Hardware-aware Sparsity for Efficient and Accurate Training

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


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.

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

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

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

Focus of this talk
- Nice and deep theory
- (1) Hardware-efficient
- (2) Expressive
- (3) Tractable projection

Block Diagonal
Hardware-Efficiency

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

Problem 1: Not block-aligned
Problem 2: Hard to parallelize the product of many factors

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

[Benes, 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

1. Sparse E2E Training
2. Reverse
3. D2S Sparsification

Sparse E2E Training

Monarch matrices

Pretrained BERT/Random Init

Initial Model

Sparse

Monarch matrices

Dense

End Model

Fine-tuning
Sparse End-to-End Training

① Sparse E2E Training

Replace dense weight matrices (e.g., attention & FFN) with Monarch matrices for efficiency
Sparse-to-Dense Training (reverse sparsification)

Reverse Sparsification

800 hrs  200 hrs

2× faster same quality

Sparse to dense conversion...

2000 hrs

$t$ (gpu hours)
Validation: Sparse End-to-End Training

Benchmark tasks

<table>
<thead>
<tr>
<th>Model</th>
<th>WikiText 103(ppl)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT-2 Small</td>
<td>20.6</td>
<td>-</td>
</tr>
<tr>
<td>Monarch GPT-2-small</td>
<td>20.7</td>
<td>2.0 x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet (acc)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViT-Base</td>
<td>78.5</td>
<td>-</td>
</tr>
<tr>
<td>Monarch ViT-Base</td>
<td>78.7</td>
<td>1.8 x</td>
</tr>
</tbody>
</table>

Other applications: PDEs solving, MRI reconstruction

Speeds up training without losing performance🚀!
Validation: Sparse-to-Dense Training

<table>
<thead>
<tr>
<th>GPT2 implementation</th>
<th>OpenWebText perplexity</th>
<th>Speeup</th>
</tr>
</thead>
<tbody>
<tr>
<td>HuggingFace</td>
<td>18.1</td>
<td>-</td>
</tr>
<tr>
<td>Monarch</td>
<td>18.1</td>
<td>1.8x</td>
</tr>
</tbody>
</table>

80% sparse training, 20% dense training

1.8x speedup from using Monarch as an intermediate representation
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
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
\[ O = \text{Dropout}(\text{Softmax}(\text{Mask}(QK^T)))V \]

Naive implementation requires repeated R/W from slow GPU HBM
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.

**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.

<table>
<thead>
<tr>
<th>Attention</th>
<th>Standard</th>
<th>FLASHATTENTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPs</td>
<td>66.6</td>
<td>75.2</td>
</tr>
<tr>
<td>HBM R/W (GB)</td>
<td>40.3</td>
<td>4.4</td>
</tr>
<tr>
<td>Runtime (ms)</td>
<td>41.7</td>
<td>7.3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>BERT Implementation</th>
<th>Training time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huggingface [91]</td>
<td>55.6 ± 3.9</td>
</tr>
<tr>
<td>Nvidia MLPerf 1.1 [63]</td>
<td>20.0 ± 1.5</td>
</tr>
<tr>
<td>FLASHATTENTION (ours)</td>
<td>17.4 ± 1.4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Model</th>
<th>Path-X</th>
<th>Path-256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Linformer [81]</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Linear Attention [48]</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Performer [11]</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Local Attention [77]</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Reformer [49]</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SMYRF [18]</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>FlashAttention</td>
<td>61.4</td>
<td>×</td>
</tr>
<tr>
<td>Block-sparse FlashAttention</td>
<td>56.0</td>
<td>63.1</td>
</tr>
</tbody>
</table>

FlashAttention yields the first Transformer that can solve Path-X

Block-sparse FlashAttention enables Path-256
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
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

- **SRAM**: 19 TB/s (20 MB)
- **HBM**: 1.5 TB/s (40 GB)
- **DRAM**: 12.8 GB/s (>1 TB)

FlashAttention Speedup

- Dropout + Masking
- Masking Only
- No Masking, No Dropout

FlashAttention Memory Reduction

- Dropout + Masking

<table>
<thead>
<tr>
<th>Model</th>
<th>Path-X</th>
<th>Path-256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Linformer [81]</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Linear Attention [48]</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Performer [11]</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Local Attention [77]</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Reformer [49]</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SMYRF [18]</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>FlashAttention</td>
<td>61.4</td>
<td>x</td>
</tr>
<tr>
<td>Block-sparse FlashAttention</td>
<td>56.0</td>
<td>63.1</td>
</tr>
</tbody>
</table>
Benchmarks and Tradeoffs: Attention Module

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

- **FlashAttention**: faster than baselines for short sequences
- **Block-sparse FlashAttention**: faster everywhere
- **Memory footprint**: up to 20x more efficient

**Details**
- Batch size: 16
- Heads: 8
- Head dim: 64
- Fwd + Bwd Pass

(more in paper!)
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?**