PART II · CHAPTER 4

Memory Hierarchy & HBM Deep Dive

The Physical Foundation of Every GPU Performance Decision — From Registers to Clusters, and the Kernel That Changed Everything: Flash Attention v1 → v2 → v3

Book: AI/ML Infrastructure from Silicon to Scale — V2 Edition | **Author:** Venkat Vinjam | **Cross-ref:** Ch 1: Roofline Ch 7: GPU Kernels Ch 11: KV Cache

> *"GPUs don't run out of compute. They run out of memory bandwidth. Every performance optimization in AI infrastructure is, at its core, a battle against the memory wall."*
>
> — Principal Performance Architect's First Principle

This chapter answers one deceptively simple question: *Why is moving data harder than computing with it?* The answer determines whether a kernel runs at 5% or 95% hardware efficiency, whether an inference system serves 10 users or 10,000, and whether your cluster achieves 40% or 80% Model FLOP Utilization (MFU).

4.1 The Memory Wall — Why It Exists and Won't Go Away

Year	GPU	Peak TFLOPS (BF16)	HBM Bandwidth	Compute:BW Ratio
2018	V100	112	900 GB/s	124 FLOP/byte
2020	A100	312	2.0 TB/s	156 FLOP/byte
2022	H100	989	3.35 TB/s	**295 FLOP/byte ← ridge**
2024	B200	~4,500 (FP8)	8.0 TB/s	562 FLOP/byte
2025	GB300	~9,000 (FP4)	12.0 TB/s	750 FLOP/byte

The Widening Gap

Compute grew ~20× from V100→B200 over 6 years. Bandwidth grew only ~9×. The compute:bandwidth ratio widened from 124→562 FLOP/byte. Operations that were compute-bound in 2018 may now be memory-bound on H100. Performance engineers must *constantly re-evaluate* which regime their operations fall into as GPU generations advance.

Why bandwidth is hard to scale: Pin count physics limits each data pin to ~64 Gbps. DRAM refresh overhead consumes 3–5% at room temperature, rising to 8–12% at AI datacenter temperatures. The H100's HBM3 interface consumes ~40–60W just for the memory subsystem before any compute. HBM addresses these constraints by stacking DRAM dies vertically with through-silicon vias (TSVs), enabling 1024-bit buses at lower voltage — dramatically better bandwidth-per-watt and bandwidth-per-area than GDDR memory.

4.2 The Five-Tier GPU Memory Hierarchy

Every performance optimization involves moving data to the fastest tier that can hold it.

TIER 1 Registers

~33 MB total (H100) · 1-cycle latency · unlimited bandwidth

65,536 × 32-bit registers per SM. Critical: register pressure limits occupancy. At 64 regs/thread → 50% occupancy; at 128 regs → 25%. **Register spilling** maps overflow to HBM L2 — turning 1-cycle access into 200–800 cycle access, a 200–800× slowdown.

TIER 2 L1/SMEM

228 KB/SM · ~80–100 TB/s aggregate · 20–30 cycle latency

Explicitly programmer-managed cache shared among threads in a block. The key mechanism for GEMM tiling: without SMEM tiling, every operand access goes to HBM. **Bank conflicts** (32 banks, 4-byte words) can reduce SMEM BW from peak to peak/32. Fix by padding inner dim: `smem[32][33]` instead of `smem[32][32]`.

TIER 3 L2 Cache

50 MB · ~12 TB/s aggregate · 200–400 cycle latency · all SMs share

Hardware-managed. Limited benefit for decode (weights far exceed 50 MB). Useful for KV cache hot-path, embedding table lookups, LayerNorm running stats. L2 hit rate in Nsight Compute indicates streaming vs. reuse patterns.

TIER 4 HBM

80–192 GB per GPU · H100: 3.35 TB/s | H200: 4.8 TB/s | B200: 8.0 TB/s · 400–800 cycle latency

Per-GPU global memory. Holds model weights, activations, KV cache — the primary bottleneck for all memory-bound operations. Achieving >80% utilization requires coalesced access, tiling, and quantization.

TIER 5 Host DRAM

1–2 TB per server · PCIe Gen5: 128 GB/s GPU↔CPU · 10,000–50,000 cycle GPU latency

CPU memory. Dataset staging, checkpoint buffers, model spill. PCIe is 26× slower than HBM. Frequent GPU↔CPU transfers are catastrophic for performance — minimize at all costs.

```
SMEM TILING — Naive GEMM vs Tiled GEMM:

Naive GEMM: For each (i,j) in C[M×N], for each k in K:
  Load A[i,k] from HBM + Load B[k,j] from HBM (2 reads per multiply)
  Total reads M=N=K=4096: 137 billion HBM reads → fully bandwidth-bound

Tiled GEMM (SMEM): Load TILE_M×TILE_K block of A + TILE_K×TILE_N of B
  into SMEM once, compute all inner products from SMEM (fast access).
  TILE=128: HBM reads reduced by 128× · AI rises to 128 FLOP/byte → compute-bound ✓
```

4.3 HBM Architecture Deep Dive

```
HBM3e Physical Architecture (H200):
  GPU Die (compute, TSMC 4nm, 132 SMs)
      | Through-Silicon Vias (TSVs) — 55 µm pitch, 1024 data bits/stack
  HBM3e Stack (×6 for 141 GB):
      ┌─ DRAM Die #4 (top)
      ├─ DRAM Die #3
      ├─ DRAM Die #2
      ├─ DRAM Die #1
      └─ Base Logic Die (signal routing, bottom)

  H100: 5 stacks × 1024-bit = 5120-bit bus → 3.35 TB/s, 80 GB
  H200: 6 stacks × 1024-bit = 6144-bit bus → 4.8 TB/s, 141 GB
  Gaming GDDR6X (RTX 4090): 384-bit bus → 1.0 TB/s (3.4× narrower)

  BW = stacks × bus_width × clock_freq × 2(DDR)
     = 5 × 1024 × 819MHz × 2 = 3.35 TB/s (H100)
```

Generation	HBM2 (V100)	HBM2e (A100)	HBM3 (H100)	HBM3e (H200)	HBM4 (MI450)
BW/stack	256 GB/s	307 GB/s	819 GB/s	1.2 TB/s	~2.4 TB/s
GPU total BW	900 GB/s	2.0 TB/s	3.35 TB/s	4.8 TB/s	**19.6 TB/s**
Capacity/stack	16 GB	16 GB	16 GB	24 GB	~54 GB
Data rate/pin	2 Gbps	3.2 Gbps	6.4 Gbps	9.6 Gbps	~16 Gbps

Why HBM capacity scales slowly: Three constraints limit adding stacks: (1) Silicon interposer area is bounded by reticle limits (~858 mm²). (2) TSV yield — thousands of 55 µm-pitch vias per stack; defect rates multiply with taller stacks. (3) Thermal extraction — each stack dissipates 5–10W internally, running 15–25°C hotter than the GPU die. These drive ~2× capacity growth per generation vs. ~3× compute growth.

Access Pattern	Achieved BW (H100)	Utilization
Sequential streaming (memcpy)	3.0–3.1 TB/s	**90–92%**
Flash Attention v3	2.7–2.9 TB/s	**80–87%**
GEMM weight loading (large batch)	2.7–2.9 TB/s	**80–87%**
GEMV (decode, matrix-vector, batch=1)	2.2–2.6 TB/s	65–78%
Scatter/gather ops	0.8–1.5 TB/s	**24–45%**
Standard attention (N=4096)	~0.8 TB/s	**24%**
Uncoalesced warp access	0.1–0.4 TB/s	**3–12%**

Coalesced vs. Uncoalesced Access — The #1 Pattern Decision

When 32 threads access 32 consecutive 4-byte addresses (128 bytes = 1 cache line): ONE HBM transaction → 100% efficiency.
When 32 threads access 32 different cache lines: up to 32 separate transactions → 3% efficiency — a 32× overhead.

Fix: Ensure thread `i` accesses `base + i × element_size`. Avoid column-major traversal of row-major arrays. For GEMM B^T: store B transposed so both A and B^T are row-traversed.

4.4 The Arithmetic Intensity Framework

Arithmetic Intensity (AI) Definition

`AI = FLOPs executed / Bytes transferred to/from HBM` [units: FLOP/byte]

This single number determines whether any operation is memory-bound or compute-bound on any given hardware.

```
KEY LLM OPERATIONS — ARITHMETIC INTENSITY:

GEMM (prefill): C[M×N] = A[M×K] × B[K×N]
  FLOPs: 2×M×N×K · Bytes: (M×K + K×N + M×N) × dtype_bytes
  M=N=K=4096, BF16: FLOPs=137.4G · Bytes=100.7M → AI=1,365 FLOP/byte
  H100 ridge=295 → compute-bound ✓ (This is why large-batch prefill is fast)

GEMV (decode, batch=1): y[M] = A[M×K] × x[K]
  FLOPs: 2×M×K · Bytes: M×K × dtype_bytes (load ENTIRE weight matrix!)
  M=K=8192, BF16: FLOPs=134.2M · Bytes=134.2M → AI=1.0 FLOP/byte
  H100 ridge=295 → DEEPLY MEMORY BOUND ✗ (decode loads weights for 1 token)

RMSNorm / LayerNorm: AI ≈ 1.25 FLOP/byte → memory-bound
  Practical throughput: HBM_BW/4 = 3.35TB/s/4 ≈ 840 GB/s of elements
  → Standalone normalization wastes 99.6% of compute capacity
  → Operator fusion with adjacent GEMM is MANDATORY

Softmax (attention row, standard): AI ≈ 5 FLOP/byte → memory-bound
FlashAttention QK matmul (S=2K): AI ≈ 250 FLOP/byte → near-ridge
Attention GEMM S=4096: AI ≈ 2048 FLOP/byte → compute-bound

Roofline rule (H100 BF16, ridge=295):
  If AI < 295: achievable TFLOPS = AI × 3.35
  If AI > 295: achievable TFLOPS = 989 (compute ceiling)
```

4.5 Memory Bandwidth Optimization Techniques

4.5.1 Operator Fusion

```
UNFUSED: y = LayerNorm(GELU(Linear(x)))
  Linear:   read x[HBM] → GEMM → write z[HBM]        16 GB
  GELU:     read z[HBM] → elementwise → write g[HBM]  32 GB
  LayerNorm:read g[HBM] → reduce → write y[HBM]       32 GB
  Total: 80 GB traffic for 16 GB of data = 5× overhead

FUSED: Single kernel
  Read x → GEMM(SMEM) → GELU(registers) → LayerNorm(registers) → write y
  Total: 32 GB = 2× overhead · Speedup: ~2.5× on memory-bound workloads

Key LLM fusion targets:
  Flash Attention:  QKᵀ + softmax + mask + ×V → single kernel (most impactful)
  MLP:              GEMM + GELU/SiLU (+ optional GEMM2)
  Residual+Norm:    add_residual + RMSNorm → merged (vLLM, TRT-LLM default)
  QKV projection:   3 linear projections → 1 wider GEMM
```

4.5.2 Quantization — Reducing Bytes Per Parameter

Format	Bits	70B Weight Size	Decode Latency (H100)	Quality Loss
FP32	32	280 GB	**83ms/tok — impractical**	None
BF16/FP16	16	140 GB	42ms/tok (baseline)	Baseline
FP8 E4M3	8	70 GB	**21ms/tok (2×)**	~0.1%
INT4 AWQ	4	35 GB	**10ms/tok (4×)**	~0.3%
NVFP4	4	35 GB	**~8ms/tok (Blackwell)**	~0.5%

4.6 Memory Across the Full System

```
DGX H100 Memory Topology (8 GPUs):
  8 × 80 GB HBM3 = 640 GB total · GPU-GPU: 900 GB/s NVSwitch · GPU-CPU: 128 GB/s PCIe

Parallelism constraints driven by memory bandwidth:
  Tensor Parallel (TP): AllReduce per layer → needs 900 GB/s → NVLink ONLY
    80 layers × 0.5ms/AllReduce over IB = 40ms/fwd pass → UNUSABLE
    80 layers × 0.05ms over NVLink = 4ms/fwd pass → acceptable
    Constraint: TP group MUST stay within NVSwitch domain (≤8 GPUs)

  Pipeline Parallel (PP): point-to-point activations → 50-100 GB/s → IB OK
    Creates pipeline bubbles ∝ PP degree; use when model exceeds single-node HBM

  Data Parallel (DP/FSDP): gradient AllReduce → 100-400 GB/s → IB OK
    FSDP shards model weights across data-parallel ranks (ZeRO-3 equivalent)
```

4.7 Memory Profiling and Diagnosis

```
Nsight Compute sections for memory analysis:

1. "Memory Workload Analysis": HBM util%, L1/L2 hit rates, SMEM util%
   Target: >80% HBM util for memory-bound; >50% SMEM for tiled kernels

2. "Memory Chart": SMEM BW >> HBM BW = good tiling ✓
                   HBM BW >> SMEM BW = poor tiling ✗

3. "Source Counters": per-line HBM traffic → diagnose uncoalesced access

4. "Roofline": auto-plots kernel, shows memory vs compute bound

NCU command:
  ncu --set full --section MemoryWorkloadAnalysis \
      --section SpeedOfLight_RooflineChart python train.py

AMD MI300X: omniperf profile --name run -- python train.py
            omniperf analyze --path run/ --roof-only

Key metrics and healthy ranges:
  HBM BW Utilization: >70% (mem-bound) · <40% → check if compute-bound
  L2 Hit Rate: 30-80% · <10% normal streaming; >90% excellent locality
  Bank Conflicts/SMEM Access: 0 · Any → pad SMEM dim (smem[32][33])
  Register Spills to Local: 0 · Any → reduce regs/thread
  Uncoalesced Global Accesses: 0 · Any → fix thread-to-address mapping
  Achieved Occupancy: >60% · <33% → register or SMEM pressure
```

4.8 Flash Attention — A Complete Deep Dive

> *"Flash Attention is not a clever trick. It is the correct way to implement attention. Everything else is wrong in proportion to how much memory it wastes."*
>
> — AI Systems Engineering Perspective

Flash Attention is the single most impactful GPU kernel innovation of the LLM era. Every major production system — vLLM, TRT-LLM, Megatron-LM, HuggingFace Transformers — uses it by default.

4.8.1 The Problem: Standard Attention Is $O(N^2)$ in Memory

```
STANDARD ATTENTION HBM TRAFFIC (Q,K,V each [N×d]):

Step 1: S = Q × Kᵀ / √d  → S [N×N]   WRITE to HBM
Step 2: P = softmax(S)    → P [N×N]   READ + WRITE to HBM
Step 3: O = P × V         → O [N×d]   READ + WRITE to HBM

Total: 4×N×d + 3×N² bytes (BF16: ×2)

For N=4096, d=128, BF16:
  N×d terms: 4 × 4096 × 128 × 2 =   4.2 MB
  N² terms:  3 × 4096² × 2     = 100.7 MB ← DOMINATES
  Total per head: ~105 MB

For LLaMA-3 70B (80 layers × 64 MHA heads):
  80 × 64 × 105 MB = 537 GB HBM traffic for attention alone!
  At H100 3.35 TB/s: 0.16 seconds minimum per fwd pass (memory only!)

CONTEXT LENGTH SCALING (why long contexts need FA):
  N=8192:  N² = 134 MB/head   → problematic
  N=32768: N² = 2.1 GB/head   → can't fit in 80 GB HBM
  N=128K:  N² = 32 GB/head    → physically impossible
  → Standard attention CANNOT support long contexts
```

4.8.2 The Key Insight: Online Softmax

Standard softmax for row i requires ALL j values simultaneously — the denominator requires the full sum. Online softmax (Milakov & Gimelshein, 2018) maintains running statistics correctable tile-by-tile.

```
ONLINE SOFTMAX TRICK:
For row x processed in tiles [x₁..xₖ] then [xₖ₊₁..xₙ]:

AFTER TILE 1:
  m₁ = max(x₁..xₖ)                    ← running maximum
  ℓ₁ = Σ exp(xⱼ - m₁)                 ← partial denominator (stable)
  O₁ = Σ exp(xⱼ - m₁) × Vⱼ / ℓ₁       ← partial output

AFTER TILE 2 (with correction):
  m₂ = max(m₁, max(xₖ₊₁..xₙ))         ← global max update
  α  = exp(m₁ - m₂)                   ← correction factor for old partial sum
  ℓ₂ = α × ℓ₁ + Σ exp(xⱼ - m₂)        ← updated denominator
  O₂ = (α × ℓ₁ × O₁ + Σ exp(xⱼ - m₂) × Vⱼ) / ℓ₂

RESULT: O₂ is MATHEMATICALLY IDENTICAL to softmax([x₁..xₙ]) × [V₁..Vₙ]
        — computed WITHOUT ever materializing the N×N matrix!

NUMERICAL STABILITY:
  Naive exp(1000) = inf (overflow), exp(-1000) = 0 (underflow)
  Online softmax subtracts running max m: keeps all values in [exp(-large), 1]
  This stability property is preserved across tiles via the correction factor α.
```

4.8.3 Flash Attention v1 — The Tiling Algorithm

FAv1 *Dao et al., NeurIPS 2022*

```
FAv1 SRAM BUDGET (H100: 228 KB SMEM per SM):
  Q tile: 64×128×2 = 16 KB  |  K tile: 16 KB  |  V tile: 16 KB  |  O tile: 16 KB
  m,ℓ vectors: ~0.5 KB  |  TOTAL: 64.5 KB → fits in 228 KB ✓

ALGORITHM:
  Init: O=zeros(N,d), ℓ=zeros(N), m=-inf(N)

  Outer loop → for each Q tile Qᵢ (N/64 blocks):
    Load Qᵢ from HBM → SRAM          ← ONE HBM read per Q block

    Inner loop → for each K,V tile Kⱼ, Vⱼ:
      Load Kⱼ from HBM → SRAM        ← ONE HBM read per K block
      Load Vⱼ from HBM → SRAM        ← ONE HBM read per V block
      Sᵢⱼ = Qᵢ × Kⱼᵀ / √d           ← SRAM matrix multiply
      Online softmax update (m,ℓ,O corrected in registers)

    Write Oᵢ from SRAM → HBM         ← ONE HBM write per Q block

HBM TRAFFIC COMPARISON (N=4096, d=128, BF16):
  Standard attention:      139 MB per head
  Flash Attention v1:        4.1 MB per head  ← 34× reduction!

  Speedup: 2–4× at N=2048; 8–12× at N=8192 (memory dominates longer)

FAv1 BACKWARD PASS BREAKTHROUGH:
  Standard: stores N×N score matrix S → O(N²) memory during training
  FAv1: stores only O and ℓ (size N each); recomputes softmax on-the-fly
  Memory: O(N²) → O(N×d)
  For 70B at seq=32K: 3.3 TB → 5 GB stored (660× memory reduction!)
  Cost: +33% compute for recomputation → worth it at any sequence > 2K
```

4.8.5 Flash Attention v2 — Warp-Level Parallelism

FAv2 *Dao, 2023*

```
FAv1 PROBLEM: 40-50% warp utilization — many warps idle during reduction steps

FAv2 KEY INNOVATIONS:

① Split K/V dimension across warps (parallelism):
  FAv1: all warps work on same (Q tile, K/V tile)
  FAv2: different warps process different K/V tiles simultaneously
        → merge via warp-level reductions
  Warp utilization: 40% → 75%+

② Parallelize over BOTH Q and K/V outer loops:
  2× more thread-level parallelism exposed

③ Algebraically simplify correction math:
  ~25% fewer non-GEMM FLOPs vs FAv1 per tile step

④ Native GQA support (H_kv = H_q / G):
  K/V loaded once, applied to G query heads
  → G× reduction in K/V HBM reads for GQA configs

PERFORMANCE (A100 SXM, N=2048, d=64, BF16):
  Standard:  ~42 TFLOPS  (12% of 312 TF peak)
  FAv1:      ~93 TFLOPS  (30% of peak)
  FAv2:     ~197 TFLOPS  (63% of peak)
  Speedup FAv2 vs standard: 4.7× · vs FAv1: 2.1×

Seq-length scaling:
  N=512:  FAv2 ~1.8× over std  N=8192:  FAv2 ~8× over std
  N=2048: FAv2 ~4.7× over std  N=32768: FAv2 ~12× over std
```

4.8.6 Flash Attention v3 — H100-Native Hardware Exploitation

FAv3 *Shah et al., 2024 — H100 exclusive*

```
H100 HARDWARE PRIMITIVES ENABLING FAv3:

① WGMMA (Warp Group Matrix Multiply Accumulate):
  H100-specific: operates on 4-warp groups (128 threads) vs per-warp MMA
  Tile: 64×16×16 per warp group (larger than FAv2's per-warp tile)
  Throughput: ~2× vs per-warp MMA · larger tiles → better SMEM→TC intensity

② TMA (Tensor Memory Accelerator):
  Dedicated hardware copies tiles HBM→SRAM ASYNCHRONOUSLY
  Does NOT consume SM compute cycles or warp execution slots
  Pipeline: TMA loads K,V tile N+1 while WGMMA computes tile N → TRUE OVERLAP

FAv3 EXECUTION PIPELINE (with TMA + WGMMA):
  TMA:   "Load K tile N+1 to SMEM"  [hardware, no SM cycles]
  WGMMA: Q × K^T (tile N)           [SM fully utilized simultaneously]
  WGMMA: P × V   (tile N)           [SM still computing]
  TMA completes → tile N+1 in SMEM, WGMMA starts immediately
  Effective throughput: ~2× v2 (both units busy, no serialization)

FAv3 FP8 (H100 Transformer Engine):
  Q,K computed in FP8 WGMMA → ~2× compute/cycle
  Online descaling per tile (per-row scaling factor)
  O accumulated in FP32 for stability · Quality impact: <0.1% degradation

PERFORMANCE (H100 SXM, causal, BF16, d=128):
  N       Standard    FAv2       FAv3       % H100 Peak
  512     124 TF      280 TF     420 TF       42%
  2048    88 TF       498 TF     680 TF       69%
  4096    71 TF       571 TF     737 TF       74%
  8192    52 TF       593 TF     750 TF       76%
  FAv3 FP8 at N=4096: ~850-900 TFLOPS (86-91% of peak!)
```

4.8.7 Flash Attention in Production

```
python# PyTorch 2.0+ SDPA — automatically selects FA backend by GPU arch import torch.nn.functional as F with torch.backends.cuda.sdp_kernel(enable_flash=True, enable_math=False): output = F.scaled_dot_product_attention(q, k, v, is_causal=True) # A100 → FAv2 · H100 → FAv3 · Very short seqs fall back to math path # HuggingFace Transformers — explicit FA model = AutoModelForCausalLM.from_pretrained( "meta-llama/Llama-3-70b", attn_implementation="flash_attention_2", torch_dtype=torch.bfloat16) # Sliding Window Attention (Mistral architecture) from flash_attn import flash_attn_func output = flash_attn_func(q, k, v, causal=True, window_size=(4096, 0)) # O(N×W) instead of O(N²) for long seqs · KV cache: only store last 4K tokens # For W=4096 vs N=128K: 32× KV cache reduction! # Profiling command to verify FA is active: # ncu --kernel-name "flash_fwd_kernel" --set full python script.py # Look for: TMA accesses + WGMMA instructions → FAv3 confirmed on H100
```

4.8.8 GQA + Flash Attention — Compounding Savings

```
COMPOUNDING SAVINGS (LLaMA-3 70B, N=4096, H_q=64, H_kv=8, G=8):
  Without GQA, without FA:  537 GB attention HBM traffic
  Without GQA, with FA:     2.5 GB
  With GQA, with FA:        1.05 GB
  Combined reduction: 537 GB → 1.05 GB = 511× less traffic!

PRODUCTION SCENARIO: LLaMA-3 70B Prefill, N=8192 (GQA + FAv3)
  Standard attention + GQA:  856 GB per forward pass
  FAv3 + GQA:                34.9 GB per forward pass

  At H200 (4.8 TB/s):
    Standard: 856 / 4800 = 178ms · FAv3: 34.9 / 4800 = 7.3ms
    Saved: 171ms per 8K-token prefill request

  At 1,000 req/min production load:
    2,850ms/sec of GPU time freed ≈ 2.85 GPU-equivalents
    At $2.50/GPU-hr: $7.13/hr = $62,400/year from FA alone.
```

Feature	Standard	FAv1 (2022)	FAv2 (2023)	FAv3 (2024)
HBM writes	**O(N²)**	**O(N×d)**	**O(N×d)**	**O(N×d)**
Warp utilization	~12%	~40%	~75%	~75–85%
WGMMA + TMA (H100)	—	—	—	**YES**
FP8 support	—	—	—	**YES (H100)**
GQA native	—	—	**YES**	**YES**
Max H100 utilization	~12%	~40%	~63%	**~76% BF16 / ~88% FP8**
Speed vs standard	1×	2–4×	4–8×	**6–14×**

Production Recommendation

H100: FAv3 via `flash-attn>=2.6` or `torch.nn.functional.scaled_dot_product_attention` with `enable_flash=True`

A100: FAv2 via `flash-attn>=2.0`

AMD MI300X: FAv2 via ROCm flash-attn fork (same algorithm, HIP backend)

Rule: Never use standard attention for sequence length > 512 in any production system.

4.9 Memory Capacity Planning for LLM Deployments

```
TRAINING MEMORY BUDGET (LLaMA-3 70B, TP=8, BF16, Adam):

  Model params:     70B/8 × 2 bytes = 17.5 GB per GPU
  Optimizer states: Without ZeRO: 70B × 16 bytes / 8 = 140 GB → EXCEEDS 80 GB!
  With ZeRO-3 (FSDP):
    Each GPU holds 1/N all states: 70B × 16 / 64 GPUs = ~17.5 GB
    + Activations + buffers: ~30–50 GB total → fits in 80 GB HBM3 ✓

INFERENCE MEMORY BUDGET (LLaMA-3 70B, FP8, TP=8, H100):

  Model weights:    70B/8 × 1 byte = 8.75 GB
  KV cache/req:     2×L×H×D×S×2 bytes = 2×80×8×128×4096×2 = 1.3 GB/req
  KV cache (32 req): 41.6 GB
  Activation buffers: ~2 GB · Runtime (NCCL, CUDA ctx): ~2–3 GB
  Total (32-req batch): ~54 GB → fits in 80 GB HBM3 ✓
  Max batch size: (80 - 12 GB overhead) / 1.3 GB ≈ 52 concurrent requests

Levers to increase batch size:
  NVFP4 weights: → 4.4 GB (vs 8.75) → ~20 GB freed for KV cache
  INT8 KV cache: 41.6 → 20.8 GB → 2× more concurrent requests
  H200 (141 GB): ~90 GB more available → ~3× more concurrent requests
```

4.10 Memory Performance Decision Tree

```
START: GPU kernel to optimize
       |
  Measure Arithmetic Intensity (Nsight Compute → Roofline tab)
       |
  ┌────▼──────────────┐
  |  AI < Ridge Point? |──YES──→ MEMORY BOUND. Apply in order:
  |  (295 H100 BF16)  |         1. Operator fusion (biggest single win)
  └───────────────────┘          2. Quantization (fewer bytes/param → less BW)
       | NO                      3. Fix access coalescing (check Nsight)
       ▼                         4. Increase SMEM tile sizes
  COMPUTE BOUND                  5. Prefetching / async copy (TMA on H100)
  Apply in order:                6. KV cache compression (inference only)
  1. Increase batch size
  2. Optimize Tensor Core tile efficiency (CUTLASS)
  3. Reduce register pressure → increase occupancy
  4. Use TC-native types: TF32, BF16, FP8, FP4
  5. Consider operator fusion (reduce launch overhead)
```

Architecture Diagrams

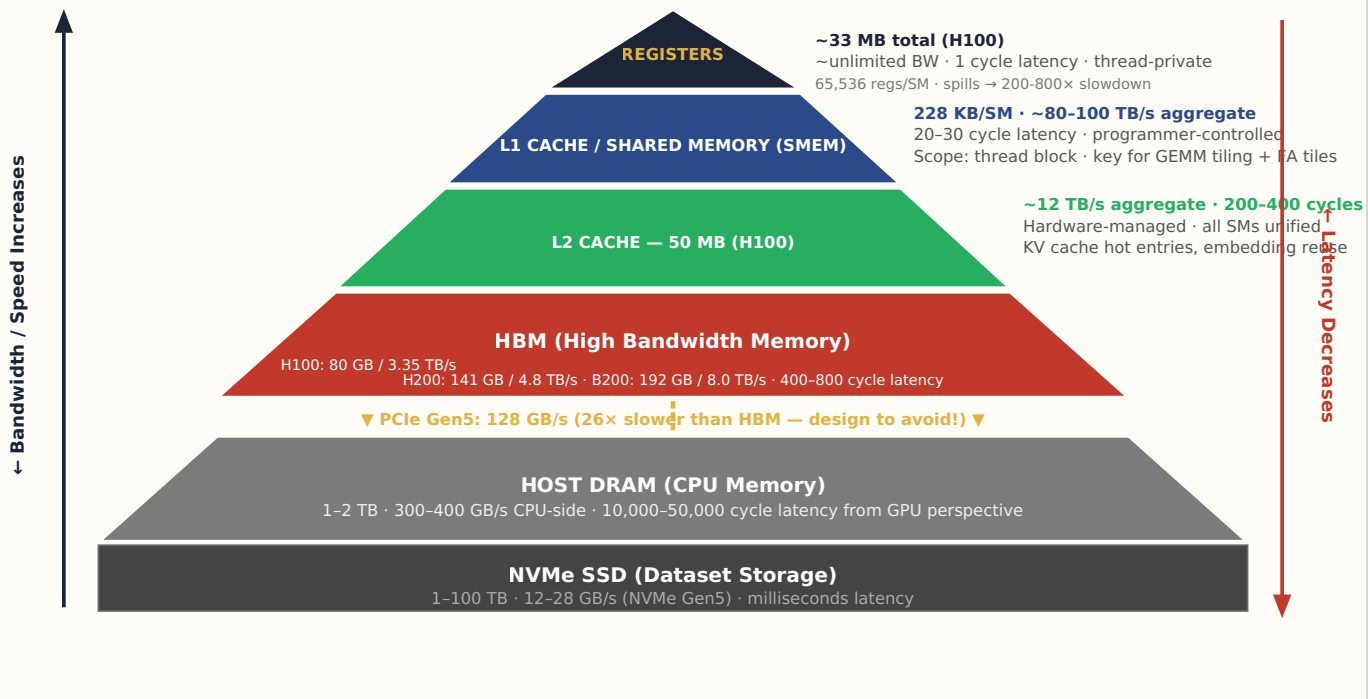


Figure 4.1 — GPU system memory hierarchy pyramid. Apex = fastest/smallest (registers, 1 cycle). Base = largest/slowest (NVMe, ms). The gold dashed PCIe barrier marks the 26× bandwidth cliff for GPU↔CPU transfers. Every performance optimization moves working data toward the apex: SMEM tiling for GEMM, Flash Attention tiles fitting in 228 KB SMEM, quantization shrinking HBM footprint.

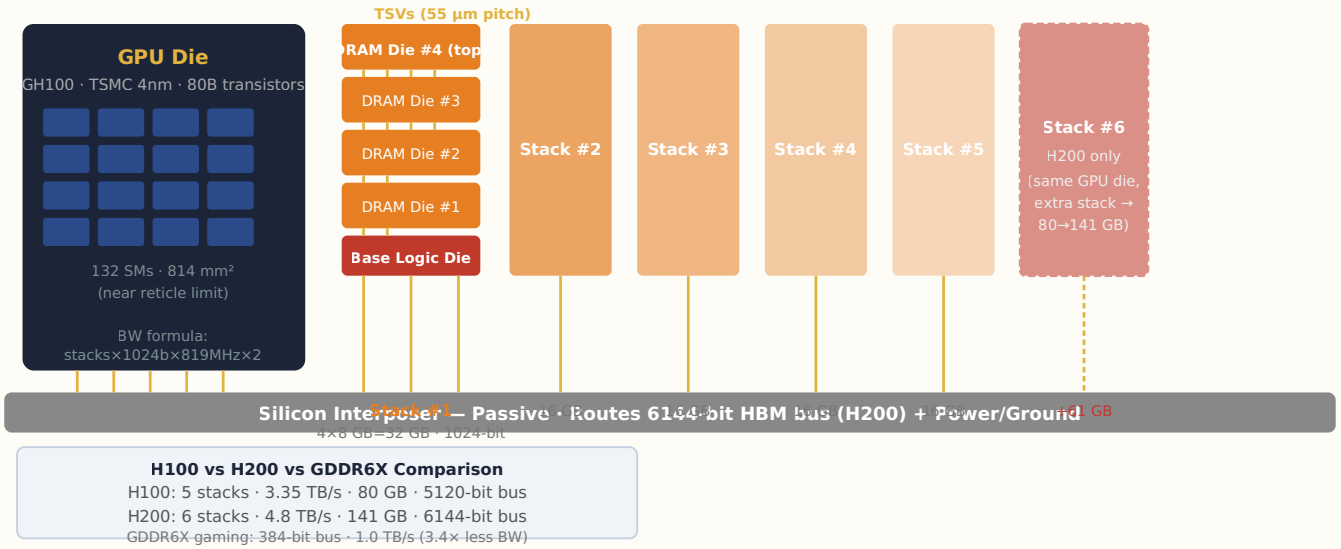


Figure 4.2 — HBM3e physical stack cross-section. Left: H100 GPU compute die (132 SMs). Center: five HBM stacks on silicon interposer, each with 4 DRAM dies + base logic die connected by TSVs at 55 µm pitch, providing 1024-bit bus per stack. Right (red dashed): H200's 6th stack — same GH100 die, one more HBM3e stack adds 61 GB (80→141 GB) and 1.45 TB/s bandwidth (3.35→4.8 TB/s). The entire package fits on a single silicon interposer bounded by reticle limits.

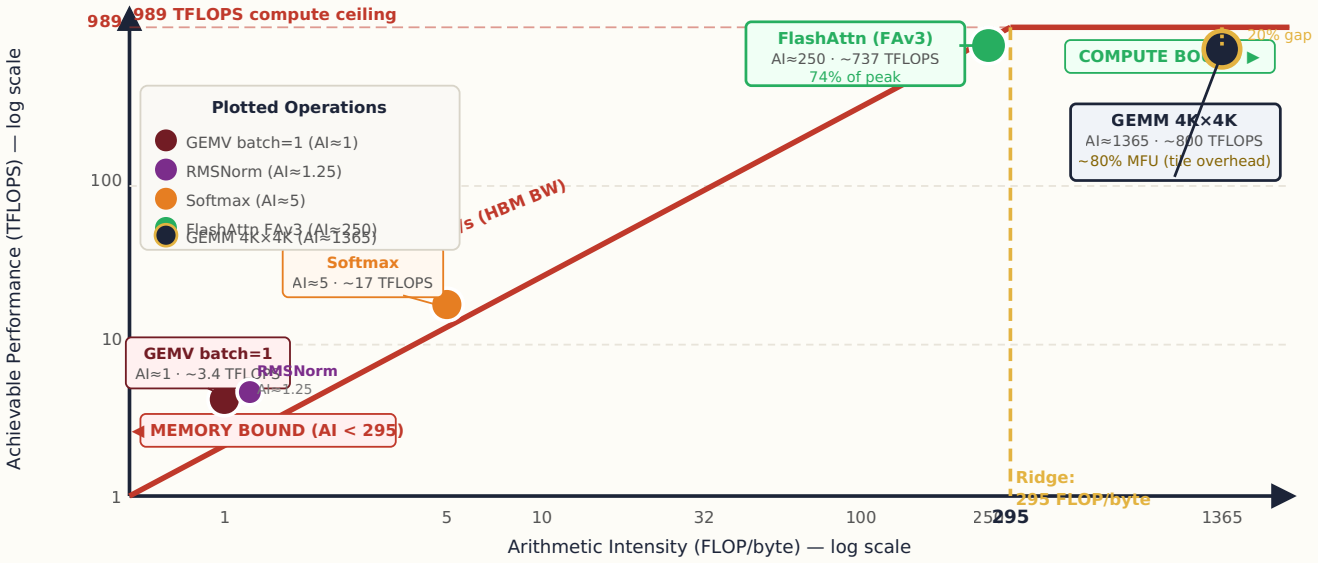


Figure 4.3 — Roofline Model for H100 SXM BF16 (both axes log-scale). Memory slope = 3.35 TB/s HBM bandwidth. Compute ceiling = 989 TFLOPS. Ridge at 295 FLOP/byte (gold dashed). Operations left of ridge are memory-bound: optimization means reducing bytes (fusion, quantization, Flash Attention). Operations right of ridge are compute-bound: optimization means improving tensor core efficiency (tile sizing, occupancy). FlashAttention (green) achieves AI≈250 — near-ridge, near-optimal — by eliminating the $O(N^2)$ HBM writes of standard attention.

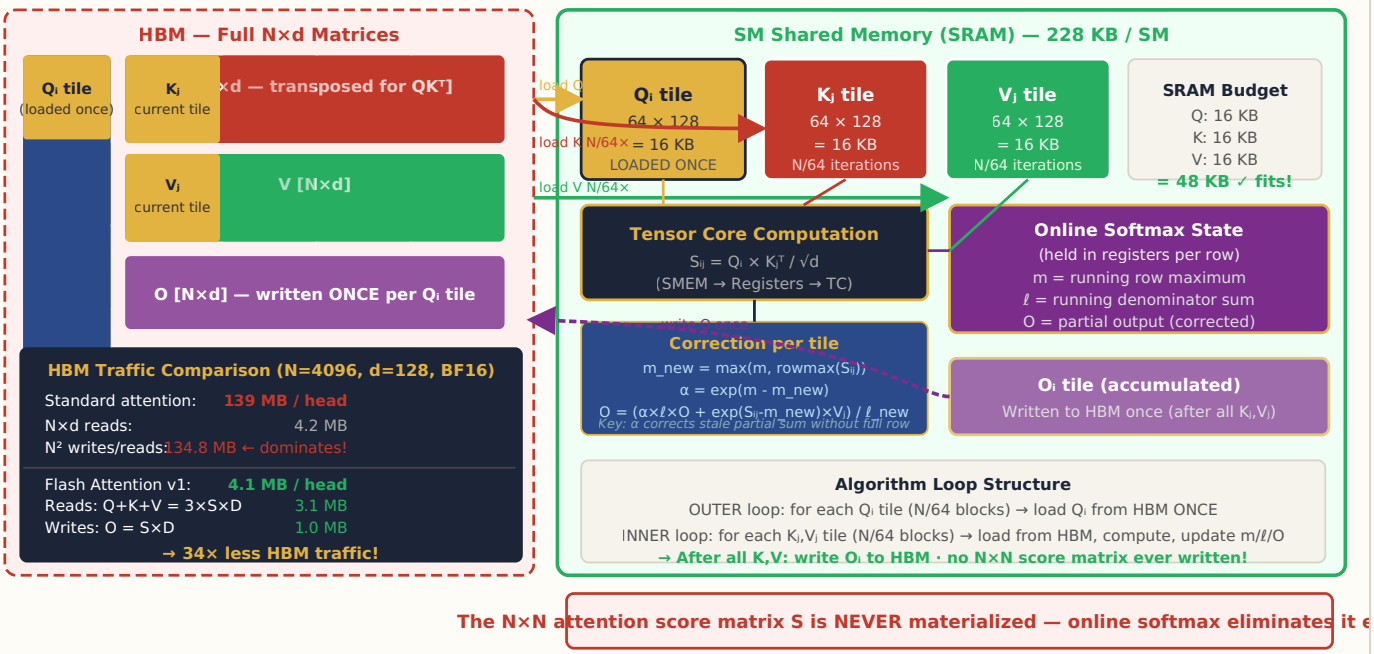


Figure 4.4 — Flash Attention tiling mechanics. Left (red): HBM holds the full Q, K, V, O matrices. The N×N score matrix S is never written to HBM. Right (green): SM shared memory workspace. Q_i (gold, 16 KB) is loaded once per outer loop. K_j and V_j tiles (16 KB each) are loaded in the inner loop N/64 times. Online softmax state (m, ℓ, O) lives in registers and is corrected per tile using factor α = exp(m_old − m_new) without ever needing the full row. Result: 139 MB → 4.1 MB per head (34× reduction).

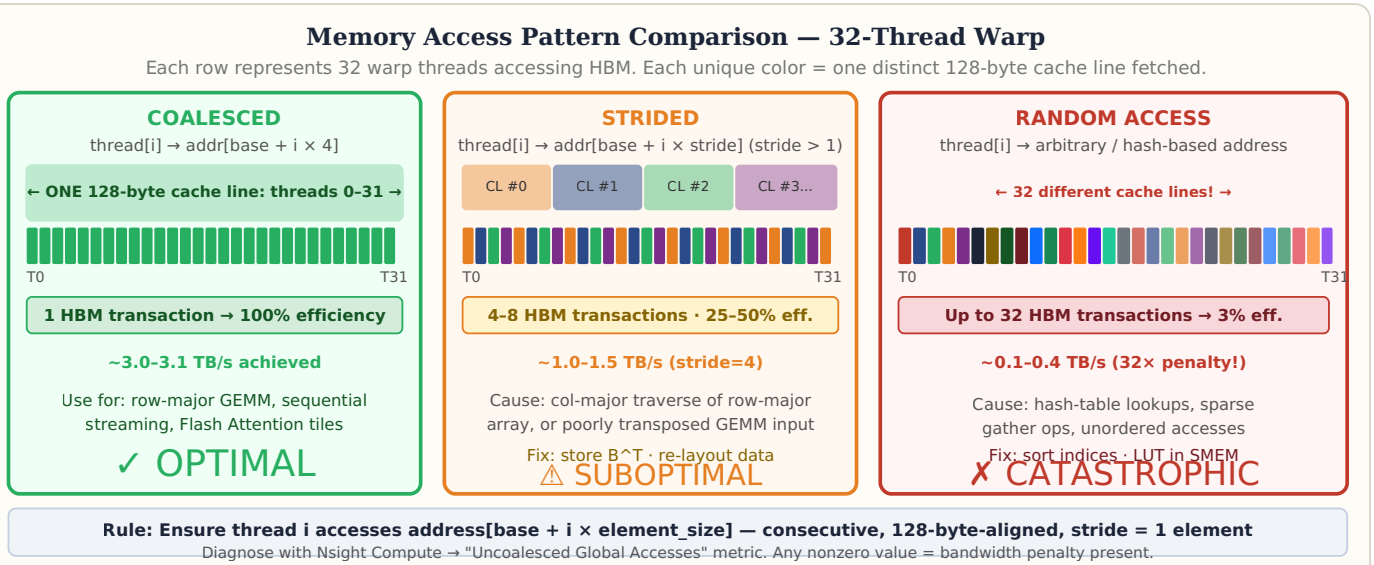


Figure 4.5 — Memory access pattern comparison. Each colored square = one 32-bit element; unique colors = different 128-byte cache lines. Coalesced (left): all 32 threads access one cache line → 1 HBM transaction → ~3 TB/s. Strided (center): 4 cache lines needed → 4+ transactions → 25–50% efficiency. Random (right): 32 different cache lines → up to 32 transactions → 3% efficiency, 0.1–0.4 TB/s. Uncoalesced access is the #1 cause of preventable HBM bandwidth waste.

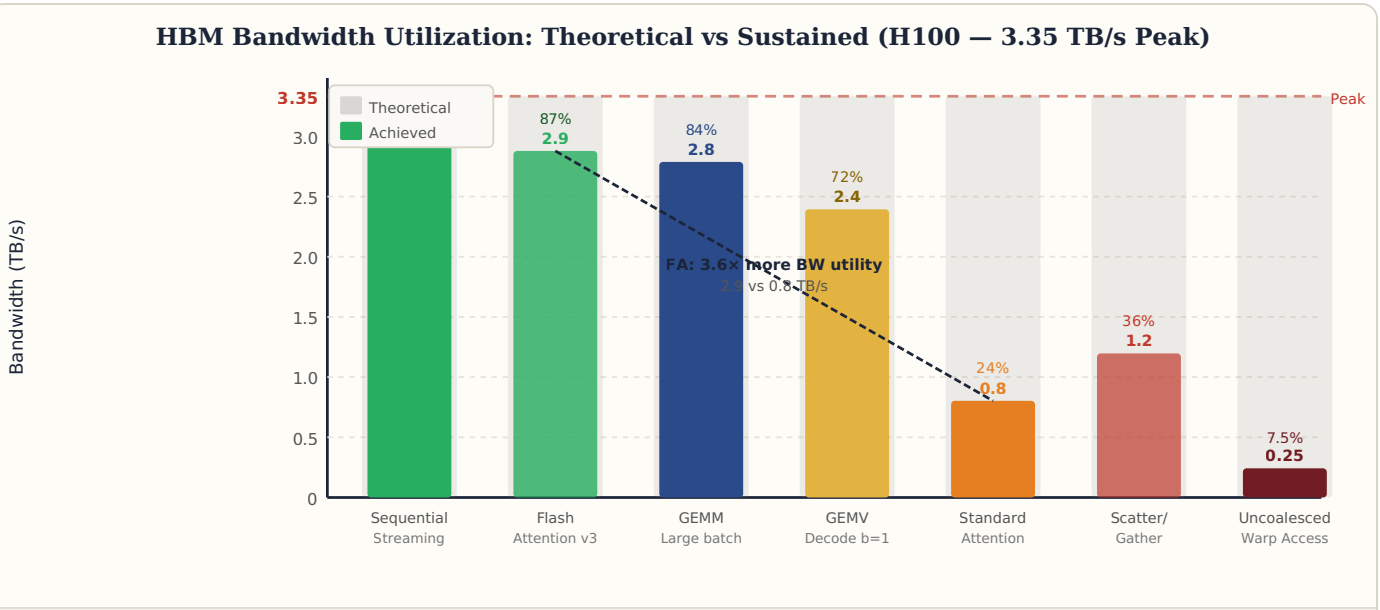


Figure 4.6 — HBM bandwidth utilization by operation type on H100 (3.35 TB/s peak). Gray = theoretical maximum; colored bars = sustained achieved bandwidth. Flash Attention v3 achieves 2.9 TB/s (87%) vs standard attention's 0.8 TB/s (24%) — a 3.6× utilization improvement from eliminating N^2 HBM writes. Uncoalesced warp access achieves only 7.5% of peak (0.25 TB/s) — the single most costly access pattern mistake. GEMV decode at 72% (2.4 TB/s) represents the irreducible cost of loading full weight matrices for a single token.

Quick Reference — H100 SXM Memory Hierarchy at a Glance

Tier	Size	Bandwidth	Latency	Scope	Primary LLM Role
Registers	256 KB/SM (~33 MB total)	~unlimited (1 cycle)	1 cycle	Thread-private	Occupancy driver; spill = 200–800× penalty. Profile: NCU "Register spills to local memory".
L1 / SMEM	228 KB/SM	~80–100 TB/s aggregate	20–30 cycles	Thread block	GEMM tiling (128×128 tiles), Flash Attention Q/K/V tiles (16 KB each). Programmer-managed.
L2 Cache	50 MB (H100)	~12 TB/s aggregate	200–400 cycles	All SMs, unified	KV cache hot blocks, embedding reuse, LayerNorm running stats. Hardware-managed.
HBM3	80 GB (H100) / 141 GB (H200) / 192 GB (B200)	3.35 / 4.8 / 8.0 TB/s	400–800 cycles	Per-GPU global	Model weights, full KV cache, activations. Primary bottleneck for memory-bound kernels.
Host DRAM	1–2 TB / server	128 GB/s (PCIe Gen5)	10,000–50,000 cycles	System-wide (via PCIe)	Checkpoint staging, dataset prefetch, model parameter spill (avoid at all costs).
NVMe SSD	1–100 TB	12–28 GB/s (Gen5)	~100 µs (ms-range)	Node storage	Dataset streaming (preloaded to DRAM), checkpoint persistence, model weights at rest.

Operation	Arithmetic Intensity	Regime on H100	Achievable TFLOPS	Primary Optimization
RMSNorm / LayerNorm	~1.25 FLOP/byte	**Deeply memory-bound**	~4 TFLOPS	Fuse with adjacent GEMM — mandatory
GEMV batch=1 (decode)	~1.0 FLOP/byte	**Deeply memory-bound**	~3 TFLOPS	Quantization (INT4/FP8), batch requests
Softmax (attention row)	~5 FLOP/byte	**Memory-bound**	~17 TFLOPS	Fuse into Flash Attention kernel
GEMV batch=32	~32 FLOP/byte	**Memory-bound**	~107 TFLOPS	Increase batch size further
FlashAttention FAv3 (S=4K)	~250 FLOP/byte	**Near-ridge (optimal)**	~737 TFLOPS (74%)	Already optimal — use FAv3 on H100
GEMM 4K×4K (prefill)	~1,365 FLOP/byte	**Compute-bound**	~800 TFLOPS (80%)	Tile sizing, occupancy, FP8 Tensor Cores

Key Number	Value	Why It Matters
H100 ridge point (BF16)	**295 FLOP/byte**	The dividing line between memory-bound and compute-bound. Know it cold.
H100 HBM3 bandwidth	3.35 TB/s	The slope of the memory roof. Decode throughput = weight_bytes / 3.35 TB/s.
H100 SMEM / SM	228 KB	Maximum tile size budget. FA tiles (Q+K+V = 48 KB) must fit here.
HBM traffic: standard attention (70B, N=4K)	537 GB/fwd pass	The problem Flash Attention solves — 511× larger than with FA + GQA.
HBM traffic: FA + GQA (70B, N=4K)	1.05 GB/fwd pass	What production LLM inference actually uses. 511× reduction vs standard.
FA HBM reduction (per head)	34× (139→4.1 MB)	The core Flash Attention result at N=4096, d=128.
FAv1 backward memory savings	660× (3.3 TB→5 GB at seq=32K)	Why long-context training became feasible in 2022.
FAv3 peak utilization	74% BF16 / 88% FP8 (N=4096)	Best achievable on H100 for attention. WGMMA + TMA enables this.
Production FA value (70B, 8K ctx, 1K req/min)	$62,400/year	Cost of memory bandwidth reclaimed by FAv3 at $2.50/GPU-hr.
Uncoalesced warp access penalty	32× BW overhead	Single most costly access pattern error. Detect with NCU immediately.

Chapter 4 Review Questions

0. **A kernel has arithmetic intensity of 50 FLOP/byte on H100 (ridge ~295 FLOP/byte). Is it memory-bound or compute-bound? What is its theoretical peak performance?**
   By how much would you need to increase the batch size to push this operation above the ridge point, assuming AI scales linearly with batch size? At what batch size would the same kernel become compute-bound on B200 (ridge ~562 FLOP/byte)?

0. **You profile a custom attention implementation and observe 0.8 TB/s HBM bandwidth on H100. Flash Attention achieves 2.8 TB/s for the same operation. Explain the exact mechanism — not "uses tiling."**
   Name the two specific HBM write operations that standard attention requires and FAv1 eliminates. Derive the HBM traffic formula for both and verify the 34× reduction at N=4096, d=128, BF16.

0. **An LLaMA-3 70B inference job runs on 8× H100 GPUs with TP=8 in BF16. What is the per-GPU memory footprint for weights alone? If you switch to INT4 (AWQ), how does the maximum concurrent batch size change?**
   Show the full calculation: weights → KV cache per request → max batch given 80 GB HBM with 12 GB runtime overhead. Then repeat for INT8 KV cache with INT4 weights.

0. **Flash Attention v3 uses WGMMA and TMA. Explain what each primitive does and why their combination yields ~1.5× throughput over FAv2 on the same H100.**
   What happens to FAv3 performance on an A100 — does it fall back to FAv2 or FAv1? What specific hardware features are missing? Explain why TMA is particularly critical for hiding HBM latency in attention's inner loop.

0. **The online softmax trick maintains running statistics m (running max) and ℓ (running sum) per row. Why is subtracting the running maximum m critical for numerical stability?**
   Walk through what would happen at sequence length N=8192 with head_dim=128 if standard non-stabilized softmax were applied tile-by-tile — which floating point exception occurs first, and at what attention score value for BF16?

0. **Describe the complete sequence of memory accesses for a single autoregressive decode step in LLM inference, from request arrival to token output.**
   For each stage — embedding lookup, attention QKV projection, attention score computation, MLP layers, output projection, sampling — identify (a) which memory tier the data lives in, (b) the approximate arithmetic intensity, and (c) whether the stage is memory-bound or compute-bound at batch=1 on H100.

Chapter Cross-References

Ch 1 Roofline model seven-layer framework — the analytical foundation for every AI calculation in this chapter. The ridge point, memory and compute regimes, and MFU interpretation all originate there.

Ch 3A/3B GPU silicon architecture — SM structure, Tensor Core generations, HBM physical integration, NVLink topology. HBM stack numbers used here are derived from Ch 3B GPU roadmap tables.

Ch 5 Thermal & TCO — the HBM3 memory subsystem consumes ~40–60W on H100 before any compute. This feeds directly into TDP budgeting, cooling design, and tokens-per-kW analysis in Ch 5.

Ch 7 GPU Kernels — CUTLASS tiling patterns, GEMM kernel anatomy, Tensor Core tile feeding via SMEM. The SMEM bank conflict and swizzled layout techniques introduced here are implemented concretely in Ch 7.

Ch 8 Quantization — FP8 E4M3/E5M2, INT4 AWQ, NVFP4 formats covered briefly here (bandwidth reduction) are analyzed in full precision/accuracy/hardware detail in Ch 8.

Ch 9 Fusion & Compiler — operator fusion strategy introduced in §4.5.1 is automated by XLA, torch.compile, and TensorRT at the graph compiler level; Ch 9 covers the compiler IR, fusion passes, and auto-tuning.

Ch 11 KV Cache — inference memory capacity planning (§4.9.2) provides the foundation. Ch 11 extends this with paged attention, prefix caching, eviction policies, and KV quantization.

Ch 17 Telemetry & Tools — Nsight Compute metric tables from §4.7 map directly to dcgm-exporter Prometheus counters and omniperf dashboard panels covered in Ch 17.