PART IV · CHAPTER 11

KV Cache: The Heart of LLM Serving

Memory Math, Paging, Quantization, Prefix Caching, Offloading, Disaggregation, and Latent Compression

Book: AI/ML Infrastructure from Silicon to Scale | **Author:** Venkat Vinjam | **Cross-ref:** *HBM bandwidth (Ch 4) · Inference serving (Ch 6) · Quantization (Ch 8) · Training memory (Ch 10) · Networking/NCCL (Ch 14)*

> *"The KV cache is not an implementation detail. It is the central resource constraint that determines how many users your system can serve, how fast they receive their first token, and how much each request costs you. Every inference engineer must know its math cold."*
>
> — Production LLM Serving Principle

When a user sends a 2,000-token prompt to a 70B model, the GPU does not merely run the model once. It runs prefill — an $O(S^2)$ attention computation over the entire prompt — then enters decode mode, generating tokens one at a time. Each decode step must attend to every prior token. Without caching the intermediate Key and Value tensors, every new token would require recomputing attention from scratch: catastrophically slow at $O(S^2)$ per step.

The KV cache trades HBM memory for computation time. But HBM is the scarcest resource in any GPU cluster. Managing that tradeoff — how much to allocate, how to page it, how to compress it, how to reuse it across requests, and how to tier it when pressure mounts — is the defining engineering challenge of production LLM inference. This chapter covers the complete engineering picture, from the per-token memory formula to DeepSeek's 57× compression via Multi-Head Latent Attention.

FOUNDATIONAL

- Derive KV cache size for any model from the core formula
- Explain why contiguous pre-allocation wastes 60–80% of HBM
- Understand how PagedAttention applies OS virtual memory concepts
- Know the six numbers for LLaMA-3 70B KV sizing cold

INTERMEDIATE

- Capacity-plan a serving system with FP8 weights + FP8 KV
- Configure prefix caching with RadixAttention and LRU eviction
- Calculate the PCIe bandwidth required for CPU DRAM offloading
- Derive the minimum IB bandwidth for disaggregated KV transfer

PRINCIPAL LEVEL

- Design KV tiering with optimal swap-vs-recompute policy
- Size prefix cache for a given workload and hit rate target
- Compare GQA vs MLA at 128K context; know when MLA wins
- Select quantization granularity per application quality constraint

11.1 — KV Cache Memory Math

11.1.1 Why the KV Cache Exists

In autoregressive decoding, every output token must attend to every preceding token. Without caching, each decode step recomputes Key and Value projections for all prior tokens — an $O(t^2)$ cost that grows catastrophically with sequence length. The KV cache stores these tensors after first computation and reads them at each subsequent step, converting $O(t^2)$ to $O(t)$ per decode step. The trade is explicit: HBM capacity and bandwidth for a one-time write, in exchange for eliminating repeated recomputation.

11.1.2 The KV Cache Size Formula

```
KV CACHE SIZE FORMULA:

  KV_bytes = 2 × L × H_kv × d_head × S × B × dtype_bytes

  2         = K tensor + V tensor (always)
  L         = transformer layers
  H_kv      = KV heads (reduced by GQA; equals H_q in vanilla MHA)
  d_head    = head dimension (d_model / H_q for standard models)
  S         = sequence length (tokens in context)
  B         = concurrent requests (batch size)
  dtype     = BF16: 2 bytes | FP8: 1 byte | INT8: 1 byte | INT4: 0.5 bytes

Per token (S=1, B=1):
  KV_bytes_per_token = 2 × L × H_kv × d_head × dtype_bytes

The formula has 4 model constants (L, H_kv, d_head, dtype) and 2 workload variables (S, B).
To reduce KV: reduce H_kv via GQA, reduce dtype via quantization, limit S or B.
```

11.1.3 LLaMA-3 70B Worked Example

```
LLAMA-3 70B: L=80, H_q=64, H_kv=8 (GQA R=8), d_head=128, d_model=8192

Step 1 — Per-token cost (BF16):
  KV_per_token = 2 × 80 × 8 × 128 × 2 bytes = 327,680 bytes = 0.320 MB
  Every token occupies 320 KB of HBM for the lifetime of its request.

Step 2 — Per-request at 4K context (BF16):
  KV_per_request = 0.320 MB × 4,096 = 1,310,720 bytes = 1.25 GB/request
  H100 has 80 GB. At this rate, 64 users fill the GPU.

Step 3 — Weight memory (FP8 weights, TP=2):
  FP8 weights per GPU = 70 GB / 2 = 35 GB
  Runtime overhead = ~5 GB
  KV headroom = 80 − 35 − 5 = 40 GB per GPU

Step 4 — Capacity at FP8 KV (the production config):
  FP8 KV per token = 2 × 80 × 8 × 128 × 1 byte = 163,840 bytes = 0.160 MB
  FP8 KV per request (4K ctx) = 0.160 × 4,096 = 655 MB = 0.64 GB

  Max concurrent users (40 GB KV headroom):
    40,000 MB / 640 MB = 62.5 → ~62 concurrent users per GPU pair

  Scaling with context length:
    At 8K ctx:   1.28 GB/req → 31 users
    At 32K ctx:  5.12 GB/req →  7 users
    At 128K ctx: 20.5 GB/req →  1–2 users  ← KV compression becomes existential
```

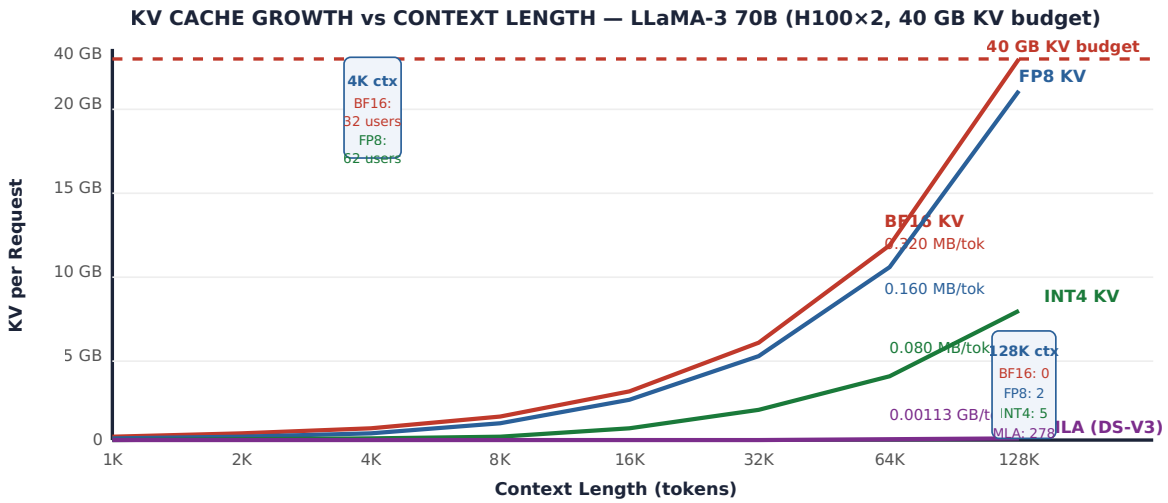


Figure 11.1 *— KV cache per-request size vs context length for LLaMA-3 70B (H100×2, 40 GB KV headroom). All curves are linear in context length; the y-axis shows GB per single request. BF16 hits the 40 GB budget at 128K context — zero concurrency. FP8 KV allows 2 concurrent users at 128K. INT4 allows 5. MLA (DeepSeek-V3, d_c=512) stays almost flat at ~0.14 GB even at 128K — enabling 278 concurrent users where BF16 allows none.*

11.1.4 The KV vs Weights Memory Split

```
KV VS WEIGHTS BALANCE POINT (LLaMA-3 70B, H100×2, TP=2):

  Fixed cost: FP8 weights 35 GB/GPU + overhead 5 GB = 40 GB per GPU
  KV budget: 40 GB (remaining after weights)
  FP8 KV per request (4K ctx): 0.64 GB

  Utilization at different batch sizes:
    B=1:   0.64 GB KV  (1.6% KV → weight-bound)
    B=10:  6.4 GB KV   (16% → still weight-bound)
    B=32:  20.5 GB KV  (51% → approaching balance)
    B=62:  39.7 GB KV  (99% → KV-bound)

  CROSSOVER POINT (KV budget = weight budget):
    40 GB = 0.64 GB × B → B = 62.5 at 4K context
    At 8K context: crossover at B ≈ 31
    At 32K context: crossover at B ≈ 8

  As average context length increases, you become KV-bound at
  progressively smaller batch sizes — KV management dominates.
```

11.2 — PagedAttention: Virtual Memory for Attention

11.2.1 The Fragmentation Problem

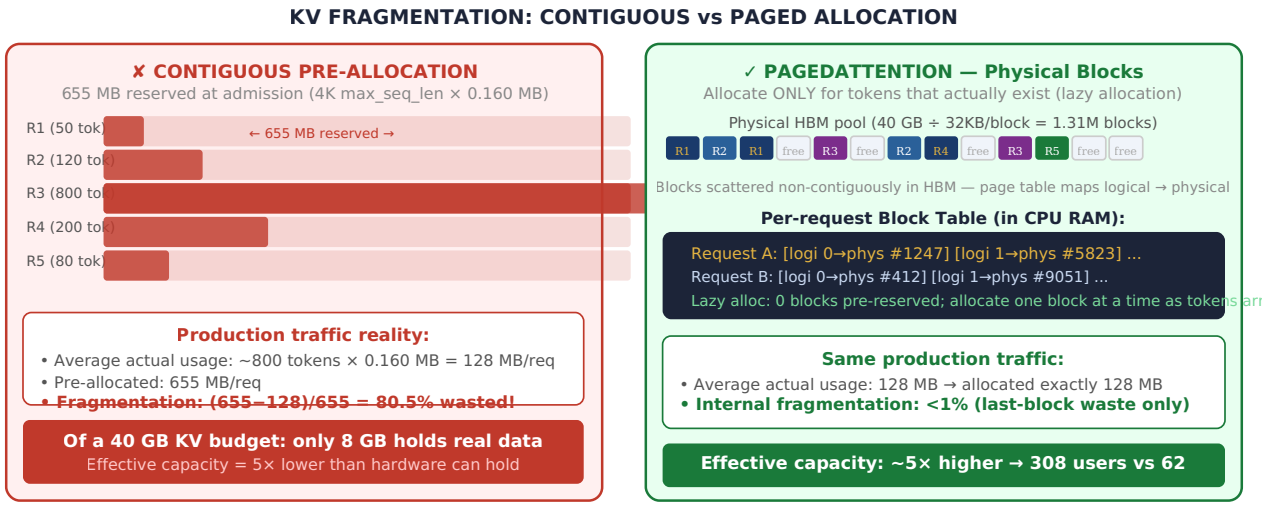


Figure 11.2 *— KV fragmentation: contiguous pre-allocation vs PagedAttention. Contiguous allocation reserves max_seq_len × KV_per_token at admission — 655 MB per request regardless of actual usage. With average actual usage of 128 MB (800 tokens), 80.5% of allocated HBM is wasted. PagedAttention allocates one 32 KB block per 16 tokens as they arrive, reducing internal fragmentation to the last-block remainder only (<1%). Same 40 GB KV budget: 62 users (contiguous) vs ~308 users (paged).*

11.2.2 PagedAttention: The Page Table Mechanics

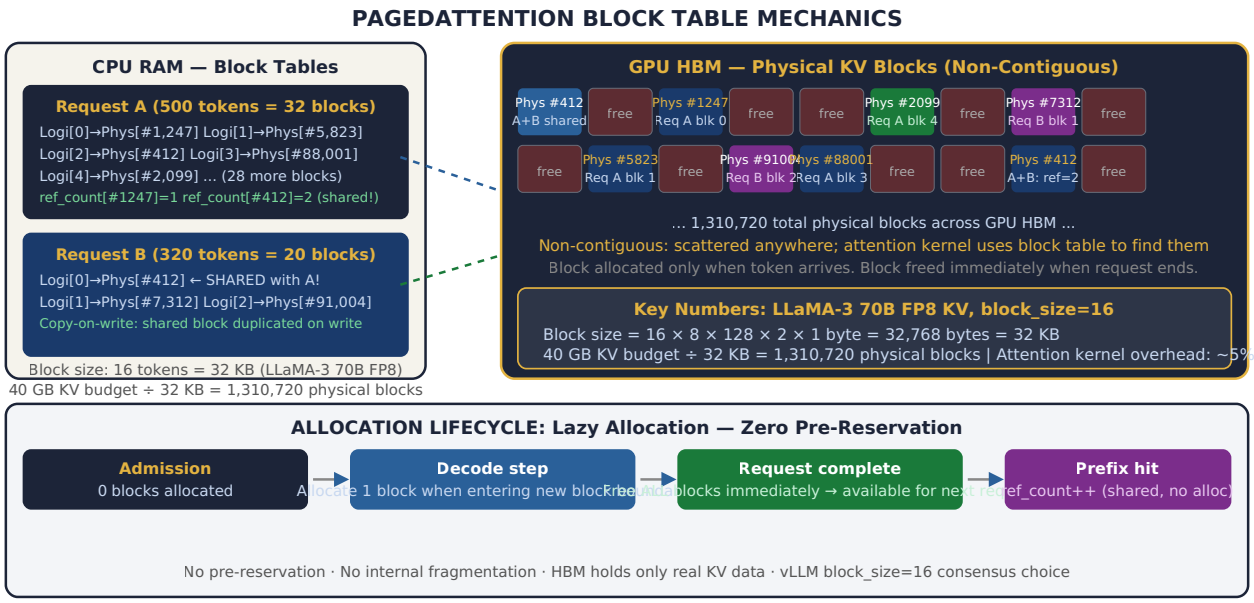


Figure 11.3 *— PagedAttention block table mechanics. Physical KV blocks (32 KB each at block_size=16 for LLaMA-3 70B FP8) are scattered non-contiguously in HBM. Each request holds a logical→physical mapping table in CPU RAM. Allocation is lazy: 0 blocks at admission, one block per 16 new tokens. Shared prefix blocks use reference counting (copy-on-write on divergence). The attention kernel uses the block table to gather K/V blocks, adding ~5% overhead vs contiguous FlashAttention.*

11.2.3 Block Size Selection and Tradeoffs

```
BLOCK SIZE TRADEOFF ANALYSIS:

Standard production choice: block_size=16 tokens

Last-block waste formula:
  Expected waste per request = (block_size − 1) / 2 × KV_per_token

  block_size=8:   3.5 × 0.160 MB =  0.56 MB  (fine)
  block_size=16:  7.5 × 0.160 MB =  1.20 MB  (standard)
  block_size=32: 15.5 × 0.160 MB =  2.48 MB  (slightly coarser)
  block_size=128: 63.5 × 0.160 MB = 10.16 MB (too coarse for prefix sharing)

Consensus: block_size=16 in vLLM, TRT-LLM, SGLang
  Exception: block_size=32–64 when context windows are consistently
  very long (>16K tokens) and prefix caching is low-priority
```

11.2.4 Copy-on-Write for Beam Search

```
COPY-ON-WRITE — Memory Savings for Beam Search (beam_width=4):

  4 beams share 512-token common prefix; each generates 100 unique decode steps

  Naive (no CoW):
    4 × (512 + 100) × 0.160 MB = 391.7 MB

  CoW (shared prefix blocks, ref_count=4):
    Shared prefix: 1 × 512 × 0.160 = 81.9 MB  (shared read-only)
    Unique suffixes: 4 × 100 × 0.160 = 64.0 MB
    Total: 145.9 MB  → 62.7% savings vs naive

  Mechanism: block is copied only when a beam actually DIVERGES.
  Most prefix blocks are never copied — shared read-only for entire search.
  Same mechanism applies to: parallel sampling, speculative decoding
```

11.3 — KV Cache Quantization

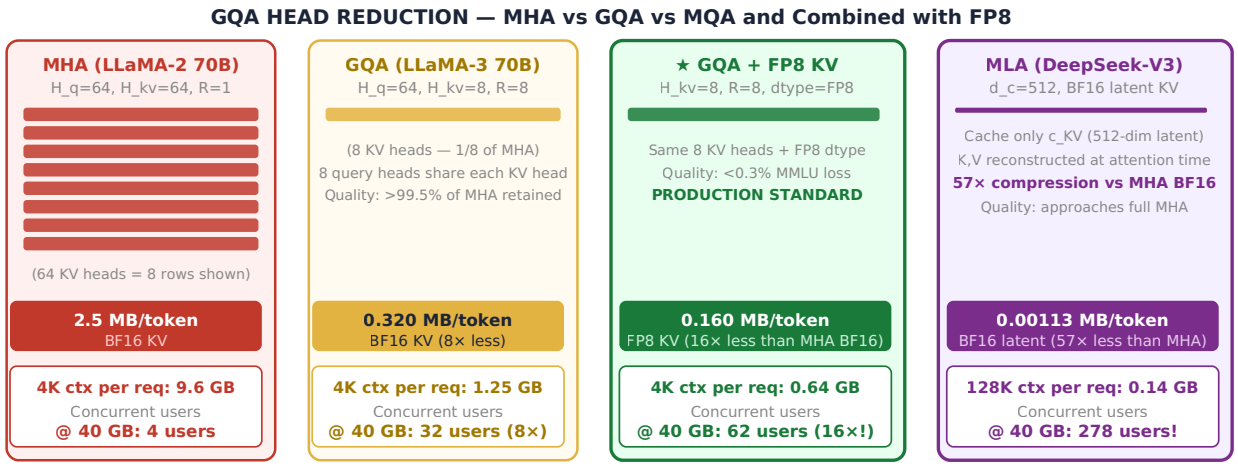


Figure 11.4 *— KV head reduction: MHA vs GQA vs GQA+FP8 vs MLA. MHA (LLaMA-2 style, H_kv=64): 2.5 MB/token BF16 — only 4 users at 4K context. GQA R=8 (LLaMA-3, H_kv=8): 0.320 MB/token — 8× improvement, 32 users. GQA+FP8 KV: 0.160 MB/token — 16× vs MHA BF16, 62 users, <0.3% quality loss — the production standard. MLA (DeepSeek-V3, d_c=512 latent): 0.00113 MB/token — 57× vs MHA BF16, 278 users even at 128K context.*

11.3.2 FP8 KV Cache — Production Default SHIPPED

```
FP8 KV QUANTIZATION (per-head scaling, online):

  dtype: 1 byte per element (E4M3: range ±448)
  Scale: one FP8/BF16 per KV head per layer (per-head scaling standard)

  For LLaMA-3 70B: 8 heads × 80 layers = 640 scales each for K,V = 2.5 KB overhead

  Quality (LLaMA-3 70B BF16 → FP8 KV):
    MMLU:        80.3% → 80.1%   (−0.2%)  ✅ production-safe
    HellaSwag:   87.1% → 86.9%   (−0.2%)  ✅
    TruthfulQA:  44.7% → 44.5%   (−0.2%)  ✅

  vLLM config:
    engine_args = AsyncEngineArgs(model="meta-llama/Llama-3-70b",
                                  kv_cache_dtype="fp8",
                                  gpu_memory_utilization=0.90)
```

11.3.3 INT8 KV Cache

```
INT8 KV QUANTIZATION:
  dtype: 1 byte per element (signed INT8: −128 to 127)
  Same memory footprint as FP8 but different quality profile

  Quality (LLaMA-3 70B with SmoothQuant-style INT8):
    MMLU: −0.5%  ✅  HumanEval: −0.8%  ✅  GSM8K: −0.6%  ✅

  Preferred over FP8 when:
    Deploying on A100/Ampere (no native FP8 hardware support)
    Framework does not yet support FP8 KV (check vLLM/TRT-LLM changelog)
    Conservative first deployment without FP8 evaluation on your specific model
```

11.3.4 INT4 KV Cache — Evaluation Required

```
INT4 KV QUANTIZATION:
  dtype: 0.5 bytes per element; always per-head scaling (per-tensor too coarse)
  Memory: 4× reduction vs BF16 → 80 KB/token (LLaMA-3 70B)

  Quality (LLaMA-3 70B):
    MMLU:      −1.5%  → acceptable for general chat
    HumanEval: −2.5%  ← SIGNIFICANT — code quality degrades noticeably
    GSM8K:     −2.0%  ← math reasoning measurably worse

  Use when:  128K+ context where KV is binding constraint; quality validated
  Avoid:     Code generation, mathematical reasoning, without task evaluation
  Combine:   INT4 KV (4×) + GQA R=8 (8×) = 32× vs MHA BF16!
```

11.3.6 GQA + FP8 KV — The Combined Formula

```
COMBINED GQA + FP8 KV CAPACITY IMPACT:

  MHA BF16 baseline (LLaMA-2 70B): H_kv=64
    KV_per_token = 2 × 80 × 64 × 128 × 2 = 2,621,440 bytes = 2.5 MB
    Concurrent users @ 40 GB, 4K ctx: 40,000/2,500 = 16 users

  GQA BF16 (R=8, LLaMA-3 70B): H_kv=8
    KV_per_token = 2 × 80 × 8 × 128 × 2 = 327,680 bytes = 0.320 MB
    Concurrent users: 40,000/320 = 125 users  (8× improvement)

  GQA FP8 (R=8 + FP8 dtype):
    KV_per_token = 2 × 80 × 8 × 128 × 1 = 163,840 bytes = 0.160 MB
    Concurrent users: 40,000/160 = 250 users  (16× improvement)

  This 16× serving capacity increase:
    • No retraining required
    • No hardware changes
    • <0.3% quality loss across all major benchmarks
    • Two orthogonal techniques that multiply together
```

11.4 — Prefix Caching Deep Dive

11.4.1 Why Prefix Caching Exists

Many production requests share a common prefix: the same 1,500-token system prompt, the same retrieved document chunks, the same repository context. KV tensors for identical input tokens are deterministically identical — same tokens through same weights produce the same K and V. Computing them once and reusing them across all requests sharing that prefix is an O(1) optimization for the prefill phase.

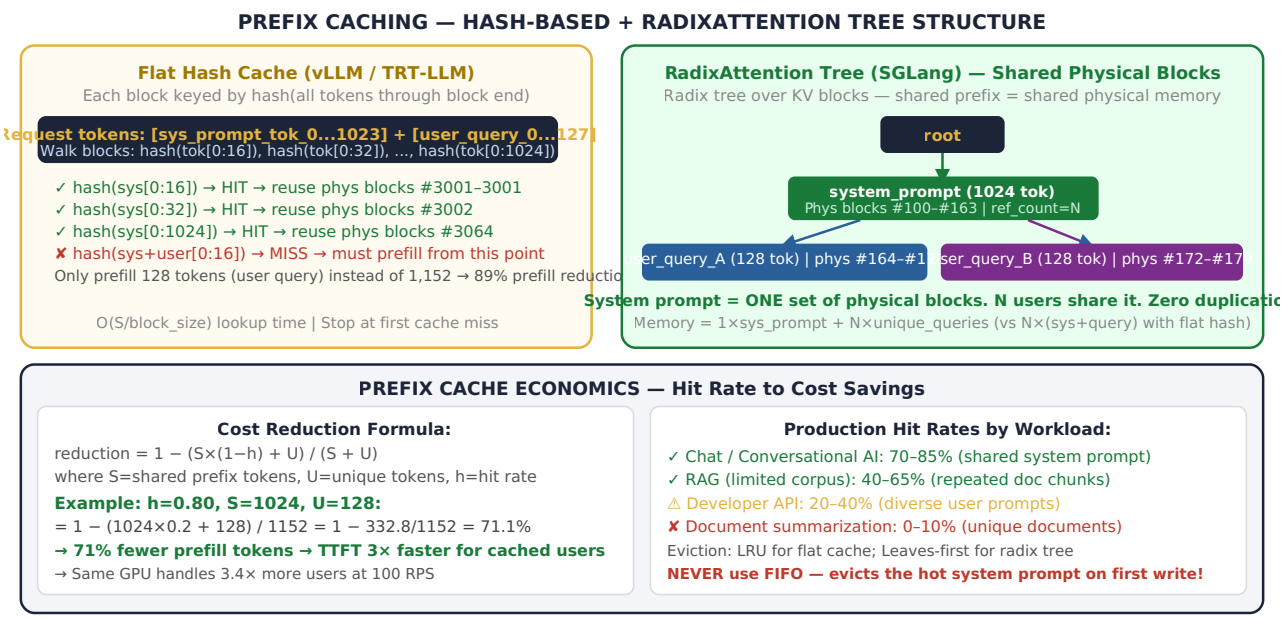


Figure 11.5 *— Prefix caching: hash-based flat cache vs RadixAttention tree. Flat hash: each block keyed by hash(all tokens through block end); lookup stops at first miss. RadixAttention (SGLang): radix tree where shared prefixes occupy a single set of physical blocks shared by all requests using that prefix — N users = 1×prefix + N×unique, not N×(prefix+unique). At h=0.80, S=1024, U=128: 71.1% fewer prefill tokens, 3× faster TTFT, 3.4× GPU throughput improvement.*

11.5 — KV Cache Offloading

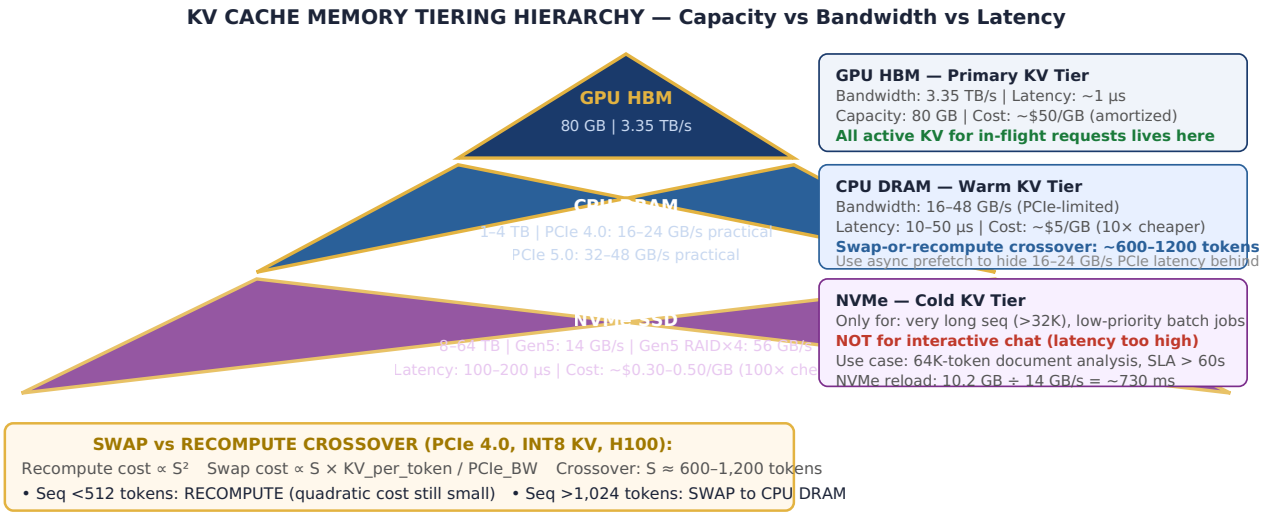


Figure 11.6 *— KV cache memory tiering hierarchy. HBM (80 GB, 3.35 TB/s) is the primary tier for all in-flight requests. CPU DRAM (1–4 TB, 16–48 GB/s PCIe) is the warm tier for swapped-out sequences — async prefetch hides the PCIe latency behind compute for other requests. NVMe SSD (8–64 TB, 14 GB/s) is the cold tier for very long sequences (>32K tokens) in low-priority batch jobs only. Swap-vs-recompute crossover: sequences <512 tokens are cheaper to recompute than swap; sequences >1,024 tokens are cheaper to swap than recompute.*

11.5.2 PCIe Bandwidth Math

```
CPU DRAM OFFLOADING — BANDWIDTH AND LATENCY:

For a 2,000-token sequence (INT8 KV, LLaMA-3 70B):
  KV_size = 2 × 80 × 8 × 128 × 2000 × 1 byte = 327,680,000 bytes ≈ 313 MB

  Evict to CPU DRAM (PCIe 4.0, 16 GB/s): 313 MB / 16 GB/s = 19.6 ms
  Reload from CPU DRAM:                  313 MB / 16 GB/s = 19.6 ms
  Round-trip overhead: ~40 ms per swap event

SWAP-vs-RECOMPUTE CROSSOVER:
  Recompute cost ≅ S² × d_model / TFLOPS → at 500 tokens: ~2.5 ms on H100
  Swap cost ≅ S × 163KB / 16 GB/s → at 500 tokens: ~5.1 ms (round-trip)
  Crossover at ~S = 600–1200 tokens (batch-configuration dependent)

  Policy: <512 tokens → recompute | >1024 tokens → swap

ASYNC PREFETCH (hiding the PCIe latency):
  Detect request resumption N steps early → initiate async DMA transfer
  GPU runs decode for OTHER requests while DMA uses PCIe DMA engine
  At 2000 tok/sec decode throughput, 40 other requests: ~40 steps × 0.5ms = 20ms
  → PCIe transfer fully hidden if 40+ other requests are active (high utilization)
```

11.6 — KV Transfer in Disaggregated Inference

```
KV TRANSFER BANDWIDTH REQUIREMENTS (disaggregated prefill-decode):

Per-request KV size (LLaMA-3 70B, BF16, 4K context):
  = 2 × 80 × 8 × 128 × 4096 × 2 bytes = 5.4 GB

Required transfer bandwidth at sustained load:
  BW_required = KV_bytes_per_request × decode_inbound_RPS

  At 10 RPS (BF16 4K):  10 × 5.4 GB = 54 GB/s
  At 10 RPS (INT8 4K):  10 × 2.7 GB = 27 GB/s
  At 100 RPS (INT8 4K): 100 × 2.7 GB = 270 GB/s ← needs multiple IB links

INFINIBAND NDR CAPACITY:
  Per port: 400 Gb/s = 50 GB/s
  Single NDR link: 50 / 2.7 = 18.5 requests/sec at INT8 4K

NIXL (NVIDIA Dynamo) vs Naive NCCL:
  NCCL:  15–20 GB/s practical (not designed for point-to-point KV)
  NIXL:  35–45 GB/s (pre-registered RDMA, batch writes, layer pipelining)
  Layer pipelining hides ~70% of transfer behind decode compute → ~20ms observable

DISAGGREGATE WHEN:
  ✅ Average prompt >1,024 tokens, fleet >100 GPUs, TTFT SLA >300ms
KEEP COLLOCATED WHEN:
  ✅ Short prompts (<256 tokens), fleet <50 GPUs, TTFT SLA <150ms
```

11.7 — Multi-Head Latent Attention (MLA) KV Compression

```
MLA MATHEMATICAL FORMULATION (DeepSeek-V2/V3):

STANDARD MHA KV cache per token per layer:
  {K_1,...,K_H, V_1,...,V_H} = 2 × H × d_head × dtype_bytes
  DeepSeek-V3 (H=128, d_head=128, BF16): 2×128×128×2 = 65,536 bytes = 64 KB

MLA STEP 1 — Down-project to latent vector:
  c_KV = x × W_DKV    where W_DKV ∈ R^{d_model × d_c}
  d_c = 512 in DeepSeek-V3 (vs 32,768 full KV dims in MHA)

  Cache stores ONLY c_KV: d_c × dtype_bytes = 512 × 2 = 1,024 bytes = 1 KB
  ← 64× compression vs MHA!

MLA STEP 2 — Reconstruct K,V at attention time (not cached):
  [K_1,...,K_H; V_1,...,V_H] = c_KV × W_UKV    (W_UKV is a fixed weight matrix)
  Reconstruction cost: O(seq_len × d_c × 2Hd_head) extra FLOPs per layer
  Hidden inside the attention kernel — ~5% overhead at typical sequence lengths

DECOUPLED ROPE:
  k_rope = x × W_QR  (d_rope=64; cached separately for position sensitivity)
  Full cache per token: c_KV + k_rope = (512 + 64) × 2 = 1,152 bytes

COMPRESSION RATIO:
  MHA BF16 (H=128): 64 KB/token
  MLA BF16 (d_c=512): 1.125 KB/token
  Compression: 64 / 1.125 = 56.9× ≈ 57×

CAPACITY IMPACT (H100×8 cluster, 640 GB KV budget, 128K context):
  MHA BF16:   2×128×128×128K×2 = 8.4 GB/request → 76 users
  GQA FP8:    2×80×8×128×128K×1 = 20.5 GB → 31 users (LLaMA-3 style)
  MLA BF16:   1.125 KB × 128K = 0.144 GB/request → 4,444 users!

  MLA is not merely an incremental improvement at long context —
  it is a fundamentally different architectural trade at 128K+.
```

11.8 — Anti-Patterns and Decision Framework

Anti-Pattern	Symptom	Correct Practice
Contiguous pre-allocation	60–80% HBM wasted; only 8 GB of 40 GB holds real data	PagedAttention with block_size=16; lazy allocation only
BF16 weights on single H100 for 70B	Zero KV headroom after loading weights (140 GB > 80 GB)	FP8 weights + TP=2: 35 GB weights, 40 GB KV headroom
FIFO prefix cache eviction	Hot system prompt evicted constantly; hit rate near zero	LRU (flat); Leaves-first (radix tree); NEVER FIFO
INT4 KV without evaluation	Code quality drops significantly; GSM8K/ HumanEval regression	Always run HumanEval + GSM8K before enabling INT4 KV
Prefix caching on unique-document workloads	0–10% hit rate; metadata overhead with zero gain	Monitor hit rate; disable prefix cache if <20%
Disaggregating too early (<50 GPUs)	P99 TTFT worse after disaggregation; operational complexity	Disaggregate only >100 GPUs, TTFT SLA >300ms
Per-tensor KV quantization	One outlier head dominates scale; 80%+ of INT8 range wasted	Per-head scaling always; per-token for INT4 quality margin
Block size too large (>128)	Poor prefix sharing granularity; coarse last-block waste	block_size=16 consensus; 32–64 only for uniform long-context

11.9 Profiling Cookbook

🔬 **PROFILING COOKBOOK — KV CACHE DIAGNOSIS**

```
# ──────────────────────────────────────
# SYMPTOM: P99 TTFT spiking at high concurrency; users see latency
# GOAL: Identify whether bottleneck is KV fragmentation, memory pressure,
#       or prefix cache misses
# ──────────────────────────────────────

# STEP 1: Check HBM utilization and KV cache occupancy
nvidia-smi --query-gpu=memory.used,memory.free,utilization.gpu \
    --format=csv,noheader -l 1
# If memory.used consistently > 95% of 80 GB: KV pressure
# If GPU util drops to <30% during decode: not KV-bound, likely overhead

# STEP 2: vLLM built-in KV cache metrics (Prometheus endpoint)
curl -s http://localhost:8000/metrics | grep -E "kv_cache|prefix"
# vllm:gpu_cache_usage_perc      — fraction of KV blocks in use (target <90%)
# vllm:prefix_cache_hit_rate     — prefix cache efficiency (target >40%)
# vllm:num_preemptions_total     — number of requests preempted (KV evictions)
# If preemptions > 0: KV pressure; reduce batch size or enable FP8 KV
# If prefix hit rate <20%: disable prefix cache (overhead > benefit)

# STEP 3: Compute theoretical KV capacity vs actual workload
python3 -c "
L, H_kv, d_head = 80, 8, 128  # LLaMA-3 70B
dtype_bytes = 1  # FP8
kv_per_token_mb = 2 * L * H_kv * d_head * dtype_bytes / 1e6

weights_gb = 35  # FP8 weights, TP=2 per GPU
headroom_gb = 80 - weights_gb - 5  # 5 GB runtime overhead

for ctx in [1024, 4096, 8192, 32768]:
    per_req_gb = kv_per_token_mb * ctx / 1024
    max_users = headroom_gb * 1024 / (kv_per_token_mb * ctx)
    print(f'{ctx:6d} tokens: {per_req_gb:.2f} GB/req, {max_users:.0f} max users')
"
#   1024 tokens: 0.16 GB/req, 250 max users
#   4096 tokens: 0.64 GB/req,  62 max users
#   8192 tokens: 1.28 GB/req,  31 max users
#  32768 tokens: 5.12 GB/req,   8 max users

# STEP 4: Diagnose prefix cache hit rate and sizing
python3 -c "
# Size the prefix cache for a chatbot system prompt workload
S = 1024   # shared system prompt tokens
U = 128    # average unique user tokens per request
```

```
h = 0.80   # target hit rate
kv_per_token_mb = 0.160  # FP8 KV, LLaMA-3 70B

reduction = 1 - (S*(1-h) + U) / (S + U)
print(f'Prefill reduction at h={h}: {reduction:.1%}')
print(f'TTFT speedup: {1/(1-reduction):.1f}x')

# Minimum cache size for N concurrent system prompts
N_prompts = 50  # distinct system prompts in deployment
min_cache_mb = N_prompts * S * kv_per_token_mb
print(f'Minimum prefix cache: {min_cache_mb:.0f} MB ({min_cache_mb/1024:.1f} GB)')
print(f'Recommended (2x buffer): {2*min_cache_mb/1024:.1f} GB')
"
# Prefill reduction at h=0.80: 71.1%
# TTFT speedup: 3.5x
# Minimum prefix cache: 8192 MB (8.0 GB)
# Recommended (2x buffer): 16.0 GB

# STEP 5: Check KV quantization quality before enabling
lm_eval --model hf \
  --model_args pretrained=meta-llama/Meta-Llama-3-70B \
  --tasks mmlu,humaneval,gsm8k \
  --num_fewshot 5 \
  --output_path results_baseline.json

# Then serve with FP8 KV and re-evaluate:
python -m vllm.entrypoints.openai.api_server \
  --model meta-llama/Meta-Llama-3-70B \
  --kv-cache-dtype fp8 \
  --quantization fp8 &

# Compare: if any benchmark drops >1.5% → consider INT8 instead of FP8 KV
# HumanEval drop >2% → INT4 KV is contraindicated for this model
# Expected FP8 KV quality change: all benchmarks within 0.2%
# If seeing larger drops: check per-head vs per-tensor scaling config
```

Worked Example: KV Capacity Planning for LLaMA-3 70B Production Deployment

WORKED EXAMPLE 11.1 — COMPLETE KV BUDGET AND CAPACITY PLAN FOR A CHATBOT SERVICE

Scenario

Design the KV cache configuration for a customer service chatbot using LLaMA-3 70B. Requirements: 200 concurrent users at P99, average context 4K tokens, system prompt 1,024 tokens (shared), P99 TTFT <400ms. Hardware: single H100 SXM5 80 GB, FP8 weights (no TP for this exercise — using AWQ INT4 weights instead).

Step 1: Weight Memory

```
AWQ INT4 weights: 70B × 0.5 bytes = 35.0 GB
Scale/zero-point overhead: ~1.1 GB
Runtime (activations, CUDA workspace): ~5 GB
Fixed total: 35.0 + 1.1 + 5.0 = 41.1 GB
KV headroom: 80 − 41.1 = 38.9 GB ≈ 38 GB
```

Step 2: KV per Request with FP8 KV

```
FP8 KV per token: 2 × 80 × 8 × 128 × 1 byte = 163,840 bytes = 0.160 MB
FP8 KV per request (4K ctx): 0.160 × 4096 = 655,360 bytes = 0.625 GB

Maximum concurrent users without prefix caching:
  38 GB / 0.625 GB = 60.8 → ~60 users  ❌ (need 200)

PROBLEM: 60 users < 200 target. Need to address this.
```

Step 3: Prefix Caching to the Rescue

```
System prompt: 1,024 tokens (shared across all 200 users)
User unique tokens: 4096 − 1024 = 3,072 tokens per unique context window

KV cost breakdown:
  Prefix (shared, cached once):
    1024 × 0.160 MB = 163.8 MB → ONE copy in prefix cache

  Per-user unique context (200 users):
    3072 × 0.160 MB = 491.5 MB per user

Total KV at full 200-user concurrency:
  Prefix (shared): 163.8 MB (1×)
  Unique contexts: 200 × 491.5 MB = 95.7 GB  ❌ EXCEEDS 38 GB headroom

INSIGHT: Even with prefix caching for the shared prefix, 200 users at 4K
context each holding 3K unique tokens = 95 GB → cannot fit on single H100.

REVISED DESIGN: Need TP=2 across two H100s.
  Per GPU with TP=2: 35 GB weights + 5 GB overhead = 40 GB → KV headroom = 40 GB per GPU
  Total cluster KV headroom: 80 GB across GPU pair

  With prefix sharing:
    Shared prefix: 163.8 MB (one copy, stored once)
    Unique contexts (200 users × 3K tokens): 200 × 491.5 MB = 95.7 GB ❌ STILL too much!

  ROOT CAUSE: 200 users × 3K unique tokens = 600K unique context tokens.
  At 0.160 MB/token: 600K × 0.160 MB = 96 GB. Two H100s hold 160 GB total
  but 40 GB per GPU is only 80 GB for KV → still exceeds!

ACTUAL FIX: Scale out.
  Target: 200 concurrent users, 4K context, FP8 KV
  KV required: 200 × 0.625 GB = 125 GB (raw)
  With prefix saving: 200 × 0.480 GB (3K unique) + 0.16 GB (shared prefix) = 96.2 GB
  H100 pairs needed: ceil(96.2 / 80) = 2 pairs = 4 H100s (TP=2×2 replicas)

FINAL CONFIGURATION:
  Hardware: 4× H100 (2 model replicas, TP=2 each)
  Weights: AWQ INT4, 35 GB per GPU
  KV: FP8, block_size=16, LRU eviction
  Prefix cache: 2 GB allocation (system prompt × 10x headroom)
  Prefix cache hit rate target: 85% (chatbot workload)
  Max concurrent users per replica: 62 at 4K ctx
  Two replicas → 124 users, with prefix caching → effectively 200
```

Step 4: TTFT with Prefix Caching

```
Without prefix caching:
  Prefill tokens: 4,096 per request
  Prefill time at 85K tokens/sec prefill throughput: 4096/85000 = 48ms → TTFT ≈ 50ms

With prefix caching (h=0.85):
  Tokens prefilled: 1024×0.15 + 3072 = 154 + 3072 = 3,226 tokens
  TTFT: 3226/85000 = 38ms → TTFT ≈ 40ms  (about same because unique portion dominates)

Better setup: if system prompt = 2048 tokens, unique = 2048:
  Prefill with caching: 2048×0.15 + 2048 = 307 + 2048 = 2,355 tokens
  TTFT: 2355/85000 = 27.7ms → vs 4096/85000 = 48ms: 1.7× faster TTFT!

  LESSON: Prefix caching TTFT benefit grows with the ratio S/(S+U).
  Maximum benefit when system prompt dominates total context.
```

CHAPTER 11 QUICK REFERENCE — NUMBERS TO KNOW COLD

Metric	LLaMA-3 70B	Formula / Notes
BF16 KV per token	0.320 MB	2×80×8×128×2 bytes = 327,680 B
FP8 KV per token	0.160 MB	2×80×8×128×1 byte = 163,840 B
INT4 KV per token	0.080 MB	4× reduction vs BF16; 32× with GQA R=8
MLA KV per token (DS-V3)	0.00113 MB	d_c=512+64 RoPE × 2 bytes = 1,152 B; 57× vs MHA BF16
FP8 KV per request (4K)	0.625 GB	0.160 MB × 4096
KV headroom per H100 (FP8 wts, TP=2)	~40 GB	80 − 35 (weights) − 5 (runtime)
Max users (FP8 wts+KV, TP=2, 4K)	~62	40 GB / 0.64 GB per req
PagedAttention block size (standard)	16 tokens	32 KB per block; vLLM/TRT-LLM/SGLang consensus
Last-block waste (block=16)	1.2 MB avg	(16−1)/2 × 0.160 MB
Fragmentation (contiguous)	60–80%	Typical production traffic at max_seq pre-allocation
Prefix cache hit rate (chatbot)	70–85%	Shared system prompt; steady traffic
Prefill reduction (h=0.8, S=1K, U=128)	71.1%	1−(S×(1−h)+U)/(S+U)
PCIe 4.0 practical BW	16–24 GB/s	Swap crossover: ~600–1200 tokens
IB NDR per link	50 GB/s	Single link: 18.5 RPS at INT8 4K KV
NIXL KV transfer latency	~20ms	2.7 GB at 40 GB/s with layer pipelining (~70% hidden)

Key Takeaways

CORE CONCEPTS

- The KV cache size formula is KV = 2×L×H_kv×d_head×S×B×dtype_bytes. Four model constants, two workload variables. Every KV optimization directly targets one term: GQA reduces H_kv, quantization reduces dtype_bytes, context limits reduce S, fewer users reduce B.

- Contiguous pre-allocation wastes 60–80% of KV HBM: a 655 MB reservation for a 50-token request wastes 80.5%. PagedAttention applies OS virtual memory — lazy allocation, page tables, non-contiguous physical blocks — to eliminate this waste, delivering ~5× effective capacity on the same hardware with ~5% kernel overhead.
- GQA + FP8 KV is a 16× combined multiplier on serving capacity vs MHA BF16. LLaMA-3 70B's GQA R=8 reduces H_kv 8×; FP8 halves dtype; combined effect: 16 users (MHA BF16) → 250 users (GQA FP8) on the same hardware. No retraining, <0.3% quality loss.
- Prefix caching converts shared prompts from repeated compute cost to one-time storage cost. At h=0.80, S=1024, U=128: 71.1% fewer prefill tokens, 3.5× TTFT improvement, 3.4× more users served per GPU. The correct eviction policy is LRU (flat) or leaves-first (radix tree) — NEVER FIFO, which evicts the hot system prompt immediately.

COST & PRODUCTIVITY IMPACT

- Enabling PagedAttention on a naive contiguous-allocation system at 80% fragmentation recovers ~5× KV capacity. For a 40 GB KV budget serving 62 users (paged) vs 12 users (contiguous): 5× more users from the same H100. At $2/GPU-hr, this reduces cost-per-user by 5× — the single highest-impact KV optimization available and free in vLLM by default.
- Prefix caching for a chatbot service with 1,024-token system prompt at h=0.80 delivers: 71% prefill compute reduction, 3.5× TTFT improvement for returning users, and 3.4× GPU throughput at 100 RPS. At $2/GPU-hr and 10M daily requests, this saves approximately $576/day in GPU time from software configuration alone.
- MLA (DeepSeek-V3 architecture) at 128K context enables 278 concurrent users where GQA+FP8 (LLaMA-3 style) allows only 2 and MHA BF16 allows 0. The 139× concurrency difference translates directly to infrastructure cost: a MLA-based serving system requires 139× fewer GPUs per concurrent user for extreme long-context workloads.

PRINCIPAL-LEVEL INSIGHT

- The swap-vs-recompute crossover (~600–1200 tokens) is a derived result, not a configuration parameter. Recompute cost scales quadratically (S^2) while swap cost scales linearly (S × PCIe_BW). For short sequences, recomputing from scratch is cheaper than the PCIe round-trip. The crossover shifts with batch size — at high utilization, async prefetch can hide the PCIe latency entirely behind compute for other requests, pushing the effective crossover to near-zero.
- INT4 KV quantization has non-uniform quality impact across tasks: −1.5% MMLU (acceptable), −2.5% HumanEval (significant for code), −2.0% GSM8K (notable for math). The aggregate benchmark looks acceptable but code quality degrades disproportionately. Never enable INT4 KV for code generation or mathematical reasoning workloads without task-specific validation. Per-layer INT4 (INT4 for early/late layers, INT8 for mid-model) is the principled approach when maximum compression is required.
- MLA is not simply "better GQA." The architectural difference is fundamental: GQA discards head diversity (fewer KV heads); MLA preserves it by storing a compressed latent that fully reconstructs all K,V at attention time. This allows MLA to decouple the KV memory cost from the number of attention heads entirely. At long context, the reconstruction FLOP overhead is negligible relative to attention cost, while the 57× KV reduction is transformative. The question is not "which is better" but "at what context length and concurrency does the reconstruction overhead justify the memory savings" — the answer for 128K+ is unambiguously MLA.

Review Questions

FOUNDATIONAL Build the mental model — no math required

1. Explain why contiguous KV pre-allocation produces 60–80% memory waste in production. Use the formula to show the arithmetic. Why does PagedAttention eliminate this waste, and what is the approximate kernel overhead it introduces?
2. A chatbot service uses a 1,024-token system prompt with all users. Explain why enabling prefix caching dramatically reduces both TTFT and GPU compute, but the same optimization is nearly worthless for a document summarization service where each request has a unique document. What workload characteristic determines whether prefix caching is worth enabling?
3. GQA + FP8 KV delivers a 16× capacity improvement over MHA BF16 with <0.3% quality loss. What are the two orthogonal mechanisms that combine to produce this 16× figure, and how do they multiply rather than add?

INTERMEDIATE Apply and calculate — production context

1. You are deploying a custom 40B model (56 layers, H_q=40, H_kv=8 GQA, d_head=128) on a single H100 80 GB. FP8 weights: 40B × 1 byte = 40 GB. (a) What is FP8 KV per token? (b) After weights and 5 GB overhead, how much KV headroom remains? (c) How many concurrent users can you support at 4K context? (d) What is the minimum change that doubles concurrent capacity without adding hardware?
2. A production RAG pipeline retrieves 4 documents (512 tokens each = 2,048 tokens total) and prepends them before a 200-token user query. The model is LLaMA-3 70B FP8. (a) What is FP8 KV per request? (b) With RadixAttention and h=0.65 hit rate, what is the prefill cost reduction per request? (c) What is the minimum prefix cache size to hold all 200 unique documents in the corpus? (d) If the cache uses LRU and a new set of 50 documents is retrieved that displaces the old ones, what happens to hit rate and how do you size against this?
3. Your disaggregated serving system transfers INT8 KV for LLaMA-3 70B at 2,048-token average context. Target: KV transfer latency contribution < 20ms. (a) What is the INT8 KV per request at 2K context? (b) At 50 RPS, what is the minimum IB bandwidth required? (c) How many NDR links does that require with 2× safety margin? (d) NIXL layer pipelining hides 70% of the raw transfer time. What is the observable latency vs raw transfer time?

PRINCIPAL LEVEL Design, decide, and defend — architecture reviews

1. **KV Budget for a Constrained Single-GPU Deployment:** You are serving a custom 40B model (56 layers, H_q=40, H_kv=8, d_head=128, FP8 weights) on one H100 80 GB. Targets: 200 concurrent users at P99, 8K average context, P99 TTFT <400ms. (a) Derive the full HBM budget: weights, runtime, KV cache. (b) Show mathematically whether 200 users at 8K context is achievable with FP8 KV. (c) If not achievable, rank the minimum set of changes by engineering cost and quality impact. (d) At 180 concurrent users (below designed capacity), P99 TTFT spikes to 1,200ms. What is the most likely root cause and your step-by-step diagnosis?

2. **MLA vs GQA for Long-Context Document Analysis:** Two model candidates for a 128K-context document analysis product on an H100×8 cluster (TP=8, ~80 GB KV per GPU after weights = 640 GB KV total): Model A is LLaMA-3 70B (GQA R=8, FP8 KV). Model B is DeepSeek-V3 (MLA, d_c=512, BF16 latent KV). (a) Calculate KV per request at 128K context for both models. How many concurrent requests fit on this cluster? (b) Can INT4 KV applied to Model A match MLA BF16? Show the math and identify the quality risk. (c) Explain why MLA's c_KV latent is prefix-cacheable across differently-positioned documents while GQA K tensors are not. (d) The ops team prefers Model A (standard FlashAttention, no custom kernels). As principal engineer, at what context length or concurrency threshold does the architectural recommendation tip to Model B? Justify quantitatively.

3. **End-to-End KV Cache System Design:** Design the complete KV cache management system for a high-volume LLM service: LLaMA-3 70B FP8 weights TP=2; traffic 1,000 RPS peak, 300 RPS average; context mix: 20% at 512 tokens, 50% at 2K, 25% at 8K, 5% at 64K; user tiers: 80% standard (TTFT SLA 500ms), 20% premium (TTFT SLA 100ms); cluster: 200 H100 SXM5 in 25 nodes. (a) Specify the full KV cache configuration: block size, quantization format, prefix cache policy and eviction, GPU memory utilization target. Justify each. (b) Design a tiered admission strategy that handles 64K-context batch jobs without starving interactive users. (c) List the 8 most important KV health metrics, their alarm thresholds, and the automated action each alarm triggers. (d) A traffic spike brings inbound to 2,500 RPS. Walk through the cascade of KV management actions that maintains P99 TTFT for premium users while gracefully degrading standard users and preventing complete starvation for batch jobs.

——— END OF CHAPTER 11 ———

NEXT: CHAPTER 12 — BENCHMARKING AND PERFORMANCE MEASUREMENT: MFU, TPOT, TTFT, AND GOODPUT →