Scalable AI: Bridging Theory, Understanding, and Practice

Fall 2026

UC Berkeley

Lecture 4: Architectures to Break Bottlenecks

MoE, Latent KV, and Sparse (or Hybrid) Long-Context

Learning goals

By the end of this lecture you should be able to:

1. **Spot the right bottleneck:** decide whether you are compute-limited or bandwidth/capacity-limited.
2. **Do accounting for modern architectural knobs:** estimate how MoE, MLA, and sparse/hybrid designs change FLOPs and/or bytes moved.
3. **Explain the systems tax:** routing + $\textsc{AllToAll}$ for MoE, reconstruction compute for MLA, and cache/batching constraints for sparse attention.
4. **Choose interventions intentionally:** match your constraint (FFN, KV, or L^2) to the architecture that directly changes the dominant term.

Roadmap for today

Part A: The framing (what we mean by "sparsity" here)

- A roofline-style view: runtime is driven by FLOPs and bytes moved
- Four meanings of sparsity
- Notation we will use consistently

Part B: MoE to break the FFN bottleneck

- Dense FFN reminder (why it dominates parameters)
- Conditional computation: total vs activated parameters
- Routing math, load balancing, and why it is a throughput feature
- Expert parallelism and a simple AllToAll volume model

Roadmap for today

Part C: MLA to break the KV bottleneck

- Why decode is dominated by "bytes of KV per generated token"
- Two knobs: head-space sharing (GQA/MQA) vs latent-space caching (MLA)

Part D: Sparse attention & hybrid stacks to break the L^2 wall

- Why FlashAttention is necessary but not sufficient at $L \sim 10^5 - 10^6$
- DSA: fast indexer $\rightarrow$ top-k token selection
- NSA: hierarchical sparse attention (window + select + compress) with gating
- Hybrid stacks: pay for attention only in a few layers (constant-state layers elsewhere)

Important Note

Important Note

- In this lecture, we will cover several architecture knobs and what they are designed to improve.
- This is by no means exhaustive. Please don't worry about memorizing anything – rather, the goal is to provide intuitions for *why* these tweaks were made to the architecture.
- It will give you the tools necessary to analyze and systematically understand future innovations and why certain labs make the decisions they do.
- Keep this in mind as we cover the material.

Part A: Framing and notation

MoE, Latent KV, and Sparse (or Hybrid) Long-Context

A roofline-style framing for architectures

For any workload, the first-order performance picture is:

$$T_{\text{total}} \gtrsim \max\left(\frac{\text{FLOPs}}{C_{\text{peak}}}, \frac{\text{bytes moved}}{\text{BW}_{\text{HBM}}}\right).$$

This lecture is about **architectural interventions** that change the two quantities inside that max:

- **Compute sparsity** reduces FLOPs (do less math).
- **State compression / sharing** reduces bytes (move fewer bytes per token).
- **Long-context mechanisms** reduce the effective L^2 term (avoid "touching everything" at long L).

Our next lecture is about **parallelism interventions** that change the two quantities inside that max.

Main idea

Modern large models are not always just "bigger Transformers." They are **selective** about where they spend compute and bandwidth.

What we mean by “sparsity”

We will use “sparsity” in four distinct meanings:

1. **Parameter sparsity (MoE):** many parameters exist, but only a small subset is activated per token.
2. **State compression / sharing (KV):** store less per token (MLA) or share more across heads (GQA/MQA).
3. **Computation sparsity (DSA/NSA):** for each query token, attend to a structured subset of past tokens/blocks.
4. **Architectural sparsity (hybrids):** not every layer is full attention; most layers are linear-time constant-state sequence layers, with only a few attention layers for global routing.

Why this taxonomy helps

If you know *which* kind of sparsity you are using, you can predict what will change: FLOPs, bytes moved, cache layouts, or communication.

Part B: Mixture of Experts (MoE) to break the FFN bottleneck

MoE, Latent KV, and Sparse (or Hybrid) Long-Context

Notation & conventions (used throughout)

These are notations we use consistently throughout the lecture. There are other section-specific notations that will be introduced in the relevant sections during the lecture.

Symbol	Meaning (units / shape)
T_{total}	Total runtime / step time / end-to-end latency (s)
FLOPs	Floating-point operations required by the workload (ops)
bytes moved	Total bytes transferred to/from high-bandwidth memory (bytes)
C_{peak}	Peak compute throughput of the device or cluster (FLOPs/s)
$\mathrm{BW}_{\mathrm{HBM}}$	Effective bandwidth between compute and HBM (bytes/s)
P	Parameter count (number of learned weights; dimensionless)
b	Bytes per stored element (BF16: 2, FP8: 1, FP32: 4)
$\boldsymbol{x}$, $\boldsymbol{u}_t$, $\boldsymbol{h}_t$	Activations / token representations (vectors); subscript t is token index
W	Weight matrix / linear map (dimensions are stated where used)
$\sigma(\cdot)$	Elementwise nonlinearity (e.g., SiLU); $\odot$ denotes elementwise product
$\mathrm{Top}\text{-}k(\cdot)$	Set of indices of the k largest entries (ties broken arbitrarily)

Notation for Part B.1 (batching, shapes, dense FFN bookkeeping)

Symbol	Meaning (and units / shape)
B	Batch size (number of sequences processed in parallel)
L	Sequence length / context length (tokens per sequence)
$B_{\text{tok}} = B \cdot L$	Token-batch size when flattening a prefill microbatch (common reshape)
D	Model width / hidden size (activation dimension)
n_{layers}	Number of transformer blocks in the stack
N	Number of *query* attention heads
H	Per-head dimension (when applicable); standard attention uses $D = N \cdot H$
F	Dense FFN hidden dimension (often $F \approx 4D$)
F_e	Per-expert FFN hidden dimension (for MoE experts)
$\boldsymbol{x} \in \mathbb{R}^D$	Input to an FFN (one token's hidden state)
$W_1, W_2 \in \mathbb{R}^{F \times D}$	FFN "up" projections (SwiGLU-style gating)
$W_3 \in \mathbb{R}^{D \times F}$	FFN "down" projection
$P_{\text{FFN,dense}}$	Dense FFN parameter count (per layer)
$\text{FLOPs}_{\text{FFN,fwd}}$	Dense FFN forward FLOPs (per layer; dominant matmul terms)

Dense gated FFN reminder (SwiGLU-style)

A standard gated FFN maps $\boldsymbol{x} \in \mathbb{R}^D$ to:

$$\mathrm{FFN}(\boldsymbol{x}) = W_3\big(\sigma(W_1\boldsymbol{x}) \odot (W_2\boldsymbol{x})\big),$$

with

$$W_1, W_2 \in \mathbb{R}^{F\times D}, \qquad W_3 \in \mathbb{R}^{D\times F}.$$

Parameters and compute (dominant terms)

Parameter count (one FFN):

$$\boxed{P_{\text{FFN,dense}} \approx 3DF.}$$

Forward FLOPs (three big matmuls, using $B_{\text{tok}} = B \cdot L$):

$$\boxed{\mathrm{FLOPs}_{\text{FFN,fwd}} \approx 6B_{\text{tok}}DF.}$$

If $F \approx 4D$, then $P_{\text{FFN,dense}} \approx 12D^2$ and $\mathrm{FLOPs}_{\text{FFN,fwd}} \approx 24B_{\text{tok}}D^2$.

Dense FFNs are efficient GEMMs — but they couple **capacity** to **per-token cost**.

Notation for Part B.2 (MoE routing, load balancing, and communication)

Symbol	Meaning
MoE structure and accounting	
N_s	Number of shared experts (always active)
N_r	Number of routed experts (conditional; indexed by $i \in \{1, \dots, N_r\}$)
K_r	Routed experts activated per token (top-K_r routing)
S_t	Set of routed experts selected for token t; $\lvert S_t \rvert = K_r$
$P_{\text{FFN,total}}$	Total FFN parameters per MoE layer (all experts)
$P_{\text{FFN,active/token}}$	Activated FFN parameters per token (shared + selected routed experts)
Router outputs and gating	
$\boldsymbol{u}_t \in \mathbb{R}^D$	Token representation entering the MoE FFN in a transformer block
$\boldsymbol{s}_t \in \mathbb{R}^{N_r}$	Router scores/logits over routed experts for token t
W_{router}	Router weight matrix (typically $W_{\text{router}} \in \mathbb{R}^{N_r \times D}$)
$p_{i,t}$	Normalized routing probability/affinity for expert i on token t
$\boldsymbol{e}_i \in \mathbb{R}^D$	Learned expert embedding (for affinity-style routers)
$g_{i,t}$	Combine weight for expert i on token t (nonzero only if $i \in S_t$)
Load balancing + systems	
n_i	# tokens dispatched to expert i in one step/microbatch
b_i	Expert-specific routing bias used for feedback control
γ	Bias-controller step size
AllToAll	Collective communication used to dispatch/collect expert activations under expert parallelism
ℓ	**LatentMoE:** routed latent dimension ($\ell < D$) for smaller expert payloads

MoE: decouple total capacity from activated compute

An MoE FFN replaces the single dense FFN with many expert FFNs:

- N_r **routed** experts (conditional; only top-K_r active per token)
- N_s **shared** experts (always active; optional but common)

Assume each expert has intermediate size F_e (same shape as the dense FFN, but smaller). Then per layer:

$$\boxed{P_{\text{FFN,total}} \approx (N_s + N_r) \cdot 3DF_e,} \qquad \boxed{P_{\text{FFN,active/token}} \approx (N_s + K_r) \cdot 3DF_e.}$$

The point

MoE introduces a third axis in the scaling trade space:

$$\text{quality} \approx f(\text{active compute},\ \text{tokens},\ \text{total capacity}).$$

MoE forward pass: what actually happens

A convenient decomposition of one MoE layer:

1. **Router scores experts:** compute scores $s_{i,t}$ for expert i and token t.
2. **Top-K_r selection:** select a small subset $S_t \subseteq \{1, \ldots, N_r\}$ with $|S_t| = K_r$.
3. **Dispatch:** send each token activation to the ranks that host its selected experts.
4. **Expert compute:** each expert runs a standard gated FFN on its assigned token batch.
5. **Combine:** weighted sum of expert outputs (plus any shared expert outputs).

A typical output formula (with shared + routed experts)

For token representation $\boldsymbol{u}_t \in \mathbb{R}^D$:

$$\boldsymbol{h}'_t = \boldsymbol{u}_t + \sum_{i=1}^{N_s} \mathrm{FFN}_i^{(s)}(\boldsymbol{u}_t) + \sum_{i \in S_t} g_{i,t}\, \mathrm{FFN}_i^{(r)}(\boldsymbol{u}_t),$$

where the gate $g_{i,t}$ is nonzero only for the selected routed experts.

Routing math: scores, top-K_r, and gating weights

There are many router variants. Two common patterns:

Classic softmax router

$$s_t = W_{\text{router}} u_t \in \mathbb{R}^{N_r}, \qquad p_{i,t} = \frac{\exp(s_{i,t})}{\sum_j \exp(s_{j,t})}, \qquad S_t = \text{Top-}K_r(p_t).$$

Combine weights are typically proportional to $p_{i,t}$ over the selected set.

A sigmoid-affinity router (DeepSeek-style)

Use a learned expert embedding $e_i \in \mathbb{R}^D$ and compute:

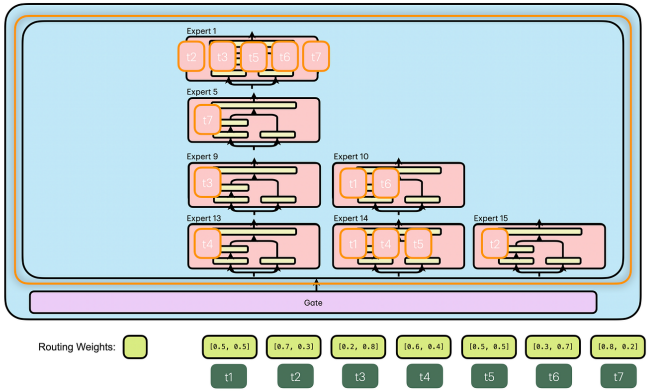
$$s_{i,t} = \sigma(u_t^\top e_i).$$

Select $S_t = \text{Top-}K_r(\{s_{i,t}\}_i)$ and normalize within the selected set:

$$\boxed{g_{i,t} = \frac{s_{i,t}}{\sum_{j \in S_t} s_{j,t}} \quad \text{for } i \in S_t.}$$

Discussion: pros and cons

Let's assume each expert lives on one GPU:



What are the benefits of having experts that each token does not need to go to all GPUs?
What are the downsides or potential problems?

Why load balancing is a throughput feature

If the router collapses onto a small subset of experts:

- those experts become hot spots,
- those ranks become stragglers,
- step time becomes "whoever is slowest."

Let $n_i^{(b)}$ be the number of tokens assigned to expert i for batch b. A rough utilization proxy is:

$$\text{utilization} \approx \frac{\mathbb{E}_b\left[n_i^{(b)}\right]}{\mathbb{E}_b\left[\max_i n_i^{(b)}\right]}.$$

If $\max_i n_i$ is much larger than the mean, you are paying for idle GPUs.

Systems interpretation

In MoE, imbalance is a **throughput bug**. It affects inference performance and quality.

One practical load-balancing mechanism: a routing bias controller

You can treat expert traffic like a control problem. One simple mechanism: add an expert-specific bias b_i to the routing decision.

Bias-adjusted selection rule

$$s_{i,t} + b_i \in \text{Top-}K_r(t) \quad \Rightarrow \quad \text{expert } i \text{ selected for token } t.$$

At the end of each training step, update b_i using a fixed step size γ:

$$\boxed{\text{overloaded} \Rightarrow b_i \leftarrow b_i - \gamma, \qquad \text{underloaded} \Rightarrow b_i \leftarrow b_i + \gamma.}$$

Why this works

Think of b_i as a feedback controller: it pushes traffic away from hot experts *without* needing a large auxiliary loss coefficient.

Case study: DeepSeek-style MoE

Use $D = 7168$, $F_e = 2048$, $N_s = 1$, $N_r = 256$, $K_r = 8$.

Each expert FFN has approximately:

$$P_{\text{expert}} \approx 3DF_e \approx 3 \cdot 7168 \cdot 2048 \approx 44\text{M params.}$$

So per MoE layer:

$$P_{\text{FFN,total}} \approx (1 + 256) \cdot 44\text{M} \approx 11.3\text{B}, \qquad P_{\text{FFN,active/token}} \approx (1 + 8) \cdot 44\text{M} \approx 0.40\text{B}.$$

Interpretation

Each token activates only a few percent of the FFN parameters in that layer, while the model can still index a very large FFN capacity across experts.

Baseline vs MoE: total vs activated FFN parameters

Design point	Total FFN params	Active params/token	Interpretation
Dense FFN baseline ($F \approx 4D$)	$\approx 12D^2$	same as total	Dense couples "total parameters" and "per-token compute"
MoE layer (N_s+N_r experts, top-K_r)	$\approx (N_s + N_r) \cdot 3DF_e$	$\approx (N_s + K_r) \cdot 3DF_e$	More total capacity without proportional per-token FFN cost

What changes in practice

You trade "one big FFN matmul" for "many smaller FFNs + routing + communication." If Expert Parallel (the way you parallelize MoE FFNs) is fast and balanced, MoE is a win; if not, the tax dominates.

Kimi K2: pushing sparsity harder under fixed activated compute

A common scaling move is: keep K_r fixed (activated experts per token), and increase N_r (more total expert capacity). Kimi K2 is a clean example because it reports a side-by-side comparison against a DeepSeek-style configuration.

Field	DeepSeek-V3	Kimi K2
# layers (n_{layers})	61	61
Total parameters	671B	1.04T
Activated parameters (per token)	37B	32B
Routed experts (N_r)	256	384
Experts active per token (K_r)	8	8
Shared experts (N_s)	1	1
Attention heads (N)	128	64
# dense layers	3	1

Nemotron 3 Nano: MoE as a default efficiency tool (even at smaller scale)

Nemotron 3 Nano is a hybrid Mamba–Transformer MoE model that emphasizes throughput and long-context efficiency.

Reported configuration highlights

- Total parameters: 31.6B (3.2B activated per forward pass; 3.6B incl. embeddings)
- Reported context length: up to 1M tokens on some long-context benchmarks
- Model width: $D = 2688$
- Expert FFN dimension: $F_e = 1856$
- Routed experts: $N_r = 128$ Active experts/token: $K_r = 6$
- Shared experts: $N_s = 2$

Interpretation

This is the same core MoE logic: keep activated compute small (few active experts) while buying extra capacity (many experts).

Nemotron 3 perspective: MoE is also about the *routed payload*

In practice, MoE performance is often decided by *movement* as much as by GEMM efficiency: small batches are weight-bandwidth-limited; large batches can become communication-limited.

AllToAll volume (worst-case).

$$\text{MoE comm per layer} \sim 2\,B_{\text{tok}}\,D\,b\,K_r \quad \Rightarrow \quad \text{per token per layer} \sim 2DbK_r.$$

For D=7168, b=2, K_r=8: $2DbK_r \approx 224$ KiB.

Nemotron 3 also tries to mitigate the AllToAll volume by using a smaller latent dimension $\ell < D$ for the expert communication.

LatentMoE (the idea)

Route and compute in a smaller latent dimension $\ell < D$ so the routed payload scales with ℓ:

$$\text{MoE comm} \sim 2\,B_{\text{tok}}\,\ell\,b\,K_r \quad \text{(instead of } 2\,B_{\text{tok}}\,D\,b\,K_r\text{)}.$$

Part C: Multi-Head Latent Attention (MLA) to break the KV bottleneck

MoE, Latent KV, and Sparse (or Hybrid) Long-Context

Notation for Part C (attention tensors, KV cache, and MLA internals)

Symbol	Meaning (and units / shape)
B	Batch size (number of sequences)
L	Sequence length / context length (tokens per sequence)
D	Model width / hidden size
n_{layers}	Number of transformer blocks in the stack
N	Number of *query* attention heads
H	Head dimension for standard attention; typically $D = N \cdot H$
N_{KV}	Number of key/value heads (MHA: $N_{\mathrm{KV}} = N$; GQA/MQA: $N_{\mathrm{KV}} \ll N$)
n_{attn}	Number of self-attention layers in a hybrid stack
Q	Queries (all heads), typically $Q \in \mathbb{R}^{B \times L \times N \times H}$
K, V	Cached keys/values, typically $K, V \in \mathbb{R}^{B \times L \times N_{\mathrm{KV}} \times H}$
K_t, V_t	Keys/values for a single token position t (one layer): $K_t, V_t \in \mathbb{R}^{N_{\mathrm{KV}} \times H}$
d_c	Cached content latent dimension in MLA
d_h^R	Cached RoPE-related key/query dimension in MLA
$\boldsymbol{c}_t, \boldsymbol{r}_t$	MLA cached latents for token t: $\boldsymbol{c}_t \in \mathbb{R}^{d_c}$, $\boldsymbol{r}_t \in \mathbb{R}^{d_h^R}$
W_c	Down-projection to content latent (typically $W_c \in \mathbb{R}^{d_c \times D}$)
$W_K^{\uparrow}, W_V^{\uparrow}$	Up-projections that reconstruct head-space K, V from $\boldsymbol{c}_t$

KV reduction: two knobs that look similar but behave differently

At long context, the decode bottleneck is often **reading KV from** HBM.

There are two cleanly different ways to shrink KV:

1. **Head-space sharing (GQA/MQA):** reduce N_{KV} (store fewer KV heads).
2. **Latent-space caching (MLA):** reduce *what is stored per token* by caching a latent and reconstructing head-space tensors on demand.

Key distinction

GQA/MQA store the same kind of cache, just *fewer heads*. MLA stores a *different representation*.

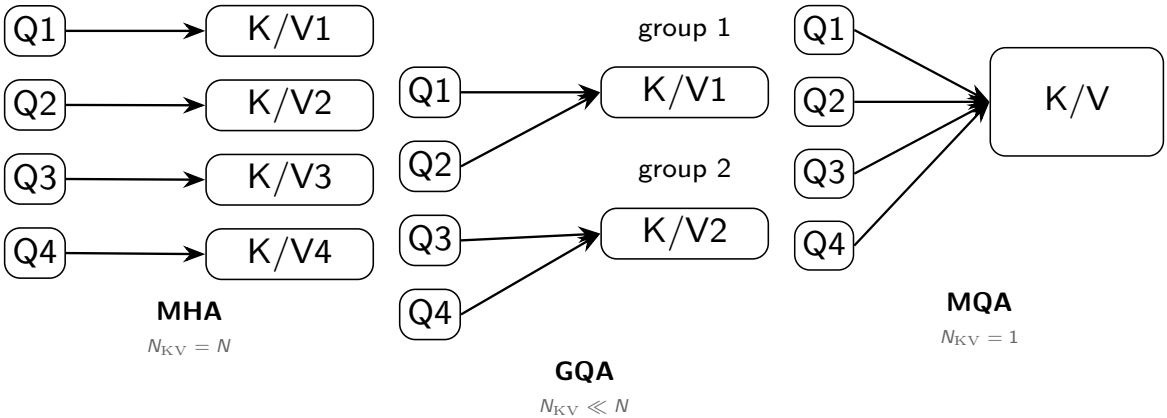
Cheat-sheet: how KV bytes scale

Mechanism	What is shared/compressed?	Decode-time KV bytes (scaling)
MHA (baseline)	No sharing; each query head has its own KV head ($N_{\mathrm{KV}} = N$)	bytes $\approx 2L\, n_{\text{layers}}\, N_{\mathrm{KV}}\, H\, b \approx 2L\, n_{\text{layers}}\, D\, b$
GQA	Query heads grouped; KV shared within each group ($N_{\mathrm{KV}} \ll N$)	bytes $\approx 2L\, n_{\text{layers}}\, N_{\mathrm{KV}}\, H\, b = (N_{\mathrm{KV}}/N) \cdot (2L\, n_{\text{layers}}\, D\, b)$
MQA	Extreme GQA: a single KV head ($N_{\mathrm{KV}} = 1$)	bytes $\approx (1/N) \cdot (2L\, n_{\text{layers}}\, D\, b)$
MLA	Cache a latent content vector plus a small RoPE-related part	bytes $\approx L\, n_{\text{layers}}\, (d_c + d_h^R)\, b$

Interpretation

GQA/MQA buy a factor of N_{KV}/N (sharing). MLA buys an absolute shrink in "state per token" (compression).

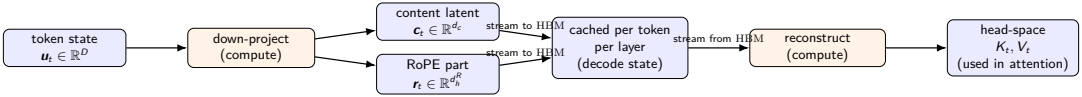
Visual: head-space sharing (MHA → GQA → MQA)



What changed?

Only the number of KV heads: the cached state scales with N_{KV}, not N.

Visual: latent-space caching (MLA) trades bytes for compute



One-line summary

Cache $(d_c + d_h^R)$ elements/token/layer (small), then reconstruct head-space tensors on demand (extra GEMMs). We basically reduce streaming to/from HBM traffic but we eat more compute. It's a trade-off.

Baseline: MHA/GQA KV cache size (bytes)

For one layer, caching keys and values for L tokens with N_{KV} KV heads of dimension H:

$$\text{KV elements per layer} \approx 2L\, N_{\mathrm{KV}}\, H.$$

Across n_{layers} layers:

$$\boxed{\text{KV bytes} \approx 2L\, n_{\text{layers}}\, N_{\mathrm{KV}}\, H\, b.}$$

For standard MHA, $N_{\mathrm{KV}} = N$ and typically $N \cdot H \approx D$, so:

$$\boxed{\text{KV bytes} \approx 2L\, n_{\text{layers}}\, D\, b.}$$

Interpretation

You can shrink KV by reducing N_{KV} (sharing) or by shrinking "state per token" (compression).

Knob 1: GQA/MQA (head-space sharing)

Everything about queries stays the same:

$$Q \in \mathbb{R}^{B \times L \times N \times H}.$$

Only keys/values have fewer heads:

$$K, V \in \mathbb{R}^{B \times L \times N_{\mathrm{KV}} \times H}, \qquad N_{\mathrm{KV}} \ll N.$$

What you buy

KV cache and decode bandwidth drop roughly linearly with N_{KV}:

$$\text{KV bytes} \propto N_{\mathrm{KV}} H.$$

Important clarity point

GQA/MQA reduce KV *bytes*, but the attention outputs are still produced for N query heads. You are mostly optimizing state and bandwidth, not eliminating attention computation.

Knob 2: MLA (latent-space caching)

MLA changes what is cached per token. Instead of caching head-space K, V, cache a smaller latent state and reconstruct tensors on demand.

Compression (what you store). For token state $\boldsymbol{u}_t \in \mathbb{R}^D$, compute and cache

$$\boldsymbol{c}_t = W_c\, \boldsymbol{u}_t \in \mathbb{R}^{d_c}, \qquad \boldsymbol{r}_t = \mathrm{RoPE}(W_r\, \boldsymbol{u}_t) \in \mathbb{R}^{d_h^R}.$$

Reconstruction (what you rebuild). When you run attention, reconstruct content KV and append the RoPE key:

$$\boldsymbol{k}_{t,i}^C = W_{K,i}^{\uparrow} \boldsymbol{c}_t, \quad \boldsymbol{v}_{t,i} = W_{V,i}^{\uparrow} \boldsymbol{c}_t, \quad \boldsymbol{k}_{t,i} = [\boldsymbol{k}_{t,i}^C; \boldsymbol{r}_t].$$

Cache scaling rule

$$\boxed{\text{KV bytes} \approx L\, n_{\text{layers}}\, (d_c + d_h^R)\, b.}$$

A subtle but important point: MLA decouples dimensions

In standard attention, it is common to assume $D = N \cdot H$. In MLA, the internal dimensions are intentionally decoupled.

Why you should care

If you blindly substitute $D = N \cdot H$ into MLA cache math, you will get nonsense. MLA chooses separate latent sizes (d_c, d_h^R, and sometimes query-specific latents) to trade off:

- cache bytes per token,
- reconstruction compute,
- model quality.

Concrete KV cache math at $L = 128\text{k}$ (back-of-envelope)

Use a representative backbone scale: $D = 7168$, $n_{\text{layers}} = 61$, $L = 128{,}000$.

If you had standard MHA in BF16

$$\text{KV bytes} \approx 2L\, n_{\text{layers}}\, D\, b = 2 \cdot 128{,}000 \cdot 61 \cdot 7168 \cdot 2 \approx 208 \text{ GiB}.$$

That is *per sequence*.

MLA in BF16 (back-of-envelope)

Cache $(d_c + d_h^R)$ elements/token/layer. With $d_c = 512$, $d_h^R = 64$:

$$\text{KV bytes} \approx L\, n_{\text{layers}}\, (576)\, b = 128{,}000 \cdot 61 \cdot 576 \cdot 2 \approx 8.4 \text{ GiB}.$$

The trade: reconstruction compute vs bandwidth saved

MLA introduces extra compute:

- down-project to latents (during prefill and while generating),
- up-project latents to head-space tensors inside attention.

Why this can still be a win

Decode is often bandwidth-bound: cutting KV bytes by an order of magnitude can reduce latency more than the added reconstruction GEMMs increase it.

This is the same move as FlashAttention, but applied to *decode state* rather than the attention score matrix:

$$\text{spend a little more compute} \quad \Rightarrow \quad \text{move a lot fewer bytes.}$$

Engineering reality

The win depends on kernel support, cache formats, and whether reconstruction can be fused or overlapped.

Part D: Sparse attention and hybrid stacks to break the L^2 wall

MoE, Latent KV, and Sparse (or Hybrid) Long-Context

Notation for Part D (sparse attention and hybrids)

Symbol	Meaning
B	Batch size (number of sequences)
L	Sequence length / context length (tokens per sequence)
D	Model width / hidden size
t	Query token position; $s < t$ denotes a past token position
k_{sel}	# selected past tokens/blocks per query token (top-k selection)
S_t	Selected index set for query token t; $\lvert S_t \rvert = k_{\text{sel}}$
H_I	# "indexer" heads in DSA-style selection
$I_{t,s}$	Indexer relevance score between query token t and past token s
$\boldsymbol{q}^I_{t,j}$	Indexer query vector for head j at query position t
$\boldsymbol{k}^I_s$	Indexer key vector for past position s
$w^I_{t,j}$	Per-query mixing weight for indexer head j (often nonnegative)
$\boldsymbol{h}_t$	Query-side representation used by the main attention kernel
$\boldsymbol{c}_s$	Cached content vector read for selected past positions (could be KV or MLA content)
n_{attn}	(Hybrid stacks) # attention layers; remaining layers are linear-time constant-state layers

Why FlashAttention is necessary but not sufficient

FlashAttention-style kernels are IO-aware and avoid materializing the $L \times L$ attention matrix in HBM. That is essential, but it does not remove the quadratic dependence on L.

A rough forward scaling for the attention score matmul is:

$$QK^\top \sim 2\,B\,L^2\,D \quad \mathrm{FLOPs}.$$

At $L \sim 10^5$–10^6, that term becomes operationally dominant even with excellent kernels.

So what has to change?

You either:

- make attention *sparse* (each query touches far fewer than L keys), or
- stop doing attention in most layers (hybrids).

Three broad ways people escape the L^2 regime

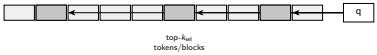
At a systems level, there are three broad strategies:

1. **Exact attention, better kernels:** improve constants and IO (FlashAttention and variants), but still L^2.
2. **Sparse attention within a layer:** compute attention on a structured subset (DSA/NSA).
3. **Hybrid stacks across depth:** use linear-time constant-state layers in most blocks; keep only a few attention layers for global routing.

This section

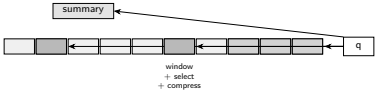
We will cover (2) in detail (DSA and NSA) and use hybrid stacks as a concrete example of (3).

Mental model: what set of past tokens can a query read?



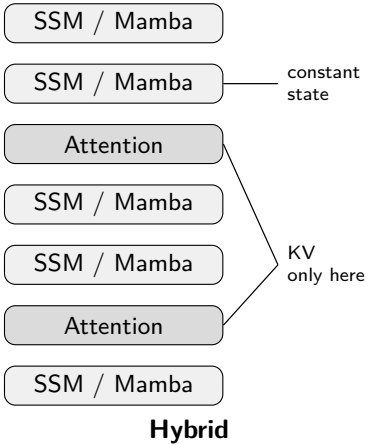
DSA

index → select



NSA

3 branches + gating



Hybrid

few attention layers

DSA: sparse within a layer | NSA: structured sparse paths | Hybrids: sparse across depth

DSA: sparse attention via a fast "lightning" indexer + top-k_{sel} selection

DeepSeek Sparse Attention (DSA) is "retrieval inside attention": score past tokens cheaply, then attend only to a small selected set.

Lightning indexer: head-wise retrieval scoring

For query token t, compute a *query-dependent head-gating vector*

$$\boldsymbol{w}_t^I \;=\; \frac{1}{\sqrt{H_I}}\, W_w \boldsymbol{h}_t \;\in\; \mathbb{R}^{H_I}, \qquad w_{t,j}^I = (\boldsymbol{w}_t^I)_j,$$

where $w_{t,j}^I$ is the gate for index head j.
Each index head produces a cheap non-negative match score

$$m_{t,s,j} = \mathrm{ReLU}\Big(\langle \boldsymbol{q}_{t,j}^I, \boldsymbol{k}_s^I \rangle\Big)\,.$$

The final retrieval score is a *mixture over heads*:

$$I_{t,s} = \sum_{j=1}^{H_I} w_{t,j}^I\, m_{t,s,j} = (\boldsymbol{w}_t^I)^\top \boldsymbol{m}_{t,s}, \qquad \boldsymbol{m}_{t,s} \in \mathbb{R}^{H_I}.$$

- $\boldsymbol{w}_t^I$ depends only on the query token t (shared across all s).
- It gates *which cheap index heads matter* for this query.

DSA: sparse attention via a fast "lightning" indexer + top-k_{sel} selection

Sparse attention on selected entries

Let $S_t = \text{Top-}k_{\text{sel}}(I_{t,:})$ be the highest-scoring past positions. Run *full* attention only on these:

$$\boldsymbol{u}_t \leftarrow \text{Attn}(\boldsymbol{h}_t,\ \{\boldsymbol{c}_s \mid s \in S_t\})\,.$$

- Two-stage pipeline: (1) score all past tokens cheaply, (2) pay full attention cost for only k_{sel} retrieved KV entries.

DSA: what it changes in the L^2 term (and what it costs)

A crude complexity comparison for the score matmul term:

$$\text{full:} \quad QK^\top \sim 2BL^2D \qquad \longrightarrow \qquad \text{sparse:} \quad QK^\top_{\text{sparse}} \sim 2BL\, k_{\text{sel}}\, D.$$

For fixed k_{sel}, the quadratic term becomes **linear in** L. Indexer is still quadratic but it's very cheap:

DSA

In real systems, DSA typically adds:

- an additional cached key buffer for the indexer (separate from the main KV/MLA cache),
- top-k selection kernels and irregular memory access,
- batching complications (prefill and decode have different indexer behavior),
- different cache formats / quantization schemes for different buffers.

Why DSA is implemented in an MQA-friendly way (and what it costs)

Sparse attention reduces compute *only* if it also plays nicely with KV sharing and cache layouts.

Alignment principle

If different query heads (or different heads within a GQA group) select different tokens, the system must fetch the *union* of those KV entries. That can preserve math savings but erase the intended KV traffic savings.

One concrete design choice

For kernel-level efficiency, a common implementation choice is to run DSA in an MQA-like mode where each selected KV entry is shared across many queries. This makes the sparse selection pattern compatible with "share once, use many times" in the attention kernel.

NSA: native sparse attention (hierarchical paths + gating)

NSA starts from a practical failure mode of simple sparsity: dropping either global context or local detail. The fix is hierarchical structure.

Three-branch structure

For each query token, compute attention along three paths:

1. **Sliding window:** recent local context
2. **Token selection:** important blocks/tokens (fine-grained)
3. **Token compression:** coarse summaries for global context

Then combine outputs with a learned gating mechanism.

Why this is nice

You get a structured way to keep both "what just happened" and "what matters globally" without paying full L^2.

Hybrid stacks (Nemotron 3): pay the attention tax only a few times

Hybrid models interleave attention with linear-time constant-state sequence layers (e.g., SSM/Mamba-style blocks).

Nemotron 3 sketch. Predominantly interleave MoE layers with cheaper Mamba-2 layers, and include only a small number of self-attention layers. Example: **Nemotron 3 Nano** uses $n_{\text{attn}} = 6$ attention layers out of $n_{\text{layers}} = 52$.

Two direct wins

- **Compute:** most layers are $O(L)$ instead of $O(L^2)$.
- **State:** KV cache exists only for the n_{attn} attention layers.

Serving implication

Mixed state: KV cache for attention layers and SSM state for sequence layers (often with different kernel/quantization sensitivities).

How to choose: DSA vs NSA vs Hybrid stacks (less clutter, clearer cues)

DSA

index $\rightarrow$ top-k_{sel}

Use when

- **Relevance is very sparse:** a few far tokens matter.
- You want **retrieval-like** access pattern at long L.

Watch out for

- **Recall risk:** if top-k misses, you lose the signal.
- **Serving tax:** fast top-k + gathers + irregular reads.

Key knobs: k_{sel}, block size, **group-consistent selection** under GQA/MQA.

NSA

window + select + compress + gating

Use when

- You need **both** local detail *and* global context.
- Relevance is **not extremely sparse** (many tokens contribute).

Watch out for

- **Tuning risk:** too much compression $\rightarrow$ blur; too little $\rightarrow$ slow.
- More moving parts: window/select/compress + gating kernels.

Key knobs: window w, selection budget k_{sel}, compression rate, gating stability.

Hybrid stacks

few attention layers + many $O(L)$ layers

Use when

- Attention is too expensive **across depth**, not just within one layer.
- You can **change the backbone and retrain** (SSM/Mamba-style layers).

Watch out for

- **Global mixing risk:** too few attention layers $\rightarrow$ weak routing.
- Mixed runtime: KV state (attn layers) + SSM state (everywhere).

Key knobs: n_{attn} and placement; (often) GQA/MLA inside the kept attention layers.

Putting it together: one modern stack (DeepSeek-V3 family)

A useful way to read recent long-context models is: each mechanism targets a different dominant term.

A representative combination

1. **MoE for the FFN bottleneck:** routed expert capacity grows while activated compute stays bounded (e.g., 256 routed experts with top-K_r=8).
2. **MLA for the KV bottleneck:** cache a latent per token per layer (e.g., d_c=512, d_h^R=64) and reconstruct on demand.
3. **Sparse attention as a long-context add-on:** add a fast indexer + top-k_{sel} selection mechanism via continued training to avoid paying full L^2 at extreme context.

Why this composition makes sense

You want to be selective along three axes at once: *which FFN parameters*, *what decode state*, and *which past tokens*.

Summary & wrap-up

MoE, Latent KV, and Sparse (or Hybrid) Long-Context

Putting it together: one bottleneck $\rightarrow$ one knob

Pressure point	**What you should change**	**Typical architecture knobs**
FFN compute/params	Reduce activated FFN compute per token while increasing total capacity	MoE (top-K_r), shared experts; Latent-MoE (shrink routed payload ℓ)
KV cache / decode BW	Reduce bytes of KV streamed per generated token	GQA/MQA ($N_{\mathrm{KV}} \downarrow$); MLA ($d_c + d_h^R \downarrow$); hybrids ($n_{\mathrm{attn}} \downarrow$)
Long-context L^2	Reduce how many keys each query token touches, or reduce attention layers	DSA/NSA ($k_{\mathrm{sel}} \ll L$); hybrids (few attention layers, mostly $O(L)$ layers)

Common theme

All three are "selectivity" tricks: select experts, select state dimensions, or select tokens/layers.

The systems stack requirements

These architectures are only fast if the systems stack supports them:

- **MoE:** high-bandwidth AllToAll (expert parallelism), robust load balancing, and good topology mapping.
- **LatentMoE:** reduces routed payload size, easing EP bandwidth and improving latency.
- **MLA:** kernel support for latent cache formats + reconstruction; quantization that preserves quality.
- **Sparse attention:** fast top-k selection kernels; cache layouts that work with continuous batching.
- **Hybrids:** runtime support for mixed state (KV + SSM), heterogeneous kernels and quantization.

Preview of the next step

The next lecture formalizes *parallelism strategies* (tensor/pipeline/data/context/expert) and how these architectures change communication patterns.

Key takeaways

1. **Architectures change the dominant term.** MoE targets FFN coupling; MLA targets KV bytes; sparse/hybrids target L^2.
2. **The systems tax is the whole story.** Routing, AllToAll, reconstruction GEMMs, and cache layouts decide whether the idea is a win.
3. **Selectivity is the unifying principle.** Modern models spend compute and bandwidth only where it matters.