Efficiency in LLMs

Hardware, Serving, and Compression —
a Practitioner's Tour of Fast Inference

Alex Smola

Boson AI

Columbia Machine Learning Summer School · 2026

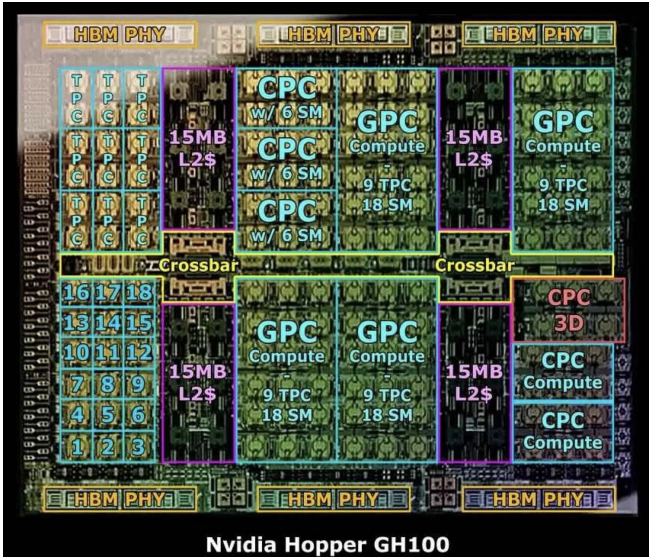
All numbers verified June 2026. They're likely wrong by December.

Part 1

Overview

why fast inference is a memory problem

H100 Die Shot



Nvidia Hopper GH100

An H100 Is Mostly Waiting for Memory

989

TFLOP/s (BF16)

3.35

TB/s memory bandwidth

$\Rightarrow$ 295 FLOP / byte

The imbalance that runs this tutorial

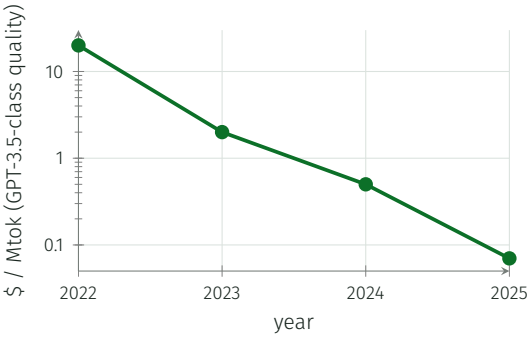
Keep the cores busy: ∼300 **FLOPs per byte fetched**.

One token reuses each weight *once* $\Rightarrow$ the chip starves.

Decode is a memory-traffic problem, not a math problem. Almost every trick here just *moves fewer bytes*.

H100 SXM5 dense BF16 / HBM3 (NVIDIA H100 whitepaper); ridge point = $989\,\mathrm{TF}/3.35\,\mathrm{TB/s}$.

Tokens Got $\sim 10\times$ Cheaper per Year



Straight line on a log axis = constant factor per year: $\sim 10\times$ **cheaper / year** at fixed capability.

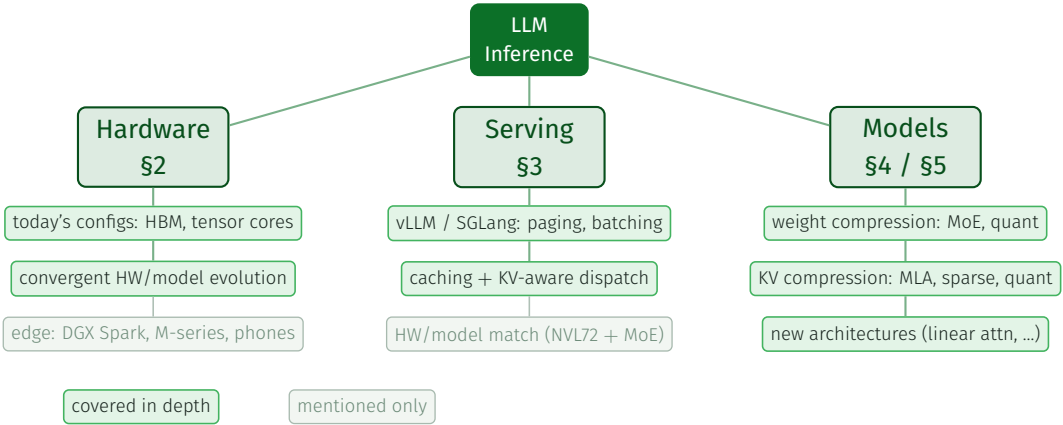
Why it matters

Most of the win comes from serving efficiency.

At scale, **inference spend dwarfs training**: train once, serve billions of times.

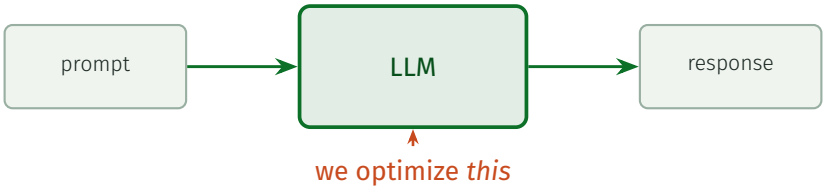
Trend "LLMflation" (a16z, 2024); illustrative GPT-3.5-class API prices. Order-of-magnitude, not exact.

The Map: What Makes Inference Fast



The tutorial's table of contents. We revisit this exact map at the end.

We Make the Box Cheaper — Not the Prompt Smarter



Out of scope
better prompt (prompt engineering), better answer (post-training, RAG). Different course.

In scope
same input, same output — fewer joules and milliseconds. Prompt *caching* (a serving trick) lands in §3.

Fixed model, fixed I/O: we change only the cost of evaluating the box.

What We Skip (and Where to Read Instead)

- **Training efficiency** — 3D parallelism, ZeRO, FP8 at cluster scale.

  Ultra-Scale Playbook

- **Audio / real-time streaming** — full-duplex speech, 160 ms voice agents.

  Kyutai Moshi [arXiv:2410.00037]

- **Video token costs** — ∼300 tok/s of frames; a KV-cache firehose (§5).

  Gemini video

- **Multi-node sharding** — tensor / pipeline / expert parallelism across racks.

  `NVIDIA/Megatron-LM`

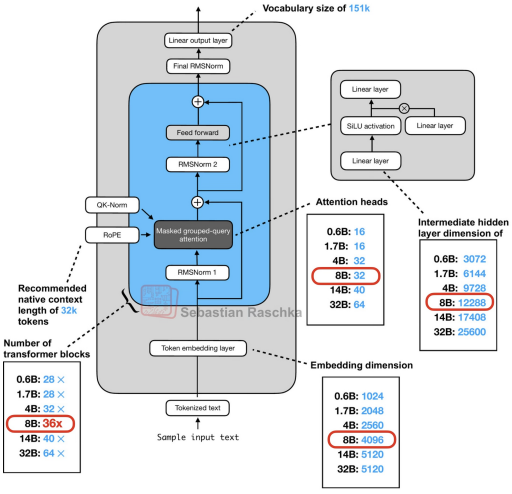
- **Speculative decoding** — draft-and-verify; orthogonal 2–5× win (§3).

  `SafeAILab/EAGLE`

Focus: single-node, autoregressive, text — where the core ideas are clearest. Everything above is a worthy rabbit hole.

Curated pointers; full resource list in §6.

LLM Blueprint: A Decoder Block, Stacked L Times



Two matmul factories

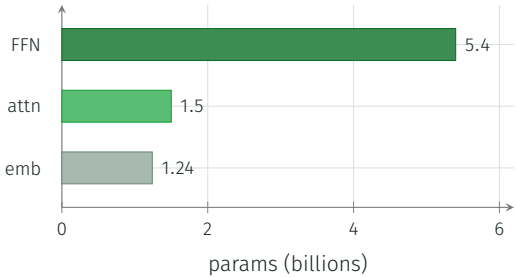
Each block = Attention (mixes tokens) + FFN (mixes features). Both big dense matmuls.

The whole tutorial: feed these two matmuls more cheaply — or skip parts of them.

Qwen3-8B layout; GQA (grouped-query attention) is already the standard — our first KV saving.

Running Example: Qwen3-8B (Where the Bytes Live)

vocab	151,936
d (hidden)	4096
layers L	36
Q / KV heads	32 / 8
head dim	128
FFN inter.	12,288
total params	**8.19 B**
weights (BF16)	∼16.4 **GB**

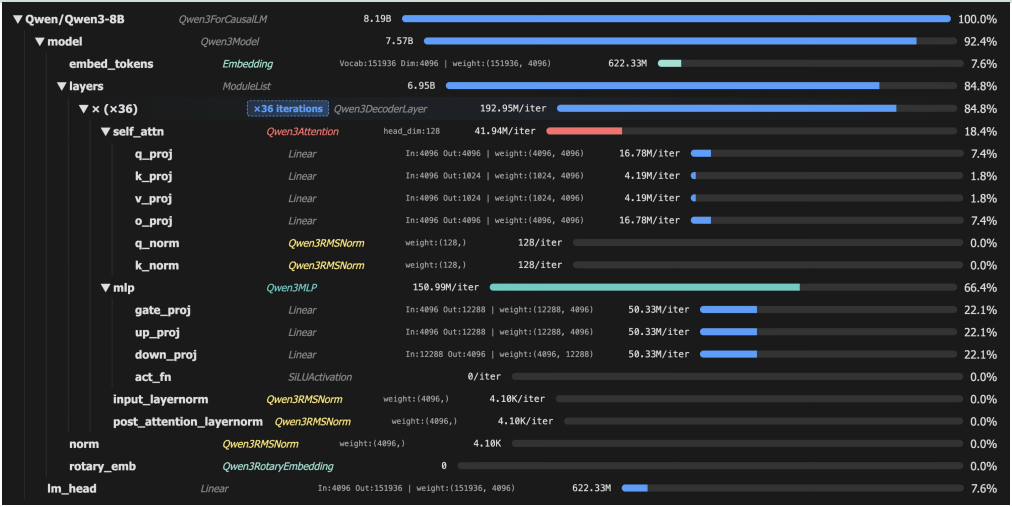


Budget: $1.24 + 1.5 + 5.4 \approx 8.2$ **B**

FFN is ∼65% of the weights — first place to hunt for savings (§4: MoE touches only a slice).

Qwen3-8B config (canonical numbers); embeddings untied ⇒ 2 × 0.62 B.

Every Shape, Every Byte: Qwen3-8B in Full



http://weavers.neocities.org/architecture-encyclopedia/Qwen_Qwen3_8B_architecture

The KV Cache: Compute Each Token's K,V Once

KV cache: K and V of every past token

Causal mask: token t attends to all earlier tokens. Their K, V never change $\Rightarrow$ compute once, **cache**, reuse forever. Each step appends *one row*.

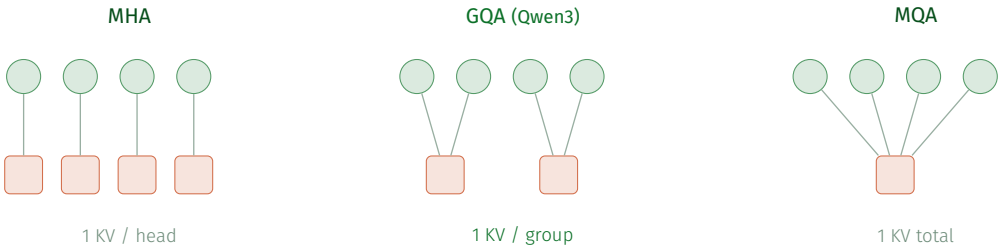
The price of remembering

$$\frac{\text{KV}}{\text{token}} = 2 \cdot L \cdot n_{\text{kv}} \cdot d_{\text{head}} \cdot 2\,\text{B}$$
$$= 2 \cdot 36 \cdot 8 \cdot 128 \cdot 2 = 147\ \text{KB}$$

At 40k tokens it is $\sim$6 **GB** per stream.

Qwen3-8B, BF16 KV. The KV cache is the memory cost that grows with the conversation (§5).

GQA: The KV Compression Everyone Already Ships

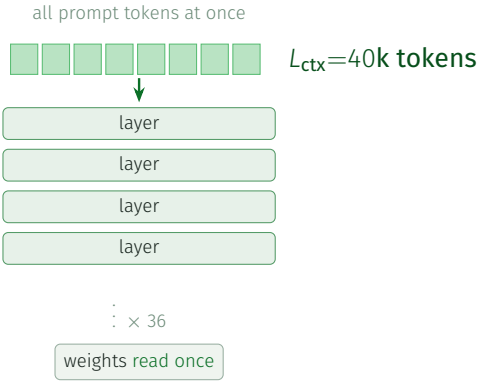


Share K,V across query heads

Qwen3-8B: 32 Q heads share just 8 KV heads $\Rightarrow 4\times$ **smaller KV cache**, free. Structural KV compression, already standard — and the template for the deeper §5 tricks (MLA, sparse, quantized KV).

MHA Vaswani et al. 2017 [arXiv:1706.03762]; MQA Shazeer 2019 [arXiv:1911.02150]; GQA Ainslie et al. 2023 [arXiv:2305.13245].

Phase 1 — Prefill: A Wide Block of Tokens, Read Weights Once



FLOPs $\approx 2 \cdot N_{\text{params}} \cdot L_{\text{ctx}}$ (linear) $+ O(L_{\text{ctx}}^2)$ (attention).
Worked at 40k:

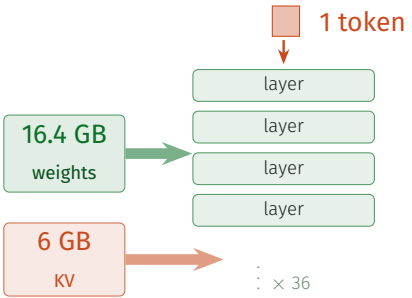
FFN (3 linear $\times L \times d \times d_{\text{ff}}$)	222 TF
Attn projections (Q,K,V,O)	60 TF
Attention scores ($O(L_{\text{ctx}}^2)$)	472 TF
total prefill compute	∼750 TF

Compute-bound

Whole prompt streams the weights *once*, but does ∼750 trillion FLOPs. Tons of math per byte ⇒ cores stay busy.

Qwen3-8B, L_{ctx}=40k, batch 1. FLOP breakdown verified; "TF" = 10^{12} FLOPs.

Phase 2 — Decode: One Token, but Stream *Everything*



FLOPs $\approx 2 \cdot N_{\text{params}} \approx$ 16 GFLOP per token (plus an $O(L_{\text{ctx}})$ attention read). But to do it, you must *move*:

all weights (BF16)	16.4 GB
all KV (40k ctx)	6 GB
bytes touched / token	∼22 GB

Bandwidth-bound

22 GB of traffic for 16 GFLOP of math — $<$ 1 FLOP/byte.

Lost the tokens that kept the cores fed. The chip now *waits on memory*.

Qwen3-8B, 40k ctx, batch 1. Weights 16.4 GB; KV 6 GB $\Rightarrow$ ∼22 GB swept *every* token.

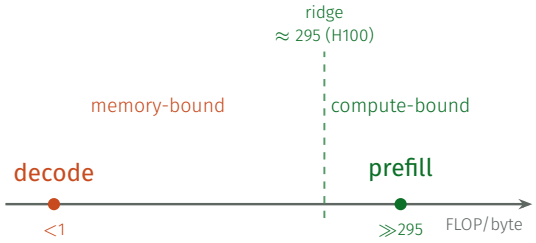
Arithmetic Intensity: FLOPs per Byte Decides Everything

$$\text{intensity} = \frac{\text{FLOPs done}}{\text{bytes moved}}$$

A matmul $(n\times k)(k\times m)$: $2nkm$ FLOPs, $\sim(nk+km)$ values read.

Intensity grows with the shared dimension — how many tokens reuse each loaded weight.

- Prefill: each weight serves L_{ctx} tokens $\Rightarrow$ intensity $\sim L_{\text{ctx}}$.
- Decode: each weight serves 1 token $\Rightarrow$ intensity ~ 1.

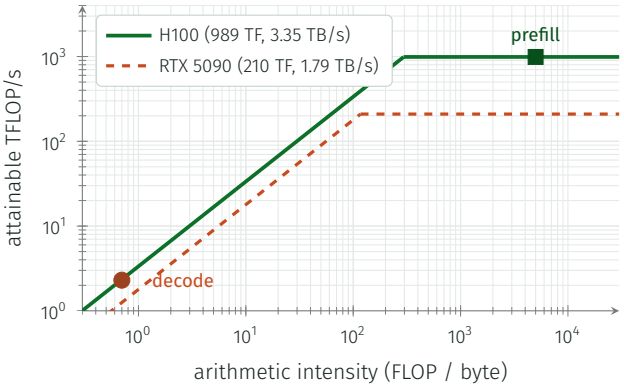


Same model, two regimes

Left of the ridge: bandwidth-limited. Right: compute-limited. Prefill and decode sit on *opposite ends*.

Ridge = peak FLOP/s ÷ bandwidth; $989\,\text{TF}/3.35\,\text{TB/s} \approx 295$ (H100 BF16).

The Roofline: Where Prefill and Decode Land



Read the picture

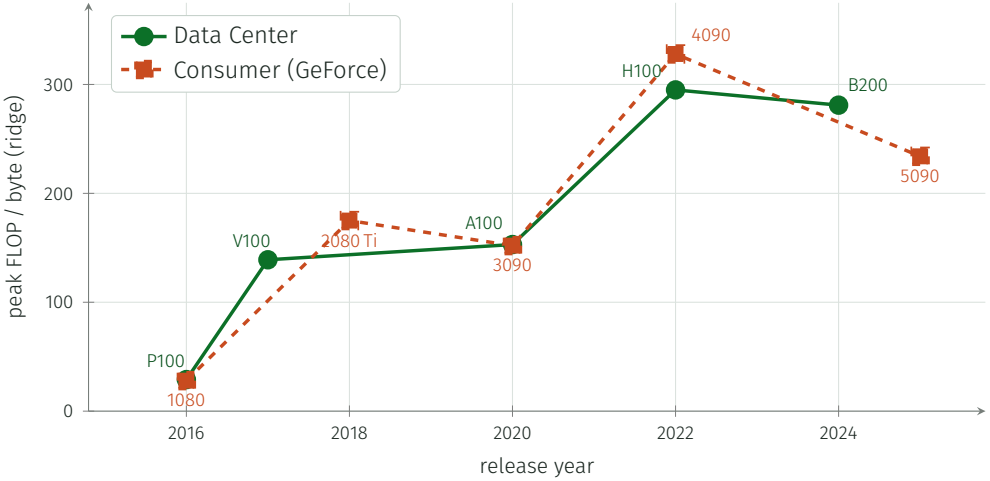
Sloped part = bandwidth wall. Flat part = compute ceiling.

Decode sits far down the slope: $< 1\%$ of peak FLOPs. Prefill hits the ceiling.

5090's wall and ceiling are both lower — but the *shape* is identical. Same physics, smaller numbers.

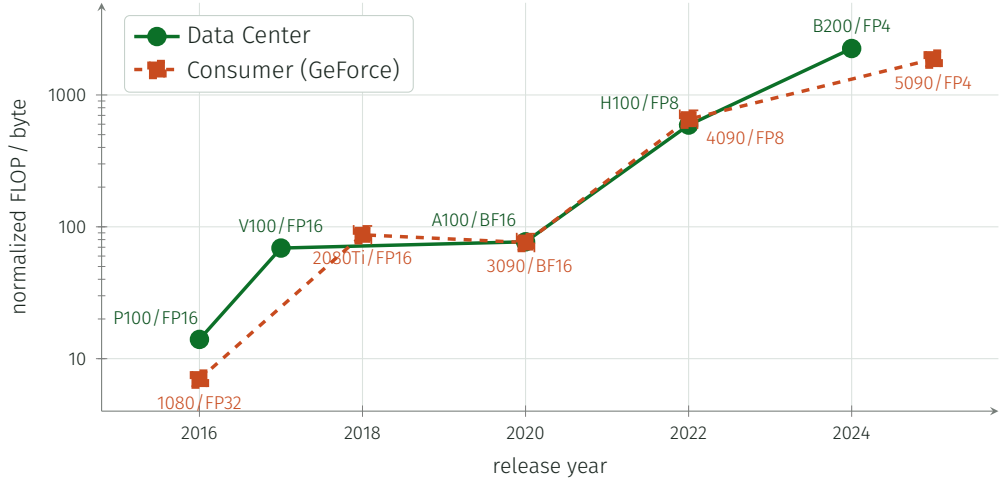
Ridge: H100 295, RTX 5090 $\approx$ 117 FLOP/byte. Dense BF16 peaks; H100 3.35 / 5090 1.79 TB/s.

At 16-Bit, the Ridge Keeps Climbing



Peak *dense* FP16/BF16 (FP16-accumulate) ÷ memory bandwidth.
NVIDIA whitepapers. GTX 1080 has no FP16 tensor core ⇒ FP32.

Per Operand Byte, the Gap Explodes ($> 1000\times$)



(peak dense FLOP/s $\div$ s_{operand}) $\div$ bandwidth, native precision per generation; the NVFP4 term uses $s = 0.5$ B.

Back of the Envelope: Prefill Buys FLOPs, Decode Buys Bytes/s

Qwen3-8B, 40k ctx, batch 1	H100	RTX 5090
peak compute (BF16)	989 TF/s	210 TF/s
bandwidth	3.35 TB/s	1.79 TB/s
prefill $= 750\,\mathrm{TF}\div$ compute	$\sim$0.76 s	$\sim$3.6 s
decode $= 22\,\mathrm{GB}\div$ bandwidth	6.6 ms/tok	12.3 ms/tok
$\Rightarrow$ decode upper bound	$\sim$150 tok/s	$\sim$81 tok/s

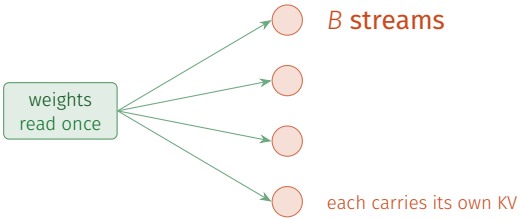
Redo it: $22\,\mathrm{GB}/3350\,\mathrm{GB/s} = 6.6\,\mathrm{ms}$, so $1/6.6\,\mathrm{ms} \approx 150$ tok/s. 5090 has $1.9\times$ less bandwidth $\Rightarrow 1.9\times$ slower decode.

The one-line summary

Prefill speed = FLOP/s. Decode speed = bytes/s. Two phases, two bottlenecks, one model.

Theoretical ceilings (zero overhead). Real decode is $\sim$50–80% of bound; prefill ignores the $O(L^2)$ attention read.

Batching Amortizes Weights — but Not KV



B streams share one weight read ⇒ intensity $\times B$. Decode marches *up* the roofline toward the ceiling — how servers hit high throughput.

But KV grows $\times B$: 32 streams $\times$6 GB = 192 GB. Now KV, not weights, owns the bus and the capacity.

The fundamental serving tension

Bigger batch = better weight amortization *but* more KV traffic and footprint. Half of §3 and most of §5 manage exactly this trade.

Weights amortize across a batch; KV does not. This is why KV compression (§5) is the long-context unlock.

Consequence 1 — The Two Phases Want Different Machines

Prefill

hungry for FLOP/s
⇒ compute monsters
(big tensor cores, FP8/FP4)

Decode

hungry for bytes/s + capacity
⇒ fat, fast memory
(HBM, lots of it)

Idea: stop running them on the same box

One wants compute, the other bandwidth ⇒ split them: a prefill pool and a decode pool, shipping KV between them. **Disaggregated serving** — built in §3.

Preview of disaggregation: DistServe (Zhong et al. 2024) [arXiv:2401.09670]; Mooncake (Qin et al. 2024) [arXiv:2407.00079].

Consequence 2 — The Local-Inference Loophole

Batch 1 decode wants bandwidth and capacity, not FLOPs. Unified memory wins.

device	memory	bandwidth	70B Q4 (40 GB) fits?
RTX 5090	32 GB	**1.79 TB/s**	no
DGX Spark (GB10)	128 GB	273 GB/s	yes
Strix Halo 395	128 GB	256 GB/s	yes
M4 Max	128 GB	546 GB/s	yes
M3 Ultra	512 GB	819 GB/s	yes (easily)

Capacity + bandwidth > raw FLOPs (for chat)

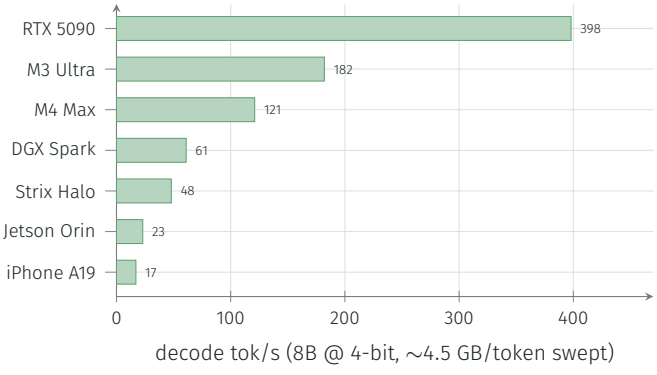
70B-Q4 (40 GB) won't even *fit* on a 5090. M3 Ultra runs it — despite half the bandwidth — because it *fits*.

The catch: prefill still hurts

DGX Spark, Llama-3.1-8B: 7,991 tok/s prefill vs 20.5 tok/s decode — a ∼390× **gap** on the *same* chip.

Edge specs from r3; DGX Spark 8B FP8 batch-1 measured (LMSYS / SGLang review, Oct 2025).

What That Buys: 8B-Q4 Decode, Batch 1



Bound $= \dfrac{\text{bandwidth}}{4.5\,\text{GB}}$. H100 (744) is off-chart right — but costs 30× a Mac mini, and you don't own it.

Reality check

Measured: ∼50–80% of bound (KV reads, dequant, attention). The *ranking* holds: bandwidth = decode speed.

Theoretical peaks = BW/4.5 GB (r3); H100 shown for scale. Realized ≈ 0.5–0.8×.

The Fixes, Part 1 — Throw Money at It (Hardware, §2)

- **More memory** — fit the model and its KV without offloading.
- **Faster memory** — HBM stacking; the only number decode cares about.
- **More transistors** — bigger dies, multi-die packages.
- **Specialized ops** — tensor cores, FP8/FP4 units for cheap matmul.

The cruel scaling laws

Per hardware generation:

compute $\sim 4\times$ ($2\times$ arch $\times$ $2\times$ format)
bandwidth $\sim 2\times$
capacity $< 1.4\times$

Why this won't save you

Compute races ahead; bandwidth and capacity lag. Intensity pressure *rises* every generation. Money buys time, not escape.

Generational rules of thumb from r1/r2 (A100→H100→B200→Rubin). DRAM squeeze 2025–26 worsens capacity.

The Fixes, Part 2 — Be Clever (the Rest of This Course)

problem	fix	where
phases fight over one box	specialized HW per phase, disaggregate	§3
weights too big to stream	MoE (fewer active) + quantization	§4
KV too big to hold / read	MLA, sparse attention, KV quant	§5
too many tokens of content	better tokenizers / patching (link only)	—

Every technique is the same move

All of them move fewer bytes per token. Hardware (§2) raises the ceiling; the algorithms (§3–§5) lower how far below it you live.

The Tour: Five Parts

	part	one line
2	Hardware	the physics of fast: HBM, tensor cores, formats
3	Serving	many users, one copy of the weights
4	Weight compression	same brain, fewer bytes (MoE + quant)
5	KV compression	how to afford 1M tokens of context
6	Wrap	the map revisited, numbers to remember

Hold onto one number

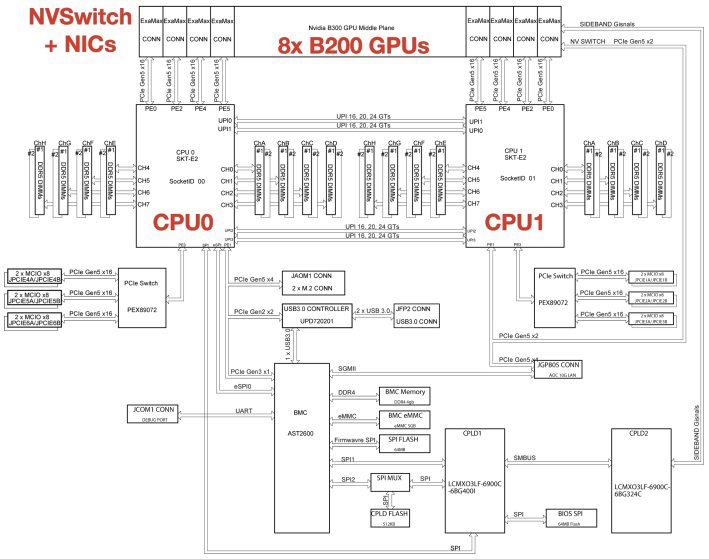
295 FLOP/byte. An H100 does 295 operations in the time it fetches one byte. Everything that follows makes that ratio survivable.

Part 2

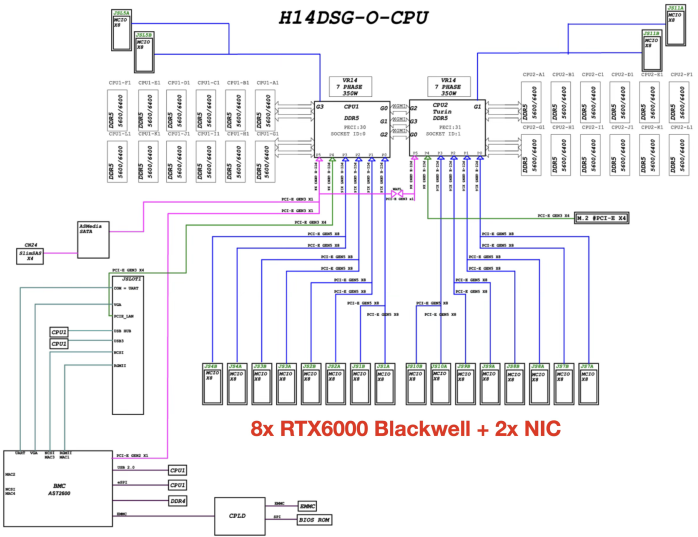
Hardware

the physics of fast

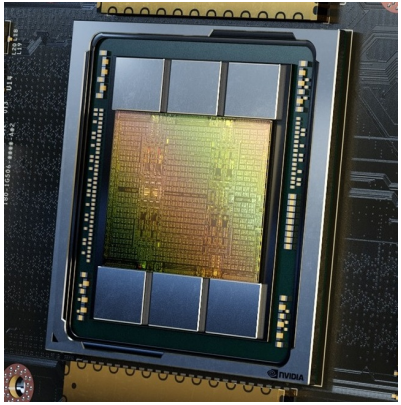
B200 Server


NVSwitch
+ NICs
ExaMax CONN
Nvdia B300 GPU Middle Plane
8x B200 GPUs
SIDEBAND Signals
NV SWITCH
PCIe Gen5 x2
PCIe Gen5 x16
UPI 16, 20, 24 GTs
CPU 0
SKT-E2
SocketID 00
CPU1
SKT-E2
SocketID 01
CPU0
CPU1
DDR5 DIMM
PCIe Switch
PEX89072
2 x MCIO x8
JPCIE4A/JPCIE4B
2 x MCIO x8
JPCIE5A/JPCIE5B
2 x MCIO x8
JPCIE6A/JPCIE6B
JAOM1 CONN
2 x M.2 CONN
USB3.0 CONTROLLER
UPD720201
2 x USB 3.0
JPP2 CONN
USB3.0 CONN
1 x USB3.0
XGP805 CONN
AQC 10G LAN
SGMII
BMC
AST2600
DDR4
BMC Memory
2GB/4gb
eMMC
BMC eMMC
eMMC 32G
Firmwave SPI
SPI FLASH
64MB
SPI1
SPI2
SPI MUX
SPI
CPLD FLASH
512KB
CPLD1
LCMXO3LF-6900C
-6BG400I
CPLD2
LCMXO3LF-6900C
6BG324C
SMBUS
SIDEBAND Signals
SPI
BIOS SPI
64MB Flash
JCOM1 CONN
DEBUG PORT
UART
eSPI0
PCIe Gen3 x1
PCIe Gen2 x2
PCIe Gen5 x4
PCIe Gen5 x2
PCIe Gen5 x16
eSPI

RTX6000 Inference Server



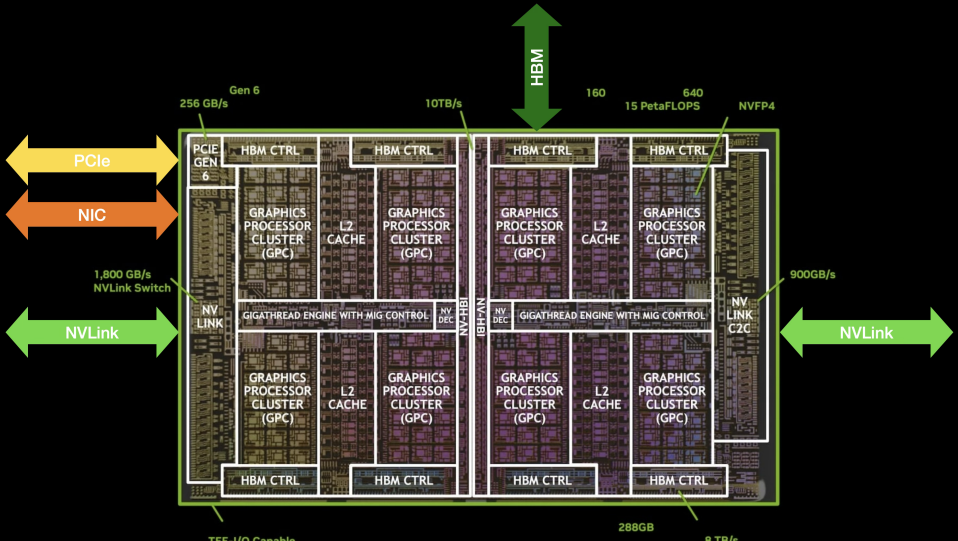
Inside One GPU: HBM on the Interposer, Links on the Edge



- HBM micrometers from the die, shared interposer — 8 TB/s, no 2-meter trace.
- PHYs + SerDes only on the die edge. Edge is scarce.
- B200 = two reticle dies bridged ∼10 TB/s; one GPU, 192 GB.

Compute $\propto$ die area; I/O $\propto$ perimeter. That mismatch shapes this whole section.

Blackwell B200 Die Shot



One Node, One Rack

HGX B200 node

- 8 B200 + 4 NVSwitch + 2 CPU + 8 NIC
- ∼1.5 TB HBM3e; 1.8 TB/s/GPU NVLink

GB200 NVL72 rack

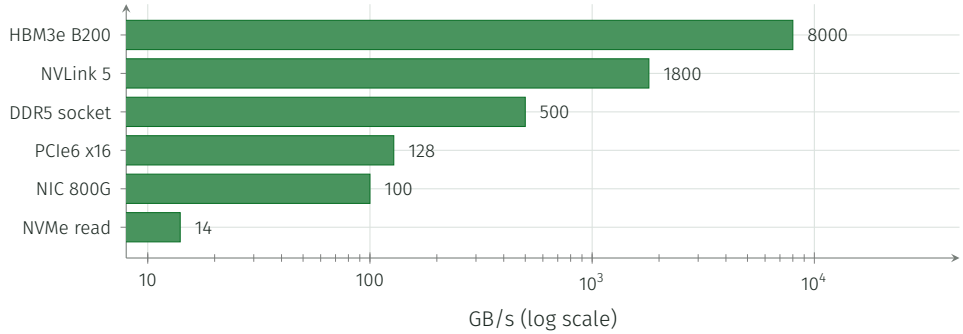
- 72 GPU + 36 Grace, one NVLink domain
- 13.5 TB HBM; 130 TB/s agg NVLink; ∼120 kW

Why a rack is one machine

72 GPUs, one 130 TB/s NVLink domain ⇒ MoE all-to-all stays inside the rack, off the ∼100 GB/s network. Large MoE was drawn to fit this box.

r1 dossier §7–8: NVIDIA GB200 NVL72.

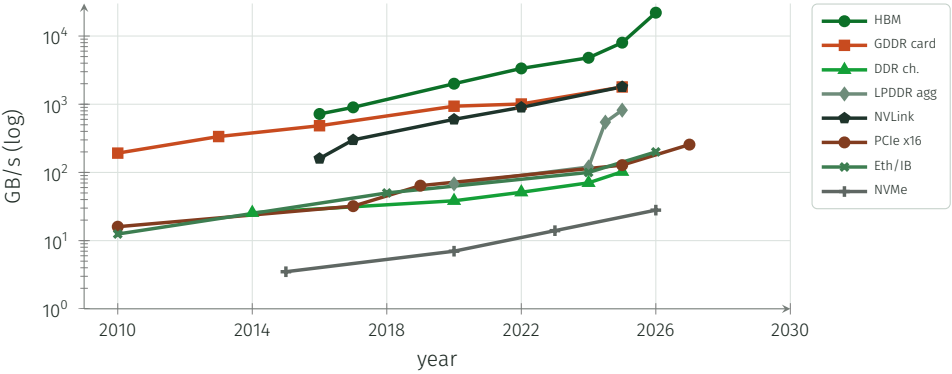
2026 Bandwidth Ladder



On-package HBM ∼60× a PCIe link, ∼80× a 800G NIC. Every chip boundary costs an order of magnitude. Keep the bytes home.

HBM3e/NVLink5/PCIe6 from r1; DDR5 socket (∼8–12ch DDR5-6400) and NVMe from r2.

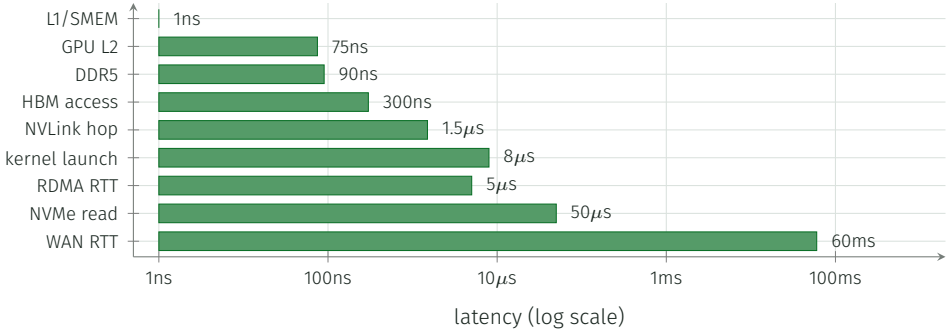
Bandwidth Over Time: Everything Doubles, HBM and LPDDR Pulled Away



Each family ∼2×/generation. HBM broke away by *stacking* DRAM dies onto the interposer.

r2 dossier CSV blocks (JEDEC, PCI-SIG, NVIDIA datasheets). Rubin/Gen7 points announced, not shipping.

2026 Latency Ladder: stream, don't chase pointers



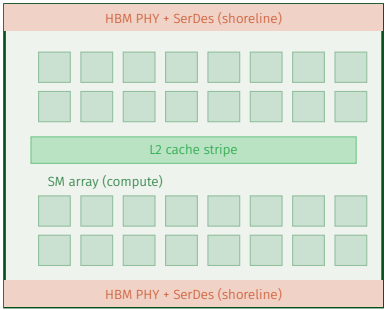
Caches cut *latency*; decode is bound by *bandwidth*. Weights read sequentially $\Rightarrow$ 300 ns HBM latency amortizes. What bites: the $\sim$16 GB streamed every token.

r2 dossier §9 (latency reference table).

Numbers Every LLM Engineer Should Know

Quantity	Value	Why it matters
H100 HBM bandwidth	3.3 TB/s	decode floor: bytes / this
B200 HBM bandwidth	8 TB/s	2.4× H100
H100 BF16 dense	1 PF	prefill ceiling
NVLink5 per GPU	1.8 TB/s	GPU⟷GPU, intra-node
PCIe5 x16 (1 dir)	64 GB/s	52× slower than HBM
800G NIC	100 GB/s	inter-node ceiling
NVMe seq read	14 GB/s	swap / KV offload tier
Qwen3-8B KV / token	150 KB	× ctx × batch
kernel launch overhead	5–15 μs	batch small ops

GPU Floorplan: Compute Is Area, I/O Is Perimeter



- SMs fill the interior: compute $\propto$ area $\propto n^2$.
- PHYs + link SerDes only on the edge: I/O $\propto$ perimeter $\propto n$.
- Worse each process shrink: $O(n^2)$ transistors, only $O(n)$ I/O pins.

The shoreline problem

Double the die side $\Rightarrow$ 4× compute, only 2× I/O. Bytes-per-FLOP keeps falling — *why* the model starves.

Beachfront/shoreline constraint: r4 §5 silicon geometry.

Spec Table 1 — The Die

	H100 SXM	B200 SXM	RTX 4090	RTX 5090
Arch	Hopper	Blackwell	Ada	Blackwell
Process	TSMC 4N	TSMC 4NP	TSMC 4N	TSMC 4NP
Die area	814 mm^2	$2\times\sim800$ mm^2	608 mm^2	744 mm^2
Transistors	80 B	208 B	76.3 B	92.2 B
TDP	700 W	1000–1200 W	450 W	575 W

Reticle limit $\sim$858 mm^2. H100 and 5090: near maxed-out monoliths. B200 = two reticle dies stitched — you can't print one bigger.

r1 dossier §1 (corrected). Reticle limit: r4 §5.

Spec Table 2 — The Memory

	H100 SXM	B200 SXM	RTX 4090	RTX 5090
Type	HBM3	HBM3e	GDDR6X	GDDR7
Capacity	80 GB	192 GB	24 GB	32 GB
Bus width	5120-bit	8192-bit	384-bit	512-bit
Bandwidth	3.35 TB/s	8.0 TB/s	1.008 TB/s	1.792 TB/s
Signaling	NRZ	NRZ	PAM4	PAM3

Capacity is the quiet crisis

Bandwidth +2.4× H100→B200; capacity only 1.4× (80→192 GB, caps at 8 stacks). 70B still won't fit a consumer card — and the 2025–26 DRAM squeeze made every gigabyte pricier.

r1 dossier §2 (corrected); DRAM squeeze: r2 §12 (DDR5 +307% Q4'25).

Spec Table 3 — Compute by Format (Dense, FP32-accumulate)

Format	H100 SXM	B200 SXM	RTX 4090	RTX 5090
FP64 (TC)	67 TF	∼40 TF	—	—
TF32	495 TF	1100 TF	—	—
BF16/FP16	989 TF	2250 TF	165 TF	210 TF
FP8	1979 TF	4500 TF	330 TF	419 TF
FP4	—	9000 TF	—	1676 TF

Marketing is 2× these — assumes 2:4 sparsity. The 5090's "3352 AI TOPS" is FP4 *sparse*; dense FP32-acc is 1676. Consumer cards also *halve* tensor throughput on FP32 accumulate.

r1 dossier §4 (RESOLVED BY LEAD AUTHOR — dense whitepaper values). Dense = no 2:4 sparsity.

On-Chip SRAM: Fast, Tiny, Not For Your Weights

Level (H100)	Size	vs HBM
Registers / SM	256 KB	on-chip
L1 / shared / SM	228 KB cfg.	∼10× faster
L2 (total)	50 MB	∼5× faster
TMEM / SM (B200)	256 KB	Blackwell only
All SRAM	∼50–130 MB	—
HBM (weights)	80–192 GB	—

Tiles, not weights

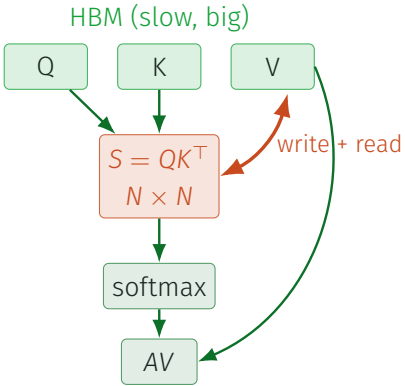
∼100 MB on-chip SRAM — your 16 GB model is ∼160× too big. SRAM holds the *tile* you compute on now; weights stream from HBM.

SRAM stopped shrinking. N5 to N3E: bitcell ∼0%, logic 1.7×. Compute gets cheaper; on-chip memory does not.

r1 §3 (cache sizes); SRAM scaling stall: r4 §6.

Flash Attention

Attention's Dirty Secret: The $N \times N$ Matrix Hits HBM

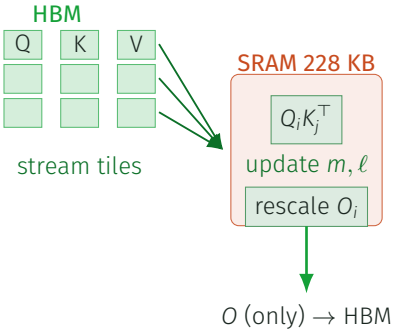


- Naive attention materializes full S to HBM: $\Theta(Nd + N^2)$ reads/writes.
- Write S, read for softmax, read again for AV — pure memory traffic.

At N=40k, d=128:
$S = 40\text{k} \times 40\text{k} \times 2\text{B} = 3.2\text{GB}$
per head, per layer, dragged through HBM every forward pass

Standard-attention IO: Dao et al. 2022, FlashAttention [arXiv:2205.14135] (Sec. 2).

FlashAttention: SRAM Tiling



- Stream Q/K/V blocks through SRAM
- keep a running max m and sum ℓ (prevent overflow/underflow)
- never materialize S

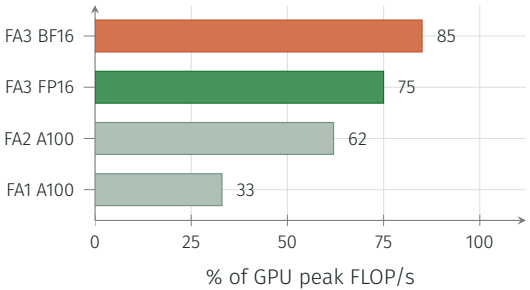
Online softmax, per new tile x:

$$m' = \max(m,\ x)$$
$$\ell' = e^{m-m'}\ell + e^{x-m'}$$
$$O' = e^{m-m'}O + e^{x-m'}V$$

HBM traffic drops from $\Theta(N^2)$ to $\Theta(N^2 d^2/M)$ (M = SRAM size) — up to $\sim 9\times$ fewer accesses, and memory now *linear* in N.

Online softmax + IO bound: Dao et al. 2022 [arXiv:2205.14135] (Alg. 1, Thm. 2).

From 25% to 85% of Peak



- **FA1**: 2–4× kernel speedup; 25–40% A100 peak.
- **FA2**: better warp partitioning
  ⇒ 50–73% (∼225 TF/s training).
- **FA3** (Hopper async TMA/WGMMA, pingpong):
  FP16 740 TF ∼75%
  BF16 ∼840 TF ∼85%
  FP8 ∼1.2 PF

PyTorch SDPA picks FA2/3 for you
FlashInfer is the serving-shaped (paged-KV) cousin used by vLLM/SGLang.

FA1 [arXiv:2205.14135]; FA2 [arXiv:2307.08691]; FA3 [arXiv:2407.08608] (FP16 740 / BF16 840 TF on H100).

Faster Formats

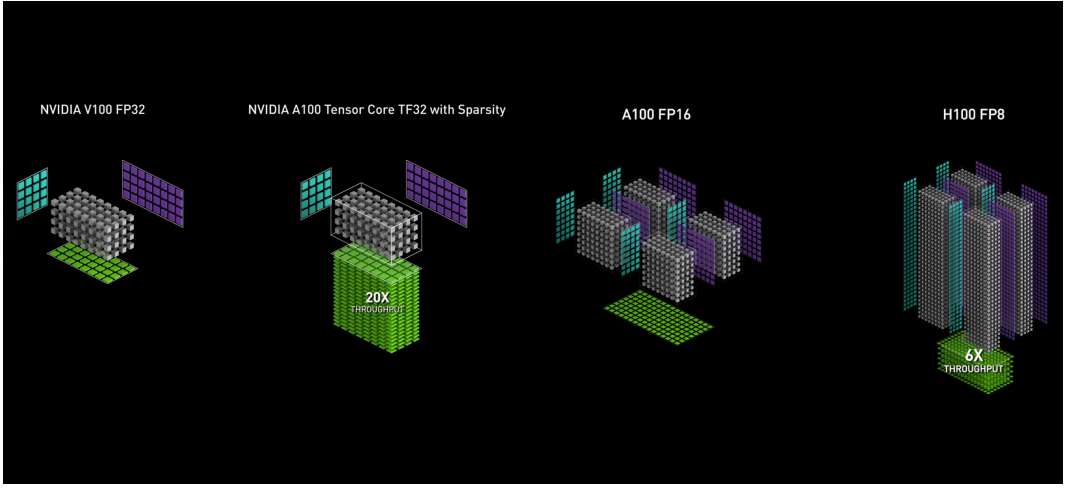
Compute Rich with Small Formats

Silicon area is *quadratic* in operand size.

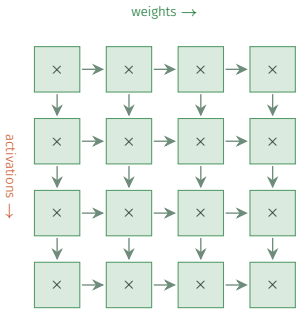
Format	Speedup	Bytes/elem	Real H100/B200 dense	First gen
FP64	1×	8	67 / 40 TF	forever
FP32	4×	4	67 / 80 TF	Pascal
FP16/BF16	16×	2	989 / 2250 TF	Volta/Ampere
FP8	64×	1	1979 / 4500 TF	Hopper
FP4	256×	0.5	— / 9000 TF	Blackwell

Real chip: each precision halving buys $\sim 2\times$, not $4\times$. Big jumps come from new *architectures*. And every halving also halves bytes moved — low precision wins twice.

Tensor Cores



Tensor Cores: Matmul-Shaped Silicon

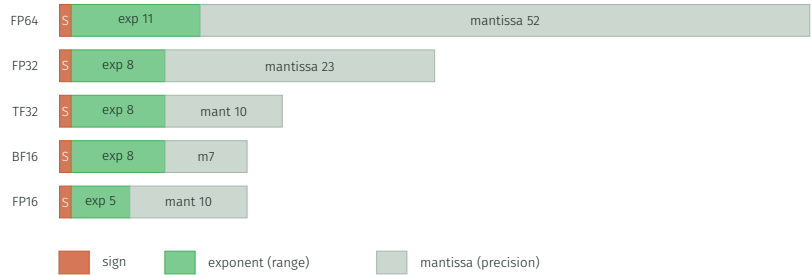


- Scalar FPU: most energy is instruction overhead, not the multiply.
- Tensor core hard-wires a MAC array (TPU-like).
- Lineage (BF16 dense):
  V100 125 → A100 312 → H100 989 → B200 2250 TF.

2:4 *structured sparsity* doubles peak again — but weights must be trained 2-of-4 zero (caution: marketing).

Dense lineage: r4 §3. Per-MAC energy (amortized HMMA ∼2–3 pJ): r4 §4 (Dally).

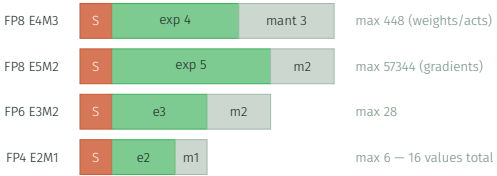
Number Formats I: Where the Bits Go



8-bit exponent for FP32/TF32/BF16 ⇒ same *range*, swap with no overflow. They differ only in mantissa (precision). FP16 trades range (5-bit exp) for precision (10-bit mant) — why training likes BF16.

Bit layouts: r4 §1 (IEEE 754, OCP, NVIDIA TF32 blog).

Number Formats II: Sub-Byte — FP8, FP6, FP4

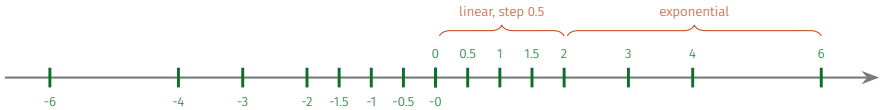


E4M3 vs E5M2

Same 8 bits, different split. E4M3 (more mantissa): forward weights/acts. E5M2 (more range): gradients. Below a byte you keep *range*, surrender *precision* — FP4 has a single mantissa bit.

r4 §1: OCP OFP8, FP6 (Blackwell MXFP6), FP4 E2M1.

FP4 Has 16 Values. You Can Count Them.

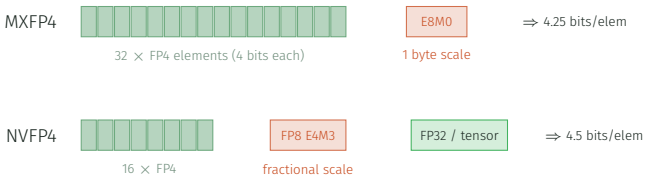


Quantization = mapping reals onto this

$\pm\{0, 0.5, 1, 1.5, 2, 3, 4, 6\}$ — the entire FP4 codebook. Spacing uniform near zero, exponential up top: more resolution where weights cluster. One global scale is hopeless; per-block scaling (next) makes 4-bit usable.

r4 §1: FP4 E2M1 codebook (OCP MX v1.0). 16 values incl. ± 0.

Block Scaling Rescues FP4: MXFP4 vs NVFP4

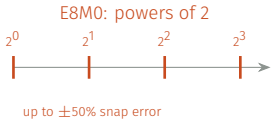


MXFP4 (OCP): block 32, E8M0 power-of-2 scale. gpt-oss ships it; 120B fits one H100.

NVFP4 (Blackwell): block 16, FP8 E4M3 fractional scale + FP32 per-tensor. Smaller block, finer scale ⇒ less error.

r4 §2: OCP MX v1.0 + NVIDIA NVFP4 dev blog. gpt-oss MXFP4: The Register 2025.

Why a Fractional Scale Beats a Power-of-Two Scale



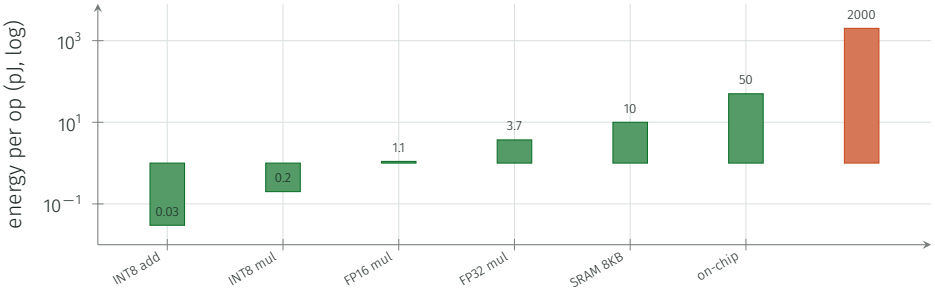
- E8M0: powers of 2 only ⇒ true max rounds up, wastes range (up to ±50%).
- E4M3: 3 mantissa bits ⇒ 12.5% steps, scale hugs the max, *clips less*.
- Block 16 vs 32: adapts to local outliers better.

Two levels

FP8 block scale handles *local* range; one FP32 per-tensor scale handles *global* magnitude — so the FP8 scale never overflows.

r4 §2: NVIDIA NVFP4 dev blog (E4M3 vs E8M0, two-level scaling).

Energy: A DRAM Access Costs ∼500 Multiplies

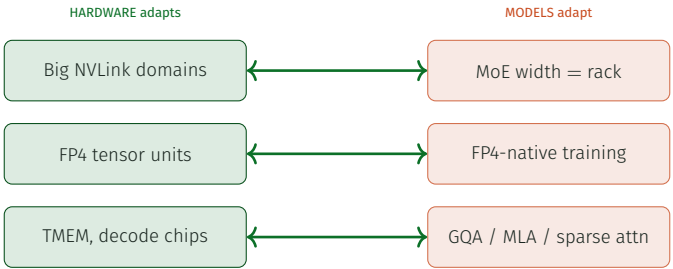


Data movement is the crime

One 64-bit DRAM read ≈ 2 nJ ≈ 2000 pJ ≈ ∼500 FP32 multiplies (3.7 pJ each). Arithmetic is free; *moving operands* is the whole budget. Every technique here moves fewer bytes.

r4 §4: Horowitz ISSCC 2014 (45 nm) + Dally 2023. DRAM 1.3–2.6 nJ; FP32 mul 3.7 pJ.

Convergent Evolution: Models and Silicon Co-Design



The loop closes

Hardware ships FP4 units and 72-GPU domains; models answer with FP4-native training and rack-sized MoE. Newest move: phase-specialized chips — GB300 NVL72 tuned for decode, prefill-optimized parts on the roadmap.

Co-design themes from r1 (NVL72, GB300) and r4 (FP4, TMEM).

Same Physics, Other Vendors

Chip	Mem	Type	TB/s	BF16 TF	FP8 TF	Scale-up
AMD MI300X	192 GB	HBM3	5.3	1307	2615†	Infinity Fabric
AMD MI325X	256 GB	HBM3e	6.0	1307	2615†	Infinity Fabric
AMD MI355X	288 GB	HBM3e	8.0	5000*	10100	Infinity Fabric
Google TPU v6e	32 GB	HBM	1.6	918	—‡	ICI 1.2 Tb/s
Google TPU v7	192 GB	HBM3e	7.4	2307*	4614	ICI 9.6 Tb/s
AWS Trainium2	96 GB	HBM3	2.9	667	1300	NeuronLink
Intel Gaudi 3	128 GB	HBM2e	3.7	1835	1835	RoCE 1.2 TB/s
NVIDIA B200	192 GB	HBM3e	8.0	2250	4500	NVLink5 1.8 TB/s

† FP8 listed sparse (2×). * estimated / from AMD FP16 datasheet. ‡ no separate FP8 spec. Dense unless noted.

HBM capacity × bandwidth, matmul ceiling on top. MI355X tops capacity (288 GB), ties B200 (8 TB/s). Same physics, same tactics across vendors.

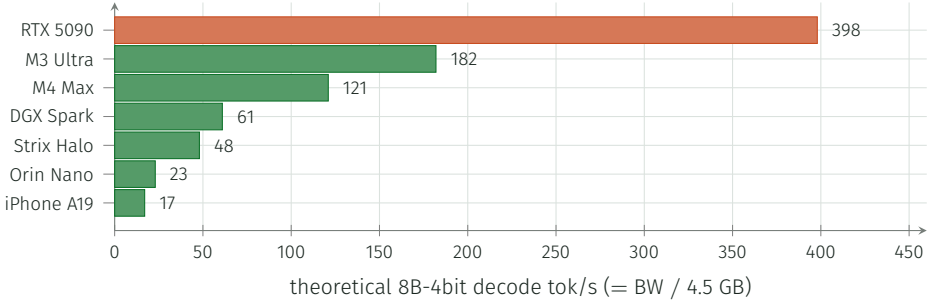
r11 block E: AMD, Google TPU, AWS, Intel Gaudi 3 white paper.

The Local-Inference Menu: Capacity × Bandwidth

Device	Mem	BW (GB/s)	AI compute	Price
Jetson Orin Nano Super	8 GB	102	67 TOPS†	$250
DGX Spark (GB10)	128 GB	273	∼1 PF FP4†	$4,000
Strix Halo (AI Max+ 395)	128 GB	256 (∼215 m.)	50 NPU TOPS	$2,000
Apple M4 Max	128 GB	546	38 TOPS	from $2,000
Apple M3 Ultra	512 GB	819	—	from $4,000
iPhone 17 Pro (A19 Pro)	12 GB	76	38 TOPS	1,000
RTX 5090 (system)	32 GB	1792	1676 TF FP4	$4–6k
H100 (per GPU)	80 GB	3350	989 TF BF16	$20–30k

† FP4/INT8 *with sparsity* (marketing). Unified memory = CPU/GPU/NPU share one pool.

Decode tok/s (aka bandwidth)



The Apple Silicon sweet spot

Realized ≈ 50–80% of these bounds. The 5090 wins on bandwidth but 32 GB won't hold a 70B; M3 Ultra's 512 GB will. And 30B-A3B *MoE* touches only ∼3B active/token — on 128 GB unified memory, the local frontier.

r3 §3 bound (BW/4.5 GB) + §4 measured (DGX Spark 8B FP8: 20.5 decode / 7,991 prefill).

The Other Pole: Put Everything in SRAM

System	SRAM	On-chip BW	Catch
Groq LPU	230 MB/chip	∼80 TB/s	7B needs 15+ chips; rack = 576
Cerebras WSE-3	44 GB/wafer	21 PB/s	wafer-scale; $M+ system
H100 (HBM ref)	—	3.35 TB/s	for contrast

Cerebras' 21 PB/s is ∼6,000× H100's HBM — no DRAM, no memory wall. But SRAM is huge and dear per bit: you buy bandwidth with silicon area and dollars, not stacked DRAM. Same question, different answer.

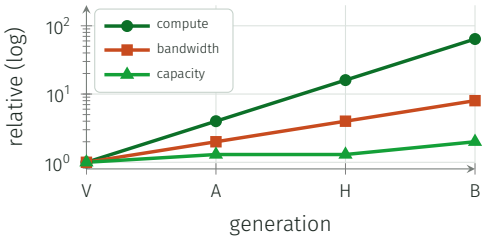
r3 §6: Groq 230 MB/∼80 TB/s; Cerebras WSE-3 44 GB/21 PB/s.

Hardware TL;DR: Arithmetic Intensity Only Goes Up

The three growth rates

Compute	$\sim 4\times$/gen ($2\times$ arch $\times$ $2\times$ fmt)
Bandwidth	$\sim 2\times$/gen (HBM, NVLink, PCIe, IB)
Capacity	$< 1.4\times$/gen (+ 2026 DRAM squeeze)

Compute outruns bandwidth outruns capacity. FLOPs-per-byte you're *allowed* to spend keeps rising — model gets hungrier, pipe doesn't keep up.



That gap is why the rest of this tutorial exists: fewer bytes per token — via serving (§3), weights (§4), KV (§5).

Part 3

Model Serving

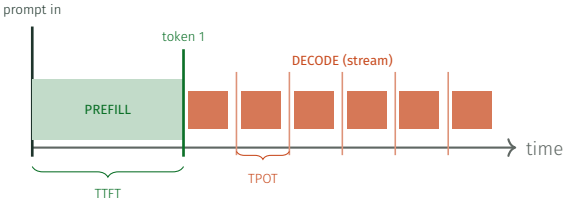
many users, one copy of the weights

Two Phases, Two Latencies, One SLO

- TTFT (first token) = prefill.
  Compute-bound; grows with prompt.
- TPOT/ITL (per token) = decode.
  Bandwidth-bound; one token/step.
- Latency $\approx$ TTFT $+ (L_{\text{out}} - 1)$ TPOT.
- **Goodput** = req/s *within* SLO.
  Big batch $\Rightarrow$ throughput $\uparrow$, TPOT $\uparrow$.

Chat SLO rule of thumb

TTFT < 1 s "feels instant".
TPOT 20–50 ms = reading speed ($\sim$10 tok/s).



One prefill, then a steady drip of decode steps.

Zhong et al. 2024, DistServe [arXiv:2401.09670]; Hao AI Lab. MLPerf SLO: TTFT/TPOT $\leq$500/30 ms.

Two Enemies: Idle GPUs and Wasted Compute

Idle GPU *problem → fix*

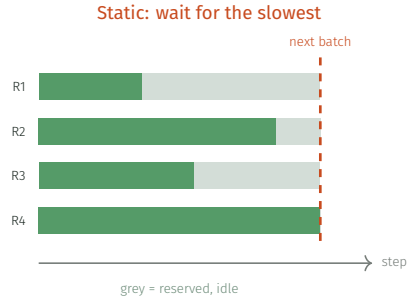
- Ragged lengths → continuous batching
- Long prefill stalls all → chunked prefill
- Imbalanced load → consistent hashing
- Prefill/decode differ → disaggregate
- Decode wastes FLOPs → speculation

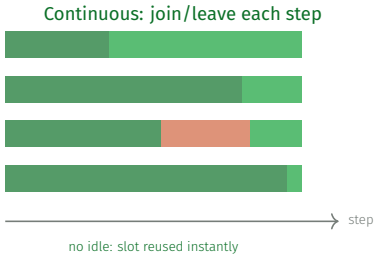
Wasted compute *problem → fix*

- Re-prefill same prefix → prefix cache
- Fragmented KV → paged blocks
- Cold cache → longest-match schedule
- Out of VRAM → pull from slower tier

One question, every slide: where does the GPU stall on memory, or recompute something we already had?

Static Batching Wastes the GPU; Batch Continuously



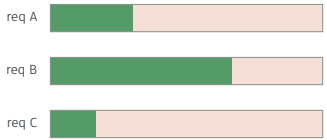


Orca's iteration-level scheduling: re-form the batch every decode step. **Up to 36.9×** vs FasterTransformer (GPT-3 175B); **up to 23×** over naive HF batching (AnyScale). Now universal: vLLM, TGI, TRT-LLM, SGLang.

Yu et al. 2022, Orca (OSDI'22); AnyScale continuous-batching benchmark.

Fighting KV Memory Fragmentation

Pre-vLLM: one contiguous chunk / request



Reserve for *max* length
⇒ internal + external + reservation waste.

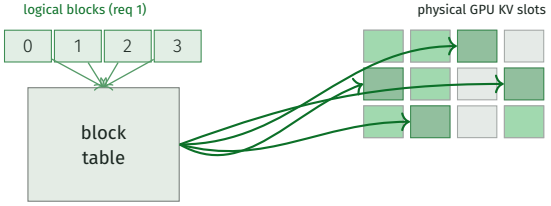
vLLM: fixed 16-token blocks, on demand

Blocks from different requests interleave;
only the *last* block of each is partly empty.

Pre-vLLM systems: only 20–40% of KV memory held real tokens. Rest was waste — it capped batch size.

Kwon et al. 2023, PagedAttention (SOSP'23 best paper) [arXiv:2309.06180]; Red Hat explainer.

PagedAttention: Virtual Memory for the KV Cache



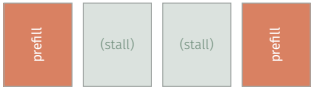
Copy-on-write: two requests share the same prefix block (`ref=2`); copy only when one diverges.

Block table maps logical → arbitrary physical blocks. Near-zero fragmentation; prefix sharing for free. **2–4× throughput** over Orca/FasterTransformer at equal latency.

Kwon et al. 2023, vLLM [arXiv:2309.06180]; `vllm-project/vllm`. V1: prefix caching on by default.

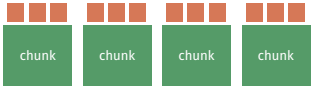
Chunked Prefill: Don’t Let One Long Prompt Stall Everyone

Naive: a 40k prefill freezes all decodes



one iteration = one token budget

Sarathi: chunk + piggyback decodes



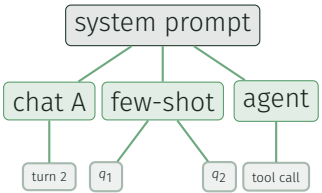
decodes never pause; prefill amortized

Stall-free batching

Fixed token budget/step = one prefill chunk (256–512 tok) + running decodes. Better TTFT *and* TBT at once. Default scheduler in vLLM V1.

Agrawal et al. 2024, Sarathi-Serve (OSDI’24) [arXiv:2403.02310]; orig. Sarathi 2023 [arXiv:2308.16369].

Prefixes Are Everywhere ⇒ Cache Hits Are Free Prefill



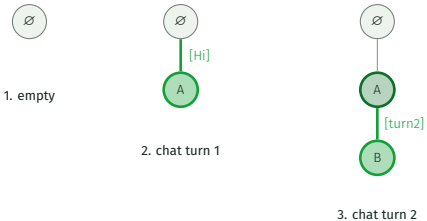
The reused prefix is most of the tokens:

- **Multi-turn chat** — replay history + system prompt.
- **Few-shot** — same exemplars, every query.
- **Self-consistency / ToT** — many samples, one prompt.
- **Agents** — shared tool descriptions, RAG templates.

The opportunity

Cache hit = skipped prefill: lower TTFT, freed compute, bigger batches. Next: how to store *all* of it.

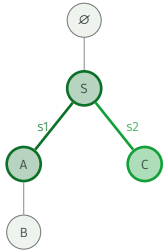
RadixAttention: A Radix Tree Over Token Sequences (1/2)



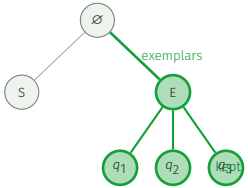
Keep finished requests' KV; index it by token sequence. On turn 2, walking to node **A** reuses the whole history — no re-prefill. Shared prefixes share a path; LRU evicts cold leaves.

Zheng et al. 2024, SGLang RadixAttention [arXiv:2312.07104]; LMSYS blog (Fig. 4).

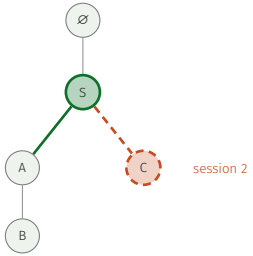
RadixAttention: Split, Share, Then Evict the Cold (2/2)



4. share system prompt S



5. few-shot batch (share E)



6. LRU evicts cold leaves

Up to 5× throughput on shared-prefix workloads

Split node S ⇒ two sessions share the system prompt. LRU drops cold *leaves* first. 3.1× p50; zero overhead at 0% hits ⇒ always on.

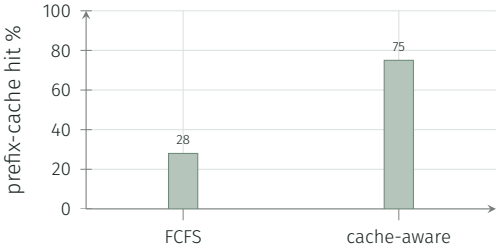
Zheng et al. 2024, SGLang [arXiv:2312.07104]; `sgl-project/sglang`.

Cache-Aware Scheduling: Run the Longest-Hit Request First

- **FCFS:** ignores the cache, re-prefills prefixes it just evicted.
- **Cache-aware:** run the request reaching the deepest cached node first ⇒ more hits, lower TTFT.
- Composes with continuous batching + paging; zero cold-start cost.

Your hit rate is your TTFT

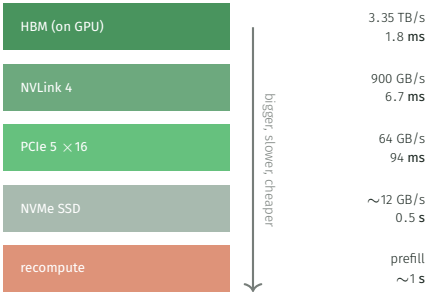
Hit rate ↑ ⇒ prefill skipped ⇒ TTFT ↓ and batch size ↑.



Scheduling order moves hit rate, not the cache size.

SGLang v0.4 cache-aware router: up to 1.9× throughput, 3.8× hit-rate.

Copy vs Recompute: 6 GB KV Cache Down the Memory Ladder



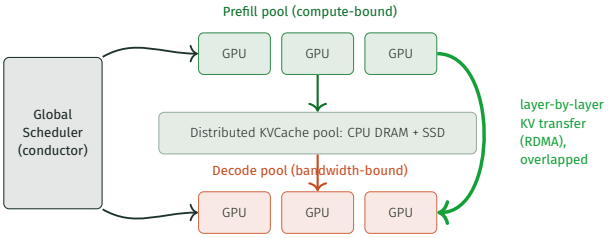
Qwen3-8B, 40k ctx:
147 KB/tok × 40k = 6 GB KV.
Reload it, or rebuild it from scratch?

Decision rule

Reload *beats* recompute only on a fast link (RDMA/NVLink). Slow link ⇒ recompute wins — so CacheGen compresses KV for the wire.

r5: 6 GB over PCIe5 ≈ 0.1 s vs recompute ≈ 1 s; `LMCache/LMCache`; CacheGen [arXiv:2310.07240].

Disaggregated Prefill/Decode: The Mooncake Architecture



Each pool tuned for its phase; conductor picks a (prefill, decode) pair per request.
Reuse prefix → chunked prefill → stream KV over RDMA → decode, all continuously batched.
Up to +525% throughput (sim), **+75% requests** under real Kimi load (>100B tokens/day).

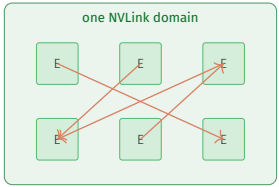
Qin et al. 2024, Mooncake (FAST'25 best paper) [arXiv:2407.00079];
`kvcache-ai/Mooncake`; also DistServe, Splitwise, Dynamo.

Match the Fleet to the Model: NVL72 ↔ Large MoE

- Large-MoE decode = all-to-all expert dispatch, every layer.
  That traffic wants one fat interconnect.
- **GB200 NVL72:** 72 GPUs, *one* NVLink domain (130 TB/s).
  Expert parallelism stays on-fabric.
- Disaggregation sizes pools separately:
  prefill-heavy (long docs) vs. decode-heavy (chat).

Convergent evolution

Models grow MoE width to a NVLink domain.
Hardware grows the domain to fit the model.
(Bridges §2 hardware and §4 MoE)

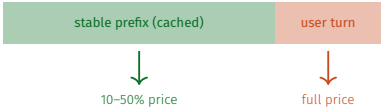


Experts (E) scattered on GPUs; all-to-all every token.

GB200 NVL72: 72 GPUs, 13.5 TB HBM3e, ∼130 TB/s aggregate NVLink5 (canonical numbers). Expert parallelism within one domain.

You've Already Paid for This: Prompt-Caching Prices

Provider	Cached read	Write / storage	Trigger
Anthropic (Claude)	0.1× input (90% off)	1.25× (5 min) / 2× (1 h)	explicit breakpoints
OpenAI	0.5× input (∼50% off)	— (free)	implicit, >1024 tok
Google Gemini	0.25× input (∼75% off)	per-minute storage	implicit/explicit, ≥32k tok



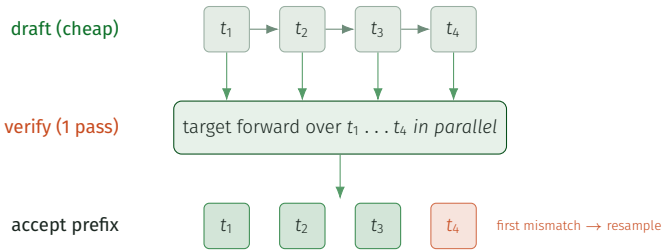
Structure your prompt

Stable part (system prompt, docs, few-shot) first ⇒ it gets cached. **Your bill is a cache-hit-rate report** — RadixAttention, surfaced as billing.

Pricing docs, verified mid-2026: Anthropic, OpenAI, Gemini.

Jobs for Lazy GPUs

Speculative Decoding: Spend Idle FLOPs to Buy Tokens



Why it's free

Decode wastes FLOPs — spend them. Cheap draft proposes k tokens; target verifies all $k{+}1$ in one forward pass = one weight sweep. Bandwidth-bound, so that pass re-reads the same bytes as a single token — but validates several.

Leviathan et al. 2023 [arXiv:2211.17192]; Chen et al. 2023 [arXiv:2302.01318].

Exact by Maximal Coupling

Draft proposes $x \sim q$ but we want emitted tokens distributed as $x \sim p$.

Rule

- Accept x w.p. $\min\left(1, \frac{p(x)}{q(x)}\right)$;
- On reject, resample $x \sim (p-q)_+ / \sum_y (p-q)_+$.

Proof

- Accepting contributes $q(x)\min\left(1, \frac{p}{q}\right) = \min(p, q)$
- Rejecting-then-resampling contributes $(p-q)_+$ (suitably normalized)
- In balance $\Pr[\text{emit } x] = \min(p, q) + (p-q)_+ = p(x)$

Maximal coupling (this is not MH or envelope rejection)
Per-token accept rate $a = \sum_x \min(p, q) = 1 - \mathrm{TV}(p, q)$.

Speedup. Accept rate a, g draft tokens/step:

$$\mathbb{E}[\text{tokens/step}] = \frac{1 - a^{g+1}}{1 - a}.$$

a	0.6	0.8	0.9
$\mathbb{E}[\text{tok}]$ (g=4)	2.31	3.36	4.10

Output distribution unchanged — no temperature hack, no quality knob.
Better draft $\Rightarrow$ smaller TV $\Rightarrow$ higher a $\Rightarrow$ longer accepted prefixes.
Wasted draft compute only on rejects.

Speculative sampling / maximal coupling: Leviathan et al. 2023 [arXiv:2211.17192] (eq. 1, Thm. 1); Chen et al. 2023 [arXiv:2302.01318].

Even faster Drafting: EAGLE-3 and MTP

EAGLE-3 — feature-level drafting

- Draft from the target's own *multi-layer features* (training-time test), not a separate model.
- Up to 6.5×; ∼3.9–4.1× on Llama-3.3-70B.
- SGLang on H100: 158 → 373 tok/s.

DeepSeek-V3 MTP — self-speculation

- One extra **Multi-Token-Prediction head**
- Built-in drafter almost for free, *no* separate model;
- 2nd-token accept 85–90%
- 1.8× TPS at batch 1 (SGLang H200).

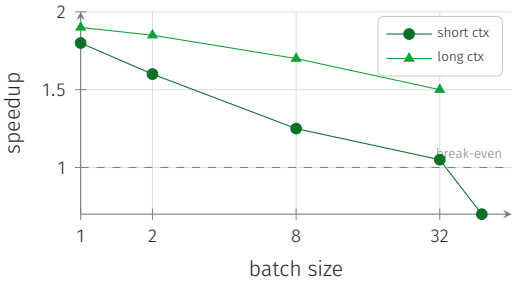
Medusa

Bolt extra heads onto a frozen model. Simplest to train; shorter accepted prefixes than EAGLE.

Kill the separate draft model. Predict from the target's own hidden states (EAGLE-3) or a trained head (MTP) — the drafter rides along almost free.

EAGLE-3 [arXiv:2503.01840] (up to 6.5×); DeepSeek-V3 [arXiv:2412.19437] (MTP self-spec, 1.8×).

When Speculation Backfires



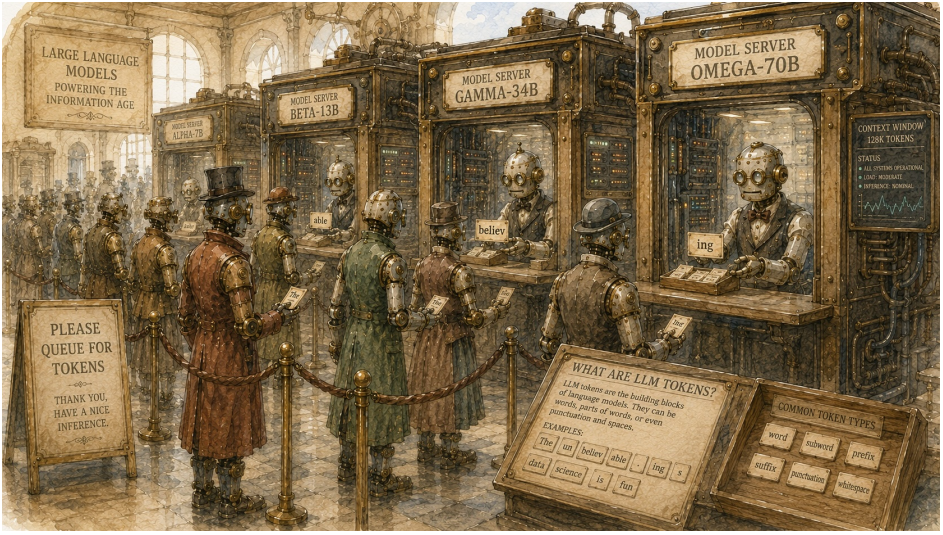
- Large batch = compute-bound
  there are no idle FLOPs left, so verifying rejected drafts is pure overhead.
- MagicDec: speedup drops below 1×
  (LLaMA-2-7B-32K: ∼0.95× at batch 32, S=1024); recovers for long context, where decode stays bandwidth-bound.

Rule of thumb

- Latency mode (small batch) ⇒ yes.
- Saturated-throughput mode ⇒ measure first — it can cost you tokens.

Chen et al. 2024, MagicDec [arXiv:2408.11049]: speedup <1× at large batch + short context.

Many Clients, One Fleet of Replicas



Same Prefix → Same Replica, Even When One Dies

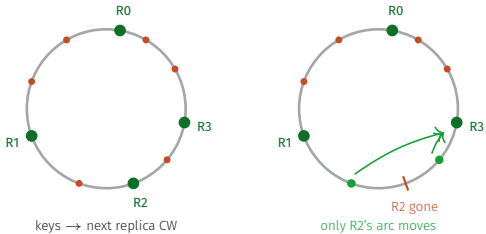
- Prefix caches live on one GPU.
- Shared prompt, different replicas ⇒ full re-prefill.

Consistent hashing

- Map keys *and* replicas onto a ring.
- Key → first replica clockwise.
- Drop a replica (server dies): only its arc ($\sim \frac{K}{N}$ keys) remaps, not $\frac{N-1}{N}$.

Virtual nodes

Each replica at 100–1000 ring points ⇒ smooth load, even churn.



Karger et al. 1997, consistent hashing; SGLang router; Dynamo/llm-d KV-aware routing.

The Serving Ecosystem

Engines & orchestration

- `vllm-project/vllm`
  PagedAttention reference stack.
- `sgl-project/sglang`
  RadixAttention, structured output.
- `NVIDIA/TensorRT-LLM`
  in-flight batching, multi-node.
- `ai-dynamo/dynamo`
  datacenter disagg + KV-aware routing.
- `ggml-org/llama.cpp`
  `ollama/ollama`
  `ml-explore/mlx`
  local/edge.

KV & cache layers

- `LMCache/LMCache`
  cross-request KV reuse & offload.
- `kvcache-ai/Mooncake`
  disaggregated KV store.
- `flashinfer-ai/flashinfer`
  paged-KV / spec-tree kernels.

Spec decoding ships in vLLM, SGLang, TensorRT-LLM. Turn it on — then measure at *your* batch size.

Engine repos linked above; serving moves recapped next. Verified mid-2026.

Serving TL;DR: Five Moves to Feed the GPU

1. **Batch continuously**
   iteration-level scheduling, never wait for the slowest (up to 23×).
2. **Page the KV**
   block table kills fragmentation (2–4×).
3. **Cache prefixes**
   radix tree reuses shared history (up to 5×).
4. **Route consistently**
   same prefix → same replica.
5. **Disaggregate phases**
   prefill and decode want different hardware (+59–498%).

Every move buys back arithmetic intensity: more useful work per byte the GPU drags out of memory.

Next (§4): make the weights themselves cheaper — MoE uses fewer of them, quantization stores fewer bits each.

vLLM, SGLang, Mooncake, DistServe — see §3 sources. Best teaching reads: LMSYS and AnyScale.

Part 4

Weight Compression

same brain, fewer bytes

BosonAI

4

Every Decoded Token Reads All of the Weights

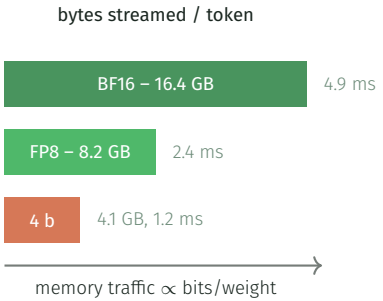
Decode bandwidth-bound (§1): one token, every weight, once.

$$t_{\text{floor}} = \frac{\text{bytes}}{\text{bandwidth}} = \frac{16.4\,\text{GB}}{3.35\,\text{TB/s}} \approx 4.9\,\text{ms}$$

Qwen3-8B, BF16, H100. Halve the bytes ⇒ halve the floor — *and* fit a smaller GPU.

Two wins from fewer bytes

Speed (bandwidth-bound decode) **and** capacity (fits the GPU you can afford).



H100 80 GB @ 3.35 TB/s; Qwen3-8B BF16 ≈16.4 GB (canonical numbers, §1).

The FFN Is Two-Thirds of the Weights

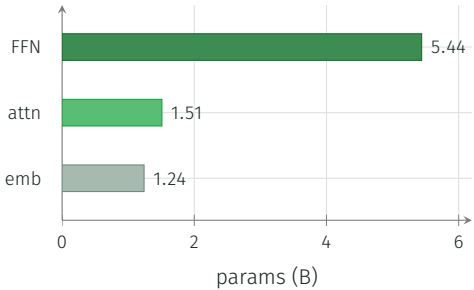
Qwen3-8B per-layer FFN

$$2{\cdot}H{\cdot}I \text{ gate/up} + I{\cdot}H \text{ down}$$

swamps attention.

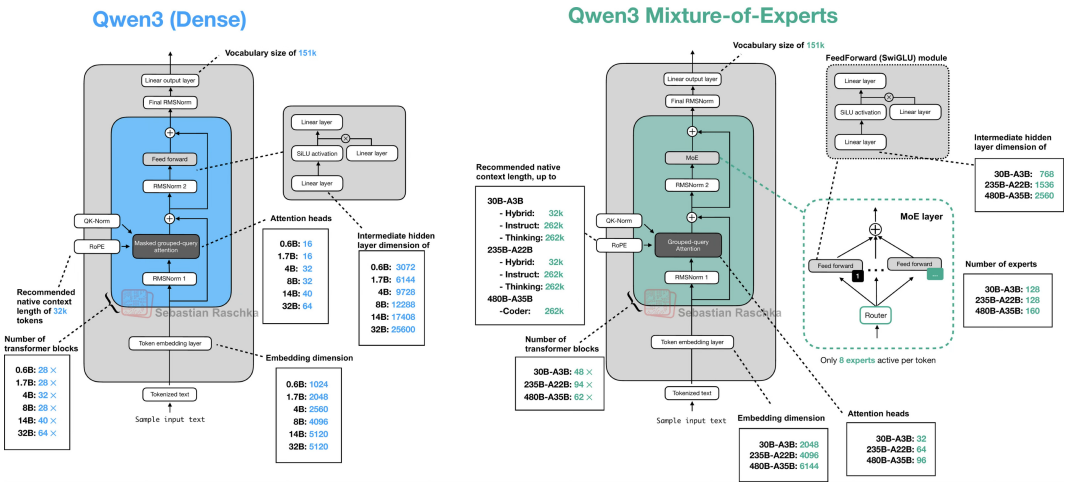
- FFN 5.44 B params (66%)
- Attention 1.51 B (18%)
- Embeddings (untied) 1.24 B

Every token runs the whole FFN.
What if each used only a *slice*?

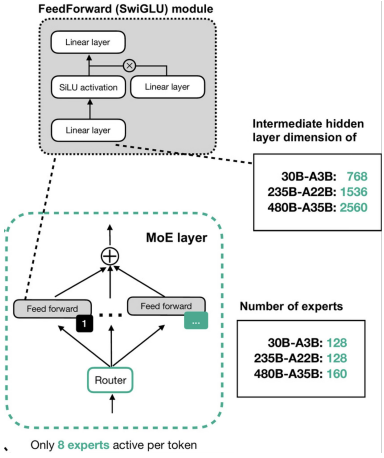


Qwen3-8B `config.json`; breakdown in r6 (8.19 B total). Qwen3 2025 [arXiv:2505.09388].

Qwen3 Dense vs. Mixture of Experts



Mixture of Experts: Learn Which Slice to Use



MoE layer. Score experts, keep top k:

$$g = \mathrm{softmax}(W_r h) \in \mathbb{R}^E, \quad \mathcal{S} = \text{top-}k(g)$$

$$y = \sum_{i \in \mathcal{S}} \frac{g_i}{\sum_{j \in \mathcal{S}} g_j} E_i(h)$$

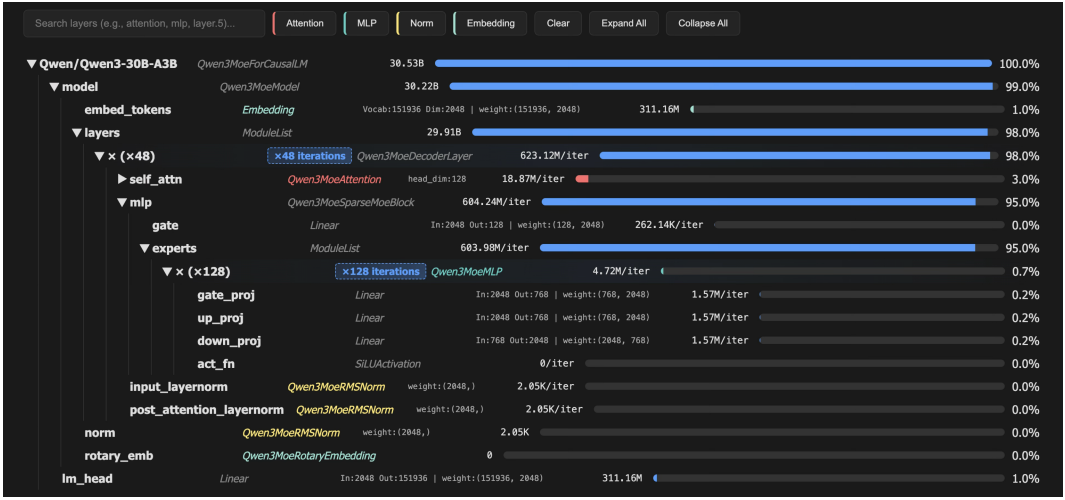
DeepSeek-V3 / Qwen3: sigmoid gate (decouples experts), renormalised top-k.

The key move

Same nonlinearity *budget*, $\sim 10\times$ fewer active FLOPs — a learned choice of which block to use.

Shazeer 2017 [arXiv:1701.06538]; DeepSeek-V3 2024 [arXiv:2412.19437] (sigmoid, aux-free).

Qwen3 Mixture of Experts Detail



Qwen3: 30B-A3B (MoE) vs. 8B (Dense)

	Qwen3-8B (dense)	**Qwen3-30B-A3B** (MoE)
Total params	8.19 B	30.5 B
Active / token	8.19 B	3.35 B
Layers	36	48
Hidden H	4096	2048
Q / KV heads	32 / 8	32 / 4
Experts	—	128, top-8
Expert FFN I	12,288 (dense)	768 (per expert)
Shared expert	—	none

Sparsity

Each expert *tiny* (4.7 M). 128-way pool buys capacity; top-8 fixes the FLOP bill. $30.5/3.35 \approx 9\times$ sparsity.

Verified `config.json` (r6). Qwen3-MoE dropped the shared expert (vs Qwen2-MoE).

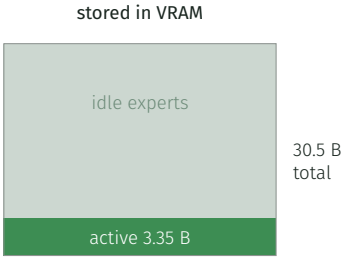
MoE Trades VRAM for FLOPs

30B-A3B vs 8B-dense

- active compute $\times 0.4$ (3.35 vs 8.19 B)
- stored weights $\times 3.7$ (30.5 vs 8.19 B)

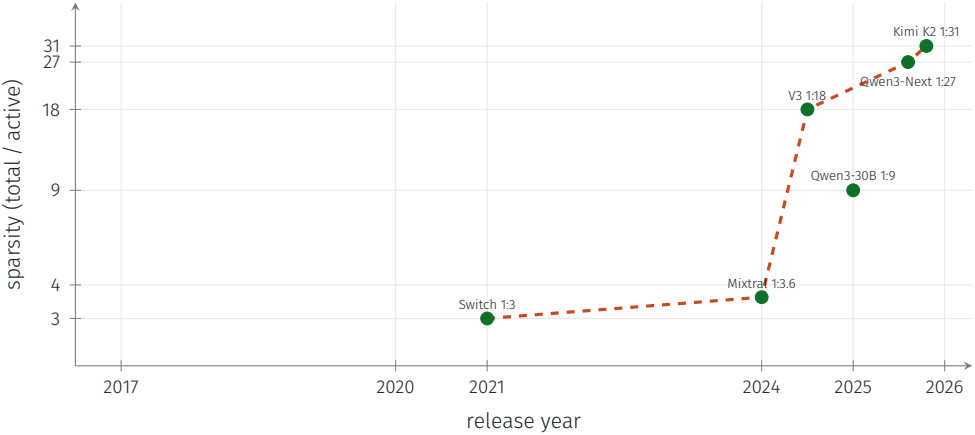
Batch size matters

- batch 1: stream only active experts
- all experts still resident in VRAM
- batch > 1: more distinct experts touched $\Rightarrow$ byte savings erode



Active/total from r6. Good trade when VRAM is cheap relative to bandwidth.

MoE Lineage: Sparsity Climbs from 1:4 to 1:31



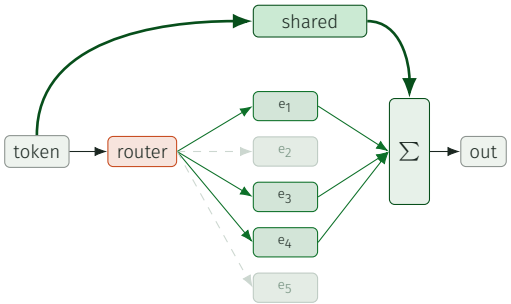
r6 lineage: Shazeer'17 → GShard → Switch → Mixtral → DeepSeek-V3 (671/37 B) → Qwen3/Kimi K2.

DeepSeekMoE: Sparse plus Dense

Best of both worlds

- Fine-grained sparse
  - split each expert into m
  - route top-mk
  - sharper specialization
- Shared expert
  always-on, absorbs common knowledge

DeepSeekMoE-16B $\approx$ Llama2-7B quality at $\sim$40% FLOPs.



Dai et al. 2024, DeepSeekMoE [arXiv:2401.06066]. Adopted by DeepSeek-V3, Kimi K2, GLM-4.5.

Load Balancing — Saving all the Experts

Collapse

- router keeps picking a few experts
- the rest starve, get no gradient, die

Auxiliary loss (Switch/GShard)

$$\mathcal{L}_{\text{aux}} = \alpha N \sum_i f_i P_i$$

- f_i = fraction of tokens routed to i
- P_i = mean gate probability
- gradient hits P_i, pushes down over-used

Aux-loss-free (DeepSeek-V3)

- per-expert bias b_i on top-k *selection* ($s_i + b_i$), not gate weight
- optimize the offset directly

$$b_i \leftarrow b_i + \gamma \,\mathrm{sign}(\bar{c} - c_i)$$

  raises under-used, lowers over-used
- *no* LM-gradient interference

Fedus et al. Switch [arXiv:2101.03961]; aux-free bias from DeepSeek-V3 [arXiv:2412.19437].

Serving MoE: All-to-All Wants One Big NVLink Domain

Expert parallelism (EP)

Experts shard across GPUs; a token's top-k lives on *other* GPUs:

1. dispatch — tokens → expert GPUs (all-to-all)
2. compute experts
3. combine — gather results back (all-to-all)

High sparsity (8/256) wants a *huge* global batch so each expert sees enough tokens.

GB200 NVL72 = 72 GPUs, one domain

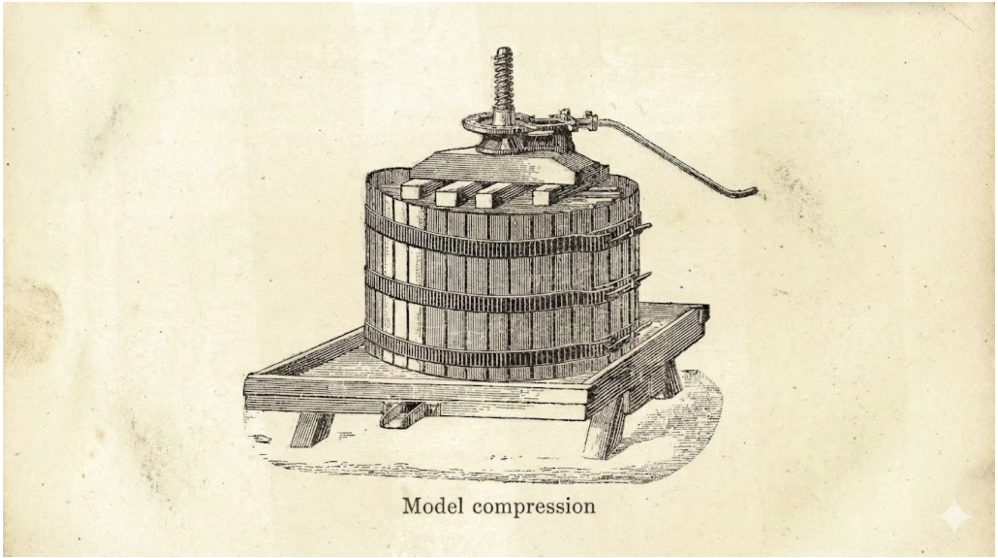
DeepSeek-V3 production (Feb 2025)

- Prefill EP32 (4 nodes)
- Decode EP144 (18 nodes, 2 routed + 1 shared expert/GPU)
- Two micro-batches overlap so attention hides dispatch/combine

Bridge to quantization: gpt-oss ships its MoE weights in MXFP4 — 120 B fits one 80 GB H100.

r6 A3: EP dispatch/combine; GB200 NVL72, gpt-oss MXFP4.

Squeezing the models



Two Ways to Spend Fewer Bits per Weight

Native-format inference

Compress to a format the GPU executes directly — INT4, FP8, NVFP4 (generation-dependent).

Very fast, but limited datatypes.

Decompress-and-discard

Compress to any codec (lattice, trellis, entropy). Decompress *per layer*, compute, throw the floats away.

Fits low RAM, but you pay the decode.

Blueprint after the author's reference deck (PTQ = post-training quantization).

Dumb Idea: gzip the Weights?

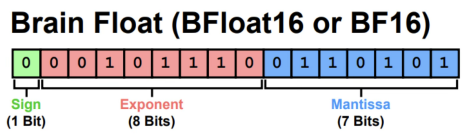
- weights are big
- LZ-family codecs are free (built-in for B200/B300)
- gzip the checkpoint?
- barely helps. The *mantissa* bits look like noise (save only 0–7%)

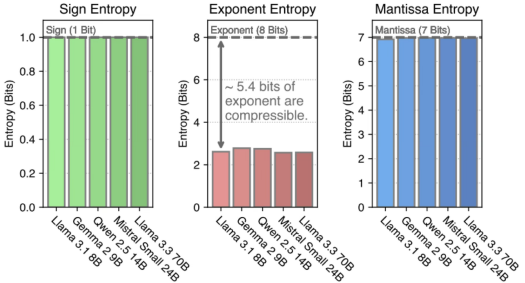
But the Exponents Are Almost Free

- BF16 reserves 8 exponent bits
- in a *trained* model: only ∼2–3 bits of entropy

$$H(e) = -\sum_{e} p(e) \log_2 p(e) \;\approx\; 2\text{–}3 \;\ll\; 8$$

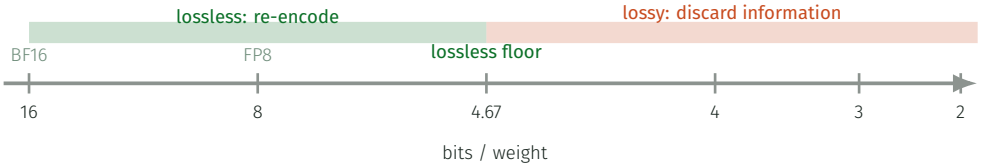
- ECF8: Huffman the exponent, sign+mantissa raw
- GPU kernel, hierarchical LUTs, decode per block to hide latency





Yang & Shrivastava et al. 2026, ECF8 [arXiv:2510.02676], ICLR 2026. zeyuyang8/ecf8.

To Go Below the Floor, Throw Away the Right Bits



The rest of this section

Below ∼4.7 bits, information *must* go. The art: drop the bits that don't matter — the data tell you which.

Lossless floor ≈ FP4.67 (ECF8 [arXiv:2510.02676]); lossy = GPTQ / AWQ / lattice / trellis.

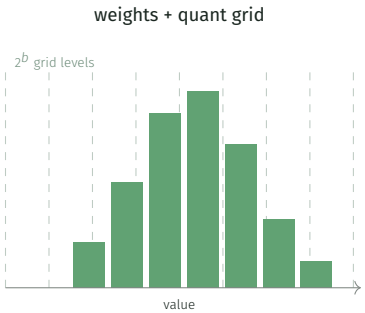
Quantization 101: Round to a Grid, Pick a Scale

Round-to-nearest with a scale s and zero-point z:

$$\hat{w} = s \cdot \left(\mathsf{clip}(\lfloor w/s \rceil + z,\ 0,\ 2^b{-}1) - z \right)$$

Scale *granularity* is the dial:

- per-tensor: 1 scale, cheap, fragile
- per-channel: 1 scale / output row
- per-group (g=128): the modern default



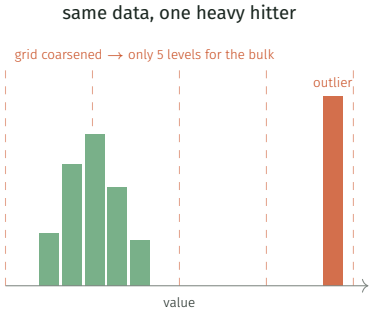
Standard RTN PTQ. Per-group g=128 is the de-facto default in llm-compressor / GPTQ / AWQ.

One Outlier Stretches the Grid for 127 Neighbours

One heavy weight forces s large enough to hold it. Grid spacing now coarse *everywhere*: the 127 ordinary weights share far fewer effective levels.

The villain of low-bit quant

Why naive 4-bit hurts. Three answers follow: compensate (GPTQ), protect (AWQ), remove (rotation).



Outlier channels are the central obstacle to W4/W4A4 (r6 B6/B9).

Aside: Training in Low Precision Is the Hard Direction

Why training fights you

- tiny gradients → underflow in FP16 → *loss scaling*
- *FP32 master weights*: $lr \cdot g$ vanishes if added in low precision
- long dot products lose low bits → *FP32 accumulate*; *stochastic rounding* keeps updates unbiased

DeepSeek-V3 FP8 recipe

- GEMMs in FP8 (E4M3) at 671 B
- tile/block scaling (act 1×128, weight 128×128)
- promote partial sums to FP32 every 128 K-elements

Inference is forgiving

Forward pass only: no gradient underflow, no accumulation over millions of steps, errors don't compound. Weights fixed — spend offline to round optimally.

Catch: *activations* carry outlier channels ⇒ W4A16 easy, W4A4 needs rotations (4.30).

4 bit for the win

$4\times$ smaller footprint $\approx 4\times$ faster decode.

DeepSeek-V3 [arXiv:2412.19437] FP8; NVFP4 pretraining 2025 [arXiv:2509.25149] (12 B, 10 T tokens).

Use the Data


AND NOW FOR SOMETHING COMPLETELY DIFFERENT
MATH INTERLUDE

GPTQ I — Beyond Round-to-Nearest

Searching under the lantern pole

- RTN minimises $\left\|W - \hat{W}\right\|^2$
- but we care about the layer's output on real inputs
- minimise it over a calibration set X

$$\left\|WX - \hat{W}X\right\|_2^2 = \operatorname{tr}\left[(W - \hat{W})^\top \underbrace{XX^\top}_{=:\,Q}(W - \hat{W})\right]$$

- curvature = the per-layer Hessian

$$Q = XX^\top, \qquad H = 2Q$$

Key idea

Calibration data turns quantization into a least-squares problem. Everything clever (GPTQ, AWQ) follows from $\left\|WX - \hat{W}X\right\|^2$.

Optimal brain damage

- OBD (2nd-order pruning)
- OBS (inverse-Hessian update)
- OBQ (greedy, per-row)
- GPTQ (fixed column order, all rows at once, lazy block updates — 175B).

Frantar et al. 2023, GPTQ [arXiv:2210.17323], ICLR 2023.

GPTQ II — Closed Form via the Schur Complement

Split the error $\delta = w - \hat{w}$ into quantized (q) and free (f) coordinates, and expand

$$(w-\hat{w})^\top Q\,(w-\hat{w}) = \begin{pmatrix}\delta_f\\ \delta_q\end{pmatrix}^\top \begin{pmatrix}Q_{ff} & Q_{fq}\\ Q_{fq}^\top & Q_{qq}\end{pmatrix}\begin{pmatrix}\delta_f\\ \delta_q\end{pmatrix}$$

Minimise over the free coordinates δ_f for a *given* quantization δ_q:

$$\partial_{\delta_f}[\cdots] = 2Q_{ff}\delta_f + 2Q_{fq}\delta_q = 0 \quad\Longrightarrow\quad \boxed{\delta_f = -\,Q_{ff}^{-1}Q_{fq}\,\delta_q}$$

Substitute back: the quantization cost

$$(w-\hat{w})^\top Q\,(w-\hat{w}) = \delta_q^\top\left[Q_{qq} - Q_{fq}^\top Q_{ff}^{-1}Q_{fq}\right]\delta_q = \delta_q^\top\left[(Q^{-1})_{qq}\right]^{-1}\delta_q$$

The bracket is the Schur complement of Q_{ff}; it equals $[(Q^{-1})_{qq}]^{-1}$. GPTQ [arXiv:2210.17323].

GPTQ III — Making It Fast: Cholesky + Lazy Blocks

Exact OBS re-touches all weights after every column — too slow at 175 B. GPTQ's three moves:

- fixed column order
  skip the "cheapest next" search
- 128-column blocks
  batch rank-1 updates into GEMMs
- Cholesky of Q^{-1}
  needed columns *are* the factor; precompute once, stable

Algorithm (per 128-col block)

Cholesky: $Q = UU^\top$

1. quantize block q: $w_q \to \hat{w}_q$
2. update the rest:

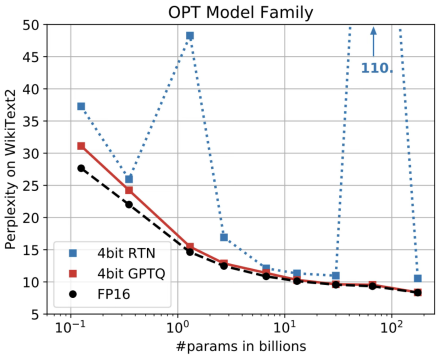
$$w_f \leftarrow w_f - Q_{ff}^{-1} Q_{fq} \left(w_q - \hat{w}_q\right)$$

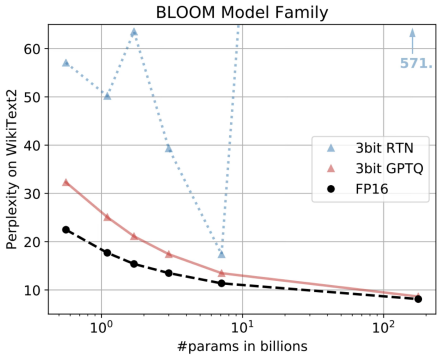
3. advance the frontier; repeat

GPTQ [arXiv:2210.17323]: fixed order, 128-col blocks, Cholesky, lazy updates.

GPTQ IV — 16 Bits for the Price of 4

- 3–4 bit ≈ negligible perplexity hit
- Marlin/Machete kernels serve W4A16
- 175B in 4-bit on a single GPU
- ∼3-4× inference speedup (A100/A6000)





GPTQ [arXiv:2210.17323] (OPT-175B); curve schematic of the paper's bits-vs-ppl trend.

Save the 1%

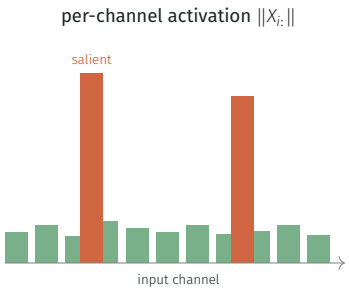

WEIGHT VECTOR
DIMENSION: N
CONTAINS BILLIONS OF WEIGHTS
THE OTHER 99%
CAN BE COMPRESSED, QUANTIZED, OR DROPPED
TOP 1%
KEEP PRECISE. THE MOST IMPORTANT.
PRESERVE AT FULL PRECISION
WEIGHT PRUNING STRATEGY
SAVE THE TOP 1%
PRESERVE WHAT MATTERS MOST.
CUT THE REST.
IMPORTANCE SCORE
$I_i = |w_i|$
IMPORTANCE
TOP 1%
INDEX (SORTED BY IMPORTANCE)
INFORMATION BOTTLENECK
SPARSE IS EFFICIENT
FOCUS ON SIGNAL

AWQ — 1% of the Channels Carry the Signal

- 0.1–1% of weight channels dominate the error
- spot them by activation magnitude
- FP16 for them fixes it — but mixed precision is ugly
- per-channel scale $s > 1$: divide salient weights by s, multiply matching activation by s

$$Wx = (W \oslash s)(s \odot x), \qquad s = (\text{act scale})^{\alpha}$$

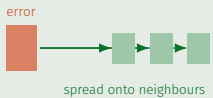
- grid-search $\alpha \in [0, 1]$ per layer



Lin et al. 2023, AWQ [arXiv:2306.00978], MLSys 2024 best paper.

Compensate vs Protect

GPTQ — compensate



2nd-order: push each column's rounding error onto the not-yet-quantized weights. *Fixes mistakes after the fact.*

AWQ — protect

Rescale the important channels so rounding never harms them. *Prevents mistakes up front.*

Both ship everywhere

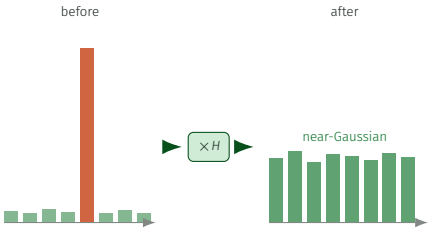
W4 PPL within noise (Llama3-8B: AWQ 6.6 vs GPTQ 6.5; 70B ≈3.3 both). Standard 4-bit kernels in vLLM / SGLang.

AWQ [arXiv:2306.00978] & GPTQ [arXiv:2210.17323]; numbers r6 B7/B8.

A permanent fix for Heavy Hitters

Heavy Hitters, Take 3: Rotate Them Into Gaussian Dust

- multiply weights (and matching activations) by an orthogonal Q
- output unchanged, but Q smears outliers across all coordinates
- distribution becomes near-Gaussian: quantize uniformly, no special channels
- Hadamard matrices: $O(n \log n)$ (fast Walsh-Hadamard)



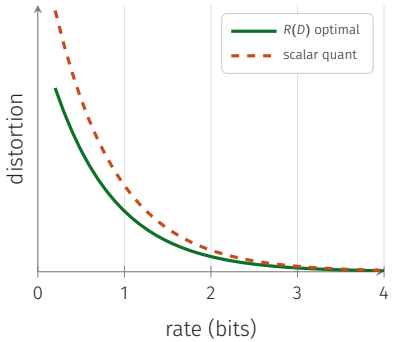
QuIP [arXiv:2307.13304], QuIP# [arXiv:2402.04396], QuaRot [arXiv:2404.00456] (W4A4KV4), SpinQuant [arXiv:2405.16406].

Teaser: Rate-Distortion Says Scalar Quant Wastes Bits

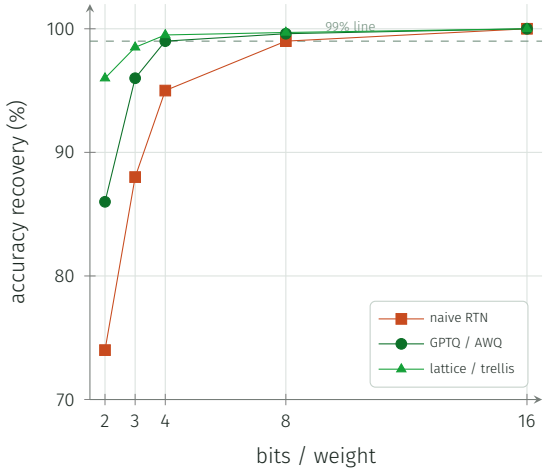
The information-theory view
(used properly on the *KV cache* in §5)

- scalar-quantizing Gaussian data wastes 0.254 bits/sample vs. optimal $R(D)$
- rotations make everything Gaussian (4.30)
- online (no-calibration) variants exist

TurboQuant does the $R(D)$ analysis properly; cashed in on the KV cache next section.



The Bits-vs-Quality Ladder for 2026



- FP8 (W8A8): effectively lossless, all scales
- INT4 / NVFP4 (W4A16): small, recoverable — the default
- 3-bit: noticeable; needs imatrix / good PTQ
- 2-bit: only with lattice/trellis (QuIP#, AQLM, QTIP) + fine-tune

Deployment rule

W4A16 cheapest for latency / single-stream; W8A8 wins async throughput.

Kurtic et al. 2024, "BF16 or Death" [arXiv:2411.02355] (>500k configs, Llama-3.1).

Weights Are Half the Story: W4A16, W8A8, W4A4

Recipe	What it speeds up	Quality cost	When
W4A16	decode bandwidth (4× less weight traffic)	∼0 loss (GPTQ/AWQ)	default, single-stream
W8A8-FP8	prefill compute + memory (∼2×)	∼lossless	H100+ production-ish
W8A8-INT8	prefill compute on older HW (A100)	small, needs SmoothQuant	INT8-only tensor cores
W4A4	prefill + decode, max compression	1–3 pt gap	research → prod

SmoothQuant migrates activation outliers into the weights via

$$s_j = \frac{\max |X_j|^{\alpha}}{\max |W_j|^{1-\alpha}}$$

so $\tilde{X} = X\,\mathrm{diag}(s)^{-1}$ and $\tilde{W} = \mathrm{diag}(s)W$ are both INT8-friendly.

Activations are the hard part

Weights quantize easily; activations carry *dynamic outliers* (LLM.int8(): emergent, ∼100×-magnitude channels appear above ∼6.7B params). W4A4 only works after rotations (QuaRot/SpinQuant) spread those outliers out.

SmoothQuant [arXiv:2211.10438]; QuaRot [arXiv:2404.00456]; SpinQuant [arXiv:2405.16406]; LLM.int8() [arXiv:2208.07339].

The Quantization Toolbox (What to Reach For)

Tool	What	Link
llm-compressor	vLLM PTQ → `compressed-tensors`	`vllm-project/llm-compressor`
GPTQModel	maintained GPTQ + AWQ / GGUF / FP8	`ModelCloud/GPTQModel`
AutoAWQ	reference AWQ quantizer	`casper-hansen/AutoAWQ`
bitsandbytes	on-the-fly 8-bit / NF4 (QLoRA)	`bitsandbytes-foundation/bitsandbytes`
unsloth	"Dynamic 4-bit" GGUF quants	`unslothai/unsloth`
llama.cpp	K-quants + imatrix; CPU/Metal/edge	`ggml-org/llama.cpp`
MLX	Apple-silicon group-wise 4/8-bit	`ml-explore/mlx`

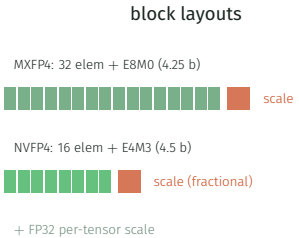
Serving rule of thumb

Serve FP8 for throughput; W4A16 (Marlin / Machete kernels) for memory-bound single-stream. Marlin ≈2.6× naive GPTQ.

The Format War is (probably) ending (slowing down) at 4 Bits (for Weights)

Native FP4, end to end

- FP-Quant / qutlass
  MR-GPTQ (GPTQ +12 Hadamard) for MX/NVFP4
  3.6× (B200) / 6× (RTX 5090) layer speedup
- NVFP4 pretraining
  12B, 10 T tokens in 4-bit ≈ FP8
- gpt-oss
  MXFP4 MoE weights (120B on one 80GB H100)



FP-Quant 2025 [arXiv:2509.23202]; NVFP4 train 2025 [arXiv:2509.25149]; NVFP4 vs MXFP4 from r4.

Weight Compression: TL;DR

Two orthogonal axes (use both)

MoE

fewer weights per token (sparse FFN).

Quantization

fewer bits per weight.

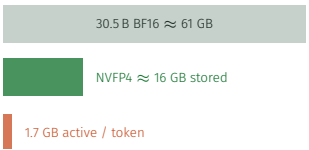
- lossless floor ≈ 4.7 bits
  below it you discard information
- GPTQ *compensates*, AWQ *protects*,
  rotation *removes* outliers
- 2-bit needs lattices / trellis

The payoff

Qwen3-30B-A3B at NVFP4

∼16 GB stored, ∼1.7 GB streamed per token.

Frontier-class quality, laptop-class memory.



30.5 B × 4.5 b ≈ 16 GB; active 3.35 B × 4.5 b ≈ 1.7 GB. r6 summary.

Part 5

KV-Cache Compression

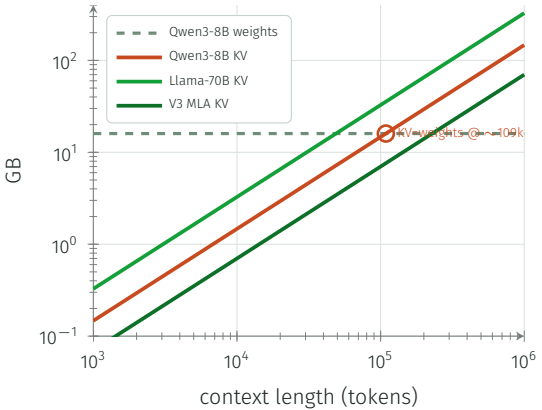
how to afford a million tokens of memory

The cache overwhelms the model

$$\text{KV bytes/token} = 2 \cdot L \cdot n_{\text{kv}} \cdot d_h \cdot b$$

Model	KB/token
Qwen3-8B (GQA)	147
Llama-3-70B (GQA)	327
DeepSeek-V3 (MLA)	70

- **1M tokens:** Qwen3-8B cache = 147 GB.
- Weights: only 16 GB BF16.
- The cache dwarfs the model.



KV by hand; DeepSeek-V2 (MLA) [arXiv:2405.04434]. All BF16, GQA 8 KV heads assumed.

Two Malicious Multipliers

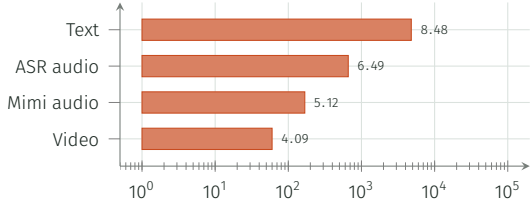
× **Batch.** KV is per-request; decode reads all of it every token.

- Batch **100** × **100k** ctx = 10^7 tokens.
- × 147 KB/token = ≈1.5 TB — 18 H100s' HBM, cache alone.
- Out of memory long before FLOPs.

The serving chain

KV size → caps batch → caps throughput → sets $/token. Shrinking KV is the lever on all four.

× **Modality.** Text rarely needs 1M tokens. Pixels and audio do.



One hour of 1 fps video at 258 tok/frame (Gemini) ⇒ a full 1M window.

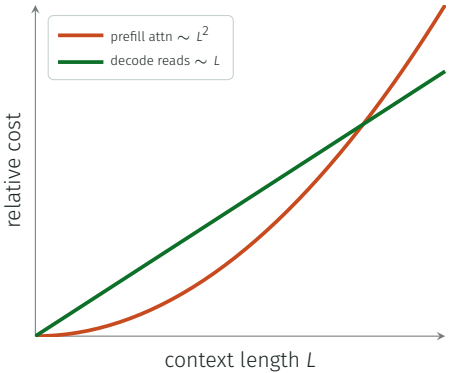
r7 §1–2; Gemini video (258 tok/frame); Mimi/Moshi (100 tok/s). Batch: 147 KB/tok, BF16 GQA.

Scaling behavior for Long Context

Prefill — quadratic. Attention $O(L^2 d)$: double context, quadruple the prompt cost.

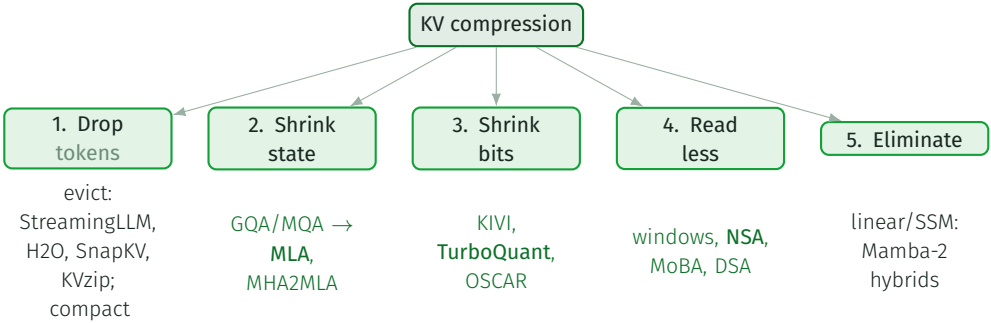
Decode — linear stream. Every new token reads the *whole* cache. Per-token latency $O(L)$, bandwidth-bound, not FLOP-bound.

Compression attacks both: fewer bytes to *store* (prefill/capacity) *and* to *stream* (decode).



r7 §1. Decode latency floor = bytes touched / bandwidth.

Five Knobs to Shrink the Cache



Knobs 1–4 are orthogonal — they multiply. Knob 5 replaces the growing cache with $O(1)$ state. We walk left to right.

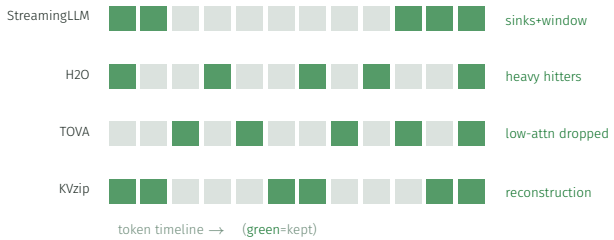
Section map. The four shrink-axes compose (closing matrix); the fifth is the radical alternative.

Drop What You Don’t Need: Eviction

Mass concentrates on sink tokens, scattered heavy hitters, the local window. Keep those, evict the rest.

- **StreamingLLM**: sinks + window ⇒ infinite streams, no finetune.
- **H2O**: keep high accumulated-attention tokens.
- **SnapKV**: prompt-end window votes for prefix.
- **TOVA**: drop lowest last-query attention. Full quality at 1/8 cache, 4.8× throughput.

KVzip: score by *reconstruction* — query-agnostic, evict once, reuse for any query. 3–4× smaller, ∼2× faster decode.

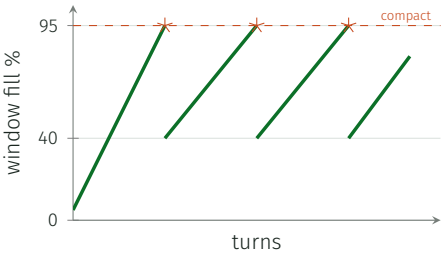


StreamingLLM [arXiv:2309.17453], H2O [arXiv:2306.14048], SnapKV [arXiv:2404.14469], TOVA [arXiv:2401.06104], KVzip [arXiv:2505.23416] `snu-mllab/KVzip`.

Compaction: Garbage-Collect the Conversation

Eviction drops *tokens*; compaction drops *turns* — same knob, one granularity up. Window fills ⇒ pause, summarize old turns, drop the raw KV, resume.

- Claude Code auto-compacts at ∼95%; one run 204k→82k tokens (−59%).
- Tiers: *micro* (stale tool output) / *full* (summarize) / *session-memory*.
- Fine for chat; dangerous for agents.



Just like a JVM GC: stop-the-world pause, lossy, unpredictable timing.

r7 §3; Claude compaction docs.

Head Surgery

Shared heads

MQA (Shazeer'19): one KV head for all queries — maximal cut, some quality loss.

GQA (Ainslie'23):

- g KV groups interpolate MQA↔MHA.
- Llama-3, Qwen3, Mistral use 8 KV heads.
- $\sim 8\times$ cut at negligible cost.

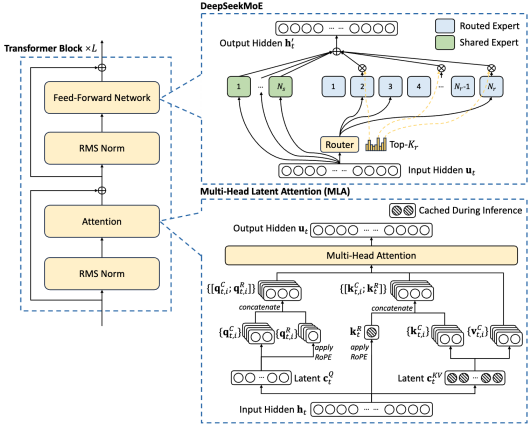
Cross-layer sharing — orthogonal:

- **CLA**: share K/V across adjacent layers.
- **YOCO**: one global cache, all cross-decoder layers; size independent of depth.

GQA is the floor. Every method here stacks on an already-GQA'd cache.

GQA [arXiv:2305.13245], MQA [arXiv:1911.02150]; cross-layer YOCO [arXiv:2405.05254] / CLA.

MLA: Low Rank Attention



Never materialize the vectors

$$
\begin{aligned}
c^{KV} &= W^{DKV} h \\
k^C &= W^{UK} c^{KV} &&= W^{UK} W^{DKV} h \\
v^C &= W^{UV} c^{KV} &&= W^{UV} W^{DKV} h \\
q^C &= W^{UQ} c^Q &&= W^{UQ} W^{DQ} h
\end{aligned}
$$

Never materialize k, v — work on c and h only. Decode touches just the 576-dim latent.

Splitting RoPE and NoPE

- Don't compress the position embeddings.
- Keep position in 64-dim k_R explicitly.
- Remainder is NoPE (compressible).

DeepSeek-V2 (MLA) [arXiv:2405.04434]: d_c=512, d_R=64, cache = 576. Latent + absorb + decoupled-RoPE, one view.

MLA: Byte Budget and Advantages

Scheme	elems/tok/layer	KB/token
MHA ($2n_h d_h$)	32,768	∼2300
GQA (8 KV)	4,096	—
MLA (d_c+d_R)	576	70

- 512 (NoPE) + 64 (RoPE) elements.
- 61 layers (V3): 70 KB/token — 93% under MHA.
- Quality *beats* MHA empirically.

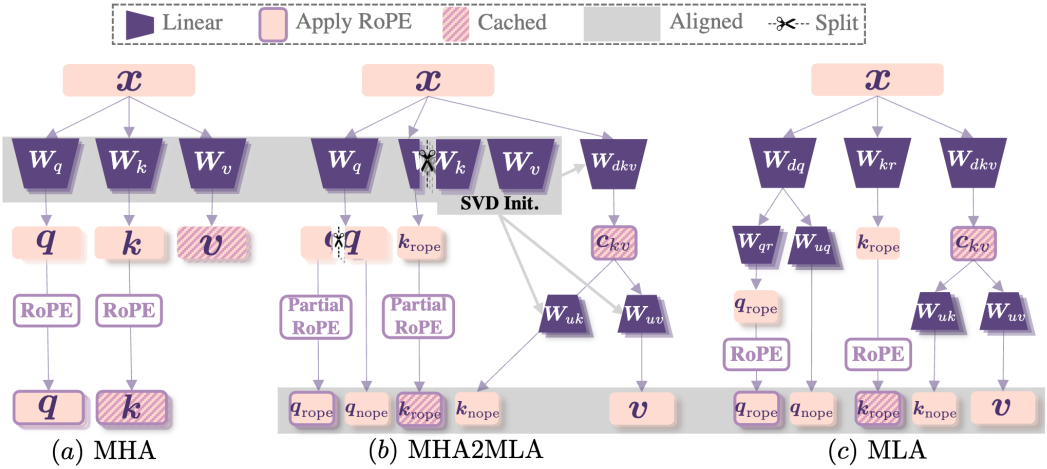
Shipped: DeepSeek-V2 → V3 → R1. At 671B/37B-active, 70 KB/token is under half of Qwen3-8B's GQA cache — on a model ∼80× bigger. 1M tokens = 70 GB, one node.

Wins: long-context, decode-heavy, capacity-bound serving on big models — if you can afford to train it.

Catch: baked in at pretraining. Post-hoc converters retrofit it (next).

DeepSeek-V2/V3 [arXiv:2405.04434]: 576 elems, 93.3% cut, "GQA 2.25 groups". TransMLA, NeurIPS'25 [arXiv:2502.07864].

MHA2MLA: Retrofit MLA onto a Finished Model



MHA2MLA: Retrofit MLA onto a Finished Model

MHA2MLA converts a trained Llama/Qwen with ∼6B tokens (0.3–1% of pretraining):

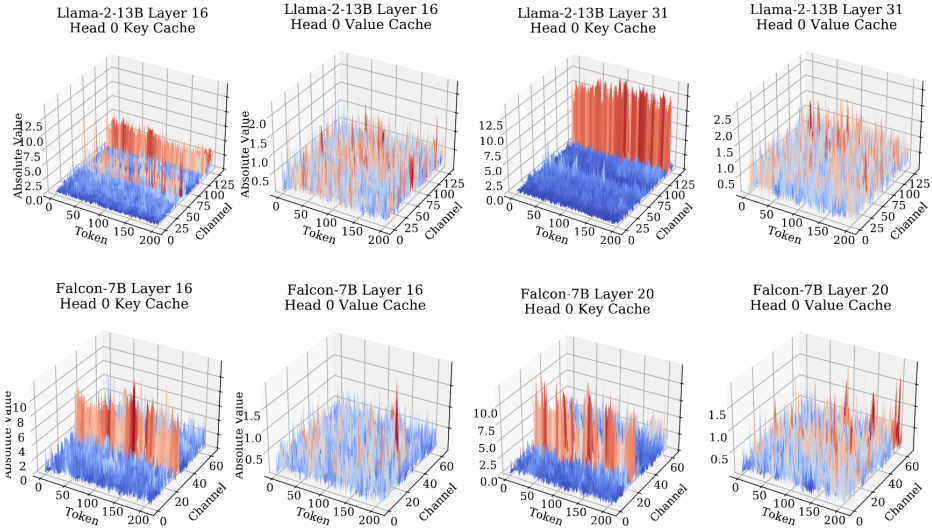
- **Partial-RoPE:** rotate only the top-r subspaces ($r{=}d_h/16$, by 2-norm); rest is NoPE.
- **Joint SVD:** factor $[\,W_{k,\text{nope}} \,\|\, W_v\,]$ *together* $\rightarrow$ one shared latent c_{kv} (∼1 pt over splitting K/V).

Base	d_{kv}	KV cut	Δacc
Llama2-7B	64	−68.75%	−0.29
Llama2-7B	32	−81.25%	−0.30
Llama2-7B	16	−87.50%	(BF16)
Llama2-13B	32	−81.25%	−0.23
SmolLM-1.7B	16	−81.25%	−1.43

81–87% less KV for $<$ 0.5 pt on big models — bigger models lose less.

Stacks with quantization. $d_{kv}{=}64$ + 4-bit HQQ $\rightarrow$ −92.19% KV, 0.5% LongBench drop.

KIVI: Quantize K and V in opposite ways

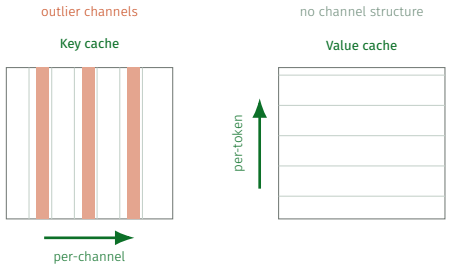


KIVI: Quantize K and V the opposite ways

Activation distribution splits in two:

- **Keys:** outliers along channels (RoPE concentrates them) ⇒ quantize K per-channel.
- **Values:** no such structure ⇒ quantize V per-token.

2-bit, asymmetric, *tuning-free*. Slice each matrix the right way.



2.6× less peak memory, up to 4× batch, 2.35–3.47× throughput, near-lossless.

KIVI [arXiv:2402.02750]; `jy-yuan/KIVI`. Also KVQuant [arXiv:2401.18079] (pre-RoPE K, 3-bit).

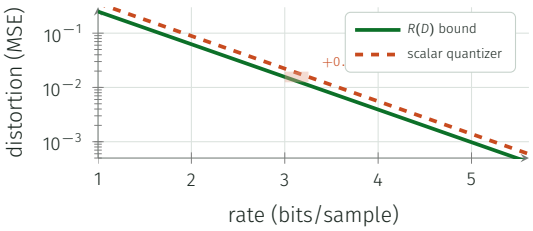
TurboQuant: Rotate to the Rate–Distortion Bound

Coordinate-by-coordinate quantization is provably wasteful. Gaussian rate–distortion bound:

$$D(R) = \sigma^2 \, 2^{-2R}.$$

A scalar quantizer sits $\frac{1}{2}\log_2 \frac{\pi e}{6} \approx 0.254$ bit/sample above it. A random rotation Gaussianizes the coordinates — a *scalar* quantizer then *reaches* $D(R)$.

Plug-in 2-bit KV. 3.5 bits quality-neutral, 2.5 bits marginal loss; MSE within $3\pi/2 \approx 2.7\times$ of the limit *at every bit-width* — online, no calibration.



The pipeline (data-oblivious, no calibration):

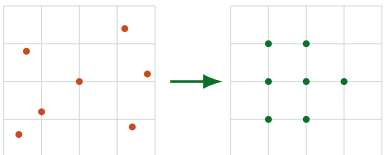
1. **Random-rotate** ⇒ near-Gaussian coordinates.
2. **Optimal scalar quantizer** per coordinate.
3. **1-bit QJL residual** to debias inner products.

TurboQuant [arXiv:2504.19874] (Zandieh et al., Google); `0xSero/turboquant`.

OSCAR: Rotate Toward What Attention Reads

INT2 gives only 4 levels. Generic Hadamard rotations (QuaRot/SpinQuant) smear error *everywhere*. **OSCAR** derives the rotation from attention statistics — pushing error where attention doesn't read.

1. Calibrate Q/K/V on a few sequences.
2. Attention-aware K/V covariance per layer.
3. Eigendecompose → fixed R_K, R_V + clips.
4. Serve INT2 (+ BF16 sink/window); fold R^{-1} into Q/out.



generic: clips badly — OSCAR: lands on grid

Metric	OSCAR INT2
Memory	∼8× smaller
Gap to BF16, Qwen3-8B	1.42 pt
Gap, Qwen3-4B-Thinking	3.78 pt
Naive INT2 rotation	≈collapse
Throughput (large batch)	up to 7×
Decode (batch-1)	up to 3×

Robust on 128k RULER-NIAH, where naive rotation collapses.

At extreme bit-widths the *basis* beats the quantizer — and it stacks on MLA/GQA.

OSCAR (Together AI, 2026) [arXiv:2605.17757]; `FutureMLS-Lab/OSCAR` r7 §6.

Read Less


NEW DIRECTIVE
READ LESS.
FOCUS ON
WHAT MATTERS.
ATTENTION
IS LIMITED
FOCUS
INFORMATION CONSUMED
WISDOM
>
VOLUME
QUALITY
OVER
QUANTITY
THE
INFORMATION
OVERLOAD
EPIDEMIC
OPTIMIZE
ACT
IMPACT
ENDLESS ARTICLES
INFINITE SCROLL
DATA DUMP
UNNECESSARY CONTEXT
VERBOSE REPORTS
READ LESS.
THINK MORE.
ACT BETTER.
TODAY:
THINK
PLAN
BUILD

Most Attention Is Local

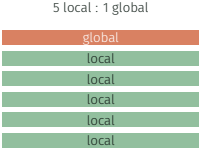
So far: shrink the cache. Sparse attention attacks a different axis — decode is bounded by bytes read, not stored. Mass is local or on a few hubs, so *read* only those.

Cost model

Full reads $O(L)$/token; sparse reads $O(k)$, $k \ll L$. Cache stays full size; the *traffic* collapses.

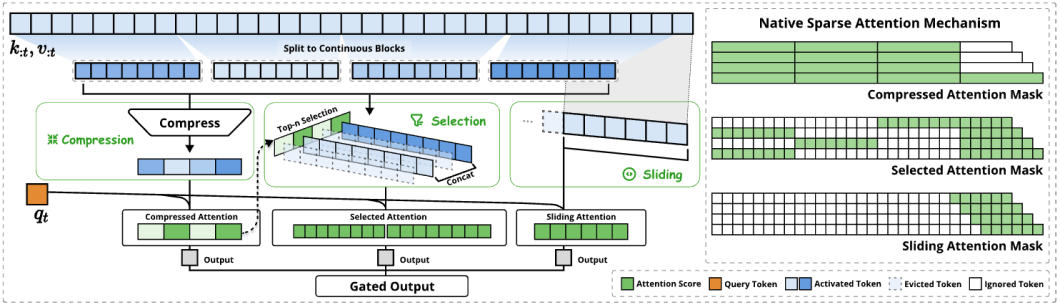
Fixed patterns (no training):

- **Sliding window** (Mistral): attend last $W \Rightarrow$ cache bounded by W.
- **Gemma-3 hybrid**: 5 local : 1 global cuts long-ctx KV from ∼60% to <15% at 32k.



r7 §7; Gemma-3 (5:1 local:global) [arXiv:2503.19786]; Mistral sliding window. Decode is a read problem, not (only) a storage problem.

NSA: Engineering three attention regimes



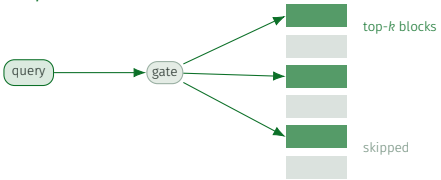
NSA matches or beats full attention while reading a fraction of the cache — mixing block-average, short-range, and block-selection.

@64k speedup: decode 11.6×, forward 9.0×, backward 6.0×.

NSA (DeepSeek) [arXiv:2502.11089]; `fla-org/native-sparse-attention`.
l=32/16, top-16 of 64-blocks, window 512; LongBench 0.469 vs 0.437.

Learned Routing, Shipped: MoBA and DSA

MoBA — MoE over *blocks*: each query routes to its top-k blocks via a gate. *Learns* where to attend; switches full↔sparse seamlessly. In production at Kimi.



DSA (DeepSeek-V3.2-Exp) — a lightning indexer (FP8, tiny) scores query↔past relevance, then a fine-grained top-k feeds only $k \ll L$ into attention $\Rightarrow O(Lk)$.

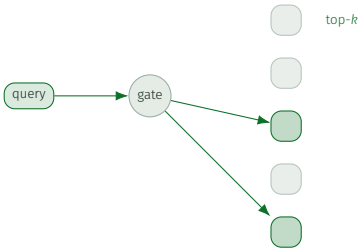
DSA halved DeepSeek's long-context API prices (∼50% cheaper decode vs V3.1). Algorithmic efficiency shows up on the invoice.

MoBA (Moonshot) [arXiv:2502.13189] `MoonshotAI/MoBA`; DSA (DeepSeek-V3.2-Exp) [arXiv:2512.02556] — halved long-context API prices.

MoE over Heads and Tokens

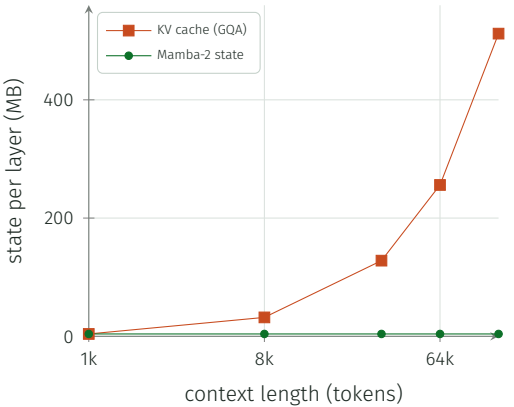
NSA, MoBA, DSA share one idea: route attention like an MoE. A gate picks a sparse subset of {heads, blocks, tokens}.

- **MoSA**: each head an expert, expert-choice routing over its own k tokens — $O(k^2+T)$ vs $O(T^2)$. Only sparse variant to beat the dense baseline (up to 27% better perplexity at matched compute).
- **CLSA / YOIO**: share the routing index across layers atop YOCO $\Rightarrow$ 7.6× decode, 17.1× throughput @128k.



MoSA [arXiv:2505.00315] (beats dense); CLSA/YOIO [arXiv:2606.06467].

Getting rid of KV entirely via State Space Models



- KV grows linearly: 512 MB/layer at 128k (8-head GQA, d_{head}=128).
- SSMs keep a fixed recurrent state. Mamba-2: ∼4 MB/layer, *constant* in N, decode $O(1)$/step.
- Entire context compressed into one hidden state.

SSD duality

Mamba-2: a class of SSMs ≡ linear attention (state $d_{\text{state}} \times d_{\text{head}}$) — run as fast matmuls, not a slow scan.

Dao & Gu 2024, Mamba-2 / SSD [arXiv:2405.21060]: fixed-size state, structured state-space duality.

Best of both worlds

Model	Linear:full ratio	Linear primitive / claim
Qwen3-Next-80B-A3B	3:1	gated DeltaNet (75% linear)
MiniMax-01	7:1	lightning attn; 100% NIAH @4M (claimed)
IBM Granite 4	9:1	Mamba-2; >70% memory cut
NVIDIA Nemotron-H	∼12:1	Mamba-2; only ∼8% attention layers
AI21 Jamba	1:7	Mamba-2 hybrid; ∼4 GB KV @256k

Pure linear models have fuzzy recall: ablate the few full-attention layers and needle-in-a-haystack drops to ∼0. A handful of full layers carry retrieval; the linear majority carries cheap throughput.

Mamba-2 [arXiv:2405.21060]; Qwen3-Next; ablation: [arXiv:2510.19861] (full layers carry recall).

What Stacks with What

	shrink state	shrink bits	read less	drop
shrink state (GQA/MLA)	—	✓	✓	✓
shrink bits (quantize)	✓	—	✓	✓
read less (sparse)	✓	✓	—	▬
drop (evict/compact)	✓	✓	▬	—

They multiply. Shrink the per-token state (MLA), then its bits (quant), then how much you read (sparse) — different axes, they compose.

Watch the overlap. Sparse-read and eviction both bet on the same redundancy (▬); doing both can double-count. MHA2MLA + 4-bit is a proven pair.

2026 defaults: GQA/MLA + FP8 KV + prefix cache; long-context: add sparse or evict.
The fifth knob, linear/SSM, *replaces* the cache rather than stacking.

Composition after r7 §11; MHA2MLA + Int4 from r9 (proven −92%).

The 1M-Token Bill of Materials

Qwen3-8B at 1M tokens: 147 GB of KV — impossible on one GPU.

Step	factor	KV
GQA baseline (1M)	—	147 GB
+ MLA-class latent	÷8	18 GB
+ 4-bit KV quant	÷4	4.6 GB
+ sparse reads (NSA)	÷10 reads	∼0.5 GB/tok traffic

Storage: a few GB. *Traffic* — which sets decode latency — falls another order of magnitude.

The whole section in one line

MLA (×8) + 4-bit KV (×4) + NSA reads (×10 fewer) ⇒ 1M tokens is feasible on one node.

Structural shrinks the state, numerical shrinks its bits, sparse shrinks the reads, semantic drops what you never needed. Four orthogonal knobs, turn them together.

Composed back-of-envelope from this section's verified factors (r7/r9). Storage vs traffic separated.

Part 6

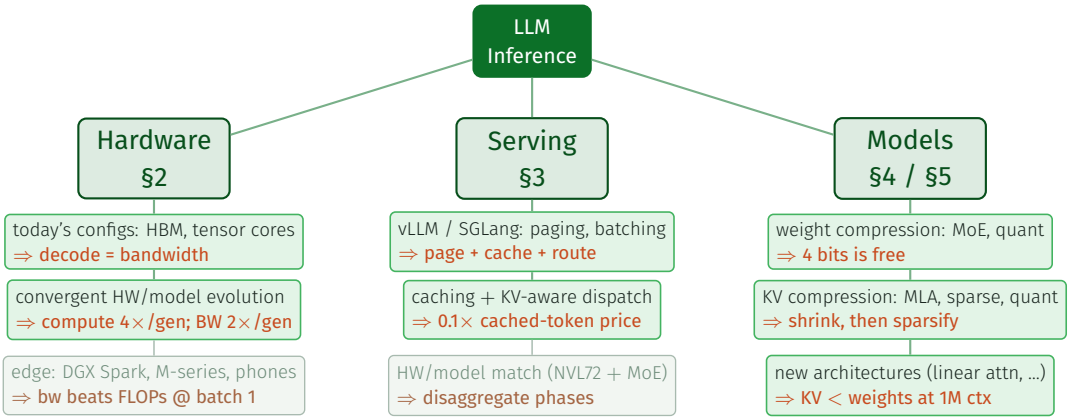
Wrap-Up & Resources

the map revisited, numbers to remember, and where to go next

6

BosonAI

The Map, Revisited



Seven Numbers to Remember

number	what it means	where
1 PF/s vs 3.4 TB/s	H100 BF16 peak compute vs. bandwidth	§2
300 FLOP/byte	ridge point: ops per byte fetched	§1–2
∼4× compute; ∼2× BW / gen	intensity pressure keeps rising	§2
150 KB / token	Qwen3-8B KV per stream (BF16)	§1, §5
∼500× energy	DRAM access vs. FP32 mul (Horowitz)	§2
0.1×	Anthropic cached-read vs. uncached	§3
∼400× gap	DGX Spark prefill vs. decode tok/s	§1–2

A fast byte beats a fast FLOP — and moving fewer bytes beats buying more hardware.

Canonical numbers — sources in the respective sections; cache price: Anthropic docs.

Every technique here moves fewer bytes.

- **MoE:** fewer weights per token.
- **Quantization:** a quarter the bytes.
- **PagedAttention:** no KV fragmentation.
- **Prefix cache:** read prompts once.
- **MLA:** low-rank KV, 8× smaller.
- **KV quant:** 2–4-bit, near-lossless.
- **Sparse attn:** ∼10× fewer entries.
- **Disaggregation:** each phase to memory.

The chip does not care *how* you saved bytes — architecture, compression, or policy.
Fewer bytes in ⇒ faster token out.

Synthesis of §1–§5. The meta-lesson: optimize data movement, not compute.

Tutorials & Courses

- **How to Scale Your Model** — Austin et al. (DeepMind 2025): roofline → parallelism → LLaMA 3 KV.
- **Ultra-Scale Playbook** — HF / nanotron: 3D parallelism, memory budgets, throughput at scale.
- **Stanford CS336** — Hashimoto & Liang (2026): LM from scratch; systems & scaling lectures.
- **GPU MODE** — 40+ lectures: CUDA → FlashAttention → quant; 6.2k★.
- **Transformer Inference Arithmetic** — kipply: KV sizing, bandwidth-bound decode; latency without experiments.
- **Making DL Go Brrrr** — Horace He: compute vs. bandwidth vs. overhead; operator fusion.
- **LLM Inference Optimization** — Lilian Weng: distillation, quant, pruning, sparsity — one post.
- **Accelerating Generative AI II** — PyTorch: `compile` → INT8 → spec → INT4 → TP; ∼10× in <1k lines.

URLs current June 2026. Surveys: Zhou et al. 2024 [arXiv:2404.14294]; Li et al. 2024 [arXiv:2412.19442].

Key Papers, by Tutorial Section

§2 Hardware / formats

- Shrivastava et al. 2025, exponent concentration [arXiv:2510.02676]
- He et al. 2025, DFloat11 [arXiv:2504.11651]

§3 Serving

- Kwon et al. 2023, vLLM / PagedAttn [arXiv:2309.06180]
- Zheng et al. 2023, SGLang [arXiv:2312.07104]
- Qin et al. 2024, Mooncake [arXiv:2407.00079]
- Zhong et al. 2024, DistServe [arXiv:2401.09670]

§4 Weight Compression

- Frantar et al. 2022, GPTQ [arXiv:2210.17323]
- Lin et al. 2023, AWQ [arXiv:2306.00978]
- Tseng et al. 2024, QTIP (trellis) [arXiv:2406.11235]

§5 KV Compression

- DeepSeek-AI 2024, MLA / DeepSeek-V2 [arXiv:2405.04434]
- Ji et al. 2025, MHA2MLA [arXiv:2502.14837]
- Yuan et al. 2025, NSA [arXiv:2502.11089]
- Kim et al. 2025, KVzip [arXiv:2505.23416]
- Together AI 2026, OSCAR [arXiv:2605.17757]
- Liu et al. 2024, KIVI (2-bit KV) [arXiv:2402.02750]
- Zandieh et al. 2025, TurboQuant [arXiv:2504.19874]

Surveys

- Zhou et al. 2024, inference efficiency [arXiv:2404.14294]
- Li et al. 2024, KV cache (TMLR) [arXiv:2412.19442]

arXiv IDs verified June 2026.

Code: Engines, Kernels, & Compression Tools

Serving Engines

- `vllm-project/vllm` — PagedAttn; 200+ models; 82.6k★
- `sgl-project/sglang` — RadixAttn, structured output; 28.9k★
- `NVIDIA/TensorRT-LLM` — NVIDIA-optimized; multi-node; 13.9k★
- `ai-dynamo/dynamo` — disaggregated serving, KV routing; 7.2k★
- `ggml-org/llama.cpp` — CPU/Metal/CUDA edge; 116k★
- `ollama/ollama` — one-command local runner; 174k★
- `ml-explore/mlx` — Apple Silicon unified memory; 26.9k★

KV Cache

- `LMCache/LMCache` cross-engine KV reuse; 8.5k★
- `kvcache-ai/Mooncake` disaggregated KV, RDMA; FAST 2025 best paper
- `snu-mllab/KVzip` query-agnostic 3–4× KV compression
- `NVIDIA/kvpress` — plug-in KV for HF Transformers

Weight Compression

- `vllm-project/llm-compressor` weight, activation & KV quant
- `ModelCloud/GPTQModel` — GPTQ/AWQ/FP8
- `IST-DASLab/FP-Quant` — MXFP4/NVFP4 export

Stars as of June 2026 (r8). Kernels: `Dao-AILab/flash-attention`, `flashinfer-ai/flashinfer`.

What We Skipped (On Purpose)

- **Multi-GPU parallelism** (TP/PP/EP)
  whole topic of its own.
  Ultra-Scale Playbook, `NVIDIA/Megatron-LM`
- **Pruning & distillation**
  shrink before you quantize.
  compression survey [arXiv:2404.14294]
- **Kernels & compilation**
  CUDA graphs, `torch.compile`.
  `gpu-mode/lectures`
- **Structured decoding**
  constrained / grammar output.
  `mlc-ai/xgrammar`
- **Multi-LoRA serving**
  many adapters on one base model.
  S-LoRA [arXiv:2311.03285]
- **Ring / context-parallel attention**
  split one long sequence across nodes.
- **Audio / video / realtime**
  streaming, full-duplex speech.
  see the §1 “what we skip” links

Each a rabbit hole that composes with everything here. Our through-line: single-node decode physics.

Curated pointers; all orthogonal to the bandwidth story this tutorial tells.

Thank You

Efficiency in LLMs —
Hardware, Serving, and Compression

Alex Smola

Boson AI

smola@boson.ai

Machine Learning Summer School · 2026

Slides + sources: http://alex.smola.org/posts/45-mlss-efficiency/main.pdf