Efficient Inference

Lecture 23 · ES 667: Deep Learning

Prof. Nipun Batra
IIT Gandhinagar · Aug 2026

Learning outcomes

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

1. Explain why LLM inference is **memory-bound** (not compute-bound).
2. Compute the **KV-cache size** for Llama 70B at 32k context.
3. Describe **paged attention** and why vLLM needed it.
4. Pick a **quantization level** (FP16 / INT8 / INT4 / INT2) for a deployment.
5. Explain **FlashAttention** tiling in 2 sentences.
6. Describe **speculative decoding** and its accept-rate dependency.

Where we are

Training a 70B LLM costs ~$100M. But that's done *once*. **Inference** runs every request, every user, every day — and it's where models earn their keep.

> REFERENCE
>
> Today maps to Chip Huyen's blog posts, HF inference docs, Dao 2022 (FlashAttention), Hinton 2015 (distillation), Leviathan 2022 (speculative decoding).

Four questions:

1. Why is LLM inference **memory-bound**, not compute-bound?
2. What is the **KV-cache** and how do we manage it?
3. What is **quantization** and how low can we go?
4. What are **FlashAttention** and **speculative decoding**?

PART 1

Prefill vs decode

Two phases · two bottlenecks

Reading vs writing · the inference analogy

KEY IDEA

Think of LLM inference like **reading a chapter** then **writing a summary**.

Prefill = reading the chapter all at once · all words processed in parallel.

Decode = writing the summary one word at a time · each new word depends on all previous, so it's strictly sequential.

That's why decode is much slower per token than prefill · we can't parallelize across the future. Almost every inference optimization (KV cache, speculative decode, FlashAttention) targets the decode phase.

Prefill vs decode · the kitchen analogy

INTUITION

A chef preparing food.

- **Banquet (prefill)** · entire 10-course order arrives at once. Bottleneck = how fast they can chop, fry, plate **in parallel**. **Compute-bound.**
- **Single dish (decode)** · need rare ingredients in sequence — run to pantry, chop, run again. Bottleneck = the **pantry runs**, not the cooking. **Memory-bound.**

Decode · why it's memory-bound

For one token of decode on a 70B model the GPU must:

1. **Fetch all 70B weights** from HBM. At 2 bytes (BF16) = **140 GB**.
2. **Fetch the KV-cache** for the context (>10 GB at 32k context).
3. **Do the math** · matrix–vector products. **Tiny** compared to the data movement.

   NVIDIA A100 · ~1.5 TB/s HBM bandwidth → moving ~150 GB takes ~0.1 s. The math finishes in a fraction of that. **The GPU spends most of its time waiting for data.**

Prefill is different · the whole prompt is processed together as a big **matrix–matrix** product → the GPU's FLOPs are saturated. Compute-bound.

PHASE	BOTTLENECK
Prefill	**compute**
Decode	**memory**

Modern inference servers (vLLM, TGI) optimize the two phases separately.

PART 2

The KV-cache

Where most of the memory pressure lives

The KV-cache explained

KV-cache · reuse past attention states instead of recomputing every step

WITHOUT CACHE · RECOMPUTE ALL K, V EVERY TOKEN

step 4: K_1, V_1 K_2, V_2 K_3, V_3 K_4, V_4 ← all recomputed!

Cost scales as $O(N^2)$ · generating N tokens takes N^2 attention computations total.

WITH CACHE · STORE PAST K, V; ONLY COMPUTE THE NEW ONE

step 1: K_1, V_1 cache: $[K_1, V_1]$

step 2: reuse K_1 new K_2 cache: $[K_1, K_2]$

step 3: reuse K_1 reuse K_2 new K_3 cache: $[K_1, K_2, K_3]$

step 4: reuse K_1 reuse K_2 reuse K_3 new K_4 cache: $[K_1, K_2, K_3, K_4]$

Cost is **O(N)** · compute one new K, V per step. Attention still attends to all N.

KV-CACHE MEMORY

$$M = 2 \cdot L \cdot H \cdot d_h \cdot T \cdot B \cdot \text{bytes}$$

Llama 70B · L=80 · H=64 · d_h=128

T=32k context · B=1 · bf16 = 2 bytes

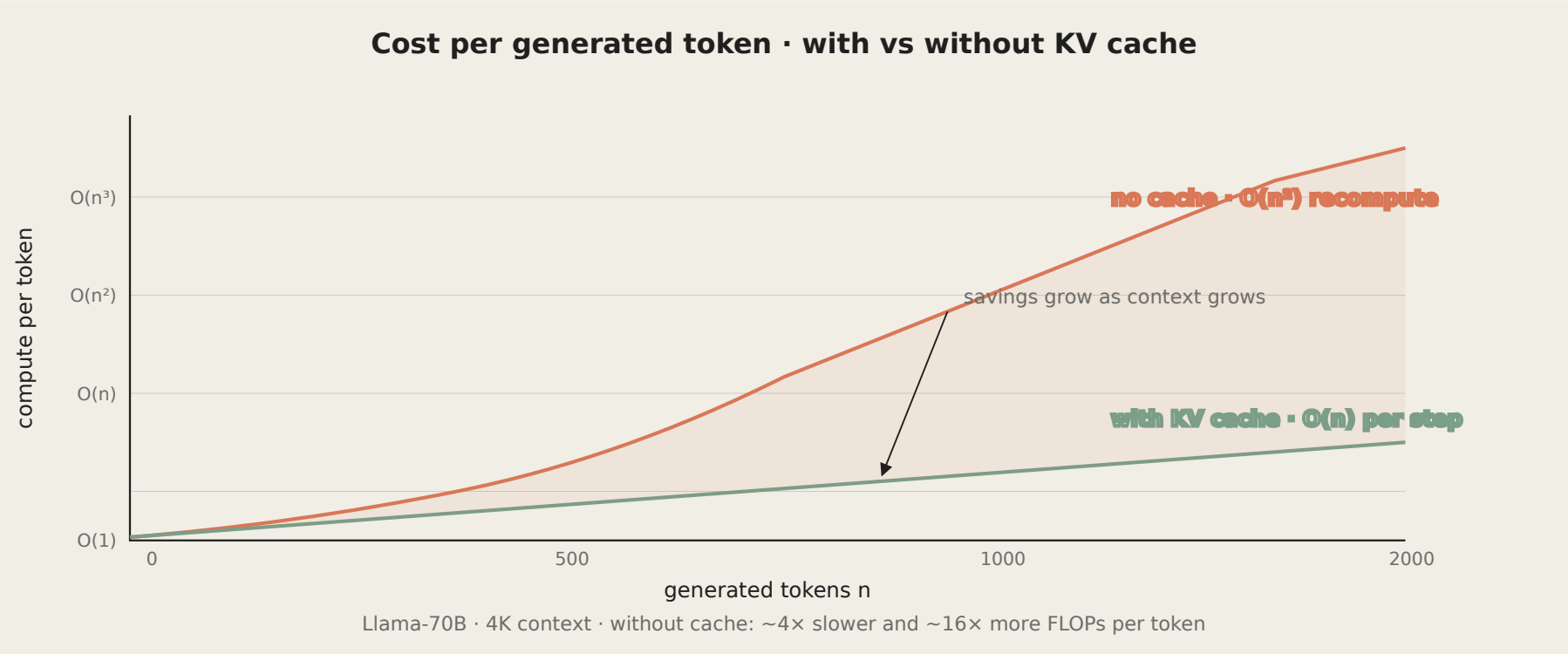
$$M = 2 \cdot 80 \cdot 64 \cdot 128 \cdot 32k \cdot 1 \cdot 2$$

$M \approx 84$ GB per request

Larger than the weights themselves.

*Modern LLM inference is **memory-bound on the KV-cache** — not compute-bound. GQA/MQA (L15), quantization, and paged attention all exist to shrink it.*

Cost with vs without cache



Why caching KV works

Observation · attention at step t computes $Q_t K_{1:t}^\top$ · we need **every past K**, but K_i doesn't change once token i is generated.

KEY IDEA

The V vectors have the same property. So cache K and V as you go — each new token does **O(1) new computation for K/V** and **O(t) for the attention dot-product**, instead of O(t²) re-doing everything.

Memory grows linearly with context, but saves an order of magnitude in compute. The reason KV-cache is the first thing *any* LLM inference stack implements.

KV-cache math · Llama 70B

DERIVATION

$$M = 2 \cdot L \cdot H_{\text{kv}} \cdot d_h \cdot T \cdot B \cdot \text{bytes}$$

- L · layers (80)
- H_{kv} · KV heads (8 with GQA — see L15)
- d_h · head dim (128)
- T · context length (32k)
- B · batch size (1)
- bytes per element (2 for BF16)

$$M = 2 \cdot 80 \cdot 8 \cdot 128 \cdot 32{,}000 \cdot 1 \cdot 2 \approx 10.5 \text{ GB}$$

With MHA (64 heads, no GQA) that would be 84 GB — **larger than the weights themselves.**

Paged attention · the parking-lot analogy

KEY IDEA

Naive memory allocation reserves a **bus-sized parking spot** for every vehicle, even scooters · huge waste.

Paged attention has only **standard car-sized spots** ("pages"). A bus uses several; a scooter uses one. Pack many more vehicles in the same lot.

For LLMs · "vehicles" = ongoing requests, each needing variable-length KV cache. Pages let us serve many short prompts in the memory that previously held one long-prompt KV cache. ~4× throughput in vLLM.

Paged attention · vLLM's big idea

Kwon et al. 2023 · **vLLM** paper.

Problem · KV-cache memory is allocated contiguously; long-context requests waste space.

Solution · **paged attention** — split the KV-cache into fixed-size pages, managed like virtual memory. Each request uses only as many pages as it needs.

> IN PRACTICE
>
> vLLM and TGI both use paged attention. **~4× throughput gain** over naive implementations in production LLM serving in 2026.

PART 3

Quantization

Run bigger models on smaller hardware

The quantization ladder

Quantization ladder · 8× memory savings, almost no quality loss

FP32 · 32 BITS
70B model · 280 GB · training default
1.00×

FP16 · 16 BITS (IEEE HALF)
140 GB · limited range, training tricky
0.50×

BF16 · 16 BITS (BRAIN FLOAT)
140 GB · FP32 range, FP16 size · 2026 default
0.50×

INT8 · 8 BITS
70 GB · negligible quality loss
0.25×

INT4 (NF4)
35 GB
0.125×

WHAT YOU LOSE AT EACH STEP

FP32 → BF16 · essentially **zero** quality loss, enables 2× larger models on same hardware.
BF16 → INT8 → NF4 · <1% accuracy drop with per-channel scaling + GPTQ/AWQ calibration · & AWQ > GPTQ > naive at low bits.

INT8 · the map-scale analogy

INTUITION

You have a satellite image of a city with precise GPS for everything (FP32). To make a tourist map (INT8), you can't keep all that precision.

1. **Find the scale** · "1 inch on the map = 1 mile in reality." This scale s is the only key piece of info.
2. **Convert** · map all real locations onto paper using s.

Quantization · find scale s to map FP32 weights into the small range $[-127, 127]$.

INT8 · derive the formula step by step

Goal · convert FP32 weights to 8-bit integers in $[-128, 127]$.

Step 1. Find max absolute value in the channel: $\max(|w|)$.
Step 2. Map the largest weight to the largest integer (127):
$\max(|w|) = s \cdot 127 \;\Rightarrow\; s = \max(|w|)/127$
Step 3. Quantize each weight:
$w_{\text{int8}} = \text{round}(w/s)$
Step 4. Reconstruct at inference: $\hat{w} = w_{\text{int8}} \cdot s$.

Store the int8 array + a single FP32 scale s per channel.

INT8 · worked numeric

Convert $w = [0.5, -0.2, 0.0, -1.0]$ from FP32 to INT8.

1. **Max abs.** $\max(|w|) = 1.0$.
2. **Scale.** $s = 1.0/127 \approx 0.007874$.
3. **Quantize.**

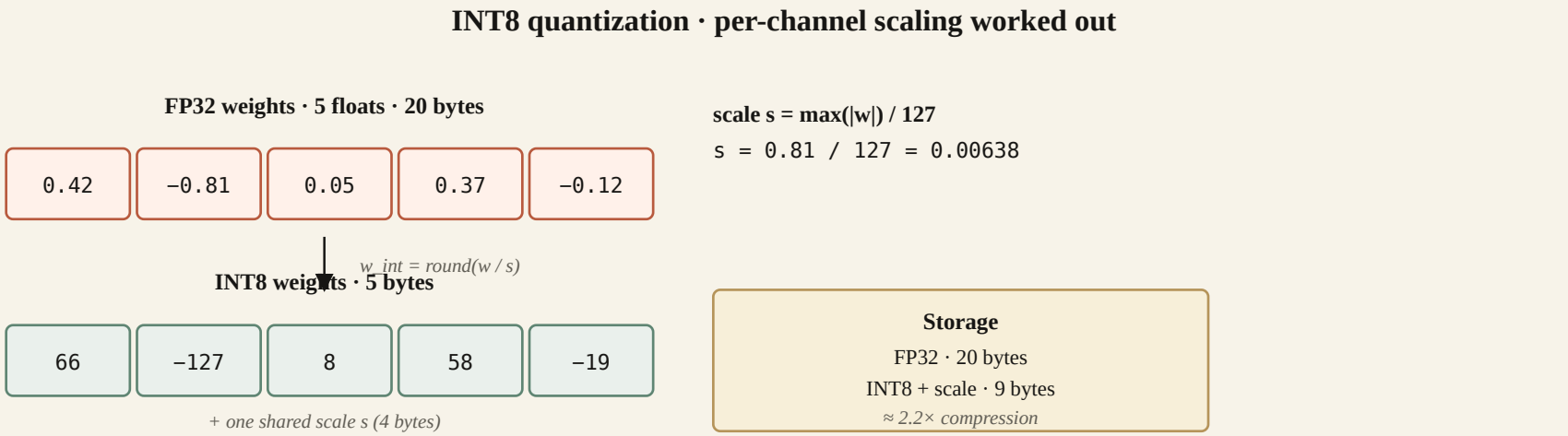
- $0.5/s \approx 63.5 \to \mathbf{64}$
- $-0.2/s \approx -25.4 \to \mathbf{-25}$
- $0.0/s = 0 \to \mathbf{0}$
- $-1.0/s = -127.0 \to \mathbf{-127}$

Stored · 4 INT8 (4 bytes) + scale FP32 (4 bytes) = **8 bytes** vs 16 bytes (4× FP32). **50% saving.**

Reconstruction. $64 \cdot 0.007874 = 0.5039$ (vs original 0.5). Max error ~0.001 — quality drop barely measurable.

PyTorch · `torch.quantization.quantize_dynamic` or `bitsandbytes`.

Quantization · arithmetic in pictures



Worked example · quantize a single weight row

DERIVATION

Channel · 5 weights · `w = [0.42, -0.81, 0.05, 0.37, -0.12]`

Step 1. $s = \max(|w|)/127 = 0.81/127 \approx 0.00638$

Step 2. $w_{\text{int}} = \text{round}(w/s) = [66, -127, 8, 58, -19]$

Step 3. Stored · the 5 INT8 values (5 bytes) plus one float scale s (4 bytes) · 9 bytes total instead of 20 bytes for FP32.

Reconstruction at inference · $\hat{w} = w_{\text{int}} \cdot s \rightarrow [0.421, -0.810, 0.051, 0.370, -0.121]$. Max error ~ 0.001. Roundoff is small; quality drop on benchmarks barely measurable.

INT4 and below · GPTQ / AWQ

At 4 bits per weight, naive quantization breaks. Two successful tricks:

- **GPTQ** (Frantar 2022) · post-training per-layer quantization that minimizes reconstruction loss.
- **AWQ** (Lin 2023) · protects the ~1% of weights that activate on important inputs.

Both work well at 4-bit; AWQ edges out GPTQ at extreme compression (3-bit, 2-bit).

IN PRACTICE

2026 practical recipe · AWQ 4-bit quantization for any LLM you're running on consumer hardware. `exllamav2` or `vLLM` both support it.

PART 4

FlashAttention

Rewrite attention for modern GPUs

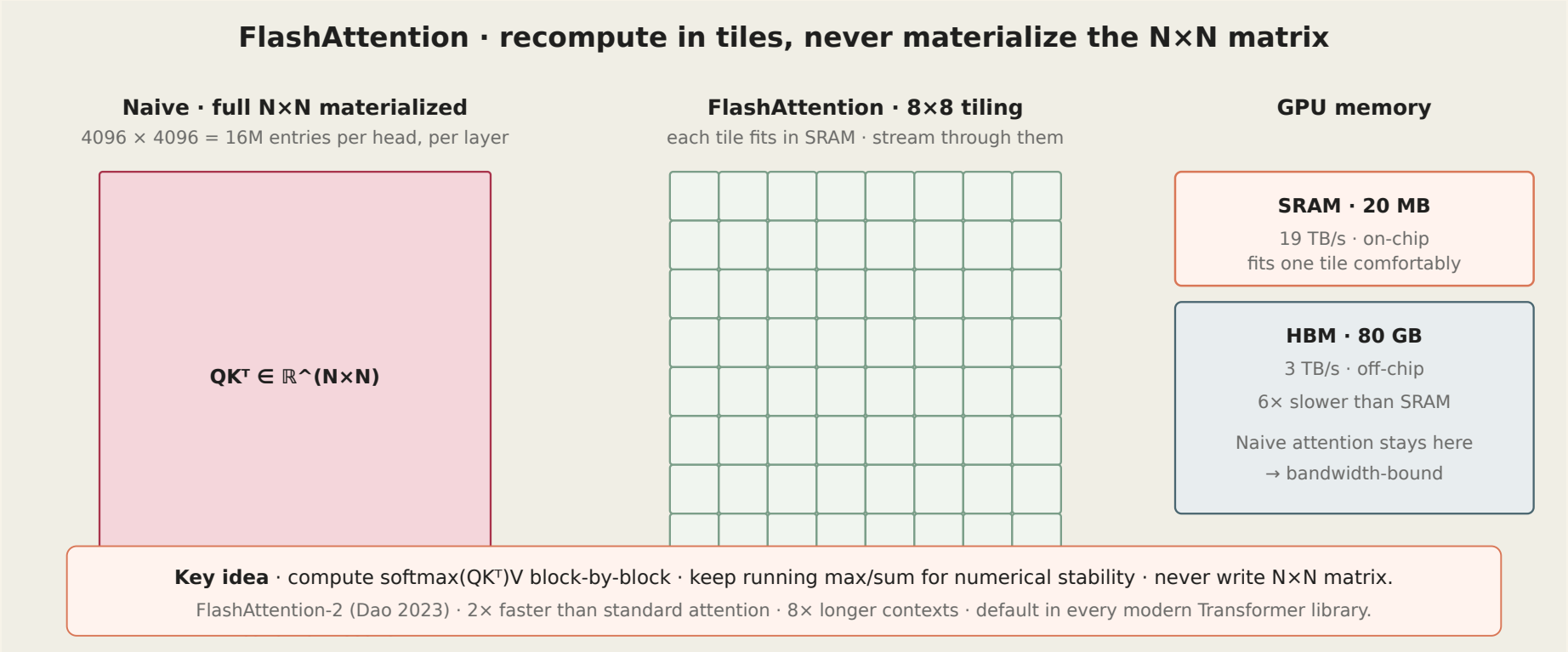
The attention memory problem

Naive attention materializes the $N \times N$ attention matrix:

```
scores = Q @ K.T                 # [B, H, N, N]   ← huge for long context
weights = scores.softmax(dim=-1)
out = weights @ V                 # [B, H, N, d_h]
```

For $N = 8192$, a single layer needs ~8 GB just for the softmax matrix. GPU HBM bandwidth becomes the bottleneck, not FLOPs.

FlashAttention · tiles in SRAM



FlashAttention · the giant-mural analogy

INTUITION

You need to paint a football-field-sized mural (the $N \times N$ attention matrix).

- **Naive** · rent a warehouse, lay out the canvas, paint everything. Huge temporary space (HBM).
- **FlashAttention** · paint one small square at a time, on a portable easel (SRAM). Special technique to keep edges matching. **Never see the whole mural at once.**

For $N = 32{,}768$, the attention matrix has ~1B elements → **4.3 GB per head per layer in FP32**. Too big to read/write to main memory efficiently.

FlashAttention · how it works

Three ideas (Dao et al. 2022):

1. **Tile** · break Q, K, V into blocks (e.g. 256×256). Each block fits in **SRAM** (~20× faster than HBM).
2. **Fuse** · do the entire (matmul + softmax + matmul) for a block in one on-chip kernel — never write the intermediate to HBM.
3. **Online softmax** · math trick to update softmax incrementally as new blocks come in. Result is **mathematically identical** to naive — not an approximation.

Memory savings · 8192-context, FP16.

- Naive · $8192^2 \cdot 2$ bytes ≈ **134 MB** per head/layer (must hit HBM).
- Flash · 256×256 tile · $256^2 \cdot 2 \approx \mathbf{131}\ \mathrm{KB}$ in SRAM. The 134 MB is **never created**.

PyTorch 2.0+ ships `F.scaled_dot_product_attention` — just use it.

FlashAttention · adoption

- PyTorch 2.0+ · built-in as `F.scaled_dot_product_attention`.
- All major inference servers use it.
- **FlashAttention 3** (2024) · further optimizations for H100 Hopper.

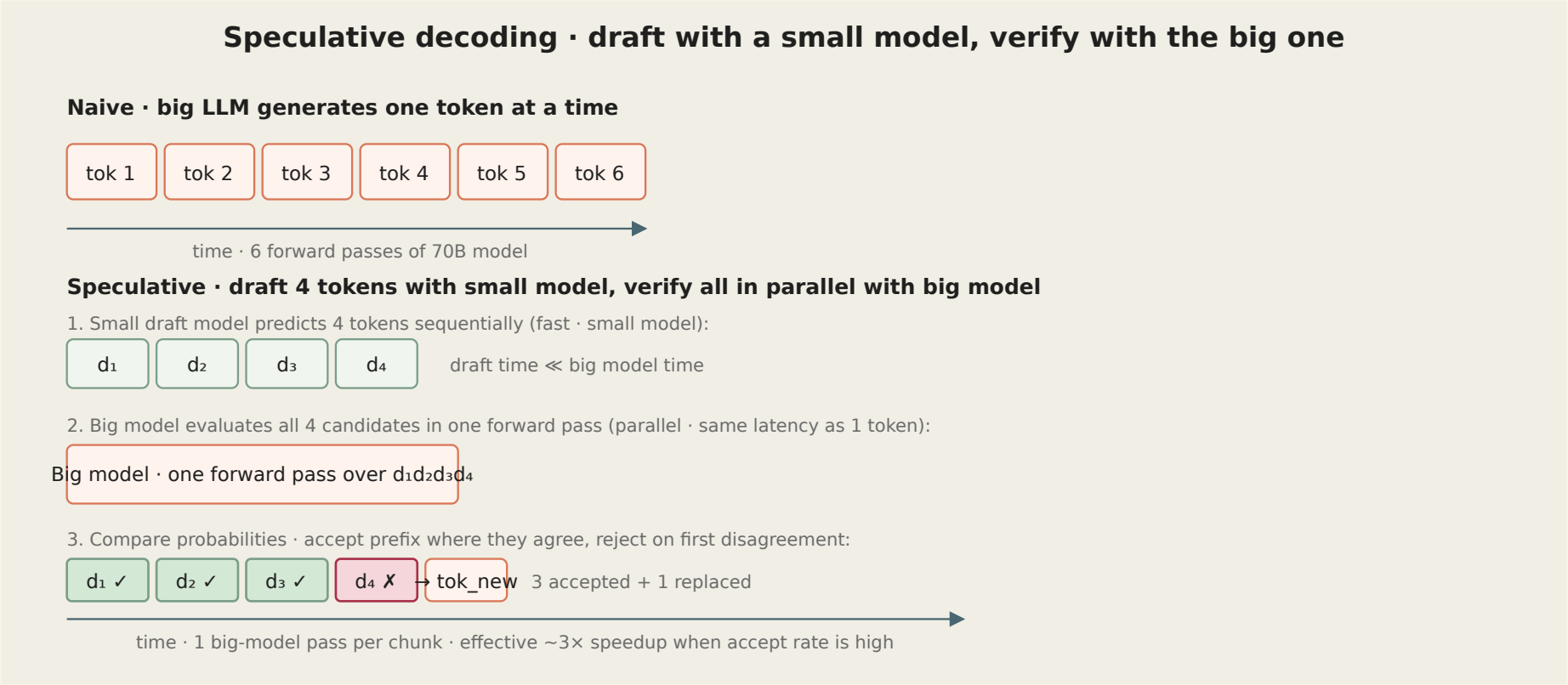
> IN PRACTICE
>
> Just call `F.scaled_dot_product_attention` — don't roll your own attention in 2026.

PART 5

Speculative decoding

Generate multiple tokens per forward pass

Speculative decoding · picture



The decode speedup trick

Autoregressive generation is **one token per forward pass**. For a 70B model this caps your throughput.

Speculative decoding (Leviathan et al. 2022):

1. A small **draft model** (~1B params) quickly guesses the next k tokens.
2. The big **verifier model** (70B) does ONE forward pass on all k in parallel.
3. Accept the longest prefix where draft and verifier agree; rewind if they diverge.

Why speculative works

When the draft is right · k **tokens per forward pass** instead of 1. 2–4× speedup.

When the draft is wrong · same cost as normal decoding (pay for one extra forward on the draft).

Net effect · draft is right ~70% of the time on typical text → ~2× speedup for free.

IN PRACTICE

Used in GPT-4, Claude 3/4, and most hosted LLMs. The "draft" is often a specially-trained small version of the verifier or a lightweight heads-only model.

PART 6

Knowledge distillation

Train a student to mimic a teacher

Distillation · the master-chef analogy

INTUITION

How does an apprentice learn from a master?

- **Hard labels** · master shows finished dish, says "make this." Apprentice knows only the goal.
- **Soft labels** · master says *"lots of tomato, a little basil, a hint of oregano — and crucially, more basil than oregano."* Apprentice learns relative proportions and "dark knowledge" of *what not to do*.

Distillation = method 2.

Distillation · the loss, term by term

Two losses:

Part 1 · standard CE against true labels:
$L_{\text{hard}} = \text{CE}(\text{labels}, \text{student})$

Part 2 · imitate the teacher's *distribution*. Use KL divergence between softened distributions (temperature $T > 1$):
$L_{\text{soft}} = \text{KL}\big(\text{softmax}(z_t/T) \,\|\, \text{softmax}(z_s/T)\big)$

Combine with weight α:

$$\mathcal{L} = \alpha\, L_{\text{hard}} + (1-\alpha)\, T^2\, L_{\text{soft}}$$

The T^2 corrects for the gradient shrinkage that softening causes (so the two losses are comparable in scale).

Distillation · worked numeric

Image of a cat. Classes [Cat, Dog, Car].

- Teacher logits $z_t = [10, 2, 1]$.
- Student logits $z_s = [5, 1.5, 1]$.
  $T = 1$. softmax(z_t) = $[0.999, 0.0003, 0.0001]$. **Almost no dark knowledge** — student barely learns relative class structure.

  $T = 4$. Soften logits: $z_t/4 = [2.5, 0.5, 0.25]$, $z_s/4 = [1.25, 0.375, 0.25]$.
- Soft teacher · softmax($z_t/4$) ≈ $[0.88, 0.08, 0.04]$
- Soft student · softmax($z_s/4$) ≈ $[0.68, 0.17, 0.15]$
  Now the student learns: increase Cat, decrease Dog and Car, **but keep Dog about 2× Car**. Rich nuanced signal that pure hard-labels would miss.

Distillation in 2026

- **DistilBERT** (2019) · 40% smaller, 60% faster, 97% of BERT's quality. Still used.
- **DistilLLaMA, DistilGPT-2** · similar story for generative.
- **Modern practice** · distill a 70B teacher into a 7B student with task-specific data · near-teacher quality at 10× cheaper inference.

> INTUITION
>
> Many "small-but-good" 2026 models (Phi, Gemma, DistilRoBERTa) are distilled from bigger siblings. The frontier labs train big, then distill to ship.

PART 7

Full inference stack · 2026

What a production LLM server does

1. **Batched inference** · pack many user requests into each forward pass.
2. **Paged attention** for KV-cache memory efficiency.
3. **INT8/INT4 quantization** for the weights.
4. **FlashAttention** for attention compute.
5. **Speculative decoding** for throughput.
6. **Continuous batching** · new requests can join mid-batch without restart.
7. **Streaming output** · send tokens as they are generated.

Put together · ~10–50× faster and cheaper than naive implementations.

The inference frameworks

FRAMEWORK	WHO	GOOD AT
vLLM	Berkeley / community	paged attention, continuous batching
TGI (Text Generation Inference)	Hugging Face	production serving
llama.cpp	community	CPU + laptop inference
TensorRT-LLM	NVIDIA	peak H100 performance
MLX	Apple	M-series Macs
ExLlamaV2	community	consumer GPU (RTX 4090)

Choose by hardware + quality needs. For teaching, **vLLM** or **llama.cpp** are easiest to install.

End-to-end · optimization stack compounded

DERIVATION

STAGE	SPEEDUP VS NAIVE
Baseline (naive fp32)	1×
+ bf16	2×
+ KV-cache	10×
+ FlashAttention-2	3×
+ INT8 quantization	1.5×
+ speculative decoding	2.5×
+ batching + paged attention	4×
Total (compounded)	**~900×**

KEY IDEA

Real production serving · ~30-100× over naive PyTorch. The rest is latency engineering (batching, KV-cache reuse across requests, model sharding).

Cost economics · 2026

DERIVATION

MODEL	COST PER 1M INPUT TOKENS	COST PER 1M OUTPUT TOKENS
Claude 3.5 Haiku	$1	$5
GPT-4o-mini	$0.15	$0.60
Llama-3 70B (self-hosted)	~$0.30	~$0.60
Claude 3.5 Sonnet	$3	$15
o1	$15	$60

IN PRACTICE

Reasoning models cost 5-10× more (inference-time compute). Same token count, much more compute per token. "Think longer" is pay-per-second.

Summary · Lecture 23 — summary

- **Prefill vs decode** · compute-bound vs memory-bound. Different optimizations.
- **KV-cache** · dominates memory for long contexts. GQA (L15) + paged attention shrink it.
- **Quantization** · BF16 → INT8 → NF4. AWQ handles 4-bit well; <1% quality loss.
- **FlashAttention** · exact attention with O(N) memory · 2–4× faster.
- **Speculative decoding** · draft model proposes, big model verifies in parallel · 2–4× throughput.
- **Distillation** · big teacher trains small student on soft targets.
- **Production stack** · all of these combined → ~10–50× over naive.

Read before Lecture 24

Anthropic interp blog; Chi et al. 2023 (Diffusion Policy); blog posts on Claude Code / computer use.

Next lecture · last one!

Frontier · Agents, Reasoning, Interpretability + course wrap-up.

NOTEBOOK

Notebook 23 · `23-kv-cache.ipynb` — take a small GPT; add KV-cache to generation loop; measure tokens/second speedup.