Hardware-aware Sparsity for Efficient and Accurate Training

Tri Dao (trid@cs.stanford.edu)
AHA Retreat - 8/25/2022

Tri Dao, Beidi Chen, Nimit Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, Christopher Ré. Monarch: Expressive Structured Matrices for Efficient and Accurate Training. *ICML 2022* ***Outstanding Paper Runner-up***.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Ruda, Christopher Ré. Flash Attention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *arXiv preprint arXiv:2205.14135.*
https://github.com/HazyResearch/flash-attention.

Sparsity: A Natural Approach to Reduce Computation

Challenges with structured linear maps (low-rank, sparse, Fourier):

Efficiency-quality tradeoffs:

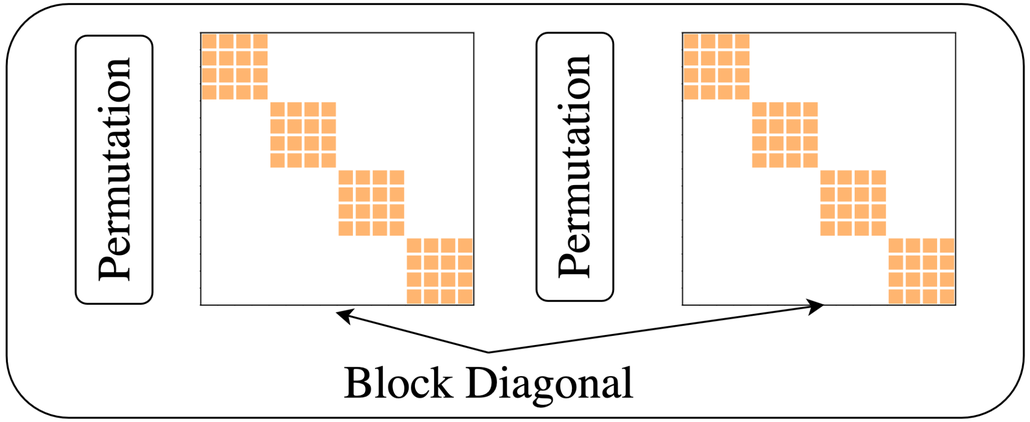
- **Efficiency**: on modern hardware (GPU)
- Quality: how **expressive** are the weight matrices (can they represent commonly used transforms)

Monarch: one of the first sparse training methods to achieve wall-clock speedup while maintaining quality.

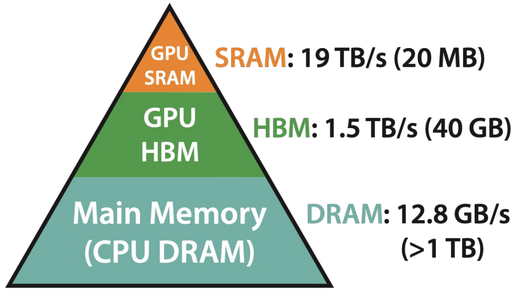
FlashAttention: 2-4x faster exact attention, 10x memory saving; fastest BERT model on cloud instances (MLPerf 2.0).

Main Idea: Hardware-aware Algorithms

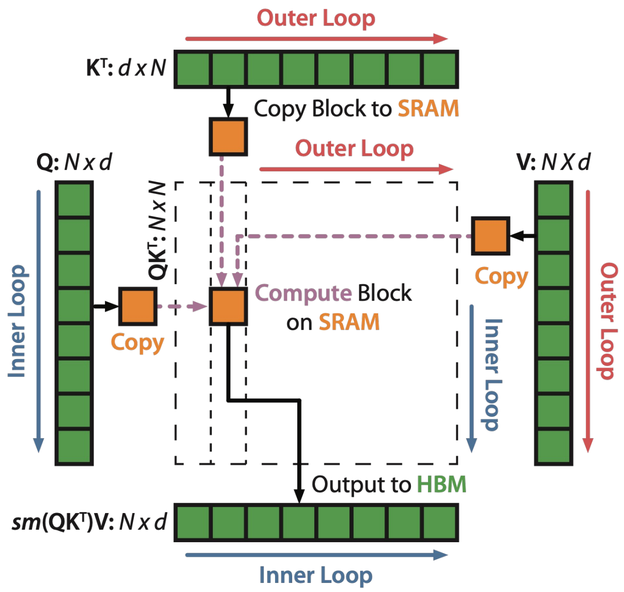
Block sparsity: friendly to block-oriented devices (GPUs)



IO-awareness: reducing reads/writes to GPU HBM yields significant speedup.



Memory Hierarchy with Bandwidth & Memory Size



FlashAttention

Monarch matrices: block-diagonal leverages efficient batch-matrix-multiply

FlashAttention: tiling to load inputs by blocks from HBM to SRAM for computation, reducing IOs

Outline

Part 1 Monarch matrices

Hardware-efficiency & Expressiveness
Ways to use sparse models
Applications: language model, computer vision

Part 2 FlashAttention

Attention is bottlenecked by memory reads/writes
Tiling and recomputation to reduce IOs
Applications: faster Transformers, better Transformers with long context

Part 3 Future Directions

Hardware-aware algorithms in other operations

Outline

Part 1 Monarch matrices

Hardware-efficiency & Expressiveness
Ways to use sparse models
Applications: language model, computer vision

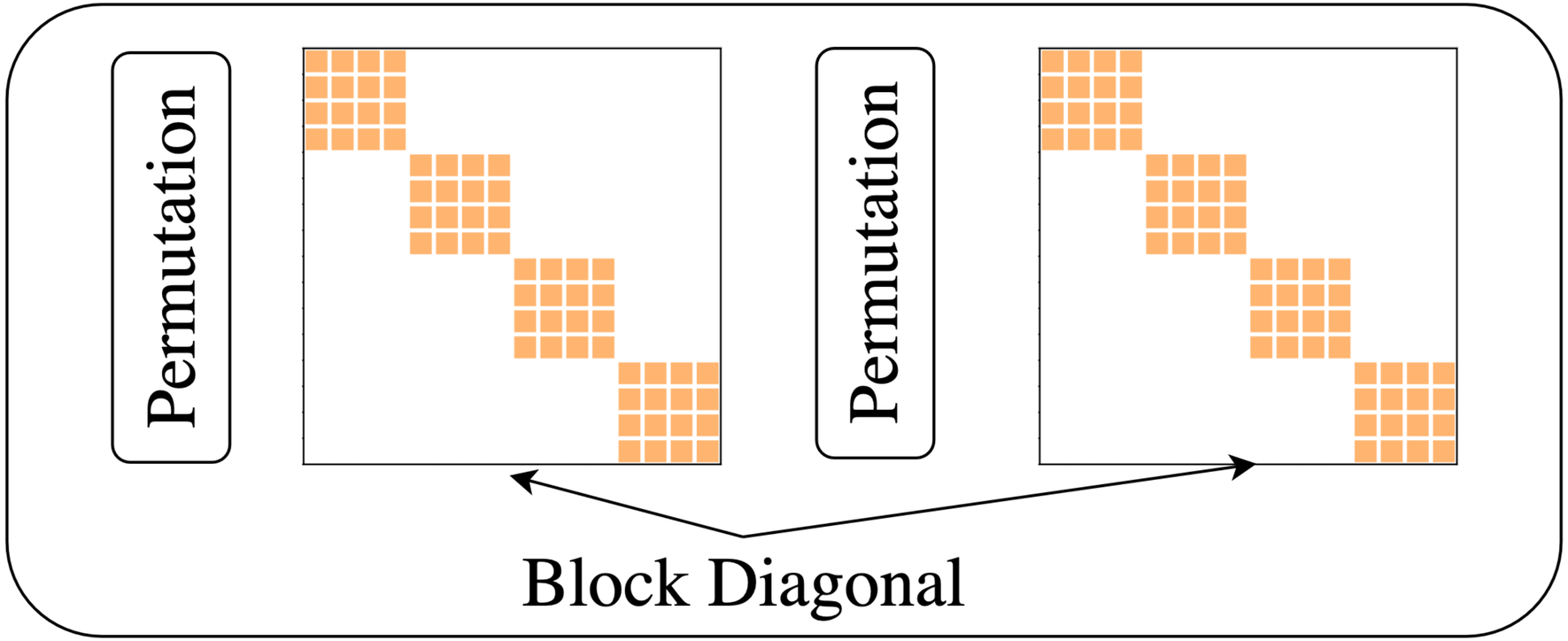
Part 2 FlashAttention

Attention is bottlenecked by memory reads/writes
Tiling and recomputation to reduce IOs
Applications: faster Transformers, better Transformers with long context

Part 3 Future Directions

Hardware-aware algorithms in other operations

Monarch Matrices: Efficient and Expressive



Focus of this talk

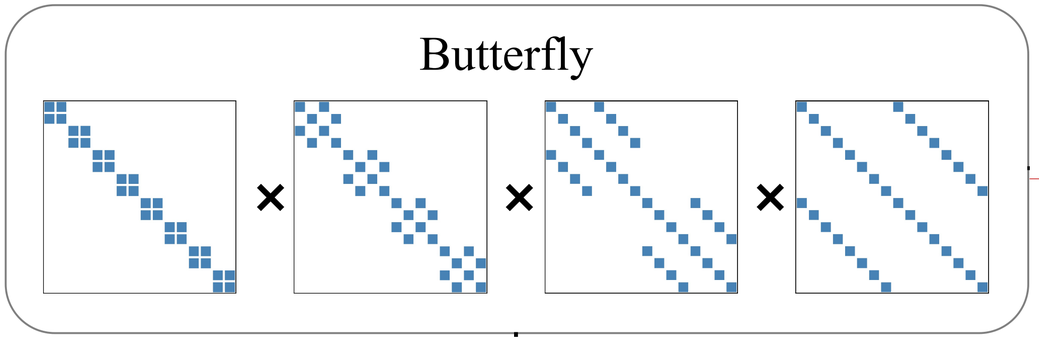
(1) Hardware-efficient
(2) Expressive

Nice and deep theory

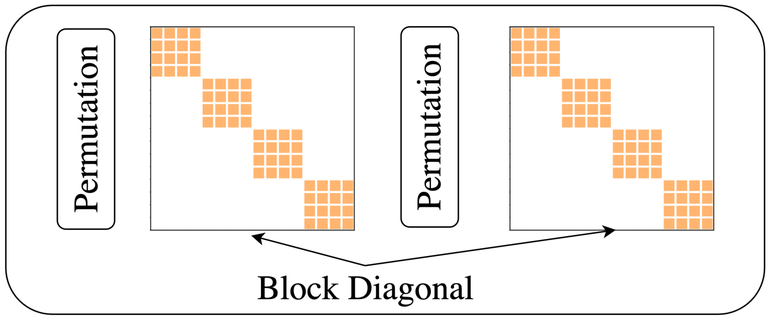
(3) Tractable projection

Motivation: divide-and-conquer structure (e.g., Cooley-Tukey FFT algorithm)

Hardware-Efficiency



Monarch



Fast in theory (O(N log N) runtime & parameters
Expressive: Can represent any structure (e.g., sparsity) almost optimally

3-5x faster on GPUs
Same expressivity

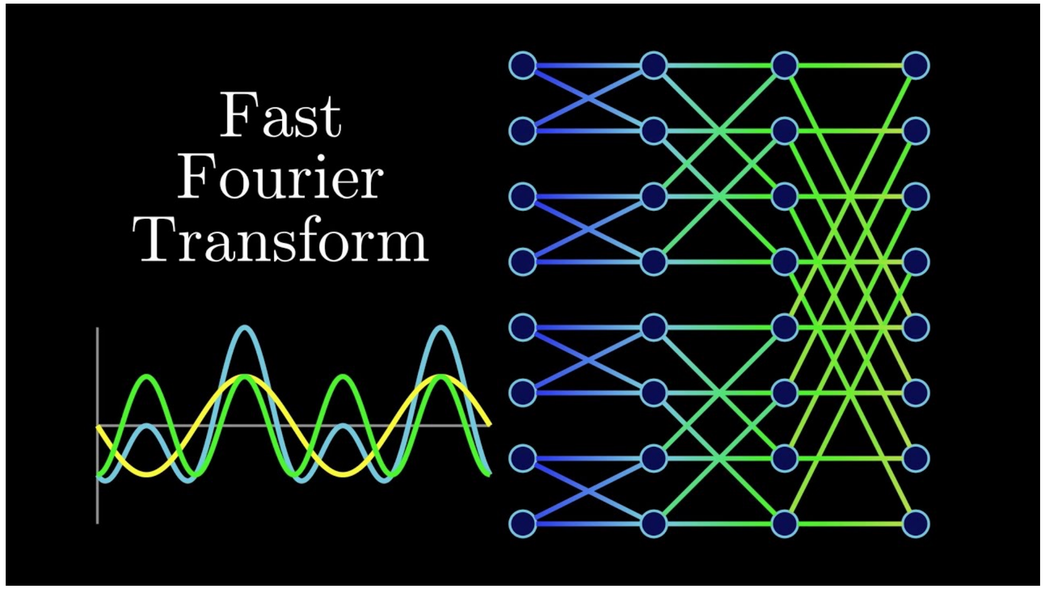
Problem 1: Not block-aligned

Problem 2: Hard to parallelize the product of many factors

Block-diagonal leverages efficient batch-matrix-multiply on GPUs

*[*Beneš, 65; *Parker, 95; Matthieu & LeCun, 14; De Sa et al., 18, Dao et al., 19]*

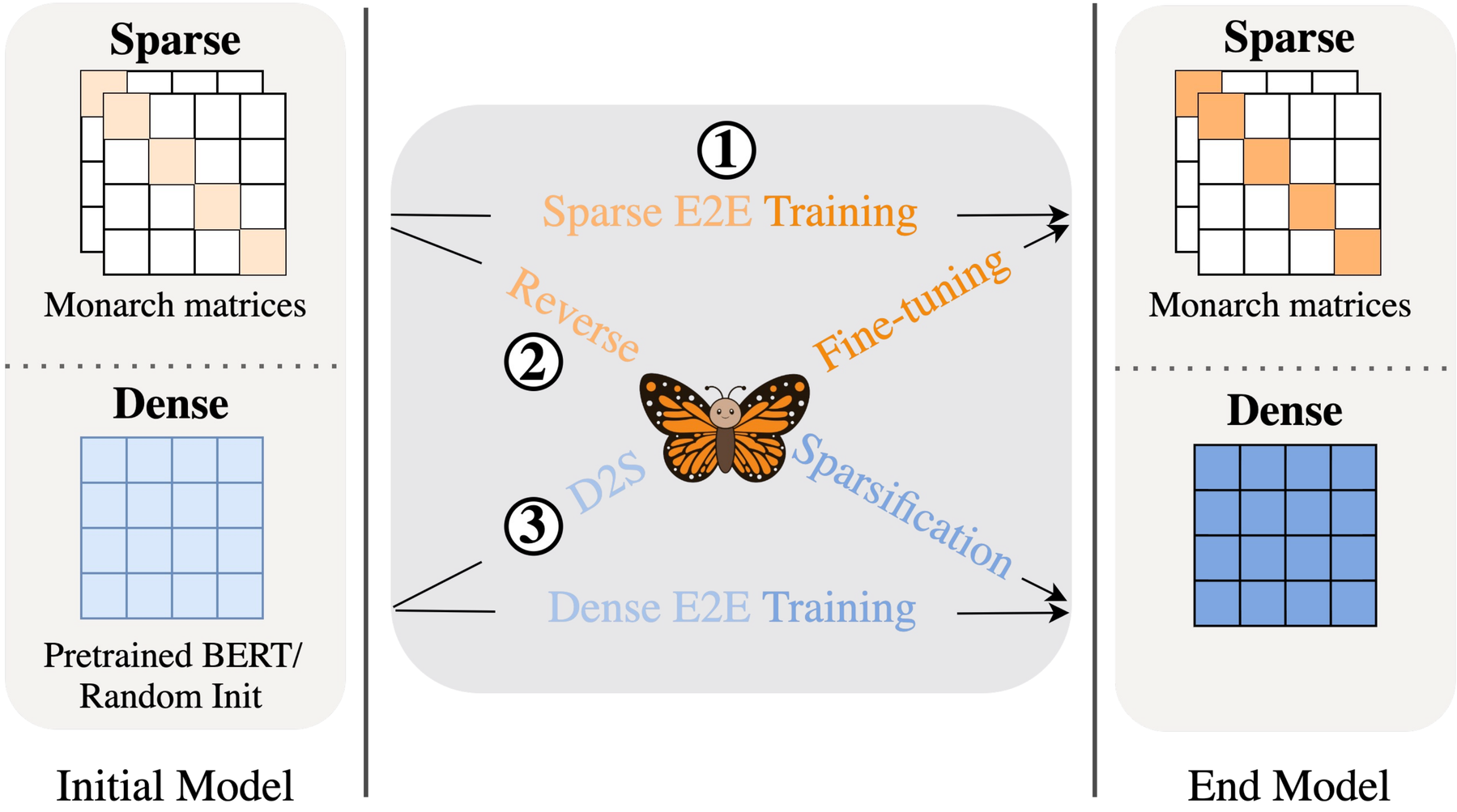
Expressiveness: Monarch Can Represent Many Structured Matrices



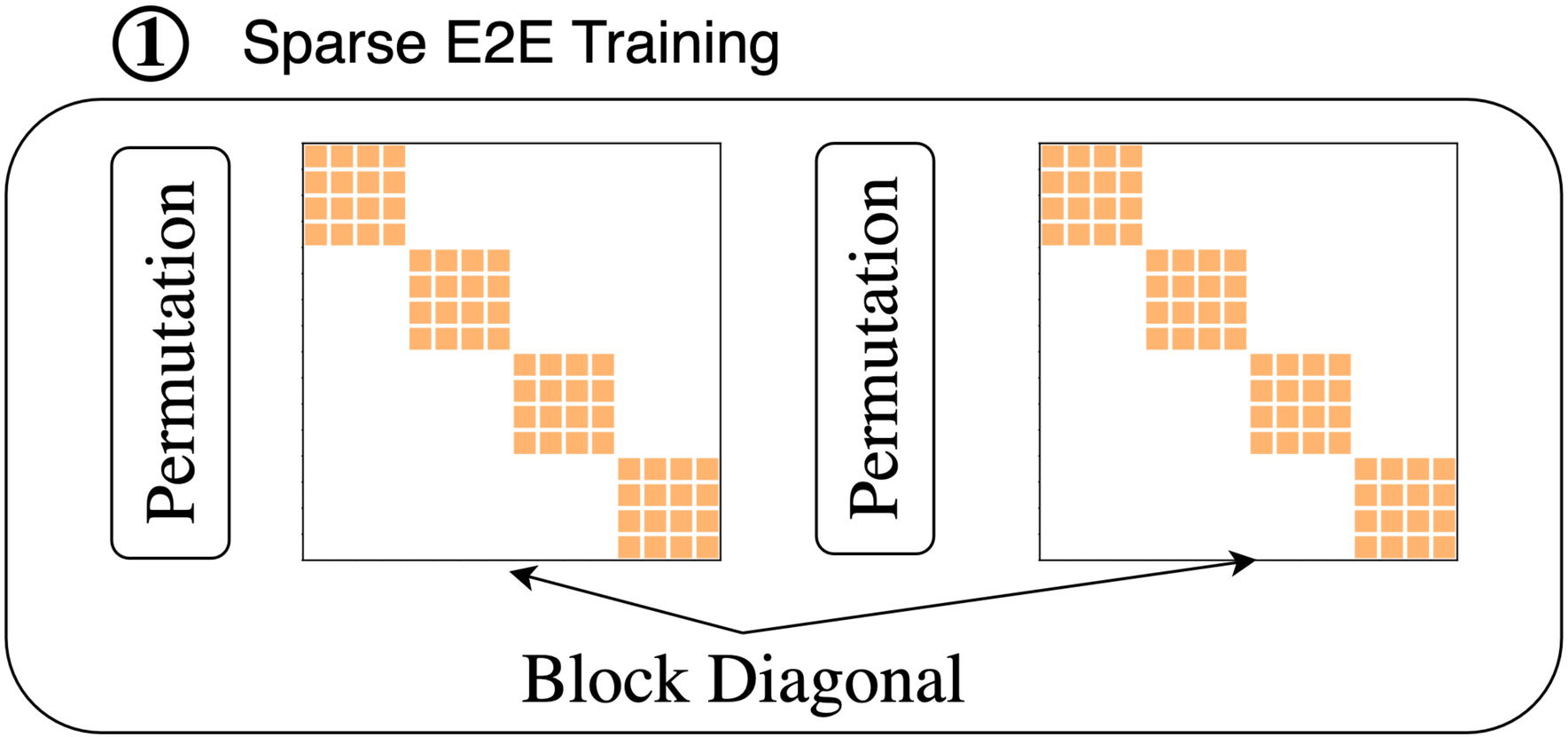
(Elementwise) sparsity & low-rank can't represent most of these structures

Monarch can represent & learn these structures

Three Ways to Use Sparse Models

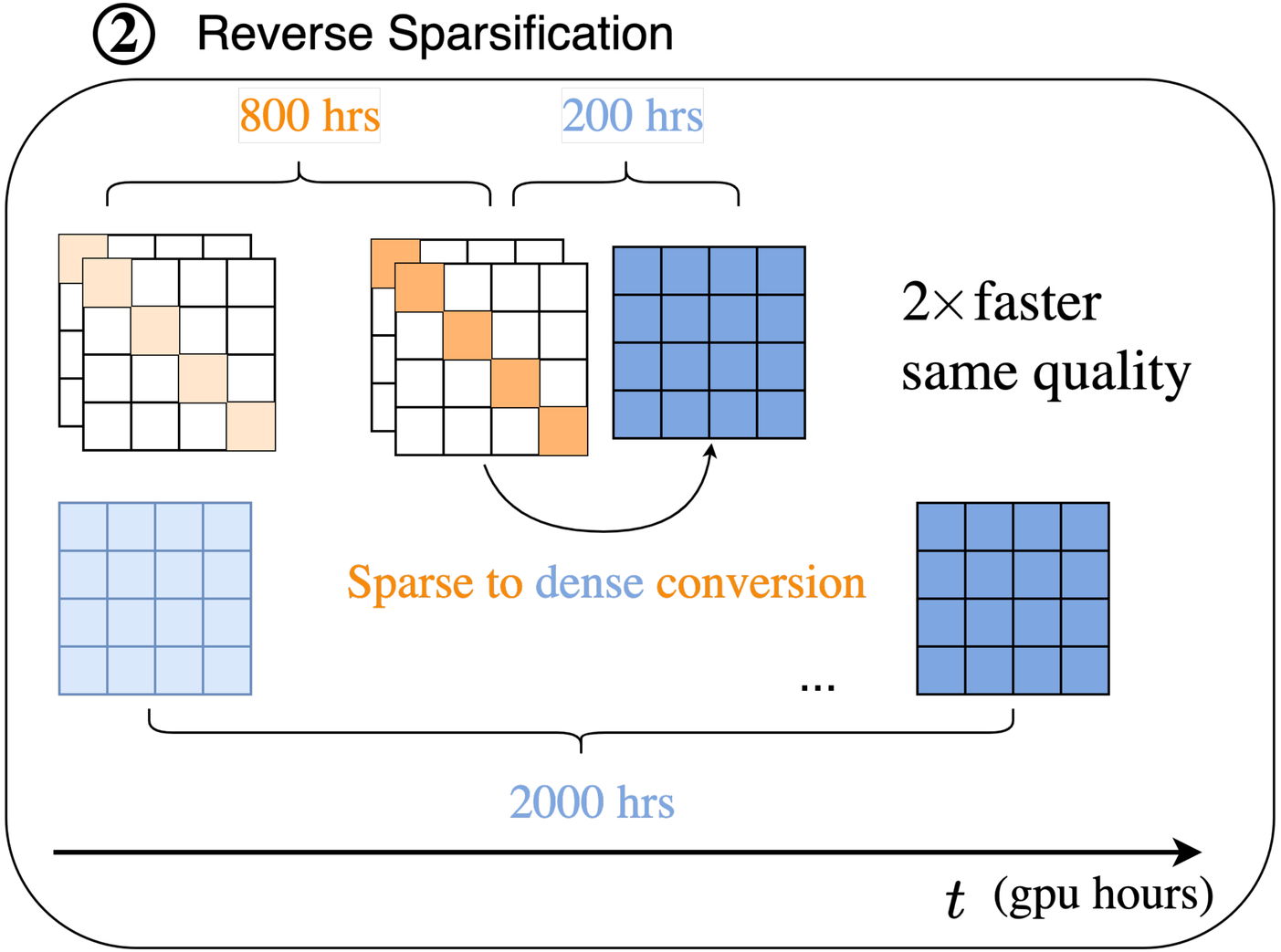


Sparse End-to-End Training



Replace dense weight matrices (e.g., attention & FFN) with Monarch matrices for efficiency

Sparse-to-Dense Training (reverse sparsification)



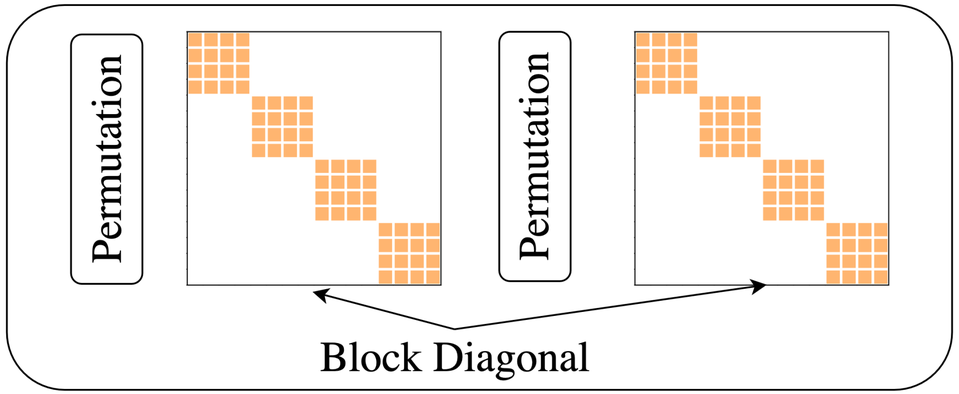
Validation: Sparse End-to-End Training

Benchmark tasks

Model	WikiText 103(ppl)	Speedup
GPT-2 Small	20.6	-
Monarch GPT-2-small	20.7	**2.0 x**

Model	ImageNet (acc)	Speedup
ViT-Base	78.5	-
Monarch ViT-Base	78.7	**1.8 x**

① Sparse E2E Training



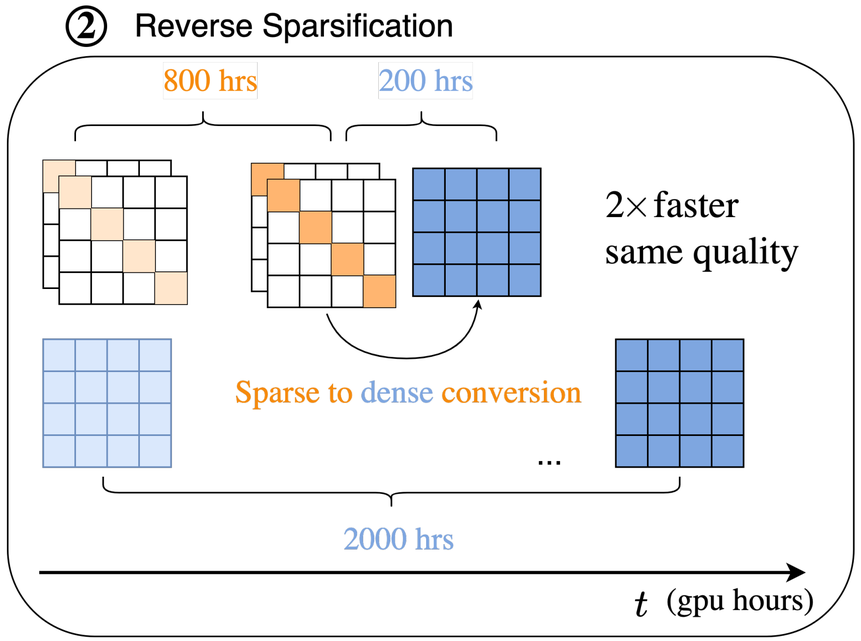
Other applications: PDEs solving, MRI reconstruction

Speeds up training without losing performance 🚀!

Validation: Sparse-to-Dense Training

GPT2 implemenation	OpenWebText perplexity	Speeup
HuggingFace	18.1	-
Monarch	18.1	**1.8x**

80% sparse training, 20% dense training



1.8x speedup from using Monarch as an intermediate representation

Summary

Monarch: hardware-efficient, expressive matrices

Ways to use: sparse end-to-end training,
sparse-to-dense training (reverse sparsification)

Upshot: wallclock-time speedup with sparse training,
maintaining model quality

Code: https://github.com/HazyResearch/monarch

Outline

Part 1 **Monarch matrices**

Hardware-efficiency & Expressiveness
Ways to use sparse models
Applications: language model, computer vision

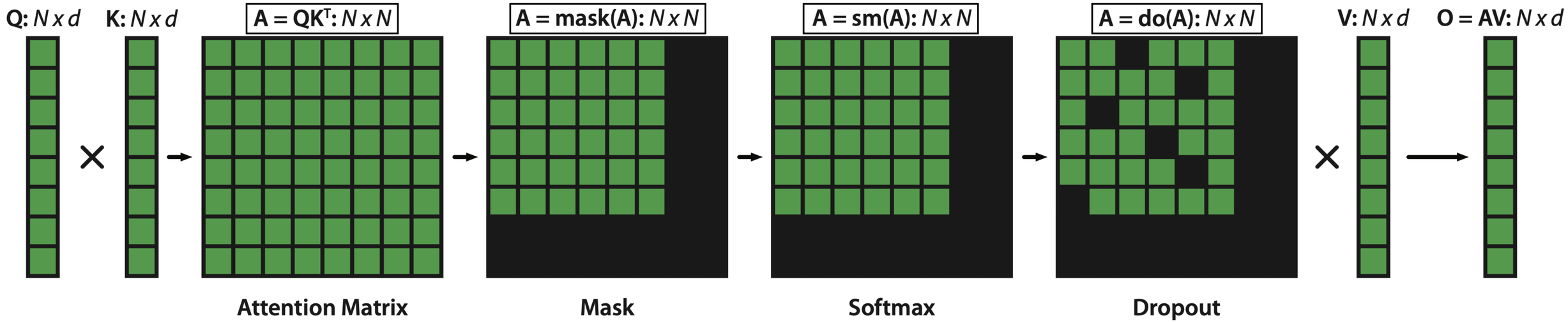
Part 2 **FlashAttention**

Attention is bottlenecked by memory reads/writes
Tiling and recomputation to reduce IOs
Applications: faster Transformers, better Transformers with long context

Part 3 **Future Directions**

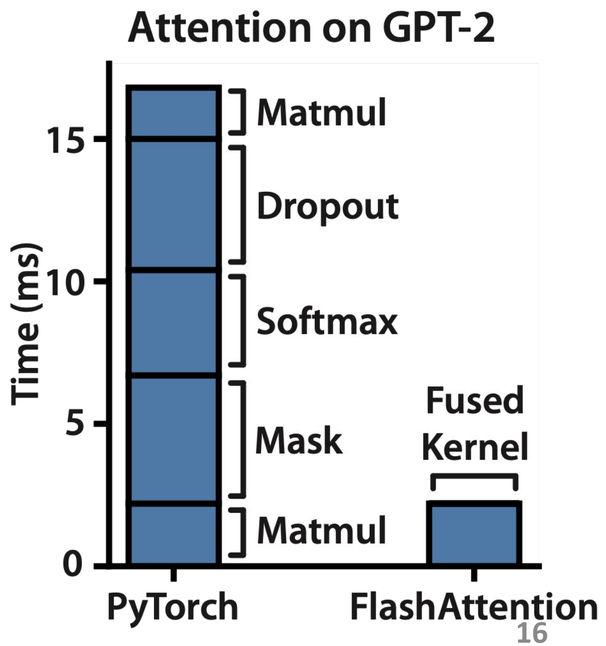
Hardware-aware algorithms in other operations

Background: Attention is Bottlenecked by Memory Reads/Writes

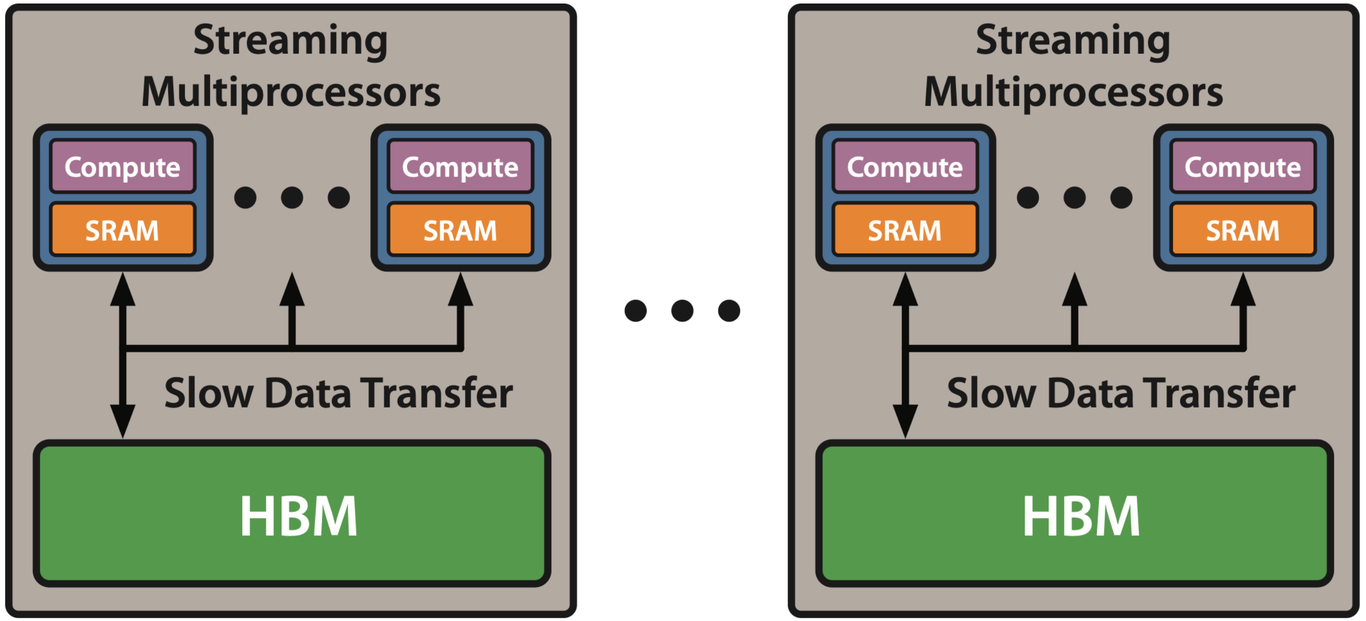


$$\mathbf{O} = \text{Dropout}(\text{Softmax}(\text{Mask}(\mathbf{QK}^T)))\mathbf{V}$$

Naive implementation requires repeated R/W from slow GPU HBM

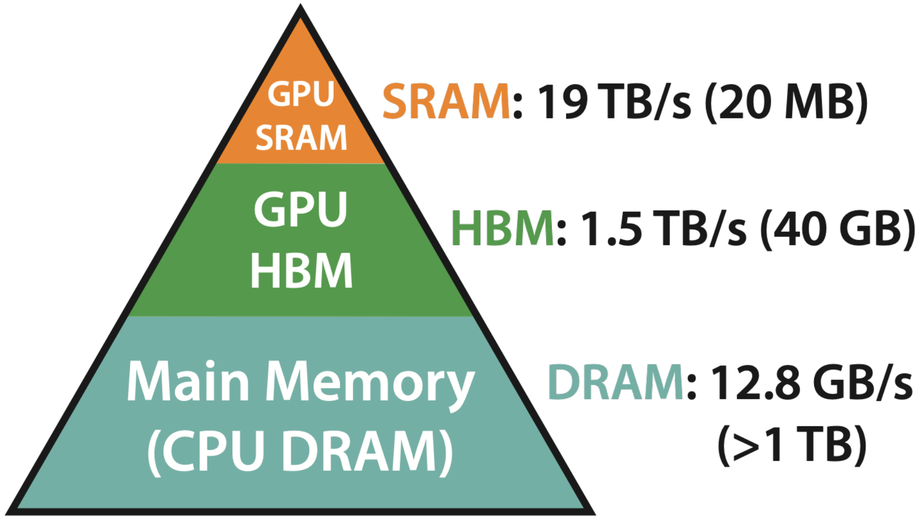


Background: GPU Compute Model & Memory Hierarchy



GPU #1

GPU #N



Memory Hierarchy with Bandwidth & Memory Size

Can we exploit the memory asymmetry to get speed up?
With IO-awareness (accounting for R/W to different levels of memory)

Blogpost: Horace He, Making Deep Learning Go Brrrr From First Principles.

How to Reduce HBM Reads/Writes: Compute by Blocks

Challenges: (1) compute softmax reduction without access to full input.
(2) backward without the large attention matrix from forward.

Tiling: Restructure algorithm to load block by block from HBM to SRAM to compute attention.

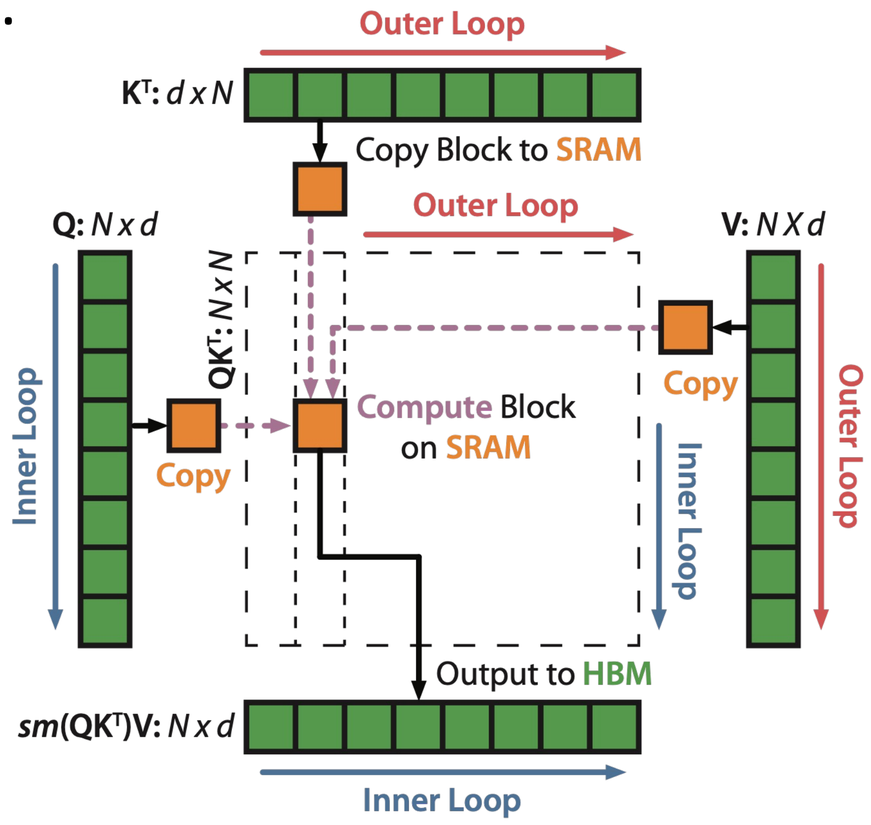
Recomputation: Don’t store attn. matrix from forward, recompute it in the backward.

Implementation: fused CUDA kernel for fine-grained control of memory accesses.

Tiling

Decomposing large softmax into smaller ones by scaling.

1. Load inputs by blocks from HBM to SRAM.
2. On chip, compute attention output wrt that block.
2. Update output in HBM by scaling.



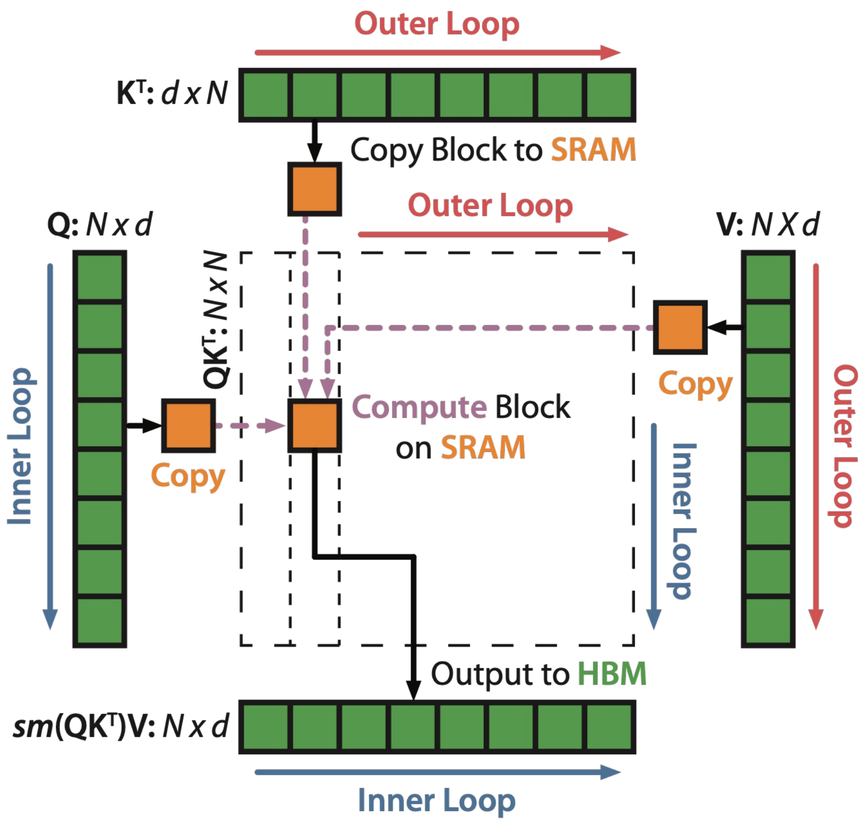
FlashAttention

Speed up forward pass by reducing IOs to HBM.

Recomputation (Backward Pass)

By storing softmax normalization factors from fwd (size N), quickly recompute attention in the bwd from inputs in SRAM.

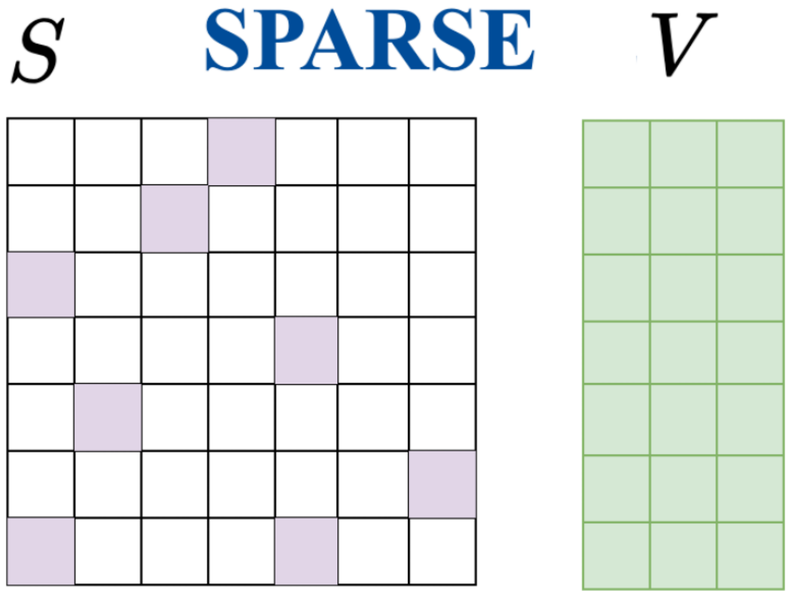
Attention	Standard	FlashAttention
GFLOPs	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtime (ms)	41.7	7.3

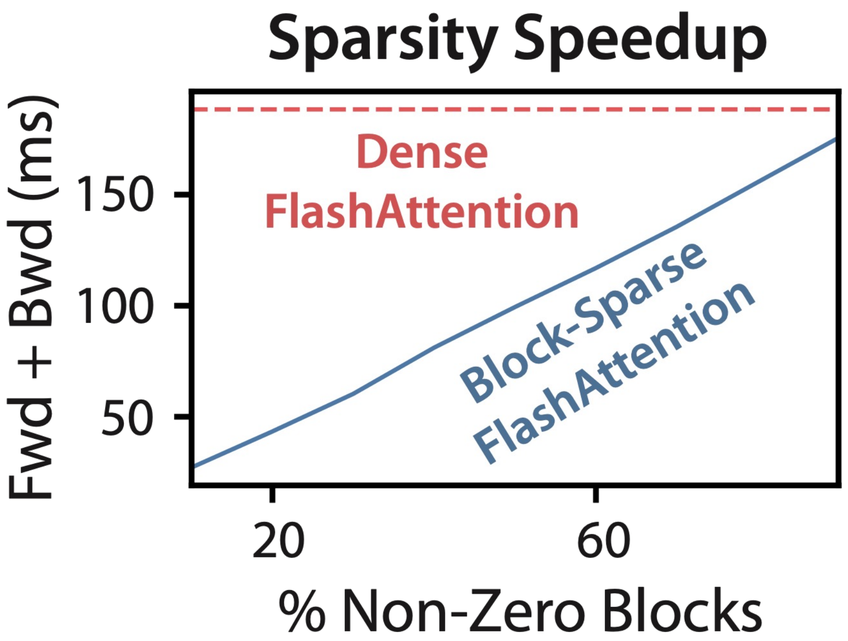


FlashAttention

Speed up backward pass even with increased FLOPs.

Extension to Block-Sparse Attention





Just skip the zero blocks: speedup proportional to nonzero fraction

Faster Training: MLPerf Record for Training BERT-large

MLPerf: (highly optimized) standard benchmark for training speed

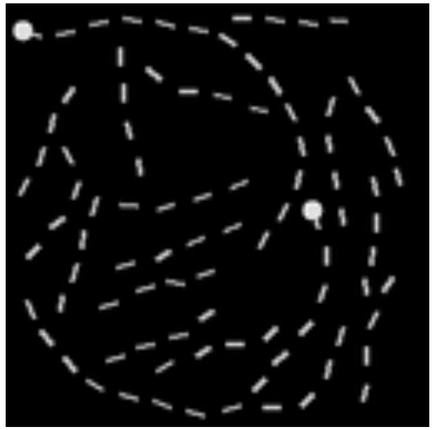
Time to hit an accuracy of 72.0% on MLM from a fixed checkpoint, averaged across 10 runs on 8 x A100 GPUs

BERT Implementation	Training time (minutes)
Huggingface [91]	55.6 ± 3.9
Nvidia MLPerf 1.1 [63]	20.0 ± 1.5
FlashAttention (ours)	$\mathbf{17.4} \pm 1.4$

FlashAttention outperforms the previous MLPerf record by 15%
(and 3.2x faster than Huggingface BERT)

Mattson et al. *MLSys 2020*.

Longer Sequences: Path-X, Path-256



Path-X: Tell whether two dots are connected — designed to push Transformers (no patches)

Requires sequence length 16K/64K for Path-X/Path-256

Model	Path-X	Path-256
Transformer	✗	✗
Linformer [81]	✗	✗
Linear Attention [48]	✗	✗
Performer [11]	✗	✗
Local Attention [77]	✗	✗
Reformer [49]	✗	✗
SMYRF [18]	✗	✗
FlashAttention	**61.4**	✗
Block-sparse FlashAttention	56.0	**63.1**

FlashAttention yields the first Transformer that can solve Path-X

Block-sparse FlashAttention enables Path-256

Tay et al. *ICLR 2020*.

Summary

FlashAttention: **fast** and **memory-efficient** algorithm for **exact** attention

Key algorithmic ideas: tiling, recomputation

Upshot: faster model training, better models with longer sequences

Code: https://github.com/HazyResearch/flash-attention

Outline

Part 1 **Monarch matrices**

Hardware-efficiency & Expressiveness
Ways to use sparse models
Applications: language model, computer vision

Part 2 **FlashAttention**

Attention is bottlenecked by memory reads/writes
Tiling and recomputation to reduce IOs
Applications: faster Transformers, better Transformers with long context

Part 3 **Future Directions**

Hardware-aware algorithms in other operations

Next Steps

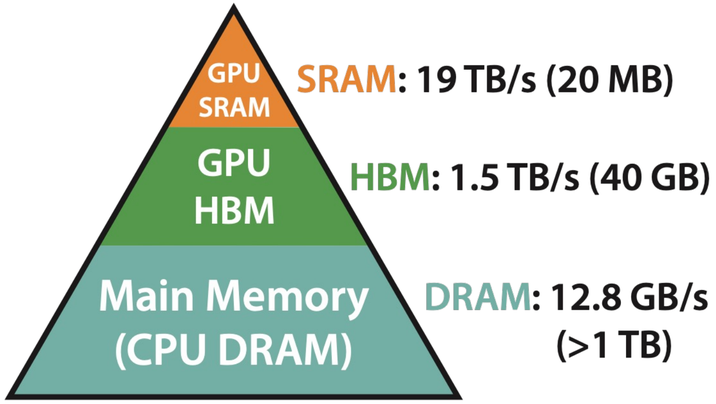
- Getting FlashAttention into more people's hands (PyTorch, HuggingFace, Jax, Triton)

- Bringing hardware-aware thinking to other ops (e.g., normalization, state-spaces)

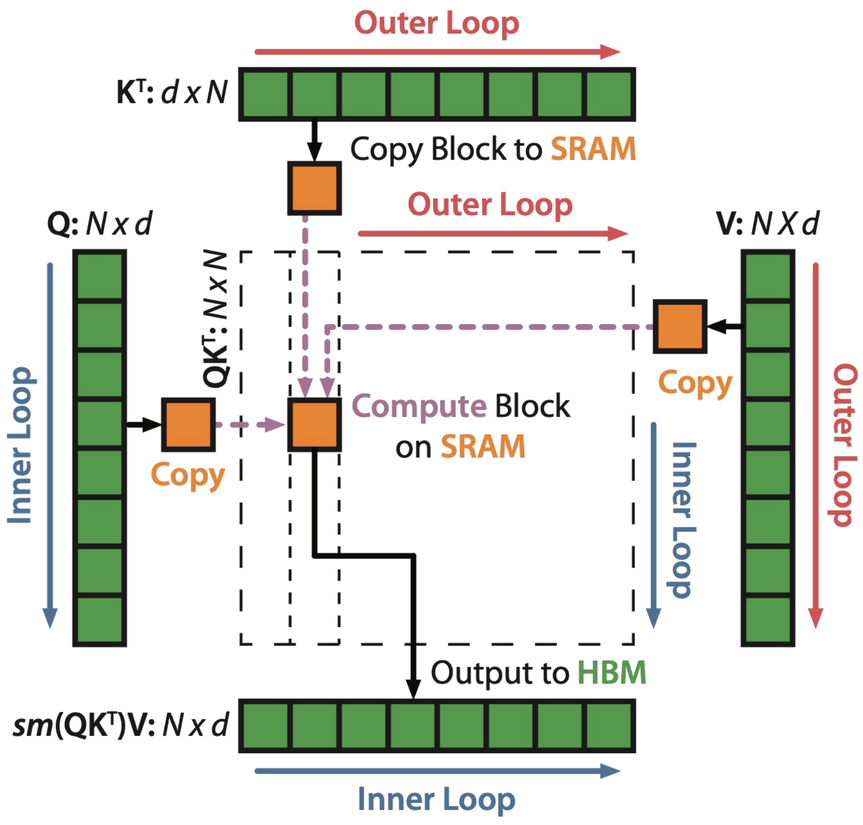
- What can we do with longer sequences?
  Language model with 8K context length? High-resolution images?

Extra Slides

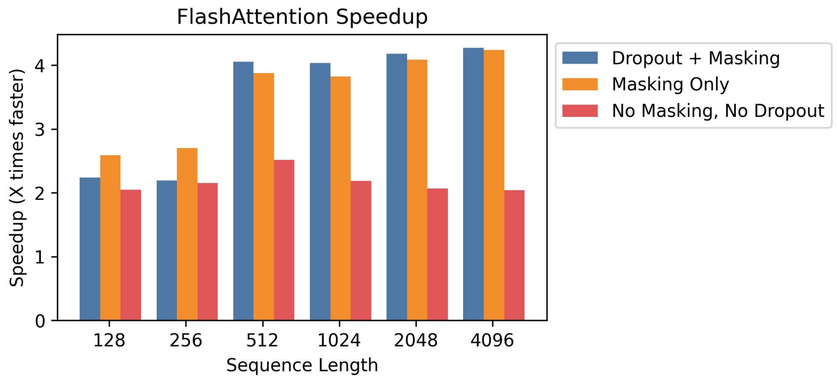
Attention: IO-Aware Tiling Yields Speed, Rem Savings, & Quality

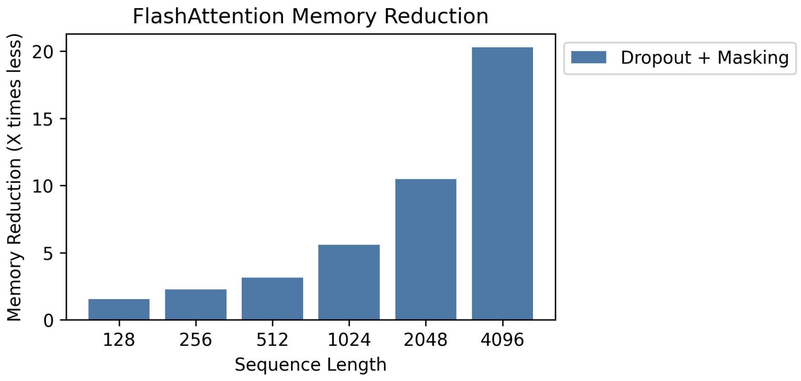


Memory Hierarchy with Bandwidth & Memory Size



FlashAttention

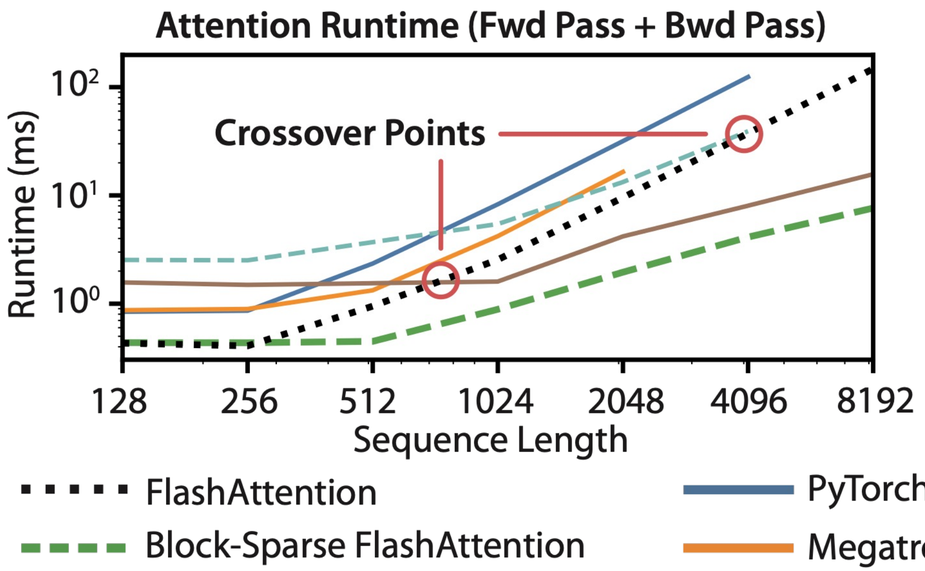




Model	Path-X	Path-256
Transformer	✗	✗
Linformer [81]	✗	✗
Linear Attention [48]	✗	✗
Performer [11]	✗	✗
Local Attention [77]	✗	✗
Reformer [49]	✗	✗
SMYRF [18]	✗	✗
FlashAttention	**61.4**	✗
Block-sparse FlashAttention	56.0	**63.1**

Benchmarks and Tradeoffs: Attention Module

Benchmarking **runtime** and **memory footprint** against sequence length



Details

Batch size: 16

Heads: 8

Head dim: 64

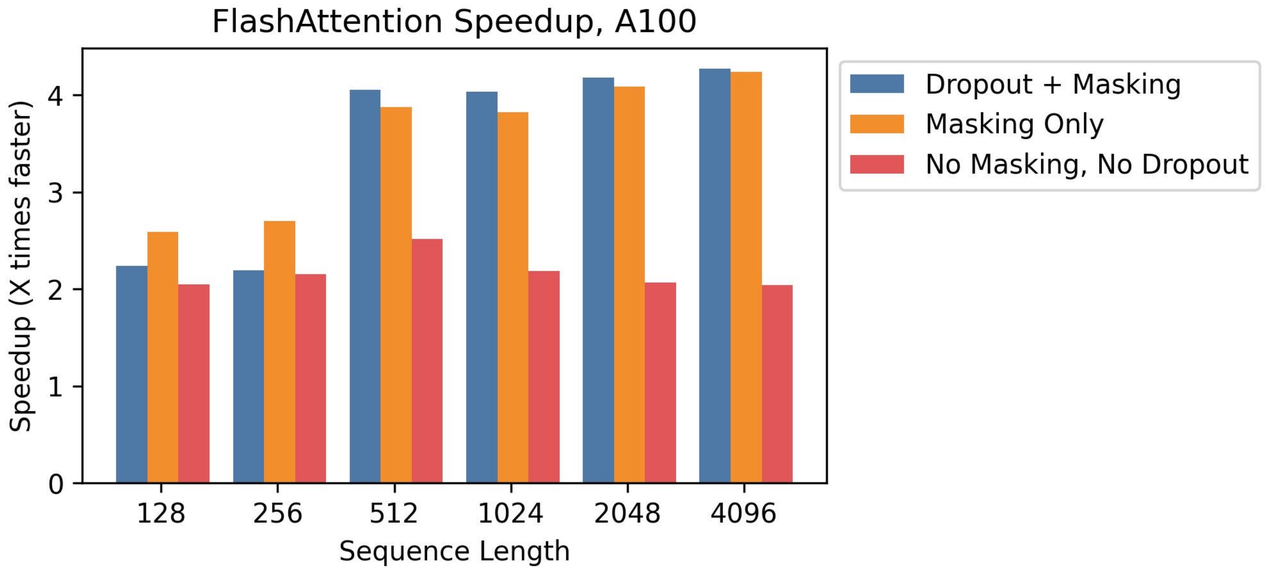
Fwd + Bwd Pass

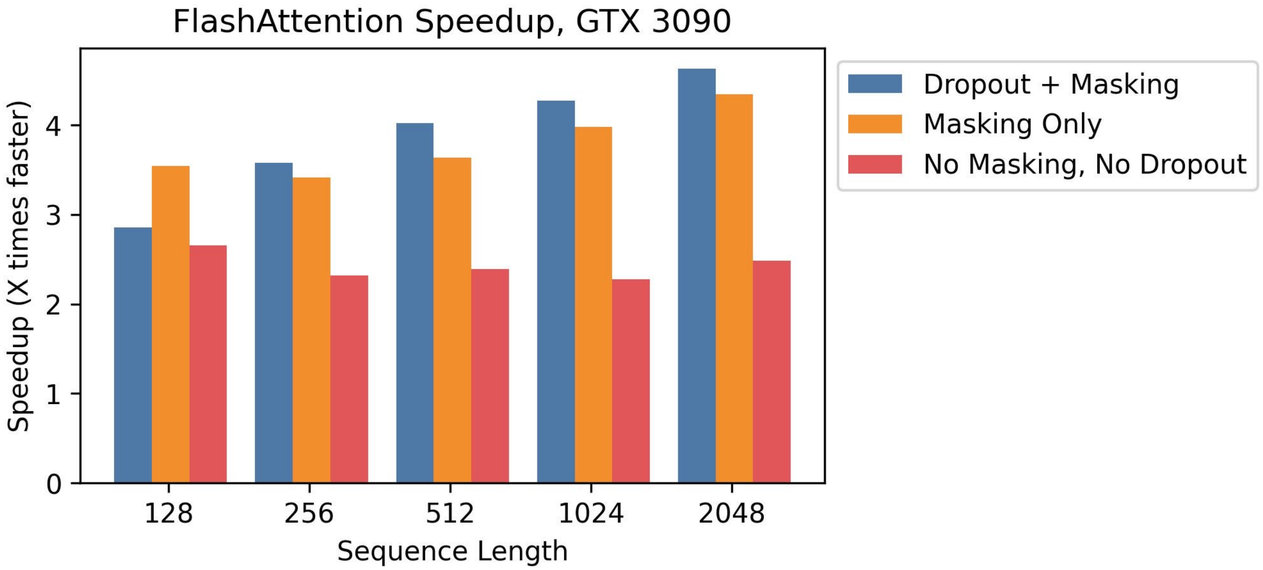
(more in paper!)

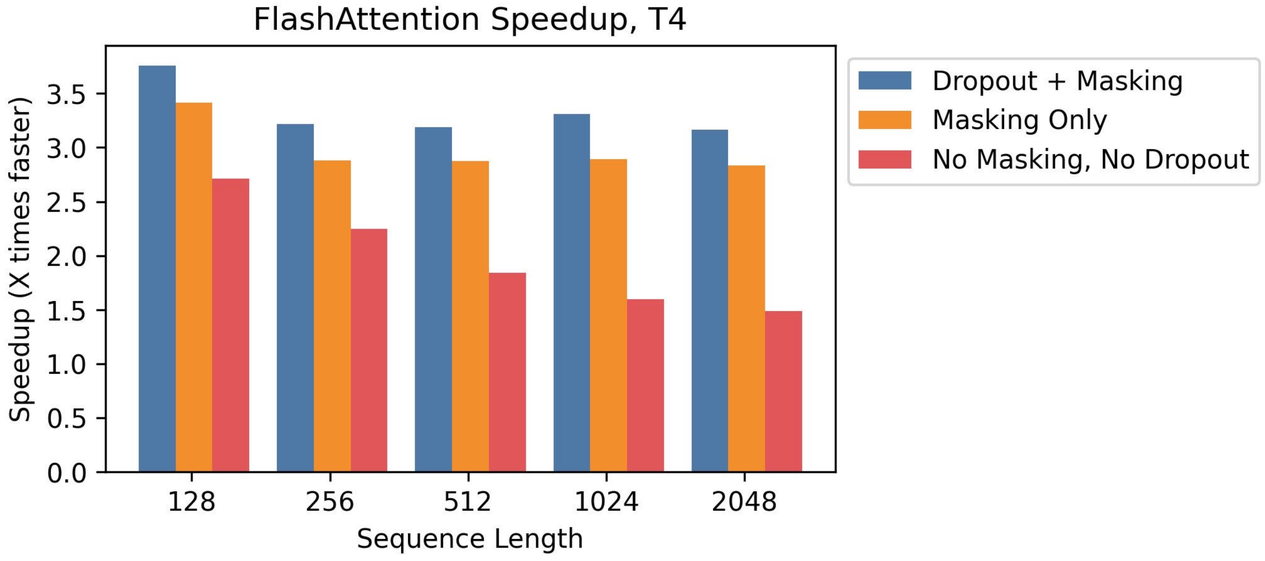
FlashAttention: faster than baselines for short sequences

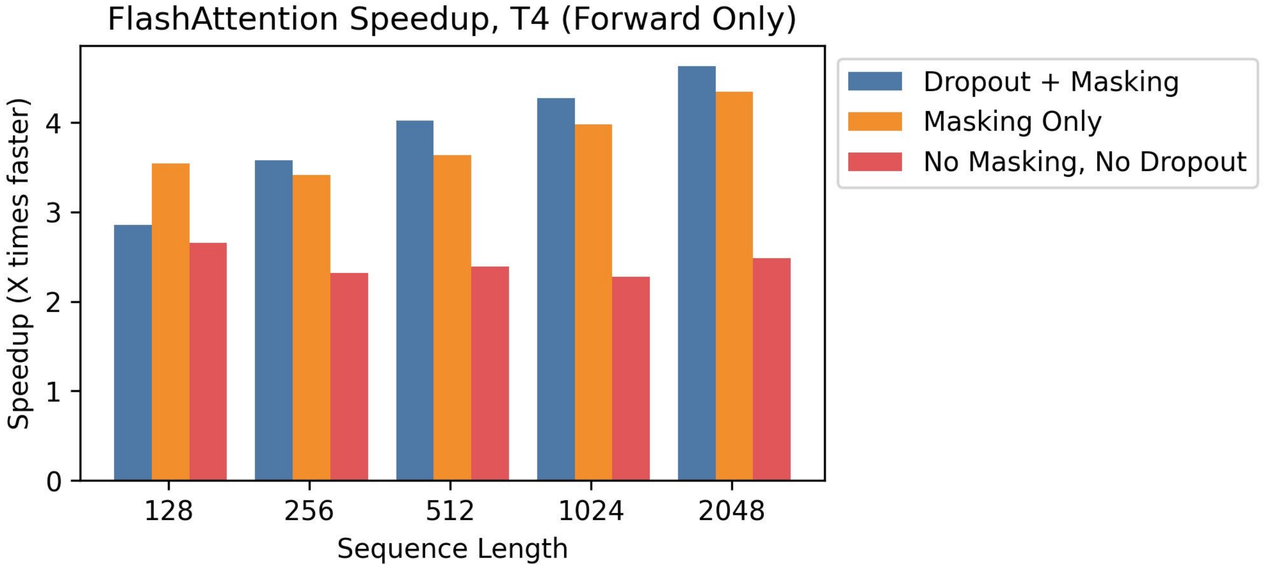
Block-sparse FlashAttention: faster everywhere

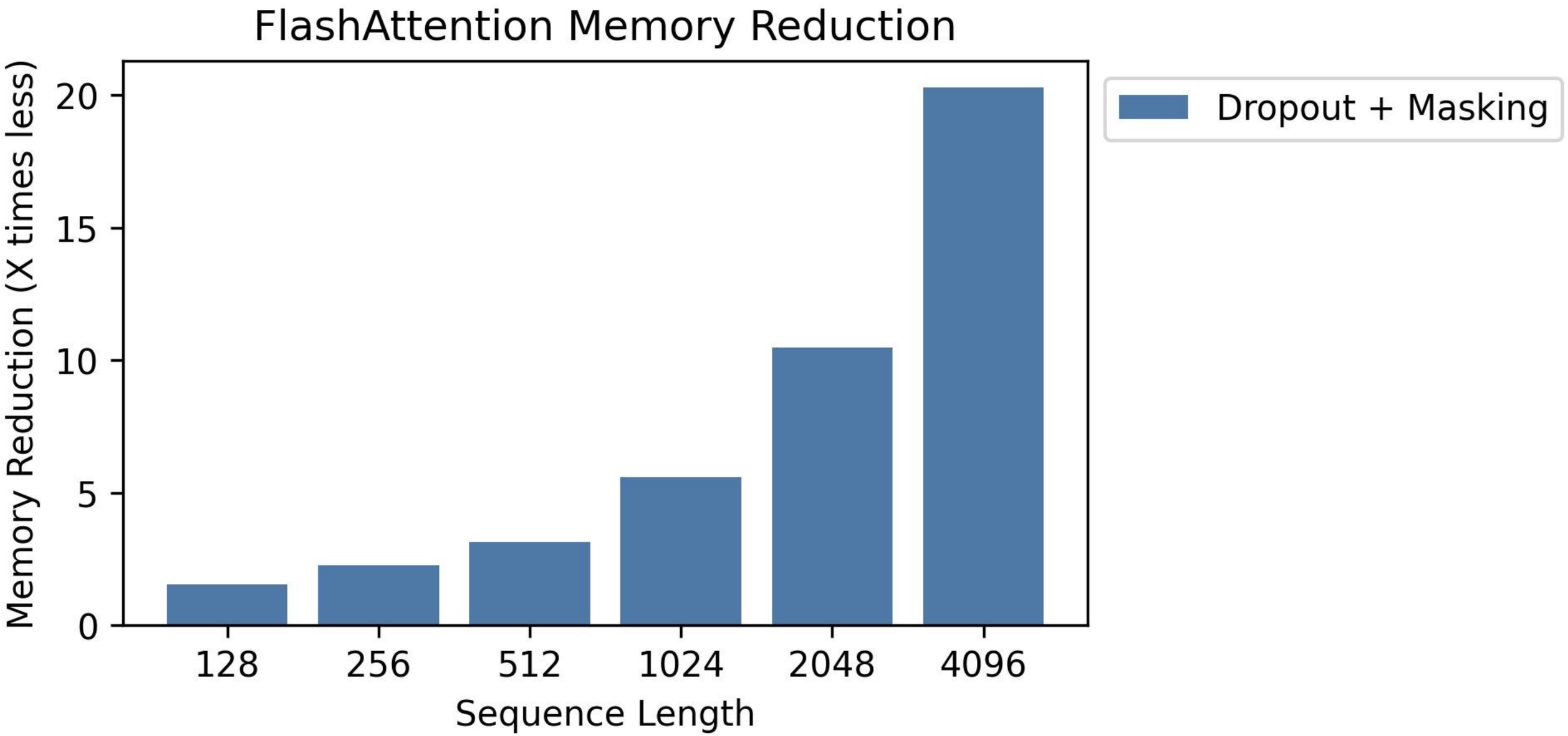
Memory footprint: up to 20x more efficient


FlashAttention Speedup, A100
Speedup (X times faster)
0
1
2
3
4
128
256
512
1024
2048
4096
Sequence Length
Dropout + Masking
Masking Only
No Masking, No Dropout


FlashAttention Speedup, GTX 3090
Speedup (X times faster)
0
1
2
3
4
128
256
512
1024
2048
Sequence Length
Dropout + Masking
Masking Only
No Masking, No Dropout


FlashAttention Speedup, T4
Speedup (X times faster)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
128
256
512
1024
2048
Sequence Length
Dropout + Masking
Masking Only
No Masking, No Dropout


FlashAttention Speedup, T4 (Forward Only)
Speedup (X times faster)
0
1
2
3
4
128
256
512
1024
2048
Sequence Length
Dropout + Masking
Masking Only
No Masking, No Dropout

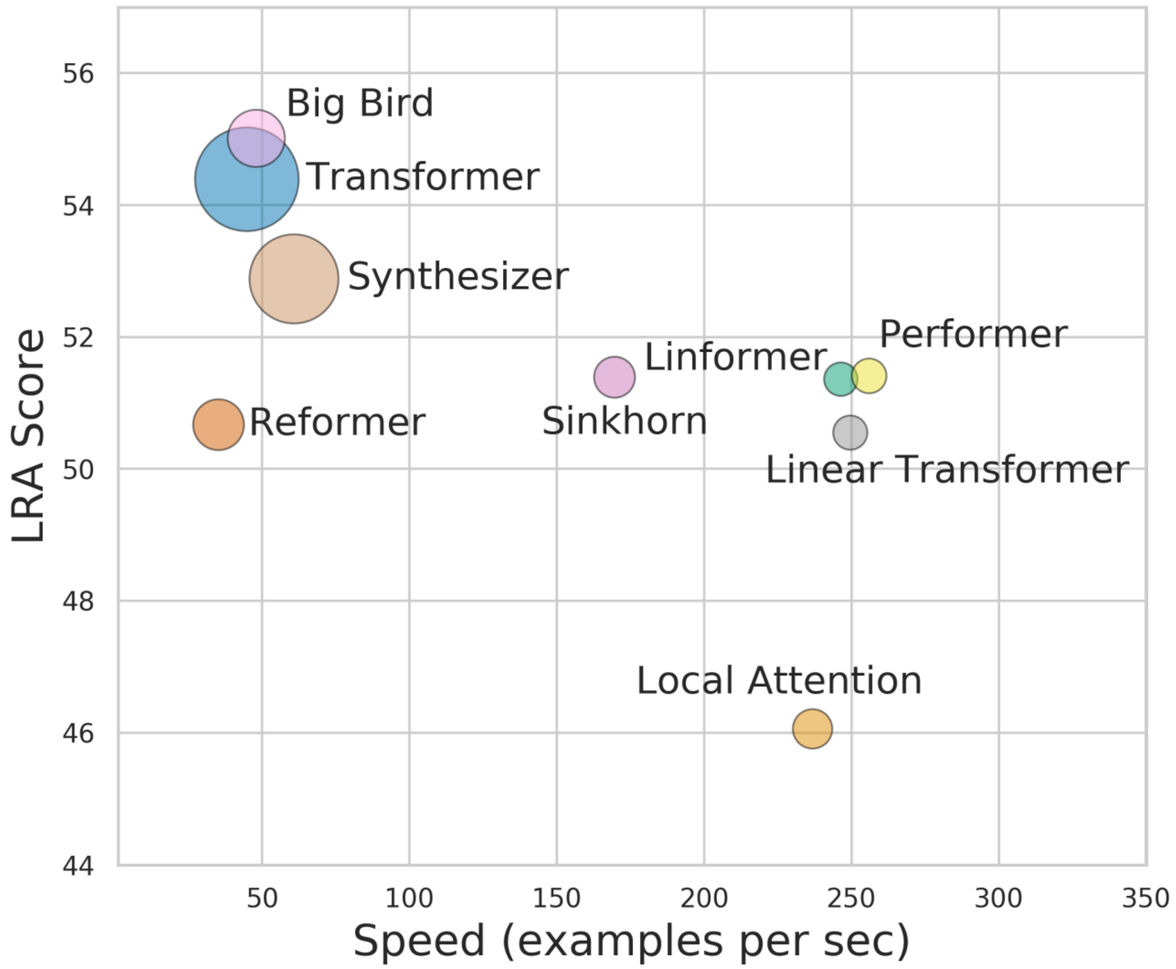

FlashAttention Memory Reduction
Memory Reduction (X times less)
0
5
10
15
20
128
256
512
1024
2048
4096
Sequence Length
Dropout + Masking

Background: Approximate Attention

Approximate attention: tradeoff **quality** for **speed**

Key ideas: reduce effective sequence length, sparsity, low rank approximation

Lack widespread adoption, reduce FLOPS but may not achieve wall-clock speedup



Is there a fast, memory-efficient, and exact attention algorithm?

Tay et al. *ICLR 2020*.