FlashAttention-2 & SpecInfer

Ammar Ahmed, Alex de la Iglesia, Aditya Singhvi

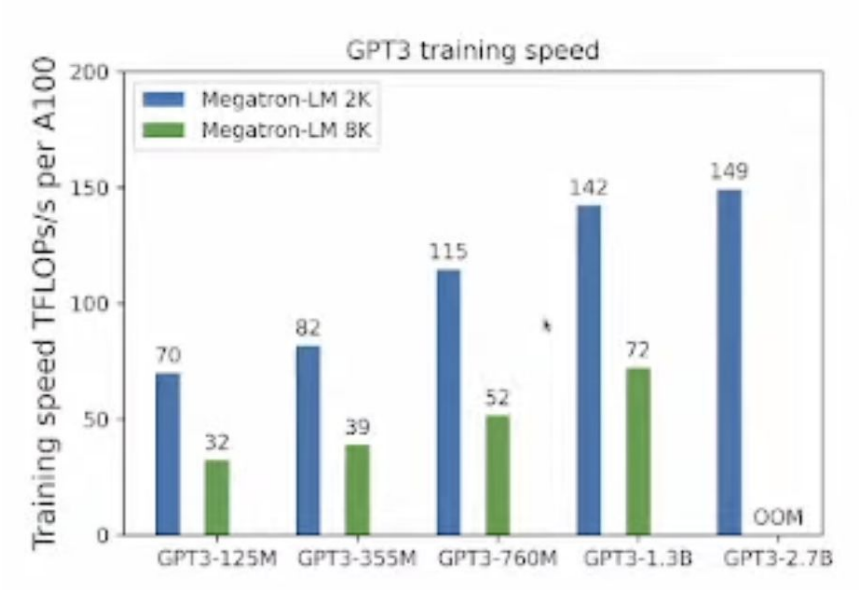
FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning

Motivation: Modeling Longer Sequences

1. Creating **new capabilities** for text (e.g. summarizing an entire book)
2. Closing the **reality gap** through higher-resolution sequences (e.g. for vision transformers)
3. Opening **new areas of development** by enabling time series data that requires long context (audio, video, etc.)

Blog Post: Ré et al., Can Longer Sequences Help Take the Next Leap in AI?

Problem: Modeling Longer Sequences



Takeaway: Not only is training slower with longer context, it is also less efficient!

Chart from Video: "FlashAttention - Tri Dao | Stanford MLSys #67", adapted from Blog: Tri Dao, FlashAttention: Fast Transformer Training with Long Sequences

Background: Existing Works vs. FlashAttention

Existing Works

- Tradeoff correctness for speed through approximation
- Focus on lowering compute, but may not lead to wall-clock speedup (as bottleneck is memory IO)

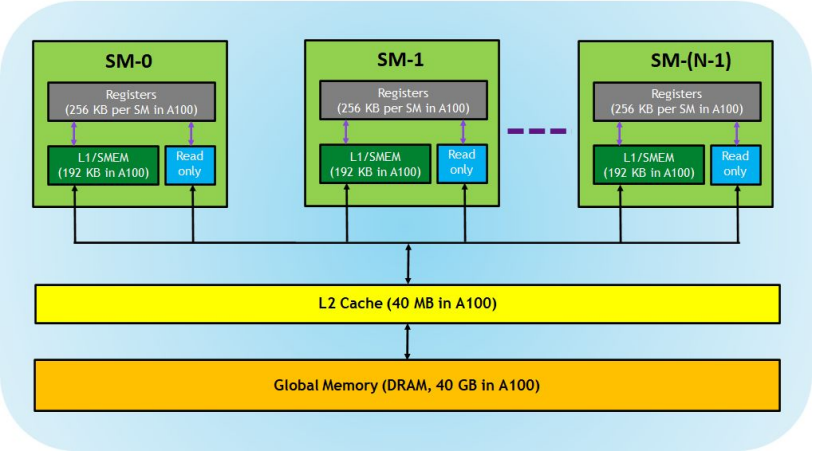
FlashAttention

- Compute exact attention with no loss in accuracy
- Create IO-aware algorithm to reduce memory bottleneck and create real wall-clock speedups

Dao et al., FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

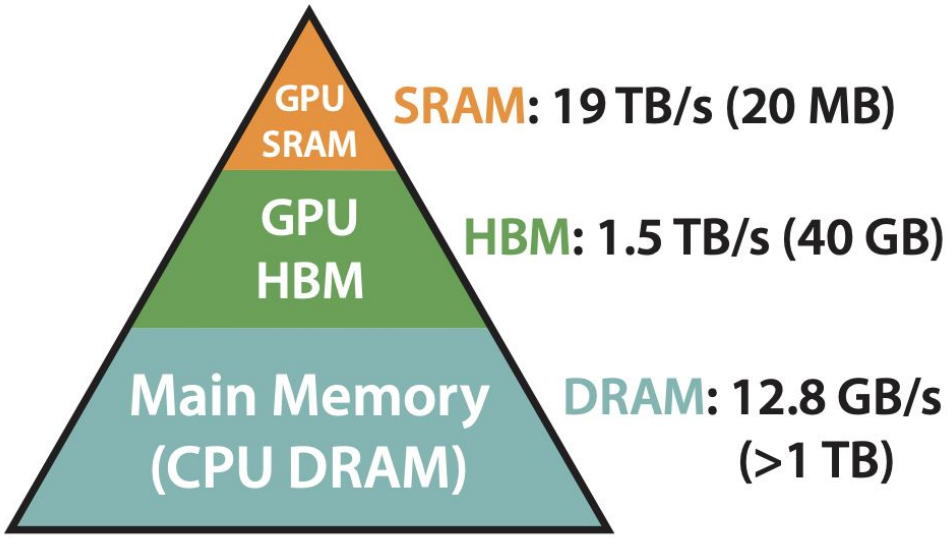
Background: GPU Architecture

- Threads are grouped into **warps** that execute same instructions in lockstep with **fast internal communication**
- A **block** consists of multiple warps on same streaming multiprocessor that **communicate using shared memory**



Blog: ArcCompute, GPU 101 Memory Hierarchy

Background: GPU Memory Hierarchy



Memory Hierarchy with Bandwidth & Memory Size

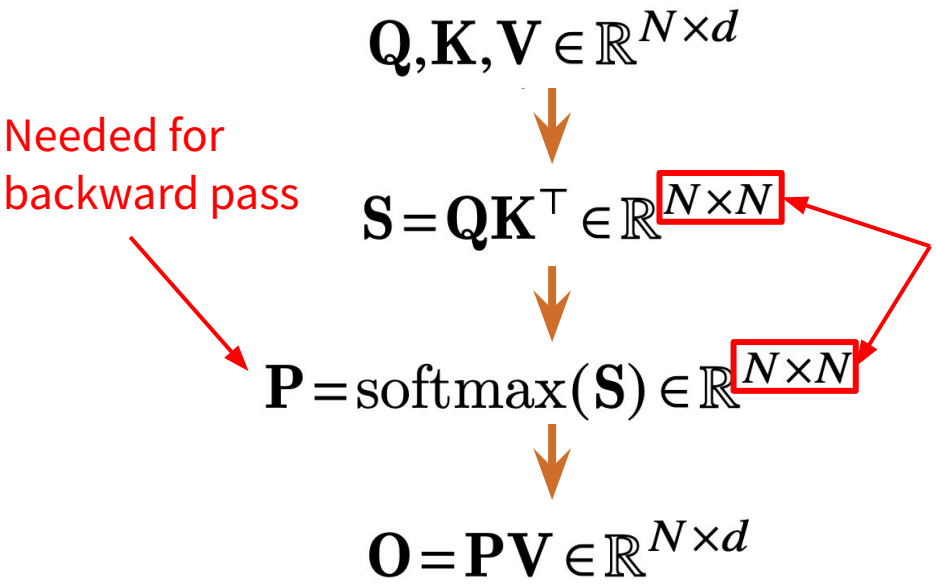
1. Inputs loaded from HBM to SRAM.
2. Computation is performed.
3. Outputs sent back to HBM.

Implications: GPU Performance

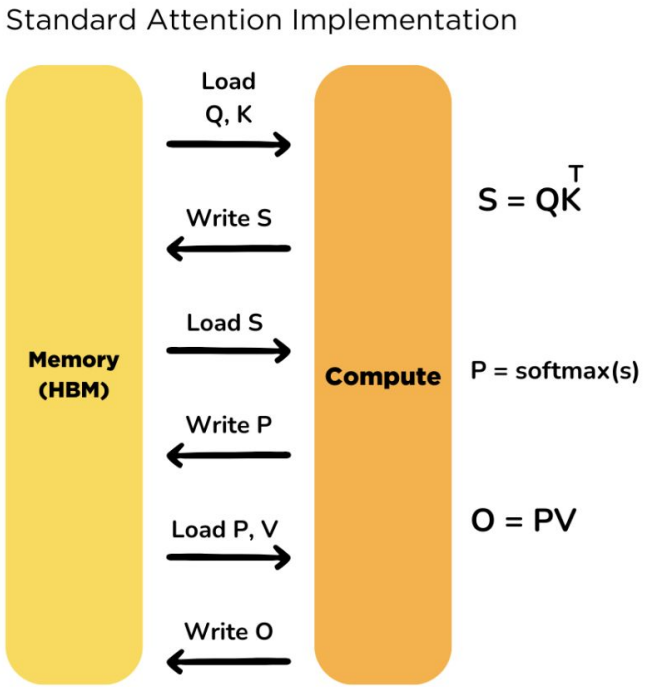
- GPUs are very fast at matrix multiplication (GEMMs), and significantly (15x) slower at other operations.
  - Implication: **GEMMs upper-bound efficient computation**.
  - Relatively more FLOPs spent doing GEMM → more efficient computation.
- We cannot change the amount of computation for exact attention — we can only change how fast this computation is done by:
  - Reducing the surrounding bottlenecks (i.e. memory I/O)
  - Making the computation more efficient by using more GEMM

Background: Bottlenecks in Traditional Attention

N: Sequence length (up to 8k - 128k)
d: Query dimension (~512)



Background: Bottlenecks in Traditional Attention



Takeaway: Back-and-forth between memory and compute, with large matrices (S, P) needing to be stored and loaded!

Even more issues when considering other operations such as masking (applied to S), dropout (applied to P), etc.

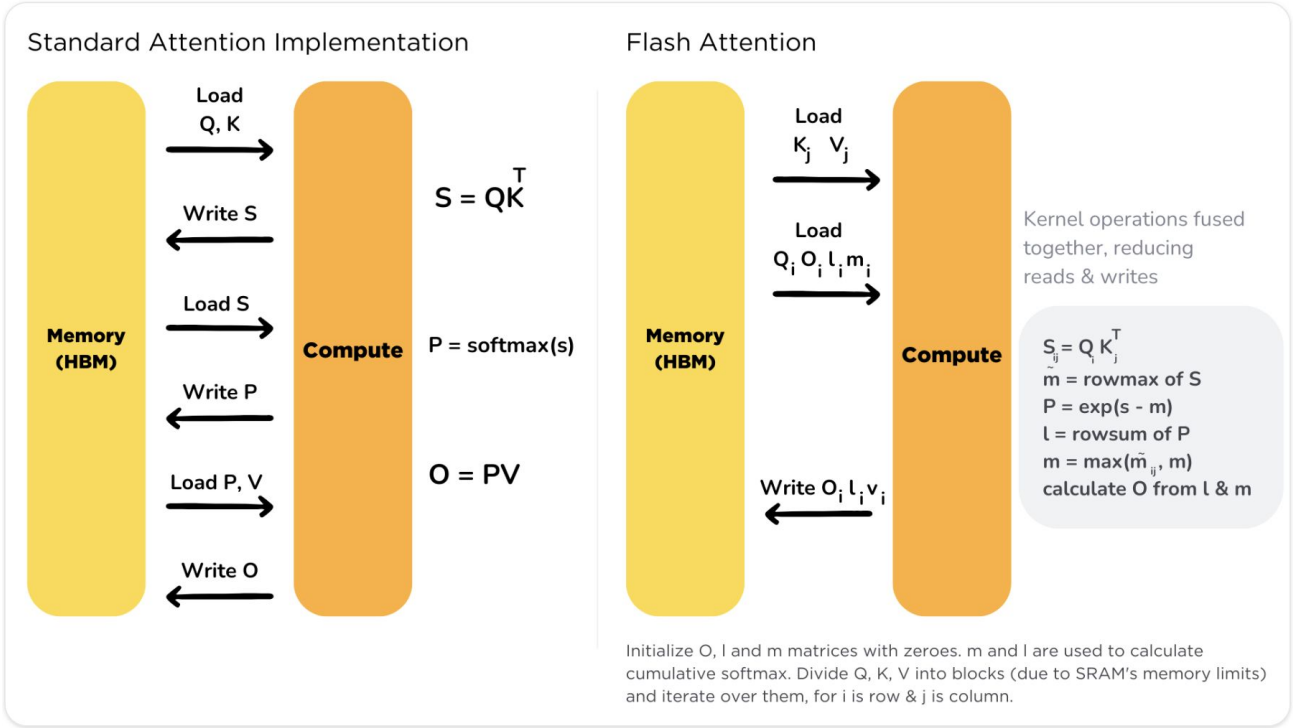
[Blog: Hugging Face, Flash Attention]

Improvement: Computing by Blocks

- **Goal:** Avoid storing and loading N x N matrices to/from HBM
  - These matrices are too large to store in SRAM at once, so we must compute them in blocks.
- **Challenge 1:** Softmax needs to look at an entire row at a time
- **Challenge 2:** We need these N x N matrices to compute gradients in backward pass

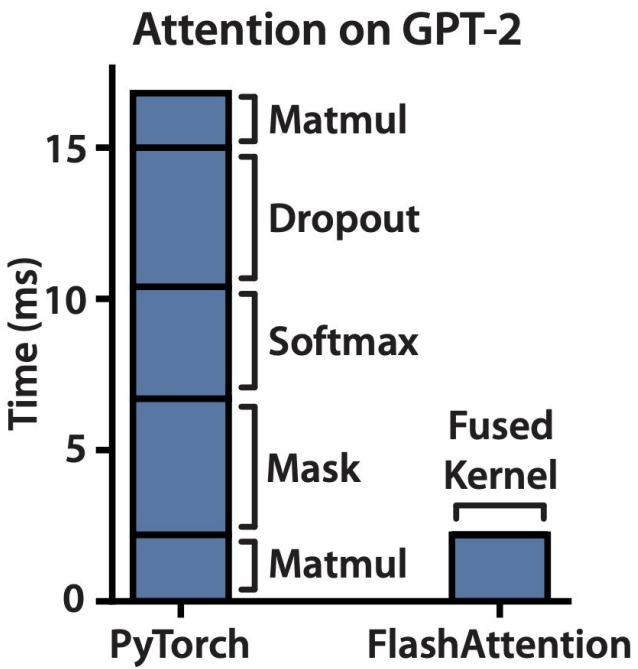
Improvement: Online Softmax

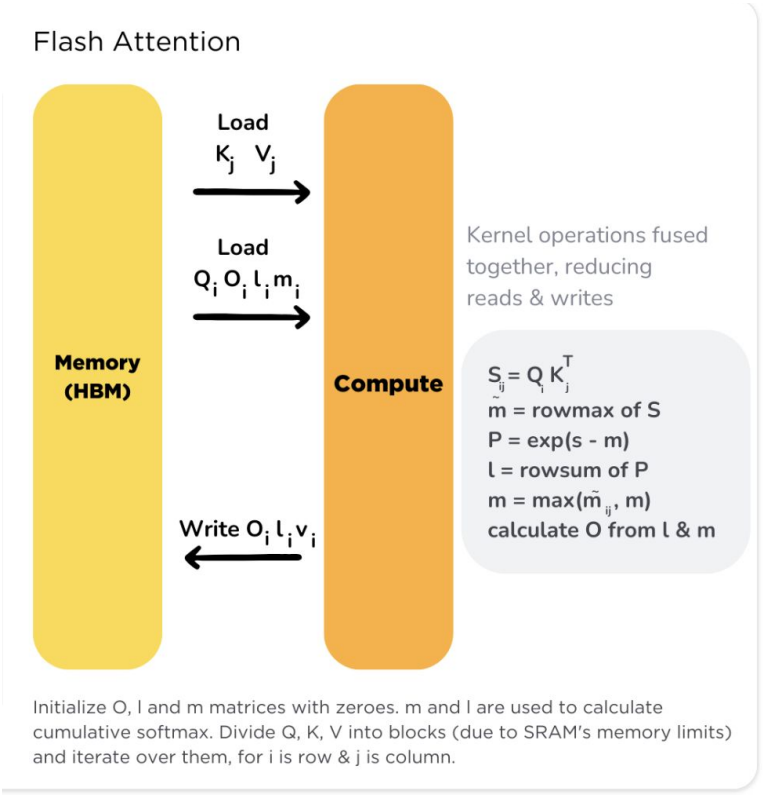
Takeaway: Adapt softmax calculation to work in blocks!



[Blog: Hugging Face, Flash Attention]

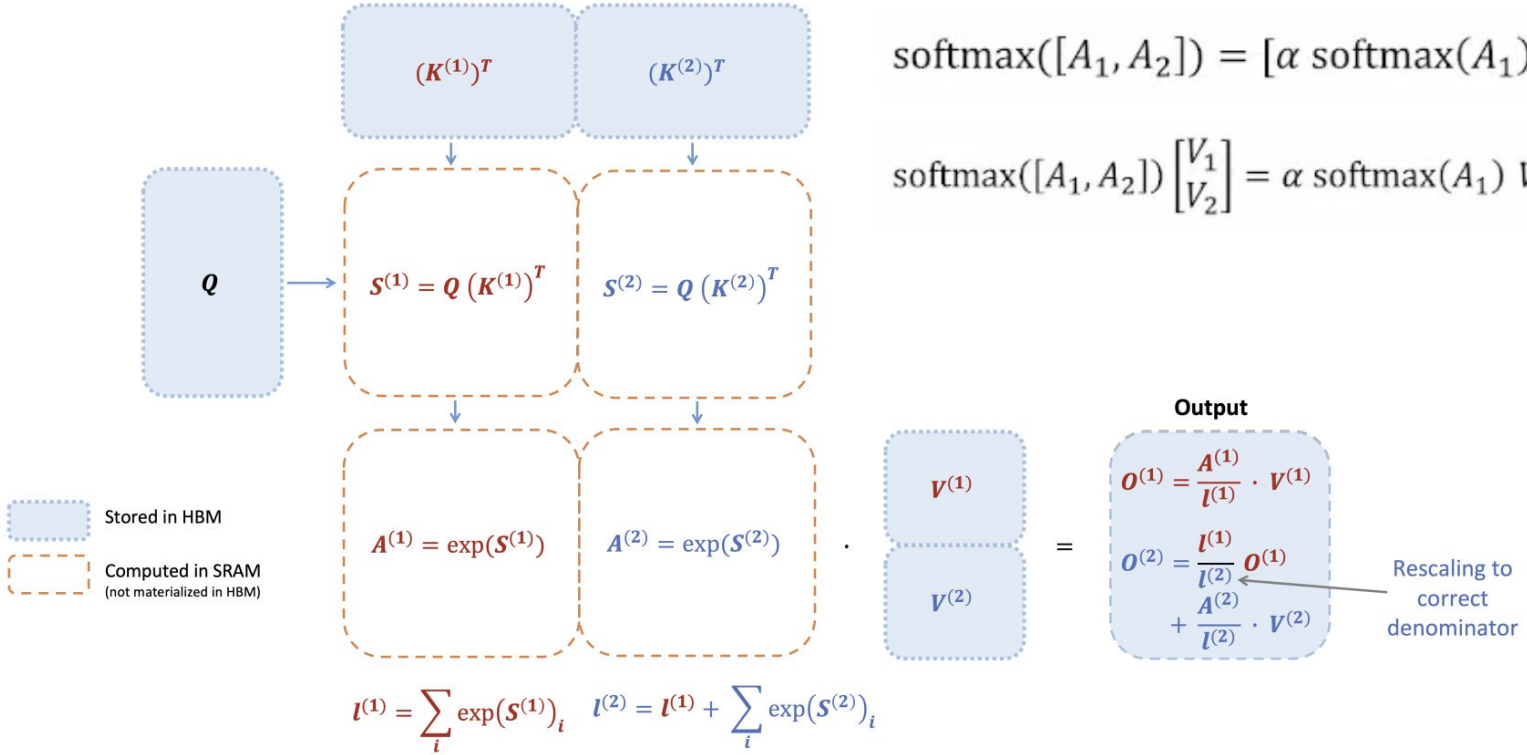
Improvement: Online Softmax





[Blog: Hugging Face, Flash Attention]

Improvement: Tiling for Online Softmax



$$\text{softmax}([A_1, A_2]) = [\alpha\ \text{softmax}(A_1), \beta\ \text{softmax}(A_2)]$$

$$\text{softmax}([A_1, A_2]) \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \alpha\ \text{softmax}(A_1)\ V_1 + \beta\ \text{softmax}(A_2) V_2$$

Improvement: Reduce Non-GEMM FLOPs in Online Softmax

- Less frequent scaling operations in FA-2 reduce non-GEMM FLOPs
- Instead of scaling the output for each block, we keep some intermediate scaling factors around and **scale at the end of the loop**.

$$\tilde{m}^{(1)} = \text{rowmax}(\mathbf{S}^{(1)}) \in \mathbb{R}^{B_r} \qquad \ell^{(1)} = \text{rowsum}(e^{\mathbf{S}^{(1)} - m^{(1)}}) \in \mathbb{R}^{B_r}$$

$$\tilde{\mathbf{O}}^{(1)} = e^{\mathbf{S}^{(1)} - m^{(1)}} \mathbf{V}^{(1)} \in \mathbb{R}^{B_r \times d} \qquad m^{(2)} = \max(m^{(1)}, \text{rowmax}(\mathbf{S}^{(2)})) = m$$

$$\ell^{(2)} = e^{m^{(1)} - m^{(2)}} \ell^{(1)} + \text{rowsum}(e^{\mathbf{S}^{(2)} - m^{(2)}}) = \text{rowsum}(e^{\mathbf{S}^{(1)} - m}) + \text{rowsum}(e^{\mathbf{S}^{(2)} - m}) = \ell$$

$$\tilde{\mathbf{P}}^{(2)} = \text{diag}(\ell^{(2)})^{-1} e^{\mathbf{S}^{(2)} - m^{(2)}}$$

$$\tilde{\mathbf{O}}^{(2)} = \text{diag}(e^{m^{(1)} - m^{(2)}}) \tilde{\mathbf{O}}^{(1)} + e^{\mathbf{S}^{(2)} - m^{(2)}} \mathbf{V}^{(2)} = e^{s^{(1)} - m} \mathbf{V}^{(1)} + e^{s^{(2)} - m} \mathbf{V}^{(2)}$$

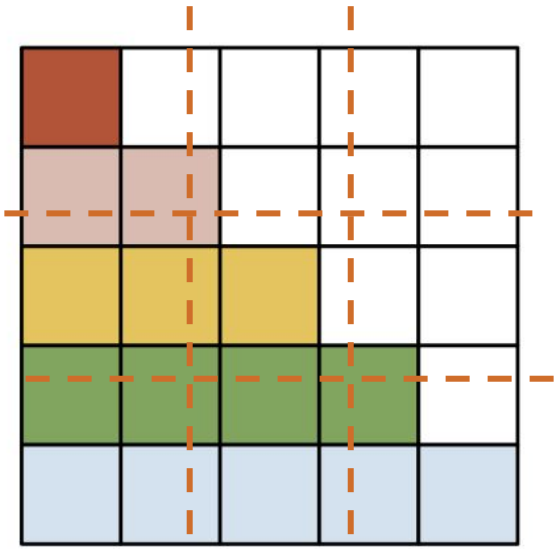
$$\mathbf{O}^{(2)} = \text{diag}(\ell^{(2)})^{-1} \tilde{\mathbf{O}}^{(2)} = \mathbf{O}.$$

Improvement: Storing Data for Backward Pass

- We need to store ℓ and m in order to recompute attention quickly in the backward pass.
- Instead of storing both separately, we store $L^{(j)} = m^{(j)} + \log(\ell^{(j)})$.

FA-2: Take Advantage of Decoder Masking

- Skip masked decode blocks, as nearly ½ of all blocks can be skipped!
- Only apply mask to blocks on diagonal – anything that is fully computed doesn't require the masking step.

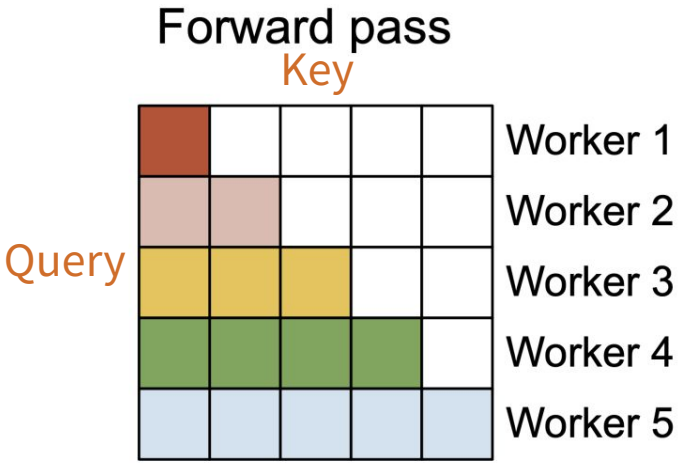


FA-1 Improvement: Recomputation

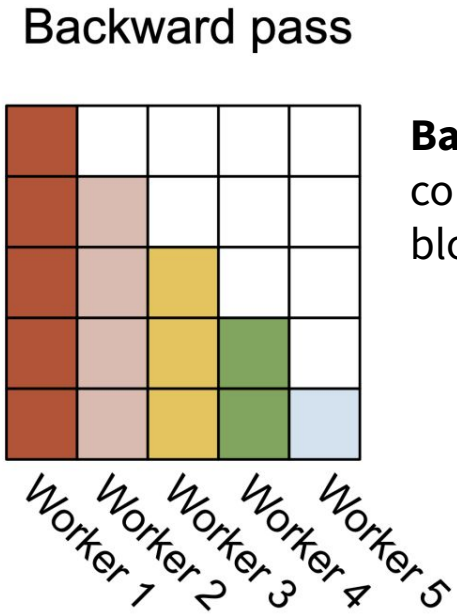
- Recompute **S**, **P** matrices during backward pass using stored intermediate values from forward pass to reduce HBM I/O
- Requires **O(N)** additional memory instead of **O(N^2)**
- **10-20x** memory saving
- **2-4x** wall-clock speedup as memory bandwidth bottleneck is mitigated
- *No further improvements made in FA-2 to recomputation*

FA-2 Improvement: Better Parallelism

- Parallelize over sequence length to support longer context windows (than FA-1).
- **Note:** Multiple thread blocks per attention head!



Forward Pass: no communication required between workers (thread blocks).



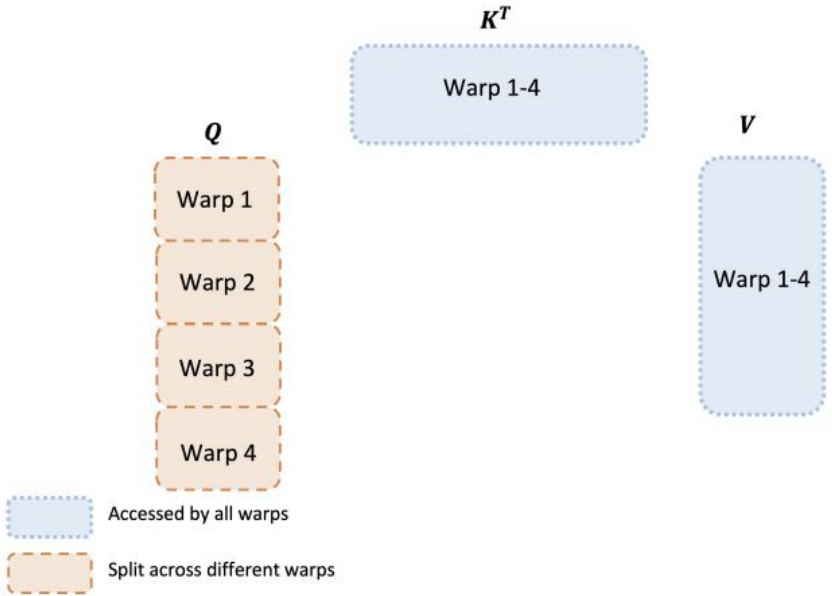
Backward Pass: Only communication across blocks is to update **dQ**.

Improvement: Work Partitioning Between Warps

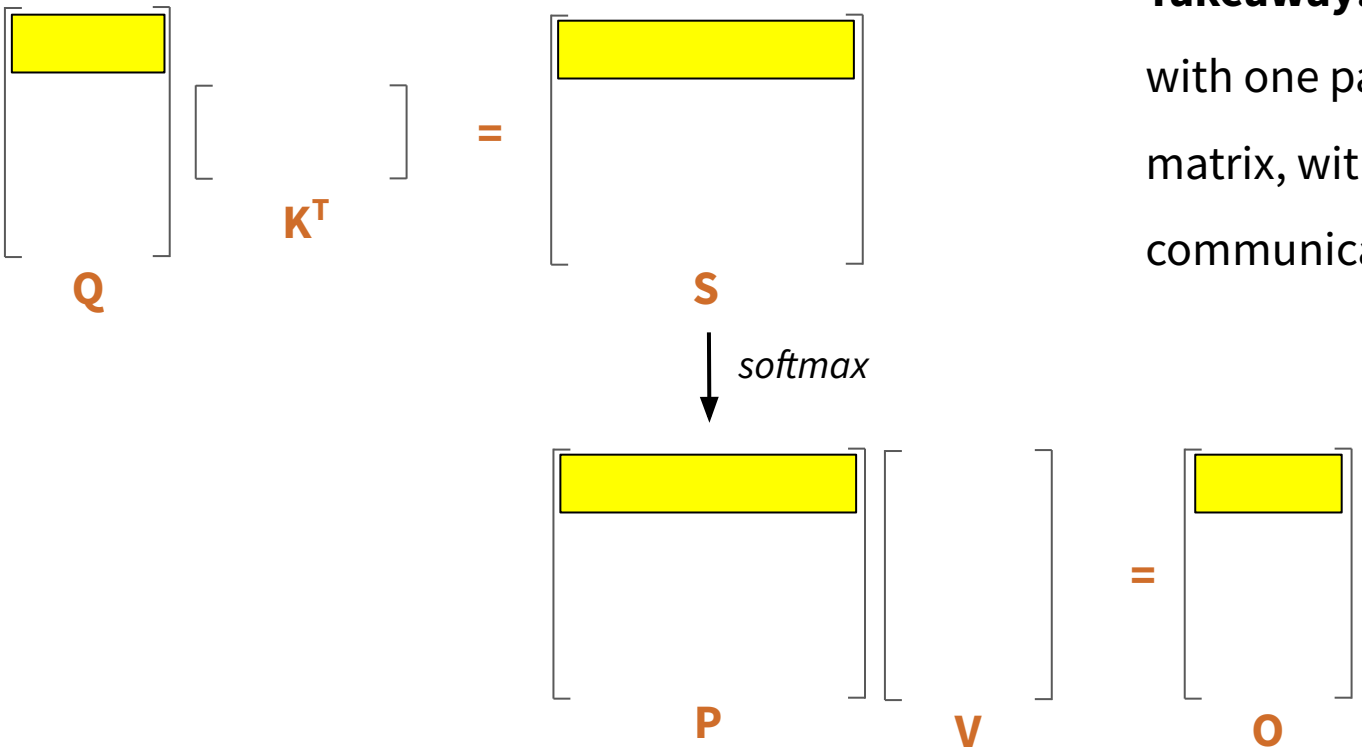
- We’ve figured out how to divide work between blocks; **how do we divide work across warps?**
- Typical: 4-8 warps per block
- Want to reduce communication across warps, as this requires shared memory

Improvement: Work Partitioning Between Warps

Solution: Divide the **Q** matrix across warps, while keeping the **K** and **V** matrices accessible by all warps.



Improvement: Work Partitioning Between Warps



Takeaway: Each warp is left with one partition of the output matrix, with no inter-warp communication required!

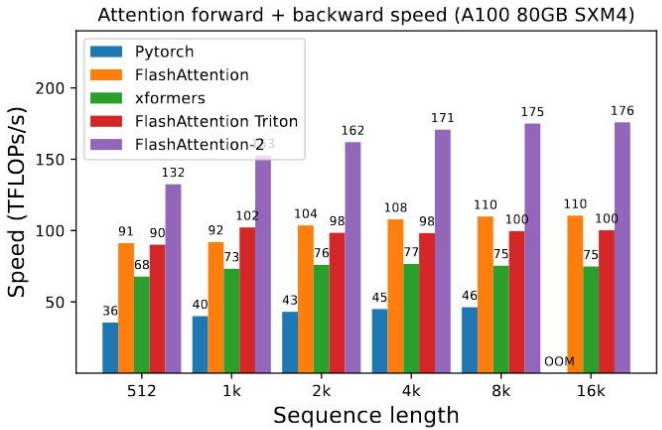
Attention Benchmarks for Training

1.7 - 3.0x faster than FlashAttention1

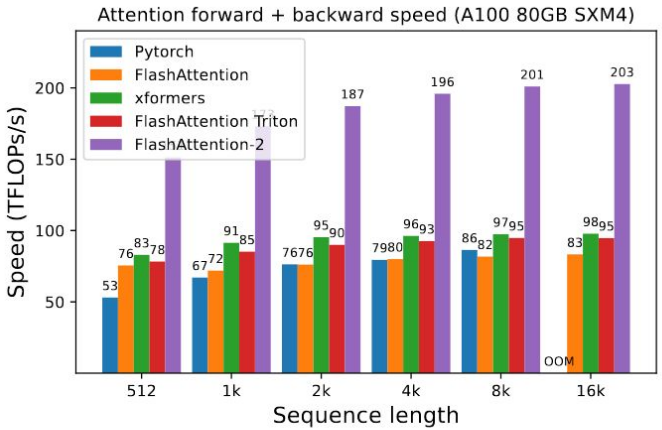
3 - 10x faster than standard attention implementations

FlashAttention2 runtime reaches 230 TFLOPS, 73% of theoretical maximum

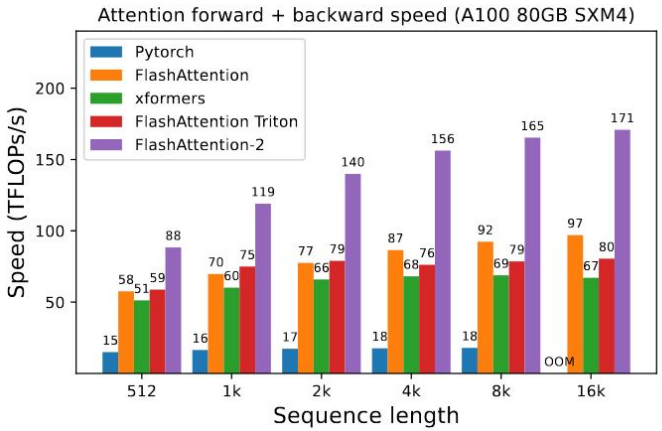
FlashAttention1 runtime reaches only 25-40% of theoretical maximum



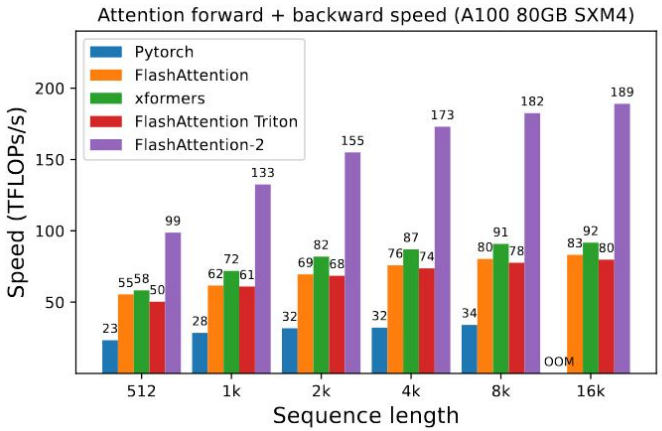
(a) Without causal mask, head dimension 64



(b) Without causal mask, head dimension 128



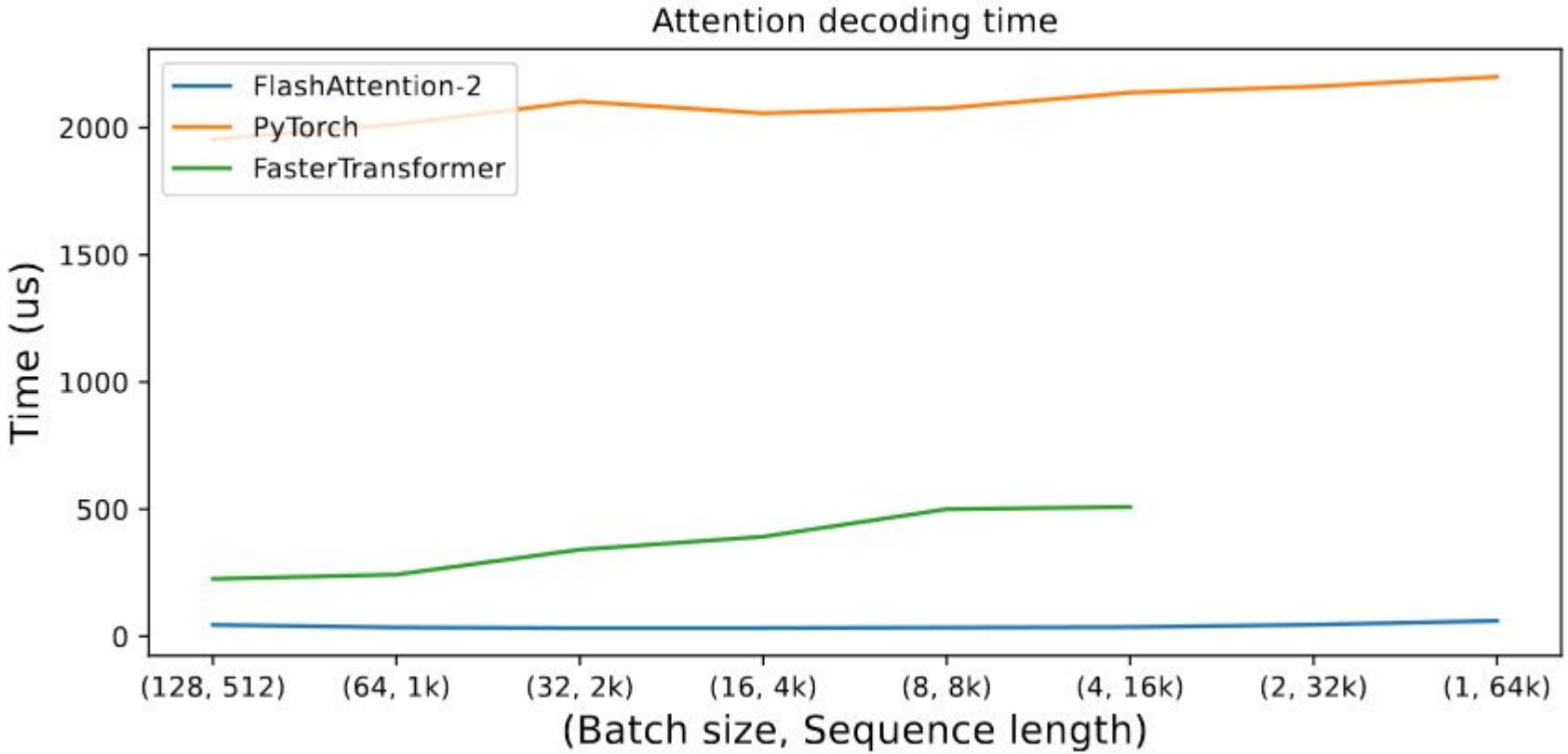
(c) With causal mask, head dimension 64



(d) With causal mask, head dimension 128

Figure 4: Attention forward + backward speed on A100 GPU

Attention Benchmarks for Inference



Future Work

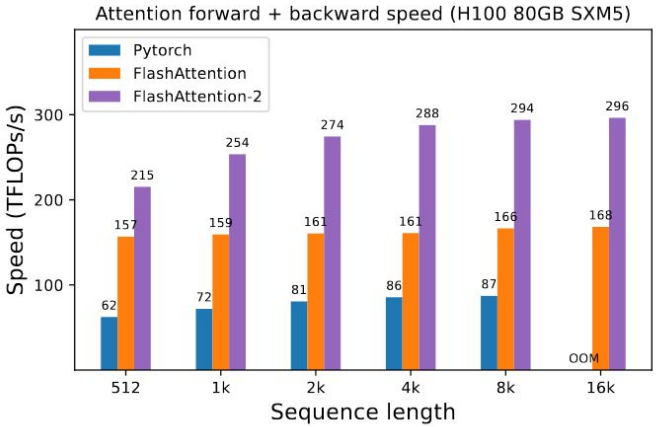
FlashAttention-3:
Fast and Accurate Attention with Asynchrony and Low-precision

Jay Shah*[1], Ganesh Bikshandi*[1], Ying Zhang[2], Vijay Thakkar[3,4], Pradeep Ramani[3], and Tri Dao[5,6]

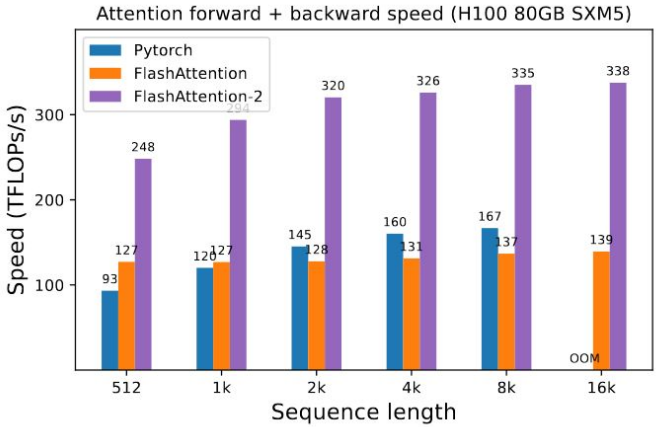
[1]Colfax Research [2]Meta [3]NVIDIA [4]Georgia Tech [5]Princeton University [6]Together AI

{jayhshah,ganesh}@colfax-intl.com, yingz@meta.com, {vithakkar,prraman}@nvidia.com, tri@tridao.me

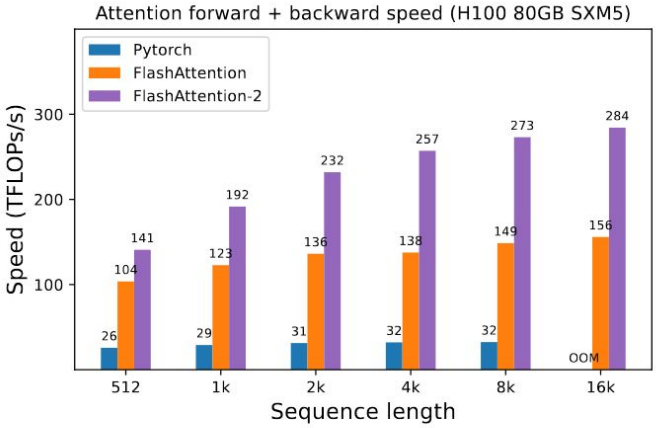
July 16, 2024



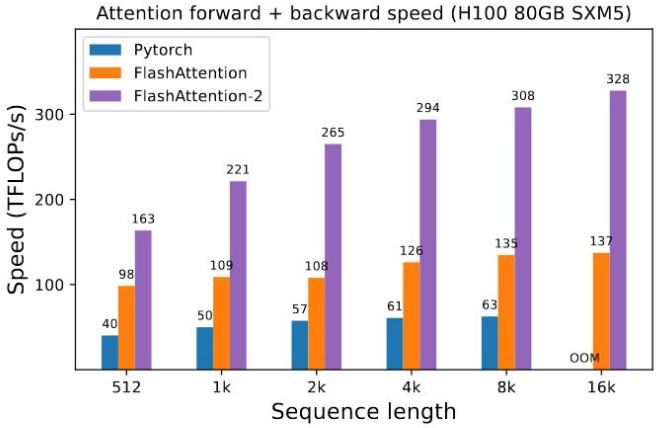
(a) Without causal mask, head dimension 64



(b) Without causal mask, head dimension 128

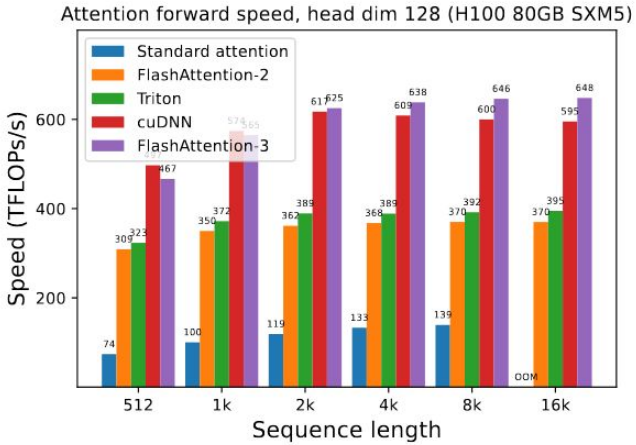


(c) With causal mask, head dimension 64

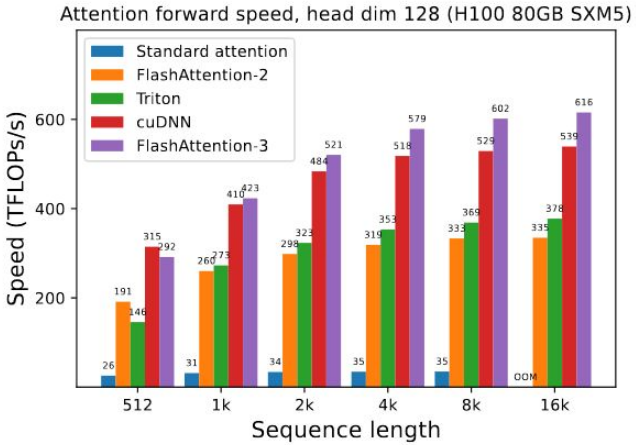


(d) With causal mask, head dimension 128

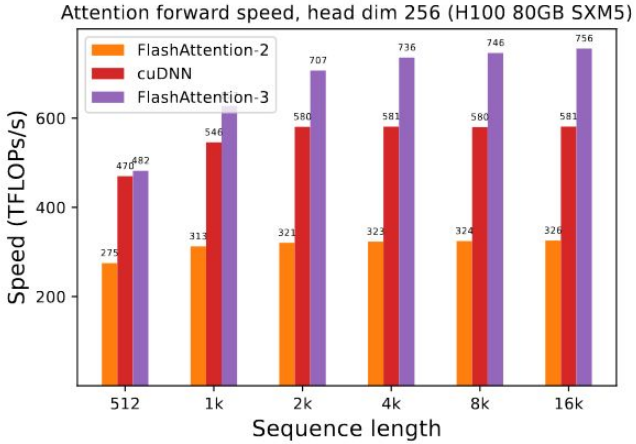
Figure 8: Attention forward + backward speed on H100 GPU



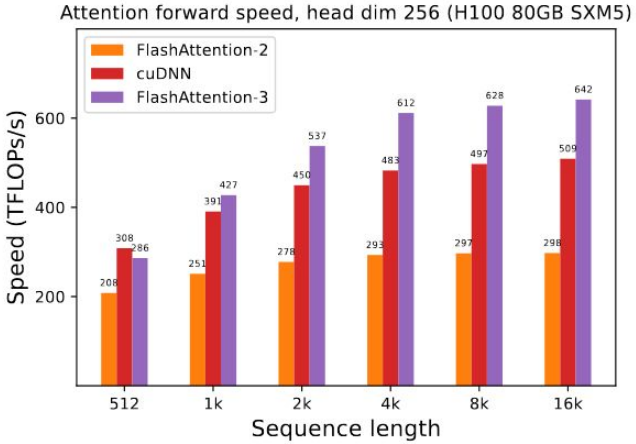
(c) Forward, without causal mask, head dim 128



(d) Forward, with causal mask, head dim 128



(e) Forward, without causal mask, head dim 256



(f) Forward, with causal mask, head dim 256

Figure 5: Attention forward speed (FP16/BF16) on H100 GPU

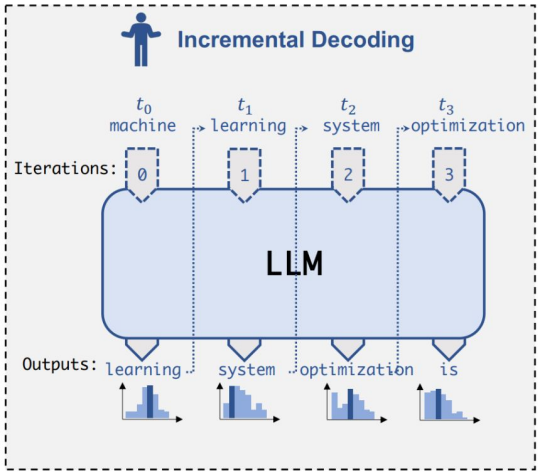
SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification

Traditional LLM Inference

Incremental decoding predicts one token at a time

All LLM parameters must be accessed to generate a single token

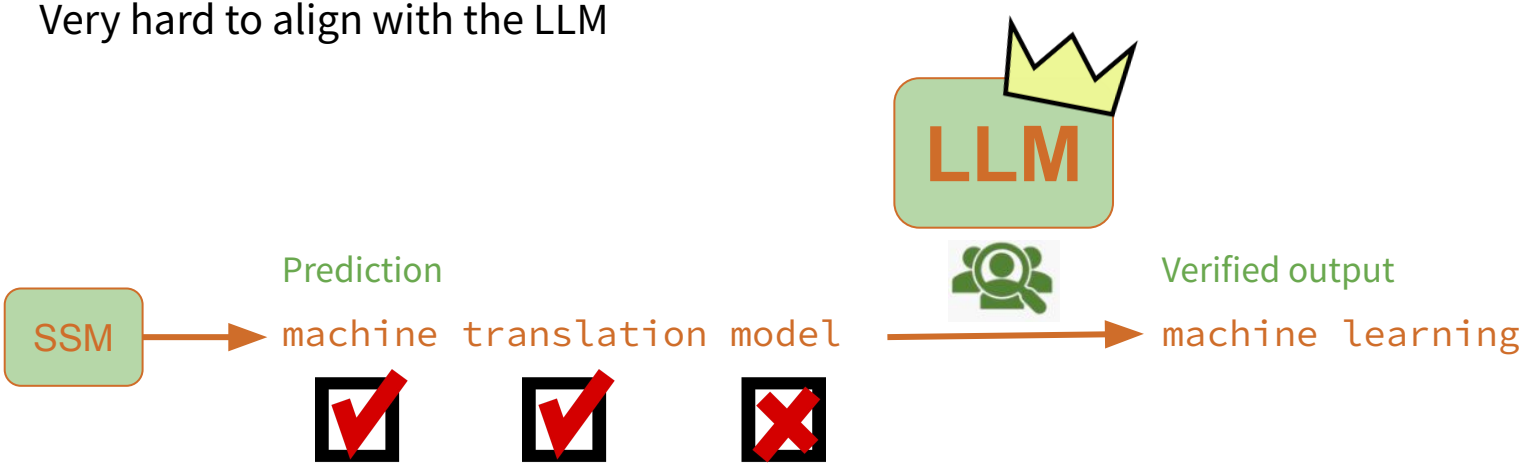
- This memory access is the bottleneck for inference



Speculative Inference

Propose a sequence of tokens to follow the prompt, and verify their correctness against the LLM's output

- Already existed: speculate using a smaller Small Speculative Model
  - Typically the same model and dataset but with 100-1000x fewer parameters
- Replace decoding from the LLM with verifying the sequence
- Very hard to align with the LLM



Improve accuracy with multiple predictions

Observation: Smaller models typically get the correct output within their top-k predictions

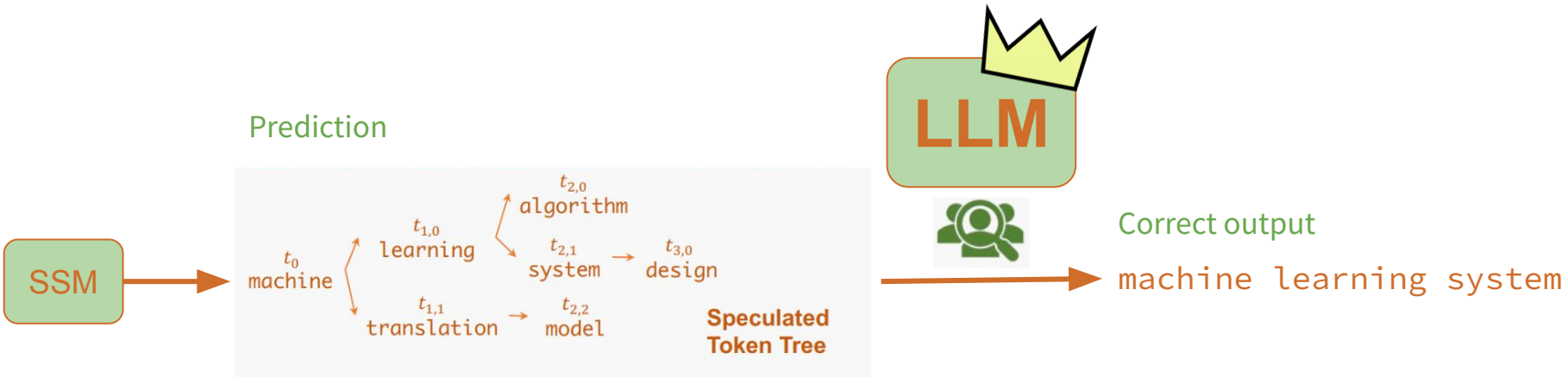
Can we verify against all of these predictions to get a higher accuracy?

	Dataset	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Greedy decoding	Alpaca	68%	77%	81%	84%	85%
	CP	69%	79%	83%	86%	87%
	WebQA	62%	72%	77%	80%	82%
	CIP	70%	81%	85%	88%	89%
	PIQA	63%	75%	79%	83%	85%
Stochastic decoding	Alpaca	54%	81%	91%	95%	97%
	CP	56%	82%	92%	95%	97%
	WebQA	52%	80%	90%	94%	96%
	CIP	57%	84%	92%	95%	97%
	PIQA	55%	82%	91%	95%	97%

Tree-based speculative inference with SpecInfer

Speculate a tree of possible tokens, and verify a valid sequence

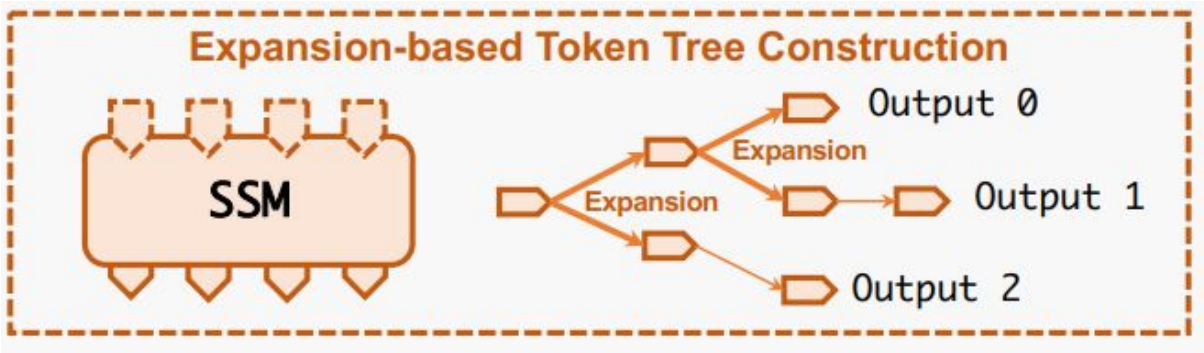
Use any verified tokens from the token tree as output for the system and continue



Constructing the tree of predictions

Expansion-based construction

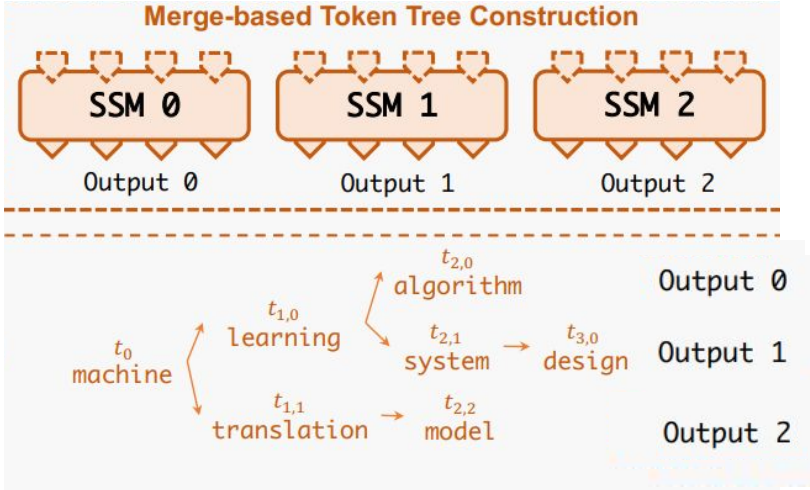
- Predict the top-k tokens at each level
- Exponential increase in speculated sequences
  - Expand only *some* child nodes to get more diverse sequences



Constructing the tree of predictions

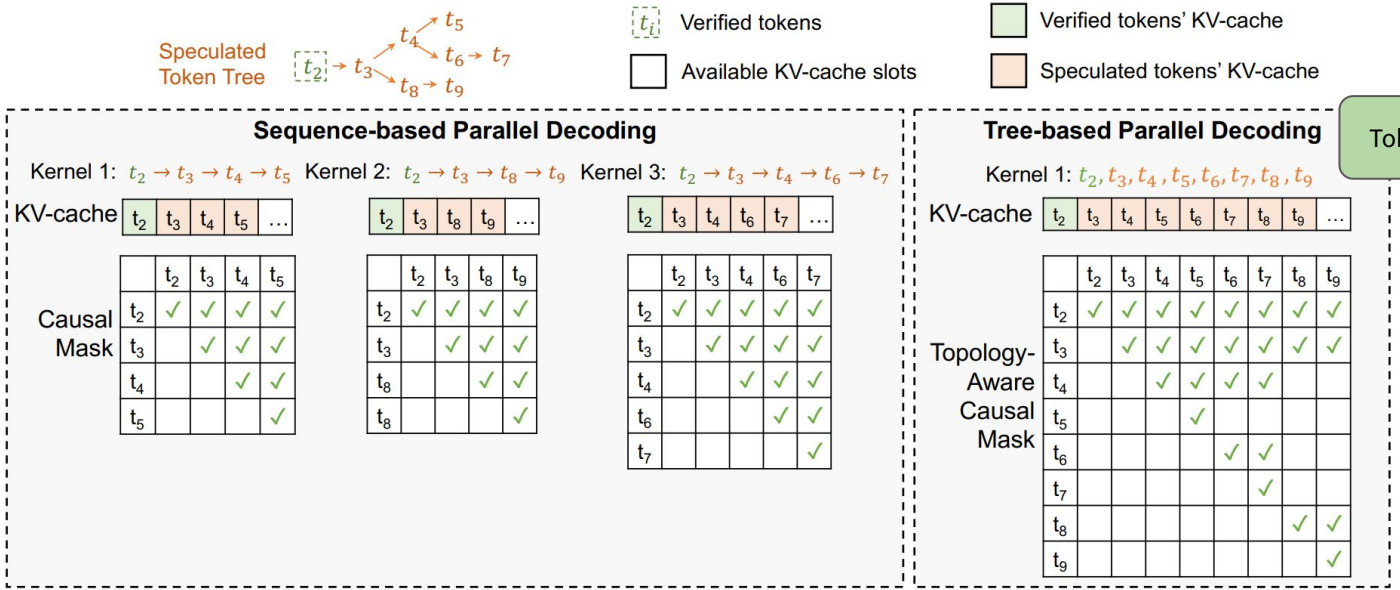
Merge-based construction

- Use multiple SSMs: Generate a separate sequence for each SSM
- Merge based on common tokens used
- Adaptive Boosting: Fine-tune SSMs with the LLM to provide more aligned answers



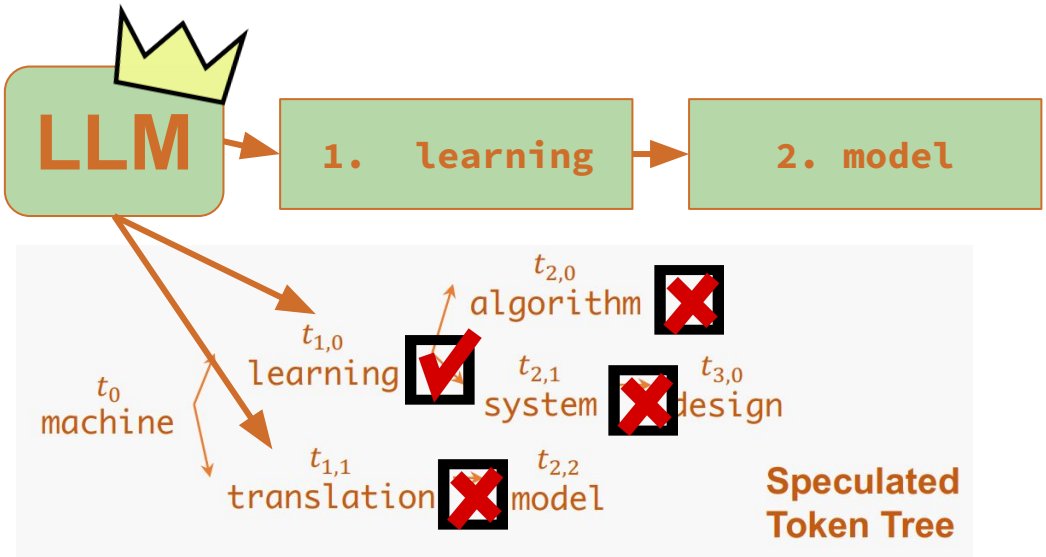
Verification: Calculating attention with *TreeAttention*

- Topology-aware causal mask: add all tokens to the KV-cache and calculate attention, but apply a mask to which tokens are used for each node
- Calculate attention in parallel across all sequences



Verification: verifying each token

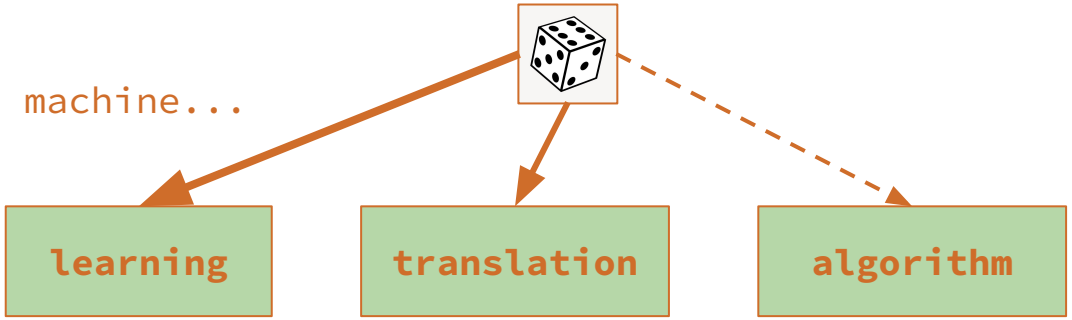
Greedy Decoding

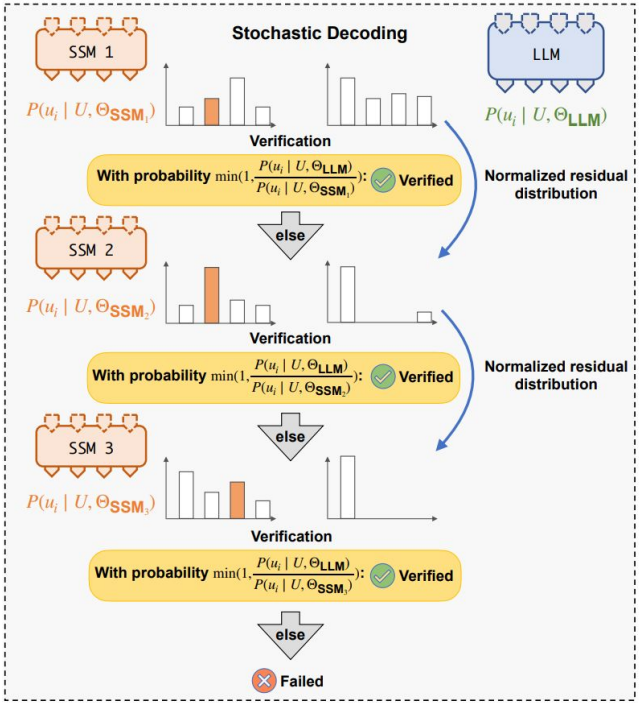


Easier to verify

Verification: verifying each token

Stochastic Decoding





More similar to sampling used in incremental decoding

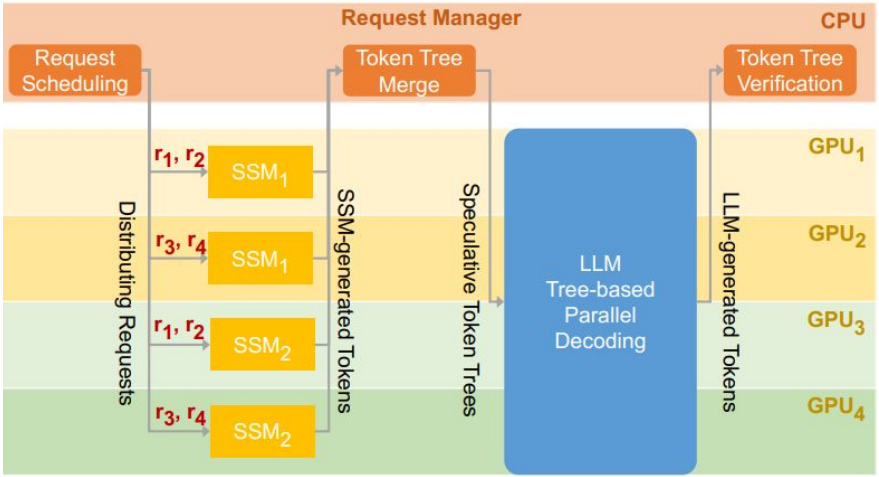
Workflow

LLM served with parallelism from MegatronLM, with an added Request Manager

- Schedules requests by iteration, not whole requests

Added overhead and extra SSM inference has little effect on memory and compute overhead

- Too many speculative sequences increases verification time



Evaluation Setup

LLMs: LLaMA-7b, OPT-30B, LLaMA-65B

SSMs: LLaMA-68M, OPT-125M

Datasets: Chatbot Instruction Prompts, ChatGPT Prompts, WebQA, Alpaca, PIQA

Hardware: 2 AWS g5.12xlarge instances

- NVIDIA A10 24GB GPUs, 48 CPU cores, 192 GB DRAM

Result: 1.2-1.5x faster serving latency than sequence-based speculative inference

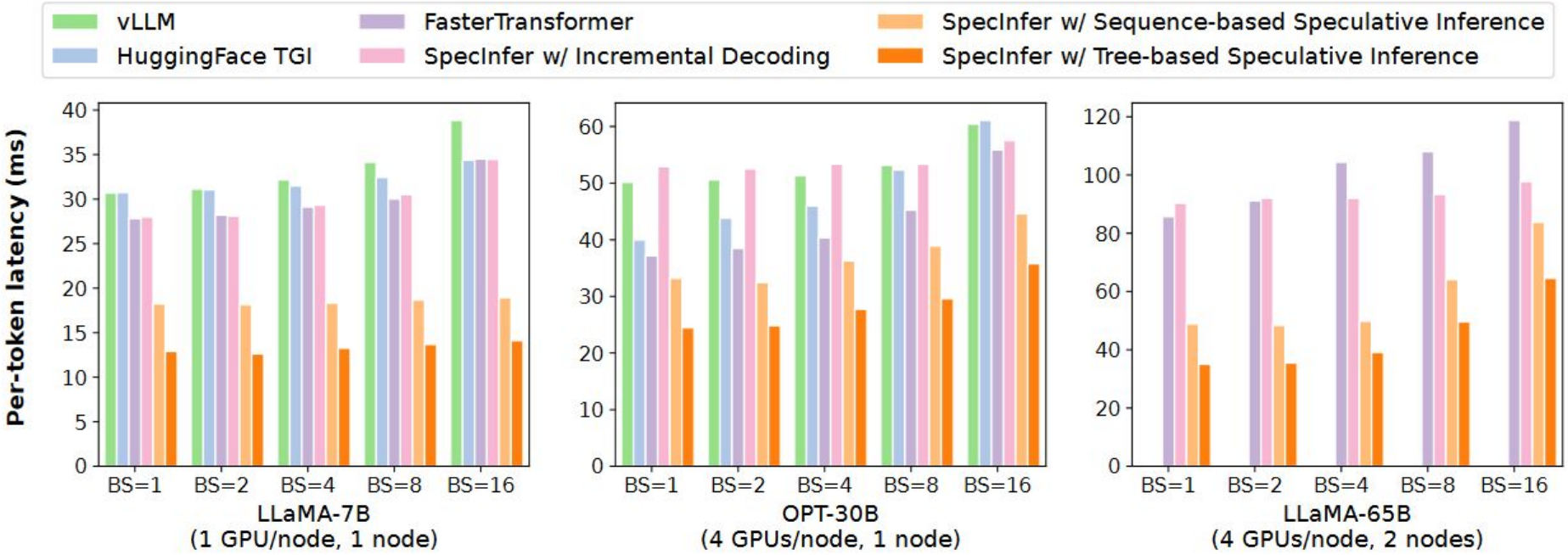


Figure 7. Comparing the end-to-end inference latency of SpecInfer with existing systems. Numbers in parenthesis show the number of GPUs and compute node used to serve each LLM. All systems parallelize LLM inference by combining tensor model parallelism (within a node) and pipeline parallelism (across nodes).

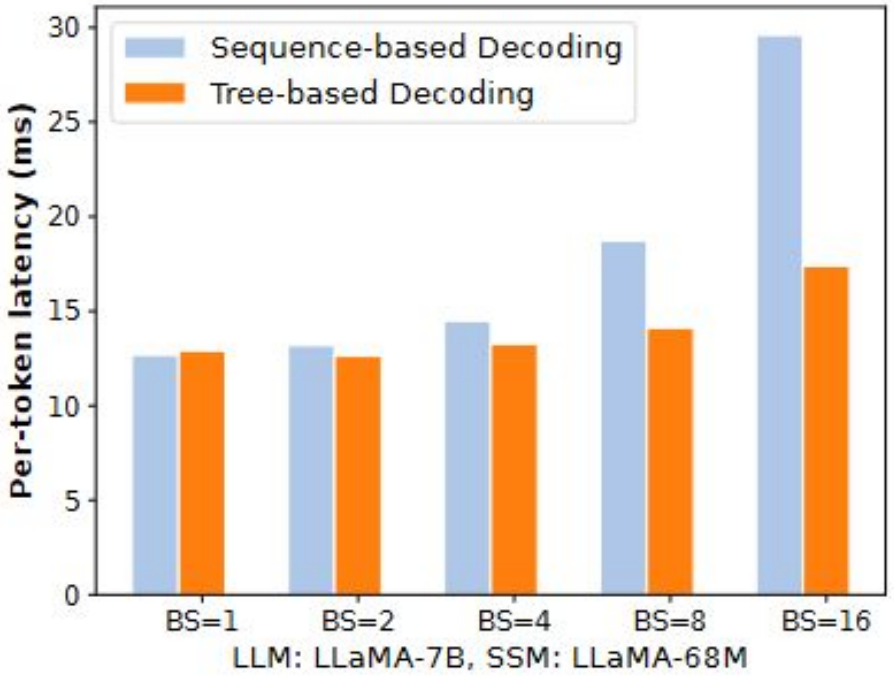


Figure 11. Comparing SpecInfer's tree-based parallel decoding with the sequence-based decoding mechanism employed by existing LLM inference systems.

Drawbacks

- As batch size increases, available GPU resources decrease, so there are less resources available for SpecInfer to perform tree-based parallel decoding
- Using a large tree width increases the latency to verify a token tree due to less sparse GPU resources that can be leveraged by SpecInfer

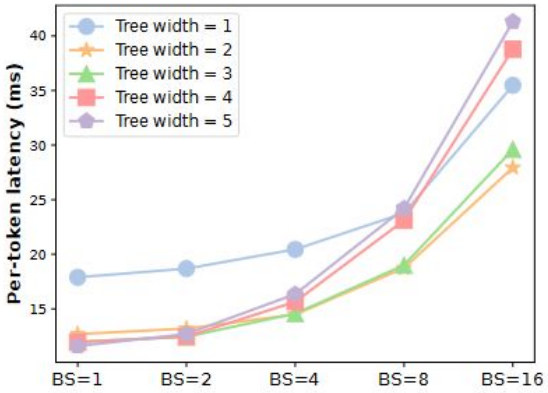
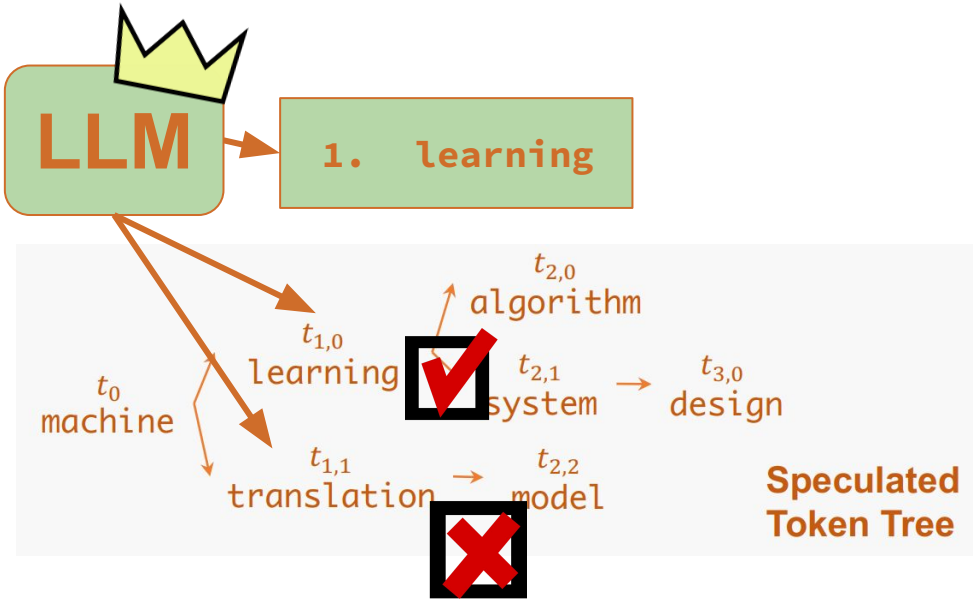


Figure 10. SpecInfer's end-to-end inference latency with different tree widths. We use LLaMA-7B and LLaMA-68M as the LLM and SSM.

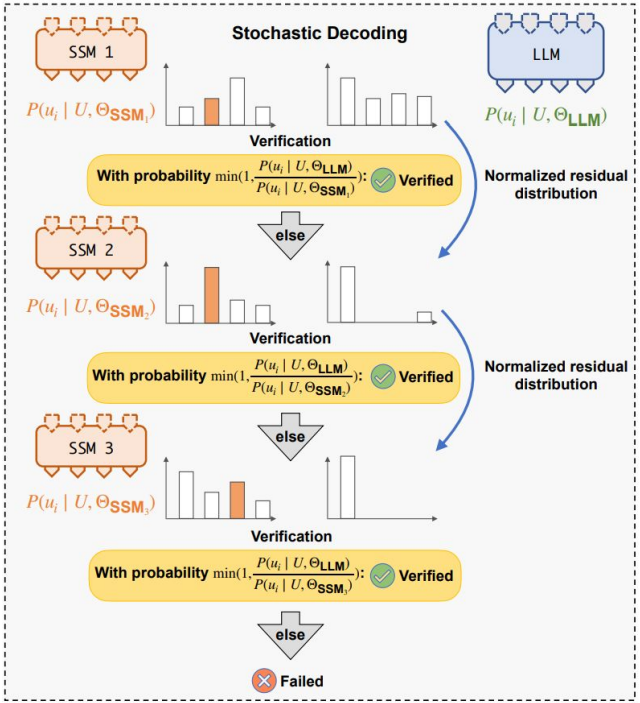
Verification: verifying each token

Greedy Decoding



Easier to verify

Stochastic Decoding



More similar to sampling used in incremental decoding

Key Result: Improved HBM Writes

Theorem 2. *Let N be the sequence length, d be the head dimension, and M be size of SRAM with $d \leq M \leq Nd$. Standard attention (Algorithm 0) requires $\Theta(Nd + N^2)$ HBM accesses, while* FLASHATTENTION *(Algorithm 1) requires $\Theta(N^2 d^2 M^{-1})$ HBM accesses.*

- **N:** Sequence length (up to 8k - 128k)
- **d:** Query dimension (~512)
- **M:** Size of SRAM on a single SM (~100KB)
- Improvement because M >> d^2
- Holds for FA-2 as well.

Dao et al., FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

SpecInfer

Basis: what if we predicted what the (slow) LLM would output next?

Idea: make a tree of possible predictions to maximize accuracy (with smaller LLMs)