

Inference Optimization

KV-Cache · Flash Attention · Quantization · Distillation · Speculative Decoding

The inference challenge

Two bottlenecks in LLM inference

Memory

Model weights:

2 bytes/param
(bfloating16)

70B model = 140 GB

Plus KV-cache, activations,
framework overhead

A100 GPU = 80 GB

Latency

Autoregressive:
one token
at a time

Each token =
full forward pass
through all layers

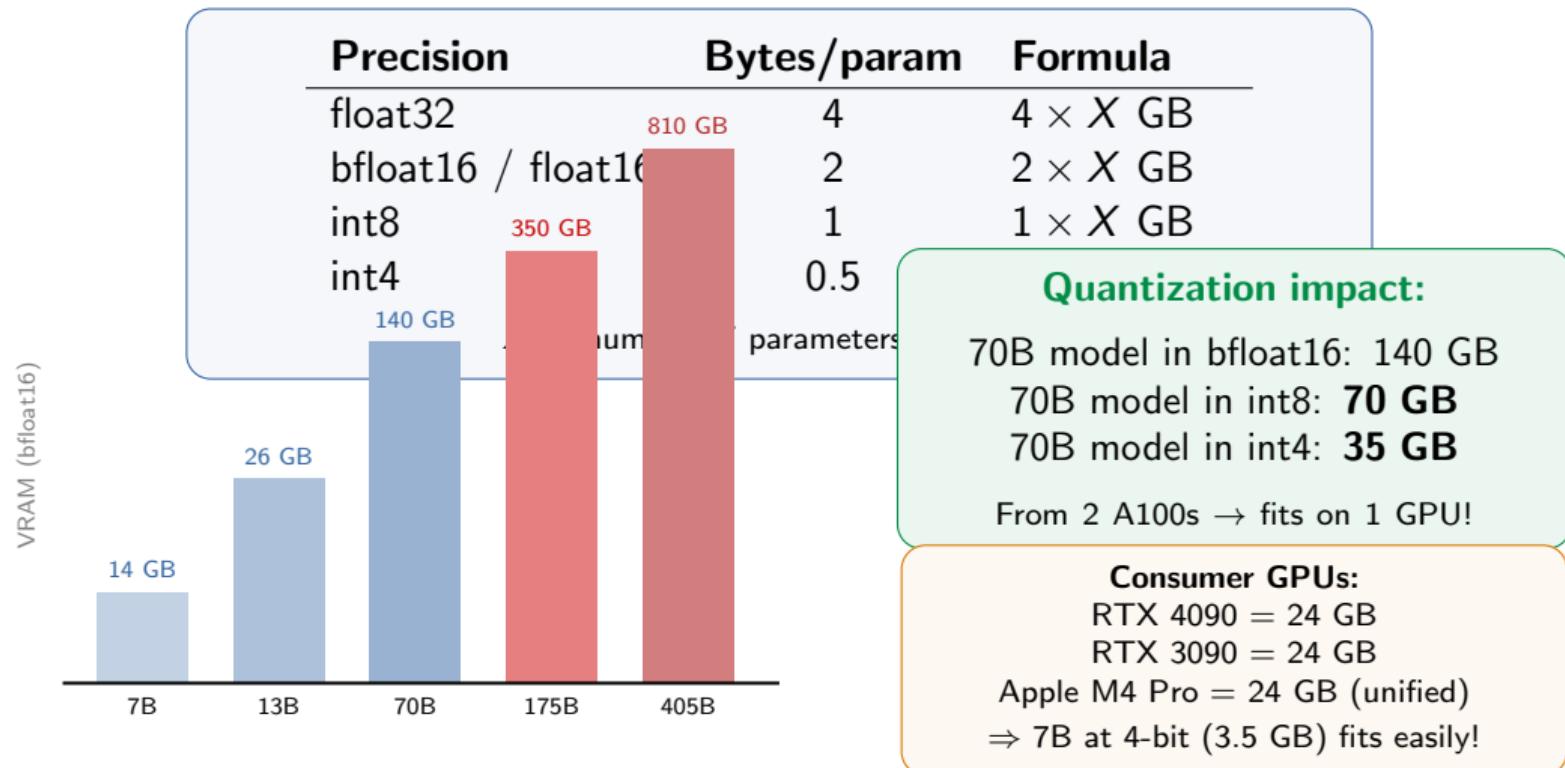
Memory-bandwidth bound,
not compute-bound

GPT-4: ~100 ms per token

$$\text{Total VRAM} = \underbrace{\text{Model Weights}}_{\text{dominant for short seq}} + \underbrace{\text{KV-Cache}}_{\text{dominant for long seq}} + \text{Activations} + \text{Overhead}$$

This lecture: techniques to reduce memory, increase throughput, and lower latency

Memory arithmetic



Part I

Attention Optimization

KV-Cache, MQA/GQA, Flash Attention

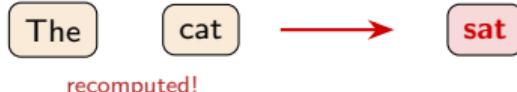
KV-Cache — the redundancy problem

Without caching: every step recomputes K, V for ALL previous tokens

Step 1:



Step 2:



Step 3:



Wasted work:

Step t recomputes
 $K_1, V_1, \dots, K_{t-1}, V_{t-1}$
from scratch

Total: $O(n^3)$ for n tokens

Key insight: in causal attention, K_i and V_i for token i **never change** once computed — they depend only on x_1, \dots, x_i , not on future tokens.

Q is always fresh (only the current token asks a “question”).

K, V for past tokens are fixed — **cache them!**

Why not cache Q? Because we only ever need q_t (the current token's query), never past queries.

KV-Cache — the solution

With KV-Cache: only compute K, V for the NEW token, then concatenate

Step t :

Cached: $K_{1:t-1}, V_{1:t-1}$
(from previous steps)

New:
 K_t, V_t, q_t

Full: $K_{1:t}, V_{1:t}$
Attention: $q_t \times K_{1:t}^T$

$$K \leftarrow [K_{\text{cache}} ; K_{\text{new}}], \quad V \leftarrow [V_{\text{cache}} ; V_{\text{new}}]$$

$$O_t = \text{softmax}\left(\frac{q_t \cdot K^T}{\sqrt{d_k}}\right) \cdot V$$

Only q_t is $[1 \times d]$; K, V are $[t \times d]$ from cache

Without cache

With cache

Compute per step $O(t \cdot d^2)$ recompute all $O(d^2)$ new token only

Total for n tokens $O(n^2 \cdot d^2)$ $O(n \cdot d^2)$

Memory Transient Grows linearly

Benchmark (200 tokens, M4 CPU): without
cache ~ 5.3 s, with cache ~ 1.1 s $\Rightarrow \sim 5 \times$ speedup

Two phases of inference: prefill vs. decode

Phase 1: Prefill

Process entire prompt
at once (in parallel)

Build the initial KV-cache
for all prompt tokens

Compute-bound

(lots of matrix multiplies)

1000-token prompt →
process all 1000 in one pass
Time: ~50–200 ms

Phase 2: Decode

Generate one token
at a time (sequential)

Append each new K, V
to the cache

Memory-bandwidth-bound

(reading weights dominates)

500-token response →
500 sequential forward passes
Time: ~5–50 seconds

The decode phase is the bottleneck: each
step reads *all* model weights from memory
but only produces *one* token. GPU arithmetic units
are mostly idle ⇒ **memory-bandwidth-bound.**
KV-Cache speeds up decode · Flash Attention helps pre-
fill · Quantization helps both (fewer bytes to read)

KV-Cache memory cost

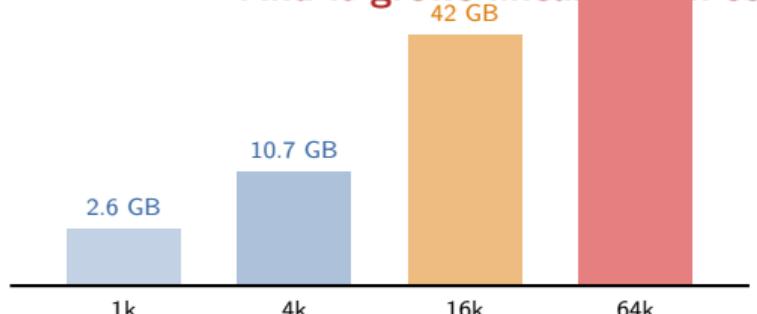
$$\text{KV-cache (bytes)} = 2 \times \text{seq_len} \times n_{\text{layers}} \times n_{\text{heads}} \times d_{\text{head}} \times \text{bytes_per_value}$$

Factor 2 = one for K, one for V (stored per layer, per head)

Example: LLaMA-2 70B, seq_len = 4

$$2 \times 4096 \times 80 \times 64 \times 128 \times 2 \overset{170\text{ GB}}{\approx} 10.7 \text{ GB}$$

And it grows linearly with sequence length



Solution: share K/V heads

MHA: each head has own K, V
(standard — large cache)

MQA: all heads share 1 K, V
(40× smaller cache!)

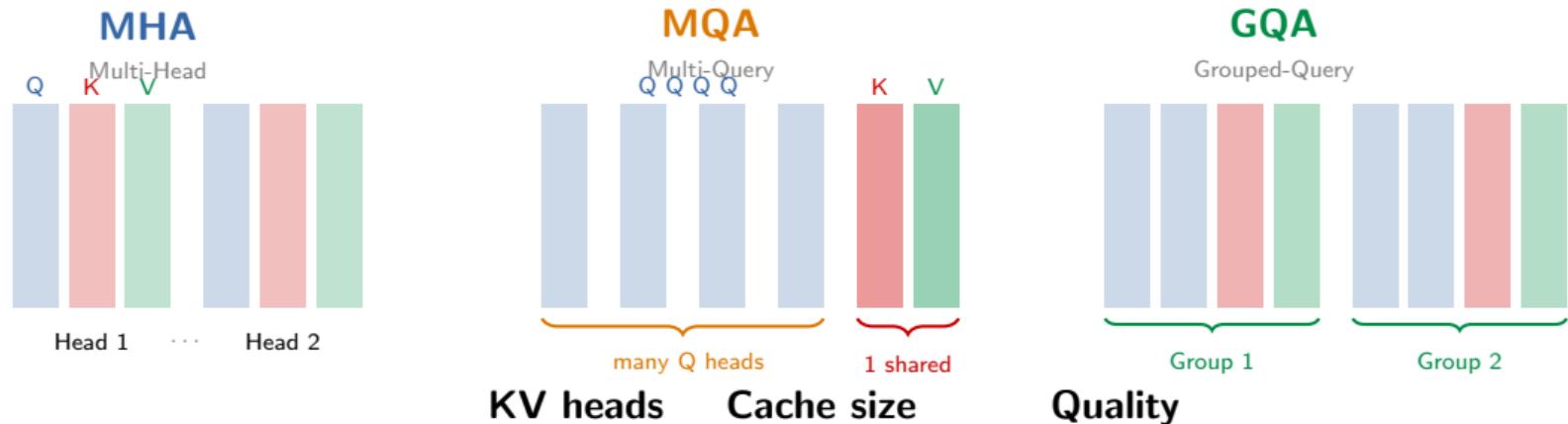
GQA: groups of heads share K, V
(good balance: quality + speed)

GQA: LLaMA-2/3, Mistral

MQA: Falcon, PaLM

MHA vs. MQA vs. GQA

How attention heads share Key and Value projections



LLaMA 2/3: GQA with 8 KV heads (32 Q heads)

Mistral: GQA with 8 KV heads

Flash Attention

Standard attention materializes the full $N \times N$ matrix QK^T in GPU memory (HBM).

At $N = 16,000$ with 40 heads in bfloat16: ~ 19 GB just for this matrix!

~~Tile Q, K, V into blocks~~

Standard Attention

↓
Store $N \times N$ in HBM

↓
Softmax

↓
Multiply by V

Memory: $O(N^2)$

Flash Attention

Process blocks in fast SRAM

↓
Accumulate with
online softmax

↓
Never store full $N \times N$

Memory: $O(N)$

Exact same output

Numerically identical
to standard attention

IO-aware

Minimizes slow
HBM reads/writes

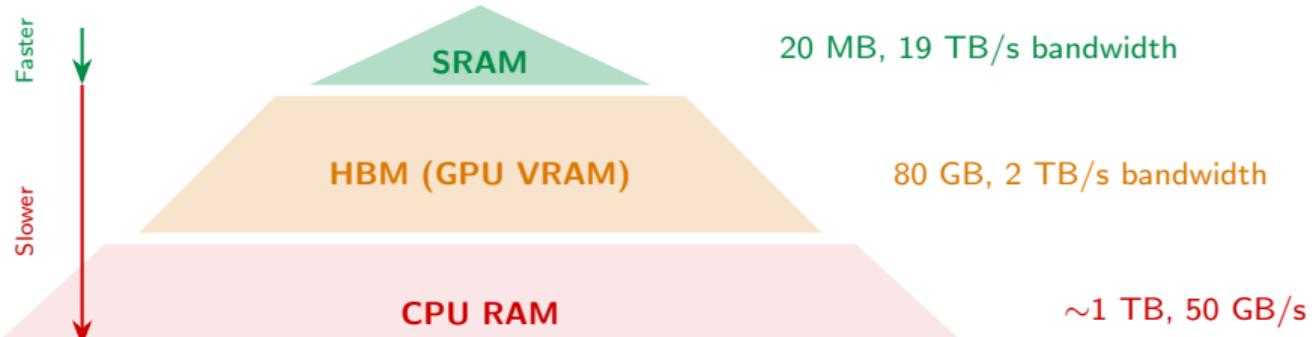
Drop-in

No model changes;
just swap the kernel

**Always
use this.
No reason
not to.**

Why Flash Attention works: the GPU memory hierarchy

GPU memory is NOT a single pool — there's a speed/size trade-off



Standard attention: compute QK^T ($N \times N$ matrix) in HBM → read/write to slow memory

Flash Attention: tile into blocks that fit in SRAM → 10× less HBM traffic

The bottleneck is *memory bandwidth*, not compute. Flash Attention does more FLOPs but is faster!

Standard: $O(N^2)$ HBM reads
 $N = 16k, 40$ heads:
~19 GB transferred

Flash: $O(N^2d/M)$ HBM reads
 $M = \text{SRAM size}$
~2 GB transferred

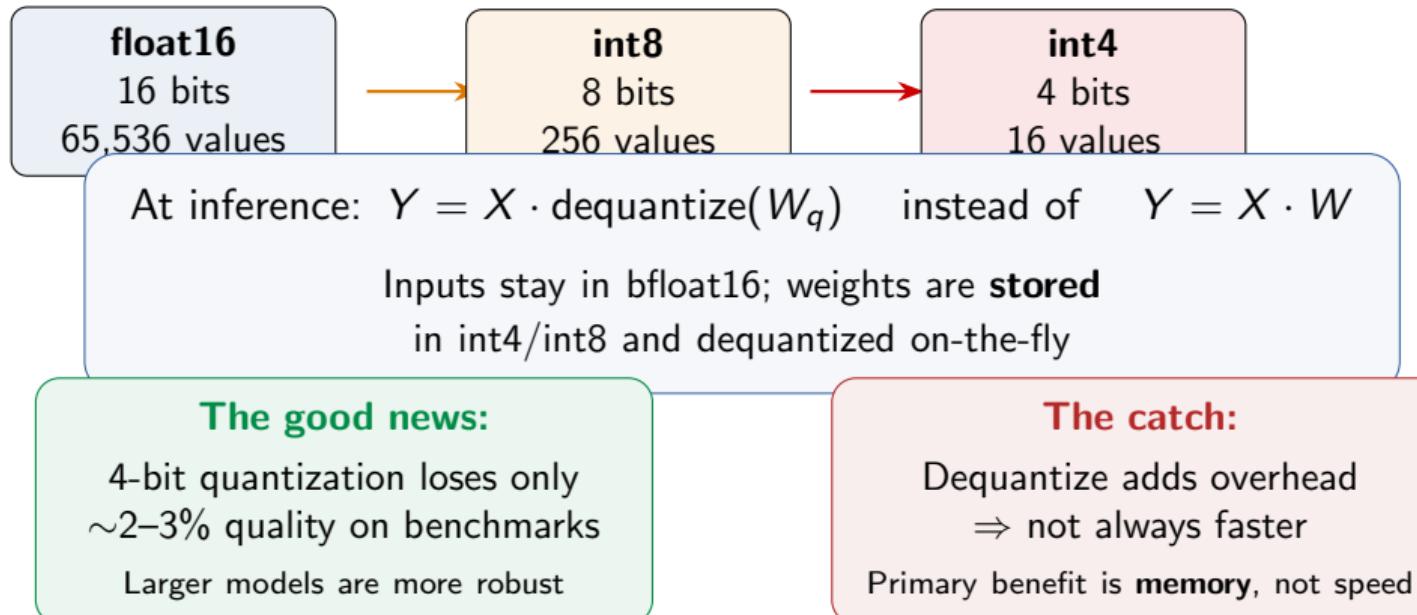
Part II

Quantization

Reducing precision to reduce memory

What is quantization?

Represent weights in fewer bits: float16 → int8 → int4



Key trade-off: memory ↓ accuracy (slightly) ↓ speed: depends on implementation

The math of quantization

Absmax (symmetric) quantization:

$$X_q = \text{round}\left(\frac{127}{\max|X|} \cdot X\right), \quad \hat{X} = \frac{\max|X|}{127} \cdot X_q$$

Scalar example: $X = [1.2, -0.5, 3.0, -2.1]$, $\max|X| = 3.0$, $s = 3.0/127 = 0.0236$

Example:

$$X_q = \text{round}([50.8, -21.2, 127.0, -88.9]) = [51, -21, 127, -89]$$

Zero-point (asymmetric) quantization:

$$X_q = \text{round}\left(\frac{X - \min X}{s}\right) + z, \quad s = \frac{\max X - \min X}{2^b - 1}$$

Block quantization:

Don't use one scale for the whole tensor — split into blocks of 64–128 values, each with its own scale factor.

Used by bitsandbytes (NF4)

so zero point
asymmetry

NormalFloat4 (NF4):

Assumes weights are normally distributed. Quantization levels are optimally spaced for $\mathcal{N}(0, 1)$.

Used in QLoRA — better than uniform int4

When to quantize: PTQ vs. QAT

PTQ

When: after training
Post Training Quantization

Data: small calibration set
(or none at all)

Speed: minutes to hours

Quality: good at 4–8 bits;
degrades at 2–3 bits

Examples: GPTQ, AWQ,
bitsandbytes

QAT

When: during training
Quantization-Aware Training

Data: full training data

Speed: full training cycle

Quality: better at extreme
compression (2–3 bits)

Examples: QLoRA (partial),
BitNet, 1-bit LLMs



In practice: PTQ dominates for LLMs (nobody wants to retrain a 70B model).
QAT is used for extreme compression or when training from scratch.

Quantization methods compared

Method	Calibration	Bits	Speed	Best for
bitsandbytes	None	4, 8	Slower	Quick experiments, QLoRA
GPTQ	Required	2–8	2× faster	Production deployment
AWQ	Required	3–4	Fast	Production, multi-modal

bitsandbytes

Zero-shot: quantize on load.
No calibration data.
Works with any nn.Linear.
Ideal for fine-tuning (QLoRA).

GPTQ

Uses Hessian (2nd order) to find optimal rounding.
Layer-by-layer quantization.
175B in ~4 GPU hours.

AWQ

Protects salient weights (top 0.1–1% by activation).
Per-channel scaling.
MLSys 2024 Best Paper.

Quick rule: bitsandbytes to try → GPTQ/AWQ to deploy → QLoRA to fine-tune

GPTQ — optimal weight quantization

Problem: given weight matrix W , find W_q that minimizes layer output change:

$$\min_{W_q} \|WX - W_qX\|_2^2$$

Use the **Hessian** $H = 2XX^T$ to know which weights are sensitive

Algorithm Process model **layer by layer** (no full backprop needed)

For each layer, quantize weights **column by column**

Use inverse Hessian H^{-1} to determine **optimal rounding**

After quantizing each column, **update remaining weights** to compensate

175B model:

Quantized in ~4 GPU hours
3–4 bits, negligible quality loss
Fits on one CPU

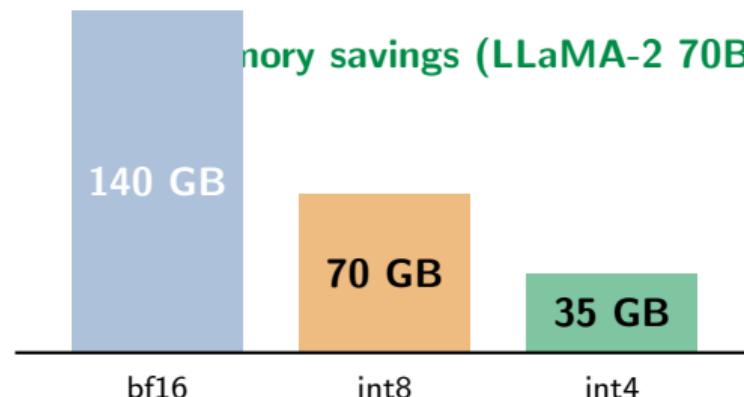
Speedup:

3.25× on A100
4.5× on A6000
(with ExL-

Quantization benchmarks

Quality (LLaMA-2, Open-LLM Leaderboard)

Method	7B avg	13B avg
Original (fp16)	54.32	58.66
bitsandbytes 4-bit	53.40	56.90
GPTQ 4-bit	53.23	57.56
Degradation	~2%	~2–3%



Speed (LLaMA-2 13B, A100)

Method	tok/s	VRAM
fp16	27.1	29.2 GB
GPTQ 4-bit	29.7	10.5 GB
bnn 4-bit	19.2	11.0 GB

Memory savings (LLaMA-2 70B, bfloat16 baseline):

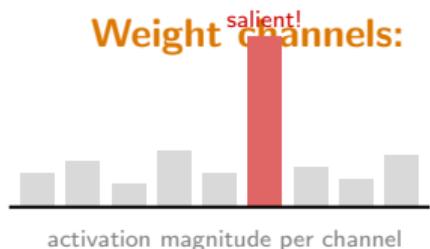
Key takeaways:

- 4-bit loses ~2% quality
- Saves ~75% memory
- GPTQ is fastest (optimized kernels)
- bnn is easiest (no calibration)

Larger models degrade less

AWQ — protecting salient weights

Key insight: not all weights are equally important.
Only **0.1–1%** of weights are “salient” — and they correspond to channels with **large activation magnitudes**. Protecting them drastically reduces quantization error.



1. Observe which channels have large activations (on calib data)
2. Apply **per-channel scaling**: multiply salient weights by s before quantization
3. Divide activations by s at

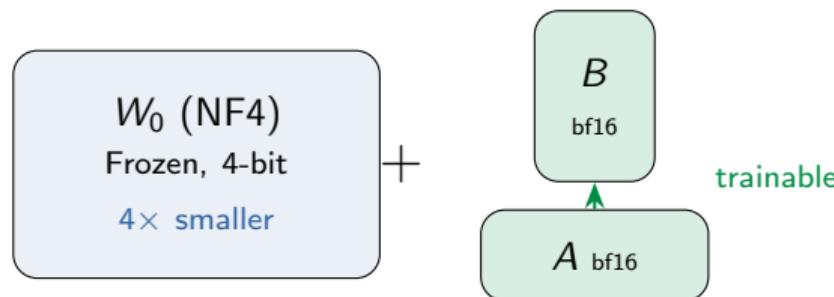
$$Q(w \cdot s) \cdot \frac{x}{s} \approx w \cdot x \quad \text{but with } \mathbf{much \ less} \text{ quantization error}$$

The scaling s protects salient channels from aggressive rounding

Result: 3–4 bit quantization with quality matching GPTQ, sometimes better.
MLSys 2024 Best Paper · Works with instruction-tuned and multi-modal models.

QLoRA — quantized fine-tuning

QLoRA (Dettmers et al., 2023): combine 4-bit quantization with LoRA
⇒ fine-tune a 65B model on a **single 48 GB GPU!**



Three innovations:

1. NF4 quantization

optimal 4-bit for
normal weights

2. Double quantization

quantize the quan-
tization scales too

3. Paged optimizers

offload optimizer states to CPU

Forward: dequantize W_0 to bf16 on the fly, compute $h = W_0x + BAx$

Backward: gradients only flow through A and B (not W_0) ⇒ tiny memory

65B on 1 GPU (48 GB):

Base: 130 GB (impossible)

QLoRA: ~33 GB

+ LoRA overhead

Quality:

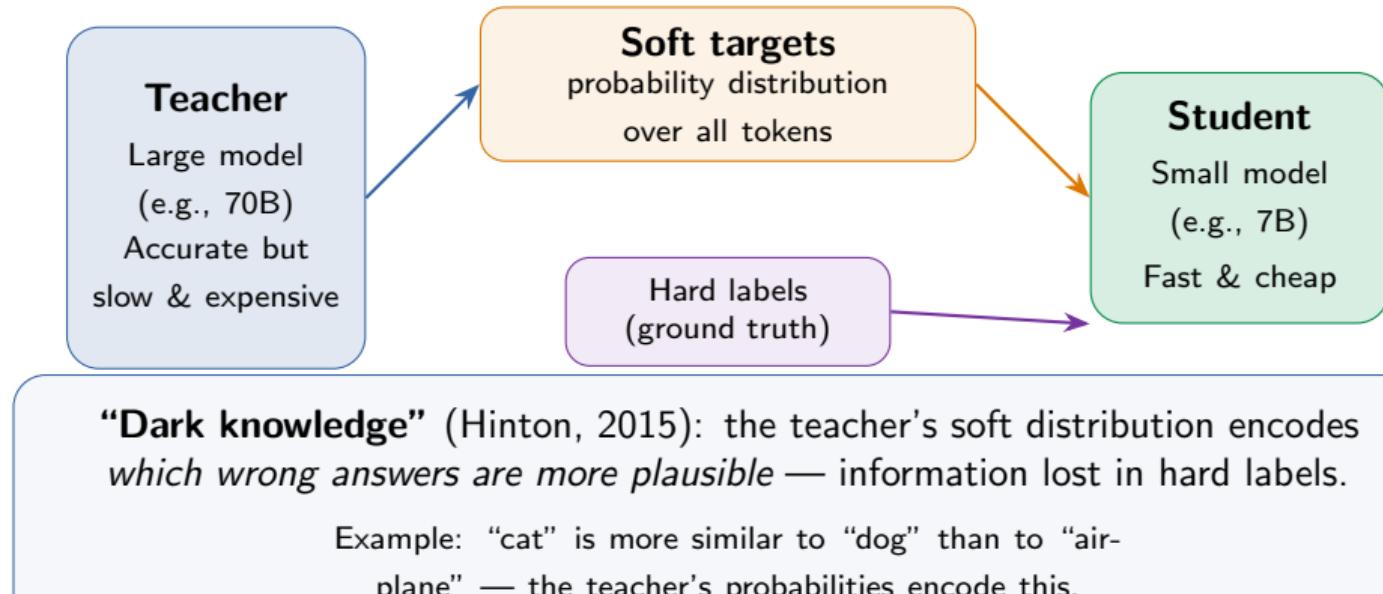
Matches 16-bit full fine-tuning
on benchmarks (chatbot arena)

Part III

Knowledge Distillation

Training a small model to mimic a large one

Knowledge distillation — teacher and student

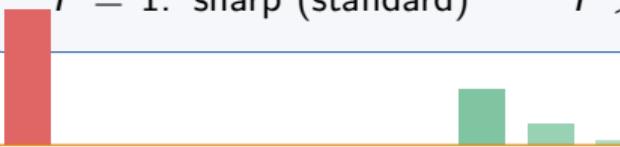


Analogy: a master chef doesn’t just say “this is tomato soup” — they explain “it’s similar to gazpacho, a bit like minestrone, nothing like chocolate cake.”

The distillation loss

Temperature-scaled softmax: $p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$

$T = 1$: sharp (standard) $T \gg 1$: soft (reveals dark knowledge)



Combined distillation loss:

$$\mathcal{L} = \alpha \cdot \underbrace{\text{CE}(y, \sigma(z_s; T=1))}_{\text{hard label loss}} + (1 - \alpha) \cdot T^2 \cdot \underbrace{D_{\text{KL}}(\sigma(z_t; T) \parallel \sigma(z_s; T))}_{\text{soft target loss}}$$

→ teacher logits

→ student logits,

Why T^2 ?

Soft targets produce gradients scaled by $1/T^2$.

Multiplying by T^2 rebalances.

T^2 corrects gradient magnitudes

Typical values:

$T = 2-20$

$\alpha = 0.1-0.5$

Lower T when student is much smaller

Notable distilled models

Student	Teacher	Compression	Quality	Method
DistilBERT	BERT (110M)	1.7× (66M)	97% GLUE	KD + cosine loss
TinyLlama	LLaMA (7B)	6.4× (1.1B)	Competitive	Pre-train 3T tokens
Minitron 4B	LLaMA 3.1 (8B)	2× (4B)	+16% vs scratch	Prune + distill
DeepSeek-R1 7B	R1 (671B MoE)	96×	Beats 32B	800k reasoning samples
Llama 3.1 8B	Llama 405B	50×	≈ teacher	Synthetic data KD

DistilBERT

6 layers (vs. 12)
1.6× faster
Triple loss: KD + CE
+ cosine hidden states

DeepSeek-R1 7B

Fine-tune Qwen-2.5 7B
on 800k curated samples
Outperforms QwQ-32B
on AIME math

NVIDIA Minitron

Prune 50% neurons
then distill
40× fewer training tokens
Optimal: prune →
KD → quantize

Key insight: a distilled 7B can outperform a non-distilled 32B — the teacher's knowledge compresses remarkably well into a smaller architecture.

Part IV

Speculative Decoding

Lossless acceleration via draft-then-verify

Speculative decoding — the idea

Key observation: verifying γ tokens in parallel (one forward pass) takes
≈ the same time as generating 1 token from the large model

Standard Decoding

Large model generates
one token at a time

Each token = 1 full
forward pass

n tokens = n passes



Speculative Decoding

Small **draft** model
guesses γ tokens fast

Large **verifier** checks
all γ at once

Accept correct ones,
resample wrong ones

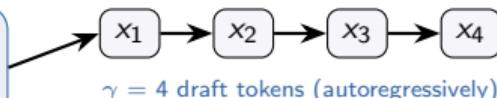
Lossless: the output distribution is **mathematically identical**
to the target model. Zero quality degradation!

Typical speedup: 2–2.5×

· Draft model must share the same tokenizer

Speculative decoding — the algorithm

Draft model
(small, fast)



$\gamma = 4$ draft tokens (autoregressively)

Target model
(large, slow)



Score ALL γ tokens in ONE forward pass

Step 3:



Accept left to right on first rejection resample from correction distribution

Accept token x_i with prob:

$$\alpha(x_i) = \min\left(1, \frac{p(x_i)}{q(x_i)}\right)$$

p = target, q = draft

On rejection, resample from:

$$p'(x) \propto \max(0, p(x) - q(x))$$

Fills in probability mass
the draft model missed

Why speculative decoding is lossless

Case 1: $q(x) \leq p(x)$
(draft underestimates)

Accept with prob
 $\frac{p(x)}{q(x)} \cdot q(x) = p(x)$
(always accept — but rejection
from other tokens can trigger
resampling from p')

n is exactly $p(x)$ — here's why:

Case 2: $q(x) > p(x)$
(draft overestimates)

Accept with prob $\frac{p(x)}{q(x)} < 1$
Excess mass is trimmed.
Correction distribution
adds back what's missing.

$$P(\text{output} = x) = q(x) \cdot \alpha(x) + \beta \cdot p'(x) = p(x)$$

where $\beta = \sum_x \max(0, p(x) - q(x))$ is the total rejection probability

Intuition: “refund for overpayment, supplement for deficiency.” perfectly matches p

Expected tokens per target forward pass: $\mathbb{E}[\text{tokens}] = \sum_{i=0}^{\gamma} \prod_{j=1}^i \alpha_j$

When draft model closely matches target: most tokens accepted \Rightarrow up to $(\gamma + 1) \times$ speedup

Chinchilla 70B: $2\text{-}2.5 \times$ speedup

Whisper large: $2.2 \times$ speedup (Chen et al., 2023)

Speculative decoding — worked example

Generating after “The capital of France is”: $\gamma = 3$ draft tokens

Token 1: “Paris”

Draft q : 0.85
Target p : 0.90

$$\alpha = \min(1, \frac{0.90}{0.85}) = 1.0$$

ACCEPT

Token 2: “,”

Draft q : 0.70
Target p : 0.75

$$\alpha = \min(1, \frac{0.75}{0.70}) = 1.0$$

ACCEPT

Token 3: “a”

Draft q : 0.40
Target p : 0.15

$$\alpha = \min(1, \frac{0.15}{0.40}) = 0.375$$

$$\text{Random } r = 0.62 > 0.375$$

REJECT

Correction: resample from $p'(x) \propto \max(0, p(x) - q(x))$

$$p(\text{"which"}) - q(\text{"which"}) = 0.50 - 0.20 = 0.30 \quad (\text{largest positive diff})$$

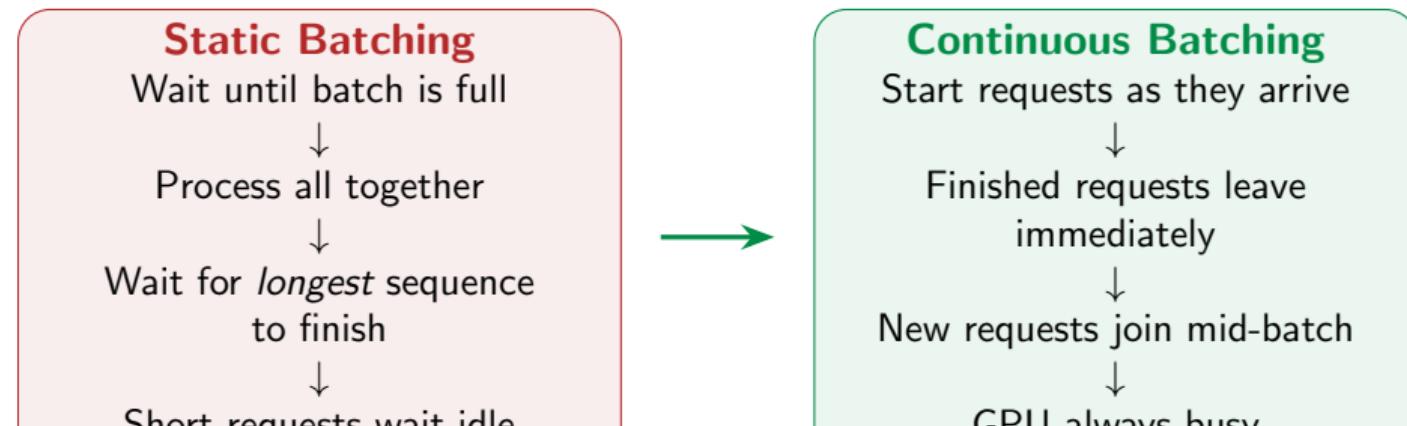
$$p(\text{"known"}) - q(\text{"known"}) = 0.10 - 0.05 = 0.05 \Rightarrow \text{Sample "which" or "known"}$$

Result: 2 accepted + 1 resampled = 3 tokens from 1 target forward pass

Standard decoding: 3 target passes. Speculative: 1

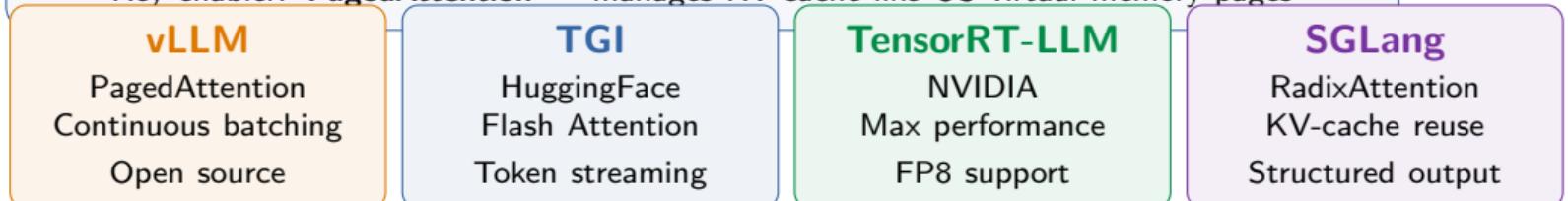
target pass + 3 draft passes ≈ 1.3 target passes

Production serving: continuous batching



Throughput improvement: up to 23× higher than static batching (vLLM paper)

Key enabler: **PagedAttention** — manages KV-cache like OS virtual memory pages



Inference optimization techniques compared

Technique	What it does	Memory	Speed	Quality
KV-Cache	Cache K, V across steps	↑ (linear in seq)	~5× ↑	Identical
MQA/GQA	Share K/V heads	KV-cache ↓	Faster	Near-identical
Flash Attn	Tile attention in SRAM	$O(N)$ vs $O(N^2)$	Faster	Identical
Quantization	Lower bit precision	50–75% ↓	Depends	~2–3% ↓
Distillation	Train smaller model	Model size ↓	Much faster	3–10% ↓
Spec. Decoding	Draft + verify	Slight ↑	2–2.5× ↑	Identical

These techniques compose! A modern deployment might use:

Flash Attention + GQA + KV-Cache + 4-bit GPTQ + Speculative Decoding

Each addresses a different bottleneck: compute, memory, bandwidth, latency

Example stack: LLaMA 3 70B + GQA

(built-in) + Flash Attention 2 + AWQ 4-bit

⇒ runs on a single A100 80 GB with 4k context at ~30 tokens/s

Practical guide

What should I use?

Always use (free wins):

- KV-Cache (always on)
- Flash Attention 2
- GQA (if architecture supports)

Need maximum speed?

- ⇒ **Distill** to a smaller model
- + quantize the distilled model
- Prune → KD → Quantize

Quick experiment?

- ⇒ **bitsandbytes** 4-bit

One line: `load_in_4bit=True`

Model too big for my GPU?

- ⇒ **Quantize** (4-bit AWQ/GPTQ)

70B → 35 GB, fits on 1 GPU

Want free speedup?

- ⇒ **Speculative decoding**
- 2× faster, zero quality loss
- Needs a matching draft model

Production deployment?

- ⇒ **GPTQ/AWQ** + vLLM

Optimized kernels, continuous batching

Questions?

Next: RAG — Retrieval-Augmented Generation