CS388: Natural Language Processing

Lecture 21: Efficiency and LLMs



TEXAS
The University of Texas at Austin

Slides credit: Greg Durrett

Announcements

- Guest lecture on the 12th
- Final projects
- HW 4 due on the 12th
- Quiz 4 on the 17th



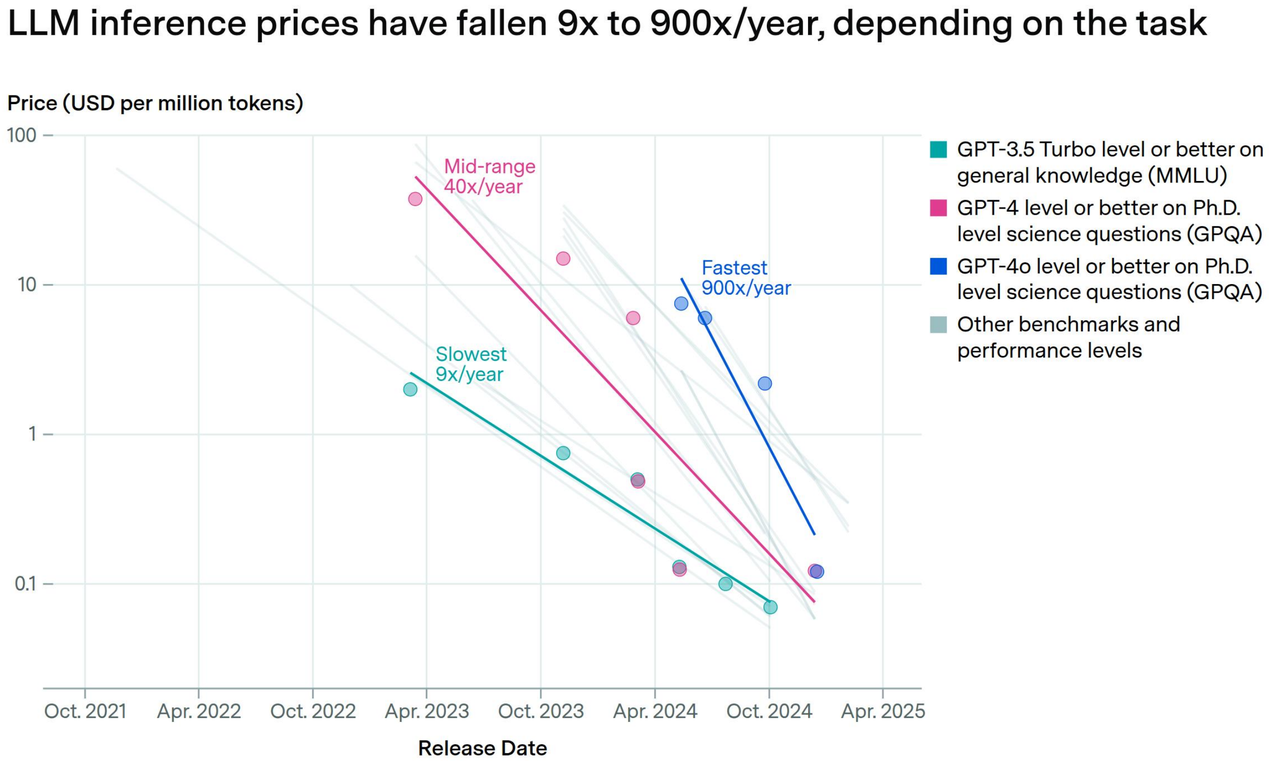
This Lecture

- Decoding optimizations: exact decoding, but faster
  - KV caching, paged attention
  - Speculative decoding, Flash attention

- Distillation
  - Soft vs. hard targets, pipelines
  - When to distill and why it works

- Parameter-efficient tuning: BitFit, LoRA, quantization

Model Cost

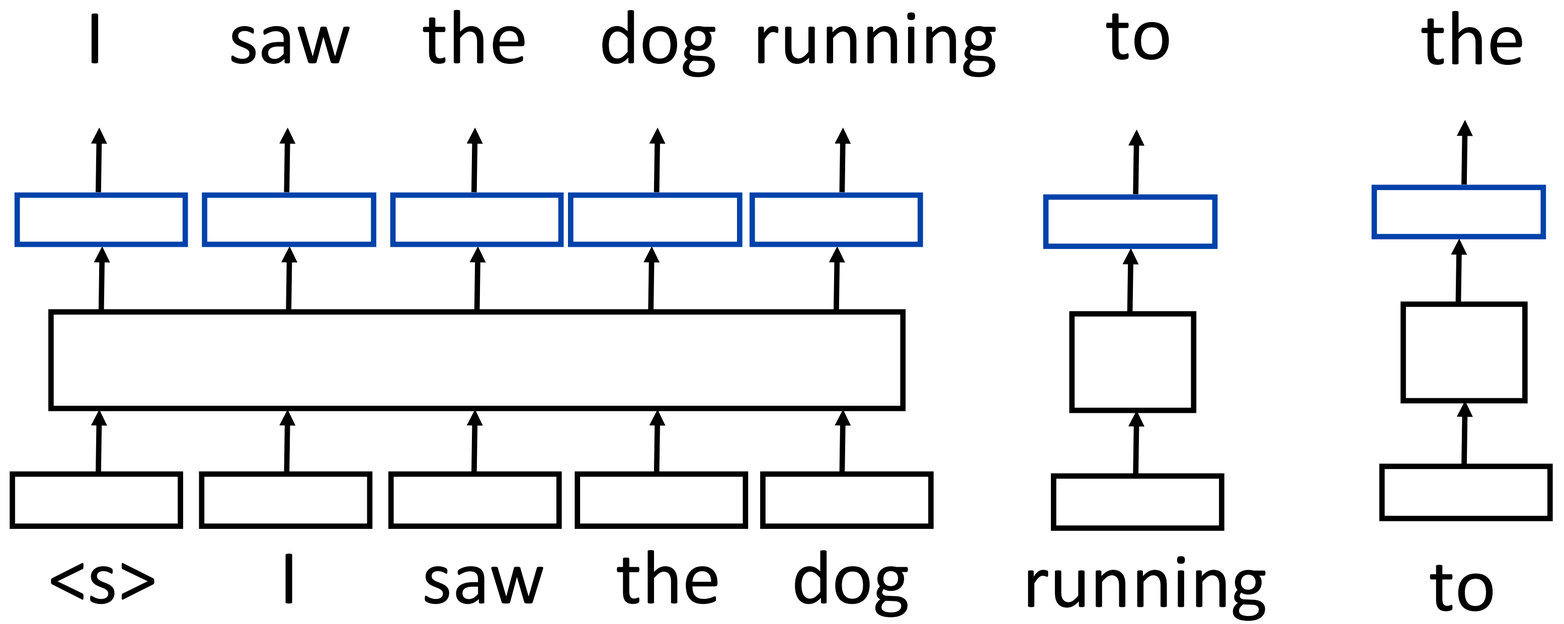


- Bigger models, yet a fixed capability gets cheaper fast
  - **9-900× cheaper per year**
- What drives this?
  - Today's lecture

Cottier et al. (2025), LLM inference prices have fallen rapidly but unequally across tasks
Gundlach et al. (2025), The Price of Progress: Algorithmic Efficiency and the Falling Cost of AI Inference

Decoding Optimizations

Decoding Basics



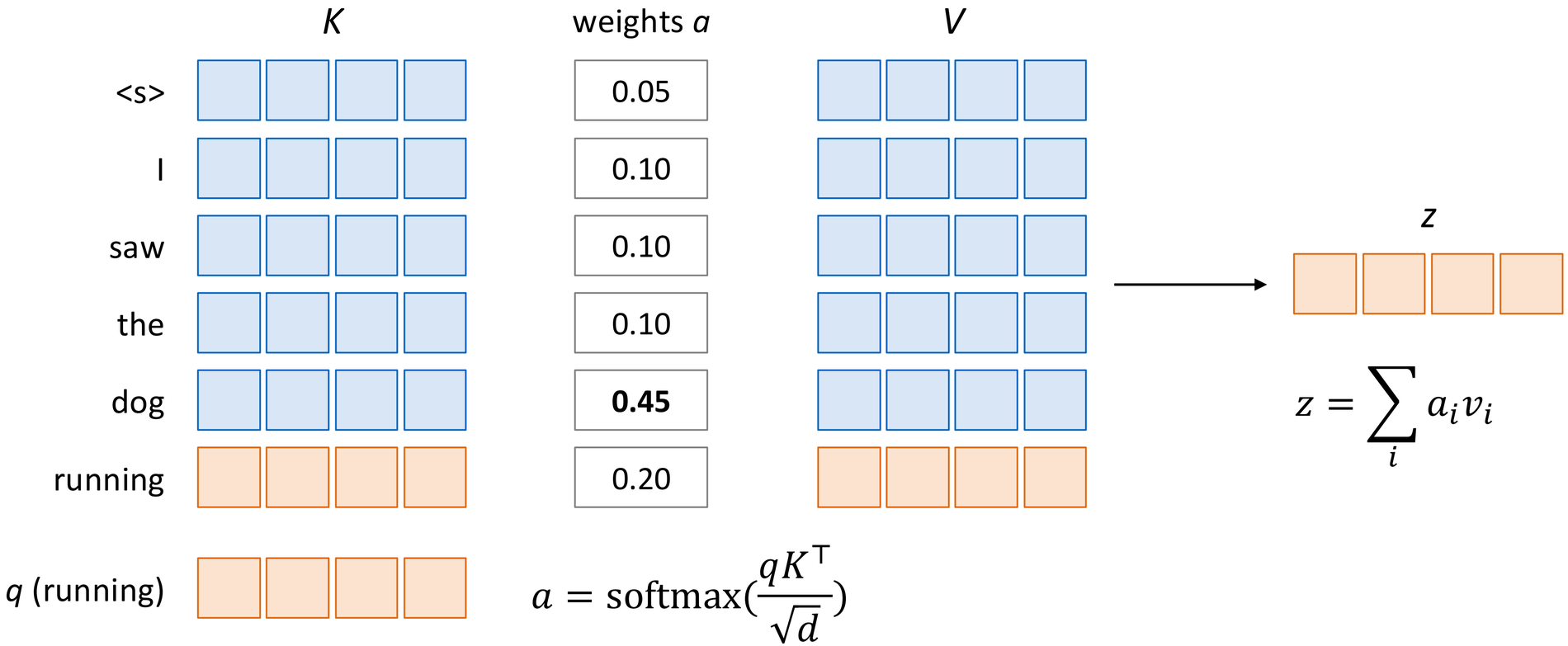
Operations for one decoder pass: $O(pL)$

Operations for k decoder passes: $O((pk + k^2)L)$

Transformer layers (non-parallelizable ops): $O(kL)$

Recall Attention

- Self-attention: K, Q, V
- K, V are the previous tokens



Vaswani et al. (2017), Attention Is All You Need

KV Caching

- Causal attention: past keys and values never change
- Naive: re-run the model on the whole prefix at every step

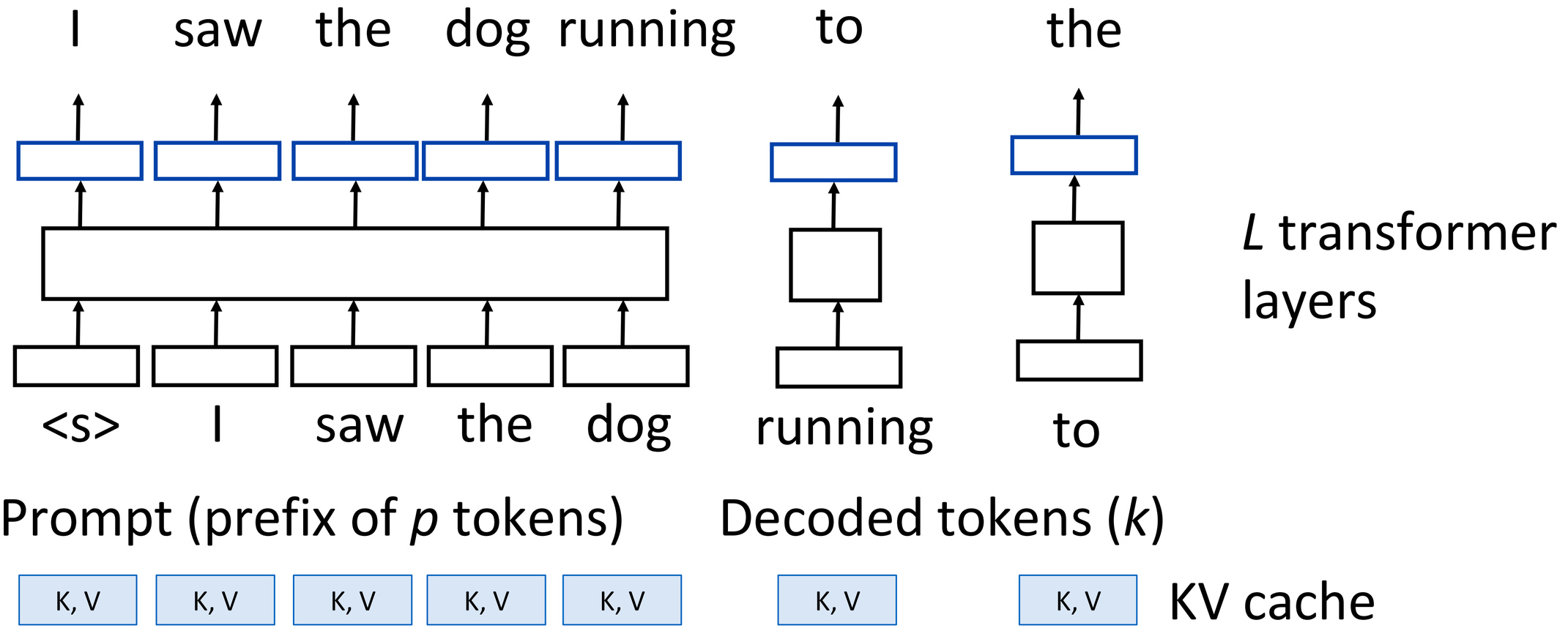
$$\sum_{i=1}^{k}(p+i-1) \approx pk + \frac{k^2}{2} \text{ token passes}$$

- **KV cache:** store each token's K, V; run only the new token

$$p + k \text{ token passes}$$

- Example: $p = 1000, k = 500 \rightarrow$ ~625K vs. 1.5K token passes

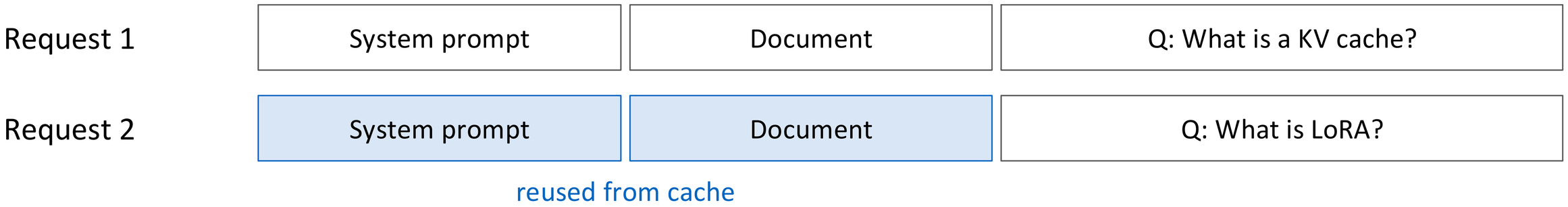
KV Cache in Action



- **Prefill:** run the prompt once, cache K, V in every layer
- **Decode:** compute K, V for the new token only, append

Prefix Caching

- Requests that share a prefix can share its KV cache
  - System prompts, documents, chat history, agent loops
- Hit only on an **exact** token match from the first token
  - One changed token → recompute everything after it
- APIs pass on the savings, e.g., cached input at 0.1× price



Anthropic, Prompt caching (Claude API docs); OpenAI (2024), Prompt Caching in the API

Cache Hit or Miss?

In cache

You are a TA for CS388.	Q: What is a KV cache?

Request 1

You are a TA for CS388.	Q: What is LoRA?

Hit: the system prompt is reused

Request 2

Today is Nov 8.	You are a TA for CS388.	Q: What is a KV cache?

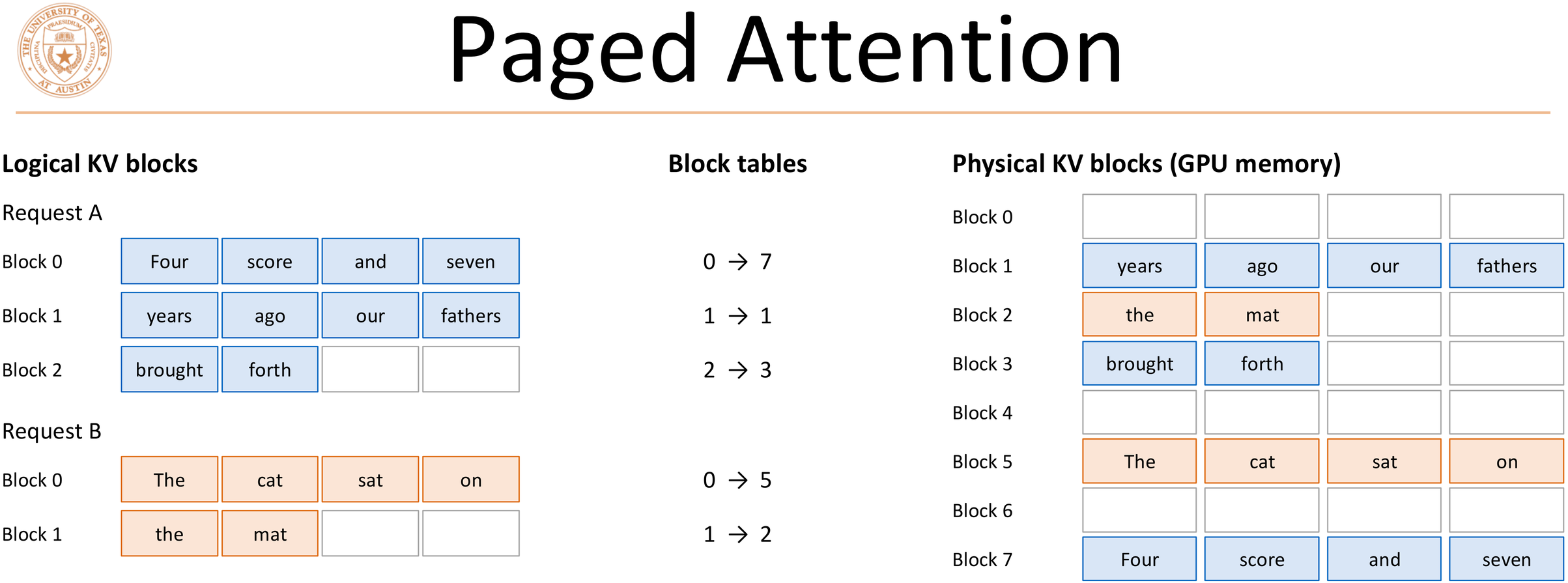
Miss: the first tokens differ, so nothing is reused

KV Cache Memory

- Per token: $2 \cdot L \cdot d$ numbers (K, V per layer)
  - OPT-13B: ≈ 800 KB per token
  - 2,048 tokens → 1.6 GB per request (on top of weights)
- Older systems reserve max-length memory up front (we don't know how long the response will be up-front)
- Only **20–38%** of KV memory held actual tokens → most reserved memory is wasted

Kwon et al. (2023), Efficient Memory Management for Large Language Model Serving with PagedAttention

Paged Attention



- KV cache stored in fixed-size blocks, like OS memory pages
- Blocks allocated on demand, anywhere in memory
- Waste < 4% → bigger batches → **2–4× throughput**

Kwon et al. (2023), Efficient Memory Management for Large Language Model Serving with PagedAttention

KV Cache

- Reduces number of operations for one decoder pass

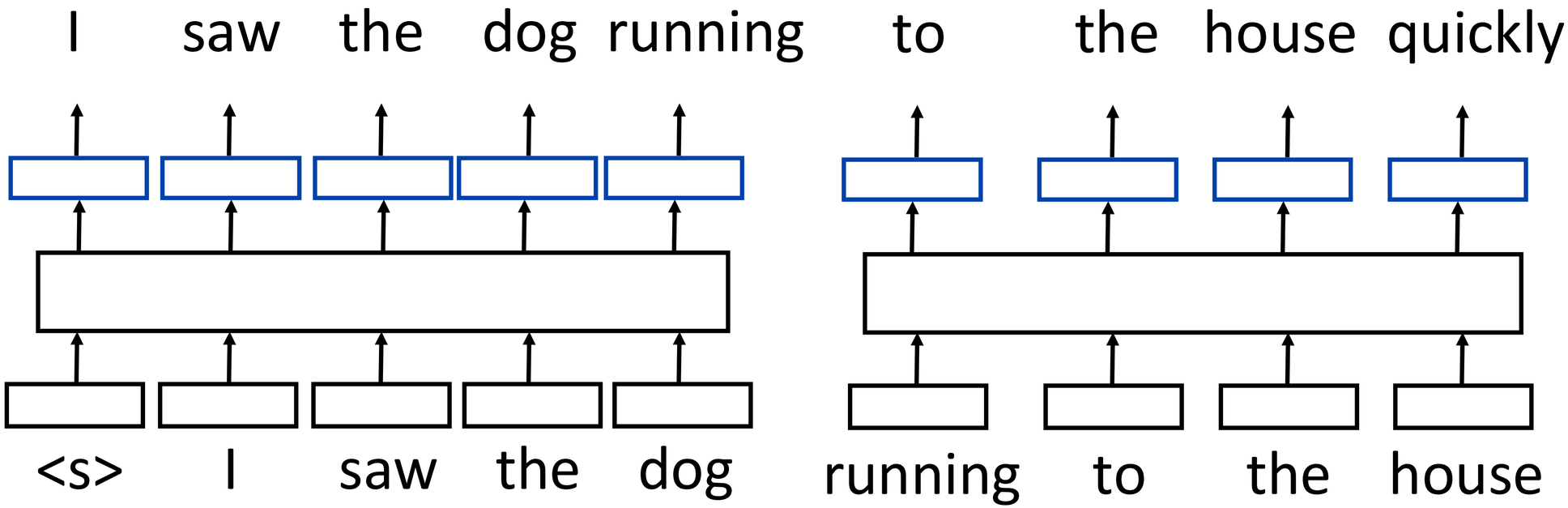
  Before: operations for k decoder passes: $O((pk + k^2)L)$

  With KV cache: operations for k decoder passes: $O((p + k)L)$

- Trades operations for memory cost: $2 \cdot L \cdot d$ numbers per token
- But memory is general cheap (and keeps getting cheaper)

- Does not address: Transformer layers (non-parallelizable ops): $O(kL)$
- Next: Speculative Decoding

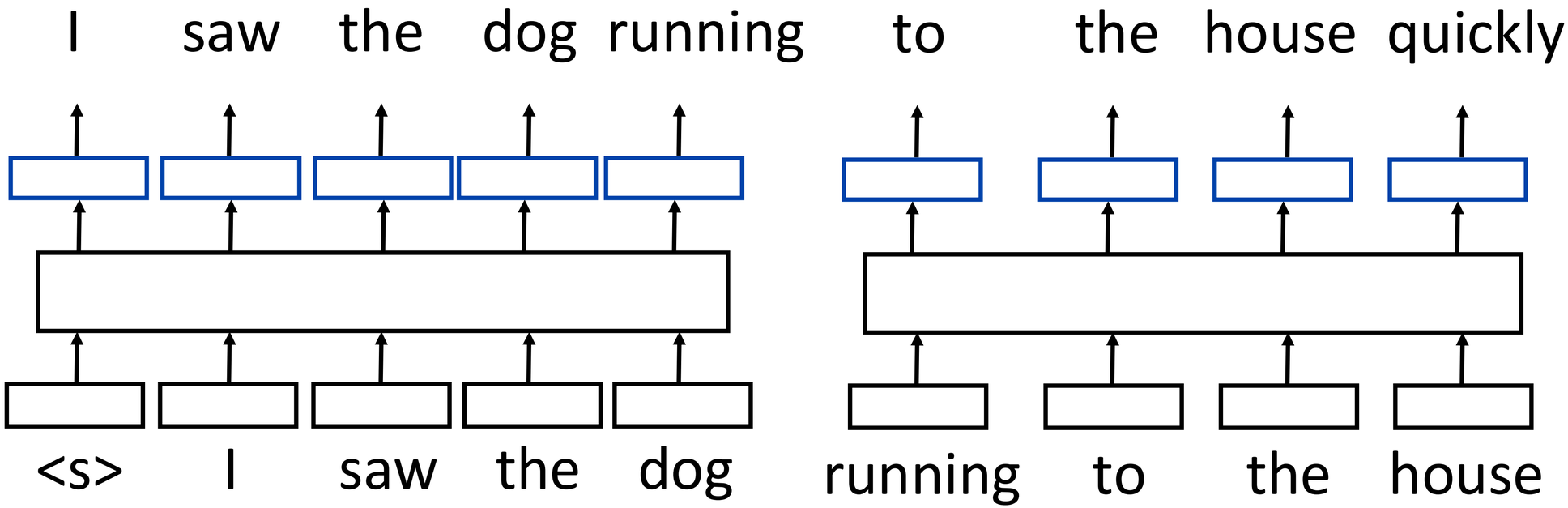
Speculative Decoding



Prompt (prefix of *p* tokens) Decoded tokens (*k*)

- Key idea a forward pass for several tokens at a time is O(*L*) serial steps, since the tokens can be computed in parallel
- Can we predict many tokens with a weak model and then "check" them with a single forward pass?

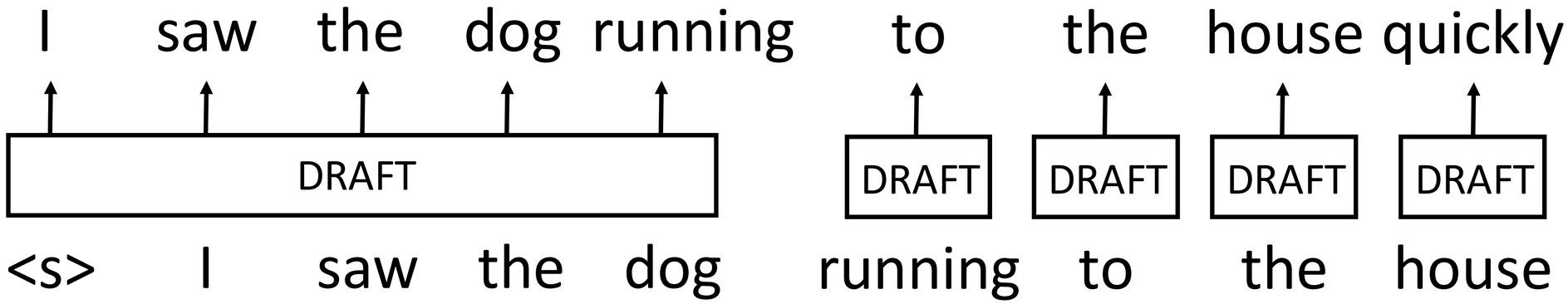
Speculative Decoding



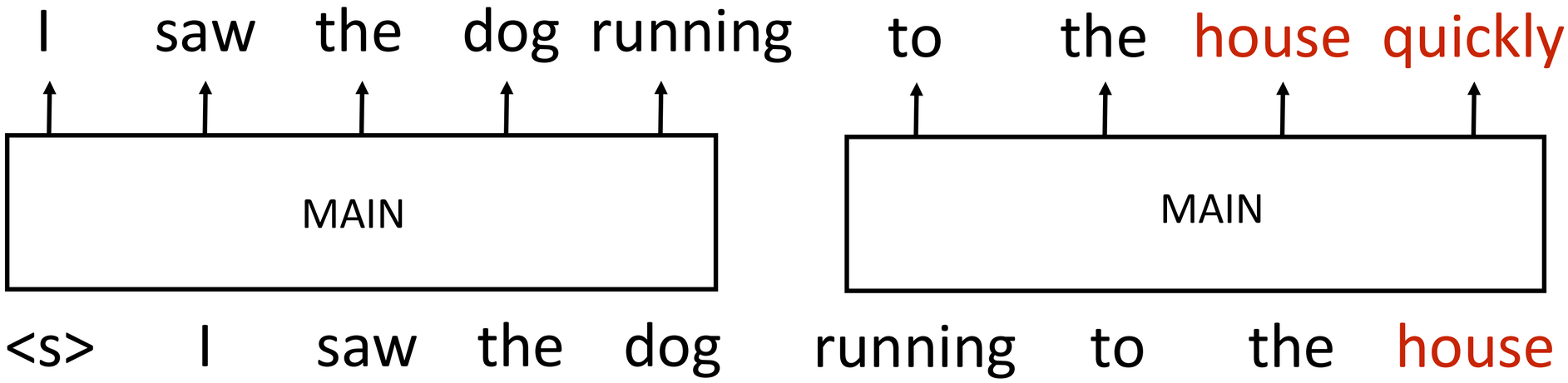
- **We can use a small, cheap model to do inference, then check that "to", "the", "house", "quickly" are really the best tokens from a bigger model**

Leviathan et al. (2023)

Speculative Decoding: Flow



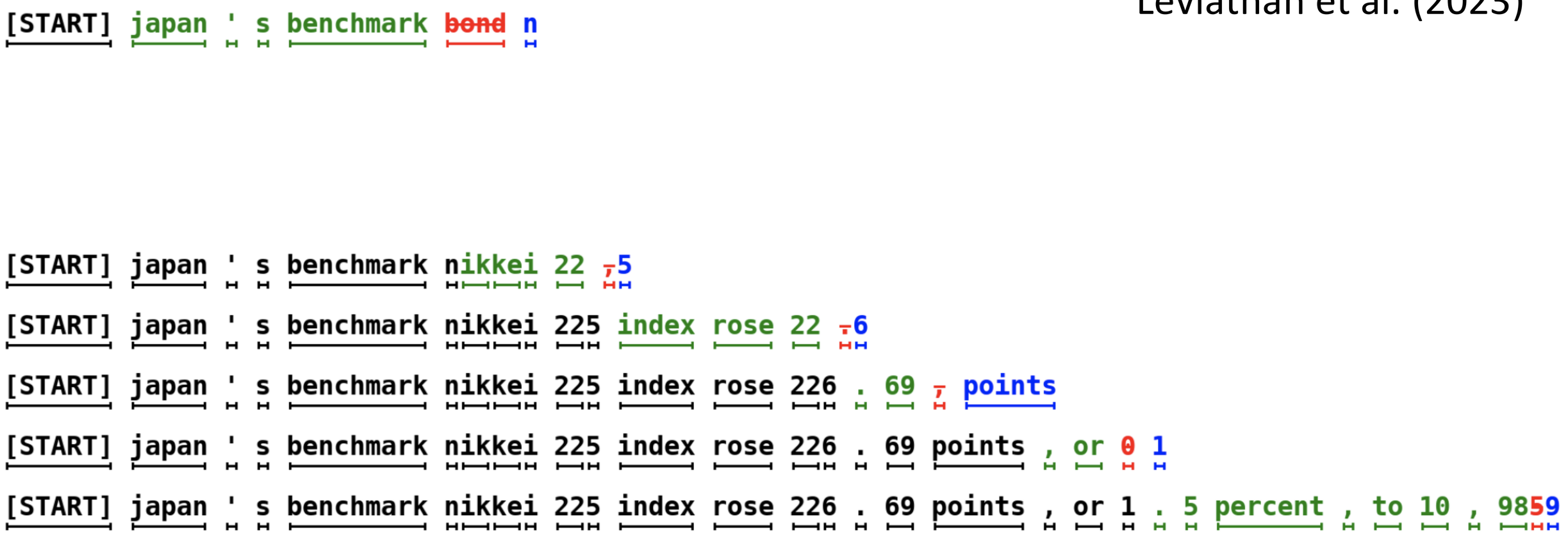
- Produce decoded tokens one at a time from a fast draft model…



- Confirm that the tokens are the max tokens from the slower main model. Any “wrong” token invalidates the rest of the sequence

Speculative Decoding

Leviathan et al. (2023)



- Can also adjust this to use sampling. Treat this as a proposal distribution $q(x)$ and may need to reject + resample (rejection sampling)

Speculative Decoding

- Find the first index that was rejected by the sampling procedure, then resample from there
- Sweet spot of chunk size
- Big chunk size: you can get lots of tokens at once but will have many more misses
- Small chunk size: have to do more checks with big model
- In practice: 4-5 tokens

Leviathan et al. (2023)

Inputs: $M_p, M_q, prefix$.
▷ Sample γ guesses $x_{1,\ldots,\gamma}$ from M_q autoregressively.
for $i = 1$ **to** γ **do**
  $q_i(x) \leftarrow M_q(prefix + [x_1, \ldots, x_{i-1}])$
  $x_i \sim q_i(x)$
end for
▷ Run M_p in parallel.
$p_1(x), \ldots, p_{\gamma+1}(x) \leftarrow$
  $M_p(prefix), \ldots, M_p(prefix + [x_1, \ldots, x_\gamma])$
▷ Determine the number of accepted guesses n.
$r_1 \sim U(0,1), \ldots, r_\gamma \sim U(0,1)$
$n \leftarrow \min(\{i - 1 \mid 1 \leq i \leq \gamma, r_i > \frac{p_i(x)}{q_i(x)}\} \cup \{\gamma\})$
▷ Adjust the distribution from M_p if needed.
$p'(x) \leftarrow p_{n+1}(x)$
if $n < \gamma$ **then**
  $p'(x) \leftarrow norm(max(0, p_{n+1}(x) - q_{n+1}(x)))$
end if
▷ Return one token from M_p, and n tokens from M_q.
$t \sim p'(x)$
return $prefix + [x_1, \ldots, x_n, t]$

Why Inference?

- Before: improve training speed
- Now: improving inference speed is a priority. Why?
- **Reason 1: Users and model type**
  - Every query needs inference; cost grows with users/usage
  - Reasoning models write more: output length ~5× per year
- **Reason 2: RL**
  - Training is mostly generation: rollouts take majority of each step
  - Faster inference = faster RL training

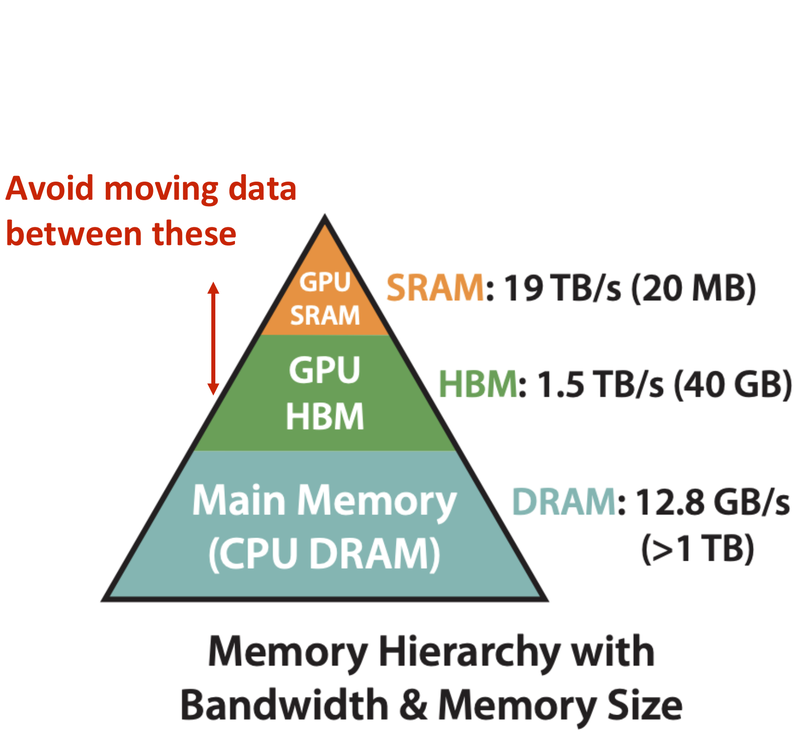
Epoch AI (2025), LLM responses to benchmark questions are getting longer over time
Qin et al. (2025), Seer: Online Context Learning for Fast Synchronous LLM Reinforcement Learning



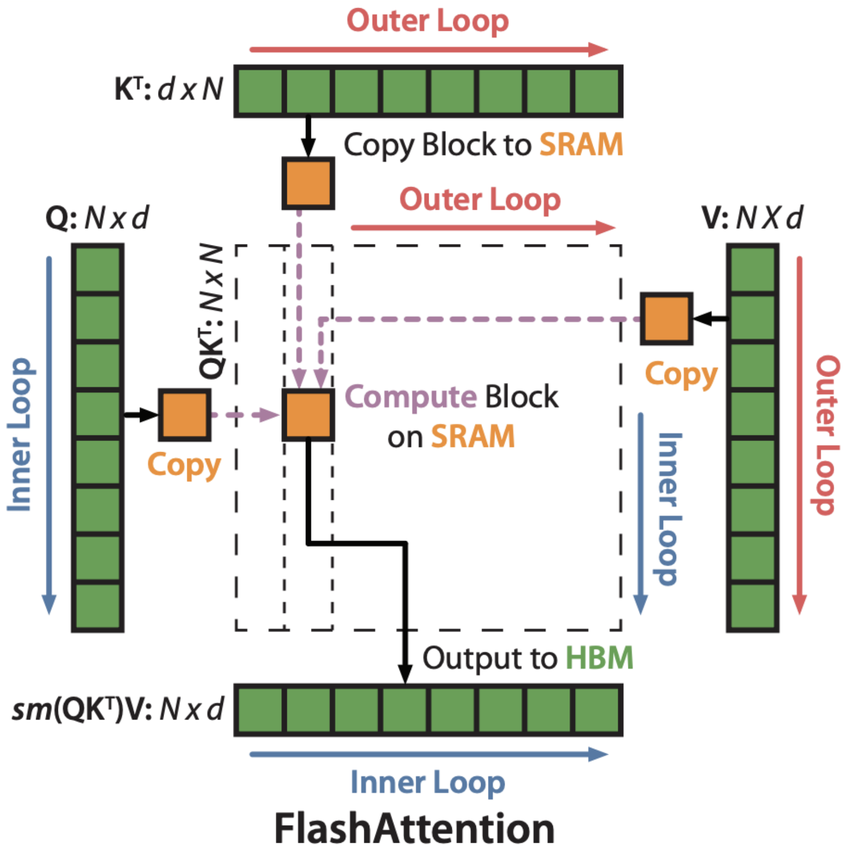
Other Decoding Improvements

- Many other approaches to speeding up require changing the model (making a faster Transformer) or making it smaller (distillation, pruning; discussed next)
- Batching parallelism: improve throughput by decoding many examples in parallel. (Does not help with latency, and it's a little bit harder to do in production if requests are coming in asynchronously)
- Low-level hardware optimizations
  - Custom kernels

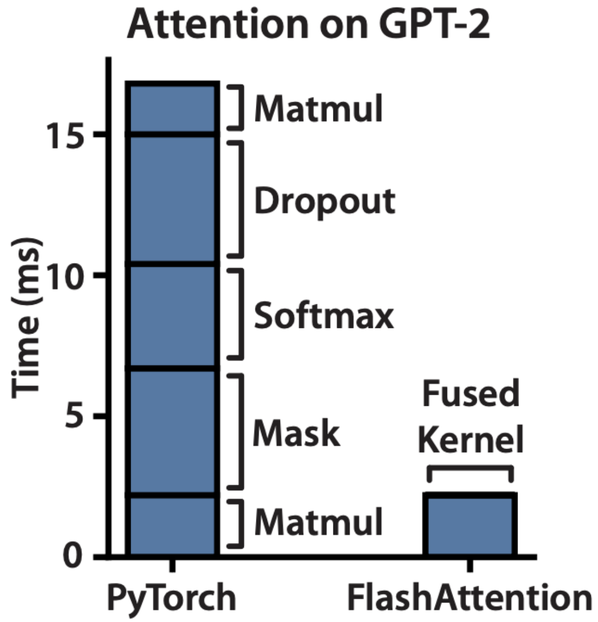
Flash Attention



Memory Hierarchy with Bandwidth & Memory Size



FlashAttention



- Does extra computation during attention, but avoids expensive reads/writes to GBU “high-bandwidth memory.” Recomputation is all in SRAM and is very fast
- Essentially: store a running sum for the softmax, compute values as needed



Flash Attention

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Algorithm 1 FlashAttention

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

[dividing stuff into blocks]

5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

[more computation, writes to HBM]



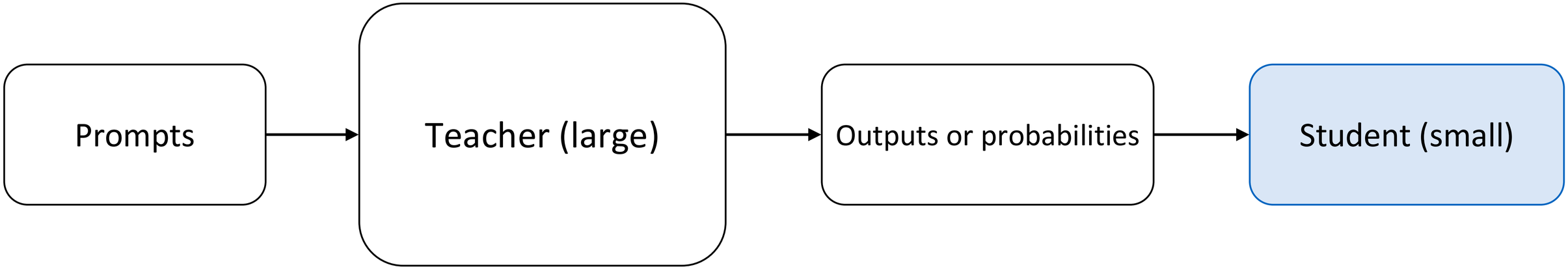
Flash Attention

Models	ListOps	Text	Retrieval	Image	Pathfinder	Avg	Speedup
Transformer	36.0	63.6	81.6	42.3	72.7	59.3	-
FlashAttention	37.6	63.9	81.4	43.5	72.7	59.8	2.4×
Block-sparse FlashAttention	37.0	63.0	81.3	43.6	73.3	59.6	**2.8**×
Linformer [84]	35.6	55.9	77.7	37.8	67.6	54.9	2.5×
Linear Attention [50]	38.8	63.2	80.7	42.6	72.5	59.6	2.3×
Performer [12]	36.8	63.6	82.2	42.1	69.9	58.9	1.8×
Local Attention [80]	36.1	60.2	76.7	40.6	66.6	56.0	1.7×
Reformer [51]	36.5	63.8	78.5	39.6	69.4	57.6	1.3×
Smyrf [19]	36.1	64.1	79.0	39.6	70.5	57.9	1.7×

- Gives a speedup for free — with no cost in accuracy (modulo numeric instability)
- Outperforms the speedup from many other approximate Transformer methods, which perform substantially worse

Distillation

Model Distillation

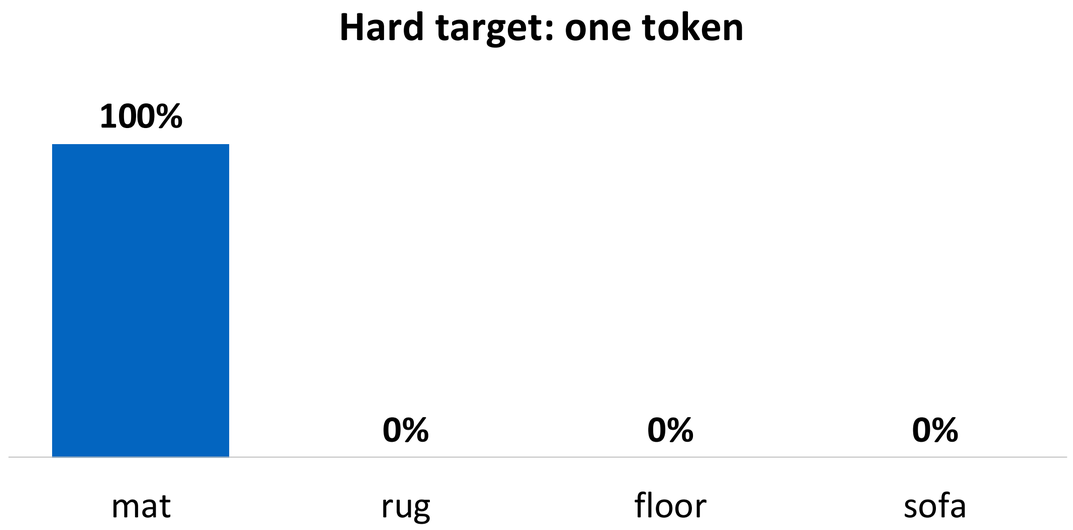


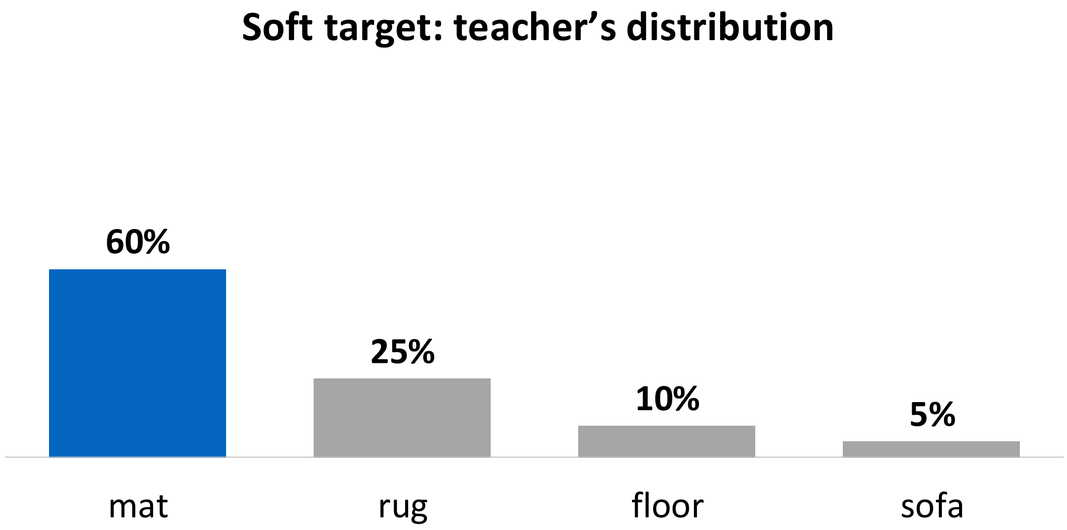
- Train a small student to imitate a large teacher
- Deploy the student: cheaper and faster
- Used everywhere: reasoning models, agents, small open models, large proprietary model competitors

Hinton et al. (2015), Distilling the Knowledge in a Neural Network

Soft vs. Hard Targets

Next token after “The cat sat on the ___”





- **Hard:** fine-tune on teacher-generated text (API access is enough)
- **Soft:** minimize $\mathrm{KL}(p_T \parallel p_S)$ at every token (needs logits)
- Soft targets carry more signal: 3% of data → 57.0% vs. 44.5% accuracy
- One reason why frontier models don’t give logits anymore

Hinton et al. (2015), Distilling the Knowledge in a Neural Network

Distilling LLMs

Method	Student trains on	Teacher access	Example
Sequence-level	Teacher's generated outputs (plain SFT)	API is enough	DeepSeek-R1-Distill
Token-level	Teacher's next-token distribution	Logits, shared tokenizer	Gemma 2 (2B, 9B)
On-policy	Student's own samples, scored by the teacher	Logits + student sampling	OPD, OPSD

- On-policy: the student learns from its own mistakes

Kim & Rush (2016), Sequence-Level Knowledge Distillation
Gemma Team (2024), Gemma 2: Improving Open Language Models at a Practical Size
Agarwal et al. (2024), On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes

Distillation Pipeline



- **Reasoning:** DeepSeek-R1 → 800K samples → Qwen, Llama (1.5B–70B)
- **Agents:** GPT-4 → 500 trajectories → Llama-2-7B, +77% on HotpotQA
- **Instruction following:** text-davinci-003 → 52K demos → Alpaca

DeepSeek-AI (2025), DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning
Chen et al. (2023), FireAct: Toward Language Agent Fine-tuning
Taori et al. (2023), Stanford Alpaca: An Instruction-following LLaMA Model

When to Distill?

- A stronger teacher exists for the task
- Serving cost or latency matters
- RL on a small model is too costly
  - R1 32B student: 72.6% on AIME distilled vs. 47.0% with RL

Table 16 | Comparison of distilled and RL Models on Reasoning-Related Benchmarks.

Model	AIME 2024		MATH	GPQA Diamond	LiveCode Bench
	pass@1	cons@64	pass@1	pass@1	pass@1
QwQ-32B-Preview	50.0	60.0	90.6	54.5	41.9
Qwen2.5-32B-Zero	47.0	60.0	91.6	55.0	40.2
DeepSeek-R1-Distill-Qwen-32B	**72.6**	**83.3**	**94.3**	**62.1**	**57.2**

DeepSeek-AI (2025), DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning
Mirzadeh et al. (2020), Improved Knowledge Distillation via Teacher Assistant

When to Distill?

- **Skip it when:**
- Teacher's terms forbid it
- Student is far too small (capacity gap)

Table 16 | Comparison of distilled and RL Models on Reasoning-Related Benchmarks.

Model	AIME 2024		MATH	GPQA Diamond	LiveCode Bench
	pass@1	cons@64	pass@1	pass@1	pass@1
QwQ-32B-Preview	50.0	60.0	90.6	54.5	41.9
Qwen2.5-32B-Zero	47.0	60.0	91.6	55.0	40.2
DeepSeek-R1-Distill-Qwen-32B	**72.6**	**83.3**	**94.3**	**62.1**	**57.2**

DeepSeek-AI (2025), DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning
Mirzadeh et al. (2020), Improved Knowledge Distillation via Teacher Assistant



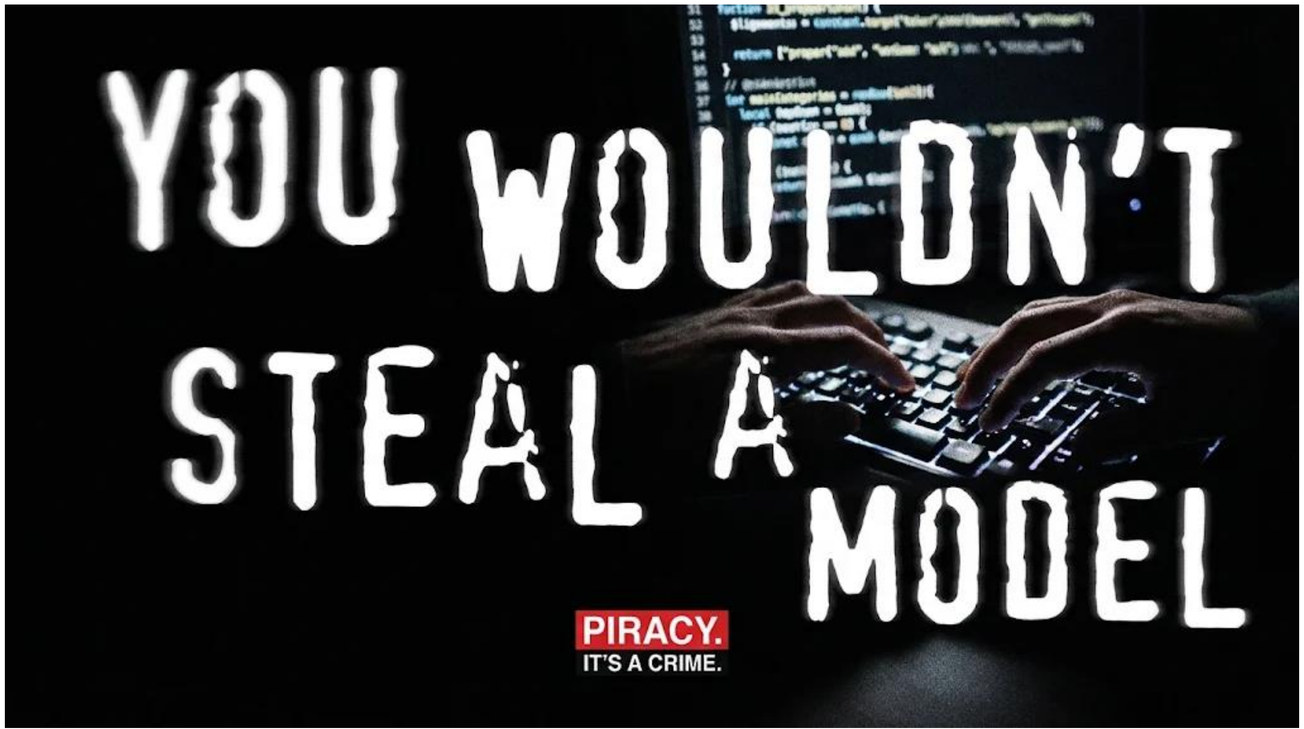
Why Distillation Works

- Big networks are easier to train…
- …but the function they learn often fits in a smaller one
- **Lottery ticket hypothesis:** dense nets contain subnetworks (10–20% of the size) that train to the same accuracy
- Distillation: learn from model that already hit the lottery ticket

Frankle & Carbin (2019), The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks
Hinton et al. (2015), Distilling the Knowledge in a Neural Network

Is Distillation Theft?

- OpenAI claims: DeepSeek distilled OpenAI models (against ToS)
- Not a copyright claim but contract claim (frontier labs have no leg to stand on, copyright-wise)
- What do you think?



Parameter-Efficient Tuning

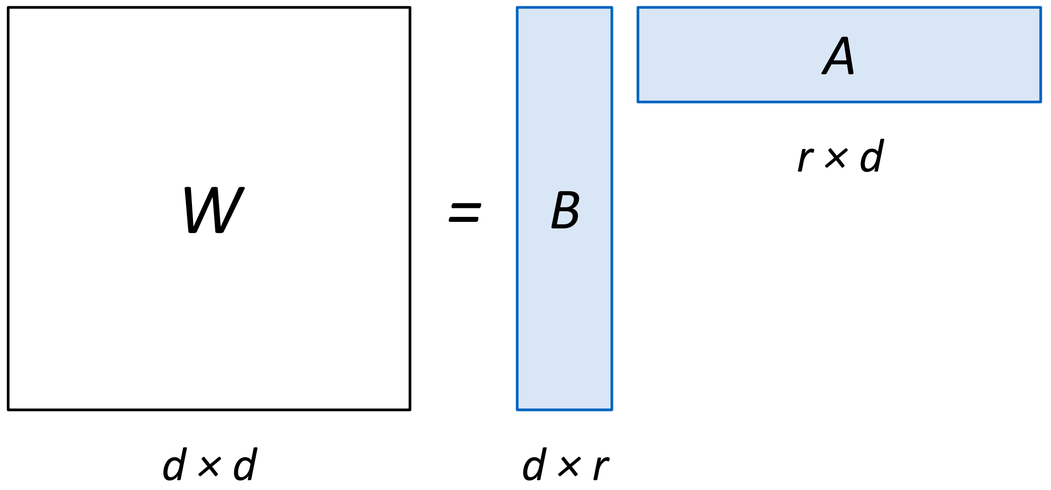


Parameter-Efficient Tuning

- Rather than train all model parameters at once, can we get away with just training a small number of them?
- What are the advantages of this?
- Typical advantages: lower memory, easier to serve many models for use cases like personalization or multitasking
- Not an advantage: faster (it's not)

Low-Rank Factorization

- A rank-r matrix factors into two thin matrices
  $W = BA$
  $B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times d}$
- When $r \ll d$, B and A are much smaller than W
- LoRA hypothesis: the fine-tuning update ΔW has low rank



Rank-1 example: 9 numbers → 6

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

Hu et al. (2021)

LoRA

- Learn weight matrices as ($W + BA$), where BA is a product of two low-rank matrices.
  - If we have a $d \times d$ matrix and we use a rank reduction of size r, what is the parameter reduction from LoRA?
- Allows adding low-rank matrix on top of existing high-rank model
- Unlike some other methods, LoRA can be "compiled down" into the model (just add BA into W)

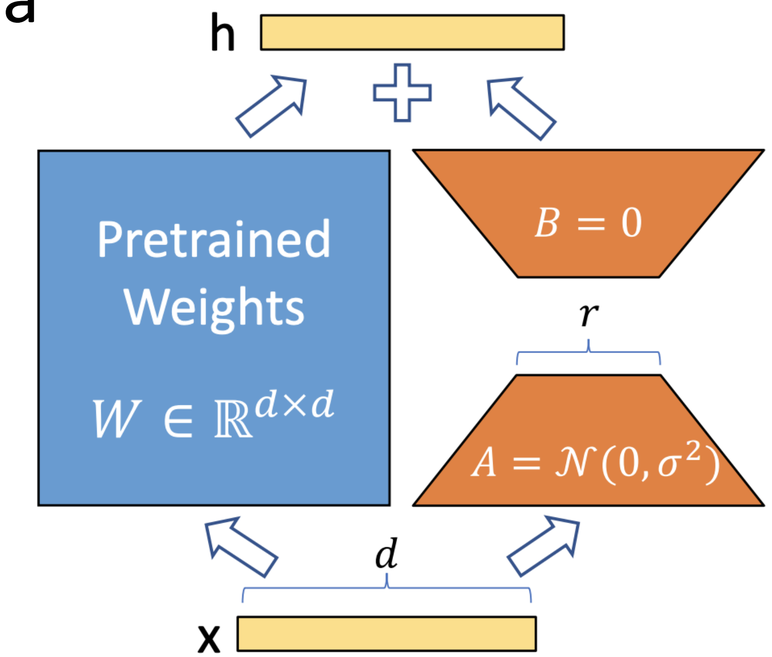


Figure 1: Our reparametrization. We only train A and B.

Hu et al. (2021)

LoRA

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB_{base} (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoB_{base} (BitFit)*	0.1M	84.7	93.7	**92.7**	62.0	91.8	84.0	81.5	90.8	85.2
RoB_{base} (Adpt^{D})*	0.3M	$87.1_{\pm.0}$	$94.2_{\pm.1}$	$88.5_{\pm1.1}$	$60.8_{\pm.4}$	$93.1_{\pm.1}$	$90.2_{\pm.0}$	$71.5_{\pm2.7}$	$89.7_{\pm.3}$	84.4
RoB_{base} (Adpt^{D})*	0.9M	$87.3_{\pm.1}$	$94.7_{\pm.3}$	$88.4_{\pm.1}$	$62.6_{\pm.9}$	$93.0_{\pm.2}$	$90.6_{\pm.0}$	$75.9_{\pm2.2}$	$90.3_{\pm.1}$	85.4
RoB_{base} (LoRA)	0.3M	$87.5_{\pm.3}$	$\mathbf{95.1}_{\pm.2}$	$89.7_{\pm.7}$	$63.4_{\pm1.2}$	$\mathbf{93.3}_{\pm.3}$	$90.8_{\pm.1}$	$\mathbf{86.6}_{\pm.7}$	$\mathbf{91.5}_{\pm.2}$	**87.2**
$\text{RoB}_{\text{large}}$ (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
$\text{RoB}_{\text{large}}$ (LoRA)	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.9}_{\pm1.2}$	$\mathbf{68.2}_{\pm1.9}$	$\mathbf{94.9}_{\pm.3}$	$91.6_{\pm.1}$	$\mathbf{87.4}_{\pm2.5}$	$\mathbf{92.6}_{\pm.2}$	**89.0**

- LoRA is much better, even better than vanilla fine-tuning on GLUE!
- LoRA is the standard way of training models. Most papers that do training do LoRA.

Hu et al. (2021)



LLM Quantization

- A significant fraction of LLM training is just storing the weights
  - Normal floating-point precision: 4 bytes per weight, gets large for 10B+ parameter models!
- How much is needed for fine-tuning?
  - The Adam optimizer has to store at least 2 additional values for each parameter (first- and second-moment estimates)
  - Memory gets very large! Can we reduce this?

LLM Quantization

	Exponent	Fraction
IEEE 754 Single Precision 32-bit Float (FP32) 	8	23
IEEE 754 Half Precision 16-bit Float (FP16) 	5	10
Google Brain Float (BF 16) 	8	7
Nvidia FP8 (E_4M_3) 	4	3

Exponent bits set the range; fraction bits set the precision

slide credit: Tianjian Li

Exponent vs. Fraction

$$\text{value} = (-1)^s \times 2^{e-127} \times 1.f$$
$$6.5 = 1.625 \times 2^2 = 1.101_2 \times 2^2$$

BF16



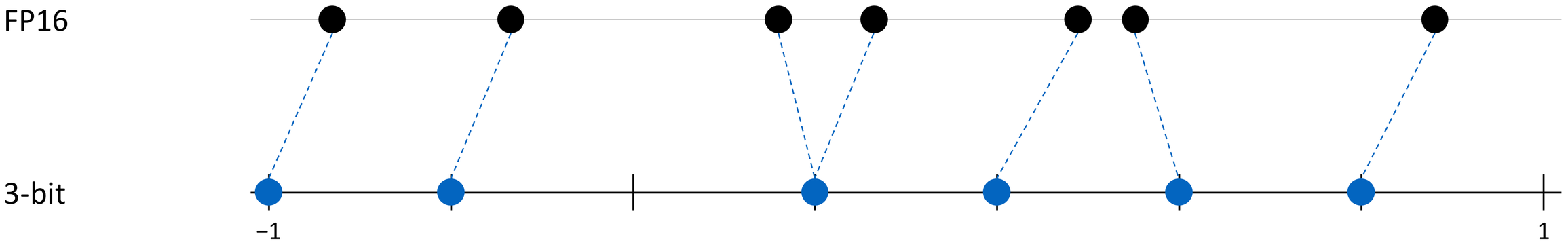
- Exponent bits → range: FP16 max ≈ 65,504, BF16 max ≈ 3.4×10^{38}
- Fraction bits → precision: step near 1 is 2^{-7} (BF16) vs. 2^{-10} (FP16)

LLM Quantization

- Fewer bits per weight: snap each weight to one of 2^b levels

$$q = \text{round}\left(\frac{w}{s}\right) + z, \qquad \hat{w} = s(q - z)$$

- Scale s and zero point z are stored per tensor, row, or group
- Each error is small, but errors add up in the layer output



GPTQ

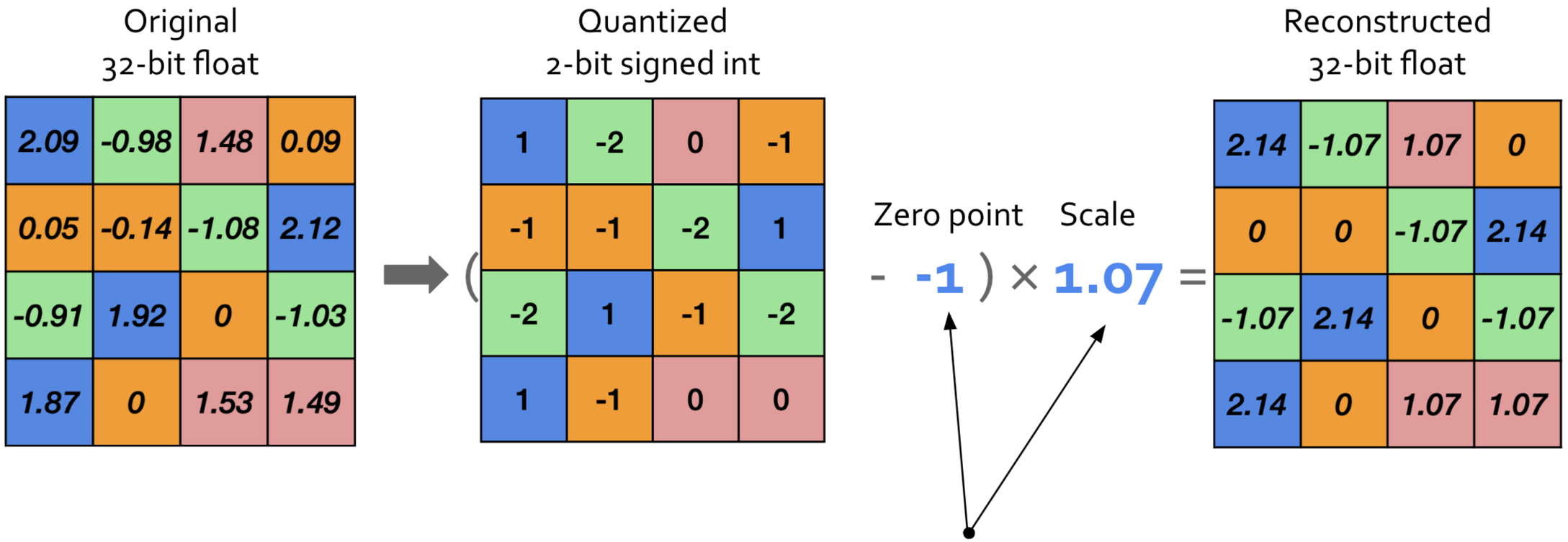
- Quantize one weight at a time; push its error onto the weights not yet quantized
- Goal: keep the layer output the same, $\min\|WX - \hat{W}X\|^2$

Toy example	w_1	w_2	w_3	Output (x = 1)
Original	0.62	0.37	0.12	1.11
Round each weight	0.50	0.25	0.00	0.75
Round + compensate	0.50	0.50	0.00	1.00

- GPTQ: 175B model → 3–4 bits in ~4 GPU hours, little accuracy loss

Frantar et al. (2023), GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers

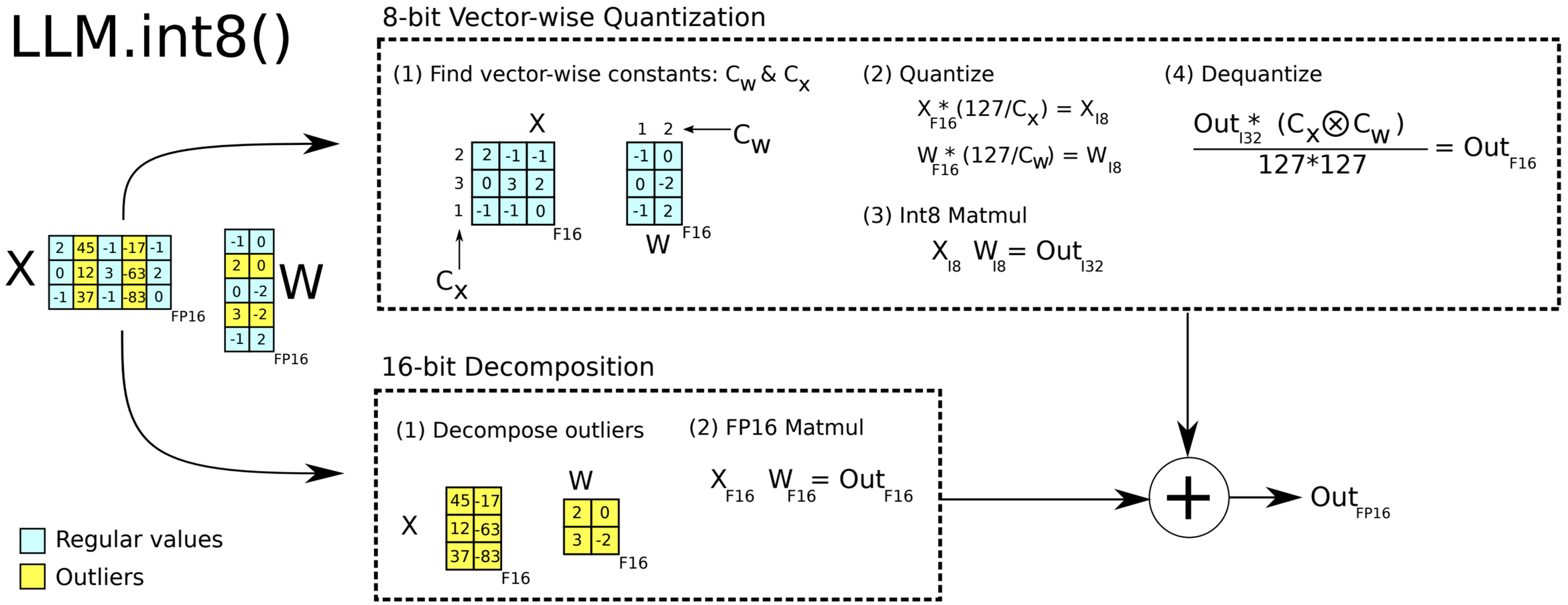
LLM Quantization



- Outlier weights can make it hard to find a good zero point/scale

slide credit: Tianjian Li

LLM Quantization



- Solution: combine 8-bit and 16-bit quantization, where most stuff is 8-bit quantized

Dettmers et al. (2022)

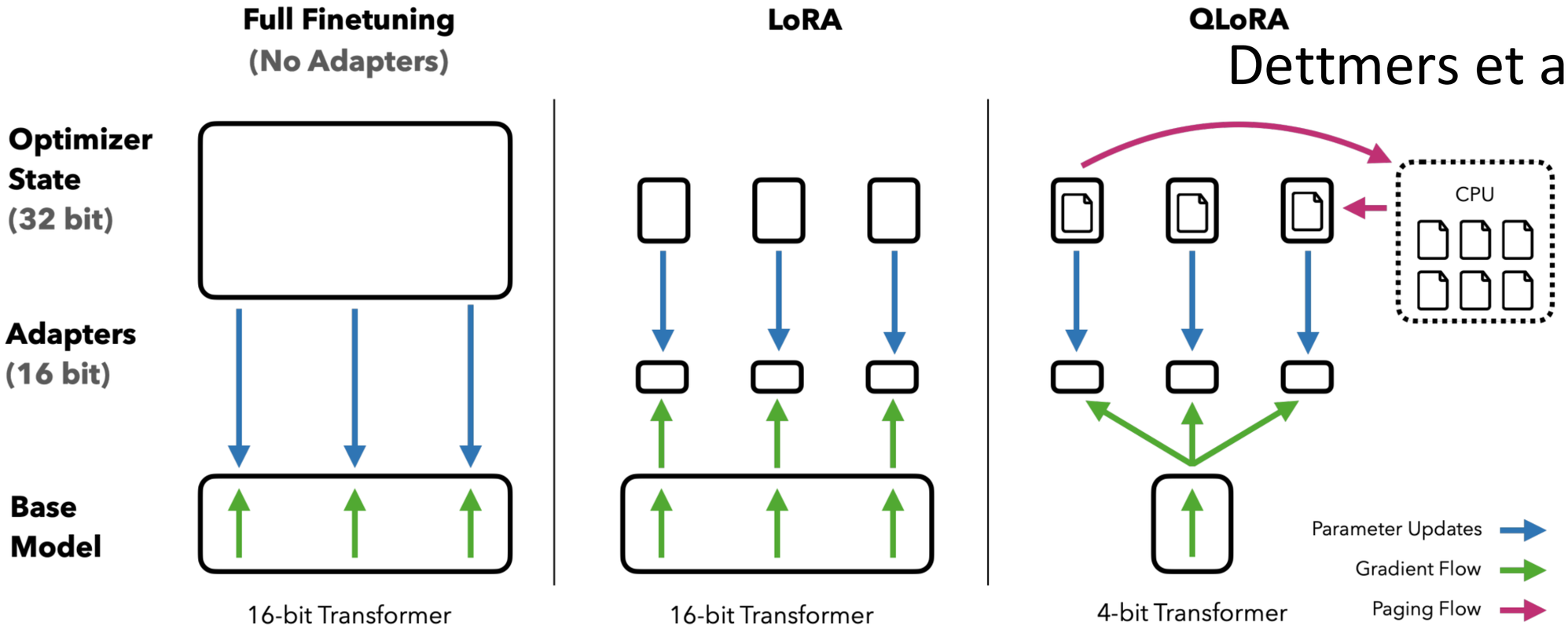
LLM Quantization

Parameters	125M	1.3B	2.7B	6.7B	13B
32-bit Float	25.65	15.91	14.43	13.30	12.45
Int8 absmax	87.76	16.55	15.11	14.59	19.08
Int8 zeropoint	56.66	16.24	14.76	13.49	13.94
Int8 absmax row-wise	30.93	17.08	15.24	14.13	16.49
Int8 absmax vector-wise	35.84	16.82	14.98	14.13	16.48
Int8 zeropoint vector-wise	25.72	15.94	14.36	13.38	13.47
Int8 absmax row-wise + decomposition	30.76	16.19	14.65	13.25	12.46
Absmax LLM.int8() (vector-wise + decomp)	25.83	15.93	14.44	**13.24**	**12.45**
Zeropoint LLM.int8() (vector-wise + decomp)	**25.69**	**15.92**	**14.43**	**13.24**	**12.45**

- Validation perplexity on language modeling. Prior Int8 techniques degrade, the decomposition maintains performance

Dettmers et al. (2022)

QLoRA: Memory-efficient training



Dettmers et al. (2023)

- 4-bit “normal float”, takes advantage of the fact that NN weights typically have a zero-centered normal distribution
- Paged optimizer state to avoid memory spikes (due to training examples with long sequence length)



Where is this going?

- **Better GPU programming:** as GPU performance starts to saturate, we'll probably see more algorithms tailored very specifically to the affordances of the hardware
- **Better hardware:** wafer-scale chips, hardwiring neural architectures

- **Small models**, either distilled or trained from scratch: as LLMs gets better, we can do with ~7B scale what used to be only doable with ChatGPT (GPT-3.5)

- **Continued focus on faster inference**: faster inference can be highly impactful across all LLM applications



Takeaways

- Decoding optimizations: speculative decoding gives a fast way to exactly sample from a smaller model. Also techniques like Flash Attention
- Model optimizations to make models smaller: pruning, distillation
- Model compression and quantization: standard compression techniques, but adapted to work really well for GPUs