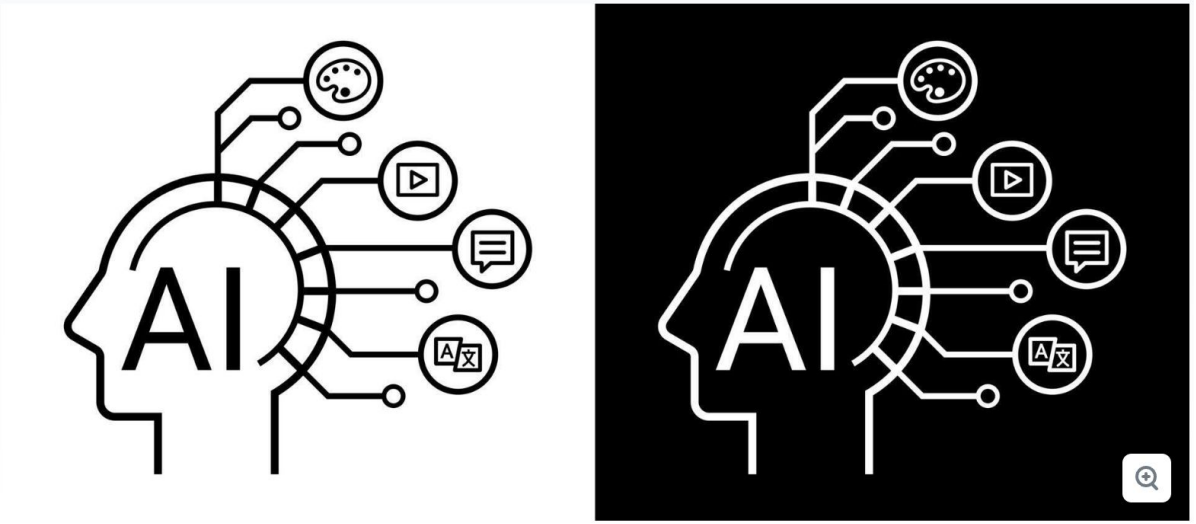
AIMS 5740
Generative Artificial Intelligence

(2026 Term 2)



Computer Science & Engineering
The Chinese University of Hong Kong

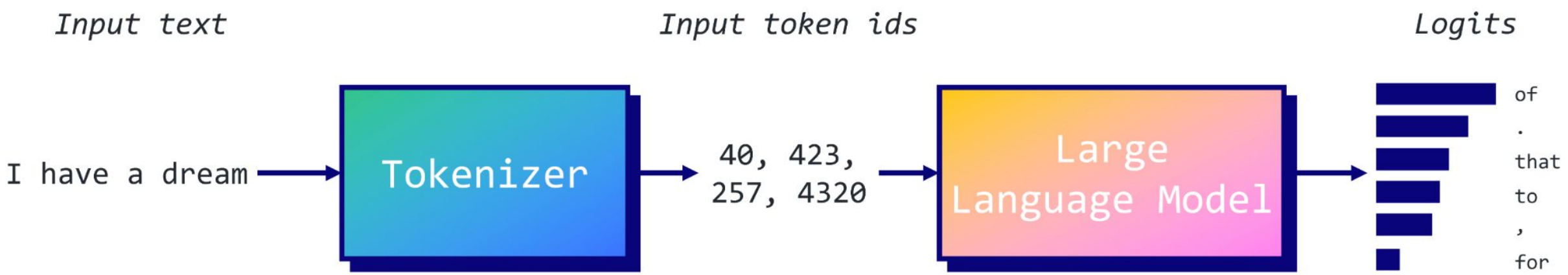
Announcement

- The first assignment is due on Feb. 22. Submit your code/short report via Blackboard (No plagiarism).
- The topics of the final project will be released on Feb. 26 (form a team before Feb. 28, maximum:4).
- The mid-term quiz is scheduled on Mar. 18 (you can only bring the course slides)

LLM Inference - Decoding

- We'll feed the text "I have a dream" to a GPT-2 model and ask it to generate the next five tokens

$$
\begin{aligned}
P(w) &= P(w_1, w_2, \ldots, w_t) \\
&= P(w_1)P(w_2|w_1)P(w_3|w_2, w_1)\ldots P(w_t|w_1, \ldots, w_{t-1}) \\
&= \prod_{i=1}^{t} P(w_i|w_1, \ldots, w_{i-1}).
\end{aligned}
$$



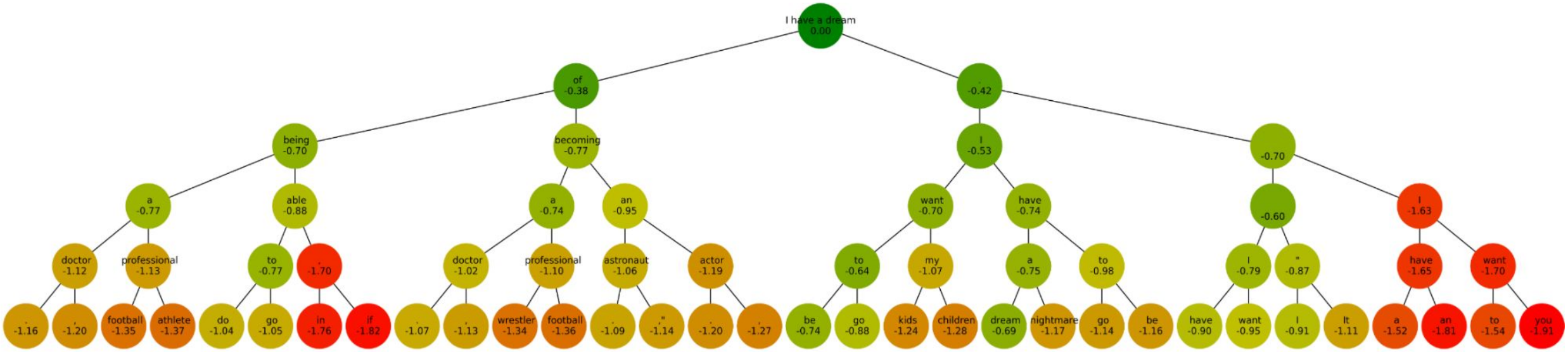
LLM Inference - Decoding

- **Greedy search** is a decoding method that takes the most probable token at each step as the next token in the sequence

- **Step 1**: Input: “I have a dream” → Most likely token: ” of”
- **Step 2**: Input: “I have a dream of” → Most likely token: ” being”
- **Step 3**: Input: “I have a dream of being” → Most likely token: ” a”
- **Step 4**: Input: “I have a dream of being a” → Most likely token: ” doctor”
- **Step 5**: Input: “I have a dream of being a doctor” → Most likely token: “.”

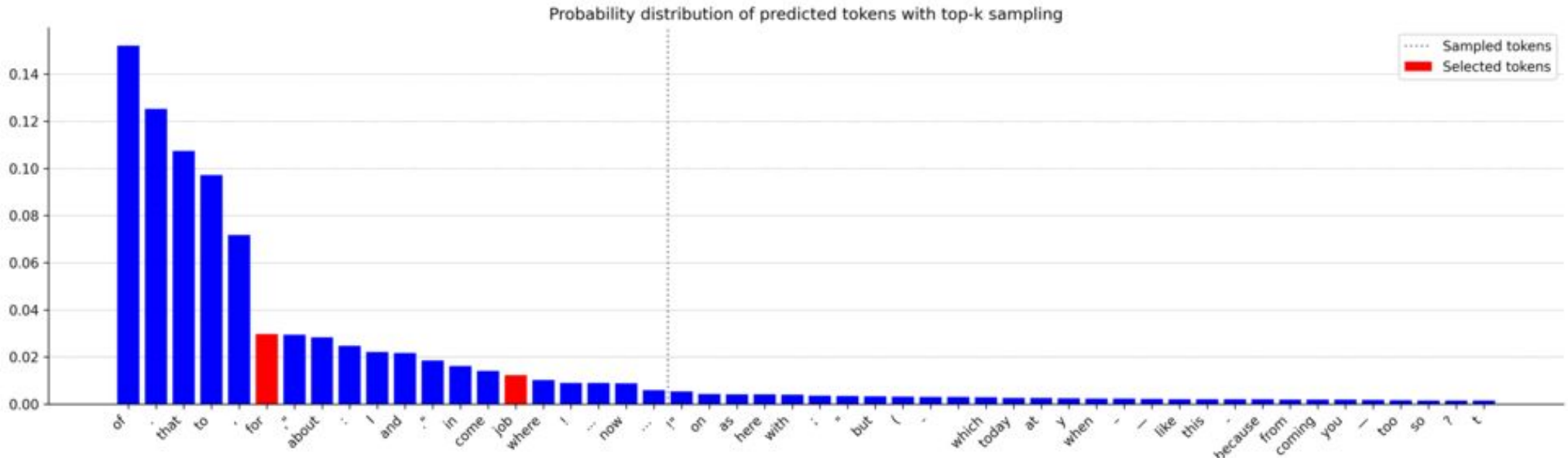
LLM Inference - Decoding

- **Beam search** takes into account the n n most likely tokens, where n, n represents the number of beams.



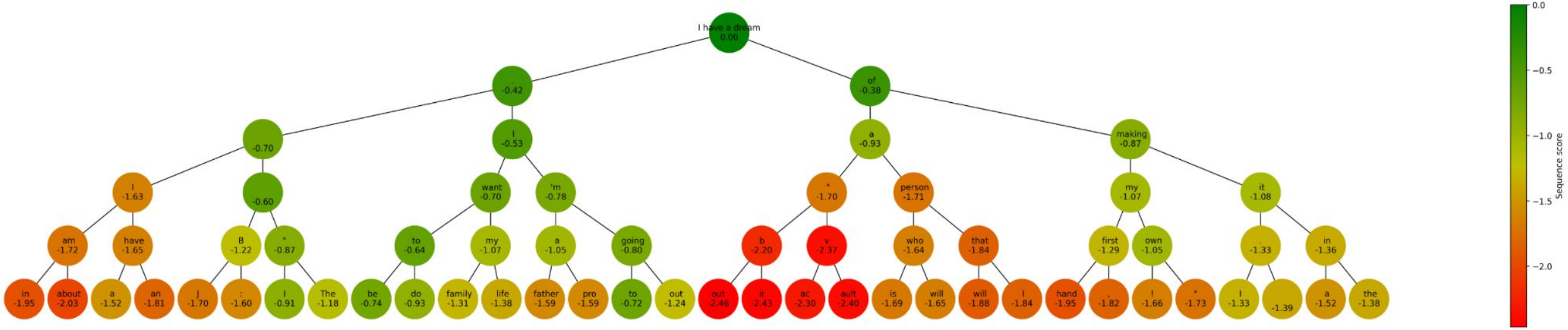
LLM Inference - Decoding

- **Top-k sampling** is a technique that leverages the probability distribution generated by the language model to select a token randomly from the k most likely options.



LLM Inference - Decoding

- Rather than selecting the top k most probable tokens, **nucleus sampling** chooses a cutoff value p such that the sum of the probabilities of the selected tokens exceeds p. This forms a “nucleus” of tokens from which to randomly choose the next token.



Important Metrics for LLM Inference

- **Time To First Token (TTFT)**: How quickly users start seeing the model's output after entering their query.

- **Time Per Output Token (TPOT)**: Time to generate an output token for each user that is querying our system.

- **Latency**: The overall time it takes for the model to generate the full response for a user. latency = (TTFT) + (TPOT) * (the number of tokens to be generated)

- **Throughput:** The number of output tokens per second an inference server can generate across all users and requests.

Two Important Notes

- **Memory Bandwidth is Key**: the speed is dependent on how quickly we can load model parameters from GPU memory to local caches/registers, rather than how quickly we can compute on loaded data

- **How optimized is an LLM inference server?** - Memory bandwidth dictates how quickly the data movement happens.

- A new metric called **Model Bandwidth Utilization** (MBU). MBU is defined as (achieved memory bandwidth) / (peak memory bandwidth) where achieved memory bandwidth is ((total model parameter size + KV cache size) / TPOT): MBU values close to 100% imply that the inference system is effectively utilizing the available memory bandwidth

Inference Optimization

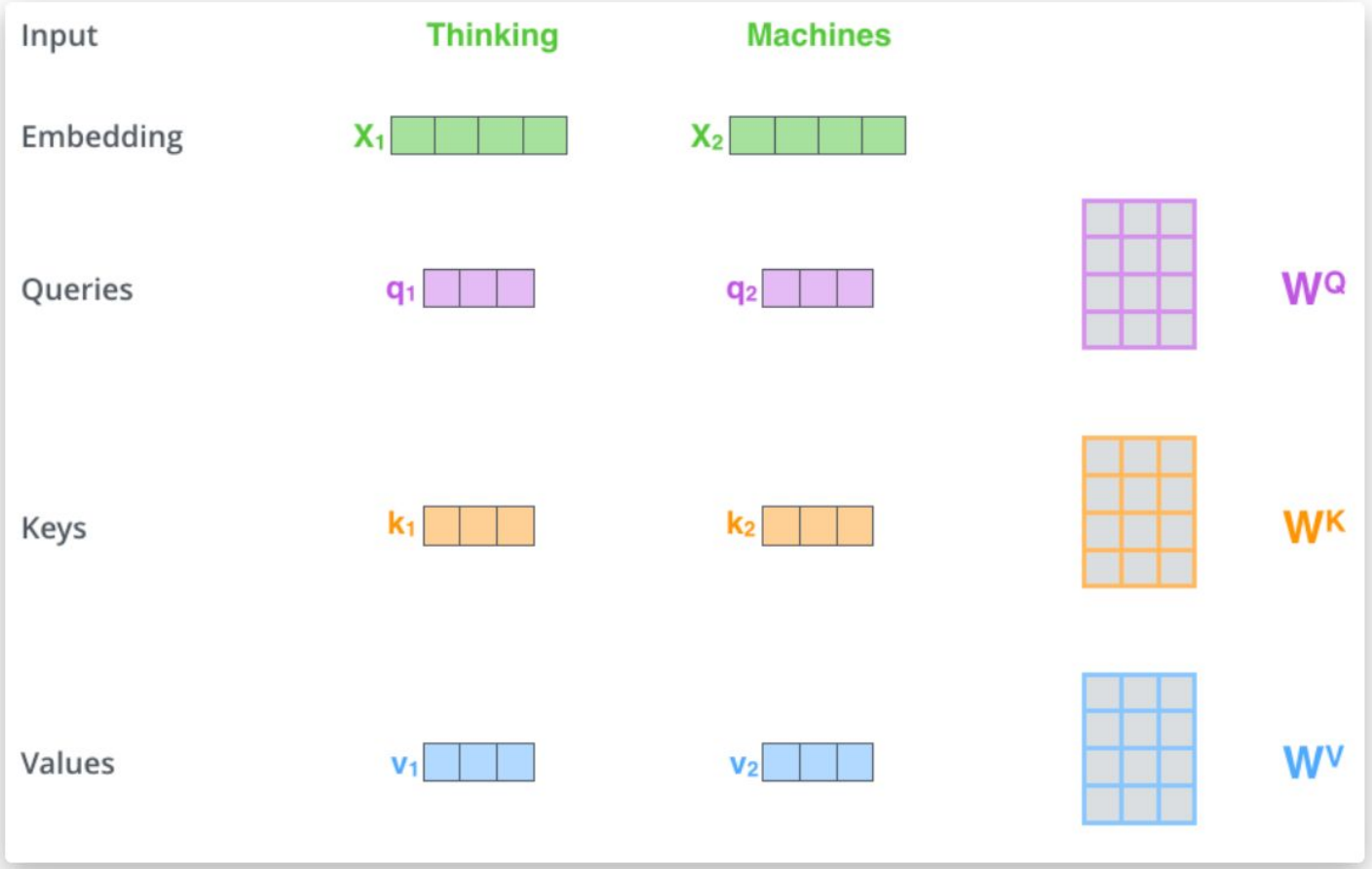
- **Flash Attention**: Optimization of the attention mechanism to transform its complexity from quadratic to linear.

- **Key-value cache**: Understand the key-value cache and the improvements introduced in Multi-Query Attention (MQA) and Grouped-Query Attention (GQA).

- **Speculative decoding:** Use a small model to produce drafts that are then reviewed by a larger model to speed up text generation. EAGLE-3 is a particularly popular solution..

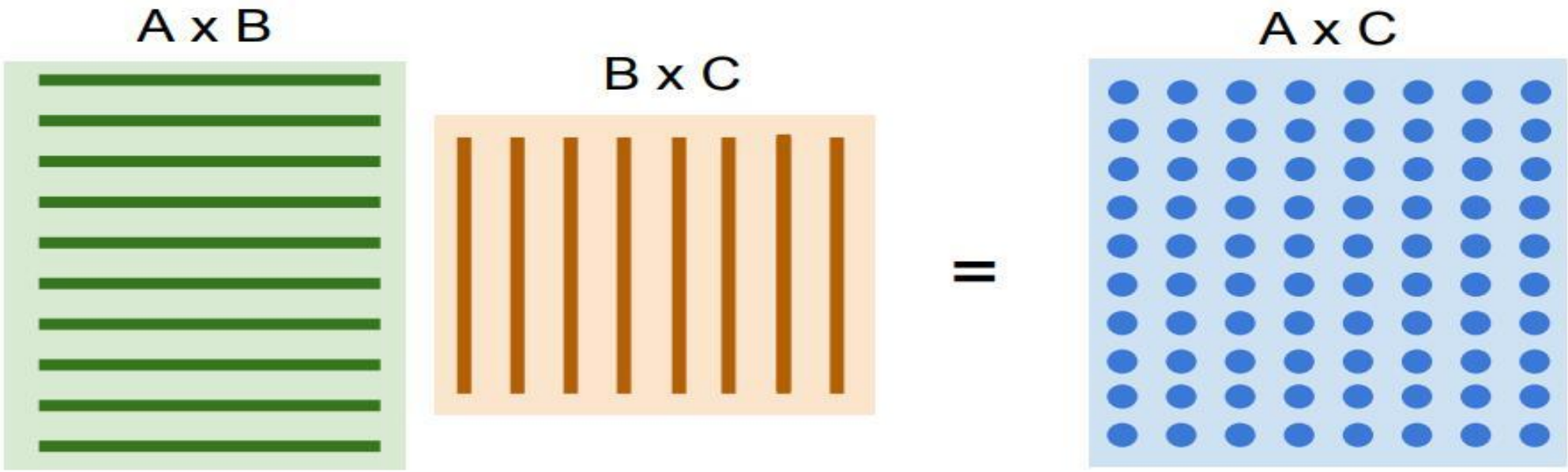
Self-Attention

$$
\begin{aligned}
\mathrm{Q} &= \mathrm{W_q X} \\
\mathrm{K} &= \mathrm{W_k X} \\
\mathrm{V} &= \mathrm{W_v X}
\end{aligned}
$$



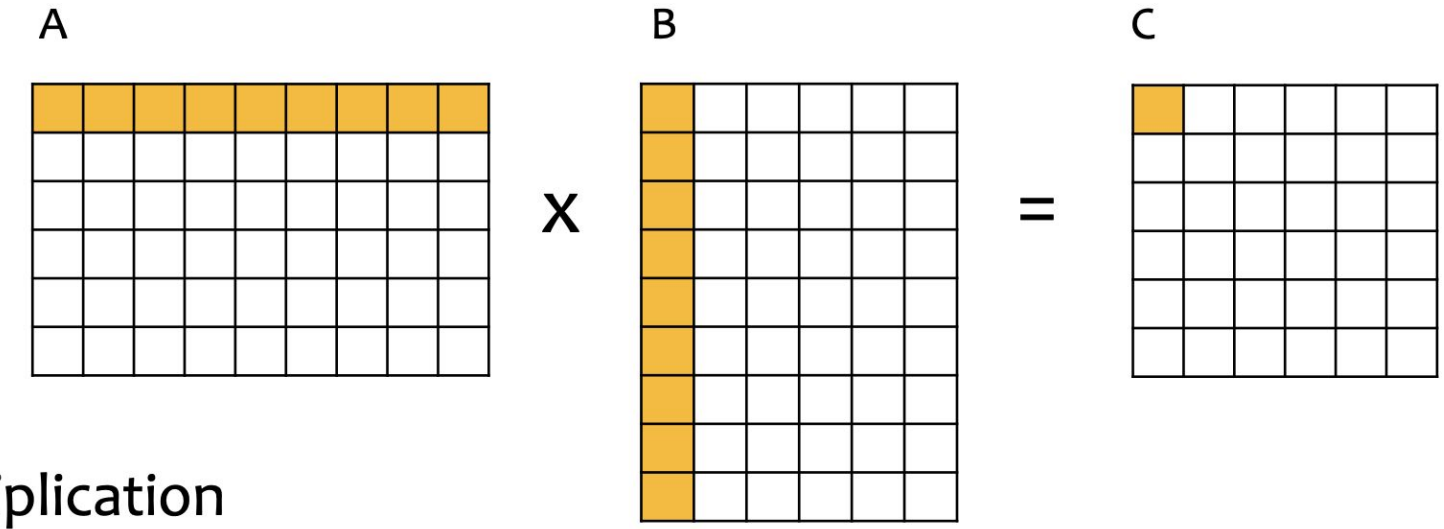
Flash Attention (1)

- **Motivation**: On long sequences, transformers are slow and require significant memory.
- **The problem**: Self attention has quadratic time/memory complexity.



Flash Attention (2)

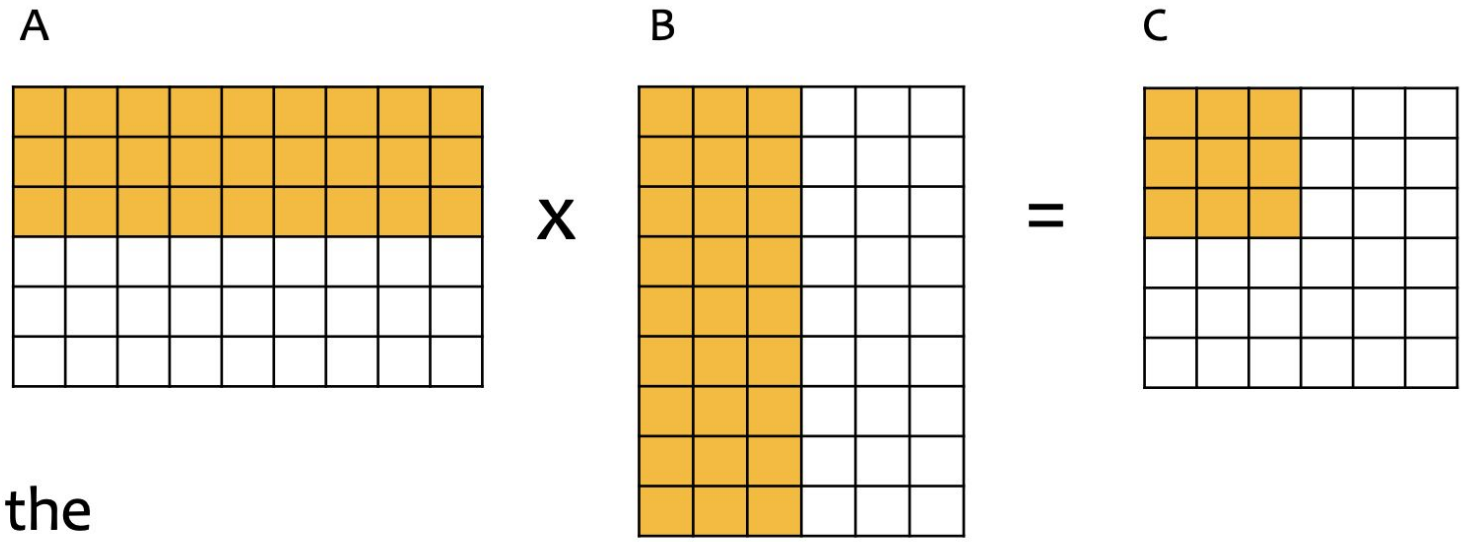
- Tiling for Matrix Multiplication



- Matrix multiplication computes each output value as a dot-product of a row/column pair from the input matrices

$$C_{ij} = \sum_{m=1}^{M} \sum_{n=1}^{N} A_{im} B_{nj}$$

Flash Attention (3)

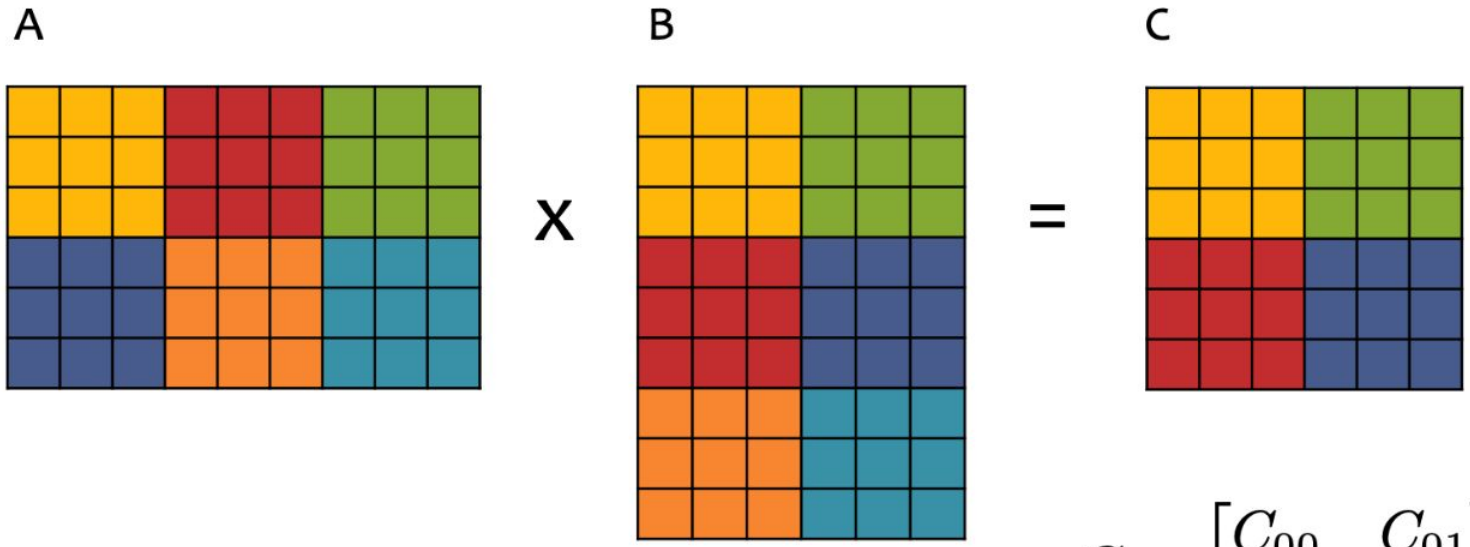


- We can view the computation as decomposing if we consider subsets of rows/columns

$$C_{(1,1):(3,3)} = A_{(1,1):(3,9)} \times B_{(1,1):(9,3)}$$

Flash Attention (4)

- Tiling capitalizes on this decomposition
- Each output tile is computed by multiplying a pair of input tiles and adding it to the appropriate output tile



$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix}$$

with each $A_{ij} \in \mathbb{R}^{3\times3}$

$$B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \\ B_{20} & B_{21} \end{bmatrix}$$

with each $B_{ij} \in \mathbb{R}^{3\times3}$

$$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

with each $C_{ij} \in \mathbb{R}^{3\times3}$

$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$

$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$

$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$

$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$

Flash Attention (5)

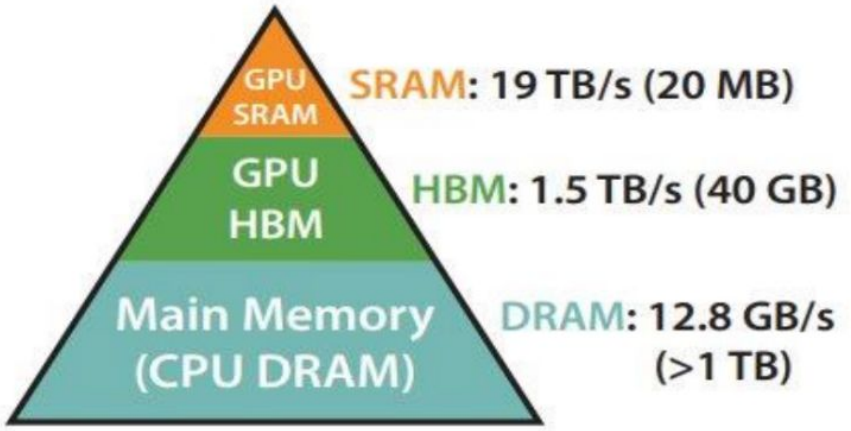
- Memory is arranged hierarchically

FlashAttention Approach

Reorganize the Attention computation to access the slow memory as little as possible

Memory Hierarchy

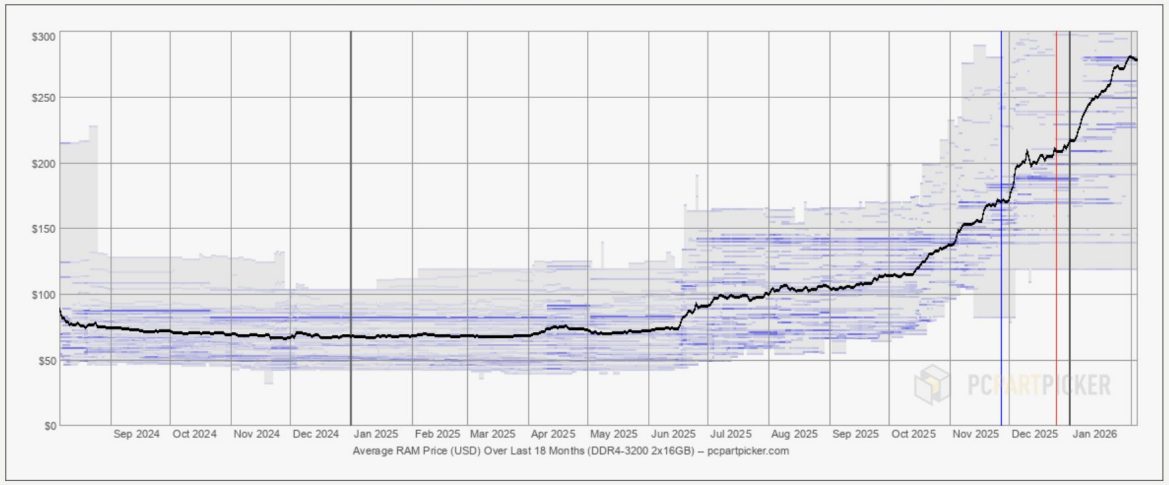
In the GPU memory hierarchy, the HBM memory is the slow memory we want to avoid accessing as much as possible



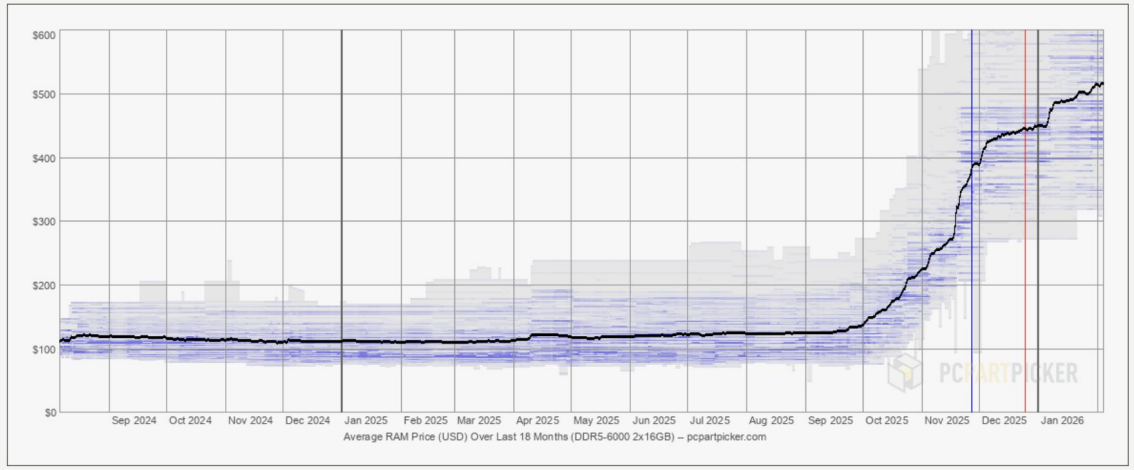
Memory Hierarchy with Bandwidth & Memory Size

Memory

DDR4-3200 2x16GB (Average price in USD over last 18 months)



DDR5-6000 2x16GB (Average price in USD over last 18 months)



micron

Micron Technology Inc

NASDAQ: MU

419.44 USD

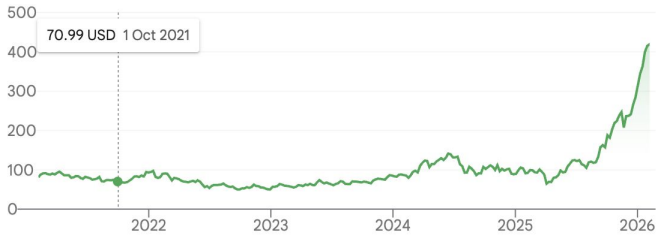
+ Follow

+338.34 (417.19%) ↑ past 5 years

Closed: 3 Feb, 7:59 pm GMT-5 • Disclaimer

After hours 413.60 −5.84 (1.39%)

1D | 5D | 1M | 6M | YTD | 1Y | 5Y | Max



Open	442.16	Mkt cap	472.08B	Dividend	0.11%
High	442.30	P/E ratio	39.87	Qtrly Div Amt	0.12
Low	407.53	52-wk high	455.48	52-wk low	61.54

SK hynix

SK Hynix Inc

KRX: 000660

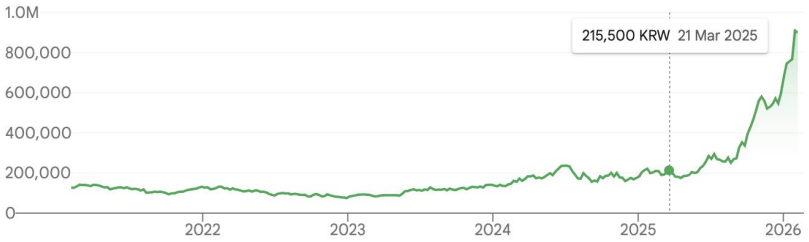
900,000 KRW

+ Follow

+772,500.00 (605.88%) ↑ past 5 years

4 Feb, 3:30 pm GMT+9 • Disclaimer

1D | 5D | 1M | 6M | YTD | 1Y | 5Y | Max



Open	884,000	Mkt cap	655.20T	Dividend	0.33%
High	906,000	P/E ratio	17.70	Qtrly Div Amt	742.50
Low	883,000	52-wk high	931,000	52-wk low	162,700

Flash Attention (5)

- Two key ideas are combined to obtain FlashAttention

- Both are well-established ideas, so the interesting part is how they are put together for attention
  1. **Tiling**: compute the attention weights block by block so that we don't have to load everything into SRAM at once
  2. **Recomputation**: don't ever store the full attention matrix, but just recompute the parts of it you need during the backward pass

Flash Attention (6)

- FlashAttention Benefit one: Faster Attention

Attention	Standard	FLASHATTENTION
GFLOPs	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtime (ms)	41.7	7.3

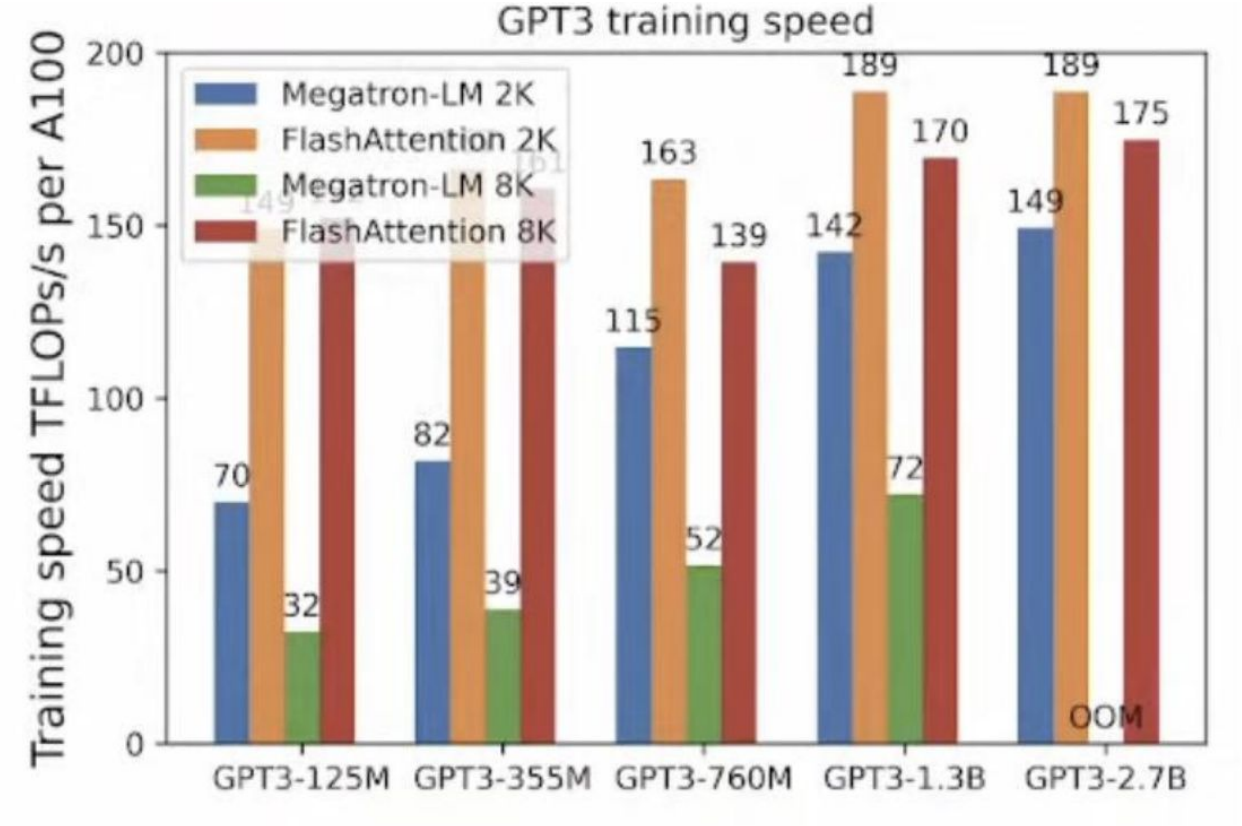
FLOPs versus HBM accesses of FlashAttention and the default PyTorch implementation at the time. Performance on an A100 GPU (seq. length 1024, head dim. 64, 16 heads, batch size 64)

Model implementations	Training time (speedup)
GPT-2 small - Huggingface [87]	9.5 days (1.0×)
GPT-2 small - Megatron-LM [77]	4.7 days (2.0×)
GPT-2 small - FLASHATTENTION	**2.7 days (3.5×)**
GPT-2 medium - Huggingface [87]	21.0 days (1.0×)
GPT-2 medium - Megatron-LM [77]	11.5 days (1.8×)
GPT-2 medium - FLASHATTENTION	**6.9 days (3.0×)**

Training time with different attention implementations. Training performed on 8 x A100 GPUs

Flash Attention (7)

- FlashAttention Benefit two: Supports Longer Sequences



Training GPT-3 with leading exact Attention implementations with sequence length 2K and 8K

Flash Attention (8)

- FlashAttention has improved versions v2 and v3.

	Optimizations	Speedup Achieved	Utilization of Hardware
Standard Attention	- Optimized Kernels	Baseline	Low (IO Bound)
FlashAttention v1	- Tiling of QK and PV ops. - Recomputation	~7x compared to baseline	Improved
FlashAttention v2	- Reduction of non-matmul operations. - Better work partitioning. - Parallelizing across sequence & heads	~2x compared to v1	70% of theoretical max (A100)
FlashAttention v3	- Async execution of multiple kernels. - Low precision arithmetic. - Optimized memory management.	1.5-2.0x compared to v2	75% of theoretical max (H100)

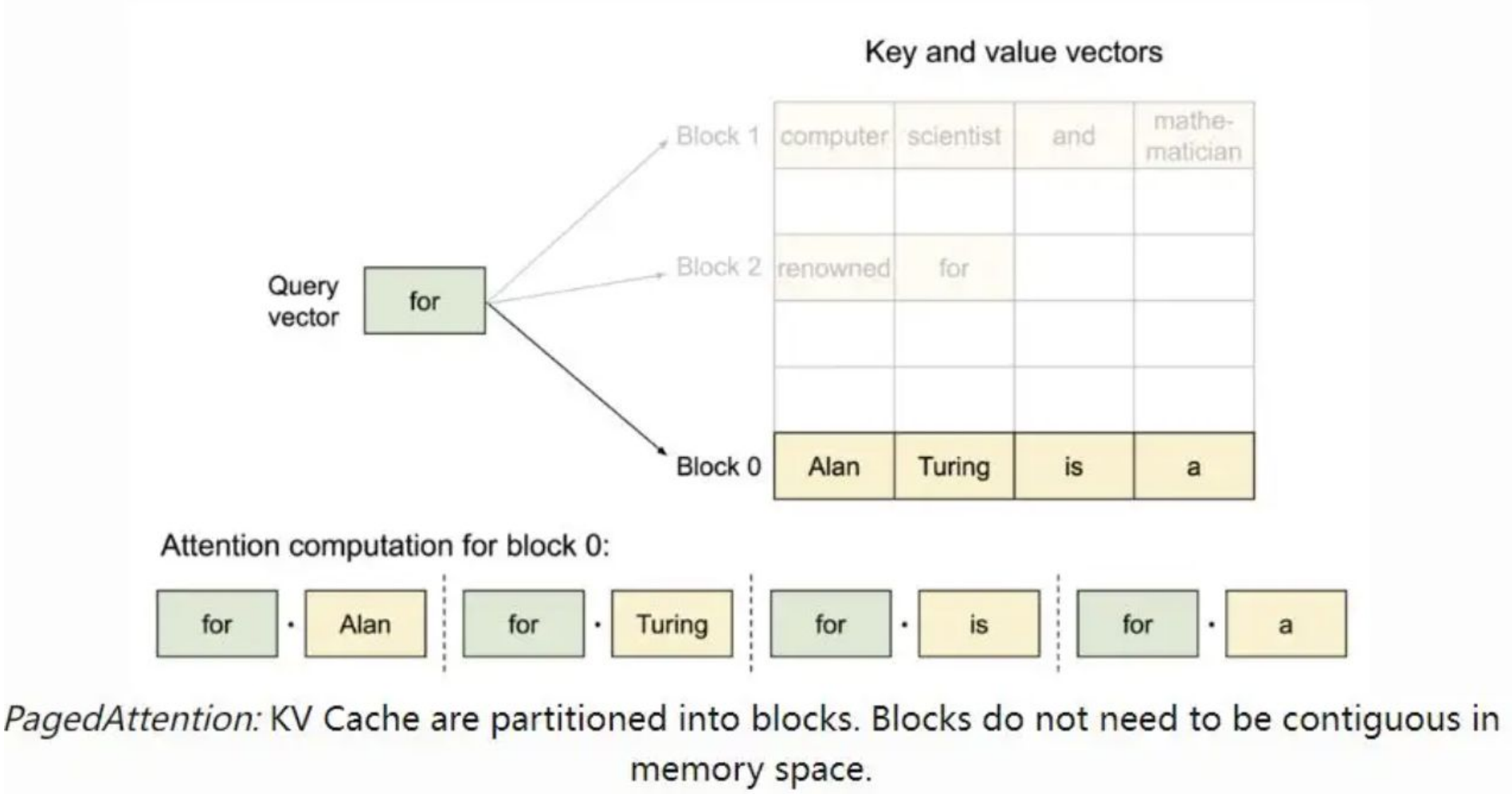
KV Caching (1)

1. "The quick brown fox..."
2. "The quick brown fox jumps..."
3. "The quick brown fox jumps over..."

```
Token 1: [K1, V1] → Cache: [K1, V1]

Token 2: [K2, V2] → Cache: [K1, K2], [V1, V2]

...

Token n: [Kn, Vn] → Cache: [K1, K2, ..., Kn], [V1, V2, ..., Vn]
```

KV Caching (2)

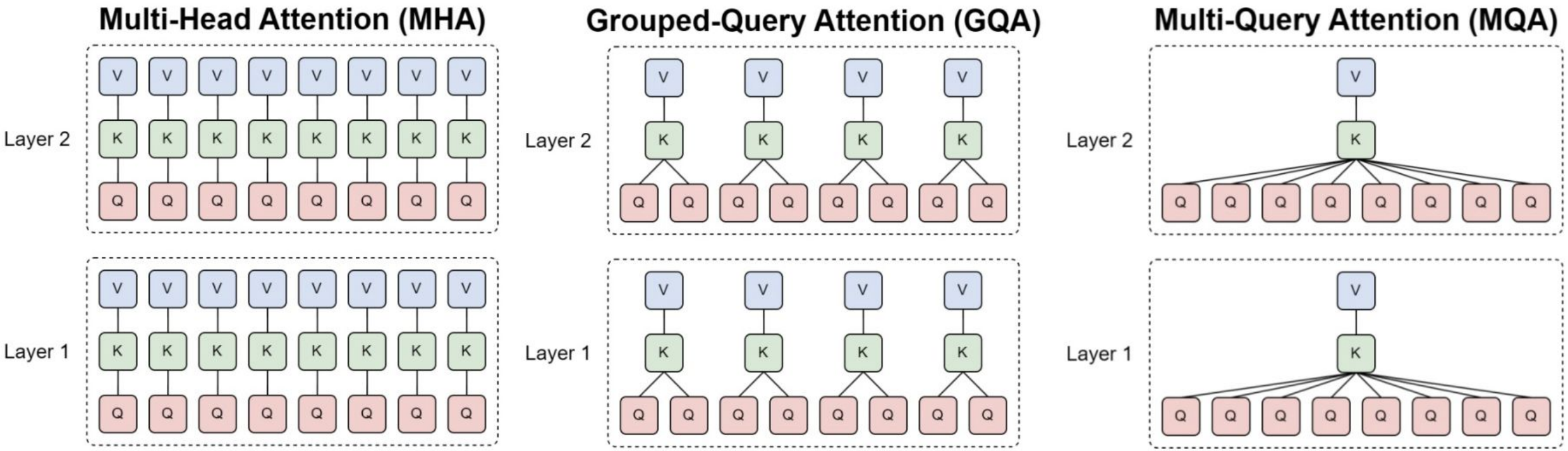
- PageAttention: optimize the kv cache in memory



PagedAttention: KV Cache are partitioned into blocks. Blocks do not need to be contiguous in memory space.

KV Caching (3)

- Multi-Query Attention (MQA) — All heads share the same key-value pairs, reducing memory usage.
- Grouped Query Attention (GQA) — A balance between MHA and MQA, where attention heads are grouped to share key-value pairs.



KV Caching (4)

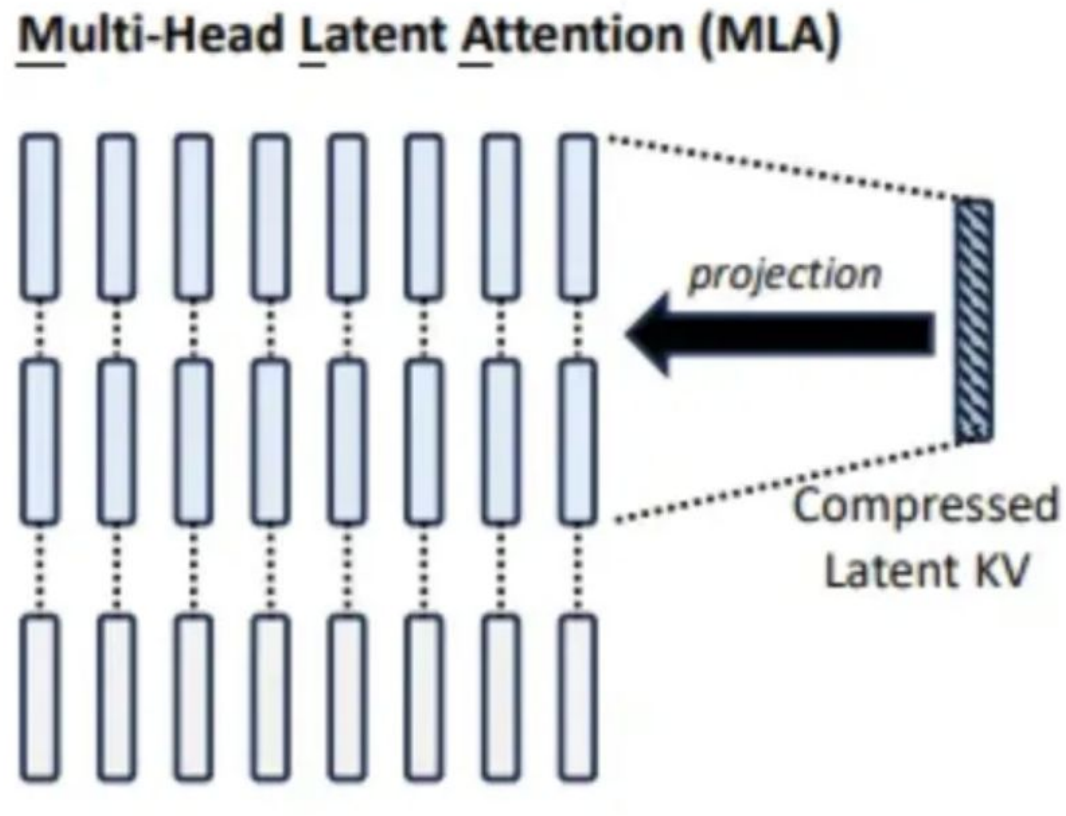
- Compared with MHA, MQA and GQA perform slightly worse, especially in many hard-tasks.

	BoolQ	PIQA	SIQA	Hella-Swag	ARC-e	ARC-c	NQ	TQA	MMLU	GSM8K	Human-Eval
MHA	**71.0**	**79.3**	48.2	75.1	71.2	**43.0**	12.4	44.7	**28.0**	4.9	**7.9**
MQA	70.6	79.0	47.9	74.5	71.6	41.9	**14.5**	42.8	26.5	4.8	7.3
GQA	69.4	78.8	**48.6**	**75.4**	**72.1**	42.5	14.0	**46.2**	26.9	**5.3**	**7.9**

Table 18: Attention architecture ablations. We report 0-shot results for all tasks except MMLU (5-shot) a

KV Caching (4)

- Multi-Head Latent Attention (MLA) jointly compresses the Key and Value into a latent vector.
- The compression dimension is much less compared to the output projection matrix dimension in MHA



KV Caching (4)

- DeepSeek v3 and R1 use MLA

Attention Mechanism	KV Cache per Token (# Element)	Capability
Multi-Head Attention (MHA)	$2n_h d_h l$	Strong
Grouped-Query Attention (GQA)	$2n_g d_h l$	Moderate
Multi-Query Attention (MQA)	$2d_h l$	Weak
MLA (Ours)	$(d_c + d_h^R)l \approx \frac{9}{2} d_h l$	Stronger

Speculative Decoding - Motivation

- Tradeoffs between Different Language Models

# Parameters	175B	13B	2.7B	760M	125M
TriviaQA	**71.2**	57.5	42.3	26.5	6.96
PIQA	**82.3**	79.9	75.4	72.0	64.3
SQuAD	**64.9**	62.6	50.0	39.2	27.5
latency	20 s	7.6s	2.7s	1.1s	**0.3s**
# A100s	10	1	1	1	**1**

Comparing multiple GPT-3 models*

Large models

Pro: better generative performance

Con: slow and expensive to serve

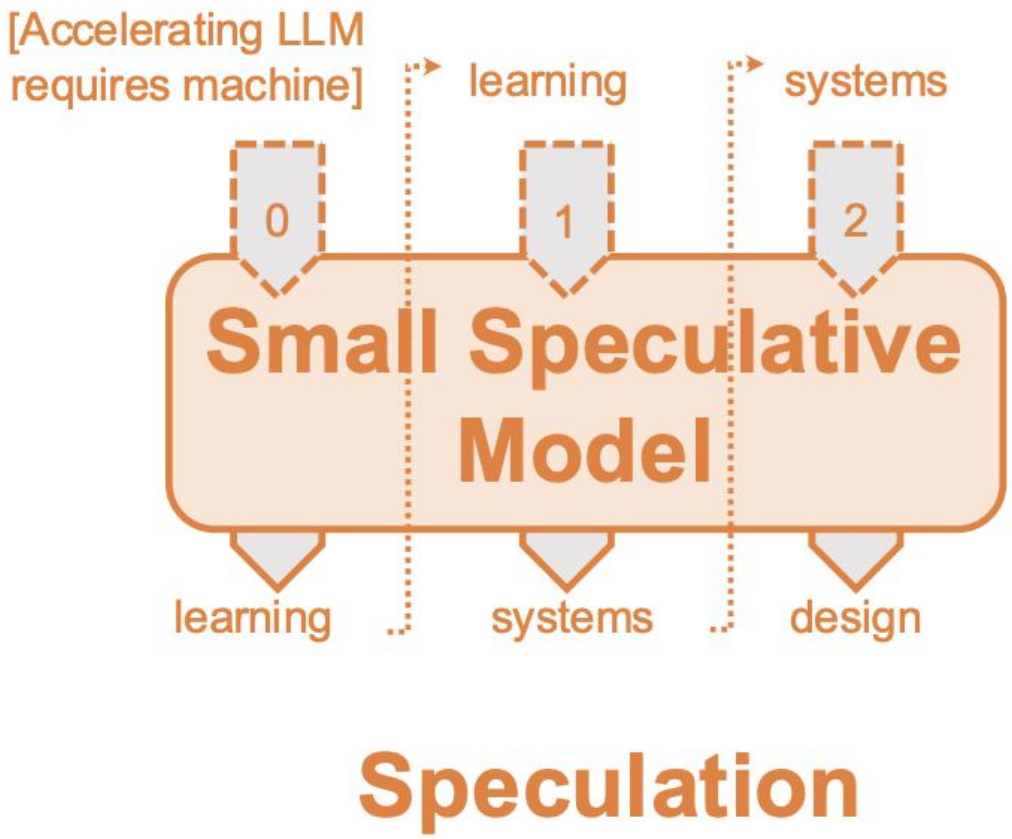
Small models

Pro: cheap and fast

Con: less accurate

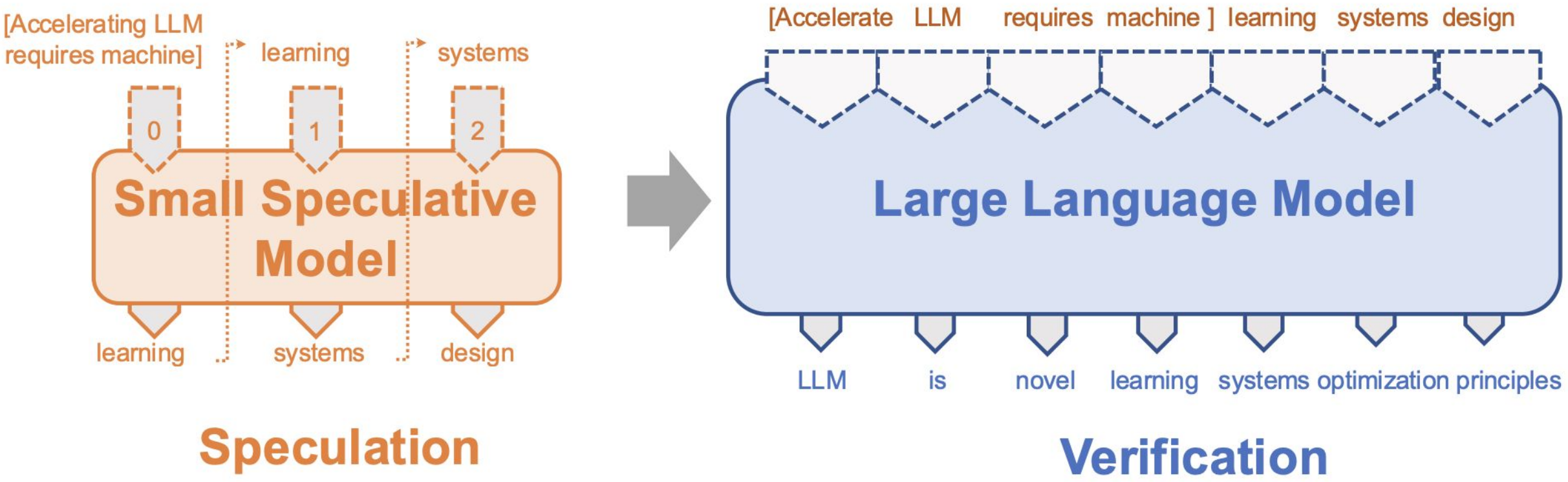
Speculative Decoding (1)

- Use a small speculative model (SSM) to predict the LLM's output
  - SSM runs much faster than LLM



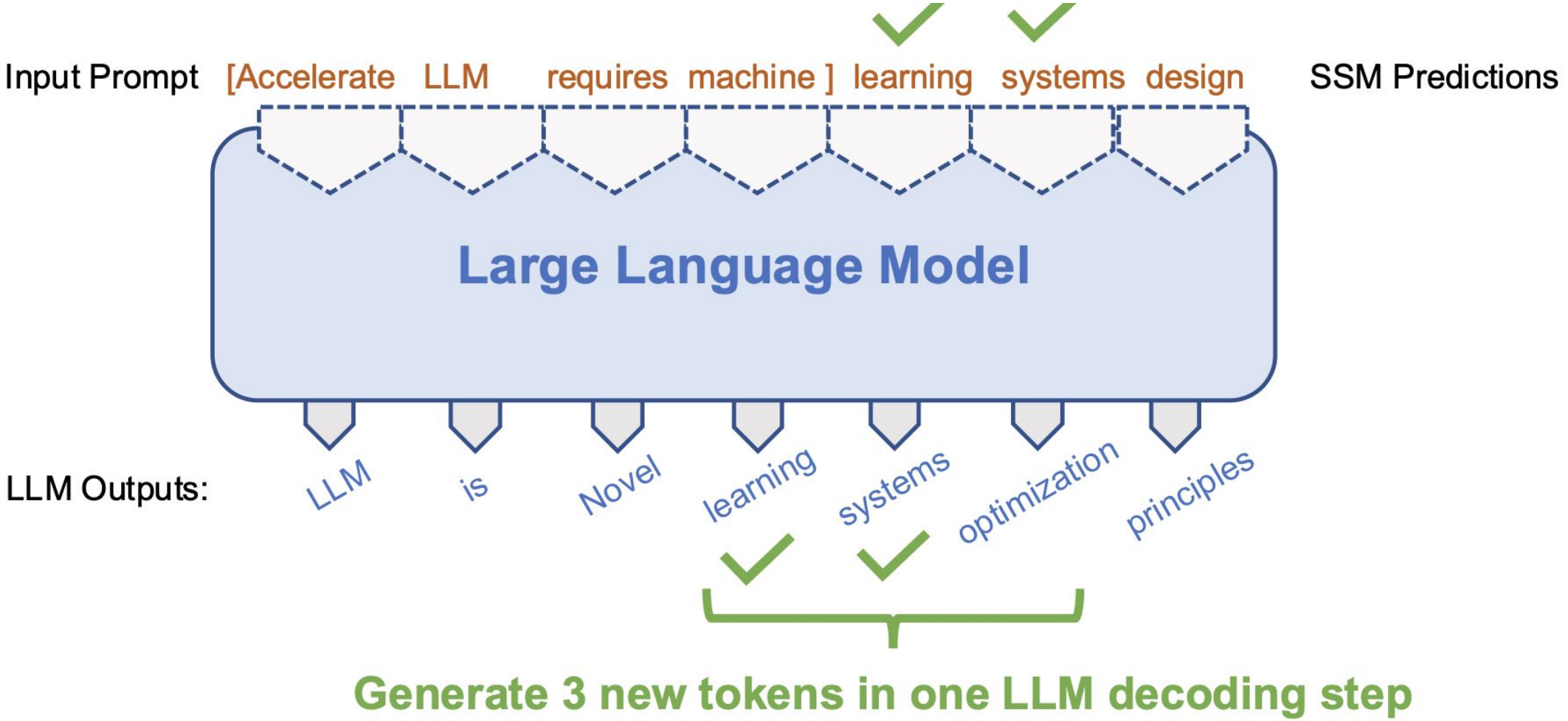
Speculative Decoding (2)

- Use the LLM to verify the SSM's prediction



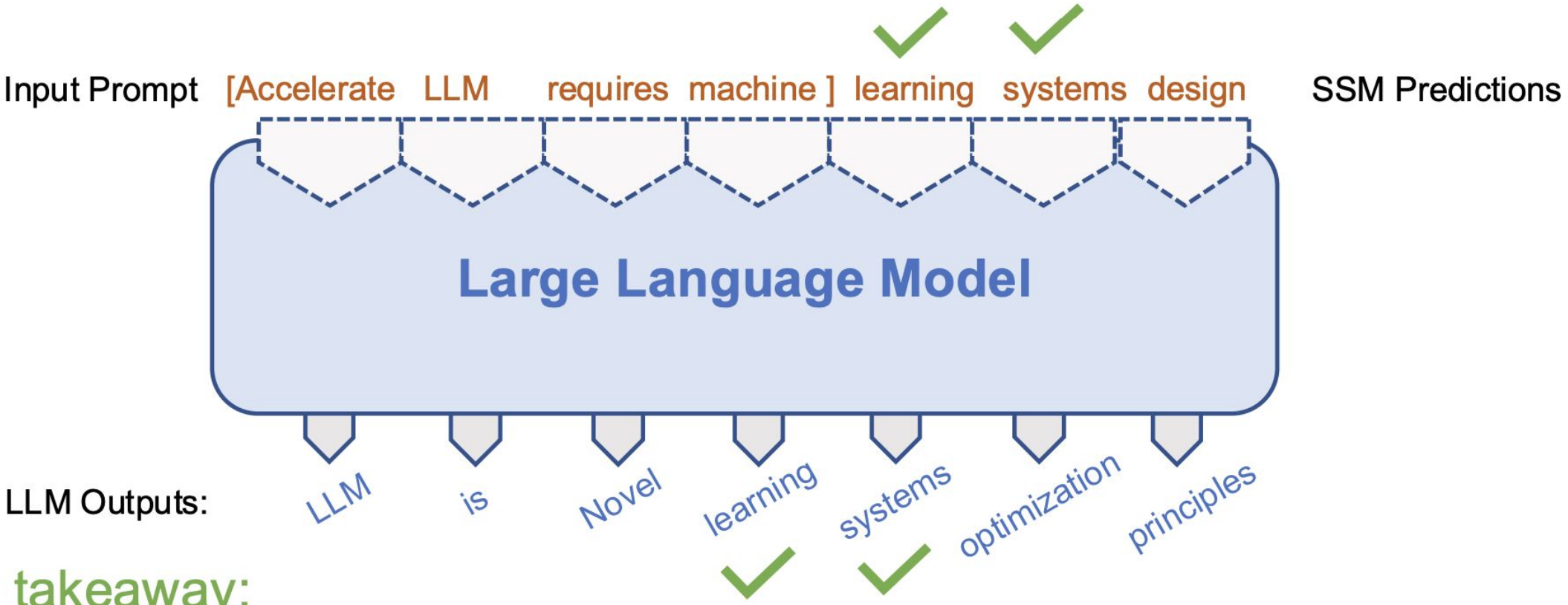
Speculative Decoding (3)

- Verifying Speculative Decoding Results



Speculative Decoding (4)

- Verifying Speculative Decoding Results



Key takeaway:

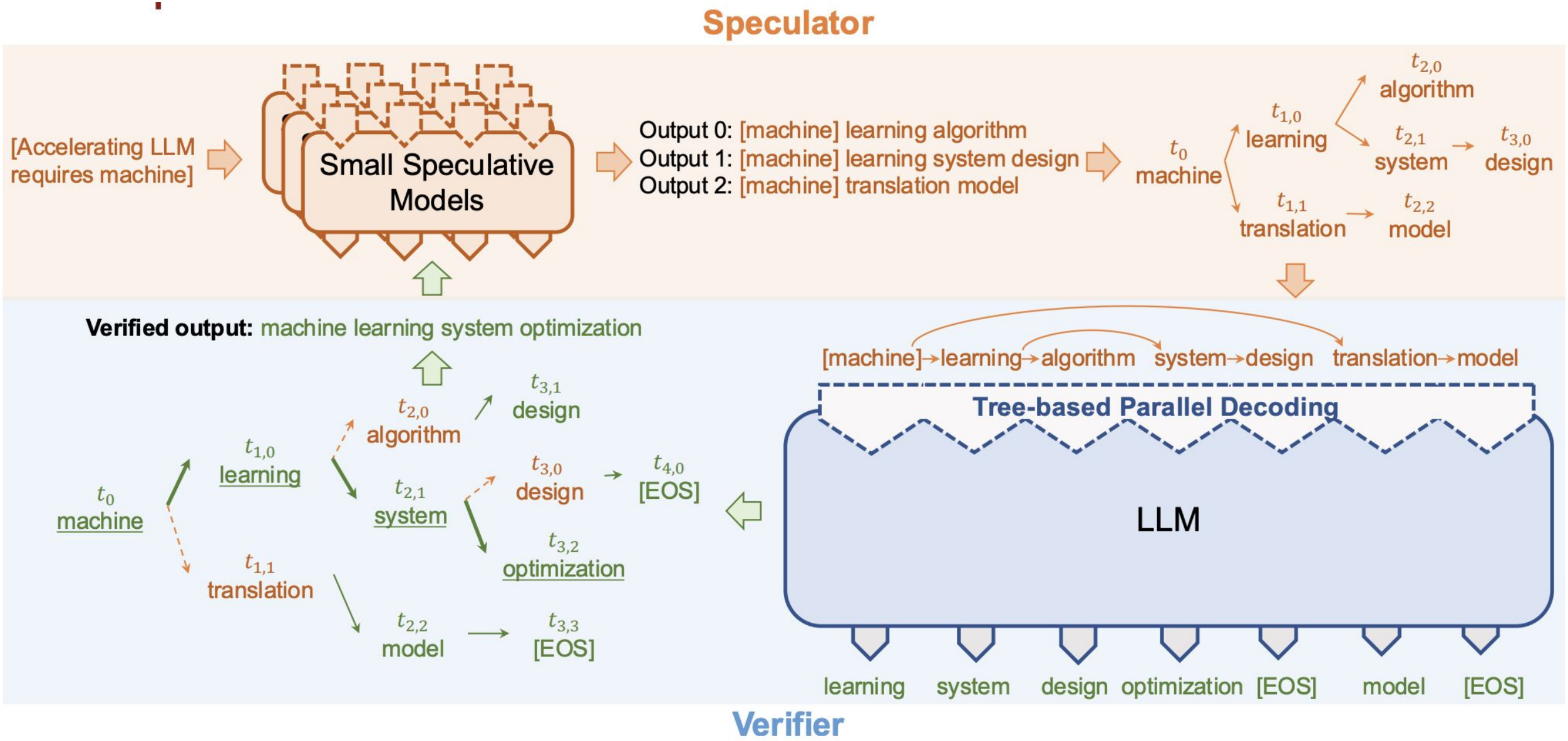
- LLM inference is bottlenecked by accessing model weights
- using LLM to decode multiple tokens to improve GPU utilization

SpecInfer: Tree-based Speculative Inference & Verification

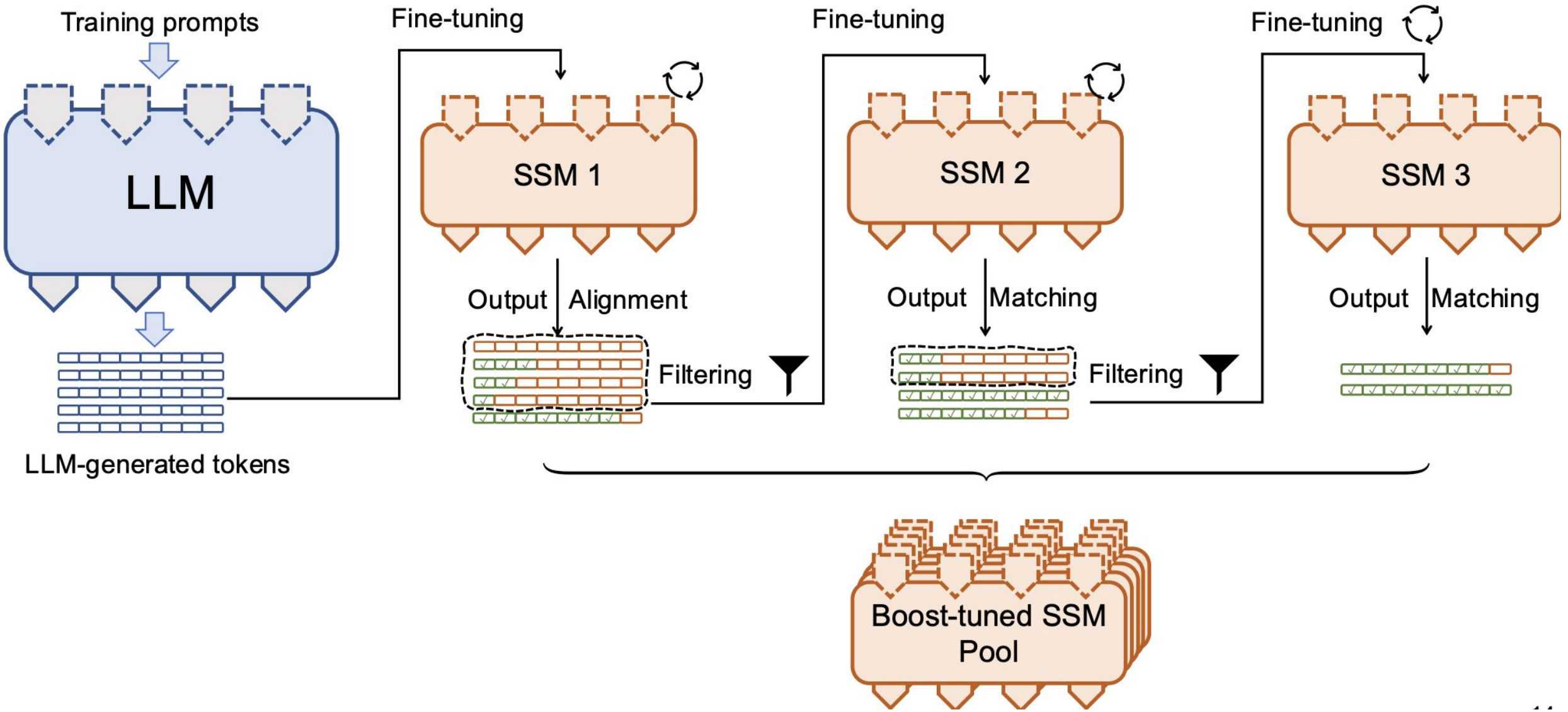
- Key idea: not use LLMs as incremental decoder, use them as parallel token tree verifier

- **Better performance:** outperform existing LLM systems by 1.3-2.4x
- **Higher efficiency:** reduce GPU memory access by 2.5-4.4x
- **Correctness:** verification guarantees end-to-end equivalence

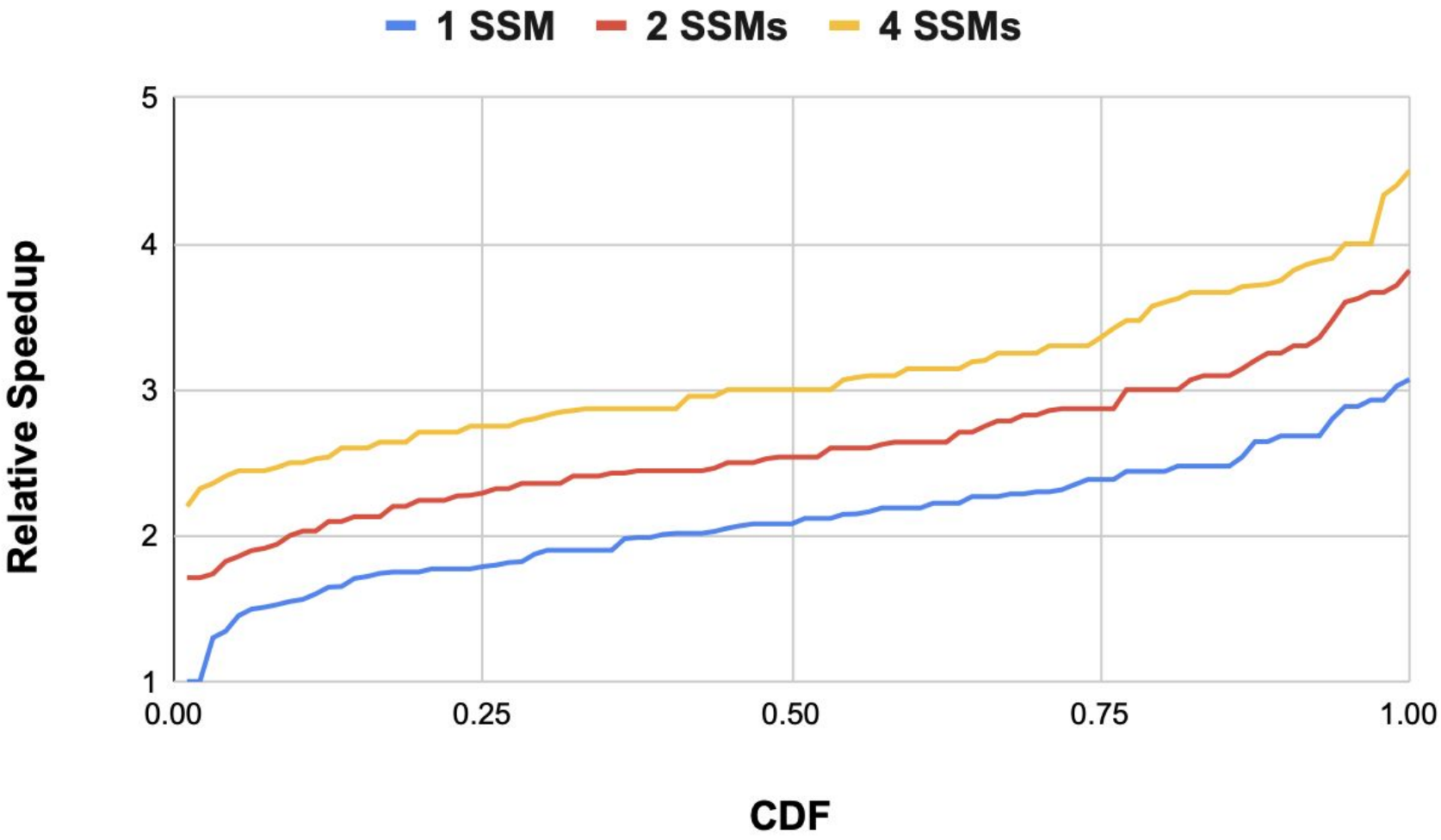
SpecInfer Workflow



Collective Boost-Tuning

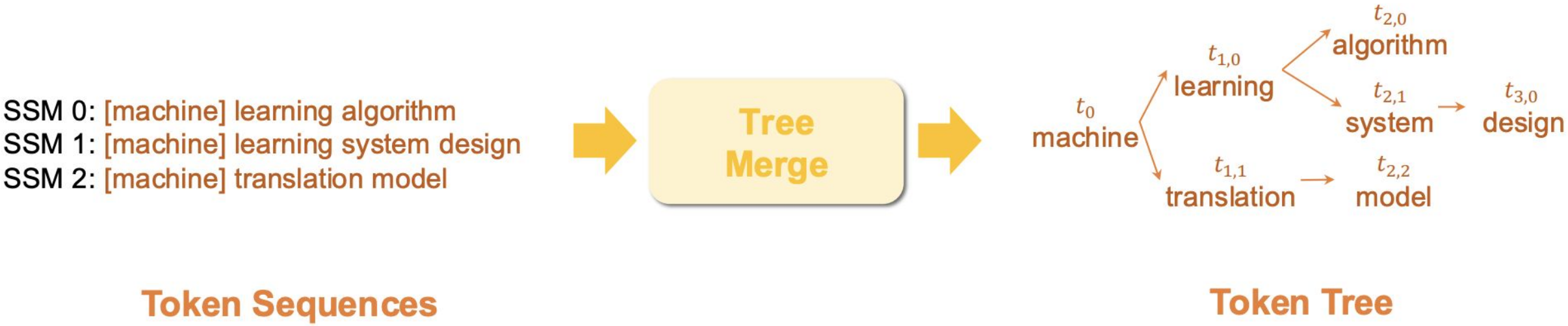


Collective Boost-Tuning

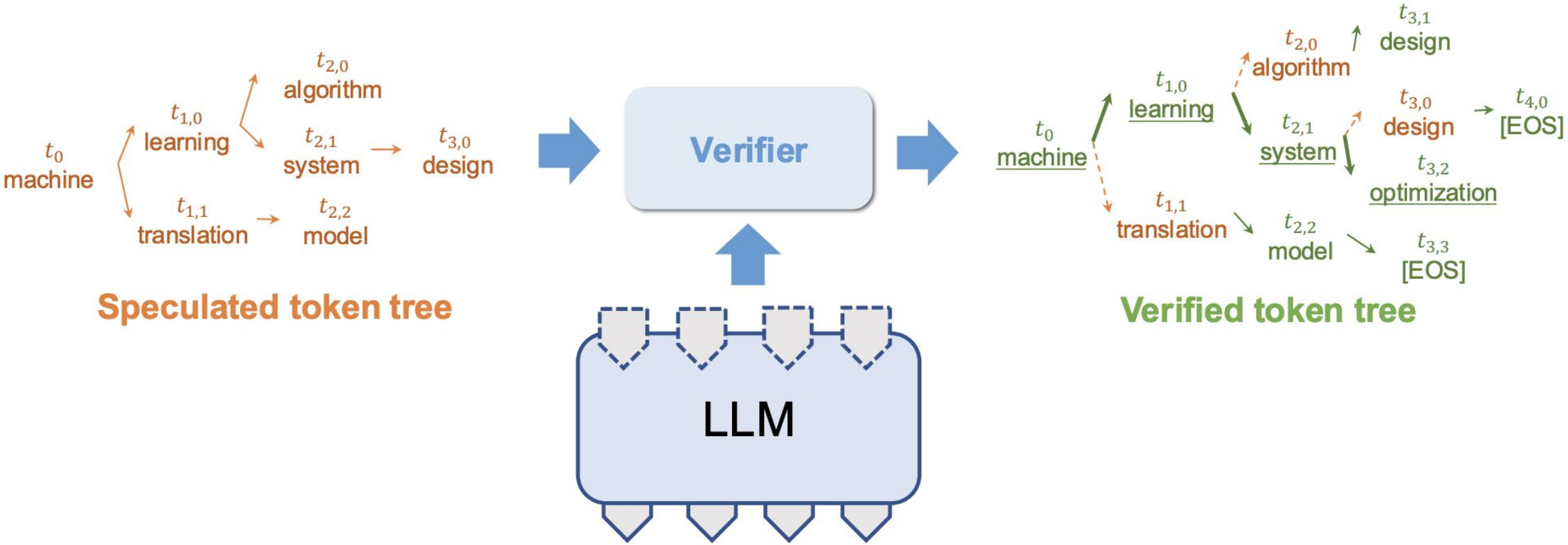


Token Tree Merge

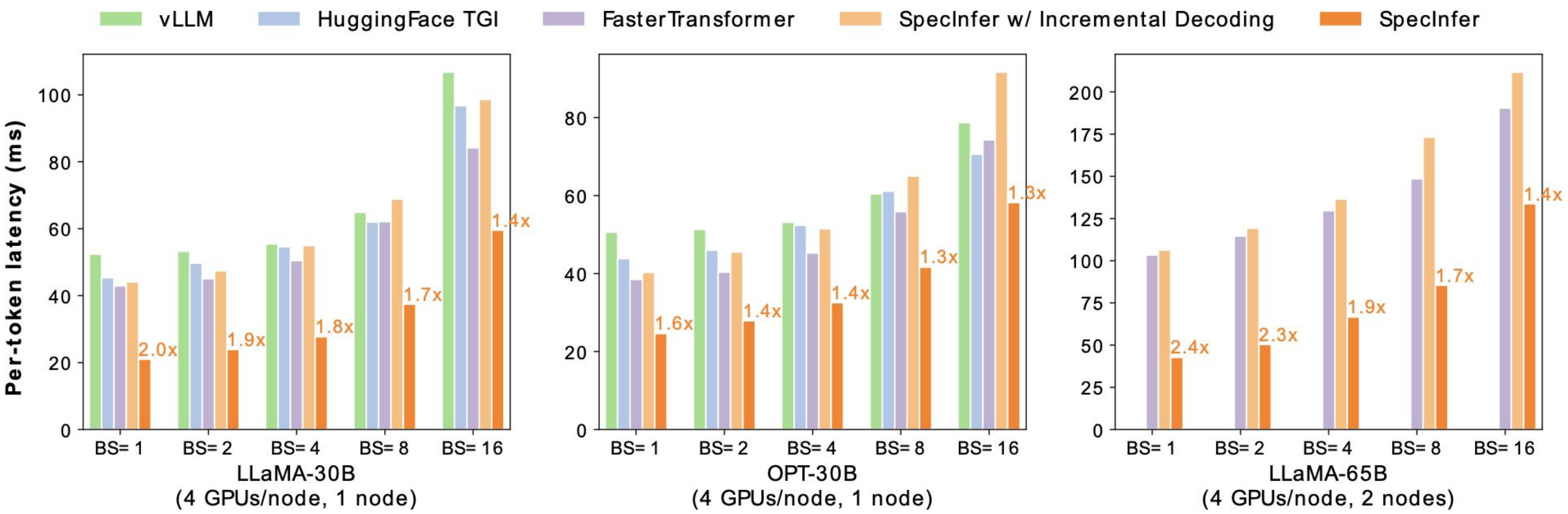
- A compact way to represent speculated tokens



Token Tree Merge

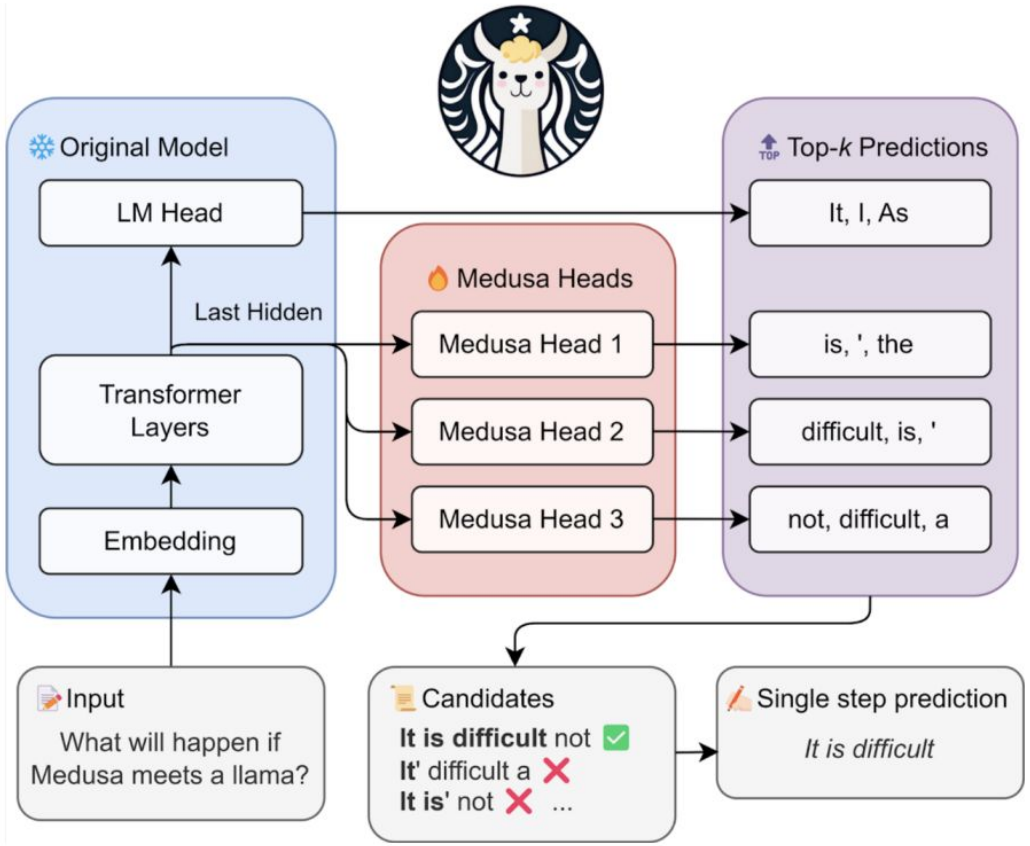


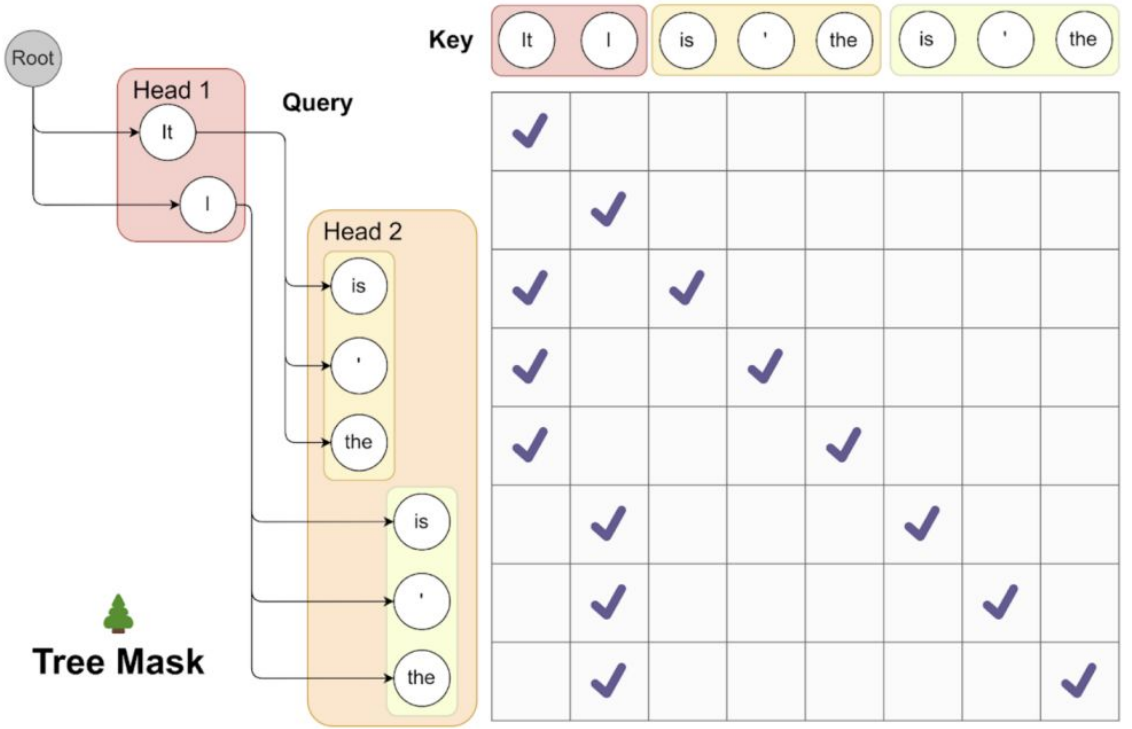
SpecInfer Accelerates LLM Inference by 1.3-2.4x



Medusa: Speculative Decoding with Multiple Decoding Heads

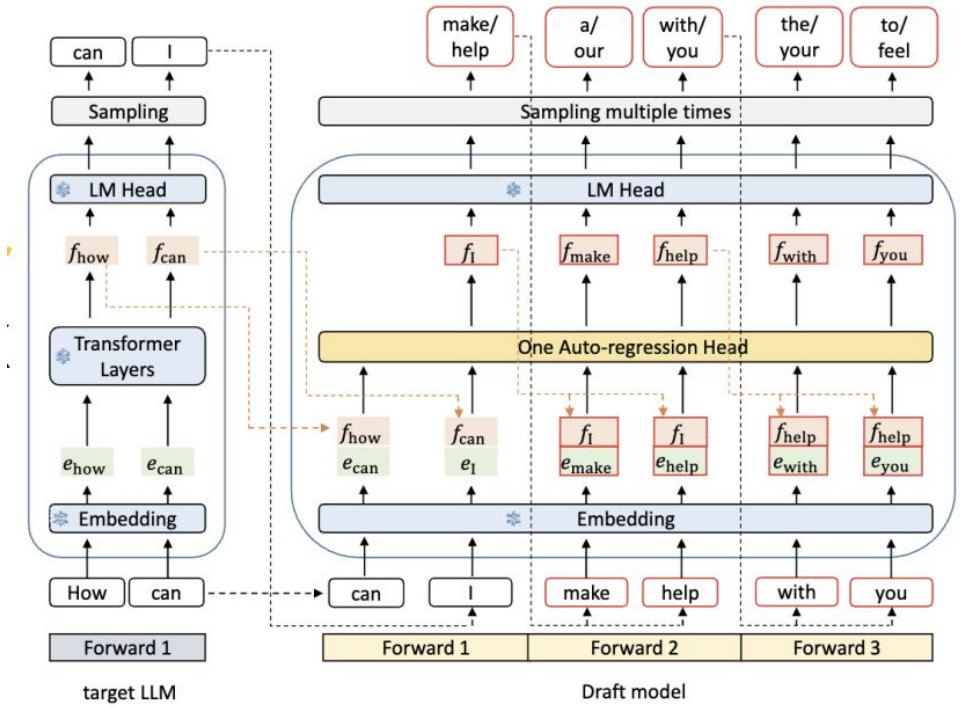
- Introduce trainable attention heads to predict future tokens

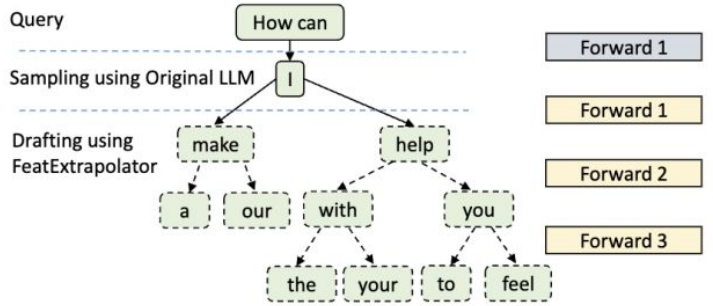




Eagle: Dynamic Speculative Token Trees

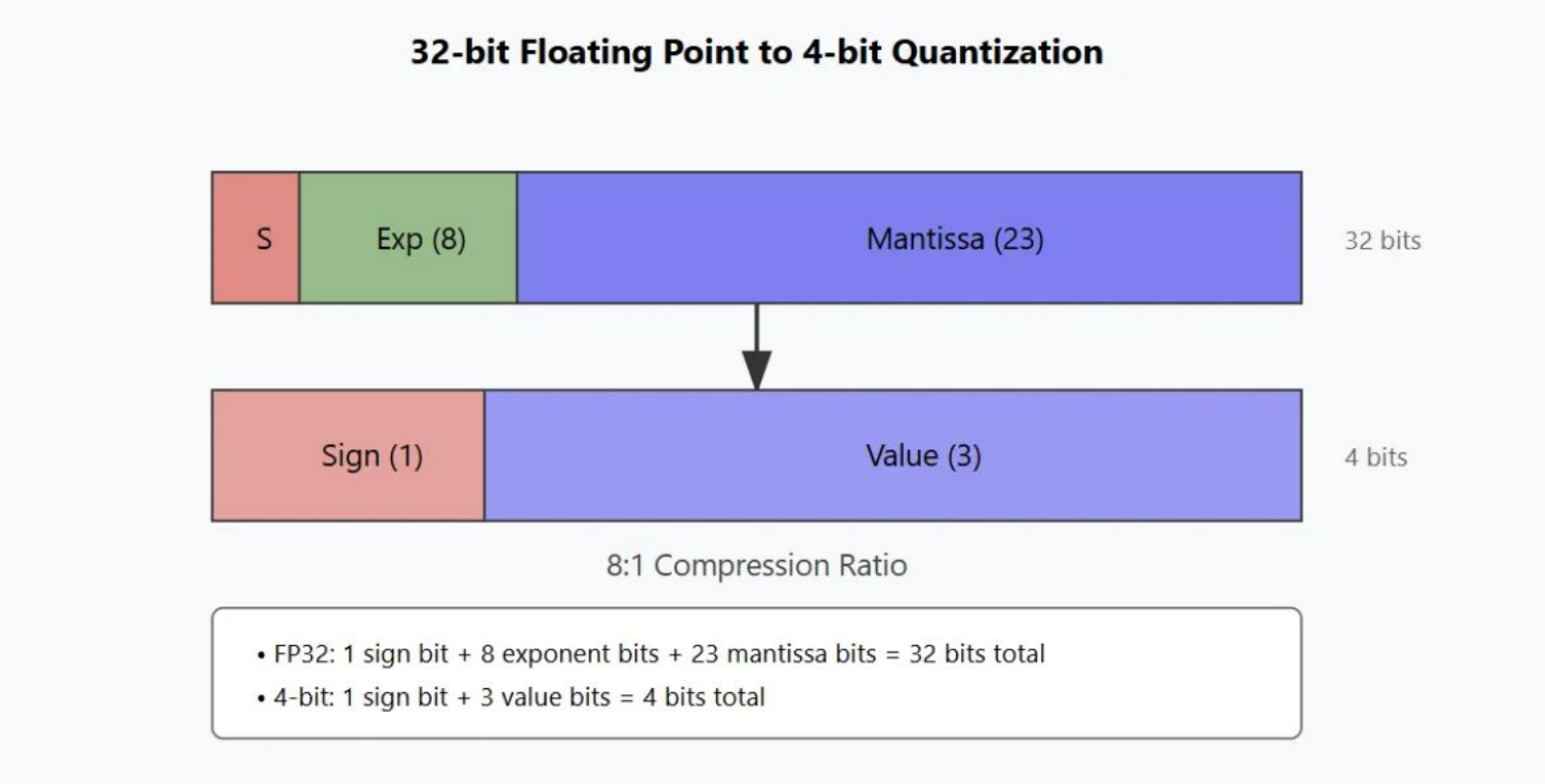
- Draft model reuses embedding and LM head layers of the LLM, and introduces a trainable attention layer
- Expand phase: draft model selects K nodes to decode in each iteration
- Rerank phase: select N nodes for verification based on accumulated likelihood





Other Ways?

- Quantization: quantization transforms 32-bit floating-point parameters into more compact representations like 8-bit or 4-bit integers, enabling efficient deployment in resource-constrained environments.



Other Ways?

- Quantization: model weights, gradients, kv cache

Format	Bit Layout	Range	Memory	Primary Use Case	Key Advantage
FP32	1 sign + 8 exp + 23 mantissa	$\pm 3.4 \times 10^{38}$	4 bytes	Training	Highest precision
FP16	1 sign + 5 exp + 10 mantissa	±65,504	2 bytes	Mixed training	Good precision-memory balance
BF16	1 sign + 8 exp + 7 mantissa	$\pm 3.4 \times 10^{38}$	2 bytes	Training & Inference	Large range with reduced precision
INT8	8 bits total	-128 to +127	1 byte	Inference	Fast computation
INT4	4 bits total	-8 to +7	0.5 bytes	Edge deployment	Minimal memory usage

Inference Framework

Easy, fast, and cheap LLM serving for everyone

vLLM is a fast and easy-to-use library for LLM inference and serving.

Originally developed in the Sky Computing Lab at UC Berkeley, vLLM has evolved into a community-dr

Where to get started with vLLM depends on the type of user. If you are looking to:

- Run open-source models on vLLM, we recommend starting with the Quickstart Guide
- Build applications with vLLM, we recommend starting with the User Guide
- Build vLLM, we recommend starting with Developer Guide

For information about the development of vLLM, see:

Inference Framework

SGLang is a high-performance serving framework for large language models and multimodal models. It is designed to deliver low-latency and high-throughput inference across a wide range of setups, from a single GPU to large distributed clusters. Its core features include:

- **Fast Runtime**: Provides efficient serving with RadixAttention for prefix caching, a zero-overhead CPU scheduler, prefill-decode disaggregation, speculative decoding, continuous batching, paged attention, tensor/pipeline/expert/data parallelism, structured outputs, chunked prefill, quantization (FP4/FP8/INT4/AWQ/GPTQ), and multi-LoRA batching.
- **Broad Model Support**: Supports a wide range of language models (Llama, Qwen, DeepSeek, Kimi, GLM, GPT, Gemma, Mistral, etc.), embedding models (e5-mistral, gte, mcdse), reward models (Skywork), and diffusion models (WAN, Qwen-Image), with easy extensibility for adding new models. Compatible with most Hugging Face models and OpenAI APIs.
- **Extensive Hardware Support**: Runs on NVIDIA GPUs (GB200/B300/H100/A100/Spark), AMD GPUs (MI355/MI300), Intel Xeon CPUs, Google TPUs, Ascend NPUs, and more.
- **Active Community**: SGLang is open-source and supported by a vibrant community with widespread industry adoption, powering over 400,000 GPUs worldwide.
- **RL & Post-Training Backbone**: SGLang is a proven rollout backend across the world, with native RL integrations and adoption by well-known post-training frameworks such as AReaL, Miles, slime, Tunix, verl and more.

Train a GPT model (1)

- Datasets: CC, Books, Wiki etc.
- Training framework: MegaTron-LM

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

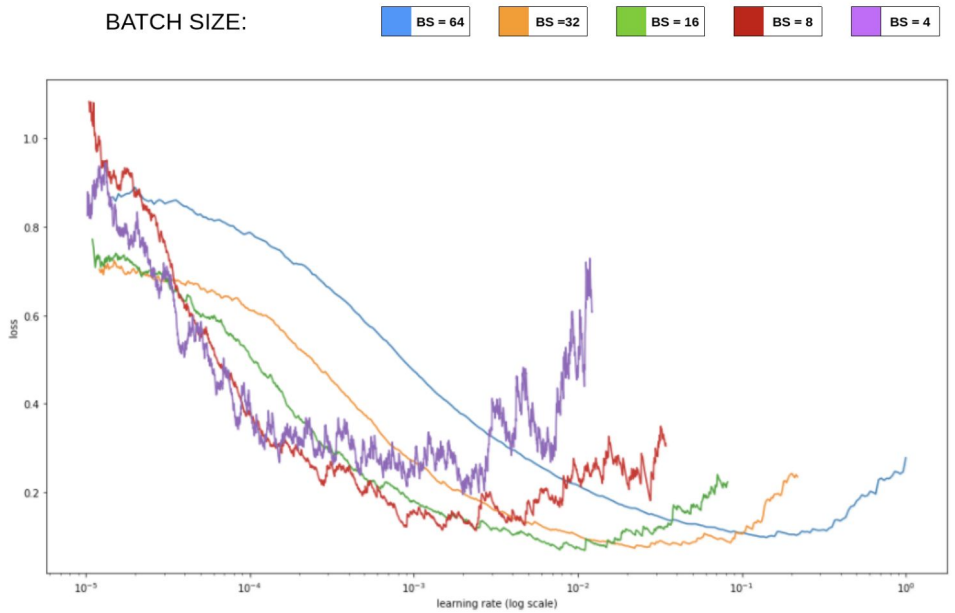
Train a GPT model (2)

- The smallest GPT-3 model (125M) has 12 attention layers, each with 12x 64-dimension heads. The largest GPT-3 model (175B) uses 96 attention layers, each with 96x 128-dimension heads.
- The 175 Billion parameters needs 175×4=700GB memory to store in FP32 (each parameter needs 4 Bytes). This is one order of magnitude larger than the maximum memory in a single GPU (48 GB of Quadro RTX 8000).

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or “GPT-3”	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

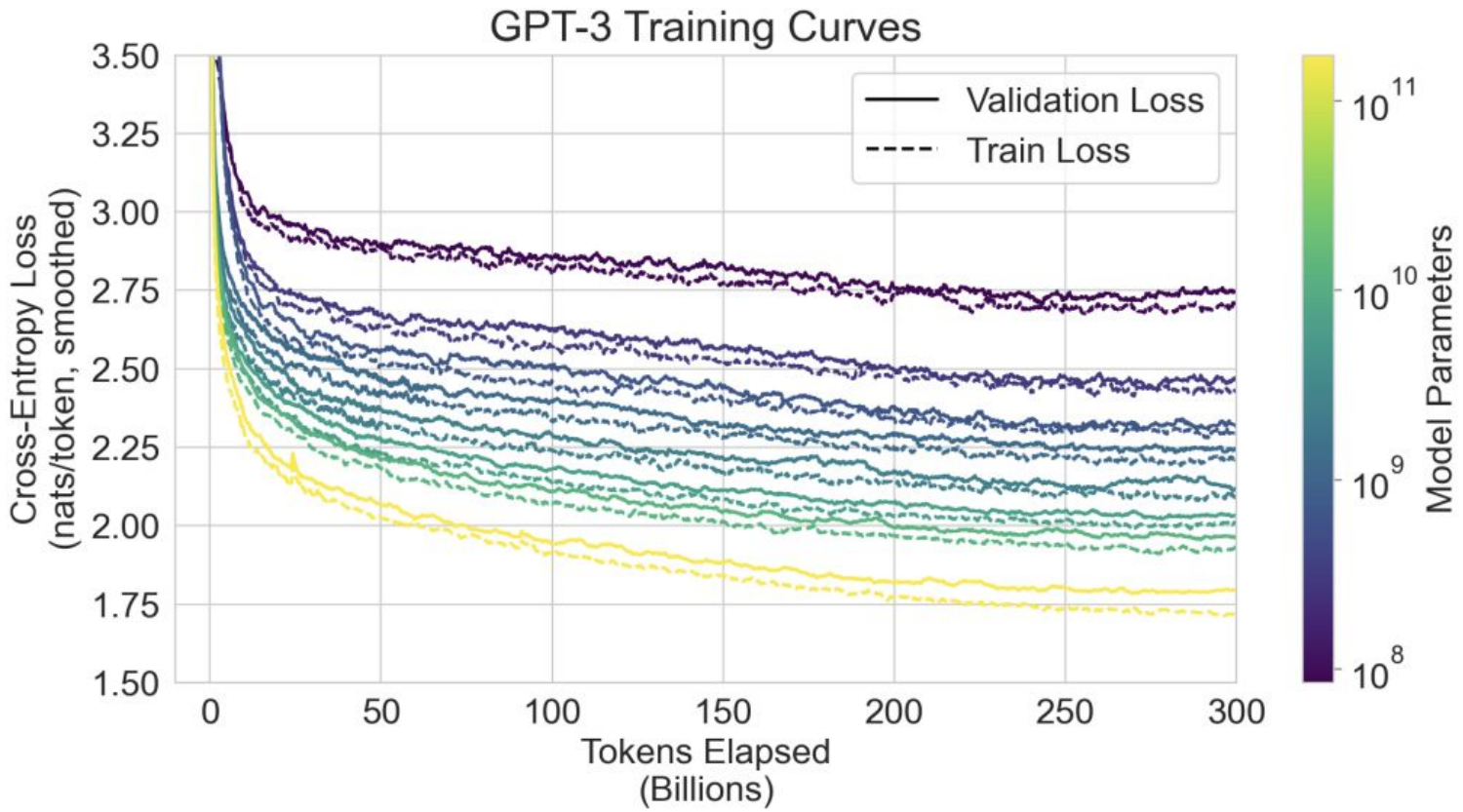
Train a GPT model (3)

- To train models of different sizes, the batch size is increased according to number of parameters, while the learning rate is decreased accordingly.
- GPT-3 125M use batch size 0.5M and learning rate of 6.0 ×10−4 , where GPT-3 175B uses batch size 3.2M and learning rate of 0.6×10−4 .



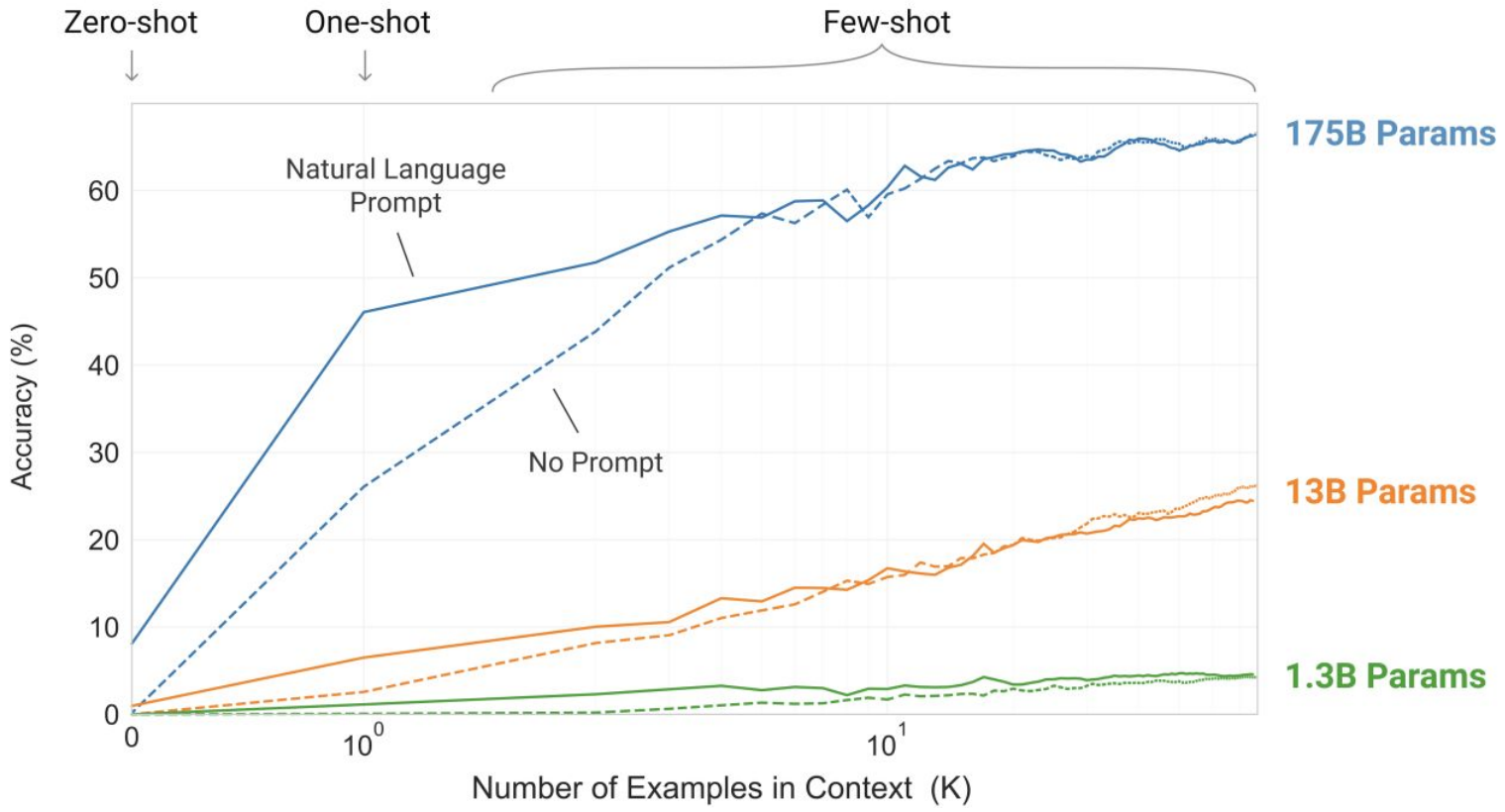
Train a GPT model (4)

- Training curve (tokens vs. loss)



Train a GPT model (5)

- Model size vs. model performance



Train a GPT model (6)

- Model evaluation

Setting	Winograd	Winogrande (XL)
Fine-tuned SOTA	**90.1**[a]	**84.6**[b]
GPT-3 Zero-Shot	88.3*	70.2
GPT-3 One-Shot	89.7*	73.2
GPT-3 Few-Shot	88.6*	77.7

Setting	PIQA	ARC (Easy)	ARC (Challenge)	OpenBookQA
Fine-tuned SOTA	79.4	**92.0**[KKS+20]	**78.5**[KKS+20]	**87.2**[KKS+20]
GPT-3 Zero-Shot	**80.5***	68.8	51.4	57.6
GPT-3 One-Shot	**80.5***	71.2	53.2	58.8
GPT-3 Few-Shot	**82.8***	70.1	51.5	65.4

Train a GPT model (7)

- Model evaluation
  - Accuracy
  - Exact Match, F1
  - BLEU, ROUGE and METEOR
  - Perplexity

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_{1:i-1})\right)$$

Where:

- N is the number of words in the sequence.
- w_i is the current word, and $w_{1:i-1}$ is the sequence of previous words.
- $P(w_i|w_{1:i-1})$ is the conditional probability of word w_i given the previous words in the sequence.

Thanks

Q&A

Prof. Yu Cheng
chengyu@cse.cuhk.edu.hk