Research Square

Preprints are preliminary reports that have not undergone peer review. They should not be considered conclusive, used to inform clinical practice, or referenced by the media as validated information.

I/O for LLM Inference: A Survey of Storage and Memory Bottlenecks

Rajarshi Chowdhury

rajarshi.chowdhury@oracle.com

Oracle (United States)

Research Article

Keywords: Large Language Model Inference, Memory Bandwidth Bottleneck, Roofline Analysis, KV Cache Optimization, Model Quantization, Inference Serving Systems

Posted Date: March 19th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9036613/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

I/O for LLM Inference: A Survey of Storage and Memory Bottlenecks

Rajarshi Chowdhury[1]

[1]Oracle America Inc., Redwood Shores, 94064, CA, USA.

Contributing authors: rajarshi.chowdhury@oracle.com;

Abstract

Deploying Large Language Models at scale has shifted the dominant bottleneck from compute during training to memory and I/O during inference. As parameter counts reach hundreds of billions and context windows stretch past a million tokens, latency and throughput are limited not by arithmetic but by data movement across the memory hierarchy. This survey decomposes inference I/O into three flows—**model weight I/O**, **Key-Value (KV) cache I/O**, and **activation I/O**—and uses roofline analysis to map each optimization to the memory-hierarchy level it targets. We cover quantization, PagedAttention, FlashAttention, speculative decoding, KV cache compression, and offloading, alongside system-level orchestration (continuous batching, disaggregated prefill-decode, prefix caching) and hardware trends (HBM scaling, CXL, processing-in-memory, unified memory). A composability analysis reveals that stacking optimizations causes the dominant bottleneck to oscillate between weight and KV cache I/O. We close by identifying open problems in unbounded-context scaling, expert caching, edge deployment, and I/O-aware benchmarking.

Keywords: Large Language Model Inference, Memory Bandwidth Bottleneck, Roofline Analysis, KV Cache Optimization, Model Quantization, Inference Serving Systems

1 Introduction

Large Language Models have become infrastructure. They power text generation, code synthesis, scientific reasoning, and a growing range of multimodal applications [1–4]. Scaling laws say bigger is better [5], so architectures keep getting larger: GPT-4 [2], Llama 3 [4], Mixtral [6]. These models now serve hundreds of millions of users daily,

and at that scale, inference cost dwarfs training cost by a wide margin. The real efficiency challenge is not how to train these models but how to run them.

Running them turns out to be an I/O problem—and the situation is getting worse. GPU compute has grown roughly 18× from the V100 (∼125 TFLOP/s) [7] through the H100 (∼990 TFLOP/s) [8] to the B200 (∼2.25 PFLOP/s) [9]. HBM bandwidth over the same window went from 900 GB/s to 3.35 TB/s to 8 TB/s—about 9×. The ridge point $I^* = \text{FLOPS/BW}$, which marks the crossover between memory-bound and compute-bound territory, has climbed from ∼139 (V100) to ∼295 (H100); the B200's larger HBM3e bandwidth pulls the FP16 ridge point back to ∼281, though at newer precisions such as FP4 and FP8 the effective ridge point remains very high. Autoregressive decode sits at $I \approx 1$ (or $I \approx B$ with batching). So every current GPU generation remains deeply memory-bandwidth-starved at decode time. We keep getting faster at arithmetic—which is not the binding constraint for decode—while the factor that does bind, moving bytes, improves more slowly overall. Add long-context applications, multi-turn agents, and inference-time compute scaling (chain-of-thought, tree search) to the mix, and the case for looking at inference through an I/O lens gets hard to wave away.

To see why, consider how training and inference differ. Training large Transformers [10] is compute-bound: large batches drive up arithmetic intensity and keep the ALUs busy. Inference is another story. It splits into two phases with very different resource profiles. *Prefill* processes the input prompt in one shot—it looks a lot like training and is usually compute-limited. *Decode* generates tokens one at a time, loading the full model weights from HBM for each token while doing comparatively little math per byte [11, 12]. Decode is squarely memory-bandwidth-bound: the bottleneck is how fast data can move from HBM to the compute cores, not how many TFLOP/s the chip can sustain.

This distinction matters beyond performance. A 32-bit DRAM read costs roughly 640 pJ at 45 nm—about two orders of magnitude more than a floating-point multiply-accumulate (∼3.7 pJ) at the same node [13]. Recent measurements confirm that data movement, not arithmetic, dominates inference energy [14, 15]. As LLM-powered applications multiply, aggregate inference cost will far exceed training cost, making I/O efficiency a first-order concern for both economics and carbon footprint.

The community has attacked this problem from many directions. Quantization shrinks weight precision [16–19]. Pruning zeros out parameters [20, 21]. FlashAttention [22–24] restructures attention to avoid HBM round-trips. PagedAttention [25] borrows virtual-memory paging to curb KV cache fragmentation. Speculative decoding [26, 27] burns spare compute to extract multiple tokens per weight load. Serving frameworks—vLLM [25], Orca [28], SGLang [29], TensorRT-LLM [30]—handle batching and scheduling. On the hardware side, HBM evolution, CXL disaggregation [31], and processing-in-memory [32] all attack the same wall. These techniques look like they belong to different subfields, but they share a single objective that usually goes unstated: *fewer bytes moved per token generated.*

Yet existing surveys [33–36] organize techniques by algorithmic category—quantization here, KV cache there, attention in a third section—without connecting

them through the data-movement lens that actually explains *why* they help. Quantizing weights and compressing the KV cache end up in separate chapters, even though both are instances of the same idea (reduce bytes per operation) applied at different points in the data flow. That separation obscures patterns worth noticing. Speculative decoding and batching look nothing alike algorithmically, but they share a roofline mechanism: both raise arithmetic intensity so that fixed data movement is amortized over more useful work. And the binding constraint—whether weights, KV cache, or activations dominate—shifts dynamically with batch size, context length, and parallelism, which no static taxonomy can capture.

This survey tries to do something different. We are, to the best of our knowledge, the first to organize the full landscape of LLM inference optimizations—from quantization through hardware design—around a unified data-movement framework rather than by algorithmic family, using roofline analysis to show *why* each technique helps and *when* it stops helping. We decompose inference data movement into three flows—weight I/O ($\mathcal{W}$), KV cache I/O ($\mathcal{K}$), and activation I/O ($\mathcal{A}$)—and use roofline analysis [37] to pin down where each bottleneck sits on current hardware (Figure 1). We then map techniques from across the stack—quantization and sparsity at the algorithm level, continuous batching and disaggregated serving at the system level, HBM scaling, CXL, PIM, and custom accelerators [38, 39] at the hardware level—to the specific flow and hierarchy tier each one targets. A composability analysis (Section 8) traces what happens when these optimizations are stacked: the dominant bottleneck flips back and forth between $\mathcal{W}$ and $\mathcal{K}$ in ways that explain why the "best next optimization" is never the same twice. We flag open I/O problems—unbounded context, MoE expert caching [6, 40, 41], edge deployment [42, 43], and the lack of I/O-aware benchmarks—and release a companion toolkit[1] that computes I/O profiles, crossover batch sizes, and roofline plots for any model from its HuggingFace config [44].

Figure 1 gives an overview. A few gaps are immediately visible: CPU DRAM and SSD remain largely untouched for activation I/O, and there is almost no work on system-level techniques that directly manage SRAM utilization.

In the rest of the paper, Section 2 covers background on Transformer inference, the memory hierarchy, and the roofline model. Sections 3–5 examine the three I/O flows in turn. Section 6 discusses system-level orchestration (batching, disaggregation, inference-time compute scaling). Section 7 surveys hardware trends including UMA for edge inference. Section 8 presents the composability analysis. Section 9 identifies open challenges, and Section 10 concludes.

2 Background

A single autoregressive decode step through a Transformer layer touches three kinds of data: weight parameters, cached key-value states, and intermediate activations. Each scales differently with batch size and sequence length, and each has a different affinity for hardware resources. We set up the relevant formalism below, grounding it in the bandwidth and capacity numbers of today's GPU memory hierarchy.

[1] Available at https://doi.org/10.5281/zenodo.18780115. Includes GPU profiles for datacenter (V100 through B200), consumer (RTX 4090), and UMA (Apple M4 Max/Ultra) hardware.

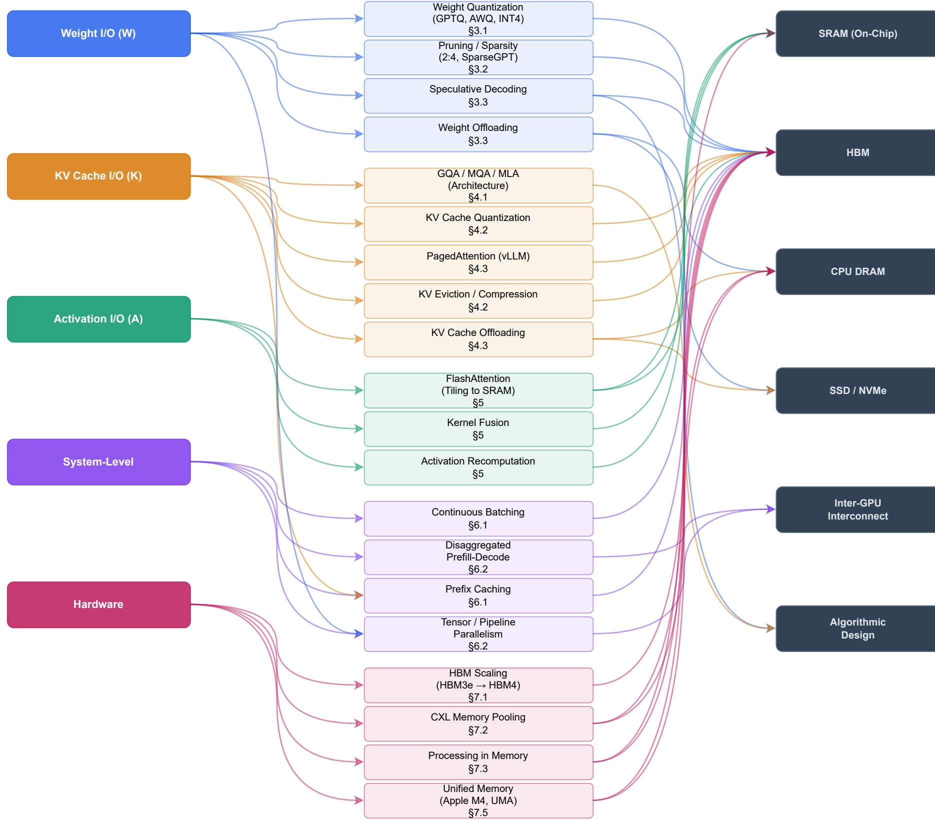

Fig. 1 Unified I/O optimization taxonomy: Each technique (center) is linked leftward to the data flow it reduces ($\mathcal{W}$, $\mathcal{K}$, $\mathcal{A}$, or system/hardware-level) and rightward to the memory-hierarchy tier it targets. Connections are colored by primary data flow; multi-colored dots mark cross-cutting techniques that span multiple flows (e.g., Prefix Caching targets both $\mathcal{K}$ and system-level orchestration). Sparsely connected tiers—CPU DRAM and SSD for activations, SRAM for system-level techniques—indicate underexplored research directions. Section numbers on each technique box provide direct cross-references into the survey.

2.1 Transformer Decoder Data Flow

Consider a standard decoder-only Transformer [10] with L layers, hidden dimension d_{model}, n_h attention heads of dimension $d_h = d_{\text{model}}/n_h$, and feed-forward intermediate dimension d_{ff}. Each layer loads four attention projection matrices $(\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V, \mathbf{W}_O)$ plus two or three FFN matrices (three for gated variants like SwiGLU [45]). The FFN accounts for roughly two-thirds of per-layer parameters and, correspondingly, two-thirds of weight I/O. At decode time the full KV cache from all preceding tokens must be read from HBM to compute attention; at prefill time the $N \times N$ attention score matrix per head must be materialized—the quadratic I/O cost that FlashAttention [22] was designed to eliminate.

The I/O distinction between the two phases boils down to the shape of the input tensor. Prefill processes N tokens at once, producing large matrix-matrix multiplies (GEMM) with high arithmetic intensity. Decode processes a single token per sequence, turning every operation into a matrix-vector multiply (GEMV) with almost no reuse of loaded weights. This shape change—GEMM to GEMV—is the root cause of the decode-phase bandwidth bottleneck and the main concern of this survey.

2.2 The Memory and Storage Hierarchy

LLM inference traverses a deep hierarchy spanning orders of magnitude in both bandwidth and capacity. Table 1 gives representative numbers for an NVIDIA H100 SXM server.

Table 1 Memory hierarchy characteristics for an NVIDIA H100 SXM GPU server [8]. Bandwidth figures are per-device peak; effective sustained bandwidth is typically 70–85% of peak.

Level	**Capacity**	**Bandwidth**	**Latency**
Register File	∼33 MB	—	<1 ns
L2 Cache	50 MB	∼12 TB/s	∼5 ns
SM Shared Mem.[2]	228 KB/SM	∼33 TB/s*	∼1 ns
HBM3 (on-package)	80 GB	3.35 TB/s	∼100 ns
CPU DDR5 (host)	512 GB–2 TB	300–400 GB/s	∼80 ns
PCIe Gen5 (GPU↔CPU)	—	64 GB/s (x16)	∼ μs
NVLink 4.0 (GPU↔GPU)	—	900 GB/s (bi-dir)	<1 μs
NVMe SSD	2–8 TB	7–14 GB/s	∼10 μs

Three numbers in this table shape everything that follows. First, the bandwidth cliff between HBM and the next tier down (CPU DRAM over PCIe) is roughly 50×. Any offloading to host memory is an order-of-magnitude hit unless carefully hidden behind compute. Second, HBM capacity—80 GB on the H100, 180 GB on the B200[3]—is what determines whether a model can be served without offloading or multi-GPU parallelism. Third, the on-chip memory hierarchy offers substantially higher bandwidth than HBM: the L2 cache provides ∼12 TB/s (3.6× HBM), while SM shared memory (SMEM)—the level FlashAttention tiles into—provides even higher per-SM bandwidth, albeit with much smaller per-SM capacity (≤228 KB).

[2]The H100 has 132 SMs, each with up to 228 KB of configurable shared memory (SMEM), for ∼29 MB in aggregate. FlashAttention tiles into per-SM SMEM, not the 50 MB L2 cache. The ∼33 TB/s figure is the aggregate bandwidth across all 132 SMs; each SM sees ∼256 GB/s. *Aggregate; not directly comparable to the single-bus L2 and HBM bandwidths.

[3]B200 specifications vary by SKU. The HGX B200 ships with 180 GB of HBM3e at 8 TB/s; the GB200 NVL72 rack variant lists 192 GB. We use 180 GB (shipping HGX configuration) throughout unless otherwise noted.

2.3 Arithmetic Intensity and the Roofline Model

The roofline model [37] gives a hardware-centric ceiling on attainable throughput as a function of a kernel's arithmetic intensity I:

$$I = \frac{\text{FLOPs}}{\text{Bytes accessed from memory}} \tag{1}$$

Attainable performance is then:

$$\text{Throughput} = \min(\pi,\ \beta \cdot I) \tag{2}$$

where π is peak compute (FLOP/s) and β is peak memory bandwidth (bytes/s). The ridge point $I^* = \pi/\beta$ marks the crossover: below it you are memory-bound, above it compute-bound.

For the H100 SXM at FP16/BF16 [8]: $\pi \approx 990\,\text{TFLOP/s}$ and $\beta \approx 3.35\,\text{TB/s}$, so $I^* \approx 295\,\text{FLOP/byte}$. Figure 2 plots the result.

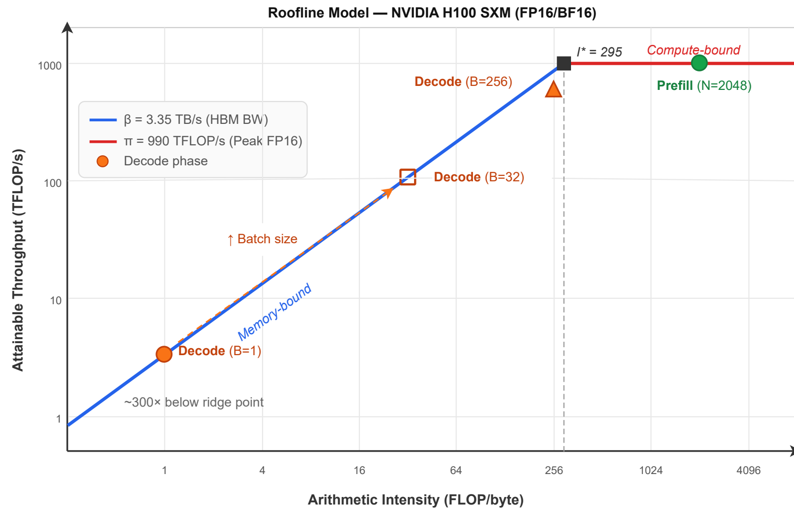

Fig. 2 Roofline model for an NVIDIA H100 SXM (FP16). The sloped region is bounded by HBM bandwidth ($\beta = 3.35\,\text{TB/s}$); the flat region by peak compute ($\pi = 990\,\text{TFLOP/s}$). Decode at $B = 1$ sits $\sim$300$\times$ below the ridge point I^*, deep in the memory-bound regime. Increasing the batch size moves decode rightward along the slope, but HBM capacity limits the maximum feasible B.

Prefill: processing N tokens through a weight matrix $\mathbf{W} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ takes $\sim 2BNd_{\text{model}}^2$ FLOPs while loading $\sim 2d_{\text{model}}^2$ bytes (FP16).

The arithmetic intensity is:

$$I_{\text{prefill}} \approx BN \quad \text{(FLOP/byte)} \tag{3}$$

At $B = 1, N = 2048$: $I_{\text{prefill}} \approx 2048 \gg I^*$—compute-bound, as expected. The weights are loaded once and reused across all N tokens.

Decode: generating one token loads the same $2d_{\text{model}}^2$ bytes of weights but does only $\sim 2Bd_{\text{model}}^2$ FLOPs. For $B = 1$:

$$I_{\text{decode}} \approx 1 \quad \text{(FLOP/byte)} \tag{4}$$

That is $\sim$300× below the ridge point. Decode is deeply memory-bound. Increasing B raises I_{decode} linearly—the basic mechanism behind batched serving.

2.4 Quantifying the Three I/O Flows

Every decode step moves data in three categories. We derive per-token byte counts for a model with L layers, hidden dimension d_{model}, FFN dimension d_{ff}, n_{kv} KV heads, head dimension d_h, weight precision b_w bits, batch size B, and current sequence length s.

Weight I/O ($\mathcal{W}$). Each layer has attention projections ($4 \cdot d_{\text{model}}^2$ parameters for Q, K, V, O under MHA) and FFN weights ($2 \cdot d_{\text{model}} \cdot d_{\text{ff}}$ for a standard FFN, $3 \cdot d_{\text{model}} \cdot d_{\text{ff}}$ for gated variants). Total bytes loaded per forward pass:

$$\mathcal{W} = L \cdot \left(4d_{\text{model}}^2 + \alpha \cdot d_{\text{model}} \cdot d_{\text{ff}}\right) \cdot \frac{b_w}{8} \tag{5}$$

where $\alpha = 2$ (standard) or $\alpha = 3$ (gated). The key property of $\mathcal{W}$ is that it does not depend on batch size: the same weights are loaded for $B = 1$ and $B = 256$. Batching amortizes weight I/O.

KV cache I/O ($\mathcal{K}$). Each layer reads the cached keys and values for all s prior tokens across all B sequences:

$$\mathcal{K} = L \cdot 2 \cdot B \cdot s \cdot n_{\text{kv}} \cdot d_h \cdot \frac{b_{\text{kv}}}{8} \tag{6}$$

where b_{kv} is the KV precision and the factor of 2 covers keys and values. Unlike $\mathcal{W}$, this grows linearly with both B and s. For Llama-3 70B at 128K context with FP16 KV cache, $\mathcal{K}$ exceeds 40 GB per decode step—larger than the $\sim$140 GB model weight load once amortized over even a modest batch.

Activation I/O ($\mathcal{A}$). Intermediate activations (the $N \times N$ attention matrix during prefill, FFN intermediates) are written to and read from HBM between kernel launches:

$$\mathcal{A}_{\text{prefill}} = L \cdot \mathcal{O}\big(B \cdot n_h \cdot N^2 + B \cdot N \cdot d_{\text{ff}}\big) \cdot \frac{b_a}{8} \tag{7}$$

The $n_h \cdot N^2$ term comes from materializing the attention scores; FlashAttention kills this term by keeping it in per-SM shared memory (SMEM). During decode, activation I/O is small compared to $\mathcal{W}$ and $\mathcal{K}$ because $N = 1$.

Which flow dominates depends on the operating point:

$$\text{Dominant flow} = \begin{cases} \mathcal{A} & \text{prefill, long context, no FlashAttention} \\ \mathcal{W} & \text{decode, small batch, short context} \\ \mathcal{K} & \text{decode, large batch or long context} \end{cases} \tag{8}$$

Sections 3–5 each tackle one of these flows.

2.5 Performance Metrics

We use the standard serving-level metrics. *Time-to-First-Token (TTFT)* is the latency from receiving a request to emitting the first output token, dominated by the prefill phase.

Time-Per-Output-Token (TPOT) captures inter-token decode latency. When the system is memory-bound:

$$\text{TPOT} \approx \frac{\mathcal{W} + \mathcal{K}(s)}{\beta_{\text{eff}}} \tag{9}$$

where β_{eff} is sustained memory bandwidth. TPOT degrades as the sequence grows ($\mathcal{K}$ increases) and improves with quantization (both $\mathcal{W}$ and $\mathcal{K}$ shrink).

Throughput is aggregate tokens per second across all concurrent requests:

$$\text{Throughput} = \frac{B}{\text{TPOT}(B, s)} \tag{10}$$

Throughput improves with batch size because $\mathcal{W}$ is shared—at $B = 256$ each sequence pays only 1/256th of the weight-loading cost. But there is a catch: increasing B also inflates $\mathcal{K}$, which scales as $\mathcal{O}(B \cdot s)$ and must stay in HBM. On an 80 GB H100 serving Llama-3 70B (about 140 GB in FP16 across two GPUs, leaving $\sim$10 GB per GPU for KV cache), each sequence at 4K context in FP16 needs roughly 1.3 GB of KV cache. That limits the batch to about $B \approx 7$ per GPU before memory runs out. This capacity wall—not compute—is what caps throughput, and it is the reason KV compression (Section 4) and memory expansion (Section 7) matter as much as bandwidth optimization. At higher batch sizes where capacity is not yet exhausted, a second ceiling appears: I_{decode} rises with B and eventually approaches I^*, transitioning from memory-bound to compute-bound, with diminishing returns from further batching.

2.6 LLMs as I/O Workloads

To put the I/O equations in concrete terms, Table 2 compiles profiles for representative open-weight models across three regimes: small/edge (1–9B), large dense, and Mixture-of-Experts.

A few patterns in the table are worth calling out. The shift from MHA to GQA (compare GPT-3 or Llama-2 7B with the Llama-3 family) cuts per-token KV cache

Table 2 I/O profiles of representative open-weight LLMs. Weight footprints are computed as total parameters × bytes per parameter. KV cache sizes assume FP16 precision and $B = 1$; all sizes in decimal GB (10^9 bytes). For GQA models, KV bytes/token/layer $= 2 \cdot n_{\text{kv}} \cdot d_h \cdot 2$. For MLA models (DeepSeek), the cached latent has dimension $d_c + d_R = 576$, stored in FP16. Min H100s assumes FP16 weights, 80 GB HBM per device, weights only (excluding KV cache headroom).

							Weights		KV Cache (FP16, B=1)			
Model	**Type**	**Attn**	L	n_h / n_{kv}	**Active** (B)	**Vocab** (K)	**FP16** (GB)	**INT4** (GB)	**Per tok/L** (bytes)	**@ 4K** (GB)	**@ 128K** (GB)	**Min H100s**
Small / Edge models												
Llama-3.2 1B [46]	Dense	GQA	16	32 / 8	1.2	128	2.5	0.6	2,048	0.1	4.3	1
Llama-3.2 3B [46]	Dense	GQA	28	24 / 8	3.2	128	6.4	1.6	4,096	0.5	15	1
Phi-3 Mini 3.8B [47]	Dense	MHA	32	32 / 32	3.8	32	7.6	1.9	12,288	1.6	52	1
Gemma-2 9B [48]	Dense	GQA	42	16 / 8	9.2	256	18	4.6	8,192	1.4	—§	1
Dense models												
GPT-3 175B [1]	Dense	MHA	96	96 / 96	175	50	350	88	49,152	19.3	N/A	5
Llama-2 7B [3]	Dense	MHA	32	32 / 32	6.7	32	13	3.4	16,384	2.1	N/A	1
Llama-2 70B [3]	Dense	GQA	80	64 / 8	70	32	140	35	4,096	1.3	N/A	2
Llama-3 8B [4]	Dense	GQA	32	32 / 8	8.0	128	16	4.0	4,096	0.5	17	1
Llama-3 70B [4]	Dense	GQA	80	64 / 8	70	128	141	35	4,096	1.3	43	2
Llama-3 405B [4]	Dense	GQA	126	128 / 8	405	128	810	203	4,096	2.1	68	11
Mistral 7B [49]	Dense	GQA	32	32 / 8	7.3	32	15	3.7	4,096	0.5	—†	1
Qwen-2.5 72B [50]	Dense	GQA	80	64 / 8	72	152	145	36	4,096	1.3	43	2
Gemma-2 27B [48]	Dense	GQA	46	32 / 16	27	256	54	14	8,192	1.5	—§	1
Falcon 180B [51]	Dense	GQA	80	232 / 8	180	65	360	90	2,048	0.7	N/A	5
Mixture-of-Experts models (open-weight)												
Mixtral 8×7B [6]	$\text{MoE}^{8/2}$	GQA	32	32 / 8	13	32	93	23	4,096	0.5	N/A	2
Mixtral 8×22B [6]	$\text{MoE}^{8/2}$	GQA	56	48 / 8	39	32	282	71	4,096	0.9	N/A	4
DeepSeek-V2 [52]	$\text{MoE}^{160/6}$	MLA	60	128 / d_c=512	21	102	472	118	1,152‡	0.3	9.1	6
DeepSeek-V3 [53]	$\text{MoE}^{N/8}$	MLA	61	128 / d_c=512	37	129	1,342	336	1,152‡	0.3	9.2	17
DeepSeek-R1 [54]	$\text{MoE}^{N/8}$	MLA	61	128 / d_c=512	37	129	1,342	336	1,152‡	0.3	9.2	17

† Mistral 7B uses sliding-window attention (window size 4,096), bounding effective KV cache regardless of context length.

‡ MLA models cache a compressed latent of dimension $d_c + d_R = 512 + 64 = 576$ values per token per layer, stored in FP16 (1,152 bytes). This represents a ∼28× reduction versus standard MHA with the same head configuration.

§ Gemma-2 models have a native context window of 8,192 tokens with alternating global/sliding-window attention (window size 4,096). The @ 4K values assume all tokens fall within the sliding window and thus reflect the same KV footprint as full global attention at that length; at longer contexts, sliding-window layers would cap their KV cache at 4,096 tokens while global layers continue to grow.

MoE superscripts denote total experts / active experts per token. DeepSeek-V3/R1 uses 1 shared + 256 routed experts (denoted N). "N/A" indicates the model's native context window does not extend to 128K tokens.

I/O by roughly an order of magnitude; GQA is now effectively mandatory for long-context serving. Interestingly, small edge models like Phi-3 Mini still use MHA with 32 KV heads, giving them 12 KB of KV cache per token per layer—3× more than the much larger Llama-3 70B with GQA. Attention design dominates KV I/O regardless of model scale.

MLA (DeepSeek-V2/V3/R1) pushes further, compressing the cache to a 576-dimensional latent per layer and enabling 128K-context inference with single-digit GB of KV cache even at 671B parameters. MoE models decouple total weight capacity from per-token weight I/O: Mixtral 8×7B activates only ∼13B of its 47B parameters per token, but all experts must be stored somewhere accessible, making total HBM capacity the constraint rather than per-token bandwidth. Vocabulary size—easy to

Table 3 Weight quantization methods compared by precision, calibration strategy, and I/O reduction relative to FP16. Perplexity deltas are representative values for Llama-2 70B on WikiText-2.

Method	Strategy	Bits	$\mathcal{W}$ Red.	ΔPPL
LLM.int8() [18]	Mixed-prec. outlier	8	2×	<0.1
SmoothQuant [19]	Activation smooth.	W8A8	2×	<0.1
GPTQ [16]	Layer-wise OBQ	4	4×	∼0.3
AWQ [17]	Salient-wt. scaling	4	4×	∼0.2
GGUF [43]	Block-wise k-quant	2–6	3–8×	varies
QuIP# [55]	Incoh. + lattice	2	8×	∼0.8
AQLM [56]	Additive codebooks	2	8×	∼0.7
FP8 [57]	Native HW format	8	2×	<0.05

overlook—also matters: Gemma-2's 256K vocabulary adds ∼1.5 GB at FP16 versus Llama-2's 32K vocabulary. And at long contexts, KV cache can rival or surpass the weight footprint: Llama-3 405B at 128K context needs 68 GB of KV cache per sequence in FP16, scaling linearly with batch size. The companion toolkit [44] can compute these profiles for any model from its HuggingFace config.

3 Model Weight I/O

Weight I/O dominates decode at small batch sizes and short contexts. For a 70B model in FP16, one decode step loads about 140 GB from HBM—the whole model—to produce a single token per sequence. On an H100 (3.35 TB/s peak), that transfer alone sets a floor of ∼42 ms per token at $B = 1$, no matter how many TFLOP/s the chip has. Every technique in this section targets the same term in Equation (5): reducing b_w or the effective parameter count.

3.1 Weight Quantization

Quantization reduces b_w and, with it, $\mathcal{W}$ in direct proportion. Going from FP16 ($b_w = 16$) to INT4 ($b_w = 4$) for a 70B model shrinks the weight footprint from 140 GB to 35 GB—a 4× improvement in decode throughput on bandwidth-limited hardware. In practice, INT4 models on consumer GPUs come close to this theoretical bound [16, 17]. Table 3 summarizes the major methods; they differ mainly in how they choose the quantization grid to limit accuracy loss.

A few observations from an I/O standpoint. The benefit is deterministic: halving b_w halves $\mathcal{W}$, regardless of GPU. The accuracy–I/O tradeoff is nonlinear: FP16→INT8 is essentially free in perplexity while halving I/O, but INT4→INT2 buys another 2× at the cost of increasingly exotic calibration (incoherence processing, additive codebooks). And because $\mathcal{W}$ does not depend on B, quantization matters less as batch size grows and $\mathcal{K}$ takes over.

There is a practical distinction between weight-only methods (GPTQ, AWQ, GGUF) and weight-activation methods (SmoothQuant, FP8). Weight-only methods dequantize to FP16 before arithmetic, spending ALU cycles that would otherwise be idle during bandwidth-limited decode—so the dequantization is effectively free when $I \approx 1$. Weight-activation methods quantize both operands, enabling native

low-precision tensor-core arithmetic that cuts I/O and boosts effective compute simultaneously, pulling I^* down.

3.2 Weight Sparsity

Sparsity reduces $\mathcal{W}$ by removing parameters rather than compressing them. Whether it actually saves I/O depends on structure. Unstructured sparsity—zeroing individual weights at arbitrary positions, achievable at 50–60% via SparseGPT [20] or Wanda [21]—gives little I/O benefit on today's GPUs because irregular access patterns underutilize HBM bandwidth, and index overhead eats into the storage savings.

Semi-structured (N:M) sparsity is a different story. NVIDIA's Ampere and Hopper architectures natively support 2:4 sparsity—exactly 2 zeros in every group of 4 weights—through dedicated sparse tensor cores [58], delivering a real 2× reduction in $\mathcal{W}$ with a regular, hardware-friendly layout. Combining 2:4 sparsity with INT8 quantization yields 4× I/O reduction—matching INT4—while potentially preserving better accuracy, which makes it an appealing option in the composability analysis of Section 8.

3.3 Storage-Tier Offloading

When a model does not fit in HBM, weights must be streamed from further down the hierarchy, and the offloading interconnect's bandwidth becomes the new bottleneck on $\mathcal{W}$.

FlexGen [59] treats offloading as a linear programming problem, computing an optimal placement schedule for weights, activations, and KV cache across GPU, CPU, and SSD. By overlapping transfers with compute on previously loaded layers, FlexGen achieves decent throughput (though not low latency) for large-batch inference on a single GPU. The insight is that large enough batches make per-layer compute take longer than the transfer, hiding the overhead.

llama.cpp [43] takes a different tack: aggressive quantization (GGUF formats, often 4-bit) combined with CPU-side inference, running large models entirely in system RAM and optionally offloading some layers to a GPU. This sidesteps the PCIe bottleneck for offloaded layers by keeping them on the CPU—trading per-core throughput for access to much larger memory pools (512 GB+ of DDR5 vs. 80 GB of HBM).

LLM in a Flash [42] pushes offloading all the way to NVMe SSDs. It exploits sparsity in FFN activations—neurons with near-zero activation do not need their weights loaded—to cut the number of weight rows fetched from flash per token. By coalescing the remaining reads into large sequential transfers that maximize SSD throughput, it achieves interactive speeds for models up to 2× the available DRAM.

Table 4 puts numbers on the bandwidth penalty at each tier. The takeaway: offloading is a last resort that must be paired with quantization and compute-transfer overlap. The exception is Unified Memory Architectures, where the offloading cliff vanishes entirely (Section 7.5).

Table 4 Effective decode latency for a 70B model (FP16, 140 GB weights) at $B = 1$ across memory tiers. Latency computed as $\mathcal{W}/\beta_{\text{sustained}}$ assuming 80% bandwidth efficiency.

Source Tier	Sustained BW	Decode Latency
HBM3 (H100)	2.68 TB/s	52 ms
Host DRAM over PCIe Gen5 x16	51 GB/s	2.7 s
NVMe Gen4 (SSD)	5.6 GB/s	25 s
HBM3 + INT4 quant	2.68 TB/s	13 ms

3.4 Mixture-of-Experts

MoE architectures reduce per-token weight I/O by activating only a subset of parameters. In Mixtral 8×7B [6], each token is routed to 2 of 8 experts, so only 2/8 of the FFN weights are loaded per token despite the model containing ∼47B total parameters. Per-token $\mathcal{W}$ is closer to a 13B dense model, while model capacity approaches something much larger.

From an I/O perspective, MoE trades bandwidth pressure for a capacity and data-movement scheduling problem. All expert weights must be accessible—ideally in HBM. For models like Switch Transformer [40] and DeepSeek-MoE [41], total parameters can reach hundreds of billions or trillions, far exceeding single-device HBM. Experts end up distributed across devices (NVLink for intra-node, InfiniBand for cross-node) or offloaded to CPU/SSD—in each case at bandwidth well below local HBM. Expert placement and routing become first-class serving decisions.

The problem is compounded by unpredictable activation patterns. PowerInfer [60] profiles expert frequencies offline and pins "hot" experts in GPU memory while keeping "cold" ones in CPU DRAM. MoE-Infinity [61] and related systems [62] build expert caches with LRU-like eviction, treating the problem as a classical cache management scenario where GPU HBM is the cache and CPU/SSD is the backing store.

The upshot: MoE gives you lower per-token $\mathcal{W}$ but higher total capacity requirements and harder orchestration. As these models continue to scale, expert placement, caching, and prefetching will become central I/O optimization problems (Section 9).

3.5 Speculative Decoding

Speculative decoding [26, 27] comes at weight I/O from a different direction: instead of making $\mathcal{W}$ smaller, it extracts more useful tokens per weight load, pushing up the arithmetic intensity of decode.

The idea is straightforward. A small draft model (say 1B parameters, or a set of lightweight prediction heads) generates γ candidate tokens autoregressively. The full target model verifies all γ candidates in one forward pass—which, processing γ tokens in parallel, runs as a GEMM with $I \approx \gamma$ rather than a GEMV with $I \approx 1$. Candidates matching the target distribution are accepted; the first rejected one is resampled. If the draft achieves per-token acceptance rate $\alpha = 0.8$ with $\gamma = 5$, roughly 4 tokens are accepted per verification pass—about 4× less per-token weight

I/O for the target model. Medusa [63] eliminates the separate draft model by training parallel prediction heads on the target model. EAGLE [64] improves acceptance rates by predicting feature-level representations rather than raw tokens.

On the roofline (Figure 2), speculative decoding moves the decode operating point rightward—from $I \approx 1$ toward $I \approx \gamma$—along the memory-bound slope. Quantization changes the slope itself (fewer bytes per op); speculative decoding leaves $\mathcal{W}$ unchanged but gets more done per byte.

4 KV Cache I/O

The KV cache is the second major decode-phase I/O flow, and in many ways the more troublesome one. Weights are static—they sit in HBM and get read every step, but at least they do not grow. The KV cache accumulates key and value projections for every token generated so far, and attention must read the entire thing at each step. As Table 2 shows, the KV cache can go from negligible at short contexts to rivaling or exceeding the weight footprint at long contexts—and the problem compounds with large batches, since each sequence keeps its own cache.

4.1 Growth Dynamics and the Crossover Point

For a dense GQA model, the KV cache per sequence grows as $\mathcal{K}_{\text{seq}} = 2 \cdot N \cdot L \cdot n_{\text{kv}} \cdot d_h \cdot (b_{\text{kv}}/8)$, where N is the current length. With batch size B the total is $B \cdot \mathcal{K}_{\text{seq}}$. Figure 3 shows this growth for several models. The crossover where KV I/O equals weight I/O during decode ($B \cdot \mathcal{K}_{\text{seq}} \approx \mathcal{W}$) occurs at $B \approx 108$ for Llama-3 70B at 4K context in FP16—a number that drops sharply with weight quantization, as we show below.

It is worth walking through the derivation. Llama-3 70B has 80 layers with 8 GQA KV heads of dimension 128. At FP16, each token adds $2 \times 8 \times 128 \times 2 = 4{,}096$ bytes per layer, or $80 \times 4{,}096 = 327{,}680$ bytes ($\sim$0.3 MB) per token across the full model. At 4K context, each sequence carries $4{,}096 \times 0.3 \approx 1.3$ GB of KV cache. The model itself is $\sim$141 GB in FP16. The total per-step read is $\mathcal{W} + B \times 1.3$, and $\mathcal{K}$ overtakes $\mathcal{W}$ when $B \times 1.3 > 141$, i.e., $B > 108$. But applying INT4 weight quantization drops $\mathcal{W}$ to $\sim$35 GB, pushing the crossover down to $B > 27$. And with INT4 KV quantization the cache drops to $\sim$0.33 GB per sequence, so the crossover lifts back to $B > 106$. This seesaw—each compression on one flow pushes the bottleneck to the other—is exactly the oscillation we quantify in the composability analysis (Section 8).

The practical consequence is that operators cannot think about weight compression and KV compression independently. A deployment that aggressively quantizes weights to INT4 but leaves KV at FP16 will almost certainly be KV-bound at any batch size above $\sim$27. Conversely, compressing KV without touching weights merely delays the weight bottleneck.

4.2 Architectural KV Reduction

The highest-leverage way to shrink $\mathcal{K}$ is to change the attention architecture itself—reducing how much gets cached per token per layer. The progression from MHA to

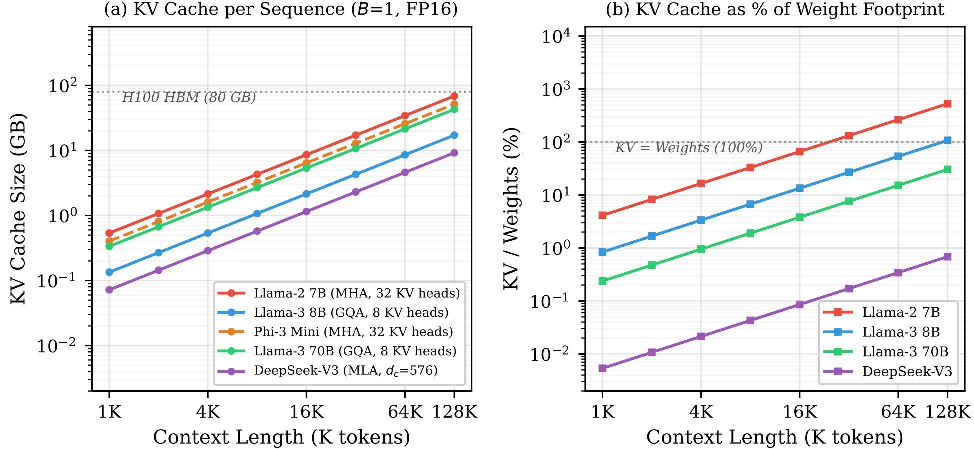

Fig. 3 (a) KV cache size per sequence as a function of context length for five representative architectures. MHA models (Llama-2 7B, Phi-3) grow 4–12× faster than GQA models (Llama-3), while MLA (DeepSeek-V3) achieves the slowest growth. (b) KV cache as a percentage of the model weight footprint: for MHA models, KV cache exceeds model weights at moderate context lengths.

MQA to GQA to MLA represents a decade of learning about this tradeoff, and the I/O numbers tell the story more clearly than perplexity tables.

Multi-Query Attention (MQA) [65] collapses all KV heads to a single head ($n_{\mathrm{kv}} = 1$), cutting KV cache by a factor of n_h relative to MHA. For a model with 64 query heads, that is a 64× reduction—dramatic, but it comes at a cost. Forcing one KV head to serve all query heads limits the model's ability to attend to different aspects of context simultaneously. In practice, MQA works well for small models and short contexts but starts to hurt quality at scale, which is why it has largely given way to GQA.

Grouped-Query Attention (GQA) [66], used in Llama-2 70B, Llama-3, Mistral, Qwen-2.5, and Gemma-2, offers a middle ground: n_{kv} KV heads are shared among groups of query heads. A typical ratio of $n_h/n_{\mathrm{kv}} = 4$ to 8 yields 4–8× KV reduction over MHA with minimal quality loss, and GQA has become the de facto standard at 7B+. (The bandwidth savings assumes KV head broadcasting happens in SRAM rather than HBM, which is the common case when using FlashAttention-style kernels that fuse the broadcast into tiled computation.) To put this concretely: Llama-2 7B (MHA, 32 KV heads) allocates 16,384 bytes per token per layer, while Llama-3 8B (GQA, 8 KV heads) allocates 4,096—a 4× difference between models of similar size. At 32K context, that gap translates to ∼17 GB versus ∼4.3 GB of KV cache per sequence. The choice of attention architecture is, in many deployments, a larger lever on serving cost than any post-hoc compression.

Multi-head Latent Attention (MLA) [52], introduced in DeepSeek-V2, takes a fundamentally different approach. Rather than reducing the number of KV heads, MLA projects keys and values into a low-rank latent of dimension $d_c \ll n_{\mathrm{kv}} \cdot d_h$ before caching. Instead of storing full K and V vectors, MLA caches a compressed latent of

dimension $d_c + d_R = 576$ per token per layer (1,152 bytes in FP16). As Figure 3 shows, this is a $\sim 28\times$ reduction versus MHA—enabling DeepSeek-V3's 671B model to hold just 9.2 GB of KV cache at 128K context, versus 43 GB for Llama-3 70B. The catch is that MLA requires an extra up-projection during attention to recover the full-rank keys and values from the latent, adding compute. But since decode is bandwidth-bound, the extra FLOPs are essentially free—the compute units would be idle anyway waiting for memory. MLA is the clearest example of a design that deliberately trades (abundant) compute for (scarce) bandwidth.

These architectural choices determine whether a given deployment is weight-bound or KV-bound, and thus which optimizations from Section 3 vs. this section deliver the most value. A model using MLA may never become KV-bound at realistic batch sizes, meaning weight compression is always the priority. A model using MHA at long contexts is almost certainly KV-bound, making KV compression or eviction essential regardless of what has been done to the weights.

4.3 KV Cache Compression

Independent of architecture, the KV cache can be compressed after the fact via quantization or eviction. These two strategies have very different failure modes, and understanding when each breaks down is important for deployment.

KV quantization applies the same precision-reduction ideas from Section 3.1 to cached keys and values. KIVI [67] and KVQuant [68] show that per-channel quantization to INT4 or even INT2 is feasible with careful calibration, halving or quartering $\mathcal{K}$ with little perplexity impact. Combined with GQA, this can reduce KV cache I/O by 16–32$\times$ relative to full-precision MHA. The nice property of KV quantization is that it is *lossless* in the capacity sense: every token remains in the cache, just at lower precision. The failure mode is subtle—quantization errors accumulate across the sequence, with keys typically more sensitive than values because they participate in the softmax attention distribution. Errors in keys can redirect attention mass to the wrong tokens, while errors in values merely add noise to the output. This asymmetry is why KIVI uses different bit widths for keys and values.

Eviction and sparsification take a different route: instead of representing all tokens at lower precision, they drop tokens entirely. H_2O [69] identifies "heavy hitter" tokens—those accumulating the highest attention scores—and keeps only these plus a window of recent tokens. StreamingLLM [70] notices that the first few tokens ("attention sinks") receive disproportionate attention mass regardless of content, and proposes retaining a fixed set of sinks plus a rolling window. These methods bound $\mathcal{K}$ to a constant regardless of true context length, enabling theoretically unbounded streaming—at the cost of losing access to evicted context.

The failure mode of eviction is more dramatic than quantization: it is not degradation but *amnesia*. A token that was unimportant at step 1,000 may become critical at step 10,000 when the conversation circles back—but if it has been evicted, the model cannot recover it. Needle-in-a-haystack evaluations expose this directly: StreamingLLM and H_2O perform well on standard perplexity benchmarks (which mostly test local coherence) but fail on tasks requiring retrieval of specific information buried deep in context. This gap between perplexity-based evaluation and real

retrieval fidelity is one of the reasons why eviction methods have seen limited adoption in production, despite their attractive I/O properties. The problem is fundamental: eviction policies must predict future attention patterns from past ones, and attention patterns in long conversations are not stationary.

A middle ground is emerging: keep a compressed representation of evicted tokens rather than discarding them outright. CacheGen [71] explores this via learned codebooks. Retrieval-augmented approaches offer another path, replacing the evicted portion of the cache with an on-demand re-encoding mechanism that fetches relevant context from external storage. Neither approach has yet achieved the simplicity of "just quantize everything," which remains the most practical option for production deployments.

4.4 Paged KV Management

In serving scenarios with dynamic batches and variable-length sequences, managing KV memory is itself a surprisingly hard problem—one that has more in common with operating systems than with machine learning. Naively pre-allocating a contiguous buffer for the maximum context per request wastes enormous HBM: a Llama-3 70B server supporting 128K context would reserve 43 GB per sequence even if most requests use only 1K tokens, crippling batch size. In a realistic workload where 90% of requests use under 2K tokens, over 95% of pre-allocated KV memory sits empty.

PagedAttention [25], from vLLM, borrows virtual-memory paging. The KV cache is split into fixed-size blocks (e.g., 16 tokens each), allocated on demand from a shared pool. A page table maps logical token positions to physical block locations, which need not be contiguous. This kills internal and external fragmentation, pushing HBM utilization from the 20–40% range under naive allocation to over 95%. The I/O payoff is indirect but important: fitting more sequences into the same HBM increases batch size, which amortizes $\mathcal{W}$ and raises throughput. For a concrete example, consider an 80 GB H100 serving Llama-3 70B in INT4 ($\sim$35 GB of weights), leaving $\sim$45 GB for KV cache. Under naive allocation with 128K max context in FP16 ($\sim$43 GB per sequence), the server can hold exactly one sequence. With PagedAttention, if actual usage averages 2K tokens ($\sim$0.7 GB per sequence), the same 45 GB supports $\sim$64 concurrent sequences—a 64$\times$ throughput improvement from better memory management alone, with zero algorithmic changes to the model.

PagedAttention also enables copy-on-write semantics for shared prefixes: if multiple sequences share a system prompt, their KV pages for those tokens point to the same physical blocks, further reducing memory pressure. This interacts well with prefix caching (Section 4.5).

4.5 KV Cache Offloading and Prefix Caching

When KV cache exceeds HBM even after compression, it can be offloaded to CPU DRAM or SSD. InfiniGen [72] prefetches KV entries from CPU to GPU by predicting which tokens will get high attention scores in the next layer. CacheGen [71] compresses KV cache with learned codebooks before transfer to reduce PCIe volume.

Prefix caching (or prompt caching) avoids redundant work by recognizing that many requests share a common prefix—a system prompt, for instance. If the KV cache for that prefix is already computed, new requests skip prefill for those tokens and just load the cached entries. SGLang [29] implements this via a radix tree that efficiently identifies shared prefixes. The savings are both compute and I/O: a cached KV copy is an HBM-to-HBM transfer at full bandwidth, much cheaper than recomputing from weights.

5 Attention and Activation I/O

Activation I/O—the intermediate tensors produced and consumed during a forward pass—is usually small relative to $\mathcal{W}$ and $\mathcal{K}$ during decode at moderate batch sizes. But two situations make it a first-class concern: the internal data movement within attention kernels, and inter-device activation transfers in model-parallel setups.

5.1 IO-Aware Attention: FlashAttention

Standard attention materializes the full $N \times N$ attention matrix in HBM before applying softmax and multiplying with V. For sequence length N and head dimension d_h, that means $O(N^2)$ HBM reads and writes—a massive I/O cost that dominates prefill wall-clock time. FlashAttention [22] rewrites attention as a tiled algorithm that tiles into per-SM shared memory (SMEM; up to 228 KB per SM on the H100, ∼29 MB in aggregate across 132 SMs), computing attention block by block and accumulating the output without ever materializing the full matrix in HBM.

The savings are substantial. Standard attention does $\Theta(N^2 d_h + N^2)$ HBM accesses. FlashAttention brings this down to $\Theta(N^2 d_h^2/M)$ where M is the per-SM shared memory capacity, eliminating the N^2 matrix traffic entirely. On an H100 with 228 KB SMEM per SM and $d_h = 128$, this translates to 5–7× wall-clock speedup during prefill [23]. FlashAttention-2 refines the tiling for better GPU parallelism; FlashAttention-3 [24] exploits Hopper-specific features (TMA, warp-group specialization). FlashDecoding [73] extends the approach to decode, where the problem becomes a matrix-vector product against the KV cache.

5.2 Activation Recomputation

Activation recomputation—discarding intermediate activations and recomputing them when needed—is standard practice during training (gradient checkpointing) [74]. It applies to inference in two situations: long-context prefill where HBM is tight (recompute rather than store intermediates), and speculative decoding verification (discard draft-phase activations and recompute during the verification pass). The tradeoff is compute for memory: recomputation costs extra FLOPs but avoids HBM traffic for large activation tensors, a good deal when bandwidth is the bottleneck.

5.3 Inter-Device Activation Transfers

Distributing a model across GPUs means shipping activations between devices at every parallel boundary. In tensor parallelism (TP) [75], each layer is split across devices,

requiring an all-reduce after every GEMM—$2 \times B \times d \times b_a$ bytes per layer per device. In pipeline parallelism (PP), activations flow between stages at $B \times d \times b_a$ per micro-batch per stage boundary.

The question is how much of the decode budget these transfers eat. On NVLink (900 GB/s), an all-reduce of a 4 KB vector ($B = 1$, $d = 4096$, FP16) takes ~10 ns—nothing. But with 8-way TP, $B = 256$, $d = 8192$, the volume grows to ~4 MB per layer (~4.5 µs on NVLink), which starts to matter. Cross-node transfers over InfiniBand (50 GB/s) are ~18× slower, forcing pipeline-parallel schedules with large micro-batches to amortize the latency. MoE expert parallelism (Section 3.4) introduces all-to-all communication where each device sends tokens to whichever device hosts the selected expert—a highly irregular pattern with poor locality.

5.4 Ring Attention and Sequence Parallelism

When contexts exceed single-device HBM capacity, the KV cache and attention must be distributed along the sequence dimension. Ring Attention [76] partitions the sequence across a ring of devices and overlaps block-wise FlashAttention-style computation with point-to-point KV block transfers around the ring. Each device computes attention over its local KV block, sends it to the next device, and receives the next block simultaneously. If per-block compute exceeds transfer time, communication is fully hidden.

Ring Attention converts a capacity problem into a communication scheduling problem. Total inter-device movement per attention layer is $2 \cdot B \cdot N_{\text{local}} \cdot n_{\text{kv}} \cdot d_h \cdot (b_{\text{kv}}/8) \cdot (P-1)$ bytes, where P is the ring size and $N_{\text{local}} = N/P$. For Llama-3 70B at 128K context with $B = 12$ across 8 devices (16K tokens each), this is ~5.6 GB per layer on NVLink—feasible to overlap at long contexts but increasingly exposed at shorter sequences where per-block compute is insufficient.

Striped Attention [77] and other sequence-parallel variants interleave token assignments for better load balancing. These compose with KV compression: Ring Attention + INT4 KV quantization cuts inter-device transfer volume by 4×.

6 System-Level I/O Optimization

Good algorithms are necessary but not sufficient—extracting their I/O benefits in a production serving environment requires careful orchestration of batching, scheduling, memory management, and caching across concurrent requests.

6.1 Serving Engines

Modern serving engines integrate the techniques surveyed above. Rather than cataloging feature lists, Table 5 maps each engine's I/O-relevant capabilities to the three data flows. No single engine wins everywhere: vLLM [25] leads in $\mathcal{K}$ management (PagedAttention, KV swapping); TensorRT-LLM [30] leads in $\mathcal{W}$ throughput (FP8 GEMMs, fused MoE kernels); SGLang [29] avoids redundant $\mathcal{K}$ via RadixAttention prefix caching; DeepSpeed-FastGen [78] minimizes $\mathcal{A}$ overhead with tensor parallelism and splitfuse scheduling. The right engine depends on which flow dominates your workload.

Table 5 Serving engine I/O capabilities. ✓ = native support; ◦ = partial or plugin-based; — = not supported. "Prefix cache" reuses KV for shared prompts; "KV offload" swaps KV to CPU on preemption.

	$\mathcal{W}$ (Weight I/O)			$\mathcal{K}$ (KV Cache)			$\mathcal{A}$ (Activation)	
Engine	Cont. batch	FP8/ INT4	Fused MoE	Paged Attn	Prefix cache	KV offload	Tensor par.	Disagg. P/D
vLLM	✓	✓	✓	✓	✓	✓	✓	◦
TensorRT-LLM	✓	✓	✓	✓	✓	—	✓	✓
SGLang	✓	✓	✓	✓	✓	—	✓	—
DeepSpeed-FastGen	✓	◦	—	—	—	—	✓	✓

6.2 Continuous Batching and Scheduling

The core tension in LLM serving: increasing B amortizes $\mathcal{W}$ (good for throughput) but grows KV memory and per-request latency (bad for tail latency). Continuous batching [28] partly resolves this by letting sequences enter and leave the batch at each decode step, rather than waiting for the longest sequence to finish. Short requests are no longer held hostage by long ones, and the batch stays full.

Scheduling policies refine the tradeoff. FCFS minimizes queueing latency but can leave the batch under-filled. Shortest-remaining-time-first minimizes average latency but needs output-length prediction. Preemptive scheduling (vLLM's KV cache swapping) lets the system handle load spikes by temporarily offloading a sequence's KV cache to CPU. For a 70B GQA model at 4K context, that is $\sim$1.3 GB per sequence—PCIe Gen5 moves it in $\sim$20 ms, acceptable for non-latency-critical preemptions.

6.3 Disaggregated Prefill and Decode

Prefill and decode have opposite I/O profiles—prefill is compute-bound, decode is memory-bound—yet they typically share hardware, forcing a compromise. Splitwise [79] and DistServe [80] propose disaggregated architectures that route prefill to high-compute GPU pools and decode to high-bandwidth GPU pools. The cost is an inter-node transfer of the prefill-generated KV cache (1.3 GB for Llama-3 70B at 4K context) before decode can begin.

Mooncake [81] extends this to a KV-cache-centric design with a shared distributed KV store, allowing any decode node to access any sequence's state—enabling flexible load balancing and cutting redundant prefill.

6.4 Chunked Prefill

Chunked prefill blurs the line between phases. Instead of processing the full prompt in one compute-heavy pass, it splits the prompt into chunks and interleaves them with decode steps from other requests. This prevents long prompts from stalling latency-sensitive decodes, and by controlling chunk size relative to the HBM budget, it enables fine-grained control over the $\mathcal{W}$–$\mathcal{K}$ tradeoff: smaller chunks reduce peak activation I/O at the cost of extra weight loads, while larger chunks amortize $\mathcal{W}$ but spike

activation memory. Sarathi-Serve and DeepSpeed-FastGen [78] implement variants; it is becoming standard in production engines.

6.5 Inference-Time Compute Scaling

A newer paradigm—inference-time compute scaling—changes the I/O picture in ways that are still being understood. Models like DeepSeek-R1 [54] and OpenAI's o1 produce extended chain-of-thought (CoT) reasoning, often generating thousands of tokens of intermediate reasoning before a short final answer. Tree-search methods multiply generated tokens further.

The I/O consequences differ qualitatively from standard generation. The KV cache scales with CoT length: a query producing 8K reasoning tokens before a 200-token answer needs $\sim 40\times$ more KV I/O than the answer alone. For Llama-3 70B, 8K CoT tokens add $\sim$2.6 GB to the KV cache per sequence, directly cutting achievable batch size. And because each reasoning token depends on all preceding ones, TPOT degrades cumulatively as described in Equation (9)—each step reads a larger cache. Tree-search further complicates things: branching continuations share a prefix (making prefix caching valuable) but require managing branching KV trees, which is harder than linear PagedAttention.

These workloads push the system toward $\mathcal{K}$-dominated regimes even at small batch sizes, amplifying the value of KV compression, MLA-style reduction, and KV-centric disaggregated serving [81].

7 Hardware Trends and Implications

The analysis so far has been grounded in today's hardware. But the memory hierarchy is not standing still, and some of the trends are not encouraging for inference.

7.1 HBM Bandwidth Scaling: The Widening Gap

Figure 4 shows the trend clearly. Across four NVIDIA GPU generations [7–9, 82], the ridge point climbed from $\sim$139 FLOP/byte on V100 to $\sim$295 on H100. The B200, benefiting from a larger HBM3e bandwidth jump (8 TB/s), sees its FP16 ridge point settle at $\sim$281—roughly flat with the H100. However, at newer precisions (FP8 at 4.5 PFLOP/s, FP4 at 9 PFLOP/s dense [9]), the effective ridge soars to $\sim$562 and $\sim$1,125 respectively, widening the gap for any kernel that can exploit low-precision arithmetic. Since decode sits at $I \approx 1$ to $I \approx B$, each generation still leaves the vast majority of performance on the table at decode time. The hardware roadmap is optimizing for training workloads, and inference is along for the ride.

HBM is evolving—HBM2e (2.0 TB/s on A100 80 GB SXM [82]) to HBM3 (3.35 TB/s on H100 [8]) to HBM3e (8 TB/s on B200, 180 GB [9]).[4] HBM4, expected from SK Hynix and Samsung, should reach 10–13 TB/s with improved stacking. These bandwidth gains ($\sim$1.5–2.4$\times$ per generation) are meaningful but still lag the

[4] Figure 4 uses the A100 40 GB SXM (1.555 TB/s, $I^* \approx 201$); the 80 GB SXM at 2.0 TB/s yields $I^* \approx 156$. We reference the 80 GB SKU in the text for consistency with the H100 80 GB used elsewhere.

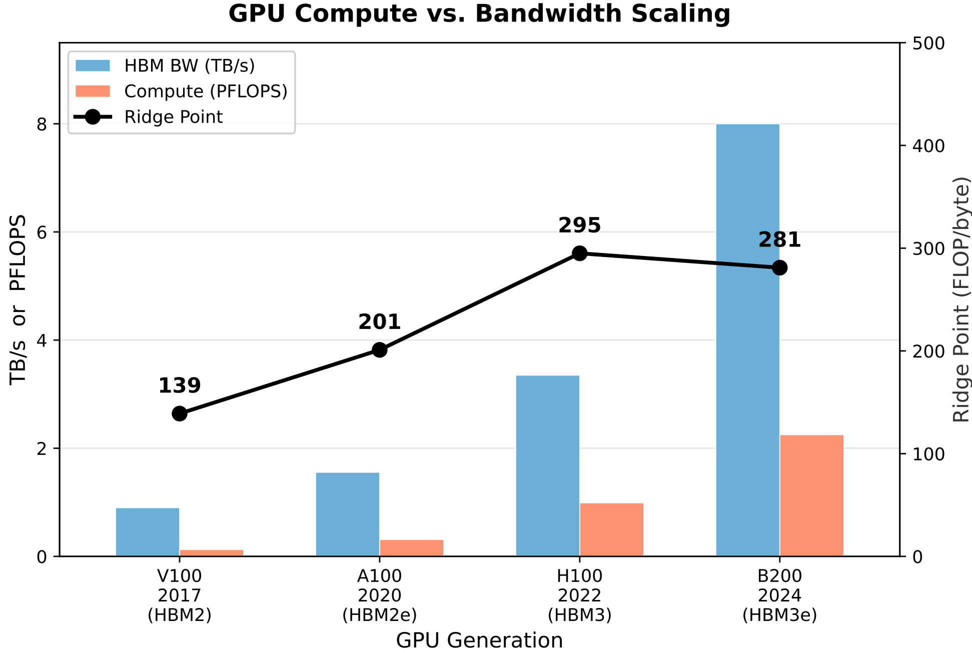

Fig. 4 GPU compute throughput and HBM bandwidth across four NVIDIA generations [7–9, 82]. While both grow, compute has historically scaled faster than bandwidth, pushing the ridge point I^* higher from V100 through H100. The B200's HBM3e upgrade provides a larger-than-usual bandwidth jump, temporarily flattening the FP16 ridge point; however, at newer precisions (FP4, FP8) the effective ridge point continues to rise, making decode *more* memory-bound over time.

pace of compute scaling at newer precisions, preserving the structural imbalance for low-arithmetic-intensity kernels like autoregressive decode.

7.2 CXL and Memory Pooling

Compute Express Link (CXL) [83] is a cache-coherent interconnect on PCIe Gen5/6 lanes that enables memory expansion and pooling. CXL 1.1 allows attaching Type 3 memory expanders at ∼32–64 GB/s—10–20× below HBM but 2–4× above NVMe. CXL 2.0 adds dynamic memory pooling across hosts. CXL 3.0 introduces shared memory and fabric switching.

For LLM inference, CXL slots in between HBM and CPU DRAM: a natural home for MoE expert weights, overflow KV cache, or infrequently accessed model layers. The latency penalty (∼200–400 ns vs. ∼100 ns for HBM) is significant for random access but manageable for streaming with software prefetch. Memory pooling is especially attractive for multi-tenant serving: KV caches from different requests could live in a shared pool and be assigned dynamically to whichever GPU handles the request—a hardware version of disaggregated serving (Section 6.3).

7.3 Processing-in-Memory and Near-Memory Compute

Processing-in-Memory (PIM) attacks the bandwidth wall by computing where the data lives, eliminating the need to move weights across the memory bus. Samsung's HBM-PIM [84] integrates SIMD elements into each HBM bank for in-situ matrix-vector operations. For decode—a sequence of GEMVs—this is a near-ideal match: weights stay put, and only the small activation vectors traverse the chip link.

The challenges are real, though. In-memory compute elements are typically low-precision (INT8 or below) and inflexible, ill-suited for layernorm, softmax, or rotary embeddings. And attention requires reading the KV cache, which is not co-located with model weights. Hybrid designs—PIM for the weight-heavy FFN layers, conventional GPU cores for attention—seem like the most promising path forward.

7.4 Custom Accelerators

Several startups have built hardware explicitly to break through the I/O wall. Groq's LPU [85] replaces off-chip HBM with large on-chip SRAM (230 MB per chip at ∼80 TB/s internal bandwidth), delivering extremely high, deterministic bandwidth from SRAM to compute. Distributing a 70B model's weights across hundreds of LPUs—each contributing its 230 MB—eliminates the HBM bottleneck entirely. Independent benchmarks report ∼280 tokens/s on Llama-3 70B in standard mode and over 1,600 tokens/s with speculative decoding [86], roughly 3–5× faster than optimized single-H100 deployments at comparable batch sizes. The price is SRAM density: you need a lot of chips (∼576 for a 70B model in mixed precision).

Cerebras's WSE-3 [39] goes wafer-scale: a single chip with 900,000 AI cores, 44 GB of on-chip SRAM, and 21 PB/s of on-wafer aggregate bandwidth. Models that fit in 44 GB (up to ∼40B parameters with INT8) see near-zero data movement latency.

Both designs validate the same insight: inference performance is bottlenecked by data movement, and the most effective hardware solutions restructure the memory hierarchy to cut that movement. The question is whether the generality trade-off is acceptable outside of pure inference workloads.

7.5 Unified Memory Architectures and Edge Inference

Everything above assumes a discrete-GPU hierarchy: weights in HBM, overflow to CPU over PCIe. Unified Memory Architectures (UMA) break this model. On Apple's M-series SoCs [87], Qualcomm's Snapdragon X Elite, and AMD's Ryzen AI, CPU, GPU, and NPU share a single physical memory pool. There is no "offloading" because there is no device boundary. Every byte is equidistant from every compute unit.

No more offloading cliff.

A 7B FP16 model that exceeds 24 GB VRAM on an RTX 4090 must split layers across PCIe, dropping from ∼1 TB/s (GDDR6X) to ∼51 GB/s—a 20× penalty. On a UMA system with 128 GB of unified LPDDR5X, the model just sits in the shared pool at uniform bandwidth. The M4 Max provides 546 GB/s accessible to both CPU and 40-core GPU, with up to 128 GB of capacity—enough for a 70B model in INT4 (∼35 GB) with ample KV headroom.

Table 6 Extended offloading comparison including UMA systems. Latency computed as $\mathcal{W}/\beta_{\text{sustained}}$ for a 7B model in INT4 (∼3.5 GB of weights) at $B = 1$, with 80% bandwidth efficiency (sustained = 0.8×peak). The UMA advantage is most pronounced for small/edge models where the entire working set fits in unified DRAM. Peak bandwidths: M4 Max 546 GB/s [87], M3 Ultra 819 GB/s [88].

Source / System	**Sust. BW**	**7B INT4**	**70B INT4**
HBM3 (H100)	2,680 GB/s	1.3 ms	13 ms
UMA LPDDR5X (M4 Max)	437 GB/s	8.0 ms	80 ms
UMA LPDDR5X (M3 Ultra)	655 GB/s	5.3 ms	53 ms
GDDR6X (RTX 4090)	806 GB/s	4.3 ms	—*
PCIe 5 offload (CPU→GPU)	51 GB/s	69 ms	686 ms
NVMe Gen4 (SSD)	5.6 GB/s	625 ms	6.3 s

* 70B INT4 (∼35 GB) exceeds RTX 4090's 24 GB VRAM; layers must be offloaded via PCIe, dropping effective bandwidth to 51 GB/s for the overflow portion.

I/O equations under UMA.

The tiered-bandwidth assumption ($\beta_{\text{HBM}} \gg \beta_{\text{PCIe}} \gg \beta_{\text{SSD}}$) collapses to two tiers—unified DRAM and on-chip caches—with a single bandwidth β_{UMA}:

$$\text{TPOT}_{\text{UMA}} = \frac{\mathcal{W} + \mathcal{K}(s, B)}{\beta_{\text{UMA}}} \tag{11}$$

This is the same as Equation (9) but with two differences. First, β_{UMA} is lower than HBM (∼400–550 GB/s for LPDDR5X vs. ∼3.35 TB/s for HBM3) but *higher* than an offloaded system (∼51 GB/s over PCIe). Second, capacity is generous: the M4 Ultra offers 192 GB—exceeding the B200's 180 GB HBM—at a fraction of the cost and power. Table 6 extends the offloading comparison.

Impact on the three flows.

UMA reshapes each flow differently. Weight I/O ($\mathcal{W}$) benefits the most: on discrete systems, exceeding VRAM means a bandwidth cliff, but on UMA the penalty is smooth—β_{UMA} applies uniformly up to full DRAM capacity, making UMA natural for edge deployment of 7–13B models in INT4 (2–7 GB). For KV cache I/O ($\mathcal{K}$), UMA allows larger KV budgets than discrete edge GPUs. An M4 Max (128 GB) running Llama-3 8B in INT4 (∼4 GB weights) retains ∼124 GB for KV cache—enough for roughly 960K tokens of FP16 context at B=1 (far beyond the model's trained window), versus ∼160K tokens on an RTX 4090 with ∼20 GB free VRAM. At a more realistic B=32, the per-sequence budget is still ∼30K tokens on the M4 Max versus ∼5K on the RTX 4090. Activation I/O ($\mathcal{A}$) also benefits: UMA eliminates CPU-GPU activation shuttling during hybrid offloaded inference (as in llama.cpp layer splitting), since shared physical memory means activations produced by CPU-hosted layers are immediately visible to GPU-hosted layers without any transfer.

The edge roofline.

UMA systems have much lower ridge points. The M4 Max GPU delivers ∼27 TFLOP/s (FP16) at 546 GB/s peak, giving $I^* \approx 49$—6× lower than the H100's $I^* \approx 295$ (both computed at peak bandwidth for consistency). Paradoxically, this *helps* LLM decode: the operating point $I \approx B$ is closer to the ridge, so the hardware utilization gap is less severe. At $B = 8$, the M4 Max achieves ∼16% of peak; the same batch on H100 reaches only ∼2.7%. Lower absolute performance, but better matched to the decode workload's arithmetic intensity.

Software.

Frameworks like llama.cpp [43] (Metal, Vulkan), MLX [89], and Qualcomm AI Engine Direct exploit UMA by allocating weights, KV cache, and activations in a single shared allocation. MLX is designed around the UMA assumption, using lazy evaluation and unified-memory tensors to avoid all host-device copies. The optimization target on UMA differs from discrete GPUs: instead of minimizing PCIe transfers, the goal is saturating β_{UMA} with weight and KV reads.

8 Composability Analysis: Stacking Optimizations

The preceding sections treated each I/O flow in isolation, which is useful for understanding mechanisms but misleading for practice. Real deployments stack multiple optimizations at once, and the natural question is: do the benefits compound, or do they run into each other? We answer this with a waterfall analysis using Llama-3 70B on an H100 as the reference platform.

8.1 I/O Waterfall: Cumulative Reduction

Table 7 traces cumulative I/O reduction as techniques are layered onto a Llama-3 70B baseline ($B = 32$, $s = 4096$, single H100), applied in order of typical deployment priority. All values can be reproduced using the companion toolkit [44].

The most striking pattern in the table is how the dominant bottleneck keeps flipping. We start $\mathcal{W}$-dominated. INT4 weights shift dominance to $\mathcal{K}$. INT4 KV cache shifts it back to $\mathcal{W}$. 2:4 sparsity keeps it at $\mathcal{W}$ with less margin. Speculative decoding flips it back to $\mathcal{K}$. This oscillation means the best next optimization always depends on which flow currently dominates—a moving target that static categorizations cannot track.

The composition is also multiplicative *across* flows but hits diminishing returns *within* each. INT4 weights give a genuine 4× on $\mathcal{W}$; adding 2:4 sparsity gives another 2× (for 8× cumulative). But INT2 weights on top of 2:4 sparsity would push accuracy below acceptable thresholds. Similarly, INT4 KV quantization yields 4× on $\mathcal{K}$, but the next step—eviction—trades context fidelity rather than providing lossless compression.

Adding everything up, the full stack gives about 12× total I/O reduction, bringing per-step data movement from 183 GB down to ∼15 GB and estimated TPOT from 68 ms to 5.6 ms at $B = 32$. This is close to the practical limit with current methods. But

Table 7 I/O waterfall for Llama-3 70B on H100 ($B = 32$, $s = 4096$). Each row adds one optimization to the row above. $\mathcal{W}$, $\mathcal{K}$, and $\mathcal{A}$ denote weight, KV cache, and activation I/O per decode step. Dominant flow indicates the binding bottleneck. TPOT estimated as $(\mathcal{W} + \mathcal{K})/\beta_{\text{eff}}$ with $\beta_{\text{eff}} = 2.68$ TB/s.

Configuration	$\mathcal{W}$ (GB)	$\mathcal{K}$ (GB)	**Total** (GB)	**Dom.** Flow	**Est. TPOT** (ms)
FP16 baseline	141	42	183	$\mathcal{W}$	68
+ INT4 weights (AWQ)	35	42	77	$\mathcal{K}$	29
+ INT4 KV cache (KIVI)	35	10.5	45.5	$\mathcal{W}$	17
+ 2:4 sparsity	17.5	10.5	28	$\mathcal{W}$	10.4
+ Spec. decoding ($\gamma = 5$, $\alpha = 0.8$)	4.4*	10.5	14.9	$\mathcal{K}$	5.6

* Effective per-token weight I/O: $\mathcal{W}/\bar{n}_{\text{accepted}}$ where $\bar{n}_{\text{accepted}} \approx 4$ tokens per verification pass. The assumption $\alpha = 0.8$ is optimistic; acceptance rates are task-dependent and degrade with larger γ, so the effective speedup in practice may be lower for diverse workloads. Note that Llama-3 70B already uses GQA with $n_{\text{kv}} = 8$, so GQA's benefit is architectural and already reflected in $\mathcal{K}$ throughout the table.

Amortization note: $\mathcal{W}$ is amortized by $\bar{n}_{\text{accepted}}$ because weights are loaded once per verification pass and shared across all γ candidate tokens. $\mathcal{K}$ is *not* amortized here because each accepted token extends the cache, so subsequent decode steps read a progressively larger $\mathcal{K}$—the per-step KV cost is not reduced, only the number of steps per useful token. If one instead accounts for the fact that the KV cache is also read only once per verification pass (producing ∼4 tokens), the amortized total would be $(4.4 + 10.5/4) \approx 7.0$ GB per effective token, yielding TPOT ≈ 2.6 ms. The table uses the conservative (unamortized $\mathcal{K}$) accounting, which better reflects steady-state behavior as the cache grows over many steps.

has the optimization stack actually closed the roofline gap? Consider the row *before* speculative decoding (INT4 weights + INT4 KV + 2:4 sparsity): total FLOPs per step are $2 \times (P/2) \times B = 2 \times 35.3 \times 10^9 \times 32 \approx 2.3 \times 10^{12}$, while total data movement is $\mathcal{W} + \mathcal{K} = 17.5 + 10.5 = 28$ GB, giving $I \approx 81$—still below the H100's dense ridge point ($I^* \approx 295$) but only $\sim 3.7\times$, not orders of magnitude. Speculative decoding changes the picture further: the verification pass processes $\gamma = 5$ tokens in parallel, multiplying FLOPs by γ while data movement stays the same (weights and KV cache are each loaded once). This pushes I to ~ 403, which *exceeds* the dense FP16 ridge point—meaning the verification pass transitions from memory-bound to compute-bound. The memory wall does not so much "hold" as get replaced: aggressive I/O optimization genuinely closes the bandwidth gap, only to expose compute as the new ceiling. This transition is itself a validation of the I/O-centric framing: the techniques surveyed here are working, and the next frontier is co-optimizing bandwidth and compute rather than treating them independently.

8.2 Profiling and Roofline Methodology

Validating these numbers requires per-kernel profiling. NVIDIA Nsight Compute provides kernel-level arithmetic intensity and HBM bandwidth, enabling direct roofline positioning. Nsight Systems adds timeline views for diagnosing pipeline bubbles and transfer overlap in multi-GPU setups. LLM-Viewer [90] computes theoretical roofline

bounds from architecture parameters alone; for decode these estimates are within 10–15% of measurements because the simple model $t_{\text{decode}} \approx \mathcal{W}/\beta_{\text{HBM}}$ dominates when the system is deeply memory-bound. One practical subtlety: attention kernels during prefill run at $I \approx d_h$ (typically 128), which is below the H100 ridge point—meaning attention is memory-bound during prefill even when FFN layers are compute-bound, a within-pass heterogeneity that complicates optimization.

9 Open Challenges and Future Directions

Despite the progress surveyed above, several hard I/O problems remain open. We frame seven of them as concrete research questions.

Can we achieve sub-linear KV growth without losing retrieval fidelity? Context windows are expanding faster than compression can keep up. At 1M tokens, even a GQA model with 8 KV heads and 80 layers needs ∼320 GB of KV cache per sequence in FP16. Current eviction methods (H_2O, StreamingLLM) cap cache size but sacrifice evicted context, hurting tasks like needle-in-a-haystack. The open question is whether an information-theoretic lower bound exists: for N tokens with entropy rate H, is $\Omega(N \cdot H)$ cache bits necessary for attention fidelity, or can learned compression exploit the low-rank structure of attention to achieve $o(N)$? State-space models [91], hybrid attention-SSM architectures, and retrieval-augmented approaches that replace caching with on-demand re-encoding are all promising.

Can expert caching be formalized as an online optimization problem? MoE expert activation is input-dependent and non-stationary, yet current systems (PowerInfer, MoE-Infinity) rely on static LRU. The problem can be cast as online weighted paging with variable page sizes [92]: E experts of size w_e, GPU capacity $C < \sum_e w_e$, request sequence driven by input tokens—find an eviction/prefetch policy minimizing total I/O. LRU achieves competitive ratio $O(k)$ where $k = C/\max_e w_e$; can learned prefetching, using the router's pre-softmax logits as a lookahead signal, reach $O(1)$? Joint optimization of expert placement and routing policy—rather than treating them independently—is also open.

Can data movement energy scale sub-linearly with model size? A 70B FP16 decode step burns ∼0.7 J per token on HBM reads alone versus ∼0.01 J on compute [13]—a 70× gap. Can PIM (eliminating bus transfers) and CXL (reducing redundant copies in multi-tenant setups) bring us below 0.1 J/token for a 70B model at FP16-equivalent accuracy? The waterfall analysis (Table 7) suggests ∼12× data-movement reduction is achievable, implying ∼0.06 J/token if hardware efficiency tracks software—a target PIM-augmented HBM could plausibly hit.

What is the I/O-optimal reasoning budget? Chain-of-thought and tree-search (Section 6.5) multiply KV I/O by reasoning length. If accuracy scales as $\log(T)$ in reasoning tokens T but cumulative KV reads scale as $O(T^2)$ (each of the T steps reads a linearly growing cache, $\sum_{t=1}^{T} t = O(T^2)$), there exists an I/O-optimal T^* beyond which extra reasoning gives negative returns per joule. Mapping this Pareto frontier—quality vs. I/O cost for a fixed HBM budget—is wide open.

What bandwidth-to-compute ratio should a decode-optimized chip have? Current GPUs target $I^* \approx 300$–400 for training, but decode runs at $I \approx 1$–32.

A decode-optimized accelerator would aim for $I^* \approx 10$–50, meaning 10× more bandwidth or 10× less compute. Groq and Cerebras validate this but sacrifice generality. UMA systems reach this sweet spot almost by accident: the M4 Max at $I^* \approx 49$ (peak bandwidth) gets 16% utilization at $B = 8$ vs. under 3% on H100. Whether UMA's lower ridge point is a design accident or a template for decode-optimized silicon is an open question—as is whether disaggregation (Section 6.3) makes the question moot by giving different hardware to each phase.

Can we standardize bytes-per-token benchmarking? MLPerf Inference includes LLM workloads but reports only throughput and latency, not I/O decomposition. We argue benchmarks should report weight bandwidth utilization ($\mathcal{W}_{\text{achieved}}/\beta_{\text{peak}}$), KV memory efficiency (allocated vs. used bytes), activation transfer fraction of step time, and energy per token decomposed into compute, HBM, and interconnect. Without this, it is impossible to know *why* a system is slow—and thus what to fix.

How do vision and audio tokens reshape the I/O profile? Multimodal LLMs encode images, audio, and video as token sequences concatenated with text. A single high-resolution image can produce hundreds to thousands of visual tokens, each generating persistent KV entries. This disproportionately inflates $\mathcal{K}$: visual tokens dominate cache size but are attended to rather than generated, creating a skewed read-to-write ratio. Attention sparsity may also differ between visual and textual tokens, suggesting modality-aware KV compression—more aggressive on visual tokens—could yield savings beyond modality-agnostic methods. This direction is largely unexplored.

10 Conclusion

The central claim of this survey is simple: the dominant bottleneck for autoregressive LLM inference is data movement, not computation. We organized the analysis around three I/O flows—model weights $\mathcal{W}$, KV cache $\mathcal{K}$, and activations $\mathcal{A}$—each dominating under different conditions: $\mathcal{W}$ at small batches and short contexts, $\mathcal{K}$ at large batches or long contexts, $\mathcal{A}$ in distributed multi-GPU setups. The roofline model provides a unifying framework: decode runs at $I \approx 1$, far below any modern accelerator's ridge point.

Our composability analysis shows that stacking INT4 weights, KV quantization, 2:4 sparsity, and speculative decoding yields about 12× cumulative I/O reduction. Before speculative decoding, the system operates ∼3.7× below the dense FP16 ridge point; with it, the verification pass crosses the ridge entirely, transitioning from memory-bound to compute-bound. The dominant bottleneck oscillates between $\mathcal{W}$ and $\mathcal{K}$—and ultimately between bandwidth and compute—as optimizations are layered, a dynamic that static taxonomies miss and that motivates the I/O-centric framing. With inference-time compute scaling (chain-of-thought, tree search) and million-token contexts on the rise, the balance is shifting further toward $\mathcal{K}$-dominated regimes, making KV compression and architectural KV reduction (GQA, MLA) increasingly important.

Across the stack—quantization and sparsity at the algorithm level, FlashAttention and PagedAttention at the system level, HBM scaling and CXL and PIM at the

hardware level—every effective technique maps to the same goal: fewer bytes per useful token. Yet hardware trends show the memory wall widening: each GPU generation increases compute faster than bandwidth. Meanwhile, Unified Memory Architectures are opening a parallel deployment frontier for edge inference, where the flat memory hierarchy eliminates offloading penalties and makes 7–13B models practical on consumer devices.

We hope this framing is useful—both for evaluating new optimizations (ask: which flow does it target, and is that flow currently dominant?) and for choosing hardware. The companion toolkit [44] makes the analysis concrete: compute the I/O profile, crossover batch size, and estimated TPOT for any new model within minutes of release.

Appendix A Notation Reference and Worked Example

Table A1 consolidates the symbols used throughout this survey. All quantities refer to a single autoregressive decode step unless noted otherwise.

Worked example: reproducing the FP16 baseline of Table 7.

We trace the computation of $\mathcal{W}$, $\mathcal{K}$, and TPOT for the first row of Table 7: Llama-3 70B in FP16 on a single H100, with $B = 32$ and $s = 4{,}096$.

Step 1: Model parameters. Llama-3 70B has $L = 80$ layers, $n_h = 64$ query heads, $n_{\text{kv}} = 8$ GQA KV heads, and $d_h = 128$ (so $d_{\text{model}} = 8{,}192$). The total parameter count is $P \approx 70.6 \times 10^9$.

Step 2: Weight I/O ($\mathcal{W}$). At FP16 ($b_w = 16$), the full weight footprint loaded per decode step is:

$$\mathcal{W} = P \times \frac{b_w}{8} = 70.6 \times 10^9 \times 2 \;\approx\; 141\,\text{GB}.$$

This is independent of B—every sequence in the batch shares the same weight read.

Step 3: KV cache I/O ($\mathcal{K}$). Each token contributes keys and values across all layers. Per token per layer:

$$2 \times n_{\text{kv}} \times d_h \times \frac{b_{\text{kv}}}{8} = 2 \times 8 \times 128 \times 2 = 4{,}096 \text{ bytes}.$$

Per token across all $L = 80$ layers: $80 \times 4{,}096 = 327{,}680$ bytes $\approx 0.31\,\text{MB}$. Per sequence at $s = 4{,}096$:

$$\mathcal{K}_{\text{seq}} = 4{,}096 \times 327{,}680 = 1.342 \times 10^9 \text{ bytes} \approx 1.3\,\text{GB}.$$

For the full batch:

$$\mathcal{K} = B \times \mathcal{K}_{\text{seq}} = 32 \times 1.3 \approx 42\,\text{GB}.$$

Step 4: TPOT. Total data movement is $\mathcal{W} + \mathcal{K} = 141 + 42 = 183\,\text{GB}$. On the H100 at $\beta_{\text{eff}} = 0.8 \times 3.35 = 2.68\,\text{TB/s}$:

$$\text{TPOT} \approx \frac{183}{2{,}680} = 0.068\,\text{s} = 68\,\text{ms}.$$

Table A1 Summary of notation.

Symbol	Description	Unit
Model architecture		
L	Number of transformer layers	—
d_{model}	Hidden (residual-stream) dimension	—
d_{ff}	FFN intermediate dimension	—
d_h	Per-head dimension (d_{model}/n_h)	—
n_h	Number of attention (query) heads	—
n_{kv}	Number of KV heads ($= n_h$ for MHA)	—
P	Total model parameters	—
α	FFN multiplier: 2 (standard) or 3 (gated)	—
Precision and serving		
b_w	Weight precision	bits
b_{kv}	KV cache precision	bits
b_a	Activation precision	bits
B	Batch size (concurrent sequences)	—
s	Current sequence length (context)	tokens
N	Number of input tokens (prefill)	tokens
I/O flows		
$\mathcal{W}$	Weight I/O per decode step	bytes
$\mathcal{K}$	KV cache I/O per decode step	bytes
$\mathcal{A}$	Activation I/O per decode step	bytes
$\mathcal{K}_{\text{seq}}$	KV cache size per sequence at length s	bytes
Hardware and performance		
π	Peak compute throughput	FLOP/s
β	Peak memory bandwidth	bytes/s
β_{eff}	Sustained (effective) memory bandwidth	bytes/s
β_{UMA}	Unified-memory bandwidth	bytes/s
I	Arithmetic intensity (FLOP/byte)	—
I^*	Ridge-point intensity (π/β)	—
Speculative decoding		
γ	Draft length (tokens per speculation)	—
α_{spec}	Token acceptance rate	—
$\bar{n}_{\text{accepted}}$	Expected accepted tokens per pass	—

Since $\mathcal{W} = 141 \gg \mathcal{K} = 42$, the system is $\mathcal{W}$-dominated, matching the first row of Table 7. Subsequent rows follow the same arithmetic with modified b_w, b_{kv}, or effective parameter counts.

Appendix B Crossover Batch Size Derivation

The *crossover batch size* B^* is the batch at which KV cache I/O overtakes weight I/O during decode. It governs which optimization family (weight compression vs. KV

compression) yields the greater return at a given operating point, and its value shifts as optimizations are applied—the oscillation described in Section 8.

General form.

Setting $\mathcal{K} = \mathcal{W}$:

$$B^* = \frac{\mathcal{W}}{\mathcal{K}_{\text{seq}}} = \frac{P \cdot (b_w/8)}{L \cdot 2 \cdot s \cdot n_{\text{kv}} \cdot d_h \cdot (b_{\text{kv}}/8)}. \tag{B1}$$

For $B < B^*$ the system is $\mathcal{W}$-dominated and weight compression has higher marginal impact. For $B > B^*$ the system is $\mathcal{K}$-dominated and KV compression (or eviction, or architectural reduction) matters more. Note that B^* depends on s: at longer contexts the crossover drops, making KV dominance more likely.

Simplification for GQA models.

Since $\mathcal{K}_{\text{seq}}$ per token per layer is $2 \cdot n_{\text{kv}} \cdot d_h \cdot (b_{\text{kv}}/8)$ bytes, we can write:

$$B^* = \frac{P \cdot b_w}{2 \cdot L \cdot s \cdot n_{\text{kv}} \cdot d_h \cdot b_{\text{kv}}}. \tag{B2}$$

The GQA ratio n_h/n_{kv} directly scales B^*: moving from MHA ($n_{\text{kv}} = n_h$) to GQA with $n_{\text{kv}} = 8$ raises B^* by $n_h/8$, making the system $\mathcal{W}$-dominated over a wider batch range.

Worked examples for Llama-3 70B ($s = 4{,}096$).

We illustrate how B^* shifts under different quantization regimes, using $L = 80$, $n_{\text{kv}} = 8$, $d_h = 128$, $P = 70.6 \times 10^9$.

Case 1: FP16 weights, FP16 KV. $\mathcal{K}_{\text{seq}} = 1.3\,\text{GB}$ (Section A).

$$B^* = \frac{141}{1.3} \approx 108.$$

At batches below 108 the system is weight-bound; above it, KV-bound.

Case 2: INT4 weights, FP16 KV. $\mathcal{W}$ drops to 35 GB; $\mathcal{K}_{\text{seq}}$ unchanged at 1.3 GB.

$$B^* = \frac{35}{1.3} \approx 27.$$

Weight quantization alone pulls B^* down by 4×, making KV the bottleneck at much smaller batches.

Case 3: INT4 weights, INT4 KV. Both flows compressed 4×. $\mathcal{W} = 35\,\text{GB}$, $\mathcal{K}_{\text{seq}} = 0.33\,\text{GB}$.

$$B^* = \frac{35}{0.33} \approx 106.$$

Symmetric quantization restores B^* close to the original value—the bottleneck returns to weights for typical batch sizes.

Case 4: INT4 weights + 2:4 sparsity, INT4 KV. $\mathcal{W} = 17.5\,\text{GB}$, $\mathcal{K}_{\text{seq}} = 0.33\,\text{GB}$.

$$B^* = \frac{17.5}{0.33} \approx 53.$$

Weight sparsity shifts the crossover back down; moderate batches become KV-dominated again.

Context-length sensitivity.

B^* is inversely proportional to s. For the INT4-weights, FP16-KV case ($B^* = 27$ at $s = 4{,}096$):

$$B^*(s) = \frac{35\,\text{GB}}{80 \times 4{,}096 \times s/10^9} = \frac{35}{0.000328 \cdot s}.$$

At $s = 32{,}768$ (32K context): $B^* \approx 3$. At $s = 131{,}072$ (128K context): $B^* < 1$, meaning even $B = 1$ is KV-dominated. This explains why long-context inference is *always* a KV problem, regardless of weight precision.

Using B^* in practice.

Given a target deployment configuration (model, quantization levels, expected batch size, context length), compute B^* from Equation (B2) and compare to B:

$$\text{Next optimization} = \begin{cases} \text{weight compression (quantize, prune)} & \text{if } B < B^*, \\ \text{KV compression (quantize, evict, GQA)} & \text{if } B > B^*, \\ \text{either (balanced)} & \text{if } B \approx B^*. \end{cases}$$

The companion toolkit [44] computes B^* automatically for any model configuration.

Declarations

Disclaimer.

The author is employed by Oracle America Inc. However, this work was conducted independently outside the scope of employment and without the use of company resources. The views expressed are solely those of the author and do not represent the views of Oracle. This survey is based solely on publicly available literature and does not use or disclose any proprietary or confidential information.

Competing interests.

The author declares no competing interests. Oracle America Inc. offers cloud infrastructure services that include GPU-based compute; this survey does not evaluate, endorse, or compare any commercial offerings.

Funding.

No funding was received for this work.

AI disclosure.
An AI tool (Claude, Anthropic) was used during manuscript preparation to assist with verifying numerical calculations, cross-checking dimensional consistency of equations, and identifying inconsistencies in hardware specifications. All technical content, analysis, arguments, and conclusions are the sole intellectual contribution of the author. The author has reviewed, verified, and takes full responsibility for all content in the final manuscript.

References

[1] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., *et al.*: Language models are few-shot learners. Advances in Neural Information Processing Systems **33**, 1877–1901 (2020)

[2] Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: GPT-4 technical report. arXiv preprint arXiv:2303.08774 (2023)

[3] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al.: Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 (2023)

[4] Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., et al.: The Llama 3 herd of models. arXiv preprint arXiv:2407.21783 (2024)

[5] Kaplan, J., McCandlish, S., Henighan, T., Brown, T.B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., Amodei, D.: Scaling laws for neural language models. arXiv preprint arXiv:2001.08361 (2020)

[6] Jiang, A.Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D.S., Casas, D.d.l., Hanna, E.B., Bressand, F., et al.: Mixtral of experts. arXiv preprint arXiv:2401.04088 (2024)

[7] NVIDIA Corporation: NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. GV100 GPU: 125 TFLOP/s (FP16 Tensor Core), 900 GB/s HBM2, 16/32 GB (2017)

[8] NVIDIA Corporation: NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core. GH100 GPU: 990 TFLOP/s (FP16 Tensor Core), 3.35 TB/s HBM3, 80 GB. SXM form factor. 132 SMs, 50 MB L2, 228 KB shared memory per SM (2022)

[9] NVIDIA Corporation: NVIDIA Blackwell Architecture Technical Brief. https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/.

GB200/B200 GPU: 2.25 PFLOP/s (FP16), 4.5 PFLOP/s (FP8), 9 PFLOP/s (FP4). 8 TB/s HBM3e, 180/192 GB (2024)

[10] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in Neural Information Processing Systems **30** (2017)

[11] Ivanov, A., Dryden, N., Ben-Nun, T., Li, S., Hoefler, T.: Data movement is all you need: A case study on optimizing transformers. Proceedings of Machine Learning and Systems **3**, 711–732 (2021)

[12] Kim, S., Hooper, C., Wattanawong, T., Kang, M., Yan, R., Genc, H., Dinh, G., Huang, Q., Suber, K., Kim, H., et al.: Full stack optimization of transformer inference: A survey. arXiv preprint arXiv:2302.14017 (2023)

[13] Horowitz, M.: 1.1 computing's energy problem (and what we can do about it). IEEE International Solid-State Circuits Conference (2014)

[14] Samsi, S., Zhao, D., McDonald, J., Li, B., Michaleas, A., Jones, M., Bergeron, W., Kepner, J., Tiwari, D., Gadepally, V.: From words to watts: Benchmarking the energy costs of large language model inference. IEEE High Performance Extreme Computing Conference (2023)

[15] Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguia, L.-M., Rothchild, D., So, D., Texier, M., Dean, J.: Carbon emissions and large neural network training. arXiv preprint arXiv:2104.10350 (2021)

[16] Frantar, E., Ashkboos, S., Hoefler, T., Alistarh, D.: GPTQ: Accurate post-training quantization for generative pre-trained transformers. International Conference on Learning Representations (2023)

[17] Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.-M., Wang, W.-C., Xiao, G., Dang, X., Gan, C., Han, S.: AWQ: Activation-aware weight quantization for LLM compression and acceleration. Machine Learning and Systems **6**, 87–100 (2024)

[18] Dettmers, T., Lewis, M., Belkada, Y., Zettlemoyer, L.: LLM.int8(): 8-bit matrix multiplication for transformers at scale. Advances in Neural Information Processing Systems **35**, 30318–30332 (2022)

[19] Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., Han, S.: SmoothQuant: Accurate and efficient post-training quantization for large language models. International Conference on Machine Learning, 38087–38099 (2023)

[20] Frantar, E., Alistarh, D.: SparseGPT: Massive language models can be accurately pruned in one-shot. International Conference on Machine Learning, 10323–10337 (2023)

[21] Sun, M., Liu, Z., Bair, A., Kolter, J.Z.: A simple and effective pruning approach for large language models. International Conference on Learning Representations (2024)

[22] Dao, T., Fu, D., Ermon, S., Rudra, A., Ré, C.: FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In: Advances in Neural Information Processing Systems, vol. 35, pp. 16344–16359 (2022)

[23] Dao, T.: FlashAttention-2: Faster attention with better parallelism and work partitioning. International Conference on Learning Representations (2024)

[24] Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., Dao, T.: FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. arXiv preprint arXiv:2407.08691 (2024)

[25] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C.H., Gonzalez, J., Zhang, H., Stoica, I.: Efficient memory management for large language model serving with PagedAttention. In: Proceedings of the 29th Symposium on Operating Systems Principles, pp. 611–626 (2023)

[26] Leviathan, Y., Kalman, M., Matias, Y.: Fast inference from transformers via speculative decoding. International Conference on Machine Learning, 19274–19286 (2023)

[27] Chen, C., Borgeaud, S., Irving, G., Lespiau, J.-B., Sifre, L., Jumper, J.: Accelerating large language model decoding with speculative sampling. arXiv preprint arXiv:2302.01318 (2023)

[28] Yu, G.-I., Jeong, J.S., Kim, G.-W., Kim, S., Chun, B.-G.: ORCA: A distributed serving system for transformer-based generative models. In: 16th USENIX Symposium on Operating Systems Design and Implementation, pp. 521–538 (2022)

[29] Zheng, L., Yin, L., Xie, Z., Huang, J., Sun, C., Yu, C.H., Cao, S., Kober, C., Gonzalez, J.E., Barrett, C., et al.: SGLang: Efficient execution of structured language model programs. Advances in Neural Information Processing Systems **37** (2024)

[30] NVIDIA: TensorRT-LLM: A framework for accelerating large language model inference. GitHub repository (2024). https://github.com/NVIDIA/TensorRT-LLM

[31] Sharma, D.D., Guz, Z., Jefferson, R.: An introduction to the Compute Express Link (CXL) interconnect. arXiv preprint arXiv:2306.11227 (2023)

[32] Mutlu, O., Ghose, S., Gómez-Luna, J., Ausavarungnirun, R.: A modern primer on processing in memory. Emerging Computing: Accelerators and Workloads (2020)

[33] Wan, Z., Wang, X., Liu, C., Alam, S., Zheng, Y., Liu, J., Qu, Z., Yan, S., Zhu, Y., Zhang, Q., et al.: Efficient large language models: A survey. Transactions on Machine Learning Research (2024)

[34] Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Wang, Z., Zhang, Z., Wong, R.Y.Y., Zhu, A., Yang, L., Shi, X., et al.: Towards efficient generative large language model serving: A survey from algorithms to systems. arXiv preprint arXiv:2312.15234 (2023)

[35] Zhou, Z., Ning, X., Hong, K., Fu, T., Xu, J., Li, S., Lou, Y., Wang, L., Zhu, Z., Cai, J., et al.: A survey on efficient inference for large language models. arXiv preprint arXiv:2404.14294 (2024)

[36] Park, Y., Zheng, K., Hessel, I., Padmanabhan, R., Prasanna, S., Ramakrishnan, B.D., Huynh, B., Deshpande, C., et al.: Inference optimization of foundation models on AI accelerators. arXiv preprint arXiv:2407.09111 (2024)

[37] Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Communications of the ACM **52**(4), 65–76 (2009)

[38] Abts, D., Ross, J., Sparber, J., Wong-VanHaren, M., Baker, M., Hawkins, T., Bell, A., Thompson, J., Kahsai, T., Kimmell, G., et al.: A software-defined tensor streaming multiprocessor for large-scale machine learning. International Symposium on Computer Architecture, 567–580 (2022)

[39] Lie, S.: Wafer-scale deep learning. IEEE Hot Chips Symposium (2023)

[40] Fedus, W., Zoph, B., Shazeer, N.: Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. Journal of Machine Learning Research **23**(120), 1–39 (2022)

[41] Dai, D., Deng, C., Zhao, C., Xu, R.X., Gao, H., Chen, D., Li, J., Zeng, W., Yu, X., Wu, Y., et al.: DeepSeekMoE: Towards ultimate expert specialization in mixture-of-experts language models. arXiv preprint arXiv:2401.06066 (2024)

[42] Alizadeh, K., Mirzadeh, I., Belenko, D., Khatamifard, K., Cho, M., Del Mundo, C.C., Rastegari, M., Farajtabar, M.: LLM in a flash: Efficient large language model inference with limited memory. International Conference on Machine Learning (2024)

[43] Gerganov, G.: llama.cpp. https://github.com/ggerganov/llama.cpp (2023)

[44] Chowdhury, R.: llm-io-survey: I/O Bottleneck Analysis Toolkit for LLM Inference. https://doi.org/10.5281/zenodo.18780115. Companion code for this survey. Parses HuggingFace model configs, computes per-token I/O breakdown, identifies W/K crossover batch sizes, and generates roofline, waterfall, and scaling

visualizations. MIT licensed. (2025)

[45] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al.: LLaMA: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023)

[46] Meta AI: Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. Meta AI Blog (2024). https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/

[47] Abdin, M., Jacobs, S.A., Awan, A.A., Aneja, J., Awadallah, A., Awadalla, H., Bach, N., Bahree, A., Bakhtiari, A., Beber, H., et al.: Phi-3 technical report: A highly capable language model locally on your phone. arXiv preprint arXiv:2404.14219 (2024)

[48] Gemma Team: Gemma 2: Improving open language models at a practical size. arXiv preprint arXiv:2408.00118 (2024)

[49] Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D.S., Casas, D.d.l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al.: Mistral 7b. arXiv preprint arXiv:2310.06825 (2023)

[50] Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Wei, H., et al.: Qwen2.5 technical report. arXiv preprint arXiv:2412.15115 (2024)

[51] Almazrouei, E., Alobeidli, H., Alshamsi, A., Cappelli, A., Cojocaru, R., Debbah, M., Goffinet, E., Hesslow, D., Launay, J., Malartic, Q., et al.: The Falcon series of open language models. arXiv preprint arXiv:2311.16867 (2023)

[52] DeepSeek-AI: DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model. arXiv preprint arXiv:2405.04434 (2024)

[53] DeepSeek-AI: DeepSeek-V3 technical report. arXiv preprint arXiv:2412.19437 (2024)

[54] DeepSeek-AI: DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. arXiv preprint arXiv:2501.12948 (2025)

[55] Tseng, A., Chee, J., Sun, Q., Kuleshov, V., De Sa, C.: QuIP#: Even better LLM quantization with hadamard incoherence and lattice codebooks. International Conference on Machine Learning (2024)

[56] Egiazarian, V., Panferov, A., Kuznedelev, D., Frantar, E., Babenko, A., Alistarh, D.: Extreme compression of large language models via additive quantization. International Conference on Machine Learning (2024)

[57] Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R.,

Ha, S., Heinecke, A., Judd, P., Kamalu, J., et al.: FP8 formats for deep learning. arXiv preprint arXiv:2209.05433 (2022)

[58] Mishra, A., Latorre, J.A., Pool, J., Stosic, D., Stosic, D., Venber, G., Micikevicius, P.: Accelerating sparse deep neural networks. arXiv preprint arXiv:2104.08378 (2021)

[59] Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., Zhang, C.: FlexGen: High-throughput generative inference of large language models with a single GPU. International Conference on Machine Learning, 31094–31116 (2023)

[60] Song, Y., Mi, Z., Xie, H., Chen, H.: PowerInfer: Fast large language model serving with a consumer-grade GPU. USENIX Symposium on Operating Systems Design and Implementation (2024)

[61] Xue, L., Fu, Y., Fang, Z., Luo, L., Li, X., Sun, J., et al.: MoE-Infinity: Offloading-efficient MoE model serving. arXiv preprint arXiv:2401.14361 (2024)

[62] Eliseev, A., Mazur, D.: Fast inference of mixture-of-experts language models with offloading. arXiv preprint arXiv:2312.17238 (2023)

[63] Cai, T., Li, Y., Geng, Z., Peng, H., Lee, J.D., Chen, D., Dao, T.: Medusa: Simple LLM inference acceleration framework with multiple decoding heads. International Conference on Machine Learning (2024)

[64] Li, Y., Wei, F., Zhang, C., Zhang, H.: EAGLE: Speculative sampling requires rethinking feature uncertainty. International Conference on Machine Learning (2024)

[65] Shazeer, N.: Fast transformer decoding: One write-head is all you need. arXiv preprint arXiv:1911.02150 (2019)

[66] Ainslie, J., Lee-Thorp, J., Jong, M., Zemlyanskiy, Y., Lebrón, F., Sanghai, S.: GQA: Training generalized multi-query transformer models from multi-head checkpoints. Empirical Methods in Natural Language Processing (2023)

[67] Liu, Z., Yuan, J., Jin, H., Zhong, S., Xu, Z., Braverman, V., Chen, B., Hu, X.: KIVI: A tuning-free asymmetric 2bit quantization for KV cache. International Conference on Machine Learning (2024)

[68] Hooper, C., Kim, S., Mohammadzadeh, H., Mahoney, M.W., Shao, Y.S., Keutzer, K., Gholami, A.: KVQuant: Towards 10 million context length LLM inference with KV cache quantization. arXiv preprint arXiv:2401.18079 (2024)

[69] Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C., et al.: H_2O: Heavy-hitter oracle for efficient generative

inference of large language models. Advances in Neural Information Processing Systems **36** (2024)

[70] Xiao, G., Tian, Y., Chen, B., Han, S., Lewis, M.: Efficient streaming language models with attention sinks. International Conference on Learning Representations (2024)

[71] Liu, Y., Li, H., Du, K., Yao, J., Cheng, Y., Huang, Y., Lu, S., Maire, M., Hoffmann, H., Holtzman, A., Ananthanarayanan, G.: CacheGen: KV cache compression and streaming for fast large language model serving. arXiv preprint arXiv:2310.07240 (2024)

[72] Lee, W., Lee, J., Seo, J., Sim, H.: InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. arXiv preprint arXiv:2406.19707 (2024)

[73] Dao, T., Haziza, D., Massa, F., Sizov, G.: Flash-Decoding for Long-Context Inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html (2023)

[74] Chen, T., Xu, B., Zhang, C., Guestrin, C.: Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174 (2016)

[75] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., Catanzaro, B.: Megatron-LM: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053 (2019)

[76] Liu, H., Zaharia, M., Abbeel, P.: Ring attention with blockwise transformers for near-infinite context. International Conference on Learning Representations (2024)

[77] Brandon, W., Mishra, A., Nrusimha, S., Panda, R., Kelly, J.R.: Striped attention: Faster ring attention for causal transformers. arXiv preprint arXiv:2311.09431 (2023)

[78] Holmes, C., Tanber, M., Rashidi, M., Zhang, R., Awan, A.A., Rasley, J., Rajbhandari, S., He, Y.: DeepSpeed-FastGen: High-throughput text generation for LLMs via MII and DeepSpeed-Inference. arXiv preprint arXiv:2401.08671 (2024)

[79] Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., Bianchini, R.: Splitwise: Efficient generative LLM inference using phase splitting. International Symposium on Computer Architecture (2024)

[80] Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., Zhang, H.: DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. USENIX Symposium on Operating Systems Design and Implementation (2024)

[81] Qin, R., Li, Z., He, W., Zhang, M., Yang, Y., Zheng, W., Xu, L.: Mooncake: A KVCache-centric disaggregated architecture for LLM serving. arXiv preprint arXiv:2407.00079 (2024)

[82] NVIDIA Corporation: NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. GA100 GPU: 312 TFLOP/s (FP16 Tensor Core), 2.0 TB/s HBM2e, 80 GB (2020)

[83] CXL Consortium: Compute Express Link (CXL) Specification, Revision 3.0. https://www.computeexpresslink.org (2022)

[84] Lee, S., Kim, S.-h., et al.: Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product. ISCA (2022)

[85] Groq, Inc.: Groq Language Processing Unit (LPU) Inference Engine. https://groq.com (2024)

[86] Artificial Analysis: Llama 3.3 70B: API Provider Performance Benchmarking. https://artificialanalysis.ai/models/llama-3-3-instruct-70b/providers. Independent benchmark: Groq achieves ∼276 tokens/s (standard) and ∼1,665 tokens/s (speculative decoding) on Llama 3.3 70B (2024)

[87] Apple Inc.: Apple M4 Family of Chips. https://www.apple.com/newsroom/2024/10/apple-introduces-m4-pro-and-m4-max/. Unified Memory Architecture with up to 128 GB LPDDR5X at 546 GB/s (M4 Max) (2024)

[88] Apple Inc.: Apple Introduces M3, M3 Pro, and M3 Max. https://www.apple.com/newsroom/2023/10/apple-introduces-m3-m3-pro-and-m3-max/. The M3 Ultra (announced 2024) provides 192 GB unified memory at 819 GB/s via dual M3 Max die (2023)

[89] Hannun, A., et al.: MLX: An Array Framework for Apple Silicon. https://github.com/ml-explore/mlx. Designed for Unified Memory: lazy evaluation, shared-memory tensors, zero host-device copies (2024)

[90] Yuan, Z., Shang, Y., et al.: LLM-Viewer: A tool for analyzing and visualizing LLM inference efficiency. arXiv preprint arXiv:2402.16363 (2024)

[91] Gu, A., Dao, T.: Mamba: Linear-time sequence modeling with selective state spaces. arXiv preprint arXiv:2312.00752 (2023)

[92] Lykouris, T., Vassilvitskii, S.: Competitive caching with machine learned advice. Journal of the ACM **68**(4), 1–25 (2021)