Modern AI Systems: Agents and System Optimizations

LLM Serving Basics I: Introduction to LLM

Juncheng Yang



Harvard John A. Paulson
School of Engineering
and Applied Sciences



HARVARD UNIVERSITY
MaDSys
Measurement and Design of Systems Lab

Logistics

- Assignment 3 due on Tuesday 11:59pm
  - NO LATE DAY for leaderboard submission
  - Check website for what needs to be submitted

- Project poster presentation option 2: **1-3pm Dec 2**
  - only sign up if you cannot attend on Dec 7
  - signup due last Sunday, send us an email ASAP if you want to present on Dec 2

Plan for today

- Introduction to LLM
  - lifecycle of an LLM request
  - transformer architecture[1]
  - dense, MoE and advanced variants

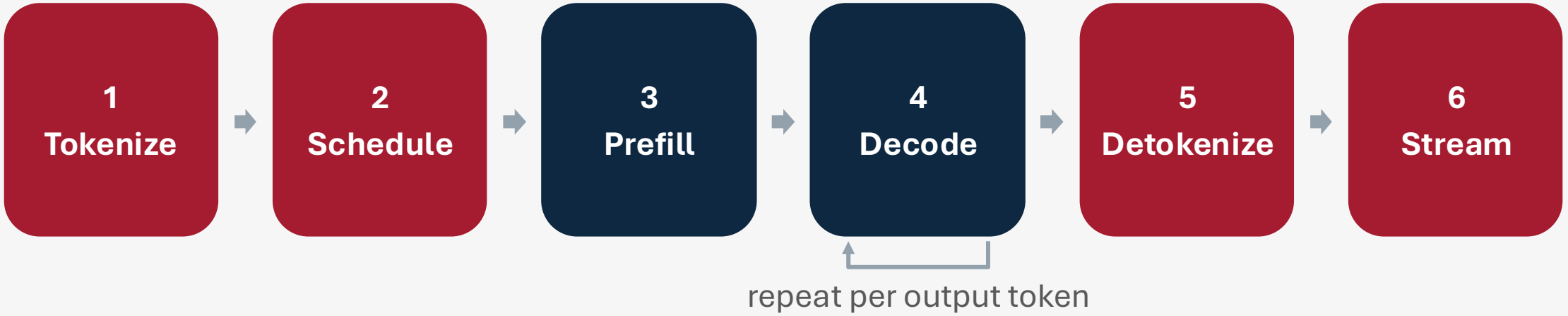
- Introduction to GPU for LLM serving[2]
  - GPU basics
  - GPU architecture
  - GPU programming and recent kernel advances

[1] This lecture assumes traditional self-attention transformer architecture

[2] We will not go into details; the goal is to help you build intuition

Lifecycle of an LLM request

Six stages from prompt to streamed text.

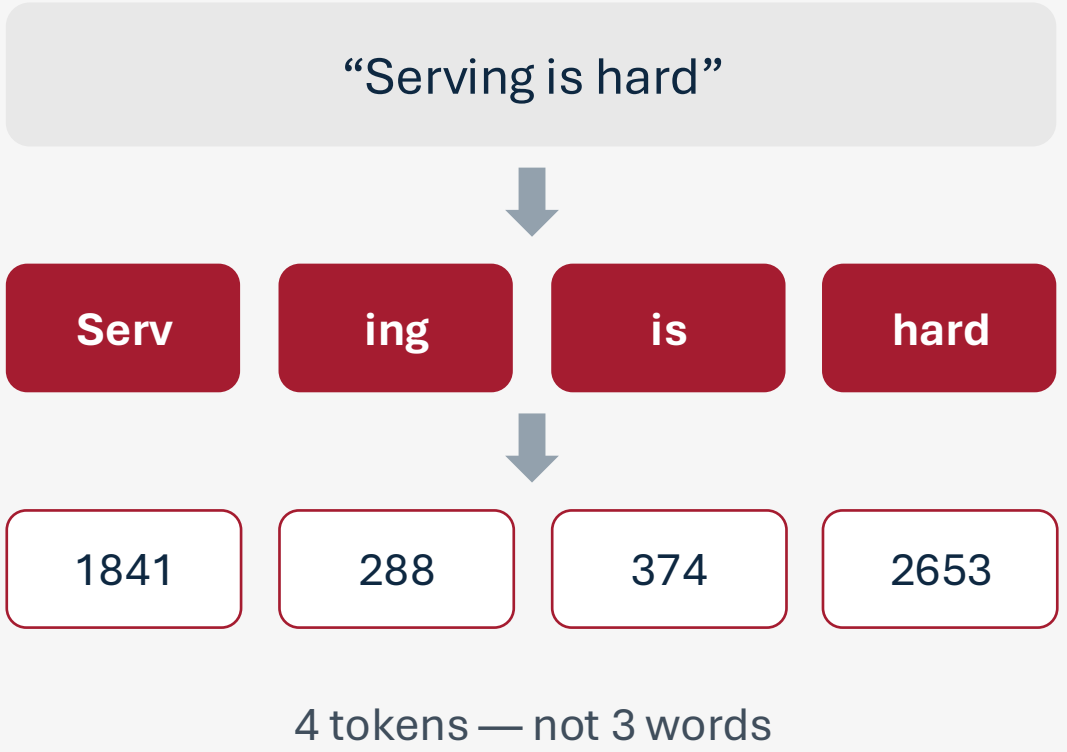


Prefill → time to first token
whole prompt in one forward pass

Decode → tokens per second
one forward pass per output token

Stage 1: Tokenization

- Text is split into tokens, not words
- About 3–4 characters per token
- Token count drives cost and memory
- Tokenizers differ across models



Byte-level BPE: start from the 256 byte values, then repeatedly merge the most frequent adjacent pair into a new token. No unknown tokens.

Stage 2: Scheduling and batching

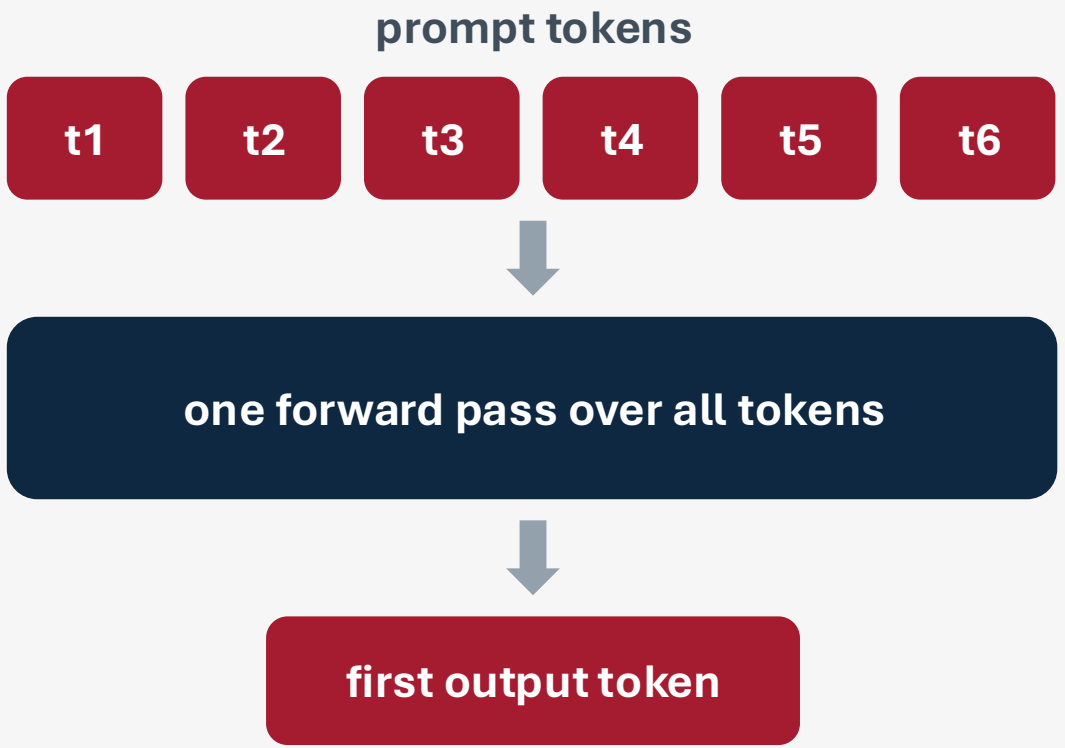
- Requests arrive independently; GPUs want batches
- The scheduler picks who runs each step
- Continuous batching: join or leave mid-flight
- Admission is capped by KV cache memory



Finished requests leave, new ones join — no waiting for a full batch

Stage 3: Prefill

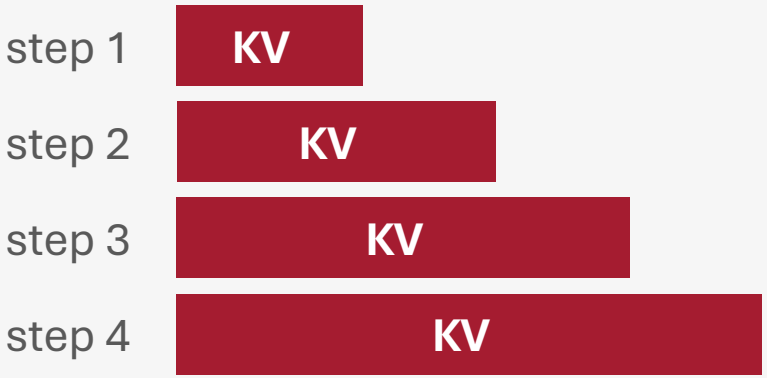
- Whole prompt runs in one pass
- All prompt tokens attend in parallel
- Compute-bound; cost grows with length
- Generate the KV cache, emits token 1



Compute-bound → time to first token

Stage 4: Decode generates logits; sampling picks the next token

STAGE 4 · DECODE



One forward pass per new token
Cache grows by one token per step
Bandwidth-bound, so batch it
Cache full → preempt or queue

STAGE 5 · SAMPLING

Temperature `temperature = 0`
0: always the top token, >1: more tokens are possible

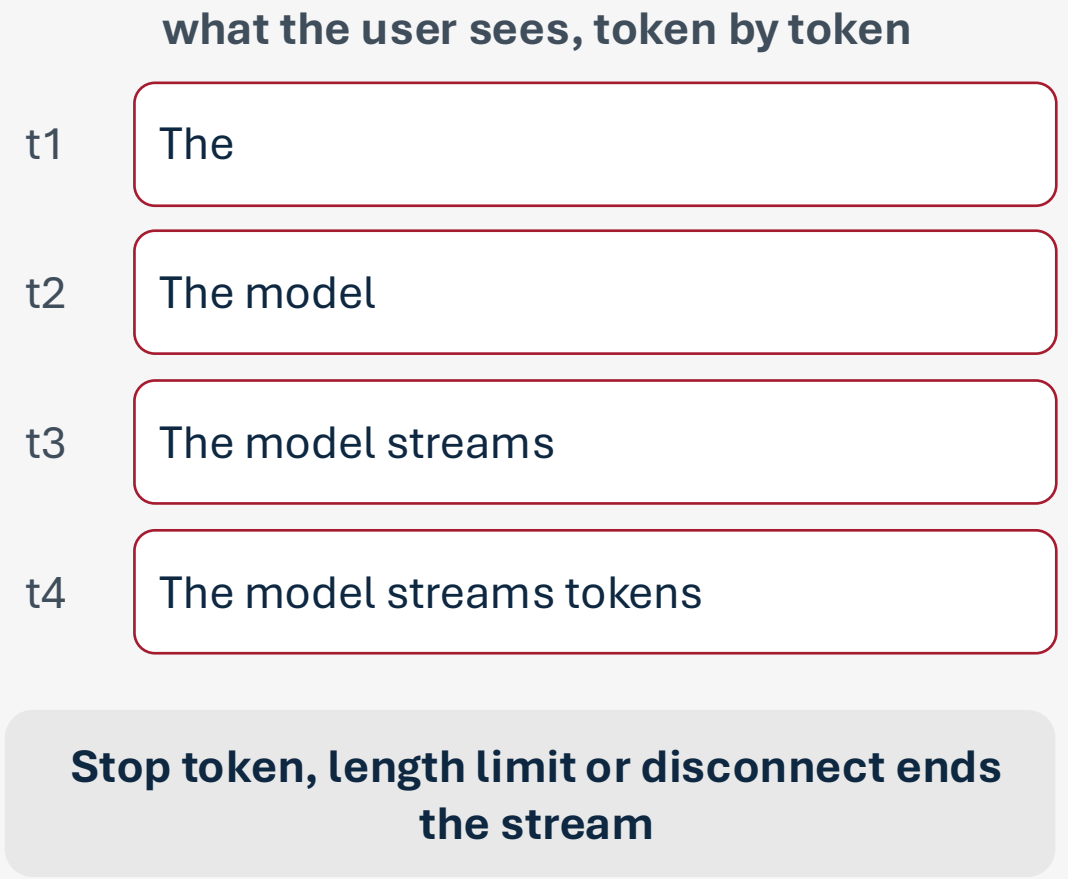
Top-k `top_k = 50`
Sample from the k most likely; k is fixed

Top-p (nucleus) `top_p = 0.9`
Fewest tokens summing to p; adapts to confidence

Same model, same prompt: these settings change the answer and how long it runs

Stage 5: Detokenize and stream

- Token IDs become text as produced
- Partial characters are buffered
- Streaming makes it interactive
- Stopping early frees cache memory



Lifecycle: what to take away

1 **Tokenize:** turns text into token IDs; a bigger vocabulary means fewer tokens but a larger embedding table.

2 **Schedule:** continuous batching lets requests join and leave every step; KV cache memory caps admission.

3 **Prefill:** the whole prompt runs in one compute-bound pass, generates the KV cache and the first token.

4 **Decode and sample:** one memory-bound forward pass per token; sampling picks each token from the logits.

5 **Detokenize and stream:** turn each token ID into text and send it as it is produced.

Peeking into the model

Notation

MODEL SHAPE

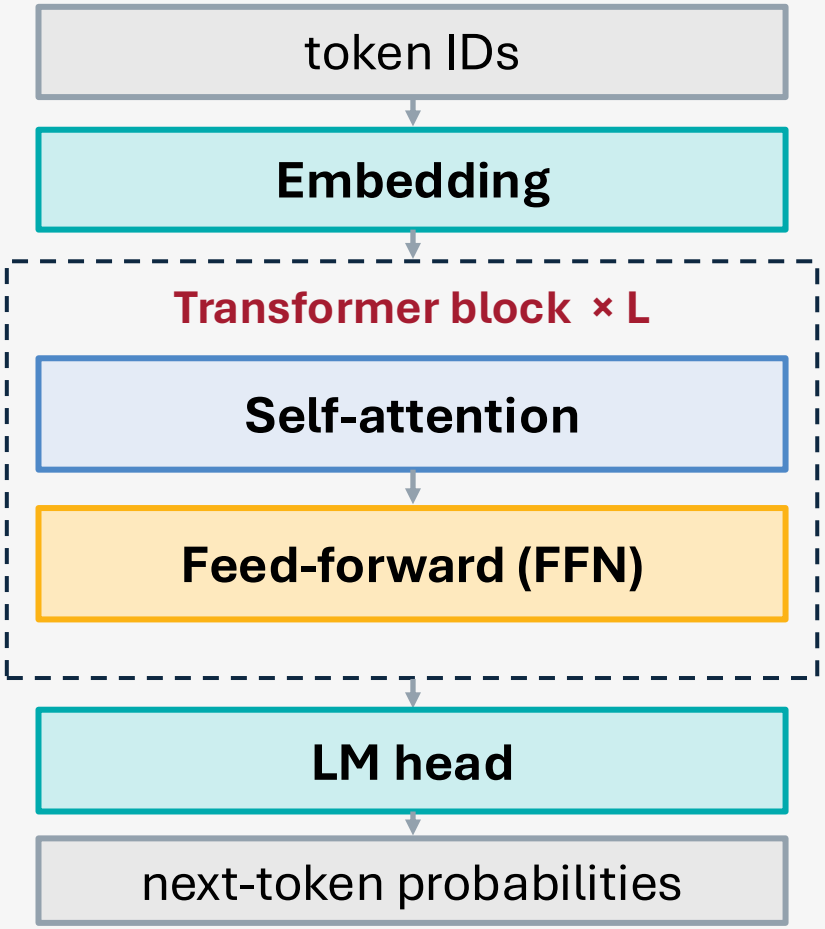
Symbol	Meaning	Llama 3 70B
d	model width	8,192
L	layers	80
H	query heads	64
H_{kv}	key/value heads	8
d_k	head dimension, d ÷ H	128
d_{ff}	MLP hidden width	28,672
V	vocabulary size	128,256

WORKLOAD AND MEMORY

Symbol	Meaning	Llama 3 70B
n	tokens in the sequence	e.g. 20,000
B	concurrent requests	e.g. 16
N	parameters	70.6B
W	weight bytes (BF16)	141 GB
KV	KV bytes per token	320 KB

KV = $2 \times L \times H_{kv} \times d_k \times 2$ B
K^T is K transposed

An LLM is a stack of identical transformer blocks



Input: text arrives as integer token IDs

Embedding: each ID becomes a vector (8,192-dim)

L = 80 blocks in Llama 3 70B, each with norm + residual

Attention: mixes information across tokens

FFN: transforms each token on its own

LM head: one score per vocabulary token

Softmax: scores become probabilities

Each block adds attention and an MLP to a residual stream, each after RMSNorm

$$h = x + \mathrm{Attn}\big(\mathrm{RMSNorm}(x)\big)$$

$$y = h + \mathrm{FFN}\big(\mathrm{RMSNorm}(h)\big)$$

$$\mathrm{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_i {x_i}^2 + \varepsilon}} \odot g$$

RMSNorm
rescales by the root-mean-square; one learned gain g per dimension

Pre-norm
normalize each sublayer's input, so deep stacks train stably

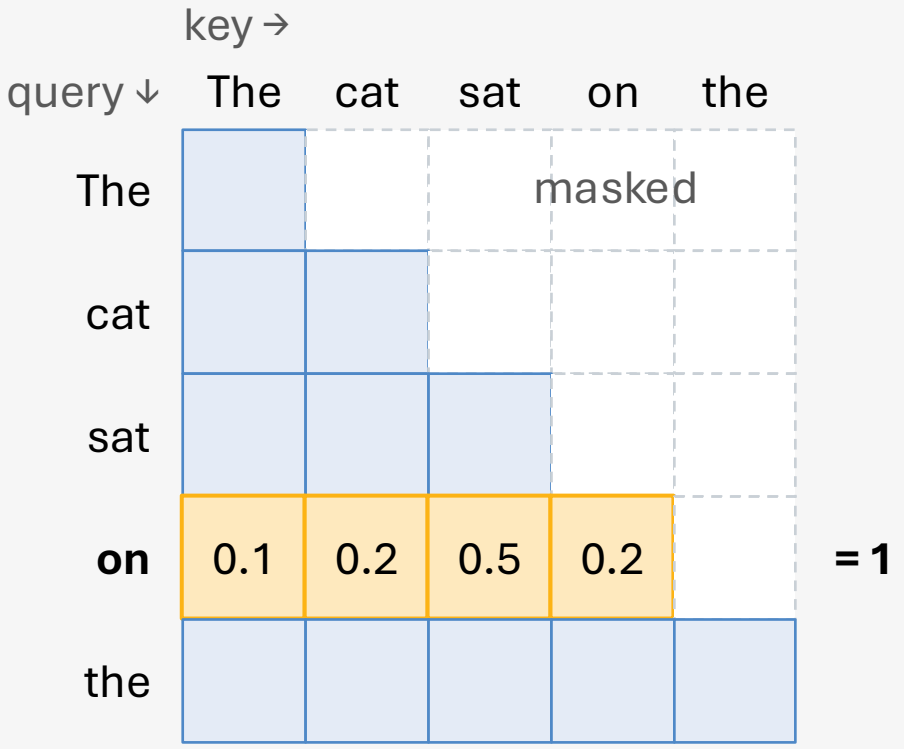
Residual stream
x flows through all 80 blocks; each sublayer only adds an update

Zhang & Sennrich, RMSNorm, NeurIPS 2019; Xiong et al., ICML 2020; Meta, Llama 3, 2024

Self-attention lets each token read every earlier token

- Each token emits a **query**, a **key** and a **value**
- Query · key scores decide **how much to read** from each token
- A **causal mask** hides future tokens
- n tokens → **n^2 scores**: cost grows with context

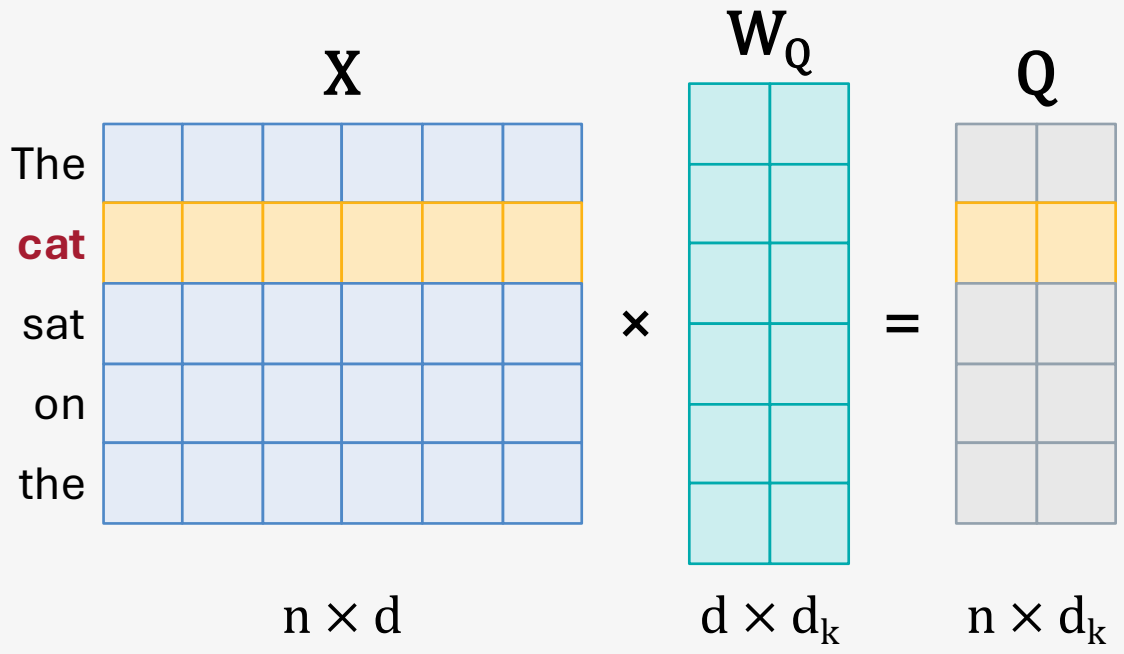
$$\mathrm{softmax}\left(QK^{\mathrm{T}}/\sqrt{d_k}\right)\cdot V$$



Why divide by $\sqrt{d_k}$?

q · k sums d_k products, so it grows with d_k; scaling keeps softmax from saturating

Each token is a row of X; weight matrices turn it into Q, K and V



$Q = X W_Q$ what each token looks for
$K = X W_K$ what each token offers
$V = X W_V$ what each token passes on

Variables

n
tokens so far (prompt + output)

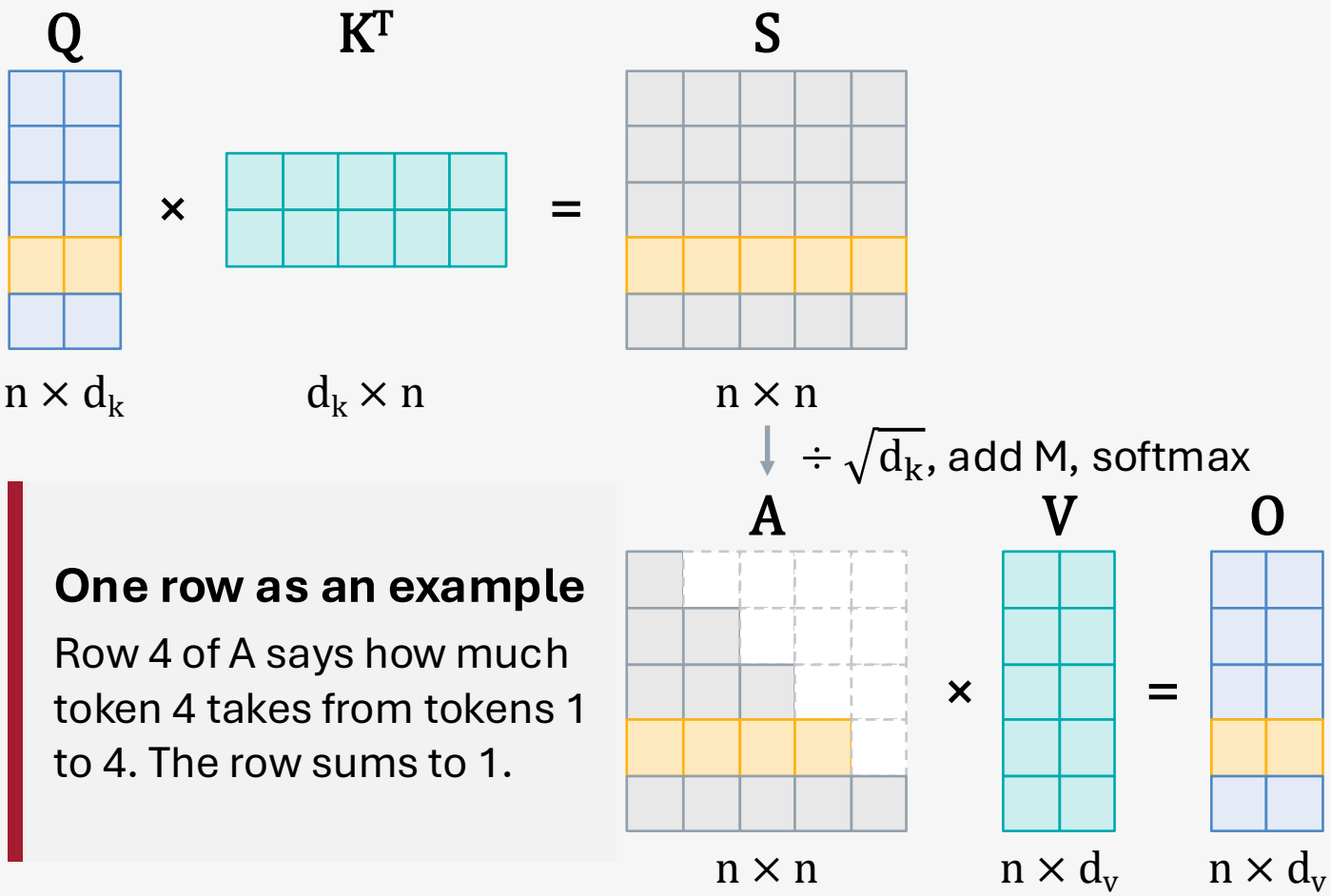
d
numbers per token vector (8,192)

d_k
numbers per query or key, one head (128)

W_Q, W_K, W_V
learned weights, d × d_k each, per head

head
one attention unit; 64 run in parallel

Q K^T scores every pair of tokens; softmax turns each row into weights



One row as an example

Row 4 of A says how much token 4 takes from tokens 1 to 4. The row sums to 1.

Variables

K^T keys turned so each column is one token

S scores: row i, column j = query i · key j

M mask: −∞ above the diagonal hides future tokens

A weights: each row sums to 1

d_v value vector dimension (128)

O output: each row is a weighted mix of V rows

Attention has an $O(n^2)$ computation complexity

Step	Formula	Shape
Input	X	$n \times d$
Queries	$Q = X\,W_Q$	$n \times d_k$
Scores	$S = QK^T/\sqrt{d_k}$	$n \times n$
Weights	$A = \mathrm{softmax}(S + M)$	$n \times n$
Output	$O = A\,V$	$n \times d_v$

FLOPs per layer

$Q\,K^T$: $2\,n^2\,d$ $A\,V$: $2\,n^2\,d$

total $\approx 4\,n^2\,d$

Memory if S were stored

$n^2 \times \text{heads} \times \text{layers} \times 2\ \text{B}$

$= 8{,}192^2 \times 64 \times 80 \times 2\ \text{B} \approx 690\ \text{GB}$

FlashAttention computes softmax tile by tile and never stores S

Multi-head attention runs 64 small attentions in parallel and mixes them with W_O

$$\text{head}_i = \text{softmax}\left(\frac{Q_i K_i^{\mathrm{T}}}{\sqrt{d_k}} + M\right) V_i$$

$$\text{MHA}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H) W_O$$

Split
Q, K and V split into H = 64 heads of d_k = 8,192 ÷ 64 = 128

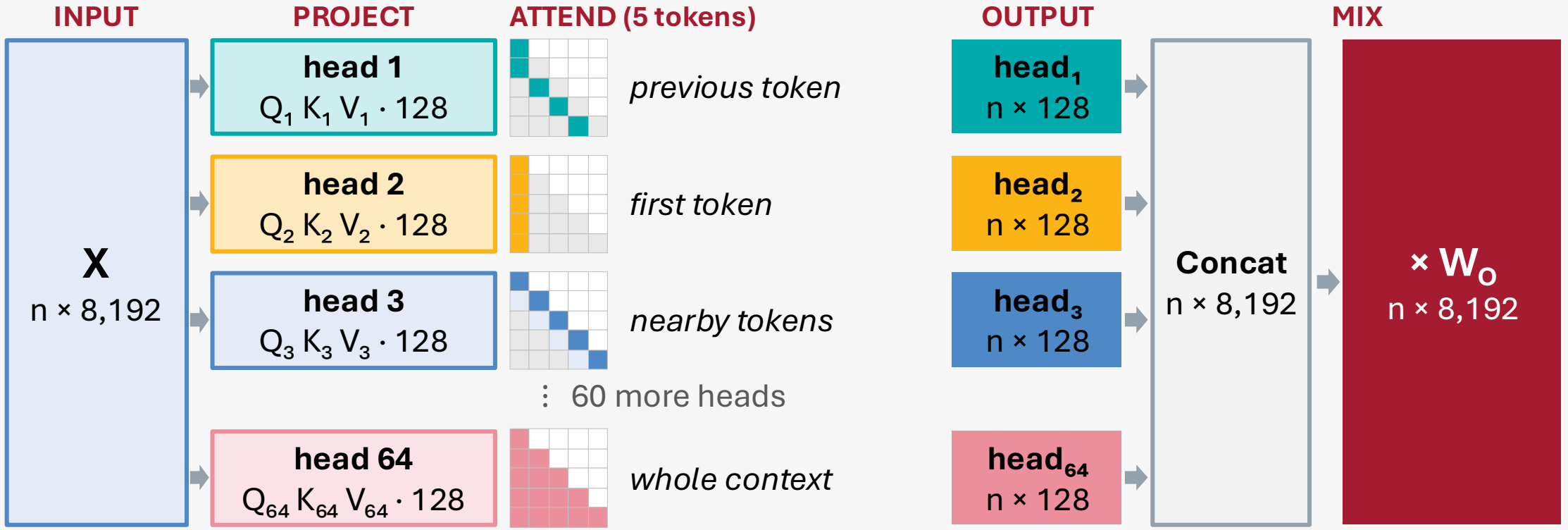
Attend
each head runs masked softmax attention on its own 128 dimensions

Mix
W_O blends the heads into one vector

Same FLOPs as one wide head: 64 × 128 = 8,192, so splitting changes what is learned, not the cost

Vaswani et al., Attention Is All You Need, NeurIPS 2017; Meta, Llama 3, 2024

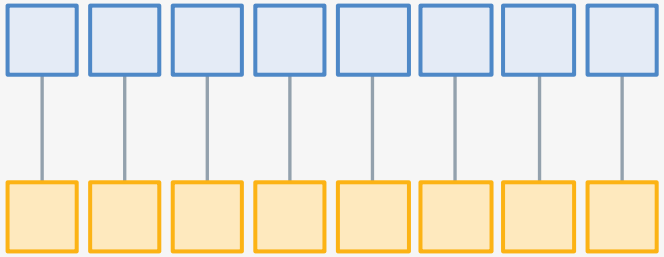
Each head attends with its own 128 dimensions, so heads learn different patterns



Sharing key/value heads shrinks the KV cache

Multi-head (MHA)

8 query heads



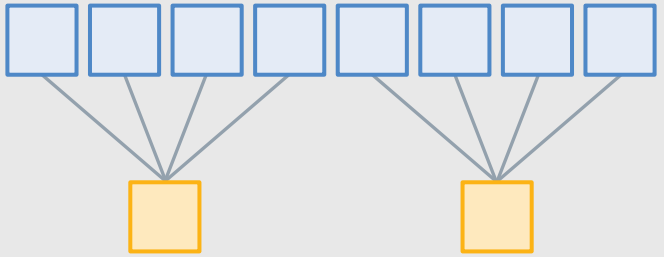
8 KV heads

1×

KV cache size

Grouped-query (GQA)

8 query heads



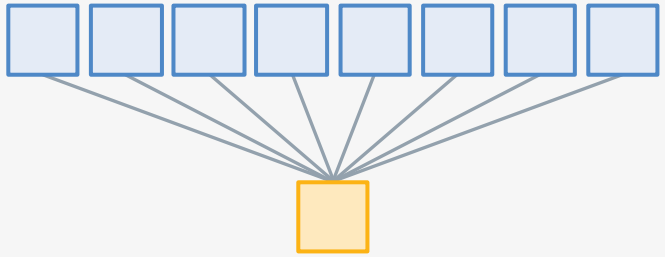
2 KV heads

¼×

KV cache size

Multi-query (MQA)

8 query heads



1 KV head

⅛×

KV cache size

KV cache per token = $2 \times L \times H_{kv} \times d_k \times$ bytes per value

Llama 3 70B: 64 query heads share 8 KV heads, so its KV cache is 8× smaller than MHA

$= 2 \times 80 \times 8 \times 128 \times 2$ B = **320 KB per token**, vs 2.5 MB with MHA

Ainslie et al., GQA, EMNLP 2023; Shazeer, MQA, 2019

Llama 3 70B as an example

$\mathbf{d = 8{,}192}$
width of a token vector

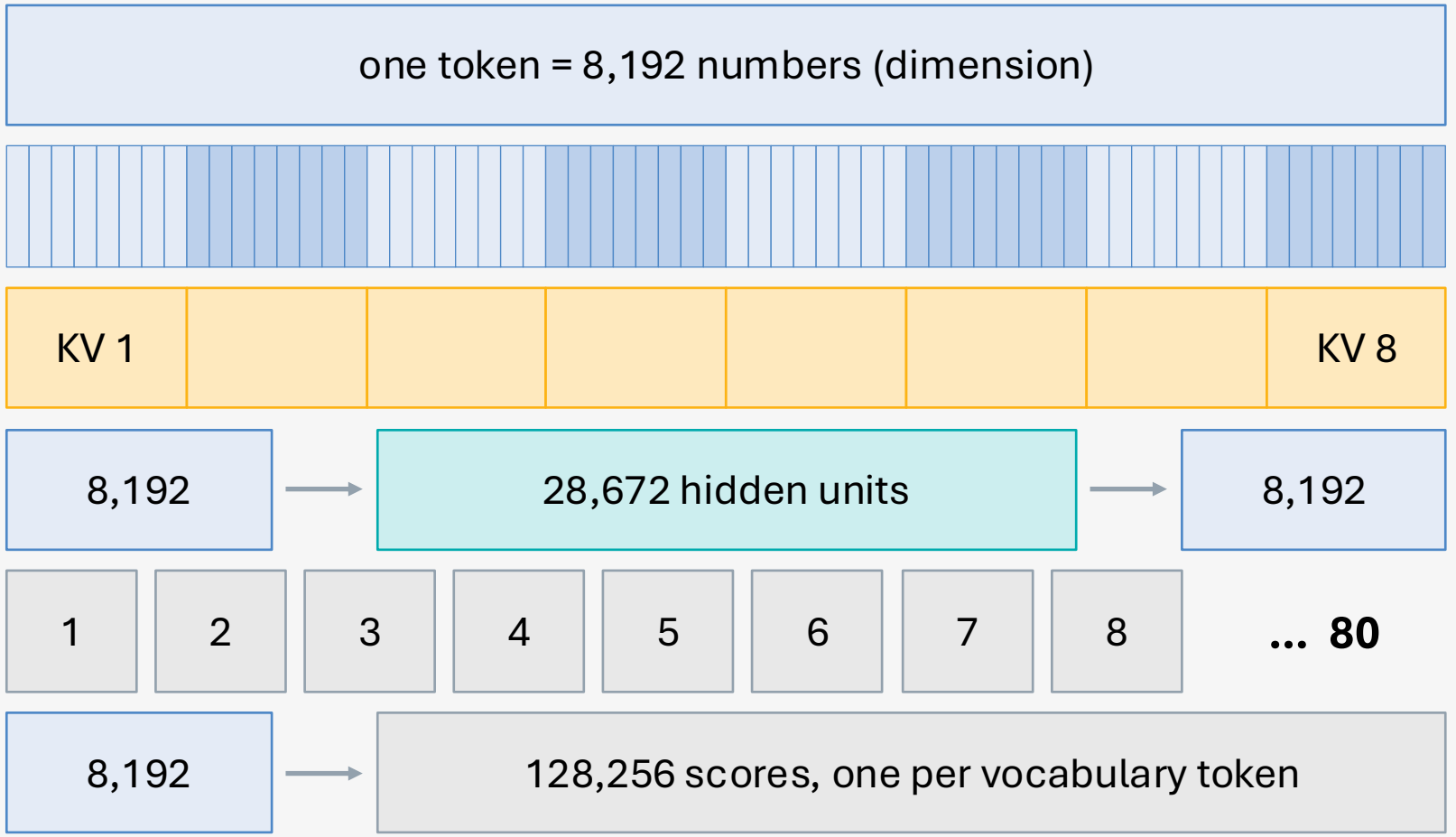
$\mathbf{h = 64,\ d_h = 128}$
64 Q heads × 128 = 8,192

$\mathbf{h_{kv} = 8}$
share 8 KV heads

$\mathbf{d_{ff} = 28{,}672}$
FFN hidden width, 3.5 × d

$\mathbf{L = 80}$
transformer blocks in a row

$\mathbf{V = 128{,}256}$
tokens in the vocabulary



Llama 3 70B: where the parameters sit

Component	Formula	Params
Attention / layer	$2d \cdot H \cdot d_k + 2d \cdot H_{kv} \cdot d_k$	151M
FFN / layer	$3 \cdot d \cdot d_{ff}$	705M
All layers	$80 \times 856M$	68.5B
Embed + LM head	$2 \cdot V \cdot d$	2.1B
Total		**70.6B**

Systems intuition: weights, compute and KV cache set prefill, decode and memory

Weights
`2 B × params`
141 GB

Compute
`≈ 2 × params`
141 GFLOPs/token

KV cache
$2 \times L \times H_{kv} \times d_k \times 2\ B$
320 KB/token

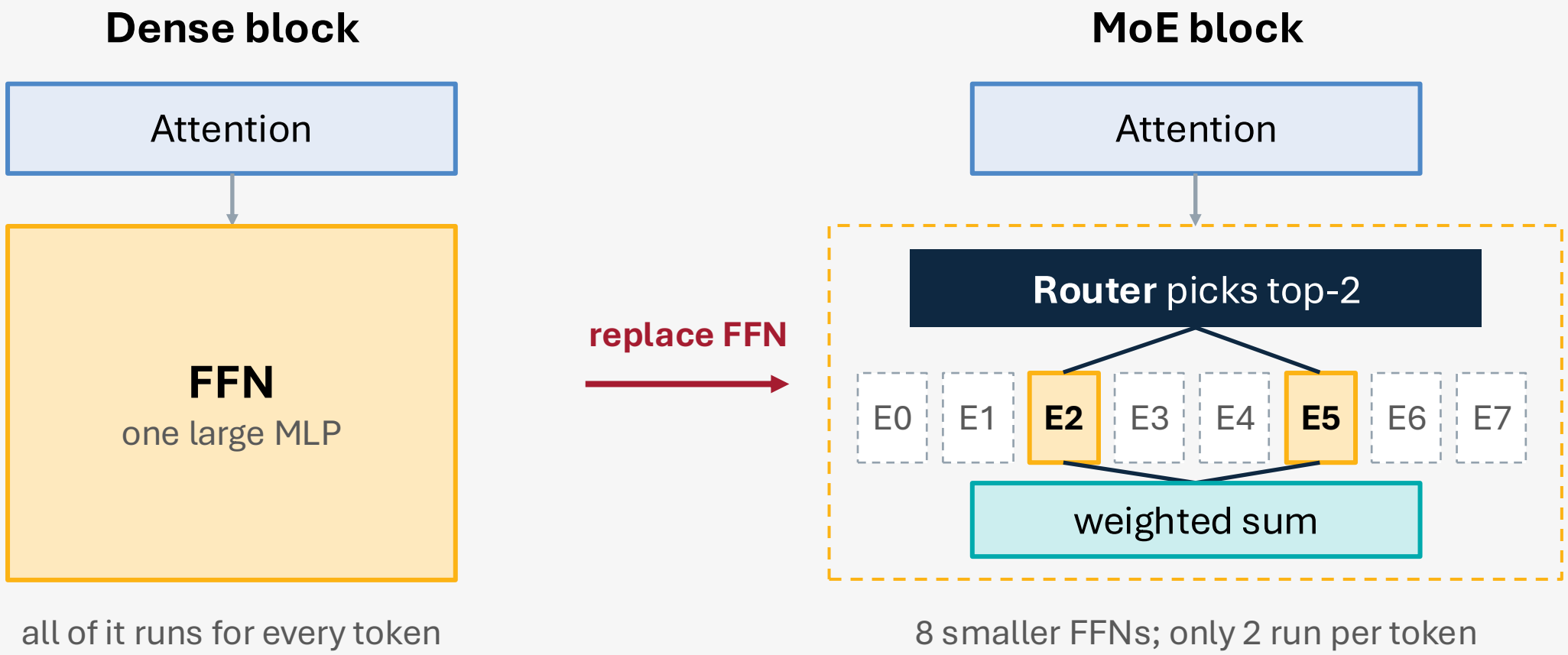
Llama 3 70B, BF16, on 2× H200: 1,978 TFLOP/s, 9.6 TB/s, 282 GB; n = 80,000 tokens, B = requests

Quantity	Formula	1 request	64 requests
Prefill (80K tokens)	$B \cdot (2N \cdot n + 2L \cdot d \cdot n^2) \div \text{FLOP/s}$	**9.95 s**	**637 s**
VRAM needed	$W + B \cdot n \cdot KV$	**167 GB**	**1,779 GB**

How do you support longer sequences?

Meta, The Llama 3 Herd of Models, 2024; NVIDIA, H200 datasheet, 2024

MoE replaces each block's FFN with a router and a set of smaller expert FFNs



Shazeer et al., MoE, ICLR 2017; Jiang et al., Mixtral of Experts, 2024

Dense vs MoE: memory scales with total parameters, compute with active

Model	Experts	Total	Active / token	Weights (BF16)	GFLOPs / token
Llama 2 70B	all (dense)	69B	69B	138 GB	138
Mixtral 8x7B	2 of 8	46.7B	12.9B	93 GB	26
Llama 3.1 405B	all (dense)	405B	405B	810 GB	810
DeepSeek-V3	9 of 257	671B	37B	1,342 GB	74

Same quality, less compute: Mixtral matches Llama 2 70B at ~1/5 the FLOPs; DeepSeek-V3 rivals Llama 3.1 405B at ~1/11 the FLOPs but needs 1.7× the memory.

GFLOPs ≈ 2 × active params. Jiang et al. 2024; DeepSeek-AI 2024; Touvron et al. 2023; Llama Team 2024

Advanced LLMs

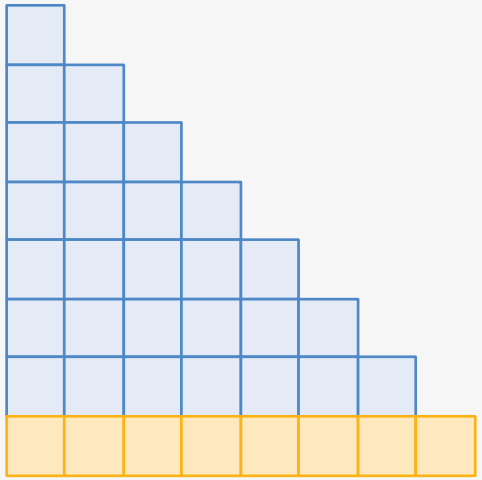
Sliding window
Multi-Latent attention
Linear attention
State-space models

Sparse attention
Looped transformers
Multi-token prediction
Diffusion LMs

ADVANCED — skim, or read more on your own

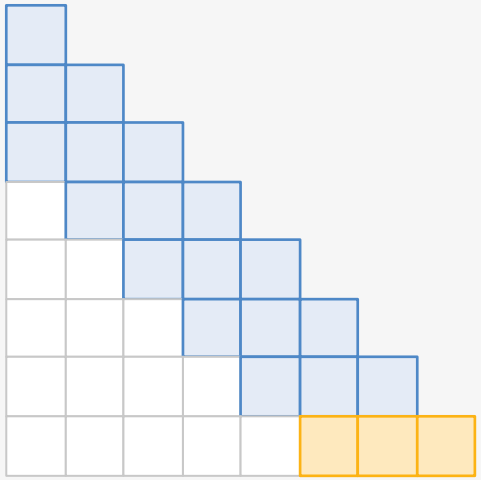
Sliding-window layers cap the KV cache; a few global layers keep long-range recall

Full causal



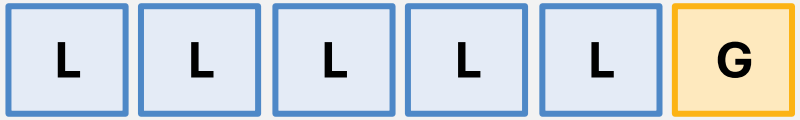
last token reads all
KV grows with n

Sliding window, w = 3



last token reads 3
KV capped at w tokens

Gemma 3 layer pattern



5 local (w = 1,024) to 1 global, repeated

Mistral 7B
every layer local, w = 4,096

Gemma 2
alternates 1 local (w = 4,096) : 1 global

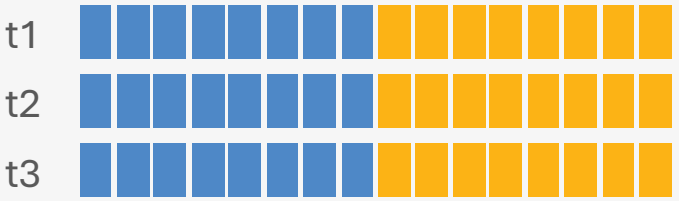
Serving effect: at 128K context, Gemma 3 stores about 6× less KV than all-global layers

MLA stores one compressed latent per token

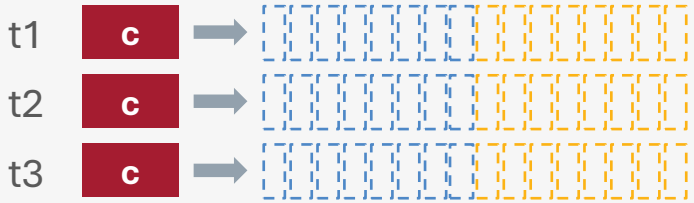
token h → compress → **latent c · cached** → expand to K, V

Store the summary, rebuild the details: K and V are recomputed from c when attention needs them

MHA caches K and V for every head



MLA caches one latent c



blue = K, yellow = V, 8 of 128 heads shown; dashed = rebuilt from c, never stored

Extra math, less memory
the up-projection is folded into the query and output weights at decode time

Serving effect: vs. DeepSeek 67B, DeepSeek-V2 cuts KV cache 93% and lifts generation throughput 5.8×

DeepSeek-AI, DeepSeek-V2, 2024; DeepSeek-AI, DeepSeek-V3, 2024

ADVANCED — skim, or read more on your own

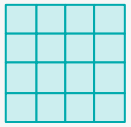
Linear-attention layers keep a fixed-size state instead of a growing KV cache

Softmax attention



KV cache: one more entry per token, forever

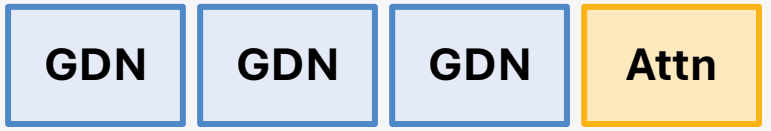
Linear attention



state S_t: $d_k \times d_v$

same size at every step, updated in place

Qwen3-Next: 48 layers



36 Gated DeltaNet + 12 attention; only the 12 keep a KV cache

Plain linear attention

each token adds its key × value to the state; a query reads the state

Gated DeltaNet

first fade the old state, then overwrite what the new key points to

α_t decay gate: how much to forget

β_t write gate: how hard to overwrite

Same 3:1 recipe: Kimi Linear, Qwen3.5

Serving effect: 3 of 4 layers hold a constant-size state per request, at any context length

Yang et al., Gated Delta Networks, ICLR 2025; Qwen Team, Qwen3-Next, 2025

State-space models also keep a fixed-size state

	Gated linear attention	State-space (Mamba)
Update	$S \leftarrow \alpha\, S + \beta\, v\, k^{T}$	$h \leftarrow \bar{A}\, h + \bar{B}\, x$
State	a $d_k \times d_v$ matrix S of key–value pairs	a small vector h per channel
Each token	adds its value × key to S, scaled by β	decays h by $\bar{A}$, then adds $\bar{B}$ × token x
Forgetting	gate α: the token sets how much S decays	$\bar{A}$: the token sets how much h decays
Roots	attention without the softmax	signal processing and RNNs

Selective
$\bar{A}$ and $\bar{B}$ depend on each token, so the layer chooses what to keep

Fixed state
each channel keeps a small state h, so decode memory stays flat as n grows

Hardware-aware scan
a parallel scan keeps the state in SRAM; Mamba-2 turns it into matmuls

Serving effect: no growing KV cache; Mamba reports up to 5× a Transformer's generation throughput

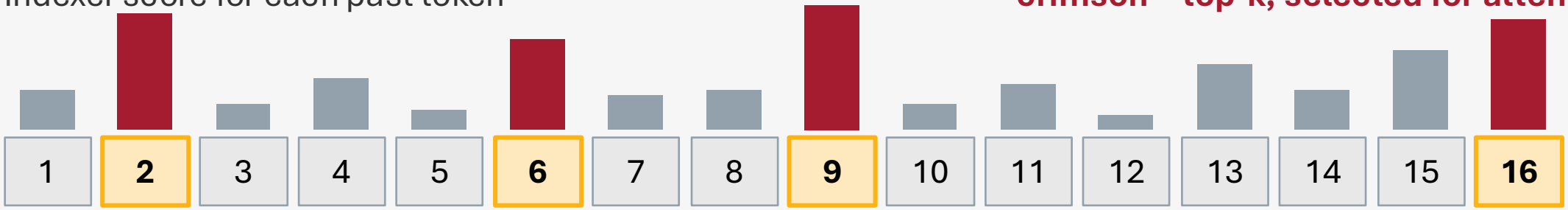
Gu & Dao, Mamba, COLM 2024; Dao & Gu, Mamba-2, ICML 2024

ADVANCED — skim, or read more on your own

Sparse attention reads only the top 2,048 past tokens per query

Indexer score for each past token

crimson = top-k, selected for attention



Shown with $k = 4$ of 16 tokens; DeepSeek-V3.2 keeps k **= 2,048**

1 Score $I_{t,s} = \Sigma_j\, w_{t,j} \cdot \mathrm{ReLU}(q_{t,j} \cdot k_s)$

2 Select $S_t = \text{top-k of } I_{t,s}$

3 Attend $o_t = \mathrm{Attn}(q_t, \{c_s : s \in S_t\})$

Indexer: few heads, FP8, a 128-number key per token

Main attention: 2,048 tokens instead of all n

Serving effect: work per token is capped at 2,048 reads, but all tokens' KV stays stored

Looped transformers reuse one block many times, so depth costs compute, not weights



Same weights every loop: more loops add compute and latency, not parameters

Shared weights
one block R is applied K times, so extra depth adds no parameters

Depth at test time
a 3.5B recurrent-depth model gets better on reasoning tasks as K grows

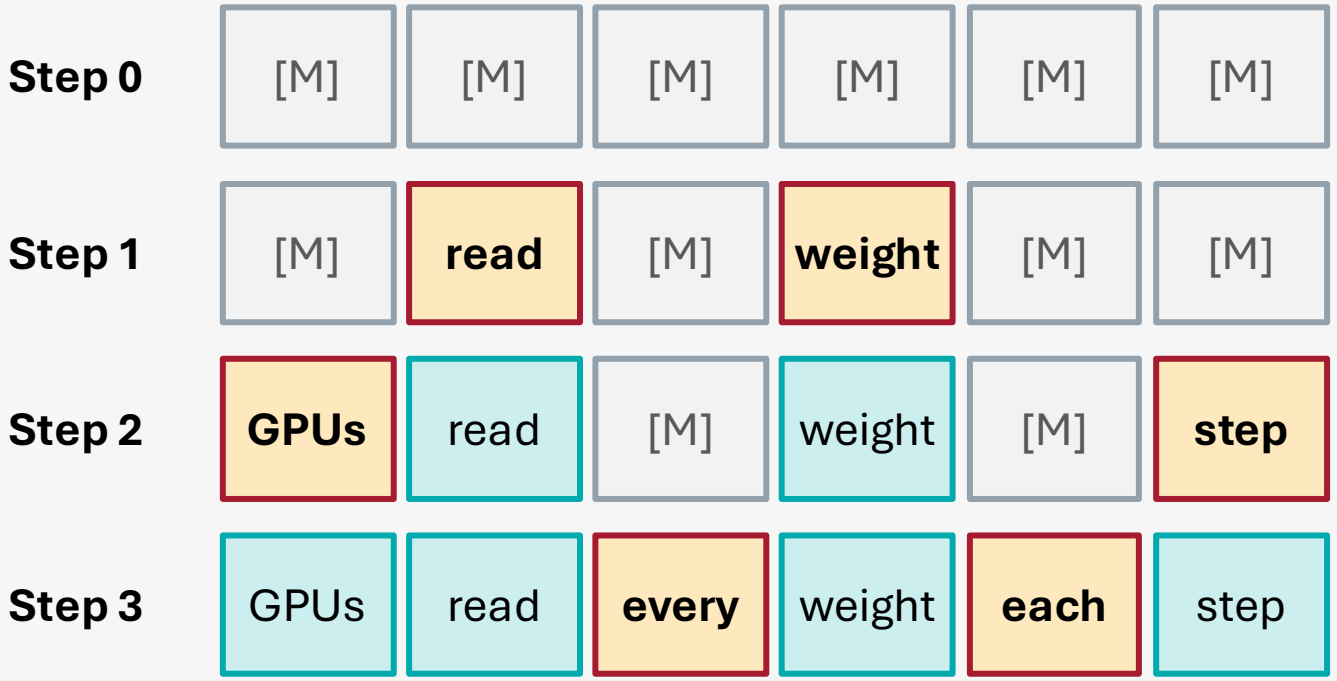
System tradeoffs
weights stay small, but each loop is sequential: latency grows with K

Compute without parameters: K loops cost K× the block's FLOPs and no extra weight memory

Dehghani et al., Universal Transformers, ICLR 2019; Geiping et al., 2025

Diffusion LMs fill in masked tokens over a few parallel refinement steps

crimson outline = filled in at this step



LLaDA (2025)

8B masked-diffusion model trained from scratch on 2.3T tokens; competitive with Llama 3 8B

Dream 7B (2025)

Starts from Qwen2.5 7B weights; the network is still a Transformer

Serving effect: one pass fills many tokens, but fewer steps cost quality

Nie et al., LLaDA, 2025; Ye et al., Dream 7B, 2025

Next class

Optional reading