Large Language Model Inference

Transformer Architecture, Memory Management, Batching, KV Cache, PagedAttention, Efficient Attention Mechanisms & vLLMs

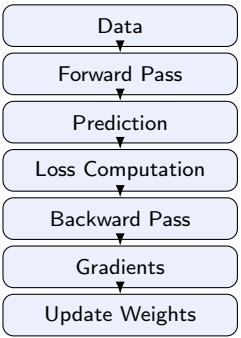
Harsh Raj - IIT Mandi - GroupBrain Talks

14 September 2026

Outline

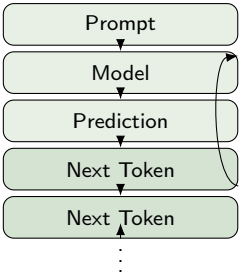
Training vs. Inference

Training



Processes many tokens **in parallel** via batching.

Inference



Autoregressive: token $t+1$ needs token t – so generation is inherently sequential by architecture.

Why Inference Is Hard

1. **Sequential dependency -> Autoregressive Nature**
   Token $t+1$ cannot be computed until token t exists – the decode loop cannot be parallelized across the sequence dimension.

2. **Growing state**
   Naively, producing token $t+1$ re-attends over all t prior tokens: $O(t)$ per step, $O(n^2)$ over a full sequence. The **KV cache** fixes the recomputation – but introduces its own memory-growth problem.

3. **Bandwidth, not compute, is usually the bottleneck during Inference.**
   Decoding streams the full weight matrix from HBM (High-Bandwidth Memory) to produce just *one* token's worth of work – decode is **memory-bandwidth-bound**, not compute-bound.

Two GPU Resources

Compute Capacity

Trillions of operations per second (TFLOP/s) the tensor cores can perform.

Memory Bandwidth

TB/sec the GPU can move data between HBM and the compute units.

Practical Consequence

During decode, a huge amount of weight data must be read for very little arithmetic per byte moved – the GPU sits idle waiting on memory, not compute.

The Central Question

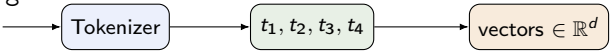
Key Idea

Nearly every inference optimization technique answers one of two questions:

1. How do we avoid moving bytes we don't need to move?
   (KV cache management, quantization, prefix caching, attention kernels)
2. How do we keep the accelerator busy while a user waits?
   (batching, scheduling, parallelism, speculative decoding)

From Text to Hidden States

Input: `"I love machine learning"`



Tokenizer: converts between raw Unicode text and a sequence of integer **token IDs** (and back). It splits text into reusable pieces called **tokens**—characters, words, subwords, or punctuation.

Example: "machine learning" $\longrightarrow$ `[machine, learning]` $\longrightarrow$ $[t_1, t_2]$

A good tokenizer represents the same text using relatively few tokens. This matters because attention becomes more expensive as sequence length grows. **Byte Pair Encoding (BPE)** is one common tokenizer.

$$\boxed{\text{Compression Ratio} = \frac{\#\text{ bytes in input}}{\#\text{ tokens produced}}}$$

Higher ratio $\Rightarrow$ fewer tokens per byte

Token vectors: Stack n token vectors of dimension d into a matrix:

$$X \in \mathbb{R}^{n \times d} = \begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1d} \\ h_{21} & h_{22} & \cdots & h_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \cdots & h_{nd} \end{bmatrix}$$

(n tokens, d-dimensional vectors)

These per-token vectors are the model's **hidden states**.

Why Do We Need Attention?

"The animal didn't cross the road because **it** was tired."

What does *"it"* refer to? Almost certainly *the animal* – but resolving that requires looking at other words in the sentence.

Attention, Intuitively

It is a mechanism that lets each token ask: *"Which other tokens should I pay attention to?"* – i.e. a way for tokens to exchange information with one another.

Query, Key, Value

The Transformer creates three projections of every token using learned matrices W_Q, W_K, W_V (learned during training):

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V$$

Query
"What am I looking for?"

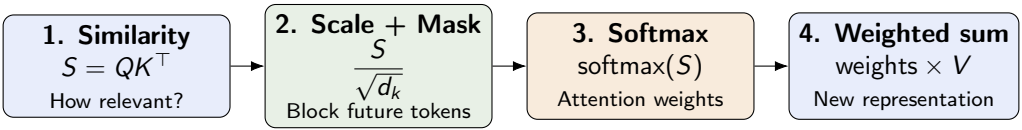
Key
"What do I offer for matching?"

Value
"Here is my information"

Example sentence: `The cat drank milk because it was thirsty.` Suppose we are processing **"it"**.

- **Query:** "What is relevant?"
- Compare its Query with the **Keys** of other tokens.
- "cat" is a strong match.
- Take useful information from "cat's" **Value**.

Computing Attention



1. **Similarity**: $QK^\top$ compares every token's Query with every token's Key, producing an $n \times n$ **attention-score matrix**.
2. **Scale + causal mask**: divide by $\sqrt{d_k}$ to keep dot products at a reasonable scale; mask positions $j > i$ so token i cannot see future tokens.
3. **Softmax**: converts the masked scores into non-negative values that sum to 1: the **attention weights**.
4. **Weighted combination**: use these weights to take a weighted sum of the **Value vectors**. This produces the new representation for each token.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\text{mask}\left(\frac{QK^\top}{\sqrt{d_k}}\right)\right)V$$

Worked Example: Computing Attention

Consider a sequence of $n = 3$ tokens with $d_k = 2$:

$$Q = K = V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

1. **Similarity scores:**

$$QK^\top = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

2. **Scale by $\sqrt{d_k} = \sqrt{2}$:**

$$\frac{QK^\top}{\sqrt{2}} \approx \begin{bmatrix} 0.71 & 0 & 0.71 \\ 0 & 0.71 & 0.71 \\ 0.71 & 0.71 & 1.41 \end{bmatrix}$$

3. **Causal mask:** token i cannot see tokens $j > i$.

Worked Example: Softmax → Attention Weights

Starting with the masked scores:

$$M = \begin{bmatrix} 0.71 & -\infty & -\infty \\ 0 & 0.71 & -\infty \\ 0.71 & 0.71 & 1.41 \end{bmatrix}$$

1. **Softmax** converts each row of scores into attention weights.

$$\boxed{\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}}$$

2. For example, for **token 2**:

$$\text{softmax}\begin{pmatrix} 0 \\ 0.71 \\ -\infty \end{pmatrix} = \begin{pmatrix} \frac{e^0}{e^0 + e^{0.71}} \\ \frac{e^{0.71}}{e^0 + e^{0.71}} \\ 0 \end{pmatrix} \approx \begin{pmatrix} 0.33 \\ 0.67 \\ 0 \end{pmatrix}$$

Worked Example: Weighted Sum of Values

Goal: Use attention weights to combine the Value vectors.

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0.33 & 0.67 & 0 \\ 0.25 & 0.25 & 0.50 \end{bmatrix}, \qquad V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

Weighted combination:

$$AV = \begin{bmatrix} 1 & 0 & 0 \\ 0.33 & 0.67 & 0 \\ 0.25 & 0.25 & 0.50 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} = \boxed{\begin{bmatrix} 1 & 0 \\ 0.33 & 0.67 \\ 0.75 & 0.75 \end{bmatrix}}$$

Example: Token 2

$$0.33[1, 0] + 0.67[0, 1] = [0.33,\ 0.67]$$

Key points

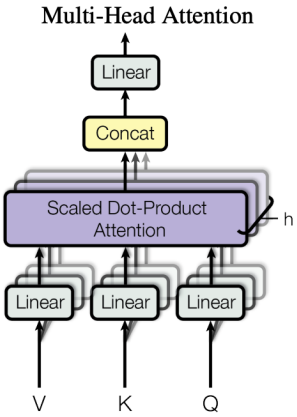
- **Self-attention has no parameters:** interactions are determined by embeddings and positional encodings only.
- **Diagonal values are expected to be high:** a token can attend strongly to itself.
- **Preventing interactions:** set the corresponding attention score to $-\infty$ **before softmax**.
- After softmax, this gives probability 0, so the model cannot learn that interaction. This is used in the **decoder**.

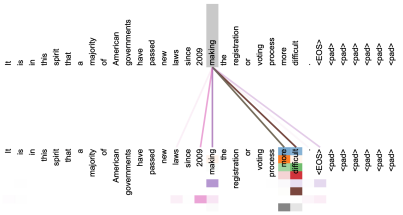
Each output token is a weighted combination of the Value vectors.

Multi-Head Attention

Idea:
Multiple attention heads run in parallel. Each head uses its own lower-dimensional projections of Q, K, and V.

- Each head can learn different relationships.
- The model learns what each head should focus on.
- Outputs from all heads are concatenated.
- A final linear layer combines them.



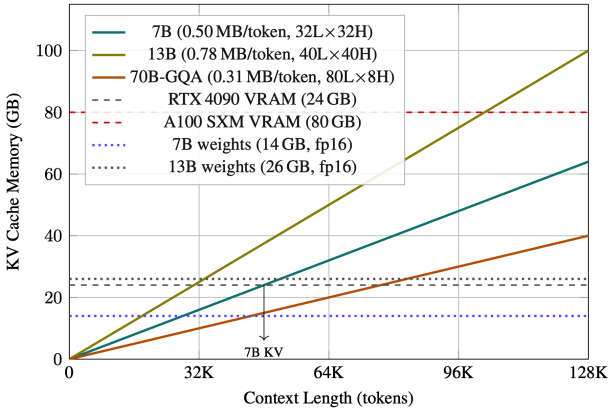


Example attention pattern:
Different attention connections indicate which tokens attend to which other tokens.

KV Caching – The Core Trick

Key Idea

Once a token's K and V vectors have been computed, they never need to be recomputed later – store them and reuse them at every subsequent decode step.



Two Phases of Generation: Prefill vs. Decode

	Prefill	**Decode**
Tokens / step	all n prompt tokens	1 new token
Dominant op.	GEMM (matrix–matrix)	GEMV (matrix–vector)
Bottleneck	compute (tensor cores)	memory bandwidth (HBM)
Scales with	prompt length ($O(n^2)$ attn)	cache length & batch size
Governs	TTFT	TPOT / TBT
Batching benefit	moderate	very high

TTFT = Time-to-First-Token **TPOT/TBT** = Time-Per-Output-Token / Time-Between-Tokens

GPU Utilization Metrics

Arithmetic Intensity

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

Arithmetic intensity measures how much computation is performed for each byte moved from memory.

Ridge Point

$$AI_{\text{ridge}} = \frac{P_{\max}}{B_{\max}}$$

The ridge point is where the limiting factor changes from memory bandwidth to computation.

Performance Bound

$$P_{\text{attainable}} \leq \min\left(P_{\max},\ AI \cdot B_{\max}\right)$$

where

- $P_{\max}$ = peak compute throughput
- $B_{\max}$ = peak memory bandwidth

Two Regions

Memory-bound

$$AI < AI_{\text{ridge}}$$

Performance is primarily limited by **memory bandwidth**.

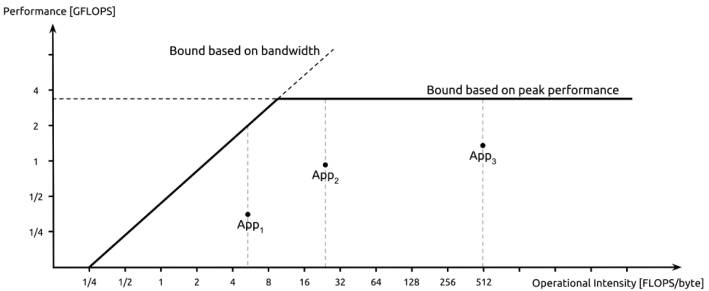
Compute-bound

$$AI > AI_{\text{ridge}}$$

Performance can become limited by **peak compute throughput**.

Key idea: Low arithmetic intensity → memory-bound; high arithmetic intensity → compute-bound.

Roofline Model: Interpreting GPU Workloads



$$\textbf{Arithmetic Intensity} = \frac{\text{FLOPs}}{\text{Bytes transferred}}$$

Roofline Model Graph

Left of the ridge

- Low arithmetic intensity
- **Memory-bound**
- Performance depends mainly on memory bandwidth

Right of the ridge

- High arithmetic intensity
- **Compute-bound**
- Performance depends mainly on peak compute

Why Caching Is Necessary

Without caching, generating token $t{+}1$ re-runs the *entire* forward pass over tokens $1 \dots t$ – cost grows quadratically across an n-token generation.

Because keys and values for past, causally-masked tokens never change, it is correct – and far cheaper – to compute each token's K, V once, store them, and simply append at every step.

KV Cache Memory Footprint

For L transformer layers, n_h KV heads, head dimension d_h, precision p bytes/element, sequence length n:

$$\text{KV bytes} = 2 \times L \times n_h \times d_h \times n \times p$$

- Factor of 2: separate storage for K and V.
- Linear in sequence length – but multiplied by **concurrent** sequences (batch size).

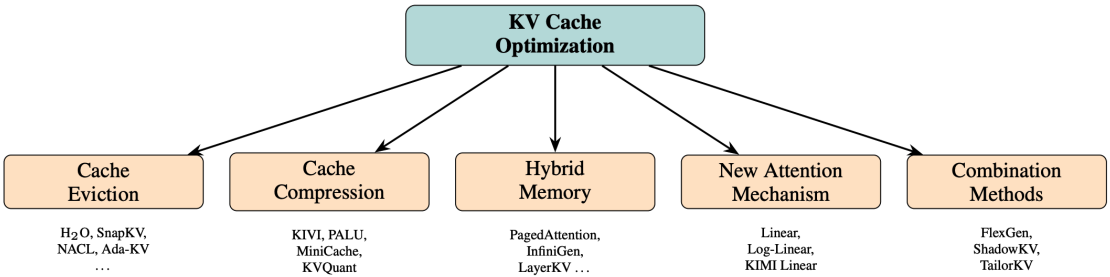
Worked Example

7B-class model, 32 layers, 32 KV heads, d_h=128, FP16 (2 bytes): a *single* sequence at 32K tokens already needs several gigabytes. Production systems serve hundreds concurrently.

Practical Trade-off

Key Observation

As context windows grow from thousands to millions of tokens, **KV cache size – not model weight size – becomes the dominant consumer of GPU memory**.



A Taxonomy of KV Cache Optimizations

1. **Cache eviction** – discard unimportant tokens (attention-score-based, sliding windows).
2. **Cache compression / quantization** – store K/V at reduced precision (INT8/INT4, mixed per-layer).
3. **Hybrid memory** – offload cold blocks to CPU/NVMe, prefetch back.
4. **Novel attention mechanisms** – redesign attention to need a smaller cache: MQA, GQA, MLA.
5. **System-level management** – paging (PagedAttention), disaggregated prefill/decode.

The Fragmentation Problem

Before PagedAttention, engines pre-allocated a **contiguous** block sized for the *maximum* possible sequence length, causing:

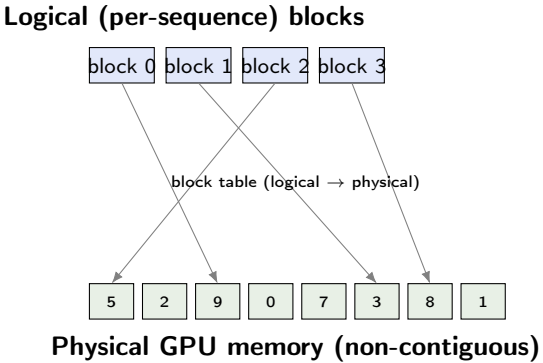
- **Internal fragmentation** – a request generating 50 tokens still reserves space for 2048.
- **No cross-sequence sharing** – even an identical system prompt occupies a separate contiguous span per sequence.

Measured Impact

Naive systems have reported up to 60–80% of allocated KV cache memory wasted.

PagedAttention

Borrows the idea of **virtual memory** from operating systems: KV cache is split into fixed-size **blocks**; a per-sequence **block table** maps logical blocks to (possibly non-contiguous) physical locations.



Benefits of PagedAttention

Key Idea

PagedAttention does not reduce the *total* KV data computed – it reduces *waste* in how memory is allocated, and turns cache sharing into a first-class system capability.

- Internal fragmentation bounded to < 1 block/sequence (often $< 4\%$ waste).
- **Copy-on-write** sharing – like OS process forking – enables efficient parallel sampling, beam search, and prefix sharing.
- Reported gains: 2–4$\times$ higher memory utilization $\Rightarrow$ more concurrent sequences per GPU.

vLLM: The Inference Engine

Suppose we have an open-source model and want to get responses from it using our own GPU. We would need to:

1. Load the model
2. Receive requests
3. Batch requests
4. Schedule requests
5. Manage KV caches
6. Allocate/free GPU memory
7. Run attention
8. Generate tokens
9. Handle multiple sequences
10. Handle GPU memory pressure

What is vLLM?

vLLM is a library/runtime that provides wrappers to handle all of this efficiently and in an optimized manner.

Why Batch at All

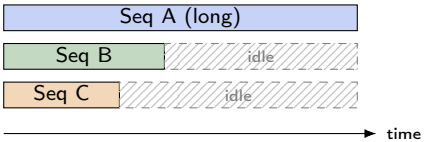
Decoding is memory-bandwidth-bound: every step must stream the *full* weight set from HBM, almost regardless of how many sequences decode simultaneously.

Key Idea

If that bandwidth cost is paid anyway, serving many sequences per forward pass amortizes it across more useful work – raising arithmetic intensity and shifting the operating point rightward on the roofline. Batching is the single largest lever for **throughput**.

Static Batching

A fixed group of requests runs to completion before the next batch starts.

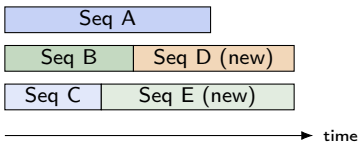


The Straggler Problem

Throughput is capped by the batch's *longest-running* member – shorter sequences finish early and their GPU slots simply idle.

Continuous (Iteration-Level) Batching

ORCA (a distributed serving system for large Transformer-based generative models (like GPT-3)) introduced **iteration-level scheduling**: the scheduler operates at the granularity of a *single forward pass*.



As soon as a sequence emits an end-of-sequence token, it is evicted immediately and a waiting request is admitted at the very next step – no idle slots. Composes naturally with PagedAttention.

Reported gains: 2–4$\times$ over static batching, up to an order of magnitude at high concurrency.

Chunked Prefill and Piggybacking

Problem: injecting a full long-prompt prefill into a running batch can stall every decode-phase sequence, spiking their per-token latency. **Solution (Sarathi-Serve):** split a long prompt's prefill into smaller **chunks**, and interleave ("piggyback") those chunks with ongoing decode steps – bounding the worst-case latency injected into concurrent decode sequences while still making progress on the new prompt.

Batching Strategies at a Glance

Strategy	Mechanism	Addressed / Left
Static batching	Fixed group runs to completion	Idles on stragglers
Continuous batching	Evict/admit at iteration granularity	Fixes idling; needs paged memory
Chunked prefill	Splits long prefills, interleaves with decode	Bounds latency spikes

MQA and GQA

Multi-Query Attention (MQA)

- All query heads share *one* K head and *one* V head
- Shrinks cache by $\sim n_h\times$
- Can noticeably hurt quality vs. MHA

Grouped-Query Attention (GQA)

- Query heads split into n_g groups, sharing K/V per group
- Tunable memory/quality trade-off
- **Uptraining** converts an MHA checkpoint to GQA cheaply
- Used by Llama-3, Mistral, Qwen2.5, etc.

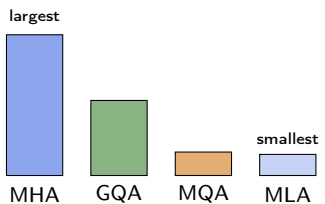
Multi-Head Latent Attention (MLA)

Introduced with DeepSeek-V2/V3: instead of sharing K/V *heads*, MLA jointly compresses each token's key and value into a single low-rank **latent vector** $c_t^{KV} \in \mathbb{R}^r$, $r \ll n_h d_h$, and reconstructs per-head K/V via learned up-projections on the fly.

- Only the compressed latent (+ a small decoupled RoPE component (positional encoder) is cached.
- DeepSeek-V2 ablations: MLA **outperforms** standard MHA in quality with a cache comparable in size to MQA.

KV Cache Size: A Relative Comparison

Mechanism	Cached qty. / token / layer	Quality vs. MHA
MHA	$2\, n_h\, d_h$	baseline
GQA	$2\, n_g\, d_h$ $(n_g < n_h)$	near-MHA, tunable
MQA	$2\, d_h$	noticeably lower
MLA (DeepSeek-V2/V3)	$\approx r + d_{\text{rope}}$ (low-rank)	matches or exceeds MHA



Sparse and Windowed Attention

Rather than compressing what is stored per token, restrict *which* past tokens each query attends to.

- **Sliding-window attention**: caps attention to the most recent w tokens – bounds compute and memory regardless of total sequence length. Often paired with periodic full-attention layers to preserve global context.
- **Learned block-sparse attention** (e.g. block-sparse FlashAttention): learns which K/V blocks matter for a given query block, skipping compute and IO for negligible blocks.

Putting It All Together

The Two Questions

1. **Move fewer bytes:** KV cache management, quantization, prefix caching, IO-aware attention kernels (FlashAttention), MQA/GQA/MLA.
2. **Keep the accelerator busy:** continuous batching, chunked prefill, PagedAttention-enabled sharing, speculative decoding, parallelism.

Bottom line: prefill is compute-bound and favors raw kernel efficiency; decode is memory-bandwidth-bound and favors batching & compression. Every serving system in this space is, at its core, an exercise in managing that asymmetry.

Thank You

Questions?

Upcoming topics to be discussed

1. Quantization for Inference
2. Speculative Decoding
3. Parallelism Strategies for Distributed Inference
4. Mixture of Experts (MoE) Inference
5. Prefill and Decoding Step Separation
6. Prefix Caching and Structured Generation
7. Long Context Inference
8. Serving Frameworks: `llama.cpp`, vLLM
9. Decoding and Sampling Strategies
10. Model Compression Beyond Quantization
11. Benchmarking and Hardware Considerations

References & Further Reading

Notes by me:

- Byte Pair Encoding – https://drive.google.com/file/d/1HlIp-YaC2bkHo6R5ZZjzLXUCtzCFqWga/view?usp=sharing
- GPU Utilization Trends – https://drive.google.com/file/d/1yJBd-1baO932s4VW52PyyS-8vaDCvbvk/view?usp=sharing
- PagedAttention and vLLMs – https://drive.google.com/file/d/1W6mxETAV83S-Kn5TAE4wRAa4DZ5x6QQ5/view?usp=sharing

Papers & articles:

- Byte Pair Encoding (Medium) – https://medium.com/@hsinhungw/understanding-byte-pair-encoding-fd196ebfe93f
- Attention Is All You Need – https://arxiv.org/abs/1706.03762
- Multi-Query Attention (MQA) – https://arxiv.org/abs/1911.02150
- GQA: Grouped-Query Attention – https://arxiv.org/pdf/2305.13245
- DeepSeek-V3 / Multi-Head Latent Attention (Medium) – https://medium.com/data-science/deepseek-v3-explained-1-multi-head-latent-attention-ed6bee2a67c4
- KV Cache – https://arxiv.org/pdf/2603.20397