PART I · CHAPTER 2

Transformer Architecture: A Performance Engineer's Deep Dive

Attention, normalization, positional encoding — seen as memory and compute problems. Every formula derived from first principles, every number checkable against LLaMA-3 70B.

Book: AI/ML Infrastructure from Silicon to Scale | **Author:** Venkat Vinjam | **Cross-ref:** Ch 4: FlashAttention Ch 6: KV cache serving Ch 11: KV paging Ch 13: MoE routing

> *"The transformer is not an algorithm. It is a data movement problem wearing an algorithm's clothes. Every optimization in LLM infrastructure is an optimization for how we move Q, K, V, and the weight matrices through the memory hierarchy."*
>
> — Performance Engineering Perspective

A performance engineer who cannot derive the FLOP and memory budget of a transformer forward pass from the model config cannot make informed decisions about hardware, batching strategy, parallelism, or quantization. This chapter gives you that derivation — every formula derived from first principles, verified with LLaMA-3 70B numbers that you can check against the published model card.

Learning Objectives

- **Level 1:** Explain what attention is doing computationally and why transformers replaced RNNs; distinguish encoder-decoder from decoder-only architectures and their system implications
- **Level 2:** Compute exact FLOPs and memory for any transformer configuration; derive KV cache sizes; classify each operation by arithmetic intensity and performance regime
- **Level 3:** Reason about architecture choices (GQA vs MHA vs MLA, MoE sparsity) as infrastructure engineering decisions with quantified cost; analyze forward and backward pass memory at every stage for training system design

2.1 Why Performance Engineers Must Understand Transformers

Every GPU workload in modern AI is a transformer workload. The model config — layers, hidden dim, heads, FFN multiplier, vocab size, sequence length, attention variant — directly and quantifiably determines hardware requirements. You cannot right-size a cluster, choose a quantization format, set a tensor parallelism degree, or engineer an SLA without being able to derive these numbers from first principles.

What you must be able to derive from a model card:

(1) total parameter count and memory in any precision, (2) FLOPs per token for prefill vs decode, (3) KV cache bytes per token per request, (4) arithmetic intensity of each operation class, (5) which operations are compute- vs memory-bound at each batch size — before running a single experiment.

The transformer's dominance traces to a single insight: replace recurrence with attention. RNNs process tokens serially — O(S) sequential steps, no parallelism. Transformers compute all-to-all attention in a single parallel operation, enabling massive GPU utilization during training. The cost is $O(S^2)$ attention complexity — the quadratic wall that every long-context system must engineer around.

2.2 Attention as Matrix Multiplication: The FLOP and Memory Breakdown

Scaled dot-product attention computes: **$\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d_k}) \times V$**. For a performance engineer, this is two matrix multiplies (QK^T and Attn×V) separated by a softmax with specific memory access patterns.

```
SCALED DOT-PRODUCT ATTENTION — step-by-step derivation:

  Input: X ∈ R^{S × d_model}   (S tokens, d_model dimensions)

  Step 1 — Project to Q, K, V:
    Q = X · W_Q    W_Q ∈ R^{d_model × d_head}    → Q ∈ R^{S × d_head}
    K = X · W_K    W_K ∈ R^{d_model × d_head}    → K ∈ R^{S × d_head}
    V = X · W_V    W_V ∈ R^{d_model × d_head}    → V ∈ R^{S × d_head}

  Step 2 — Compute attention scores:
    S_matrix = Q · K^T / √d_k    → R^{S × S}     ← THE QUADRATIC TERM

  Step 3 — Causal mask (decoder-only):
    M[i,j] = 0 if j ≤ i,  -inf if j > i   → position i cannot see future tokens

  Step 4 — Softmax:
    A = softmax(S_matrix + M)              → R^{S × S},  each row sums to 1

  Step 5 — Weighted sum:
    O = A · V                              → R^{S × d_head}

  Complete formula: Attention(Q,K,V) = softmax( Q·K^T / √d_k + M ) · V

ATTENTION FLOP ANALYSIS — single head, single layer:
  S = sequence length, d = head dimension = d_model / H_q

  QK^T product:   2 × S × d × S  = 2S²d   FLOPs  (outer product)
  Softmax:        ~5 × S²         FLOPs  (≈1% of total — negligible vs GEMM)
  AV product:     2 × S × S × d  = 2S²d   FLOPs
  Total per head: 4S²d FLOPs  (ignoring softmax)

WHY √d_k? The softmax saturation problem:
  Without scaling: for random Q,K with unit variance, σ(Q·K^T) ≈ √d_k
  At d_k=128: σ ≈ 11.3 → softmax inputs in range [-30,+30]
  Softmax gradient ∂σ/∂x ≈ σ(1−σ) → near 0 → vanishing gradients
  Dividing by √d_k normalizes variance back to 1.0 → healthy gradient flow.

HBM MEMORY TRAFFIC (standard attention, BF16):
  Reads:   Q(S×d) + K(S×d) + P(S×S) + V(S×d) = 4Sd + S²  × 2 bytes
  Writes:  S(S×S) + P(S×S) + O(S×d)           = 2S² + Sd  × 2 bytes
  Total:   (4Sd + 3S²) × 2 bytes   ← the S² term kills standard attention

  S=4096: attention matrix per head = 4096² × 2B = 32 MB
  80 layers × 64 heads = 163 GB to HBM — impossible without FlashAttention!

  FlashAttention (→ Ch 4 §4.8) eliminates N² HBM traffic entirely:
  → A is computed in SRAM tiles, never written to HBM
  → HBM access drops from O(S²) to O(S·d)  [compute stays O(S²)]
```

2.3 Multi-Head Attention: Parallelism, KV Cache, and Head Specialization

Running attention once gives one view of token relationships. Multi-head attention runs H parallel attention operations on lower-dimensional projections, giving the model H independent perspectives simultaneously.

```
MULTI-HEAD ATTENTION QKV PROJECTION FLOPs (LLaMA-3 70B):

Config: d_model=8192, H_q=64 heads, H_kv=8 (GQA), d_head=128

Q projection:  2 × d_model × H_q × d_head  = 2 × 8192 × 64 × 128 = 134.2M FLOPs
K projection:  2 × d_model × H_kv × d_head = 2 × 8192 × 8 × 128  =  16.8M FLOPs
V projection:  same as K                                          =  16.8M FLOPs
Output proj:   2 × d_model × H_q × d_head  = 2 × 8192 × 8192      = 134.2M FLOPs
Total QKV+O:   302.0M FLOPs per token per layer

KV CACHE MEMORY (the serving cost that persists across tokens):
  Per token per layer (BF16):
    2 (K+V) × H_kv × d_head × 2 bytes = 2 × 8 × 128 × 2 = 4,096 bytes = 4 KB/token/layer

  Full model (L=80 layers, S=4096 ctx, BF16):
    KV_bytes = 2 × L × H_kv × d_head × S × 2
             = 2 × 80 × 8 × 128 × 4096 × 2 = 5.37 GB per request

  On H100 (80 GB HBM), 10 GB for weights (TP), 10 GB for activations:
    Available for KV: 60 GB / 5.37 GB = ~11 concurrent 4K requests (BF16)
    INT8 KV: 2.7 GB/request → 22 concurrent requests (2× more serving capacity)
```

2.3.1 Head Specialization — What Interpretability Research Shows

Different heads in a transformer learn qualitatively different relationships. This is not just empirical observation — it has direct implications for pruning, distillation, and understanding which heads to keep under compression.

Head Type	What It Learns	Typical Layer Range	Pruning Risk
Positional heads	Attend to adjacent tokens — local syntactic structure, prev/next word	Early (0–8)	Medium
Syntactic heads	Subject-verb agreement, pronoun coreference, dependency arcs	Middle (8–40)	High
Induction heads	Pattern completion: [A][B]…[A] → predict [B]. Mechanistic basis of in-context learning	Deep (40+)	Critical
Copying heads	Directly copy tokens from context to output — essential for in-context retrieval	Deep	Critical

Heads Are "Free" in Compute — Not in Memory:

Adding more query heads does not proportionally increase FLOP cost — total projection cost is fixed at d_model × d_model. What scales with H is the attention score computation (H × S^2 per layer) and the KV cache size. This is why GQA can reduce KV heads without reducing Q heads — quality loss is minimal, memory savings are large (see §2.10).

2.4 Encoder-Decoder vs. Decoder-Only: Architecture and System Implications

The original transformer ("Attention Is All You Need", Vaswani 2017) used a full encoder-decoder structure for machine translation. Modern LLMs have almost universally converged to decoder-only. Understanding both architectures — and why decoder-only won — is foundational knowledge for a performance engineer who must reason about cross-attention costs, KV cache structure, and inference memory footprint.

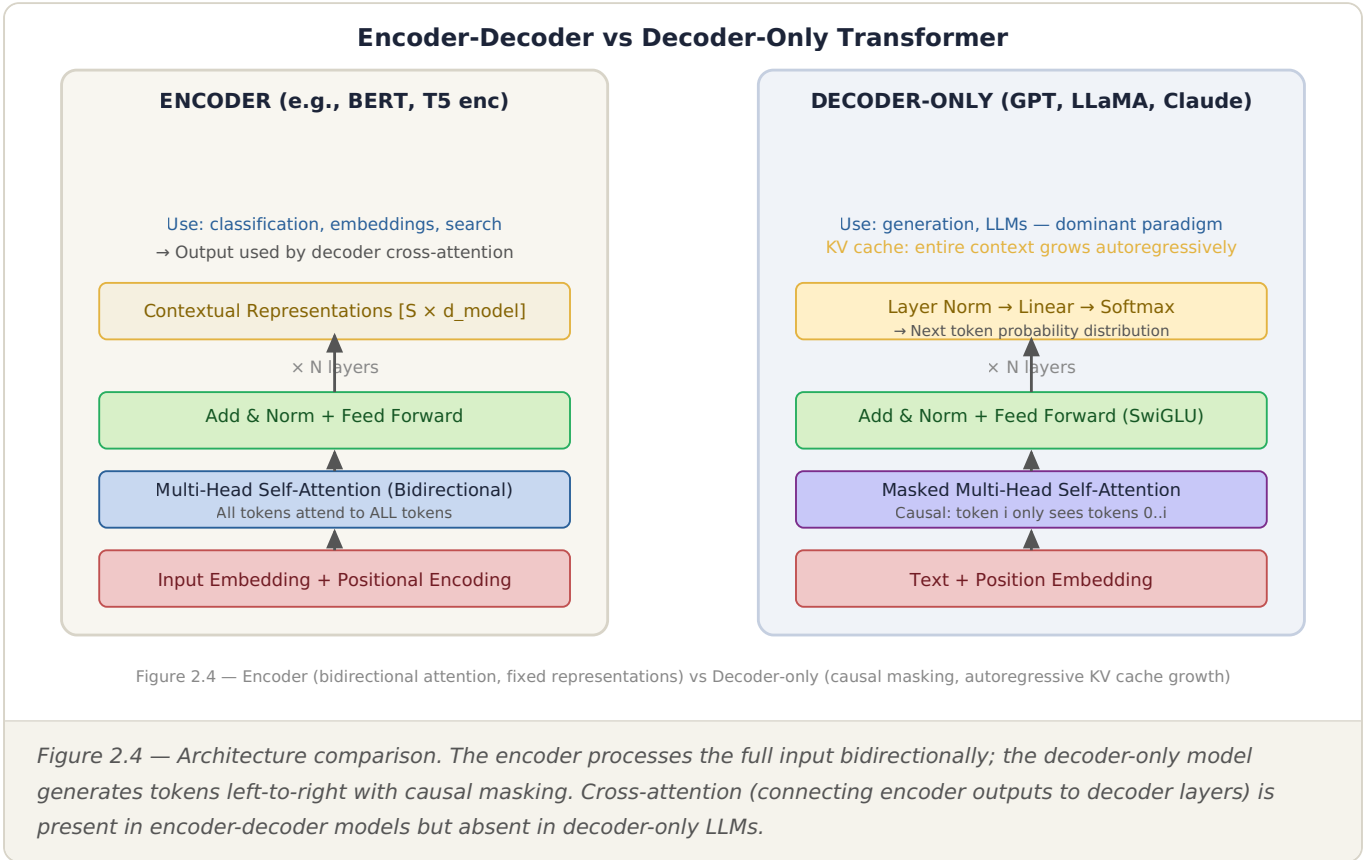


Figure 2.4 — Encoder (bidirectional attention, fixed representations) vs Decoder-only (causal masking, autoregressive KV cache growth)

Figure 2.4 — Architecture comparison. The encoder processes the full input bidirectionally; the decoder-only model generates tokens left-to-right with causal masking. Cross-attention (connecting encoder outputs to decoder layers) is present in encoder-decoder models but absent in decoder-only LLMs.

Property	Encoder (BERT, RoBERTa)	Encoder-Decoder (T5, BART)	Decoder-Only (GPT, LLaMA, Claude)
Attention type	Bidirectional self-attention	Enc: bidirectional; Dec: causal + cross	Causal self-attention only
Use case	Classification, embeddings, search	Translation, summarization, seq2seq	Open-ended generation — dominant paradigm
KV cache	None (no autoregressive generation)	Encoder KV computed once; decoder KV grows	Entire context KV grows with each token
Cross-attention	No	Yes — decoder attends encoder output (extra cost)	No — self-attention only
Memory at inference	O(1) after encoding	O(n_enc + n_dec) — smaller decoder KV	O(n_total) — all tokens share one KV
Industry trend	BERT-class still used for retrieval	Declining for new model development	**Dominant: GPT-4, Claude, Gemini, LLaMA**

2.4.1 Tokenization, Embeddings, and Positional Injection

Before any attention computation, raw text must be converted to vectors the GPU can process. The pipeline: raw text → tokens → integer IDs → dense embeddings → add positional information → feed to transformer blocks.

```
TOKENIZATION AND EMBEDDING PIPELINE:

  "The cat sat on mat" → tokenizer → [101, 532, 992, 87, 671]

  Embedding lookup:  E = W_emb[token_ids]    W_emb ∈ R^{vocab × d_model}
  LLaMA-3 70B: W_emb ∈ R^{128256 × 8192} → 128256 × 8192 × 2B = 2.1 GB

  Add positional information: X = E + PE  (or multiply with RoPE during attention)

  Sinusoidal PE (original, Vaswani 2017):
    PE(pos,2i)   = sin(pos / 10000^(2i/d))
    PE(pos,2i+1) = cos(pos / 10000^(2i/d))
  → Fixed, no learnable parameters, hard limit at training length

  RoPE (modern standard): applied inside attention, not added to embeddings
  → See §2.6 for full RoPE derivation
```

2.5 RMSNorm vs. LayerNorm

```
LAYERNORM vs. RMSNORM:

LayerNorm: y = ((x - μ) / σ) × γ + β
  μ = mean(x), σ = std(x) — requires 2 passes over data
  FLOPs/element: ~8 (mean, variance, subtract, divide, scale, shift)
  Memory AI: ~1.5 FLOP/byte → deeply memory-bound

RMSNorm (LLaMA, Mistral, Qwen, Gemma — modern standard):
  y = x / RMS(x) × γ,   where RMS(x) = √(mean(x²))
  FLOPs/element: ~5 (square, mean, sqrt, divide, scale) — no mean subtraction
  Memory AI: ~1.25 FLOP/byte → memory-bound

WHY RMSNorm won in production:
  1. 40% fewer FLOPs — measurable at 80-layer models, 10K tokens/s serving
  2. Simpler gradient: no mean subtraction → cleaner backward pass
  3. Empirically: negligible quality difference on modern LLMs at scale

PRODUCTION IMPLICATION:
  Both are deeply memory-bound (AI < 2). Standalone, they waste 99%+ of
  compute. Solution: ALWAYS fuse with adjacent GEMM or elementwise op.
  Unfused RMSNorm at 10B tokens/day costs ~12 GB/s of wasted HBM BW —
  trivial individually but adds up across 80 layers × 10,000 requests/sec.

  Pre-norm vs Post-norm:
  Pre-norm (modern):  x_out = x + Sublayer(Norm(x))   → stable deep training
  Post-norm (original): x_out = Norm(x + Sublayer(x)) → sensitive to init
```

2.6 Positional Encoding: RoPE — Full Derivation and Context Extension

2.6.1 RoPE Design Goal and Full Derivation

Transformers have no inherent notion of token order — attention treats input as a set. RoPE's key design goal: encode positions so that the dot product $\langle q_m, k_n \rangle$ depends only on token content and the *relative* distance $(m-n)$, not absolute position values.

```
ROPE DESIGN GOAL AND DERIVATION:

  Find f(·, pos) such that: ⟨f(q, m), f(k, n)⟩ = g(q, k, m−n)

  Rotation matrices achieve this exactly:
    f(x, pos) = R(pos) · x    (rotate x by angle proportional to position)
    ⟨R(m)q, R(n)k⟩ = q^T R(m)^T R(n) k = q^T R(n−m) k
    Because R(a)^T R(b) = R(b−a)  → dot product depends ONLY on (n−m) ✓

  2D rotation (basis of RoPE):
    R(θ) × [x₁, x₂]^T = [x₁cosθ − x₂sinθ, x₁sinθ + x₂cosθ]^T

  RoPE generalization to d_head dimensions:
    Form d_head/2 pairs: (0,1), (2,3), ..., (d_head−2, d_head−1)
    Pair (2i, 2i+1) rotated by angle: θᵢ(pos) = pos / θ_base^(2i/d_head)

    Frequencies:   θᵢ = 1 / θ_base^(2i / d_head)
    High-freq (small i): fast rotation → encodes short-range positions
    Low-freq  (large i): slow rotation → encodes long-range positions

  Implementation (no explicit matrix):
    q_rot[2i]   = q[2i]   cos(m·θᵢ) − q[2i+1] sin(m·θᵢ)
    q_rot[2i+1] = q[2i]   sin(m·θᵢ) + q[2i+1] cos(m·θᵢ)   for all i

PRODUCTION θ_base VALUES:
  θ_base = 10,000   → max reliable context ≈   4,096 tokens (original Transformer)
  θ_base = 500,000  → max reliable context ≈ 128,000 tokens (LLaMA-3, Qwen-2.5)
  θ_base = 1,000,000 → max context ≈ 256,000+ tokens (experimental)
  Rule: doubling θ_base roughly doubles reliable context length.
```

2.6.2 YaRN: Extending Context Without Full Retraining

```
YARN CONTEXT EXTENSION (Peng et al. 2023):

  Problem: Model trained at L_train → θ frequencies calibrated for that range.
  At L_test >> L_train, low-frequency dimensions see unseen rotation magnitudes.

  YaRN solution: scale θ_base to reduce rotation speed for low-freq dimensions.
  For dimension pair i, define wavelength λᵢ = 2π / θᵢ = 2π · θ_base^(2i/d_head)

  If λᵢ < r·L_train:  high-freq, no change (short-range encoding intact)
  If λᵢ > s·L_train:  low-freq, scale θᵢ → θᵢ / (L_test / L_train)
  Otherwise:           interpolate between the two regimes

  YaRN requires only fine-tuning (1000–5000 steps), NOT full retraining.
  LLaMA-3 uses θ_base=500,000 to natively support 128K tokens.
```

PE Method	Used By	Context Extension	FLOPs	Production Status
Sinusoidal (fixed)	Original Transformer (Vaswani 2017)	Hard limit at training length	Zero	Legacy — not used in new models
Learned absolute	GPT-2, BERT	Hard limit; cannot generalize	Zero	Legacy
ALiBi	MPT, BLOOM	Good linear decay bias	Minimal	Declining — RoPE preferred
RoPE **PRODUCTION**	LLaMA, Mistral, Gemma, Qwen, DeepSeek	Excellent with YaRN/ LongRoPE (4–32×)	~2% of attn FLOPs	**Universal production standard**

2.7 Feed-Forward Networks and the Activation Bottleneck

```
FFN FLOP COMPARISON (LLaMA-3 70B, d_model=8192, d_ffn=28,672):

Standard FFN (GPT-style):
  FFN(x) = max(0, x·W₁ + b₁)·W₂ + b₂
  Two matrices: W₁ ∈ [d_model × d_ffn], W₂ ∈ [d_ffn × d_model]
  FLOPs: 2 × 2 × S × d_model × d_ffn (but not used in LLaMA-3)

SwiGLU FFN (LLaMA-3, Mistral, Gemma — modern standard):
  FFN(x) = (SiLU(x·W_gate) ⊗ x·W_up) × W_down
  Three matrices: W_gate, W_up ∈ [d_model × d_ffn], W_down ∈ [d_ffn × d_model]
  FLOPs: 3 × 2 × S × d_model × d_ffn = 3 × 2 × S × 8192 × 28,672

  Note: d_ffn is SMALLER in SwiGLU to compensate for third matrix:
    Standard FFN: d_ffn = 4 × d_model = 32,768
    SwiGLU FFN:   d_ffn = 8/3 × d_model ≈ 21,845 (LLaMA rounds to 28,672)
    Net FLOP cost: similar to standard FFN despite 3 matrices

Per token per layer:
  FFN FLOPs = 3 × 2 × 1 × 8192 × 28,672 = 1,409 M FLOPs
  QKV+O FLOPs (from §2.3)             =   302 M FLOPs
  Attention score FLOPs (at S=1):      =     0.13 M FLOPs (negligible)

  FFN fraction at decode S=1: 1409 / (1409 + 302) = 82.4%  ← FFN DOMINATES
  FFN fraction at prefill S=4096: 1409 / (1409 + 302 + 67) = 79%

→ Quantizing FFN weights delivers the highest bandwidth reduction per bit.
  INT4 FFN: 4× less weight bytes → 4× faster decode (bandwidth-bound).
```

2.8 Complete FLOP Budget — LLaMA-3 70B

```
FULL TRANSFORMER LAYER FLOP FORMULA:
  Model: LLaMA-3 70B — d_model=8192, L=80, H_q=64, H_kv=8, d_head=128, d_ffn=28,672

  Per transformer block, sequence length S, batch B=1:

  ① Q projection  [S×8192] × [8192×8192]:          2·S·8192²      = 134.2M·S
  ② K projection  [S×8192] × [8192×1024]:           2·S·8192·1024  =  16.8M·S
  ③ V projection  [S×8192] × [8192×1024]:           2·S·8192·1024  =  16.8M·S
  ④ QK^T scores  64 heads × [S×128] × [128×S]:     2·64·S·128·S   = 16,384·S²  ← QUADRATIC
  ⑤ A·V output   64 heads × [S×S] × [S×128]:       2·64·S·S·128   = 16,384·S²  ← QUADRATIC
  ⑥ O projection [S×8192] × [8192×8192]:            2·S·8192²      = 134.2M·S
  ⑦ FFN W_gate   [S×8192] × [8192×28672]:           2·S·8192·28672 = 469.8M·S
  ⑧ FFN W_up     [S×8192] × [8192×28672]:           2·S·8192·28672 = 469.8M·S
  ⑨ FFN W_down   [S×28672] × [28672×8192]:          2·S·28672·8192 = 469.8M·S
  ──────────────────────────────────────────────────────────────────

  At S=1 (single decode step):
    Linear ops: 134.2+16.8+16.8+134.2+469.8+469.8+469.8 = 1,711M FLOPs
    Attn (④+⑤): 32,768 FLOPs  (effectively zero vs linear)
    FFN share: 1,409 / 1,711 = 82.4%  [FFN utterly dominates decode]

  Full model (×80 layers):
    Decode total:  1,711M × 80 = 136.9 GFLOP per decode step
    Prefill S=4096: (1,711M × 4096 + 2 × 16,384 × 4096²) × 80 = 604 TFLOP

TPOT LOWER BOUND ANALYSIS (batch=1, H100):
  H100 compute:  989 TFLOPS BF16 → compute floor = 136.9 GFLOP / 989 TFLOPS = 0.14 ms
  H100 BW:       3.35 TB/s → BW floor = 140 GB / 3.35 TB/s = 41.8 ms

  BW floor / compute floor = 41.8 / 0.14 = 303×  ← decode is ALWAYS bandwidth-limited!
  Crossover batch size: ~295 (arithmetic intensity ridge of H100)

THE 6ND RULE (training FLOPs estimation):
  Training FLOPs ≈ 6 × N × D  (N=params, D=training tokens)
  LLaMA-3 70B on 15T tokens: 6 × 70×10⁹ × 15×10¹² = 6.3×10²⁴ FLOPs
  At H100 cluster (1000 GPUs, 300 TFLOP/s sustained): 21,000 GPU-days ≈ $36M

QUADRATIC CROSSOVER (attention FLOPs > FFN FLOPs):
  Attn FLOPs = FFN FLOPs when:  4·H_q·S² = 3·2·d_model·d_ffn
  LLaMA-3 70B: 4·64·S² = 1,409M → S ≈ 65,000 tokens
  → Below 65K tokens, FFN dominates. Long-context RAG/document enters attn regime.
```

2.9 Forward and Backward Pass — Memory at Every Stage

Training a transformer requires holding significantly more in memory than inference: not just the current activations, but intermediate tensors needed for gradient computation in the backward pass. Understanding memory at every stage is essential for activation checkpointing decisions, pipeline parallelism design, and gradient accumulation strategy.

2.9.1 Full Forward Pass Data Flow

```
FORWARD PASS MEMORY — single transformer block (LLaMA-3 70B, B=4, S=2048, BF16):

STAGE              OPERATION                        TENSOR SIZE
──────────────────────────────────────────────────────────────────────
1. Input           x_l ∈ R^{B×S×d}                 4×2048×8192×2 =  128 MB
2. RMSNorm         x_norm = norm(x_l)               store: 128 MB
3. Q projection    Q = x_norm × W_Q                 4×2048×8192×2 =  128 MB
4. K projection    K = x_norm × W_K (GQA)           4×2048×1024×2 =   16 MB
5. V projection    V = x_norm × W_V (GQA)           4×2048×1024×2 =   16 MB
6. RoPE            Q_rot, K_rot (in-place)           0 extra bytes
7. FlashAttention  O = FA(Q_rot, K_rot, V)           128 MB out; S×S NEVER stored
8. Output proj     Y = O × W_O                       128 MB
9. Residual add    x_attn = x_l + Y                  128 MB (store for backward)
10. RMSNorm        x_norm2 = norm(x_attn)            store: 128 MB
11. FFN gate       gate = SiLU(x_norm2 × W_gate)     4×2048×28672×2 = 448 MB
12. FFN up         up = x_norm2 × W_up               448 MB
13. FFN hidden     h = gate ⊗ up                     448 MB (MUST store for backward)
14. FFN down       y_ffn = h × W_down                128 MB
15. Residual add   x_{l+1} = x_attn + y_ffn          128 MB (output)
──────────────────────────────────────────────────────────────────────
Activation memory WITHOUT checkpointing: ~1,850 MB per block
Activation memory WITH selective checkpointing (store x_l, x_attn only): ~256 MB/block
  → Recompute everything else during backward
  → Memory reduction: 85%  |  Compute overhead: ~30% (one extra forward pass)

FFN activations account for 76% of layer activation memory → primary recompute target
```

2.9.2 Flash Attention Backward — No S×S Storage

```
FLASH ATTENTION BACKWARD PASS:

Standard attention backward:
  Requires storing A (the S×S attention matrix) → O(S²) memory
  At S=8192, H_q=64, BF16: 64 × 8192² × 2B = 8.6 GB PER LAYER

Flash Attention backward (Dao et al. 2022):
  Stores only: Q, K, V, O (O(N) total), PLUS log-sum-exp (LSE) per row
  LSE[i] = log(Σⱼ exp(Sᵢⱼ/√d_k))  → one scalar per query position

  During backward: recompute A_ij = exp(S_ij/√d_k − LSE[i]) in SRAM tiles
  → Never materialize the full S×S matrix in HBM.

  Memory saved per layer at S=8192, BF16:
    S×S matrix: 64 × 8192² × 2B = 8.598 GB (SAVED)
    LSE:        64 × 8192 × 4B  = 2 MB     (stored)
    Net saving per layer:         8.596 GB per layer

  Flash Attention is not just an optimization — it is the enabling technology
  for any context window above ~4K tokens in training.
```

2.9.3 Pipeline Parallelism Stage Memory

```
PIPELINE PARALLELISM MEMORY (LLaMA-3 70B, P=8 stages):

  Layers per stage = 80/8 = 10 layers

  With selective checkpointing:
    Activation per stage = 10 × 256 MB = 2.56 GB
    Weight per stage = 140 GB / 8 = 17.5 GB
    Total per GPU ≈ 17.5 + 2.56 = 20.1 GB  (fits on 80 GB H100 with headroom)

  Pipeline bubble fraction:
    Bubble = (P−1) / (M + P−1)   (M = micro-batches)
    M=16, P=8: bubble = 7/23 = 30%  → use 1F1B interleaved schedule
    1F1B reduces effective bubble to ~15%
    Target M ≥ 4×P to keep bubble below 20%
```

> **Activation Memory Quick Reference (LLaMA-3 70B, per layer):**
>
> Without checkpointing: ~1,850 MB · With selective checkpointing: ~256 MB (85% reduction) · Flash Attention removes 8.6 GB of S×S storage per layer at S=8192 · FFN activations are the dominant recompute target (76% of activation memory).

2.10 Modern Variants: GQA, MQA, MLA, MoE

2.10.1 KV Cache Reduction Variants

```
KV CACHE VARIANTS — detailed comparison:

MHA (Multi-Head Attention, baseline):
  H_q query heads, H_kv = H_q key-value heads
  KV bytes/token/layer = 2 × H_kv × d_head × 2 = 2 × 64 × 128 × 2 = 32,768 bytes

GQA (Grouped-Query Attention — LLaMA-3 standard):
  H_q = 64, H_kv = 8 (GQA ratio R = 8)
  Group mapping: Q-head q uses KV group q // (H_q/H_kv)
  KV bytes/token/layer = 2 × 8 × 128 × 2 = 4,096 bytes  ← 8× reduction
  Quality loss: ~0.2–0.5% at G≥8 (empirically verified at 70B+ scale)
  Production use: LLaMA-3, Mistral, Gemma, Qwen, Falcon

MQA (Multi-Query Attention, extreme):
  H_q = 64, H_kv = 1
  KV bytes/token/layer = 2 × 1 × 128 × 2 = 512 bytes  ← 64× reduction
  Quality loss: ~1–2% on complex reasoning tasks
  Use: edge inference, strict memory budgets (Gemma 2B, Falcon 7B)

MLA (Multi-head Latent Attention, DeepSeek-V2/V3):
  Compress KV to low-rank latent: c_KV = x × W_DKV  (W_DKV ∈ R^{d_model × d_c})
  Reconstruct K,V at attention time: [K,V] = c_KV × W_UKV
  Only c_KV is cached; W_UKV stays in L2/SMEM (not seq-length-dependent)

  DeepSeek-V3: d_c=512, d_rope=64, d_model=7168, H_q=128
  MLA bytes/token/layer ≈ (d_c + d_rope) × 2 = 576 × 2 = 1,152 bytes

  Comparison (DeepSeek-V3 config, S=8192, L=61, BF16):
    MHA:    61 × 2 × 128 × 128 × 8192 × 2 = 32.8 GB
    GQA R=16: 61 × 2 × 8 × 128 × 8192 × 2 =  2.05 GB
    MLA:    61 × 576 × 8192 × 2           =  576 MB  ← 57× vs MHA!
```

2.10.2 Model Family Reference Table

Model	H_q	H_kv	Variant	KV Reduction vs MHA	Notes
LLaMA-2 7B	32	32	MHA	1× (baseline)	Legacy baseline
LLaMA-3 8B	32	8	GQA	4×	—
LLaMA-3 70B	64	8	GQA	**8×**	Production standard
LLaMA-3.1 405B	128	8	GQA	16×	—
Mistral 7B	32	8	GQA	4×	—
Gemma 2B	8	1	MQA	8× (edge)	Mobile focused
DeepSeek-V3	128	—	MLA	**57× vs MHA**	d_c=512, d_rope=64

2.10.3 Causal Masking — The Root of KV Cache

```
CAUSAL MASKING AND THE KV CACHE OPPORTUNITY:

  Causal mask M for decoder-only models:
    M[i,j] = 0   if j ≤ i   (position i CAN attend to position j)
    M[i,j] = -∞  if j > i   (position i CANNOT attend to future token j)

  THE INSIGHT: K[0..t-1] and V[0..t-1] are IDENTICAL for all future tokens.
  (Keys and values are deterministic functions of input tokens + model weights.)

  THEREFORE: Compute K,V once during prefill, cache and REUSE at every decode step.

  WITHOUT KV cache: generating token t requires recomputing O(t × d) work each step
  WITH KV cache:    only compute K,V for NEW token; O(t) BW to load cached K,V

  Consequence chain:
    Causal mask → stable K,V per position → cacheable
    → KV dominates inference memory → GQA/MLA reduce that cost
    → Smaller KV → more concurrent requests → higher throughput

  Encoder models (BERT): bidirectional attention → no stable K,V → no KV cache possible
  → BERT encodes whole sequence each time, never autoregressively
```

2.10.4 MoE: Mixture-of-Experts

```
MIXTURE-OF-EXPERTS (MoE) ARCHITECTURE:

  N expert FFNs, router sends each token to K of N experts (K << N)

  Mixtral 8×7B: N=8, K=2, d_ffn=14,336 per expert
  Active FLOPs per token: K/N × total_FFN_FLOPs = 2/8 = 25% ... WAIT:
    Actually: each token hits K=2 experts, each with full FFN
    Active FLOPs = K × FFN_flops_per_expert = same as 2× dense 7B FFN
    Total params: N × FFN_params + shared = 8 × 7B = 47B total, 7B active

  Router: linear layer x → softmax → top-K selection
    Router FLOPs: negligible (d_model × N = tiny vs FFN)
    Expert capacity = batch_size × K / N × capacity_factor
    Capacity factor > 1.0 needed to handle uneven routing → token dropping risk

  KEY SYSTEM COST: All-to-All collective at every FFN layer:
    Each GPU holds 1 expert (expert parallelism)
    Tokens assigned to remote experts are sent via All-to-All
    Volume: batch_size × K × d_model × 2B bytes each direction
    At batch=32, K=2, d_model=7168: 32 × 2 × 7168 × 2 = 917 KB bidirectional

  Load balancing loss: auxiliary loss to prevent routing collapse to few experts
  See Ch 13 for full MoE serving system design including expert placement.
```

2.11 Attention Variants — Full Math Reference

This section is the single-page reference for design reviews and production investigations. All attention variants, their exact memory formulas, and the canonical formula are collected here.

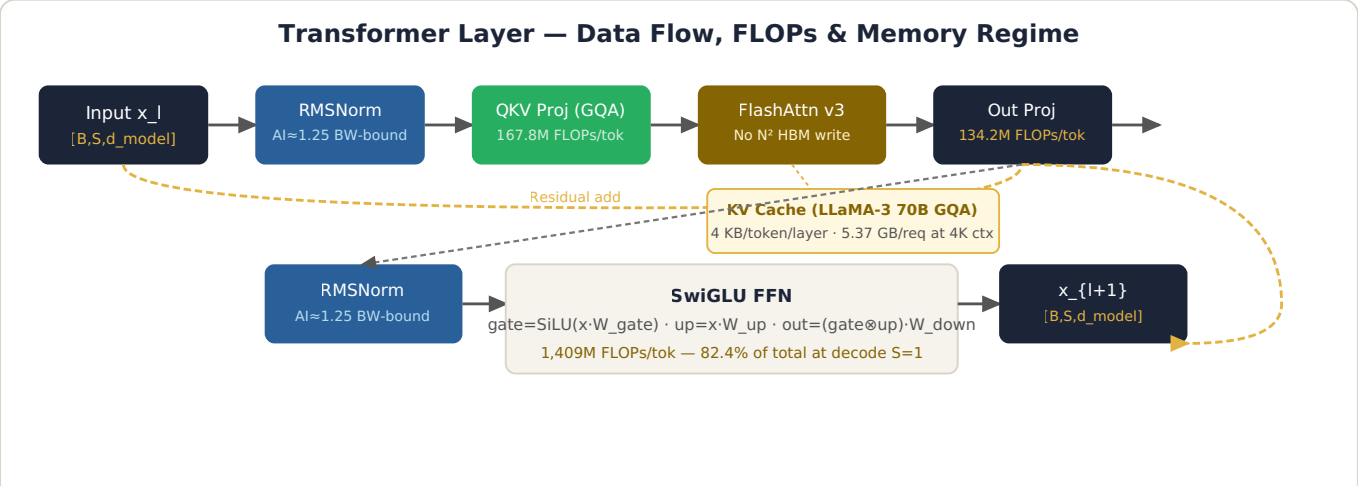
```
FULL KV CACHE MEMORY FORMULA — all variants:

  Notation: L=layers, H=Q heads, G=KV groups, d=head dim, S=seq len, B=batch, P=bytes/elem

  ┌──────────┬──────────────────────────┬──────────────────────────────┬──────────────┐
  │ Variant  │ Cache Content            │ Formula                      │ Relative     │
  ├──────────┼──────────────────────────┼──────────────────────────────┼──────────────┤
  │ MHA      │ K+V, one per Q head      │ L×2×H×d×S×B×P                │ 1× (base)    │
  │ MQA      │ K+V, one shared head     │ L×2×1×d×S×B×P                │ 1/H          │
  │ GQA      │ K+V, G shared groups     │ L×2×G×d×S×B×P                │ G/H = 1/R    │
  │ MLA      │ c_KV + k_rope            │ L×(d_c + d_rope)×S×B×P       │ varies       │
  └──────────┴──────────────────────────┴──────────────────────────────┴──────────────┘

  Concrete numbers (L=80, H=64, d=128, BF16, S=8192, B=1):
  MHA:        80 × 2 × 64 × 128 × 8192 × 2 = 27.5 GB
  MQA:        80 × 2 × 1  × 128 × 8192 × 2 =  430 MB
  GQA (G=8):  80 × 2 × 8  × 128 × 8192 × 2 =  3.4 GB   ← standard LLaMA-3 70B
  MLA (est):  80 × 576 × 8192 × 2          =  754 MB

VARIANT DECISION TABLE:
  Variant | KV Bytes/Tok/Layer | Quality vs MHA  | Extra Compute    | Best Use
  MHA     | 2 × H × d × P     | Baseline        | None             | Small models, training
  MQA     | 2 × d × P         | −1–2% reasoning | None             | Edge, strict budget
  GQA     | 2 × G × d × P     | −0.2–0.5% (G≥8) | None             | Production 70B+
  MLA     | (d_c+d_rope) × P  | ~0% (MHA-level) | Up-proj GEMM     | Frontier + memory
```



igure 2.11 — One transformer layer data flow. FFN dominates at 82.4% of decode FLOPs. FlashAttention eliminates N² HBM traffic. KV cache persists across decode steps

Figure 2.11 — Transformer layer anatomy with FLOP breakdown. At decode batch=1, the FFN accounts for 82.4% of FLOPs and all operations are memory-bandwidth-bound. At large-batch prefill, large GEMM operations become compute-bound.

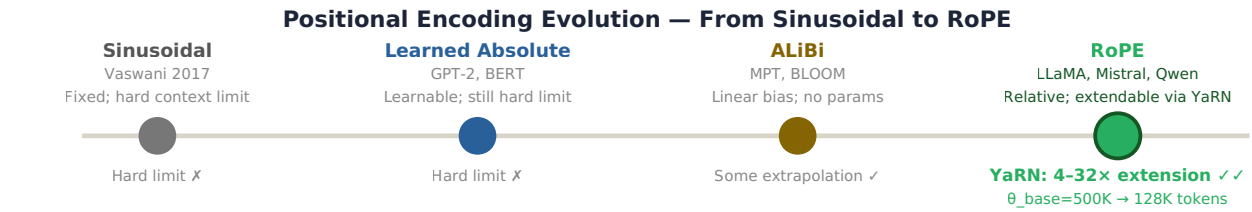


Figure 2.12 — Positional encoding evolution. RoPE with YaRN is the universal production standard for context-length extension in modern LLMs.

Figure 2.12 — Positional encoding evolution from sinusoidal to RoPE. RoPE's relative-position property enables context extension via frequency rescaling (YaRN) without full retraining.



Figure 2.13 — KV cache memory at S=8192 BF16. GQA (standard) already delivers 8× over MHA; MQA and MLA push further with quality/compute tradeoffs.

Figure 2.13 — KV cache memory comparison. The engineering decision between MHA/GQA/MQA/MLA determines concurrent requests per GPU and thus cost per inference.

Key Takeaways

CORE ARCHITECTURE CONCEPTS

- The FFN consumes 82.4% of FLOPs at decode (batch=1) and 79% at prefill. This is why FFN weight quantization (INT4/FP8) has the highest bandwidth ROI of any quantization decision.
- KV cache size = 2 × L × H_kv × d_head × S × dtype_bytes. LLaMA-3 70B at 4K context in BF16: 5.37 GB per concurrent request. GQA (H_kv=8 vs H_q=64) delivers 8× reduction with negligible quality loss — directly 8× more concurrent users per GPU.
- Decoder-only LLMs (GPT, LLaMA, Claude) use causal masking to enable KV caching. Bidirectional encoder models (BERT) cannot use a KV cache — each inference must recompute attention from scratch.
- Causal masking is not just a quality choice — it is the architectural root of the entire KV cache optimization chain: causal mask → fixed K,V per position → cacheable → GQA/MLA reduce that cache cost → more concurrent requests.

PERFORMANCE & BANDWIDTH BOTTLENECKS

- Decode at batch=1 TPOT is always bandwidth-limited, never compute-limited. H100 bandwidth floor (41.8ms for 70B BF16) is 303× the compute floor (0.14ms). No kernel optimization helps — only quantization (fewer bytes) or batching (higher arithmetic intensity).
- Flash Attention removes the N^2 HBM materialization, saving 8.6 GB per layer at S=8192. This is not optional above 4K context in training — it is the enabling technology for long-context models.
- Activation checkpointing trades 30% extra compute for 85% activation memory reduction per layer. FFN activations (76% of layer memory) are the primary recompute target.

PRINCIPAL-LEVEL ARCHITECTURE INSIGHTS

- Every attention variant is an infrastructure cost decision: MHA vs GQA vs MLA determines KV cache size, which determines serving capacity per GPU. A 1% quality improvement requiring 4× more GPUs costs ~$100K/year on a 100-GPU serving cluster.
- MLA (DeepSeek) reduces KV cache 57× vs MHA but adds an up-projection GEMM at each decode step. This GEMM is compute-bound at batch=1 — trading bandwidth cost for compute cost. The tradeoff breaks even around batch~8.
- MoE routing introduces All-to-All communication at every FFN layer. At top-2/8 routing with expert parallelism, 75% of expert compute happens remotely. Communication overhead determines whether MoE saves cost or adds it — see Ch 13.
- The 6ND rule (training FLOPs ≈ 6 × N × D) lets you estimate full pretraining cost before writing a line of code. LLaMA-3 70B on 15T tokens ≈ $36M compute at H100 cluster pricing.

Review Questions

FOUNDATIONAL Build the mental model — no math required

1. In plain language, explain why a transformer with GQA (H_kv=8) can serve more concurrent users than one with MHA (H_kv=64), even though both produce the same output quality at the same sequence length.
2. What is the attention KV cache, and why does it grow with sequence length during autoregressive decoding? Give a concrete example with numbers for a 70B model at 1000 tokens context.
3. Why did RMSNorm replace LayerNorm in modern LLMs? Name two reasons — one computational, one system-level — and explain why both matter for production inference.
4. Explain the difference between encoder-only (BERT), encoder-decoder (T5), and decoder-only (LLaMA) architectures. Which can use a KV cache and why? What does bidirectional attention give up compared to causal masking?

INTERMEDIATE Apply and calculate — production context

1. For LLaMA-3 70B (L=80, d=8192, H_q=64, H_kv=8, d_ffn=28672, BF16) serving 50 concurrent requests at S=2048: (a) Compute total KV cache memory. (b) H100 has 80 GB HBM; weights take 140 GB with TP=2, runtime overhead 10 GB. How many H100s needed? (c) Switch to FP8 KV — how many H100s now?

2. Compute arithmetic intensity of the FFN down projection (W_down: [28672 × 8192]) at batch=1 vs batch=64, BF16. Which is memory-bound on H100 (ridge 295 FLOP/byte)? At what exact batch size does this operation cross the ridge?

3. A model is trained at 4K context using θ_base=10,000 in RoPE. You want to extend to 128K context using YaRN. (a) What is the new θ_base needed? (b) Which RoPE frequency dimensions are most affected? (c) Why does YaRN require only fine-tuning (~1000 steps) rather than full retraining?

PRINCIPAL LEVEL Design, decide, and defend — architecture reviews

1. Your team is choosing between GQA (H_kv=8), MQA (H_kv=1), and MLA (d_c=512) for a new 13B model targeting 100K token context serving. Quantify: (a) KV cache bytes per request at 100K tokens for each variant. (b) Available concurrent requests on a single H100 (80 GB, 26 GB weights). (c) Reconstruction compute cost for MLA at batch=32. Recommend one variant with quantitative justification.

2. Design training infrastructure for LLaMA-3 70B: (a) Memory budget per GPU with 8-way tensor parallelism, batch=8, S=4096, with and without activation checkpointing. (b) At what pipeline parallelism degree and micro-batch count does bubble overhead drop below 15%? (c) Flash Attention backward saves how much HBM per layer at this sequence length? Show all calculations.