [46] and GPUDirect RDMA [47] remove the host bounce buffer from the path entirely, and benchmarking efforts measure the result in practice [44]. RDMA-first object storage designs apply the same idea to the remote tier [82]. Section 2.6 describes why these matter: the $\min$ terms in Equation 4.4 mean that the interconnect, rather than the device, is frequently the binding constraint. Raising β_{link} raises Y and moves the crossover toward restoration, which changes the answer rather than removing the question.

Scheduling, disaggregation, and parallelism. At the application level, systems reduce restoration overhead by scheduling around it. Sarathi piggybacks decodes onto chunked prefills to keep the GPU busy [2]; Splitwise [49] and DistServe [80] disaggregate prefill from decode across machines, and assessments of disaggregation quantify when it pays [43]. KVDirect targets distributed disaggregated inference [6], Fast-dLLM applies parallel decoding to diffusion LLMs [65], and request scheduling across multiple engines is an active area [30, 39]. Niyama proposes QoS-driven scheduling that removes the silos between request classes [20], and queueing-theoretic analyses characterize the latency structure of inference serving [69]. Heterogeneous fleets add another dimension [22, 38], as does the resilience of the accelerators themselves [12]. These systems operate at a coarser granularity than the policy described here: they decide which engine or which phase runs a request, whereas the policy decides, within a single restore, how to split one request's blocks. The two compose: a scheduler that has placed a request on a particular GPU has thereby fixed the X the policy reads.

The restore-versus-recompute trade-off directly. A small number of systems address the restore-versus-recompute question itself, and they divide into static approaches, which fix the decision from an offline profile, and dynamic approaches, which revise it at runtime. The prevailing static form is a tunable threshold expressed in tokens or in bytes, selected by benchmarking a deployment and thereafter held fixed [39, 64, 66]; tier-selection policies that route a request to a fixed tier by request size belong to the same family. Their cost is the one quantified in Section 5.1: the threshold holds only for the configuration it was measured on, and Figure 5.1 reports that configuration changes move restoration time by two to three orders of magnitude. On the dynamic side, Jiang *et al.* propose I/O-aware *partial* KV-cache recomputation, overlapping recomputation of some tokens with the transfer of others to hide latency [27], and Jin *et al.* ask "compute or load KV cache? why not both?" and likewise pursue a hybrid [28]. Both split a request across the two paths, and both differ from the allocation of Chapter 5 in how the split is chosen: these systems determine it from profiling and tunable parameters calibrated to a deployment, whereas Equation 5.4 computes it in closed form from throughputs observed at decision time and validates the result against explicit objective parameters. Characterization studies describe the behavior of deployed systems without prescribing a policy: Wang *et al.* characterize KV-cache behavior at a large cloud provider and describe threshold-based tier selection [64]; Mamo *et al.* compare KV-cache management strategies head to head [42]; Xu *et al.* survey optimization strategies for scalable inference [66]; and Ren studies the I/O behavior of offloading models and caches to NVMe specifically [53]. Chapter 3 follows this tradition, and Chapter 4 extends it from measurement of a few systems to modeling of many. The difficulty of tuning multi-tier caches is itself a studied problem: Carver identifies which storage-system parameters matter enough to tune [5]; Estro *et al.*

report the difficulty of finding optimal multi-tier cache configurations [16] and use knee detection to guide simulations of them [15]; and adaptive multi-objective configuration has been applied to tiered storage more broadly [79]. This literature reports that offline configuration of multi-tier caches is difficult, which bears directly on attempts to fix the KV-cache decision offline, while performance-model-driven approaches to LLM inference specifically [50, 73] and benchmarking efforts across accelerators [1, 9] supply the modeling foundation this work builds on.

Chapter 7

Future Work

Chapter 5 established the I/O-aware allocation as a result derived from a performance model. The work that follows falls into two categories: making the allocation a deployable mechanism and evaluating it as such, and extending the model beyond the assumptions of Section 4.1. The immediate goal is to implement the allocation of Equation 5.4 as a KV-cache management mechanism inside a production-grade serving system, and to evaluate it against the simulator that produced it. Implementation requires integration at two interfaces. The inference engine exposes KV-cache operations through a connector [61], which is where a restore request is intercepted and where the residency classification of Section 5.2 is already available. A tiered cache layer such as LMCache [39] supplies the per-tier residency metadata and the achieved transfer rates that furnish Y; the engine supplies X. Because the allocation reads only quantities that both components already maintain, the mechanism requires no additional instrumentation, and the integration reduces to intercepting the restore path and issuing two sets of block requests instead of one. Evaluating the implementation against the simulator serves two purposes. The first is to test the performance model, by comparing predicted and measured restoration and recomputation times over configurations the simulator claims to cover. The second is to quantify what the model omits: the runtime cost of the decision in situ rather than in isolation, the noise in the observed X and Y and the averaging window over which they must be smoothed to be usable, and the interaction between a per-request allocation and the engine's own batching and scheduling decisions. Divergence between the two is itself informative, since it localizes which of the omissions in Section 4.1 matters most. The model evaluates configurations in isolation. A loaded system has restores queueing behind other restores and behind asynchronous store traffic, which reduces the effective Y, and recomputation contending with prefill and decode work from other requests, which reduces the effective X. Modeling the two paths as queues rather than as fixed-rate resources would allow the allocation to be evaluated as part of a scheduled population rather than per request, and would connect it to admission control and to per-tenant objectives, which this work does not capture. The allocation resolves a two-way choice between one compute path and one storage path. Real hierarchies contain more (*e.g.*, DRAM, local NVMe, and remote object storage), and a single request's blocks may span all of them. The natural generalization treats the hierarchy as a flow network: each tier contributes a source with capacity equal to the blocks it holds, each path into HBM contributes an edge with capacity equal to its effective bandwidth, shared links appear as edges whose capacity is jointly consumed by every tier behind them, and the GPU appears as a further source with capacity X. Minimizing completion time then becomes a maximum-flow problem [17, 18], solvable in polynomial time by Edmonds–Karp [14] and applied in comparable resource-routing settings [4, 33].

Chapter 8

Conclusion

Existing LLM inference cache systems offload KV-cache data aggressively, because retaining it is what makes long-context serving affordable, and then restore it unconditionally. The measurements and the model presented in this report show that unconditional restoration is not justified by the underlying I/O characteristics. Chapter 3 characterized the I/O behavior of an offloading KV-cache. Of the three cache operations, only restore lies on the critical path, and its cost is governed by the slowest tier a request's blocks occupy rather than by the volume of data, producing a piecewise cost curve with a step at each tier boundary. Restoration and recomputation are substitutes: both produce identical key and value vectors, and they consume disjoint resources. Measured on an H200 and an RTX A5000 under an identical model, workload, and storage device, the ordering between them reversed, which establishes it as a property of the accelerator and storage pairing rather than of the workload. Chapter 4 reduced the KV-cache storage stack to a performance model of per-tier capacity, per-tier and interconnect bandwidth, and GPU arithmetic rate, which allowed the ordering to be evaluated over a catalog of published hardware rather than over the two machines available for measurement. Chapter 5 reported three results from that model. First, the domain of solutions is large: across 1,764 hardware configurations, hardware selection alone moves restoration time by two to three orders of magnitude at a fixed context length, and the recomputation curves of nearly every GPU pass through the restoration distributions, so neither mechanism dominates. Second, because the two paths consume disjoint resources, a restore request can be divided between them; the division that equalizes their completion times assigns blocks in proportion to each path's throughput and makes the system behave as a single path of bandwidth $X + Y$. That division provably minimizes request completion time, costs 5.07 μs to compute, and requires no state beyond quantities a deployed system already maintains. Third, the tail-latency and throughput objectives reduce to a single bound $\tau = \min(p, 1/r)$ on the same quantity, so the division that minimizes completion time is also the one that satisfies both objectives whenever any division does. Evaluated across the configuration space, it raises the fraction of configurations meeting both objectives by 2–10$\times$ over always-restore and always-recompute. Chapter 5 also reported the behavior of the fixed mechanisms under a well-performing cache. As the cache-hit ratio rises, an always-restore system fetches more blocks, saturates the storage path, and fails its objectives in an increasing fraction of configurations, so a higher cache-hit ratio degrades tail latency under that mechanism. Treating a cached block as a candidate for restoration rather than as an obligation removes that degradation.

Chapter 9

Acknowledgments

We thank the anonymous HotStorage reviewers for their valuable feedback. We also thank Danny Harnik, Thomas Parnell, Alexander Joukov, and Tahsin Ahmed for their contributions. This work was made possible in part thanks to IBM support; a SUNY/IBM Alliance award; and NSF awards CNS-2106263, CNS-2106434, CNS-2524208, CNS-2214980, and OAC-2230078.

Bibliography

[1] Amey Agrawal, Nitin Kedia, Anmol Agarwal, Jayashree Mohan, Nipun Kwatra, Souvik Kundu, Ramachandran Ramjee, and Alexey Tumanov. On evaluating performance of llm inference serving systems. *arXiv preprint arXiv:2507.09019*, 2025.

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

[3] BIZON Tech. Gpu benchmarks. https://bizon-tech.com/gpu-benchmarks, 2026. Accessed: 2026-05-13.

[4] Merve Bulut and Evrencan Özcan. Optimization of electricity transmission by ford–fulkerson algorithm. *Sustainable Energy, Grids and Networks*, 28:100544, 2021.

[5] Zhen Cao, Geoff Kuenning, and Erez Zadok. Carver: Finding important parameters for storage system tuning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 43–57, 2020.

[6] Shiyang Chen, Rain Jiang, Dezhi Yu, Jinlai Xu, Mengyuan Chao, Fanlong Meng, Chenyu Jiang, Wei Xu, and Hang Liu. Kvdirect: Distributed disaggregated llm inference. *arXiv preprint arXiv:2501.14743*, 2024.

[7] Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. {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)*, pages 187–201, 2025.

[8] Yihua Cheng, Kuntai Du, Jiayi Yao, and Junchen Jiang. Do large language models need a content delivery network? *arXiv preprint arXiv:2409.13761*, 2024.

[9] Krishna Teja Chitty-Venkata, Siddhisanket Raskar, Bharat Kale, Farah Ferdaus, Aditya Tanikanti, Ken Raffenetti, Valerie Taylor, Murali Emani, and Venkatram Vishwanath. Llm-inference-bench: Inference benchmarking of large language models on ai accelerators. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1362–1379. IEEE, 2024.

[10] Kexin Chu, Zixu Shen, Sheng-Ru Cheng, Dawei Xiang, Ziqin Liu, and Wei Zhang. Mcam: Efficient llm inference with multi-tier kv cache management. In *2025 IEEE 45th International Conference on Distributed Computing Systems (ICDCS)*, pages 571–581. IEEE, 2025.

[11] Kexin Chu, Zixu Shen, Dawei Xiang, and Wei Zhang. Safekv: Safe kv-cache sharing in llm serving. In *Machine Learning for Computer Architecture and Systems 2025*, 2025.

[12] Shengkun Cui, Archit Patke, Hung Nguyen, Aditya Ranjan, Ziheng Chen, Phuong Cao, Gregory Bauer, Brett Bode, Catello Di Martino, Saurabh Jha, et al. Story of two gpus: Characterizing the resilience of hopper h100 and ampere a100 gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1145–1164, 2025.

[13] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[14] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM*, 19(2):248–264, 1972.

[15] Tyler Estro, Mário Antunes, Pranav Bhandari, Anshul Gandhi, Geoff Kuenning, Yifei Liu, Carl Waldspurger, Avani Wildani, and Erez Zadok. Guiding simulations of multi-tier storage caches using knee detection. In *2023 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2023.

[16] Tyler Estro, Pranav Bhandari, Avani Wildani, and Erez Zadok. Desperately seeking... optimal {Multi-Tier} cache configurations. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[17] Lester R. Ford Jr and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[18] Lester R Ford Jr and Delbert R Fulkerson. A simple algorithm for finding maximal network flows and an application to the hitchcock problem. *Canadian journal of Mathematics*, 9:210–218, 1957.

[19] FS. Understanding pcie versions, lanes, and slot types explained. https://www.fs.com/blog/understanding-pcie-versions-lanes-and-slot-types-explained-28862.html, 2025. Accessed: 2026-05-13.

[20] Kanishk Goel, Jayashree Mohan, Nipun Kwatra, Ravi Shreyas Anupindi, and Ramachandran Ramjee. Niyama: Breaking the silos of llm inference serving. *arXiv preprint arXiv:2503.22562*, 2025.

[21] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[22] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. Mélange: Cost efficient large language model serving by exploiting gpu heterogeneity. *arXiv preprint arXiv:2404.14527*, 2024.

[23] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*, 2024.

[24] Zhiyi Hu. Demystifying nccl: An in-depth analysis of gpu communication protocols and algorithms. In *2025 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2025.

[25] IBM. Accelerating ai inference with ibm storage scale. https://research.ibm.com/blog/accelerating-ai-inference-with-ibm-storage-scale, 2026. Accessed: 2026-06-09.

[26] Jinwoo Jeong and Jeongseob Ahn. Accelerating llm serving for multi-turn dialogues with efficient resource management. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 2, 2025.

[27] Chaoyi Jiang, Lei Gao, Hossein Entezari Zarch, and Murali Annavaram. Efficient llm inference with i/o-aware partial kv cache recomputation. *arXiv preprint arXiv:2411.17089*, 2024.

[28] Shuowei Jin, Xueshen Liu, Qingzhao Zhang, and Z. Morley Mao. Compute or load kv cache? why not both?, 2025.

[29] Thomas Joshi, Herman Saini, Neil Dhillon, Kaoutar El Maghraoui, et al. Paged attention meets flex-attention: Unlocking long-context efficiency in deployed inference. *arXiv preprint arXiv:2506.07311*, 2025.

[30] Rupinder Kaur, Arghavan Asad, Seham Al Abdul Wahid, and Farah Mohammadi. A survey of advancements in scheduling techniques for efficient deep learning computations on gpus. *Electronics*, 14(5):1048, 2025.

[31] Minsu Kim, Seongmin Hong, RyeoWook Ko, Soongyu Choi, Hunjong Lee, Junsoo Kim, Joo-Young Kim, and Jongse Park. Oaken: Fast and efficient llm serving with online-offline hybrid kv cache quantization. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 482–497, 2025.

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

[33] Myint Than Kyi and Lin Lin Naing. Application of ford-fulkerson algorithm to maximum flow in water distribution pipeline network. *International Journal of Scientific and Research Publications*, 8(12):306–310, 2018.

[34] Ang Li. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[35] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. Llm inference serving: Survey of recent advances and opportunities. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2024.

[36] Qingyuan Li, Ran Meng, Yiduo Li, Bo Zhang, Liang Li, Yifan Lu, Xiangxiang Chu, Yerui Sun, and Yuchen Xie. A speed odyssey for deployable quantization of llms. *arXiv preprint arXiv:2311.09550*, 2023.

[37] Shiyao Li, Xuefei Ning, Luning Wang, Tengxuan Liu, Xiangsheng Shi, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. Evaluating quantized large language models. *arXiv preprint arXiv:2402.18158*, 2024.

[38] Ruihan Lin, Zezhen Ding, Zean Han, and Jiheng Zhang. Large-scale llm inference with heterogeneous workloads: Prefill-decode contention and asymptotically optimal control. *arXiv preprint arXiv:2602.02987*, 2026.

[39] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, et al. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.

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

[41] LLM-d. Llm-d kv cache. https://github.com/llm-d/llm-d-kv-cache, 2026. Accessed: 2026-05-13.

[42] Oteo Mamo, Olga Kogiou, Hyunjin Yi, and Weikuan Yu. Comparative characterization of kv cache management strategies for llm inference. *arXiv preprint arXiv:2604.05012*, 2026.

[43] Tiyasa Mitra, Ritika Borkar, Nidhi Bhatia, Ramon Matas, Shivam Raj, Dheevatsa Mudigere, Ritchie Zhao, Maximilian Golub, Arpan Dutta, Sailaja Madduri, et al. Beyond the buzz: A pragmatic take on inference disaggregation. *arXiv preprint arXiv:2506.05508*, 2025.

[44] Isa Muradli. Insights into gpudirect data transfer through nixl benchmarking. In *2025 IEEE International Conference on eScience (eScience)*. IEEE, 2025.

[45] Quan Nguyen-Tri, Mukul Ranjan, and Zhiqiang Shen. Attention is all you need for kv cache in diffusion llms. *arXiv preprint arXiv:2510.14973*, 2025.

[46] NVIDIA. Nvidia gpudirect storage. https://docs.nvidia.com/cuda/archive/11.4.2/pdf/GDS.pdf, 2023. Accessed: 2026-05-13.

[47] NVIDIA. Nvidia gpudirect rdma. https://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf, 2026. Accessed: 2026-05-13.

[48] NVIDIA. Nvidia nvlink. https://www.nvidia.com/en-in/data-center/nvlink, 2026. Accessed: 2026-05-13.

[49] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[50] Rajeev Patwari, Ashish Sirasao, and Devleena Das. Forecasting llm inference performance via hardware-agnostic analytical modeling. *arXiv preprint arXiv:2508.00904*, 2025.

[51] Nearchos Potamitis, Lars Henning Klein, Chongyang Xu, Attreyee Mukherjee, Bardia Mohammadi, Niket Tandon, Laurent Bindschaedler, and Akhil Arora. Cache saver: A modular framework for efficient, affordable, and reproducible llm inference. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*, 2025.

[52] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX conference on file and storage technologies (FAST 25)*, pages 155–170, 2025.

[53] Zebin Ren. 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*, 2025.

[54] ShareAI Lab. Sharegpt-chinese-english-90k: A bilingual chinese-english human-machine dialogue dataset. https://huggingface.co/datasets/shareAI/ShareGPT-Chinese-English-90k, 2023. Hugging Face dataset repository (Accessed: 2026-06-09).

[55] Ying Sheng. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning (ICML*, 2023.

[56] Zhiyuan Shi, Qibo Qiu, Feng Xue, Zhonglin Jiang, Li Yu, Jian Jiang, Xiaofei He, and Wenxiao Wang. Heterocache: A dynamic retrieval approach to heterogeneous kv cache compression for long-context llm inference. *arXiv preprint arXiv:2601.13684*, 2026.

[57] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

[58] He Sun, Li Li, Mingjun Xiao, and Chengzhong Xu. Breaking the boundaries of long-context llm inference: Adaptive kv management on a single commodity gpu. *arXiv preprint arXiv:2506.20187*, 2025.

[59] UserBenchmark. Userbenchmark. https://www.userbenchmark.com, 2026. Accessed: 2026-05-13.

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

[61] VLLM. vllm documentation. https://docs.vllm.ai/en/latest, 2026. Accessed: 2026-05-13.

[62] James Vo. Sparseaccelerate: Efficient long-context inference for mid-range gpus. *arXiv preprint arXiv:2412.06198*, 2024.

[63] Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. Openchat: Advancing open-source language models with mixed-quality data. In *International Conference on Learning Representations*, volume 2024, pages 57021–57040, 2024.

[64] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. {KVCache} cache in the wild: Characterizing and optimizing {KVCache} cache at a large cloud provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 465–482, 2025.

[65] Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dllm: Training-free acceleration of diffusion llm by enabling kv cache and parallel decoding. *arXiv preprint arXiv:2505.22618*, 2025.

[66] Yichun Xu, Navjot K Khaira, and Tejinder Singh. Kv cache optimization strategies for scalable and efficient llm inference. *arXiv preprint arXiv:2603.20397*, 2026.

[67] Huan Yang, Renji Zhang, Mingzhe Huang, Weijun Wang, Yin Tang, Yuanchun Li, Yunxin Liu, and Deyu Zhang. Kvshare: An llm service system with efficient and effective multi-tenant kv cache reuse. *arXiv preprint arXiv:2503.16525*, 2025.

[68] Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. Kvlink: Accelerating large language models via efficient kv cache reuse. *Advances in Neural Information Processing Systems*, 38:133797–133824, 2026.

[69] Yuqing Yang, Lei Jiao, and Yuedong Xu. A queueing theoretic perspective on low-latency llm inference with variable token length. In *2024 22nd International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*, pages 273–280. IEEE, 2024.

[70] Jiayi Yao. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the twentieth European conference on computer systems*, 2025.

[71] Hancheng Ye, Zhengqi Gao, Mingyuan Ma, Qinsi Wang, Yuzhe Fu, Ming-Yu Chung, Yueqian Lin, Zhijian Liu, Jianyi Zhang, Danyang Zhuo, et al. Kvcomm: Online cross-context kv-cache communication for efficient llm-based multi-agent systems. *Advances in Neural Information Processing Systems*, 38:17882–17928, 2026.

[72] Lu Ye, Ze Tao, Yong Huang, and Yang Li. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11608–11620, 2024.

[73] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, et al. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.

[74] Fei Zeng. H2o: Heterogeneity-aware hierarchical orchestration for memory-efficient on-device llm inference. *IEEE Transactions on Mobile Computing*, 2025.

[75] Chen Zhang, Kuntai Du, Shu Liu, Woosuk Kwon, Xiangxi Mo, Yufeng Wang, Xiaoxuan Liu, Kaichao You, Zhuohan Li, Mingsheng Long, et al. Jenga: Effective memory management for serving llm with heterogeneity. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 446–461, 2025.

[76] Qiuyang Zhang, Kai Zhou, Ding Tang, Kai Lu, Cheng Li, Zhenyu Yang, Peng Xu, and Jiguang Wan. Scoutattention: Efficient kv cache offloading via layer-ahead cpu pre-computation for llm inference. *arXiv preprint arXiv:2603.27138*, 2026.

[77] Ranran Zhen, Juntao Li, Yixin Ji, Zhenlin Yang, Tong Liu, Qingrong Xia, Xinyu Duan, Zhefeng Wang, Baoxing Huai, and Min Zhang. Taming the titans: A survey of efficient llm inference serving. In *Proceedings of the 18th International Natural Language Generation Conference*, pages 522–541, 2025.

[78] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

[79] Xianzhe Zheng, Zhengheng Wang, Ruiyan Ma, Rui Wang, Xiyu Wang, Rui Chen, Peng Zhang, Sicheng Pan, Zhangheng Huang, Chenxin Wu, et al. Adaptive multi-objective tiered storage configuration for kv cache in llm service. *arXiv preprint arXiv:2603.08739*, 2026.

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

[81] Xuanhe Zhou, Junxuan He, Wei Zhou, Haodong Chen, Zirui Tang, Haoyu Zhao, Xin Tong, Guoliang Li, Youmin Chen, Jun Zhou, et al. A survey of llm$\times$ data. *arXiv preprint arXiv:2505.18458*, 2025.

[82] Yu Zhu. An rdma-first object storage system with smartnic offload. In *Proceedings of the SC'25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2025.