Efficient Inference: Compute Less, Move Fewer Bytes

At inference the model is often waiting on memory, not arithmetic

Prof. Nipun Batra

ES 667 · Deep Learning · IIT Gandhinagar

Diagnose before optimizing

Why this belongs in a deep-learning course

An accurate model that cannot meet its **latency, memory, or cost** budget is not usable.

Why this belongs in a deep-learning course

An accurate model that cannot meet its **latency, memory, or cost** budget is not usable.

Moving weights and cached activations is often the bottleneck — not arithmetic.

Why this belongs in a deep-learning course

An accurate model that cannot meet its **latency, memory, or cost** budget is not usable.

Moving weights and cached activations is often the bottleneck — not arithmetic.

Architecture, quality, and deployment are **coupled**. A design that ignores inference cost is only half a design. This lecture is the systems half.

Why this belongs in a deep-learning course

An accurate model that cannot meet its **latency, memory, or cost** budget is not usable.

Moving weights and cached activations is often the bottleneck — not arithmetic.

Architecture, quality, and deployment are **coupled**. A design that ignores inference cost is only half a design. This lecture is the systems half.

one worked calculation (KV-cache memory) · one essential distinction (prefill vs decode)

The two phases of autoregressive inference

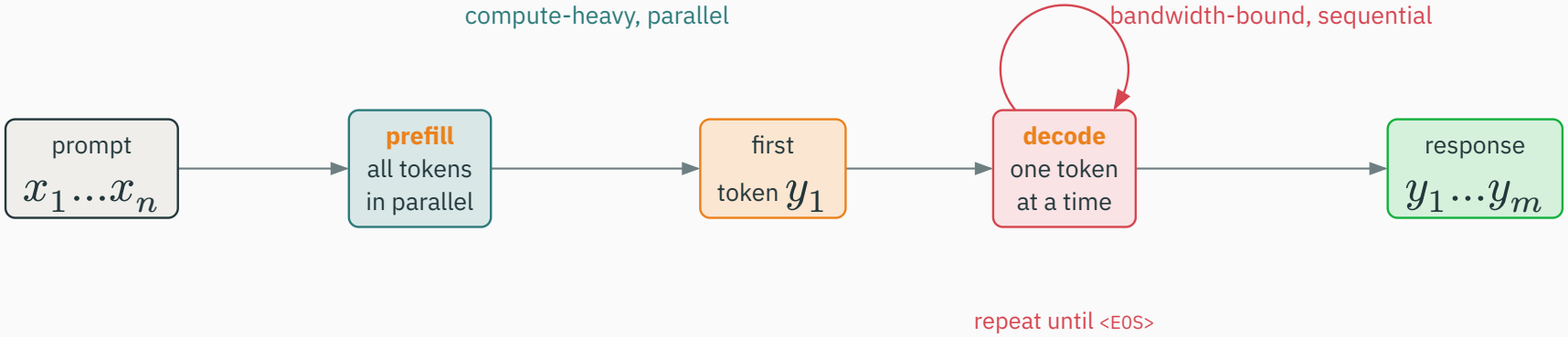
V VISUAL

Generation is **not** one uniform computation:

The two phases of autoregressive inference

V VISUAL

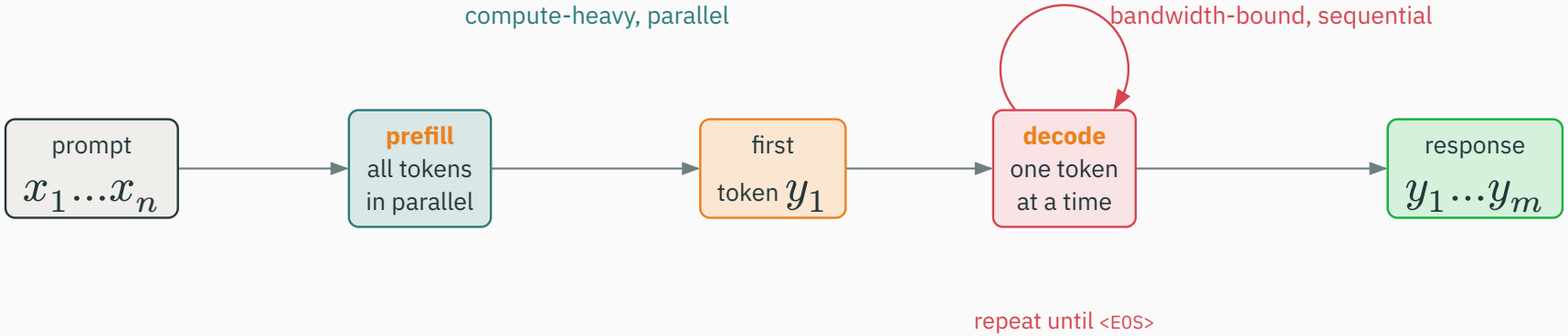
Generation is **not** one uniform computation:



The two phases of autoregressive inference

V VISUAL

Generation is **not** one uniform computation:



processing the prompt and producing one token at a time have **different shapes, costs, and optimizations**

Prefill — process the prompt in parallel

D DERIVATION

Feed **all** n prompt tokens through the model at once:

Prefill — process the prompt in parallel

D DERIVATION

Feed **all** n prompt tokens through the model at once:

- large **matrix–matrix** products keep the accelerator busy;
- each weight is reused across **many** token vectors — high arithmetic intensity;
- attention cost still grows with sequence length ($O(n^2 d)$).

Prefill — process the prompt in parallel

D DERIVATION

Feed **all** n prompt tokens through the model at once:

- large **matrix–matrix** products keep the accelerator busy;
- each weight is reused across **many** token vectors — high arithmetic intensity;
- attention cost still grows with sequence length ($O(n^2 d)$).

prefill is typically **compute-bound** — the arithmetic units are the limit

Decode — one token at a time

D DERIVATION

Produce the next token (or a small batch), then repeat:

Decode — one token at a time

D DERIVATION

Produce the next token (or a small batch), then repeat:

- each step needs the **full model weights** plus **all prior key/value state**;
- the work is **matrix–vector-like**: little reuse per fetched weight;
- steps are **sequential** — token $t+1$ waits for token t.

Decode — one token at a time

D DERIVATION

Produce the next token (or a small batch), then repeat:

- each step needs the **full model weights** plus **all prior key/value state**;
- the work is **matrix–vector-like**: little reuse per fetched weight;
- steps are **sequential** — token $t + 1$ waits for token t.

> decode is typically **memory-bound** — the device spends its time fetching weights and cache

Compute-bound versus memory-bound

D DERIVATION

Two different resources can be the ceiling:

Compute-bound versus memory-bound

D DERIVATION

Two different resources can be the ceiling:

Regime	What limits you
compute-bound	arithmetic units saturated; more FLOPs = more time
memory-bound	device idles waiting to **fetch** weights / activations

Compute-bound versus memory-bound

D DERIVATION

Two different resources can be the ceiling:

Regime	What limits you
compute-bound	arithmetic units saturated; more FLOPs = more time
memory-bound	device idles waiting to **fetch** weights / activations

optimize the **right** thing: shrinking FLOPs has little effect unless it also lowers byte traffic

The bytes intuition

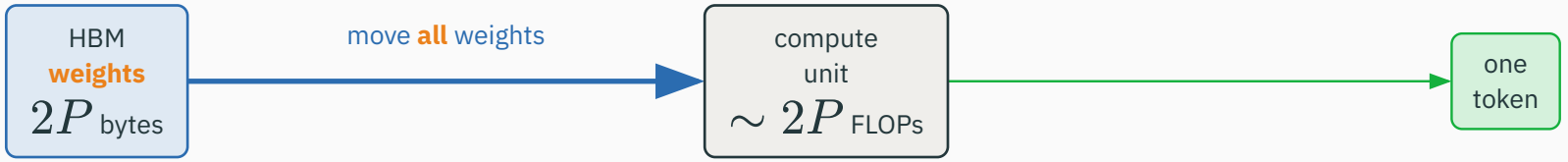
V VISUAL

For a single generated token, a huge matrix–vector product does **few operations per byte** of weight fetched:

The bytes intuition

V VISUAL

For a single generated token, a huge matrix–vector product does **few operations per byte** of weight fetched:

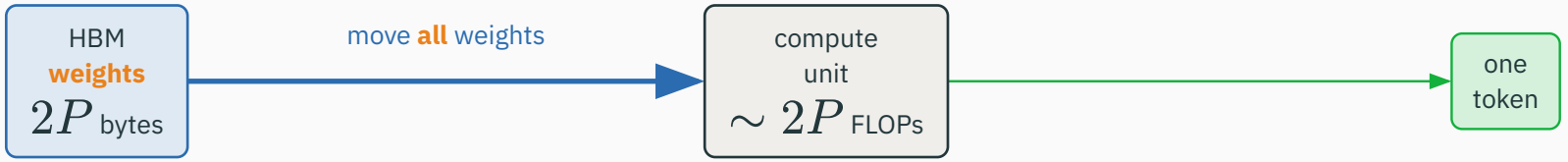


≈ 1 FLOP per byte fetched $\longrightarrow$ stalls on memory

The bytes intuition

V VISUAL

For a single generated token, a huge matrix–vector product does **few operations per byte** of weight fetched:



≈ 1 FLOP per byte fetched $\longrightarrow$ stalls on memory

the accelerator can sit **idle** even with abundant FLOPs — it is waiting on the weight fetch

The roofline picture

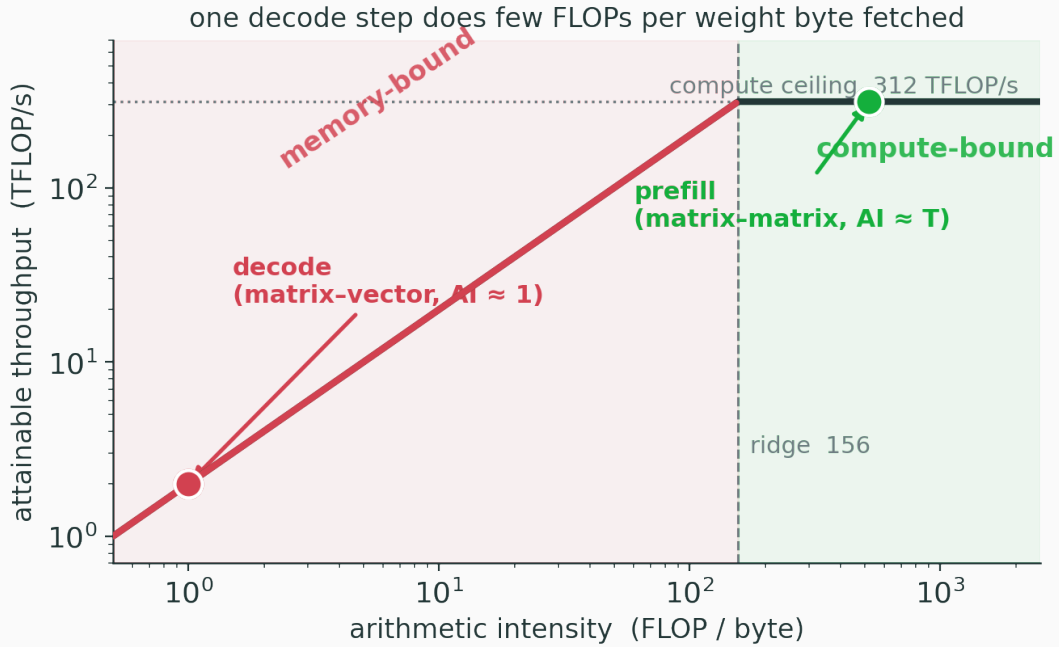
V VISUAL

Attainable throughput is capped by **bandwidth** until arithmetic intensity crosses the ridge:

The roofline picture

V VISUAL

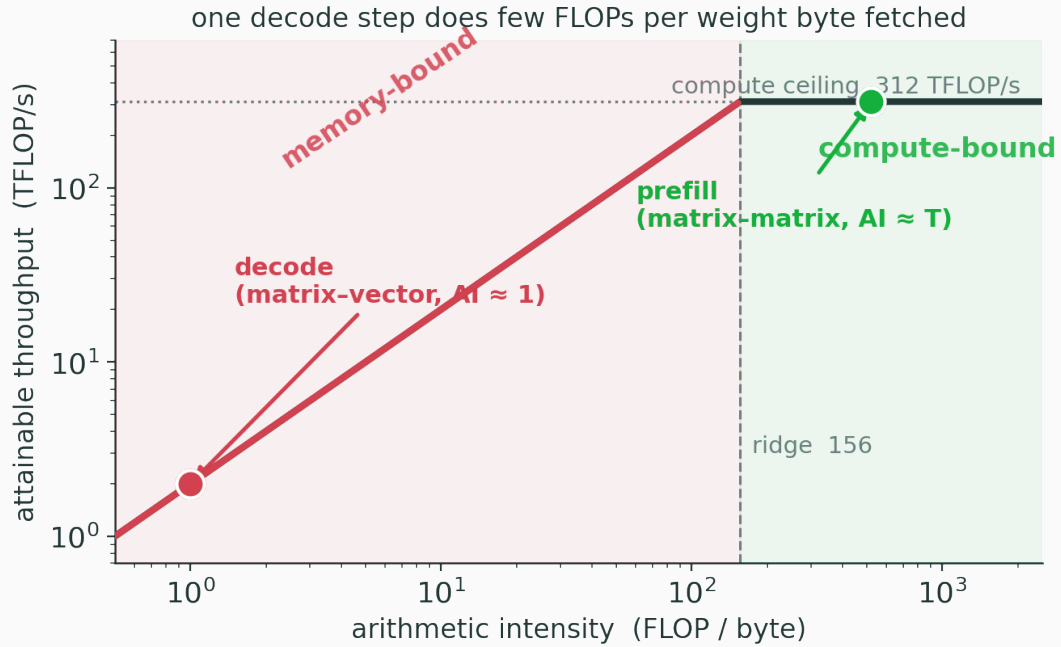
Attainable throughput is capped by **bandwidth** until arithmetic intensity crosses the ridge:



The roofline picture

V VISUAL

Attainable throughput is capped by **bandwidth** until arithmetic intensity crosses the ridge:



decode (AI ≈ 1) sits deep in the memory-bound region; **prefill** (AI $\approx T$) reaches the compute ceiling

The bytes intuition, said plainly

Q QUESTION

The bytes intuition, said plainly

Q QUESTION

abundant FLOPs, starved bandwidth → the accelerator waits on memory

The bytes intuition, said plainly

Q QUESTION

abundant FLOPs, starved bandwidth → the accelerator waits on memory

decode reads $\approx 2P$ bytes to do $\approx 2P$ FLOPs: about **one** FLOP per byte

The bytes intuition, said plainly

Q QUESTION

abundant FLOPs, starved bandwidth → the accelerator waits on memory

decode reads $\approx 2P$ bytes to do $\approx 2P$ FLOPs: about **one** FLOP per byte

This is why a device with "enough FLOPs" can still be slow at generation. The fix is rarely "more compute" — it is **move fewer bytes** and **reuse what you already moved**.

The inference budget worksheet

Q QUESTION

Every deployment must name its constraints **before** optimizing:

The inference budget worksheet

Q QUESTION

Every deployment must name its constraints **before** optimizing:

latency	throughput	VRAM	quality	cost
time to token	tokens / s	fits the device?	accuracy	$ / token

The inference budget worksheet

Q QUESTION

Every deployment must name its constraints **before** optimizing:

latency	throughput	VRAM	quality	cost
time to token	tokens / s	fits the device?	accuracy	$ / token

every optimization improves **one or more** of these — with a **trade-off** in the others

KV cache: reuse what attention already computed

Recall causal attention

D DERIVATION

To attend from a new token t, a layer needs the keys and values of **all** tokens $1, \ldots, t$:

Recall causal attention

D DERIVATION

To attend from a new token t, a layer needs the keys and values of **all** tokens $1, \ldots, t$:

$$\text{Attention}(q_t, K_{1:t}, V_{1:t}) = \text{softmax}\left(q_t K_{1:t}^\top / \sqrt{d}\right) V_{1:t}$$

Recall causal attention

D DERIVATION

To attend from a new token t, a layer needs the keys and values of **all** tokens $1, \ldots, t$:

$$\mathrm{Attention}(q_t, K_{1:t}, V_{1:t}) = \mathrm{softmax}\left(q_t K_{1:t}^{\top} / \sqrt{d}\right) V_{1:t}$$

the new query reads the **whole prefix** — so the prefix's keys and values must be available

The naive mistake

D DERIVATION

Re-run the entire Transformer on the full prefix after **every** new token:

The naive mistake

D DERIVATION

Re-run the entire Transformer on the full prefix after **every** new token:

```
for t in range(1, T):
    # recompute K, V for ALL of tokens 1..t  ← wasteful
    K, V = project_all(x[:t])
    h = attention(q[t], K, V)
```

The naive mistake

D DERIVATION

Re-run the entire Transformer on the full prefix after **every** new token:

```
for t in range(1, T):
    # recompute K, V for ALL of tokens 1..t  ← wasteful
    K, V = project_all(x[:t])
    h = attention(q[t], K, V)
```

Tokens $1, \ldots, t-1$ have not changed, yet their key/value projections are recomputed at every step — $O(T)$ redundant work per token, $O(T^2)$ overall.

The KV-cache idea

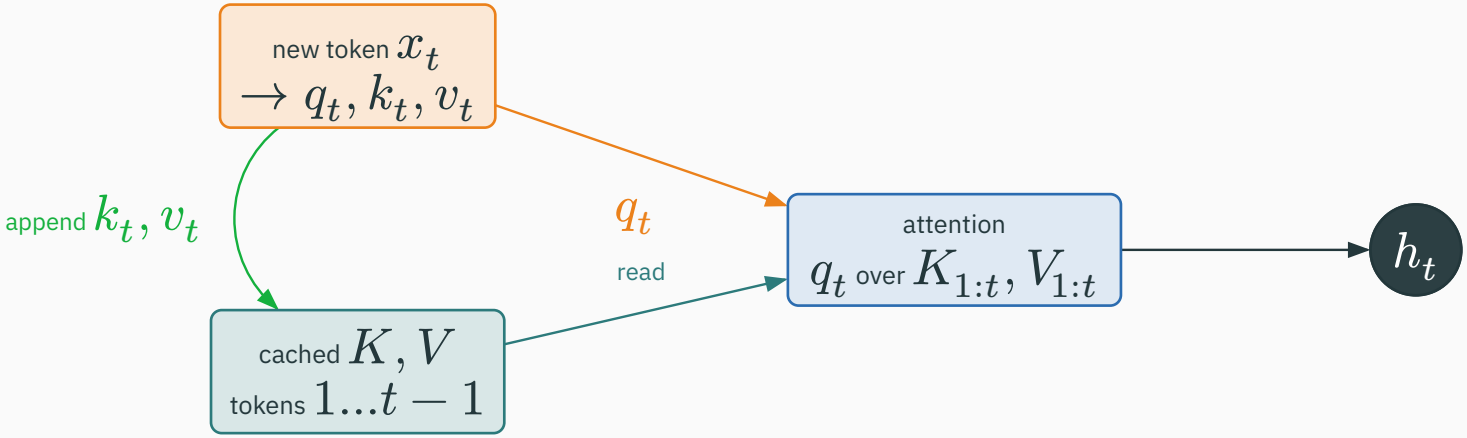
V VISUAL

Compute each token's K, V **once**, then keep them:

The KV-cache idea

V VISUAL

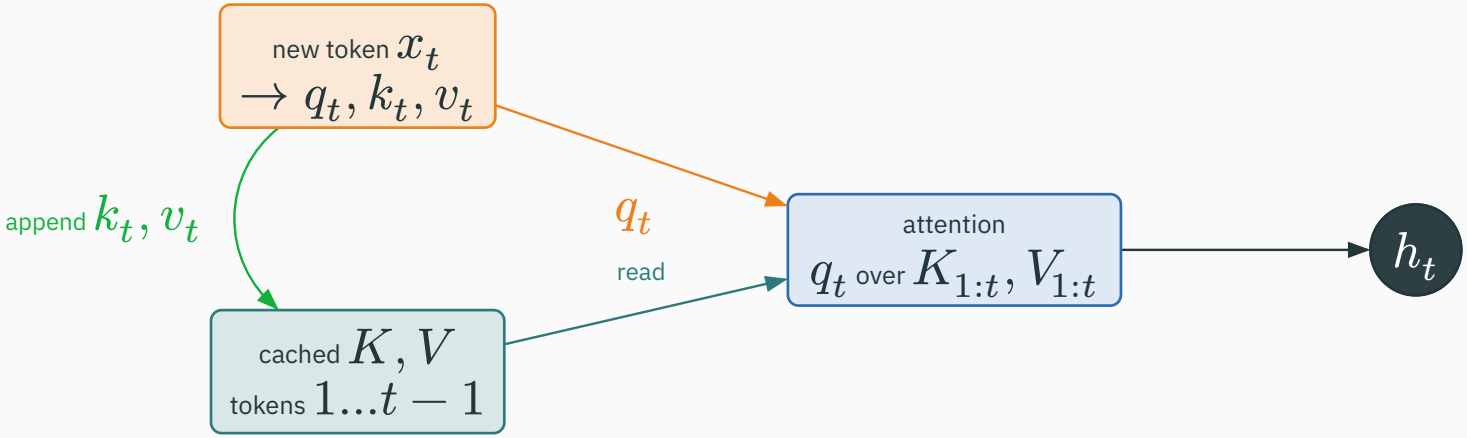
Compute each token's K, V **once**, then keep them:



The KV-cache idea

V VISUAL

Compute each token's K, V **once**, then keep them:



earlier tokens $\rightarrow$ cached K, V · new token $\rightarrow$ new q, k, v · append k_t, v_t and read the whole cache

Interactive: the KV cache

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/kv-cache

Step a short prompt through decode: watch each new token compute only its own q_t, k_t, v_t, **append** k_t, v_t to the growing cache, and attend over the whole stored history — no recomputation of old projections.

What caching saves — and what it does not

D DERIVATION

What caching saves — and what it does not

D DERIVATION

Saves — the projections of old tokens. Each step computes only q_t, k_t, v_t for the **new** token.

Does not remove — attention over the growing history. The new query still **reads** all cached keys/values.

What caching saves — and what it does not

D DERIVATION

Saves — the projections of old tokens. Each step computes only q_t, k_t, v_t for the **new** token.

Does not remove — attention over the growing history. The new query still **reads** all cached keys/values.

KV cache turns repeated **projection** into a **memory lookup** — the read still grows with context

KV-cache shape, per layer

D DERIVATION

At every layer we store **two** tensors:

KV-cache shape, per layer

D DERIVATION

At every layer we store **two** tensors:

$$K, V \in \mathbb{R}^{B \times T \times n_{\text{kv}} \times d_h}$$

KV-cache shape, per layer

D DERIVATION

At every layer we store **two** tensors:

$$K, V \in \mathbb{R}^{B \times T \times n_{\text{kv}} \times d_h}$$

Symbol	Meaning
B	batch size (concurrent sequences)
T	current sequence length
n_{kv}	number of key/value heads
d_h	dimension per head

KV-cache shape, per layer

D DERIVATION

At every layer we store **two** tensors:

$$K, V \in \mathbb{R}^{B \times T \times n_{\text{kv}} \times d_h}$$

Symbol	Meaning
B	batch size (concurrent sequences)
T	current sequence length
n_{kv}	number of key/value heads
d_h	dimension per head

two tensors, at **every** one of the L layers

KV-cache memory approximation

D DERIVATION

Sum the two tensors over all L layers, at b bytes per stored value:

KV-cache memory approximation

D DERIVATION

Sum the two tensors over all L layers, at b bytes per stored value:

$$M_{\mathrm{KV}} \approx 2 \cdot L \cdot B \cdot T \cdot n_{\mathrm{kv}} \cdot d_h \cdot b$$

KV-cache memory approximation

D DERIVATION

Sum the two tensors over all L layers, at b bytes per stored value:

$$M_{\mathrm{KV}} \approx 2 \cdot L \cdot B \cdot T \cdot n_{\mathrm{kv}} \cdot d_h \cdot b$$

factor 2	L layers	$n_{\mathrm{kv}} \cdot d_h$	b bytes
K and V	per layer	head geometry	fp16 $\rightarrow 2$

KV-cache memory approximation

D DERIVATION

Sum the two tensors over all L layers, at b bytes per stored value:

$$M_{\mathrm{KV}} \approx 2 \cdot L \cdot B \cdot T \cdot n_{\mathrm{kv}} \cdot d_h \cdot b$$

factor 2	L layers	$n_{\mathrm{kv}} \cdot d_h$	b bytes
K and V	per layer	head geometry	fp16 $\rightarrow 2$

linear in L, in B, and in T — three ways for the cache to grow

Board calculation

D DERIVATION

Take $L = 32$, $B = 1$, $T = 4096$, $n_{\text{kv}} = 8$, $d_h = 128$, fp16 so $b = 2$:

Board calculation

D DERIVATION

Take $L = 32$, $B = 1$, $T = 4096$, $n_{\text{kv}} = 8$, $d_h = 128$, fp16 so $b = 2$:

$$M_{\text{KV}} \approx 2 \cdot 32 \cdot 4096 \cdot 8 \cdot 128 \cdot 2$$

Board calculation

D DERIVATION

Take $L = 32$, $B = 1$, $T = 4096$, $n_{\text{kv}} = 8$, $d_h = 128$, fp16 so $b = 2$:

$$M_{\text{KV}} \approx 2 \cdot 32 \cdot 4096 \cdot 8 \cdot 128 \cdot 2$$

Multiply in stages:

$$2 \cdot 32 = 64 \quad \rightarrow \quad \cdot 4096 = 262{,}144 \quad \rightarrow \quad \cdot 8 = 2{,}097{,}152$$

Board calculation

D DERIVATION

Take $L = 32$, $B = 1$, $T = 4096$, $n_{\text{kv}} = 8$, $d_h = 128$, fp16 so $b = 2$:

$$M_{\text{KV}} \approx 2 \cdot 32 \cdot 4096 \cdot 8 \cdot 128 \cdot 2$$

Multiply in stages:

$$2 \cdot 32 = 64 \quad \rightarrow \quad \cdot 4096 = 262{,}144 \quad \rightarrow \quad \cdot 8 = 2{,}097{,}152$$

$$\cdot 128 = 268{,}435{,}456 \quad \rightarrow \quad \cdot 2 = 536{,}870{,}912 \text{ bytes}$$

Board calculation

D DERIVATION

Take $L = 32$, $B = 1$, $T = 4096$, $n_{\text{kv}} = 8$, $d_h = 128$, fp16 so $b = 2$:

$$M_{\text{KV}} \approx 2 \cdot 32 \cdot 4096 \cdot 8 \cdot 128 \cdot 2$$

Multiply in stages:

$$2 \cdot 32 = 64 \quad \rightarrow \quad \cdot 4096 = 262{,}144 \quad \rightarrow \quad \cdot 8 = 2{,}097{,}152$$

$$\cdot 128 = 268{,}435{,}456 \quad \rightarrow \quad \cdot 2 = 536{,}870{,}912 \text{ bytes}$$

≈ 512 **MiB — for a single sequence, before concurrency**

The key trade-off

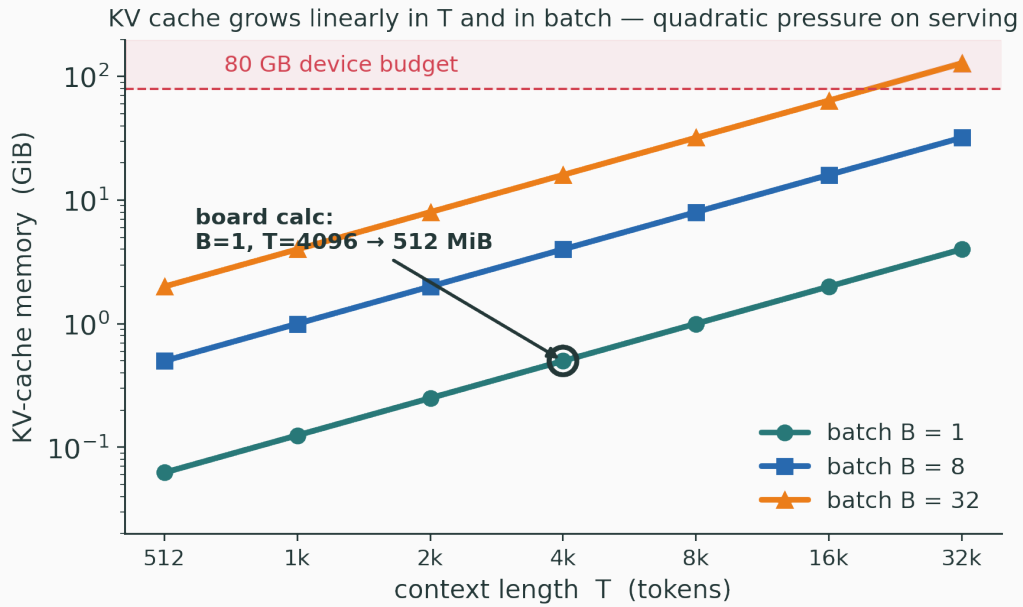
V VISUAL

KV cache buys decode speed by **spending memory**:

The key trade-off

V VISUAL

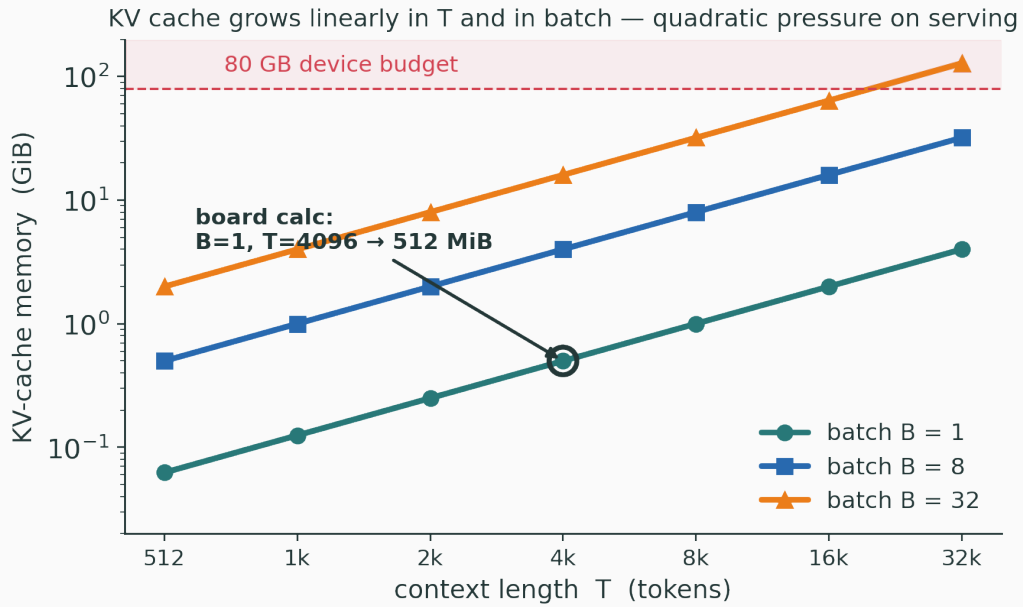
KV cache buys decode speed by **spending memory**:



The key trade-off

V VISUAL

KV cache buys decode speed by **spending memory**:



long contexts and high concurrency make **memory** — not parameters — the dominant capacity limit

Architectural response: fewer KV heads

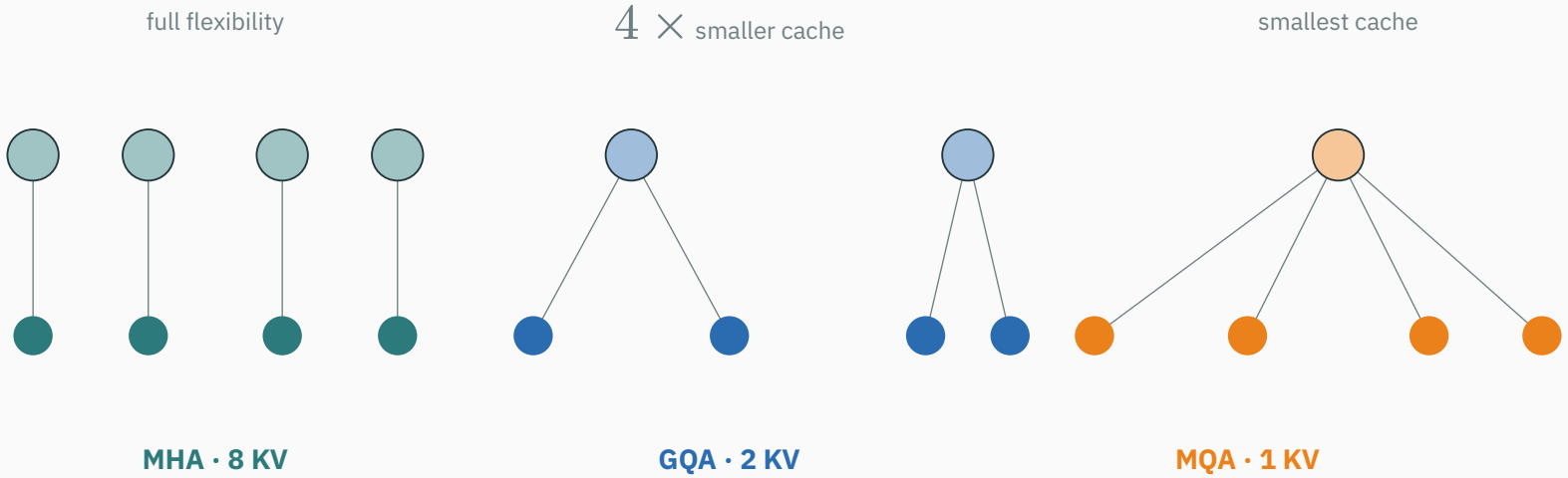
V VISUAL

Multi-query (MQA) and grouped-query (GQA) attention **share** keys/values across query heads:

Architectural response: fewer KV heads

V VISUAL

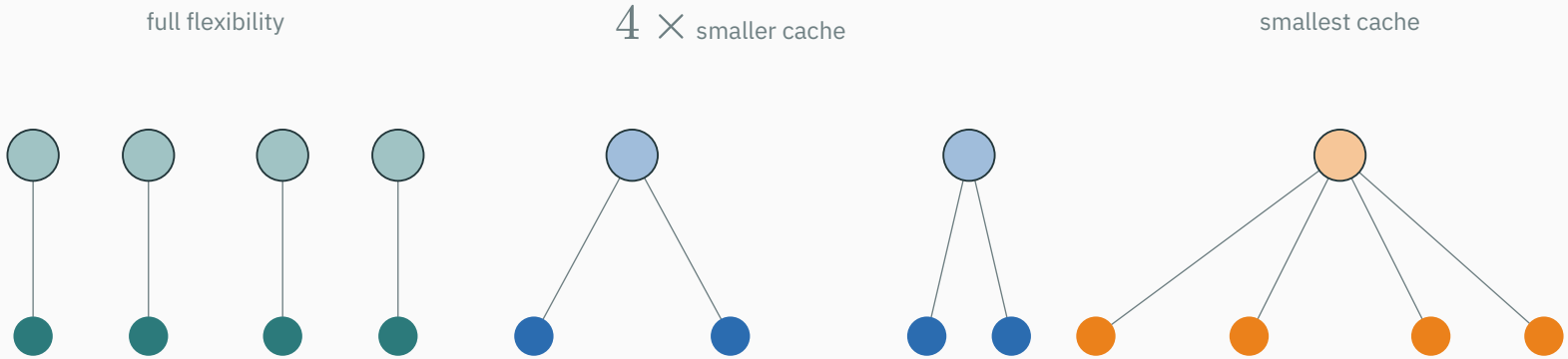
Multi-query (MQA) and grouped-query (GQA) attention **share** keys/values across query heads:



Architectural response: fewer KV heads

V VISUAL

Multi-query (MQA) and grouped-query (GQA) attention **share** keys/values across query heads:



$n_{\text{kv}} < n_{\text{query}}$ shrinks the cache by exactly $n_{\text{query}}/n_{\text{kv}}$, keeping many query perspectives

The main inference levers

Quantization: store fewer bits

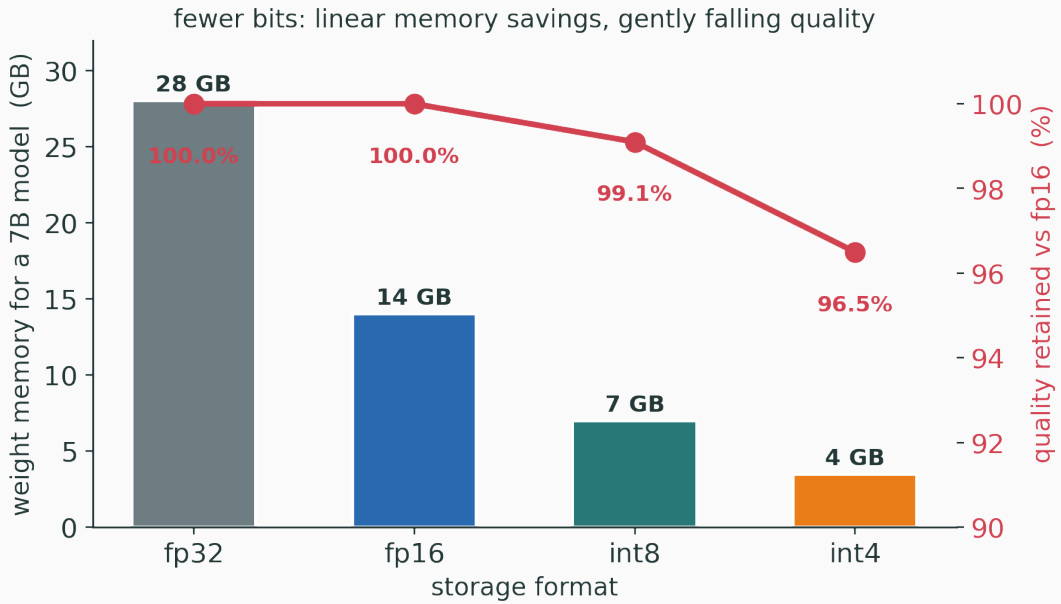
V VISUAL

Represent weights with **8-bit** or **4-bit** values instead of 16:

Quantization: store fewer bits

V VISUAL

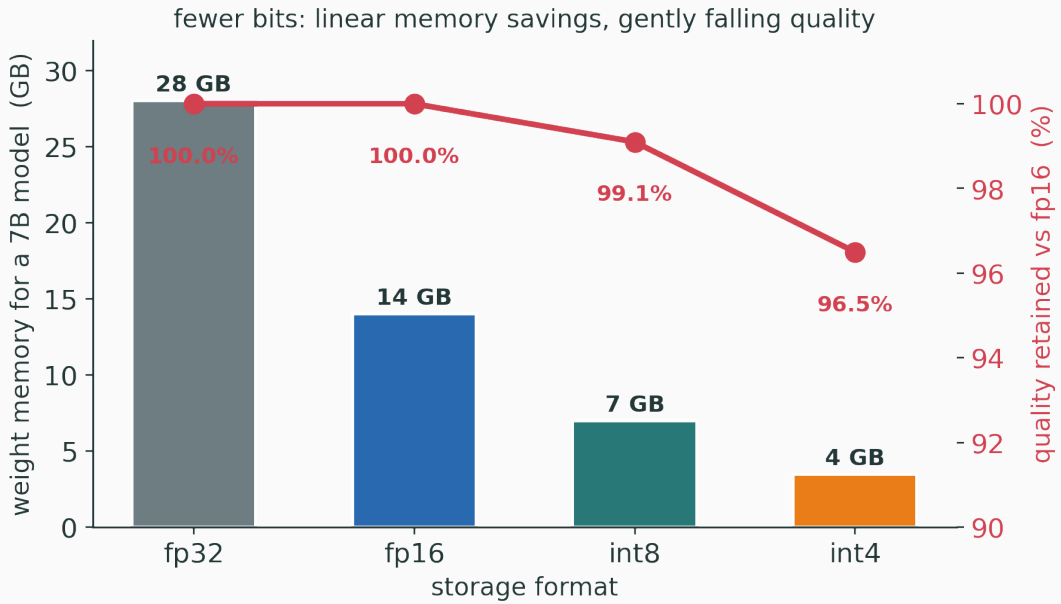
Represent weights with **8-bit** or **4-bit** values instead of 16:



Quantization: store fewer bits

V VISUAL

Represent weights with **8-bit** or **4-bit** values instead of 16:



fewer bytes ⟶ less memory and faster bandwidth-bound decode; approximation can cost quality

Quantization is not one thing

D DERIVATION

Distinguish **what** you quantize and **when**:

Quantization is not one thing

D DERIVATION

Distinguish **what** you quantize and **when**:

- **weight-only** — shrink the parameters (helps the memory-bound decode read);
- **activation / KV-cache** — also quantize the running state (helps long-context memory);
- **post-training (PTQ)** vs **quantization-aware training (QAT)** — calibrate after, or train with, low precision.

Quantization is not one thing

D DERIVATION

Distinguish **what** you quantize and **when**:

- **weight-only** — shrink the parameters (helps the memory-bound decode read);
- **activation / KV-cache** — also quantize the running state (helps long-context memory);
- **post-training (PTQ)** vs **quantization-aware training (QAT)** — calibrate after, or train with, low precision.

each choice trades quality, memory, and engineering effort differently

Attention kernels: avoid giant intermediates

V VISUAL

Standard attention can build a large $T \times T$ score matrix. IO-aware kernels compute **exact** attention while moving less:

Attention kernels: avoid giant intermediates

V VISUAL

Standard attention can build a large $T \times T$ score matrix. IO-aware kernels compute **exact** attention while moving less:

the $T \times T$ score matrix is **never** written to HBM

Attention kernels: avoid giant intermediates

V VISUAL

Standard attention can build a large $T \times T$ score matrix. IO-aware kernels compute **exact** attention while moving less:

the $T \times T$ score matrix is **never** written to HBM

FlashAttention is the canonical example — same answer, far less high-bandwidth-memory traffic (we do not teach the CUDA)

Batching and scheduling

D DERIVATION

Serving many requests **together** raises hardware utilization:

Batching and scheduling

D DERIVATION

Serving many requests **together** raises hardware utilization:

- one weight fetch is amortized across the whole batch (recall: decode is memory-bound);
- but requests have **different** prompt lengths and generation durations.

Batching and scheduling

D DERIVATION

Serving many requests **together** raises hardware utilization:

- one weight fetch is amortized across the whole batch (recall: decode is memory-bound);
- but requests have **different** prompt lengths and generation durations.

The systems problem: pack varied requests to keep the device busy **without** stalling any one of them. We name the problem — we do not build a serving stack.

Paged / block-based KV storage

D DERIVATION

Treat the KV cache as **reusable memory blocks**, not one contiguous reservation per request:

Paged / block-based KV storage

D DERIVATION

Treat the KV cache as **reusable memory blocks**, not one contiguous reservation per request:

Contiguous — reserve max-length up front → fragmentation and waste.

Paged — allocate small blocks as the sequence grows → share and reclaim freely.

Paged / block-based KV storage

D DERIVATION

Treat the KV cache as **reusable memory blocks**, not one contiguous reservation per request:

Contiguous — reserve max-length up front → fragmentation and waste.

Paged — allocate small blocks as the sequence grows → share and reclaim freely.

the resource-management idea — not the framework internals — is what matters here

Speculative decoding

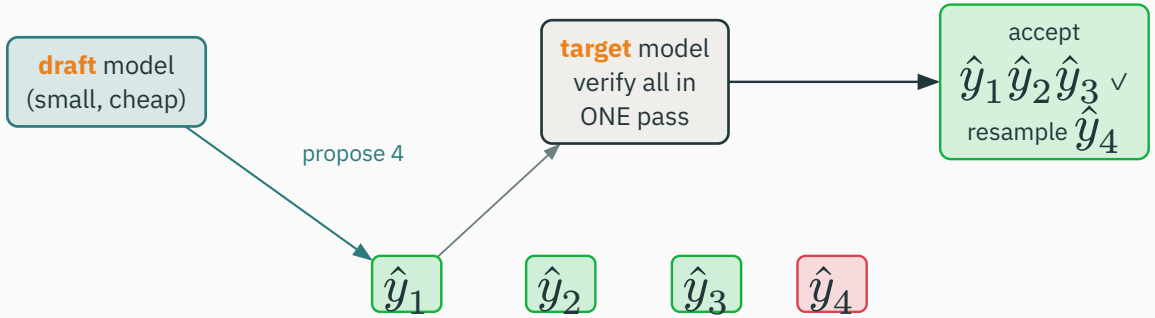
V VISUAL

Break the sequential decode bottleneck with a cheap **draft** and a trusted **verify**:

Speculative decoding

V VISUAL

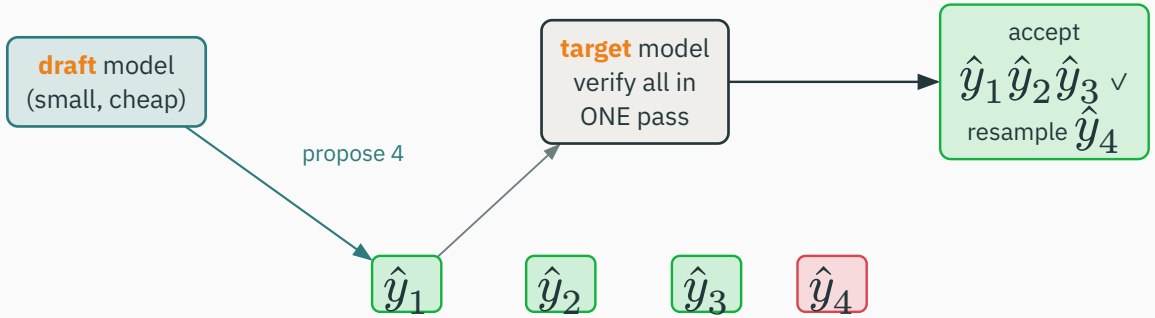
Break the sequential decode bottleneck with a cheap **draft** and a trusted **verify**:



Speculative decoding

V VISUAL

Break the sequential decode bottleneck with a cheap **draft** and a trusted **verify**:



the **target** model preserves output quality; speedup depends on how often draft tokens are accepted

Why speculative decoding can help

D DERIVATION

It trades **cheap draft computation** for **fewer sequential expensive target steps**:

Why speculative decoding can help

D DERIVATION

It trades **cheap draft computation** for **fewer sequential expensive target steps**:

- one target forward pass verifies several proposed tokens **in parallel**;
- accept the matching prefix $\rightarrow$ many decode steps collapse into one.

Why speculative decoding can help

D DERIVATION

It trades **cheap draft computation** for **fewer sequential expensive target steps**:

- one target forward pass verifies several proposed tokens **in parallel**;
- accept the matching prefix $\rightarrow$ many decode steps collapse into one.

It does **not** make the big model unnecessary — the target still verifies every token, so the output follows the target's distribution.

Distillation: change the model itself

D DERIVATION

Train a smaller **student** to imitate a larger **teacher**'s output distribution:

Distillation: change the model itself

D DERIVATION

Train a smaller **student** to imitate a larger **teacher**'s output distribution:

$$L_{\text{distill}} = T^2 \, D_{\text{KL}}(\text{softmax}(z_{\text{teacher}}/T) \| \, \text{softmax}(z_{\text{student}}/T))$$

Distillation: change the model itself

D DERIVATION

Train a smaller **student** to imitate a larger **teacher**'s output distribution:

$$L_{\text{distill}} = T^2 \, D_{\text{KL}}(\text{softmax}(z_{\text{teacher}}/T) \| \, \text{softmax}(z_{\text{student}}/T))$$

this changes the **model**, not only the serving procedure — soft targets carry more signal than one hard label

Interactive: knowledge distillation

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/knowledge-distillation

Sweep the temperature T and watch the teacher's **soft** label distribution flatten — see how the student learns the relative ranking of alternatives, not just the top-1 answer.

Pruning and sparsity

D DERIVATION

Remove weights (or whole structures) that contribute little:

Pruning and sparsity

D DERIVATION

Remove weights (or whole structures) that contribute little:

Unstructured — zero out individual small weights. High sparsity, but hardware rarely speeds it up.

Structured — drop whole heads / channels / layers. Less sparsity, but a **real** smaller-faster model.

Pruning and sparsity

D DERIVATION

Remove weights (or whole structures) that contribute little:

Unstructured — zero out individual small weights. High sparsity, but hardware rarely speeds it up.

Structured — drop whole heads / channels / layers. Less sparsity, but a **real** smaller-faster model.

sparsity helps only when the **hardware** can skip the removed work — structure is what buys speed

Interactive: quantization and pruning

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/quantize-prune

Dial the bit-width down from 16 to 4 and sweep a pruning threshold — watch weight memory shrink, the quantization error grow, and task quality hold then fall as you push the compression too far.

Mixture of experts: conditional compute

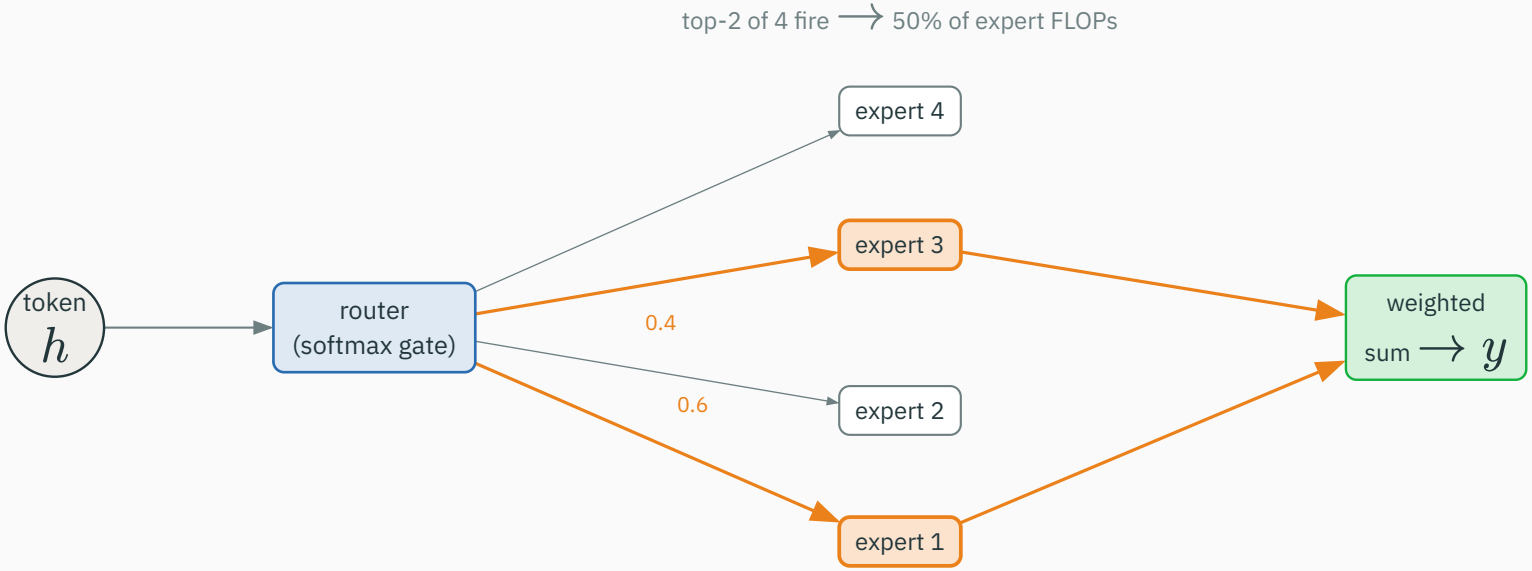
V VISUAL

Route each token to a **few** of many expert sub-networks — grow parameters without growing per-token FLOPs:

Mixture of experts: conditional compute

V VISUAL

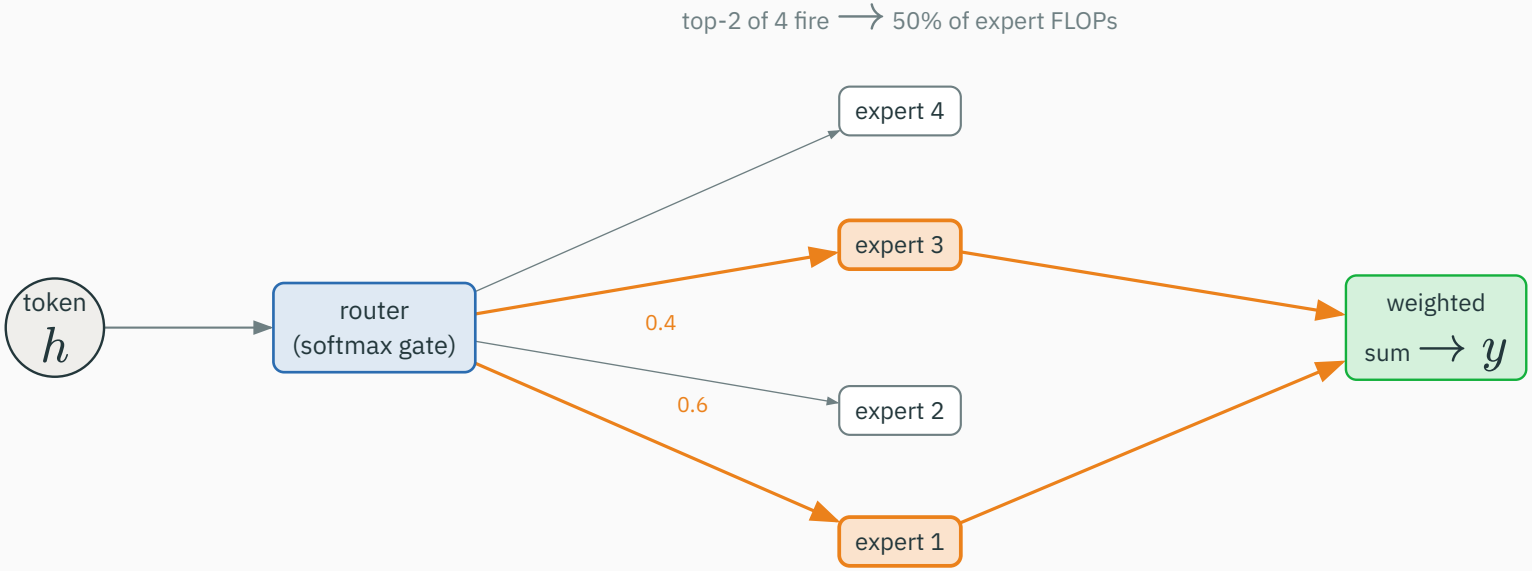
Route each token to a **few** of many expert sub-networks — grow parameters without growing per-token FLOPs:



Mixture of experts: conditional compute

V VISUAL

Route each token to a **few** of many expert sub-networks — grow parameters without growing per-token FLOPs:



a router picks the top-k experts; only those run — capacity scales, active compute does not

MoE worked example

D DERIVATION

8 experts, top-2 routing: each token fires $2/8 = 25\%$ of the expert capacity.

MoE worked example

D DERIVATION

8 experts, top-2 routing: each token fires $2/8 = 25\%$ of the expert capacity.

Mixtral 8×7B	**total** params	**active** per token
full model	≈ 47 B	≈ 13 B

MoE worked example

D DERIVATION

8 experts, top-2 routing: each token fires $2/8 = 25\%$ of the expert capacity.

Mixtral 8×7B	**total** params	**active** per token
full model	≈ 47 B	≈ 13 B

$\approx 3.6\times$ **fewer active parameters than total — capacity you do not pay for every token**

Interactive: mixture of experts

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/mixture-of-experts

Send tokens through a router and watch the top-k gate light up a small subset of experts — total parameters climb while the **active** compute per token stays fixed.

A decision table

V VISUAL

Lever	Mainly attacks	Main cost
KV cache	repeated prefix computation	VRAM
quantization	weight / cache bandwidth	approximation, engineering
efficient attention kernel	intermediate-memory traffic	specialized implementation
batching / paging	underutilization, fragmentation	scheduling complexity
speculative decoding	sequential target steps	draft model, variable gain
distillation	model size / latency	retraining, possible quality loss

Apply the systems lens across the course

Diffusion connection

D DERIVATION

Diffusion sampling repeatedly invokes the **denoiser** — the same “repeated call” cost structure:

Diffusion connection

D DERIVATION

Diffusion sampling repeatedly invokes the **denoiser** — the same “repeated call” cost structure:

- **faster samplers** — fewer denoising steps (the sequential bottleneck);
- **latent-space** diffusion — smaller tensors per call;
- **batching** and **quantization** — cheaper individual calls.

Diffusion connection

D DERIVATION

Diffusion sampling repeatedly invokes the **denoiser** — the same “repeated call” cost structure:

- **faster samplers** — fewer denoising steps (the sequential bottleneck);
- **latent-space** diffusion — smaller tensors per call;
- **batching** and **quantization** — cheaper individual calls.

every trick here reduces the cost of **repeated invocations** — LLM decode and diffusion sampling rhyme

Representation-learning connection

D DERIVATION

A good frozen encoder is also an **engineering asset**:

Representation-learning connection

D DERIVATION

A good frozen encoder is also an **engineering asset**:

- compute features **once**; attach lightweight **linear probes** downstream;
- a strong representation makes the whole downstream system **cheaper**, not only more accurate.

Representation-learning connection

D DERIVATION

A good frozen encoder is also an **engineering asset**:

- compute features **once**; attach lightweight **linear probes** downstream;
- a strong representation makes the whole downstream system **cheaper**, not only more accurate.

accuracy and efficiency are the same coin: a better representation buys both

Case exercise — pick a system

Q QUESTION

Choose one and reason about its inference profile:

Case exercise — pick a system

Q QUESTION

Choose one and reason about its inference profile:

System	Dominant pressure
chat, long prompts, many users	KV-cache memory + concurrency
mobile image classifier	device VRAM + compute
text-to-image demo, latency cap	repeated denoiser calls

Case exercise — pick a system

Q QUESTION

Choose one and reason about its inference profile:

System	Dominant pressure
chat, long prompts, many users	KV-cache memory + concurrency
mobile image classifier	device VRAM + compute
text-to-image demo, latency cap	repeated denoiser calls

choose **two** optimizations and name the expected trade-off — we return to this at the end

The one-sentence synthesis

Q QUESTION

The one-sentence synthesis

Q QUESTION

Efficient inference is mostly about avoiding repeated work and moving fewer bytes.

The one-sentence synthesis

Q QUESTION

Efficient inference is mostly about avoiding repeated work and moving fewer bytes.

KV cache, prefix reuse, FlashAttention, quantization, batching — all instances of this one idea

Closing bridge

The final lecture stacks **agents, reasoning systems, and interpretability** on top of this same foundation:

Closing bridge

The final lecture stacks **agents, reasoning systems, and interpretability** on top of this same foundation:

models + objectives + optimization + representations + **inference constraints**

Closing bridge

The final lecture stacks **agents, reasoning systems, and interpretability** on top of this same foundation:

models + objectives + optimization + representations + **inference constraints**

deployment constraints are **first-class** model-design constraints, not an afterthought

Detailed systems slides

Parameter memory comes first

D DERIVATION

For P parameters, the weights alone need:

Parameter memory comes first

D DERIVATION

For P parameters, the weights alone need:

Storage format	Approx. weight memory
fp32	$4P$ bytes
fp16 / bf16	$2P$ bytes
int8	P bytes
4-bit	$P/2$ bytes, plus metadata

Parameter memory comes first

D DERIVATION

For P parameters, the weights alone need:

Storage format	Approx. weight memory
fp32	$4P$ bytes
fp16 / bf16	$2P$ bytes
int8	P bytes
4-bit	$P/2$ bytes, plus metadata

a first-order figure — runtime also pays for KV cache, activations, and framework overhead

Weight-memory example

D DERIVATION

A **7-billion**-parameter model in fp16:

Weight-memory example

D DERIVATION

A **7-billion**-parameter model in fp16:

$$7 \times 10^9 \times 2 \text{ bytes} \approx 14 \text{ GB}$$

Weight-memory example

D DERIVATION

A **7-billion**-parameter model in fp16:

$$7 \times 10^9 \times 2 \text{ bytes} \approx 14 \text{ GB}$$

for weights **alone**.

Weight-memory example

D DERIVATION

A **7-billion**-parameter model in fp16:

$$7 \times 10^9 \times 2 \text{ bytes} \approx 14 \text{ GB}$$

for weights **alone**.

Why is a "16 GB" device still inadequate? Weights (14 GB) leave almost nothing for KV cache, activations, and overhead — long-context generation needs **far** more headroom.

Arithmetic intensity intuition

D DERIVATION

The **same weights**, two very different reuse patterns:

Arithmetic intensity intuition

D DERIVATION

The **same weights**, two very different reuse patterns:

Phase	Op shape	Reuse per fetched weight
prefill	matrix–matrix	high (many token vectors)
decode	matrix–vector	low (one token vector)

Arithmetic intensity intuition

D DERIVATION

The **same weights**, two very different reuse patterns:

Phase	Op shape	Reuse per fetched weight
prefill	matrix–matrix	high (many token vectors)
decode	matrix–vector	low (one token vector)

$$\text{less reuse per byte} \implies \text{memory bandwidth matters more}$$

Arithmetic intensity intuition

D DERIVATION

The **same weights**, two very different reuse patterns:

Phase	Op shape	Reuse per fetched weight
prefill	matrix–matrix	high (many token vectors)
decode	matrix–vector	low (one token vector)

$$\text{less reuse per byte} \implies \text{memory bandwidth matters more}$$

no hardware-specific roofline number needed — the **shape** already tells the story

Causal attention cost, briefly

D DERIVATION

A full attention layer compares token pairs:

Causal attention cost, briefly

D DERIVATION

A full attention layer compares token pairs:

$$O(T^2 d)$$

Causal attention cost, briefly

D DERIVATION

A full attention layer compares token pairs:

$$O(T^2 d)$$

KV caching removes the recomputation of **old projections** during decode. It does **not** remove the new query's need to attend across the stored history — the cache read still scales with T.

Cache append at one layer

D DERIVATION

At token t, extend the cache and attend:

Cache append at one layer

D DERIVATION

At token t, extend the cache and attend:

$$K_{1:t} = [K_{1:t-1}\,;\,k_t], \qquad V_{1:t} = [V_{1:t-1}\,;\,v_t]$$

Cache append at one layer

D DERIVATION

At token t, extend the cache and attend:

$$K_{1:t} = [K_{1:t-1}\,; k_t], \qquad V_{1:t} = [V_{1:t-1}\,; v_t]$$

Only q_t, k_t, v_t are freshly computed; the attention for q_t **reads** all cached keys and values.

Cache append at one layer

D DERIVATION

At token t, extend the cache and attend:

$$K_{1:t} = [K_{1:t-1}\,;\,k_t], \qquad V_{1:t} = [V_{1:t-1}\,;\,v_t]$$

Only q_t, k_t, v_t are freshly computed; the attention for q_t **reads** all cached keys and values.

```
k_t, v_t = project(x_t)          # new token only
K = cat(K_cache, k_t); V = cat(V_cache, v_t)
h_t = attention(q_t, K, V)       # read the whole history
K_cache, V_cache = K, V          # append for next step
```

Recompute versus cache

V VISUAL

Strategy	Compute	Memory	Decode latency
recompute full prefix	very high	low cache	poor
store KV cache	lower	grows with T	much better

Recompute versus cache

V VISUAL

Strategy	Compute	Memory	Decode latency
recompute full prefix	very high	low cache	poor
store KV cache	lower	grows with T	much better

this table is the **central systems trade-off** of the lecture — spend memory, avoid repeated compute

KV-cache calculation, step by step

D DERIVATION

Start from the formula and substitute:

KV-cache calculation, step by step

D DERIVATION

Start from the formula and substitute:

$$2 \cdot L \cdot B \cdot T \cdot n_{\text{kv}} \cdot d_h \cdot b, \quad L = 32, B = 1, T = 4096, n_{\text{kv}} = 8, d_h = 128, b = 2$$

KV-cache calculation, step by step

D DERIVATION

Start from the formula and substitute:

$$2 \cdot L \cdot B \cdot T \cdot n_{\text{kv}} \cdot d_h \cdot b, \quad L = 32, B = 1, T = 4096, n_{\text{kv}} = 8, d_h = 128, b = 2$$

$$2 \cdot 32 \cdot 4096 \cdot 8 \cdot 128 \cdot 2 = 536{,}870{,}912 \text{ bytes}$$

KV-cache calculation, step by step

D DERIVATION

Start from the formula and substitute:

$$2 \cdot L \cdot B \cdot T \cdot n_{\text{kv}} \cdot d_h \cdot b, \quad L = 32, B = 1, T = 4096, n_{\text{kv}} = 8, d_h = 128, b = 2$$

$$2 \cdot 32 \cdot 4096 \cdot 8 \cdot 128 \cdot 2 = 536{,}870{,}912 \text{ bytes}$$

$$536{,}870{,}912 \ / 1024^2 = 512 \quad \Longrightarrow \quad \approx 512 \text{ MiB} \quad \text{(before concurrency)}$$

Concurrency changes the answer

V VISUAL

If **32** requests each need this cache:

Concurrency changes the answer

V VISUAL

If **32** requests each need this cache:

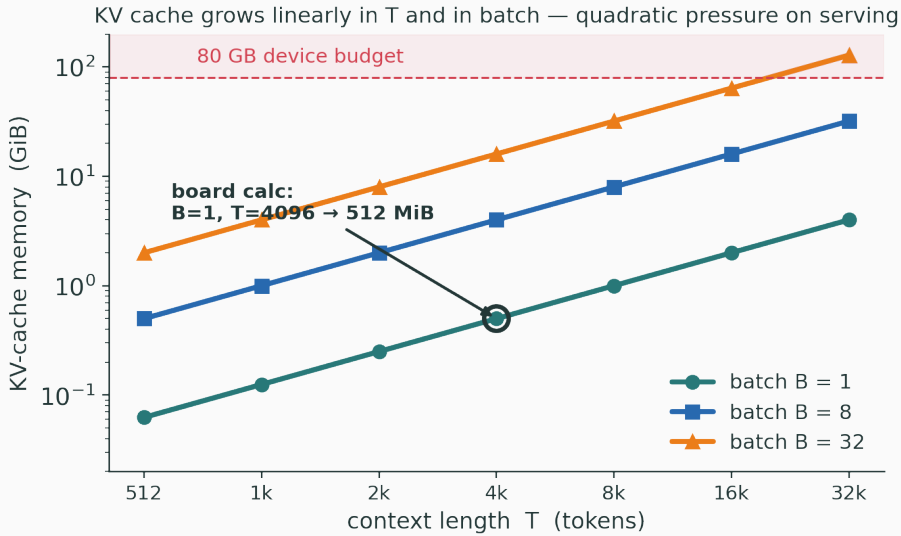
$$32 \times 512\ \text{MiB} \approx 16\ \text{GiB}$$

Concurrency changes the answer

V VISUAL

If **32** requests each need this cache:

$$32 \times 512 \text{ MiB} \approx 16 \text{ GiB}$$

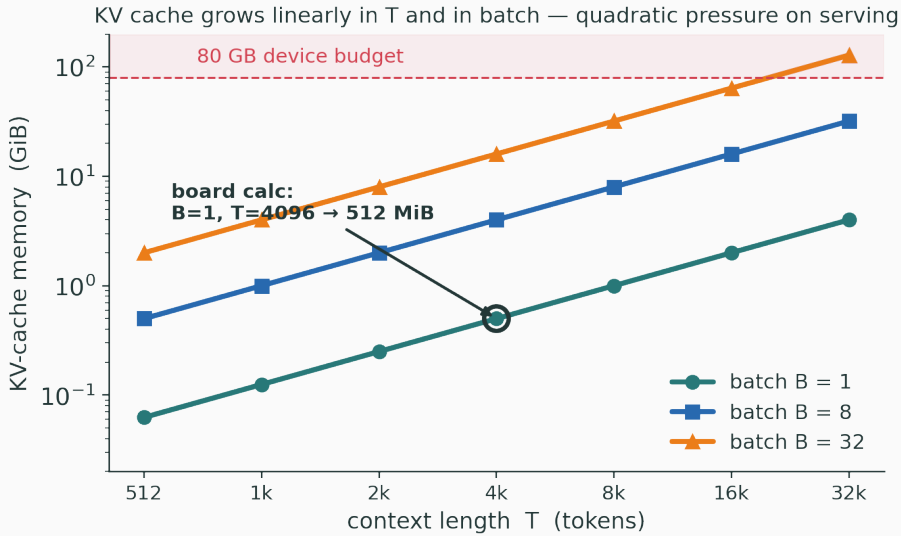


Concurrency changes the answer

V VISUAL

If **32** requests each need this cache:

$$32 \times 512 \text{ MiB} \approx 16 \text{ GiB}$$



long contexts × many users make cache capacity a **product-level** constraint, not a footnote

Grouped-query attention

D DERIVATION

Standard MHA uses one key/value head per query head. GQA uses **fewer**:

Grouped-query attention

D DERIVATION

Standard MHA uses one key/value head per query head. GQA uses **fewer**:

$$n_{\text{kv-heads}} < n_{\text{query-heads}}$$

Grouped-query attention

D DERIVATION

Standard MHA uses one key/value head per query head. GQA uses **fewer**:

$n_{\text{kv-heads}} < n_{\text{query-heads}}$

n_{kv}	cache size	flexibility
$= n_{\text{query}}$ (MHA)	$1\times$	full
group (GQA)	$n_{\text{query}}/n_{\text{kv}}$ smaller	most
$= 1$ (MQA)	smallest	least

Grouped-query attention

D DERIVATION

Standard MHA uses one key/value head per query head. GQA uses **fewer**:

$n_{\text{kv-heads}} < n_{\text{query-heads}}$

n_{kv}	cache size	flexibility
$= n_{\text{query}}$ (MHA)	$1\times$	full
group (GQA)	$n_{\text{query}}/n_{\text{kv}}$ smaller	most
$= 1$ (MQA)	smallest	least

a direct knob on the cache term $n_{\text{kv}} \cdot d_h$ — retains multiple query perspectives

Quantization is approximate representation

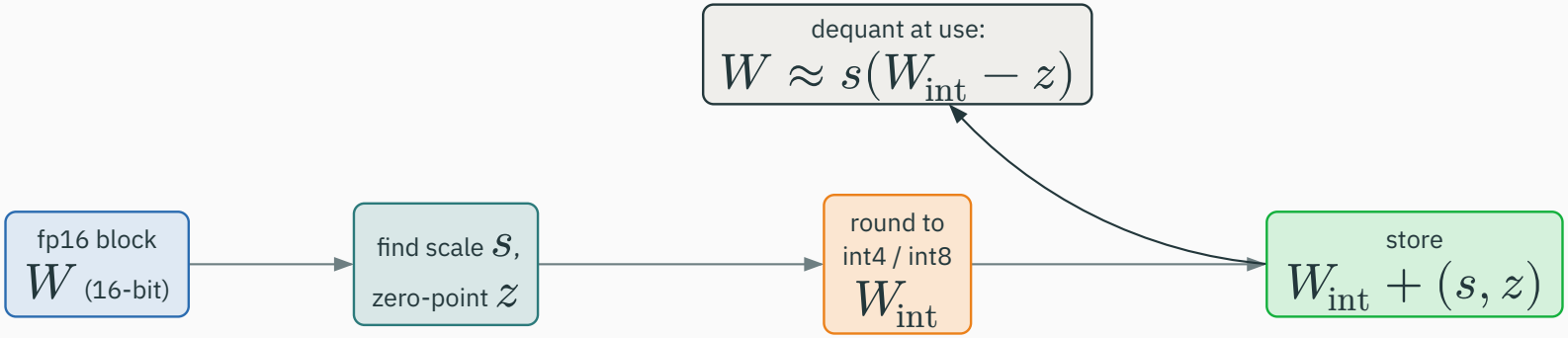
V VISUAL

Store a real-valued block approximately as a low-bit integer tensor plus metadata:

Quantization is approximate representation

V VISUAL

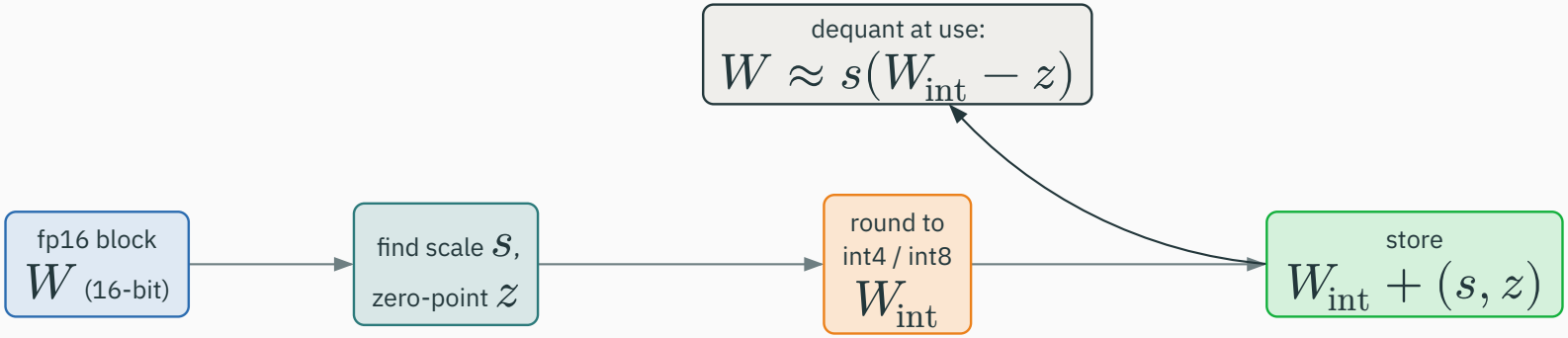
Store a real-valued block approximately as a low-bit integer tensor plus metadata:



Quantization is approximate representation

V VISUAL

Store a real-valued block approximately as a low-bit integer tensor plus metadata:



$W \approx s(W_{\text{int}} - z)$ (scale s, optional zero-point z)

Quantization design choices

D DERIVATION

Choice	Question it answers
per-tensor or per-channel scale	how much variation must be preserved?
weight-only or activations too	where is the memory / bandwidth pressure?
8-bit or 4-bit	what quality loss is acceptable?
post-training or trained-aware	can we afford to adapt the model?

Quantization design choices

D DERIVATION

Choice	Question it answers
per-tensor or per-channel scale	how much variation must be preserved?
weight-only or activations too	where is the memory / bandwidth pressure?
8-bit or 4-bit	what quality loss is acceptable?
post-training or trained-aware	can we afford to adapt the model?

finer scales preserve outliers; coarser scales save more — the central quantization tension

Quantization is not free speed

D DERIVATION

Fewer bits **can** reduce bandwidth and fit a bigger model — but the gain is conditional:

Quantization is not free speed

D DERIVATION

Fewer bits **can** reduce bandwidth and fit a bigger model — but the gain is conditional:

Speedups depend on **hardware and kernel support**. Dequantization, metadata, and irregular operations can eat much of the theoretical saving.

Quantization is not free speed

D DERIVATION

Fewer bits **can** reduce bandwidth and fit a bigger model — but the gain is conditional:

Speedups depend on **hardware and kernel support**. Dequantization, metadata, and irregular operations can eat much of the theoretical saving.

a useful systems-literacy correction: “smaller weights” $\neq$ “automatically faster”

FlashAttention as a memory algorithm

D DERIVATION

The attention **equation** is unchanged:

FlashAttention as a memory algorithm

D DERIVATION

The attention **equation** is unchanged:

$$\mathrm{softmax}\left(QK^\top/\sqrt{d}\right)V$$

FlashAttention as a memory algorithm

D DERIVATION

The attention **equation** is unchanged:

$$\mathrm{softmax}\left(QK^{\top}/\sqrt{d}\right)V$$

The kernel **tiles** the computation so the large score matrix need not be written to slow high-bandwidth memory. Same mathematics, different **data movement**.

FlashAttention as a memory algorithm

D DERIVATION

The attention **equation** is unchanged:

$$\mathrm{softmax}\left(QK^\top/\sqrt{d}\right)V$$

The kernel **tiles** the computation so the large score matrix need not be written to slow high-bandwidth memory. Same mathematics, different **data movement**.

teach it as “same answer, fewer bytes moved” — an IO-aware rewrite, not a new model

Prefix caching

D DERIVATION

Many requests share a **system prompt** or **document prefix**:

Prefix caching

D DERIVATION

Many requests share a **system prompt** or **document prefix**:

- compute that prefix’s KV state **once**;
- reuse it for every later request that shares it.

Prefix caching

D DERIVATION

Many requests share a **system prompt** or **document prefix**:

- compute that prefix's KV state **once**;
- reuse it for every later request that shares it.

> the serving analogue of dynamic programming — never redo the shared sub-computation

Continuous batching

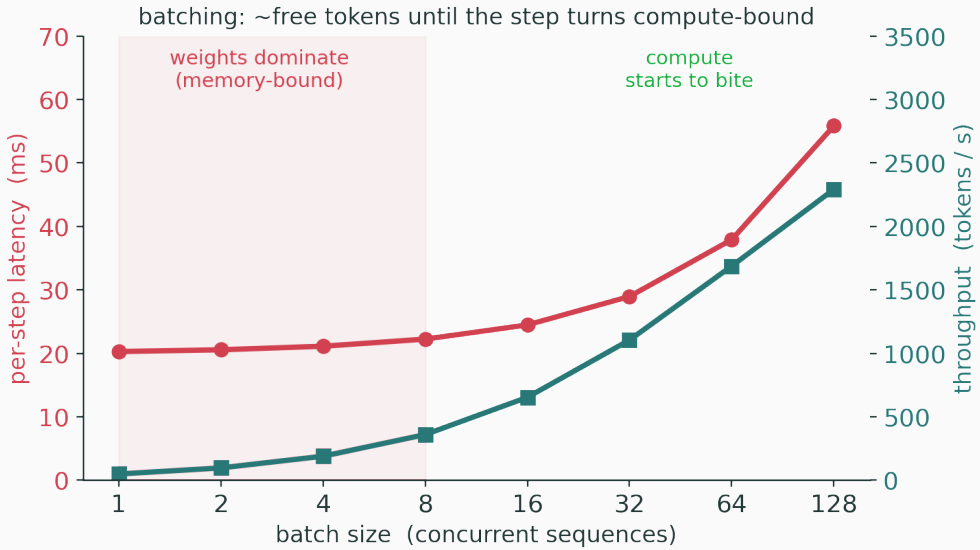
V VISUAL

Requests arrive and finish at **different** times; a scheduler inserts new ones into free capacity:

Continuous batching

V VISUAL

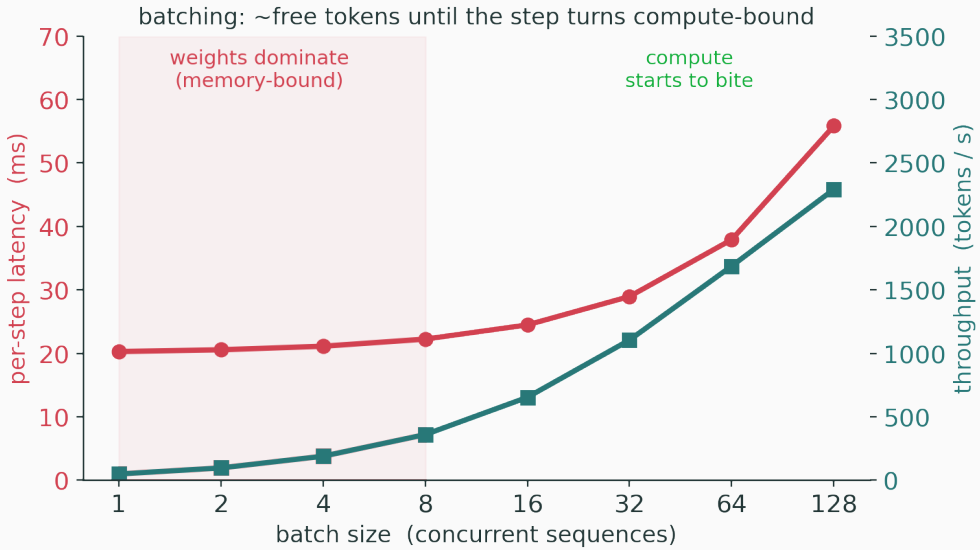
Requests arrive and finish at **different** times; a scheduler inserts new ones into free capacity:



Continuous batching

V VISUAL

Requests arrive and finish at **different** times; a scheduler inserts new ones into free capacity:



the target becomes **throughput and tail latency** — not one request's speed in isolation

Blocked KV allocation

D DERIVATION

Instead of one contiguous cache region per request, allocate **blocks** as sequences grow:

Blocked KV allocation

D DERIVATION

Instead of one contiguous cache region per request, allocate **blocks** as sequences grow:

- reduces fragmentation;
- makes it easy to **share** and **reclaim** memory across requests.

Blocked KV allocation

D DERIVATION

Instead of one contiguous cache region per request, allocate **blocks** as sequences grow:

- reduces fragmentation;
- makes it easy to **share** and **reclaim** memory across requests.

keep framework names off the slide — the resource-management idea is the lesson

Speculative decoding, expanded

D DERIVATION

Speculative decoding, expanded

D DERIVATION

1. a small **draft** model proposes several tokens;
2. the large **target** model checks them together in one pass;
3. **accept** a matching prefix;
4. **sample a correction** where the draft diverges.

Speculative decoding, expanded

D DERIVATION

1. a small **draft** model proposes several tokens;
2. the large **target** model checks them together in one pass;
3. **accept** a matching prefix;
4. **sample a correction** where the draft diverges.

```
draft_tokens = draft.generate(context, k=4)      # cheap, sequential-but-small
logits = target(context + draft_tokens)          # ONE target pass, parallel
accept = longest_matching_prefix(draft_tokens, logits)
context += accept + resample_if_rejected(logits) # target stays the authority
```

Acceptance rate determines speedup

D DERIVATION

The whole payoff hinges on how often the draft is **right**:

Acceptance rate determines speedup

D DERIVATION

The whole payoff hinges on how often the draft is **right**:

- draft guesses **poorly** → verification rejects early → little sequential work saved;
- draft predicts a **long correct prefix** → one target pass replaces many decode steps.

Acceptance rate determines speedup

D DERIVATION

The whole payoff hinges on how often the draft is **right**:

- draft guesses **poorly** $\rightarrow$ verification rejects early $\rightarrow$ little sequential work saved;
- draft predicts a **long correct prefix** $\rightarrow$ one target pass replaces many decode steps.

$$\text{draft quality} + \text{cheapness} \implies \text{useful speculation}$$

Acceptance rate determines speedup

D DERIVATION

The whole payoff hinges on how often the draft is **right**:

- draft guesses **poorly** $\rightarrow$ verification rejects early $\rightarrow$ little sequential work saved;
- draft predicts a **long correct prefix** $\rightarrow$ one target pass replaces many decode steps.

$$\text{draft quality} + \text{cheapness} \implies \text{useful speculation}$$

a well-matched, fast draft is the entire game — a slow or inaccurate draft can even **lose**

Distillation changes the model

D DERIVATION

Train the student against the teacher’s temperature-softened distribution:

Distillation changes the model

D DERIVATION

Train the student against the teacher's temperature-softened distribution:

$$L_{\text{distill}} = T^2 \, D_{\text{KL}}(\text{softmax}(z_{\text{teacher}}/T) \| \, \text{softmax}(z_{\text{student}}/T))$$

Distillation changes the model

D DERIVATION

Train the student against the teacher's temperature-softened distribution:

$$L_{\text{distill}} = T^2 \, D_{\text{KL}}(\text{softmax}(z_{\text{teacher}}/T) \| \, \text{softmax}(z_{\text{student}}/T))$$

Soft targets convey **relative alternatives** — "mostly cat, a little dog, never car" — far richer than a single hard label. The T^2 factor keeps gradient magnitudes stable as T rises.

Compression versus serving optimization

V VISUAL

Approach	Retraining?	Main effect
quantization	often no	fewer bits / lower bandwidth
KV cache	no	avoid repeated prefix work
speculative decoding	no target retraining	fewer sequential target steps
distillation	yes	smaller / faster model

Compression versus serving optimization

V VISUAL

Approach	Retraining?	Main effect
quantization	often no	fewer bits / lower bandwidth
KV cache	no	avoid repeated prefix work
speculative decoding	no target retraining	fewer sequential target steps
distillation	yes	smaller / faster model

some levers **change the model**; others only **change how it is served** — know which you are pulling

Case exercise: choose levers

Q QUESTION

Scenario	Reasonable pair (name the trade-off)
A · long-doc chat, many users	GQA + paged KV · quality-neutral, scheduling cost
B · offline phone classifier	distillation + int8 · retrain effort, small quality dip
C · image generator, 2 s budget	fewer sampler steps + quantization · slight fidelity loss

Case exercise: choose levers

Q QUESTION

Scenario	Reasonable pair (name the trade-off)
A · long-doc chat, many users	GQA + paged KV · quality-neutral, scheduling cost
B · offline phone classifier	distillation + int8 · retrain effort, small quality dip
C · image generator, 2 s budget	fewer sampler steps + quantization · slight fidelity loss

there is no free lever — every choice pays somewhere in quality, cost, or engineering

Common misconceptions

V VISUAL

Misconception	Correction
“KV cache makes attention constant-time.”	avoids old projections; cache **reads** still grow with context
“Quantization always improves every metric.”	trades precision; relies on efficient kernels
“FlashAttention changes Transformer math.”	same attention, different **memory movement**
“A draft model replaces the target model.”	the target still **verifies** every output token

Retrieval questions

Q QUESTION

Retrieval questions

Q QUESTION

1. Why do prefill and decode stress hardware **differently**?

Retrieval questions

Q QUESTION

1. Why do prefill and decode stress hardware **differently**?
2. What **exactly** is stored in a KV cache?

Retrieval questions

Q QUESTION

1. Why do prefill and decode stress hardware **differently**?
2. What **exactly** is stored in a KV cache?
3. Which variables make cache memory **grow**?

Retrieval questions

Q QUESTION

1. Why do prefill and decode stress hardware **differently**?
2. What **exactly** is stored in a KV cache?
3. Which variables make cache memory **grow**?
4. Why can 4-bit weights help a **bandwidth-bound** workload?

Retrieval questions

Q QUESTION

1. Why do prefill and decode stress hardware **differently**?
2. What **exactly** is stored in a KV cache?
3. Which variables make cache memory **grow**?
4. Why can 4-bit weights help a **bandwidth-bound** workload?
5. Which optimization changes the **underlying model** rather than only serving it?

Memory-budget worksheet

D DERIVATION

Before loading a model, estimate **all four** terms:

Memory-budget worksheet

D DERIVATION

Before loading a model, estimate **all four** terms:

$$\underbrace{\text{weights}}_{2P} + \underbrace{\text{KV cache}}_{\text{grows with } BT} + \text{temporary activations} + \text{runtime overhead}$$

Memory-budget worksheet

D DERIVATION

Before loading a model, estimate **all four** terms:

$$\underbrace{\text{weights}}_{2\text{P}} + \underbrace{\text{KV cache}}_{\text{grows with } BT} + \text{temporary activations} + \text{runtime overhead}$$

For a **serving** system, multiply the KV-cache term by **expected concurrency** — not by one idealized request. This is where “it fit in the demo” becomes “it OOMs in production”.

Context-window policy is a systems policy

D DERIVATION

A larger maximum context improves user experience but **raises cache memory and attention work**:

Context-window policy is a systems policy

D DERIVATION

A larger maximum context improves user experience but **raises cache memory and attention work**:

- truncation · retrieval · summarization · sliding windows;

Context-window policy is a systems policy

D DERIVATION

A larger maximum context improves user experience but **raises cache memory and attention work**:

- truncation · retrieval · summarization · sliding windows;
- these are **product** choices shaped by the **same** compute constraints.

Context-window policy is a systems policy

D DERIVATION

A larger maximum context improves user experience but **raises cache memory and attention work**:

- truncation · retrieval · summarization · sliding windows;
- these are **product** choices shaped by the **same** compute constraints.

this is where inference engineering meets retrieval and agent design

Model parallelism, one placement slide

D DERIVATION

When one device cannot hold the model, split parameters or layers across devices:

Model parallelism, one placement slide

D DERIVATION

When one device cannot hold the model, split parameters or layers across devices:

$$\text{more devices} \quad \neq \quad \text{linear speedup}$$

Model parallelism, one placement slide

D DERIVATION

When one device cannot hold the model, split parameters or layers across devices:

$$\text{more devices} \quad \neq \quad \text{linear speedup}$$

Distributing the model makes larger models **feasible**, but adds **communication overhead**. Pipeline- and tensor-parallel algorithms stay outside this core lecture.

Compilation and fused operations

D DERIVATION

Launching many tiny operations separately wastes time:

Compilation and fused operations

D DERIVATION

Launching many tiny operations separately wastes time:

- **compilers** and **fused kernels** combine operations;
- fewer launches, fewer intermediate memory reads/writes.

Compilation and fused operations

D DERIVATION

Launching many tiny operations separately wastes time:

- **compilers** and **fused kernels** combine operations;
- fewer launches, fewer intermediate memory reads/writes.

another instance of the core idea: **avoid unnecessary movement and coordination**

Quantization evaluation protocol

D DERIVATION

Never judge a compressed model on average accuracy alone — test:

Quantization evaluation protocol

D DERIVATION

Never judge a compressed model on average accuracy alone — test:

- representative **task quality**;
- **long-context** behavior;
- **rare / error-sensitive** cases;
- **latency and memory** on the **actual target hardware**.

Quantization evaluation protocol

D DERIVATION

Never judge a compressed model on average accuracy alone — test:

- representative **task quality**;
- **long-context** behavior;
- **rare / error-sensitive** cases;
- **latency and memory** on the **actual target hardware**.

Average benchmark accuracy can hide a damaging quantization failure that only surfaces on the tail.

Cost per useful outcome

Q QUESTION

The relevant quantity is **not** merely tokens per second:

Cost per useful outcome

Q QUESTION

The relevant quantity is **not** merely tokens per second:

$$\frac{\text{cost}}{\text{useful, correct, safe outcome}}$$

Cost per useful outcome

Q QUESTION

The relevant quantity is **not** merely tokens per second:

$$\frac{\text{cost}}{\text{useful, correct, safe outcome}}$$

a slower model can win if it avoids retries; a fast model is wasteful if its outputs need repeated correction

Energy and hardware awareness

D DERIVATION

Moving data across the memory hierarchy costs **time and energy**:

Energy and hardware awareness

D DERIVATION

Moving data across the memory hierarchy costs **time and energy**:

- efficient inference can lower operational cost **and** environmental impact;
- but **demand growth** can offset per-query savings.

Energy and hardware awareness

D DERIVATION

Moving data across the memory hierarchy costs **time and energy**:

- efficient inference can lower operational cost **and** environmental impact;
- but **demand growth** can offset per-query savings.

Avoid the simplistic “efficient therefore green” claim — count the **total** workload, not the per-query number.

Serving versus research benchmark

V VISUAL

Benchmark focus	Production focus
one model, one batch, peak throughput	variable requests, tail latency, reliability
fixed prompt length	long and changing contexts
clean hardware state	memory fragmentation and concurrency

Serving versus research benchmark

V VISUAL

Benchmark focus	Production focus
one model, one batch, peak throughput	variable requests, tail latency, reliability
fixed prompt length	long and changing contexts
clean hardware state	memory fragmentation and concurrency

question any performance claim that omits its **workload assumptions**

Retrieval questions, extended

Q QUESTION

Retrieval questions, extended

Q QUESTION

1. Why can a model **fit** in memory yet **fail** under concurrent long-context use?

Retrieval questions, extended

Q QUESTION

1. Why can a model **fit** in memory yet **fail** under concurrent long-context use?
2. What is the difference between quantizing **weights** and quantizing the **KV cache**?

Retrieval questions, extended

Q QUESTION

1. Why can a model **fit** in memory yet **fail** under concurrent long-context use?
2. What is the difference between quantizing **weights** and quantizing the **KV cache**?
3. Which optimization changes **arithmetic scheduling** but not model **output**?

Retrieval questions, extended

Q QUESTION

1. Why can a model **fit** in memory yet **fail** under concurrent long-context use?
2. What is the difference between quantizing **weights** and quantizing the **KV cache**?
3. Which optimization changes **arithmetic scheduling** but not model **output**?
4. Why does serving **require** workload assumptions?

Final bridge

models + objectives + representations + **inference systems**

Final bridge

models + objectives + representations + **inference systems**

are all needed to understand a frontier deployment

Final bridge

models + objectives + representations + **inference systems**

are all needed to understand a frontier deployment

The final lecture uses this checklist to demystify **agents, reasoning systems, and interpretability** — they are built on the foundation this course has assembled, inference constraints included.

Summary

Student takeaways

V VISUAL

Student takeaways

V VISUAL

1. Prompt processing (**prefill**) and token-by-token **decode** have different performance profiles.

Student takeaways

V VISUAL

1. Prompt processing (**prefill**) and token-by-token **decode** have different performance profiles.
2. KV caching trades **memory** for **avoided repeated computation**.

Student takeaways

V VISUAL

1. Prompt processing (**prefill**) and token-by-token **decode** have different performance profiles.
2. KV caching trades **memory** for **avoided repeated computation**.
3. Many inference optimizations reduce **memory traffic** rather than changing neural-network mathematics.

Student takeaways

V VISUAL

1. Prompt processing (**prefill**) and token-by-token **decode** have different performance profiles.
2. KV caching trades **memory** for **avoided repeated computation**.
3. Many inference optimizations reduce **memory traffic** rather than changing neural-network mathematics.
4. Deployment constraints are **first-class** model-design constraints.

The levers, one last look

V VISUAL

Lever	Changes	Pay with
KV cache · GQA	serving	VRAM
quantization	representation	precision + kernels
FlashAttention	data movement	specialized kernel
batching · paging	scheduling	complexity
speculative decoding	scheduling	a draft model
distillation · pruning · MoE	the model	training effort

At inference, the model is often **waiting on memory**, not arithmetic.

Efficient inference = avoid **repeated work** + move **fewer bytes**.

Next: **Frontier Topics & Course Synthesis** — agents, reasoning, interpretability.