LLM Inference System Characteristics

LLM Inference: Overview

What is LLM Inference?

Given a prompt, an LLM predicts the **next token**:

```
"The capital of France is"
    → Tokenizer → [token IDs]
    → Embedding Lookup → [vectors]
    → Transformer Layer 1 → ... → Layer 32
    → LM Head → [logits over vocabulary]
    → Sample → "Paris"
```

What is LLM Inference?

Given a prompt, an LLM predicts the **next token**:

```
"The capital of France is"
    → Tokenizer → [token IDs]
    → Embedding Lookup → [vectors]
    → Transformer Layer 1 → ... → Layer 32
    → LM Head → [logits over vocabulary]
    → Sample → "Paris"
```

Key insight: LLMs generate text **one token at a time** (autoregressive).

Autoregressive Generation

Each generated token becomes input for the next. We cannot generate token $n+1$ until we've generated token n.

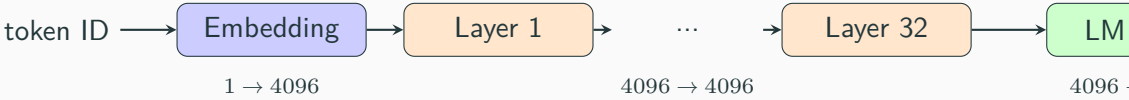
Running Example: Llama-3.1-8B

Parameter	Value	Description
Layers	32	Number of transformer layers
Hidden dimension (d)	4,096	Size of token representations
Query heads	32	Number of attention heads for queries
KV heads	8	Number of KV heads (GQA)
Head dimension	128	Dimension per attention head
FFN dimension	14,336	Intermediate size in feed-forward network
Vocabulary size	128,256	Number of possible tokens

Total: ~8 billion parameters

Model Architecture

High-Level Dataflow



- **Embedding:** Token ID $\to$ 4,096-dim vector (pure memory lookup)
- **32 Transformer Layers:** Attention + FFN
- **LM Head:** Project to vocabulary size, sample next token

Transformer Layer Structure

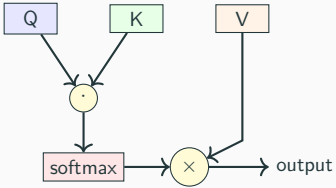
Two blocks per layer:

1. **Attention block:** RMSNorm $\rightarrow$ Self-Attention $\rightarrow$ Residual add
2. **FFN block:** RMSNorm $\rightarrow$ SwiGLU FFN $\rightarrow$ Residual add

Attention Mechanism

Self-attention allows each token to gather information from all previous tokens:

1. **Projections:** $Q = xW_Q$, $K = xW_K$, $V = xW_V$
2. **Scores:** $\text{softmax}(QK^\top / \sqrt{d_h})$
3. **Output:** Weighted sum of values, then W_O projection



Multi-Head Attention Variants

The KV cache problem: During generation, we cache K and V to avoid recomputation.

Variant	Q Heads	KV Heads	KV Cache Size	Quality
MHA	32	32	$1\times$ (baseline)	Best
MQA	32	1	$1/32\times$	Lower
GQA	32	8	$1/4\times$	Good

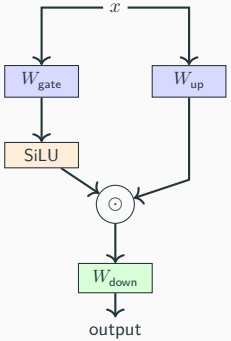
Llama-3.1-8B uses GQA: Each of 8 KV heads is shared by 4 query heads.

$\Rightarrow$ $4\times$ less KV cache memory, ~same quality

Feed-Forward Network (SwiGLU)

$$\text{FFN}(x) = W_{\text{down}} \cdot (\text{SiLU}(W_{\text{gate}} \cdot x) \odot W_{\text{up}} \cdot x)$$

- $W_{\text{gate}}, W_{\text{up}}$: $4096 \to 14336$
- W_{down}: $14336 \to 4096$
- SiLU: $z \cdot \sigma(z)$
- $\odot$: element-wise multiply (gating)



Local vs. Global Operations

Key insight: Attention is the *only* operation where tokens interact!

Operation	Scope	Description
Embedding	Per-token	Each token ID maps to its own vector
RMSNorm	Per-token	Normalizes each token independently
Q, K, V projections	Per-token	Matrix-vector multiply per token
FFN (SwiGLU)	Per-token	Same transformation for each token
Attention	**Global**	**Each token attends to all previous**

Local vs. Global Operations

Key insight: Attention is the *only* operation where tokens interact!

Operation	Scope	Description
Embedding	Per-token	Each token ID maps to its own vector
RMSNorm	Per-token	Normalizes each token independently
Q, K, V projections	Per-token	Matrix-vector multiply per token
FFN (SwiGLU)	Per-token	Same transformation for each token
Attention	**Global**	**Each token attends to all previous**

Token-independent ops are embarrassingly parallel. Attention is what makes LLMs understand context.

KV Cache, Prefill, and Decode

The Redundancy Problem

When generating token $t+1$, naive approach processes entire sequence:

$$[x_1, x_2, \ldots, x_t, x_{t+1}]$$

The Redundancy Problem

When generating token $t+1$, naive approach processes entire sequence:

$$[x_1, x_2, \ldots, x_t, x_{t+1}]$$

But we already computed $K_1, K_2, \ldots, K_t$ and $V_1, V_2, \ldots, V_t$ in previous steps!

The Redundancy Problem

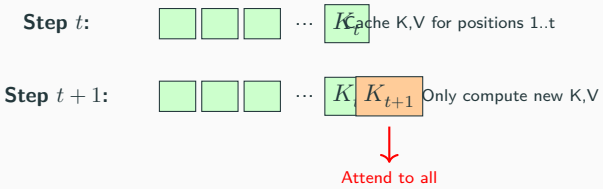
When generating token $t+1$, naive approach processes entire sequence:

$$[x_1, x_2, \ldots, x_t, x_{t+1}]$$

But we already computed $K_1, K_2, \ldots, K_t$ and $V_1, V_2, \ldots, V_t$ in previous steps!

Solution: Cache keys and values, reuse them.

KV Cache



KV cache trades compute for memory:

- Store K and V for all previous tokens
- Each new token: compute only its K, V, append to cache
- Attend using the full cache

Two Phases of Inference

Prefill (prompt processing)

- Process p prompt tokens in parallel
- Compute and cache all K, V
- Apply **causal mask** (token i only sees $1..i$)
- **Compute-bound** (matrix-matrix ops)

Decode (token generation)

- Process 1 token at a time
- Compute new K, V; append to cache
- Attend to entire cache
- **Memory-bound** (matrix-vector ops)

Phase	Tokens/step	Bottleneck	Arithmetic Intensity
Prefill	p	Compute	High
Decode	1	Memory bandwidth	Low (~1)

KV Cache Memory Requirements

For Llama-3.1-8B with GQA:

$$\text{KV cache per token} = 2 \times n_{\text{layers}} \times n_{\text{kv\_heads}} \times d_h \times \text{bytes}$$
$$= 2 \times 32 \times 8 \times 128 \times 2 = 128 \text{ KB (FP16)}$$

KV Cache Memory Requirements

For Llama-3.1-8B with GQA:

$$\text{KV cache per token} = 2 \times n_{\text{layers}} \times n_{\text{kv\_heads}} \times d_h \times \text{bytes}$$
$$= 2 \times 32 \times 8 \times 128 \times 2 = 128 \text{ KB (FP16)}$$

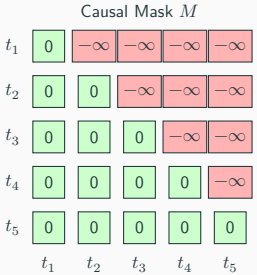
Context Length	KV Cache Size
4K tokens	0.5 GB
8K tokens	1 GB
32K tokens	4 GB
128K tokens	**16 GB**

At 128K context, KV cache equals model weight size!

Causal Masking in Prefill

During prefill, we process all tokens in parallel but enforce causality:

$$\text{MaskedAttn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_h}} + M\right)V$$



After softmax: $-\infty \to 0$

Token i only attends to $1..i$

Arithmetic Intensity Analysis

Recap: Arithmetic Intensity

$$\text{AI} = \frac{\text{FLOPs}}{\text{Bytes transferred}}$$

A100 GPU: Peak = 312 TFLOPS / 2 TB/s = **156 FLOP/byte**

Recap: Arithmetic Intensity

$$\text{AI} = \frac{\text{FLOPs}}{\text{Bytes transferred}}$$

A100 GPU: Peak = 312 TFLOPS / 2 TB/s = **156 FLOP/byte**

- AI $>$ 156: **Compute-bound** (GPU busy computing)
- AI $<$ 156: **Memory-bound** (GPU waiting for data)

Linear Layers: Prefill vs Decode

For weight matrix $W \in \mathbb{R}^{d_{in} \times d_{out}}$:

Phase	Input Shape	FLOPs	Bytes (weights)	AI
Prefill	(p, d_{in})	$2 \cdot p \cdot d_{in} \cdot d_{out}$	$2 \cdot d_{in} \cdot d_{out}$	$\approx p$
Decode	$(1, d_{in})$	$2 \cdot d_{in} \cdot d_{out}$	$2 \cdot d_{in} \cdot d_{out}$	≈ 1

Linear Layers: Prefill vs Decode

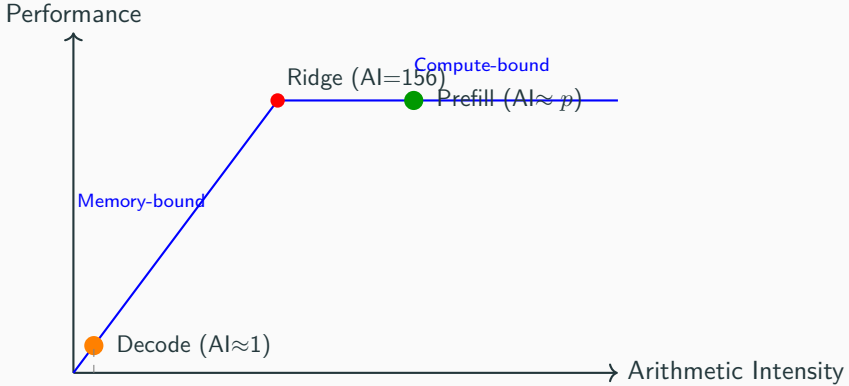
For weight matrix $W \in \mathbb{R}^{d_{in} \times d_{out}}$:

Phase	Input Shape	FLOPs	Bytes (weights)	AI
Prefill	(p, d_{in})	$2 \cdot p \cdot d_{in} \cdot d_{out}$	$2 \cdot d_{in} \cdot d_{out}$	$\approx p$
Decode	$(1, d_{in})$	$2 \cdot d_{in} \cdot d_{out}$	$2 \cdot d_{in} \cdot d_{out}$	≈ 1

Prefill: Amortize weight loading across p tokens $\Rightarrow$ AI $\approx p$

Decode: Each weight loaded once, used once $\Rightarrow$ AI ≈ 1

Decode is Memory-Bound



Decode runs at <1% GPU utilization for linear layers!

Attention Arithmetic Intensity

Prefill attention (processing p tokens):

$$\text{AI}_{\text{prefill}} = \frac{p \cdot d}{d + d_{kv}} \approx 0.8p \quad \text{(for Llama-3.1-8B)}$$

Attention Arithmetic Intensity

Prefill attention (processing p tokens):

$$\text{AI}_{\text{prefill}} = \frac{p \cdot d}{d + d_{kv}} \approx 0.8p \quad \text{(for Llama-3.1-8B)}$$

Decode attention (1 token, s cached tokens):

$$\text{AI}_{\text{decode}} = \frac{n_q}{n_{kv}} = \frac{32}{8} = 4 \quad \text{(GQA helps!)}$$

Attention Arithmetic Intensity

Prefill attention (processing p tokens):

$$\text{AI}_{\text{prefill}} = \frac{p \cdot d}{d + d_{kv}} \approx 0.8p \quad \text{(for Llama-3.1-8B)}$$

Decode attention (1 token, s cached tokens):

$$\text{AI}_{\text{decode}} = \frac{n_q}{n_{kv}} = \frac{32}{8} = 4 \quad \text{(GQA helps!)}$$

Compare to MHA: decode AI would be 1, not 4. **GQA improves decode efficiency.**

Summary: Decode is the Bottleneck

Operation	Prefill AI	Decode AI
Linear layers	$\approx p$	≈ 1
Attention	$\approx 0.8p$	$\approx n_q/n_{kv}$
Normalization	≈ 4	≈ 4

With prompt length $p = 1000$: prefill is **compute-bound**.

At decode: **everything is memory-bound**.

Summary: Decode is the Bottleneck

Operation	Prefill AI	Decode AI
Linear layers	$\approx p$	≈ 1
Attention	$\approx 0.8p$	$\approx n_q/n_{kv}$
Normalization	≈ 4	≈ 4

With prompt length $p = 1000$: prefill is **compute-bound**.

At decode: **everything is memory-bound**.

The fundamental challenge: Loading 16 GB of weights for *one* token's computation.

Batching: The Key Optimization

Why Batching Works

At batch size 1, each weight is loaded once, used once (AI ≈ 1).

Batching: Process B sequences simultaneously, reuse weights B times.

Batch Size	FLOPs	Bytes	AI
1	$2 \cdot d_{in} \cdot d_{out}$	$2 \cdot d_{in} \cdot d_{out}$	1
B	$B \cdot 2 \cdot d_{in} \cdot d_{out}$	$2 \cdot d_{in} \cdot d_{out}$	B

Why Batching Works

At batch size 1, each weight is loaded once, used once (AI ≈ 1).

Batching: Process B sequences simultaneously, reuse weights B times.

Batch Size	FLOPs	Bytes	AI
1	$2 \cdot d_{in} \cdot d_{out}$	$2 \cdot d_{in} \cdot d_{out}$	1
B	$B \cdot 2 \cdot d_{in} \cdot d_{out}$	$2 \cdot d_{in} \cdot d_{out}$	B

Arithmetic intensity scales linearly with batch size!

To become compute-bound on A100: need $B \approx 156$.

Attention Doesn't Batch Well

For attention, each sequence has its own KV cache:

Batch Size	FLOPs	Bytes (KV cache)	AI
1	$4 \cdot s \cdot d$	$4 \cdot s \cdot n_{kv} \cdot d_h$	n_q/n_{kv}
B	$B \cdot 4 \cdot s \cdot d$	$B \cdot 4 \cdot s \cdot n_{kv} \cdot d_h$	n_q/n_{kv}

Attention Doesn't Batch Well

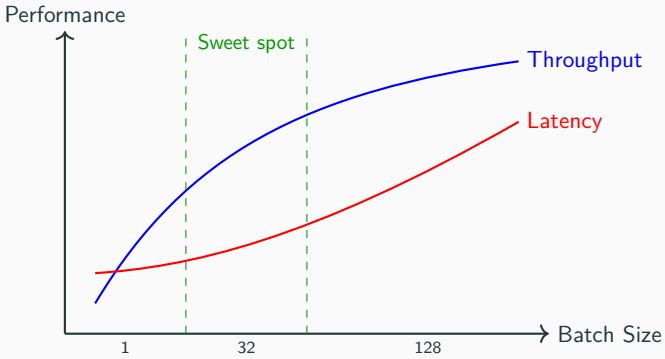
For attention, each sequence has its own KV cache:

Batch Size	FLOPs	Bytes (KV cache)	AI
1	$4 \cdot s \cdot d$	$4 \cdot s \cdot n_{kv} \cdot d_h$	n_q/n_{kv}
B	$B \cdot 4 \cdot s \cdot d$	$B \cdot 4 \cdot s \cdot n_{kv} \cdot d_h$	n_q/n_{kv}

Batching does NOT improve attention's AI — both compute and memory scale with B.

Exception: **prefix sharing** (shared system prompts) does help!

The Batching Trade-off



- **Throughput** (tokens/sec total): improves with batch size
- **Latency** (time per request): barely increases at small batches!

At small B: GPU is memory-bound anyway—extra work is "free".

Memory Limits Batch Size

For Llama-3.1-8B at 4K context:

- Model weights: ~16 GB
- KV cache per sequence: ~0.5 GB
- On 80 GB GPU: room for ~128 concurrent sequences

Memory Limits Batch Size

For Llama-3.1-8B at 4K context:

- Model weights: ~16 GB
- KV cache per sequence: ~0.5 GB
- On 80 GB GPU: room for ~128 concurrent sequences

Practical batch sizes: 32–64 sequences

Key insight: Batching is nearly universal in LLM serving because it improves throughput with minimal latency penalty.

System Challenges from Batching

1. **Variable sequence lengths:** Requests have different prompt/output lengths
   - Solution: *Continuous batching* — dynamically add/remove requests
2. **Memory fragmentation:** KV caches grow unpredictably
   - Solution: *PagedAttention* — manage KV cache in fixed blocks
3. **Prefill-decode interference:** Different compute characteristics
   - Solution: *Chunked prefill* — interleave carefully

System Challenges from Batching

1. **Variable sequence lengths:** Requests have different prompt/output lengths
   - Solution: *Continuous batching* — dynamically add/remove requests
2. **Memory fragmentation:** KV caches grow unpredictably
   - Solution: *PagedAttention* — manage KV cache in fixed blocks
3. **Prefill-decode interference:** Different compute characteristics
   - Solution: *Chunked prefill* — interleave carefully

These techniques form the core of vLLM, TensorRT-LLM, and SGLang.

Architectural Variants

Mixture of Experts (MoE)

Replace dense FFN with E expert FFNs; router selects top-k per token:

$$\text{MoE}(x) = \sum_{i \in \text{TopK}(r(x))} g_i \cdot \text{FFN}_i(x)$$

Model	Total Params	Active Params	Experts	Top-K
Llama-3.1-8B	8B	8B	1 (dense)	—
Mixtral 8x7B	47B	13B	8	2
DeepSeek-V2	236B	21B	160	6

MoE decouples total parameters from active parameters.

MoE Trade-offs

Benefits:

- More parameters = better quality
- Same compute per token as smaller dense model
- 8× parameters, 2× compute (top-2)

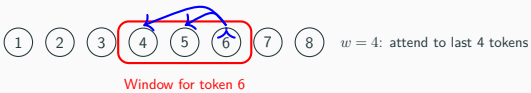
Challenges:

- All expert weights must fit in memory
- Load balancing: want even routing
- Communication in distributed settings

Sliding Window Attention

Standard attention: $O(n^2)$ — each token attends to all previous.

Sliding window: Limit attention to last w tokens:



- Mistral 7B: $w = 4096$
- Bounds KV cache and compute regardless of sequence length
- Information propagates through layer stacking

Multi-head Latent Attention (MLA)

MLA (DeepSeek-V2) compresses K, V into low-dimensional latent:

$$c = W_{\text{compress}} \cdot x, \quad K = W_K \cdot c, \quad V = W_V \cdot c$$

Cache c instead of K and V.

Method	KV Cache per Token
MHA	$2 \cdot n_h \cdot d_h$
GQA	$2 \cdot n_{kv} \cdot d_h$ (4× less)
MLA	d_c (~5× less)

Trade-off: Extra compute to decompress during attention.

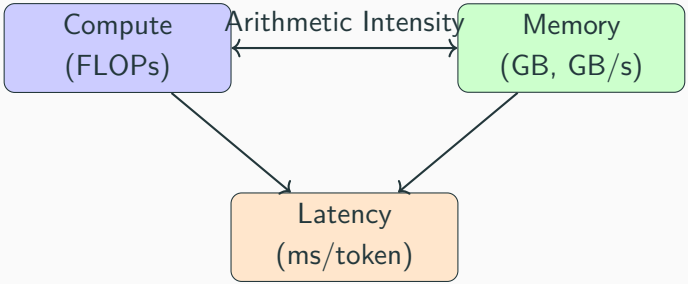
Summary

Key Takeaways

1. **LLMs generate tokens autoregressively** — sequential dependency limits parallelism
2. **Attention is the only global operation** — everything else is per-token
3. **Two phases:** Prefill (compute-bound, parallel) vs Decode (memory-bound, sequential)
4. **KV cache** trades compute for memory — critical for long contexts
5. **Decode is memory-bound** (AI 1) — loading weights dominates
6. **Batching** improves throughput by reusing weights across sequences

The Fundamental Trade-off

LLM inference is dominated by memory bandwidth



Data movement is the bottleneck — just like CPUs, but at a different scale.