FlashAttention

Zhitong Guo, Xinran Wan, Haoze He, Alicia Sui

Overview

1. Motivation
2. Challenges and Related Work
3. FlashAttention
   a. Tiling
   b. Recomputation
   c. Block-sparse FlashAttention
4. Evaluation
5. Future Directions

Motivation: Modeling Longer Sequences

- Large context needed to understand books, plays, etc. (NLP)
- Higher resolution necessary to better, more robust insight. (CV)
- Other data format: time series, audio, video, etc. modeled with sequences of millions of steps.

Challenge: Transformers struggle with long sequences…

… due to quadratic time & memory complexity in sequence length.

Related Works

Question: Better attention ⇒ Better Transformer models?

Solution: Approximate attention methods

- Reduce complexity to linear or near-linear wrt. sequence length
- Retain model performance as much as possible

Sparse Approximation

Low-rank Approximation

Combination of The Two

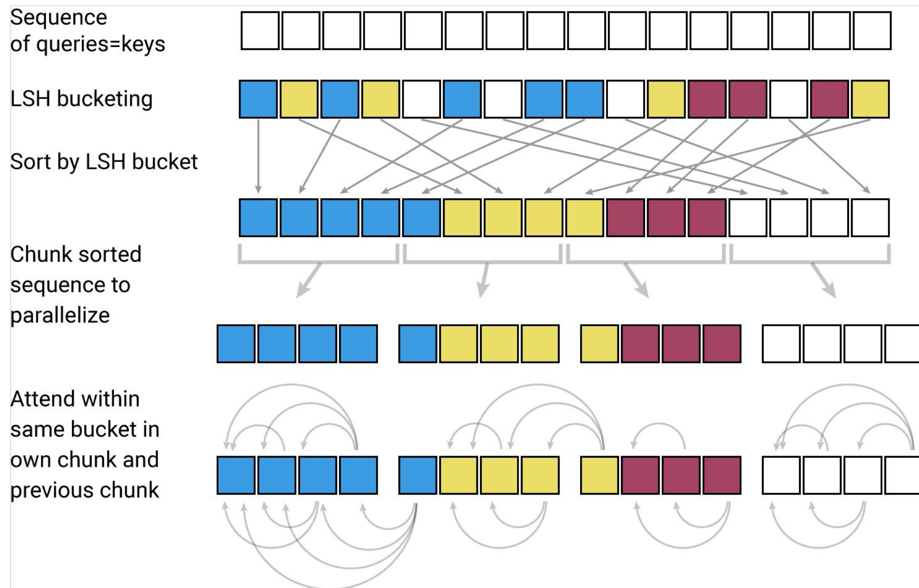
Related Works: Sparse Approximation

Reformer: The Efficient Transformer[1]

1. Locality Sensitive Hashing (LSH): reduce the complexity of attending over long sequences
2. Reversible Residual Layers: use available memory more efficiently

[1] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *The International Conference on Machine Learning (ICML)*, 2020.

Related Works: Sparse Approximation



Steps:

1. A hash function: match similar vectors together, instead of searching through all possible pairs of vectors.

2. rearrange sequence and divide it into segments for parallel processing

3. Apply attention within those much shorter chunks and their adjoining neighbors to cover the overflow

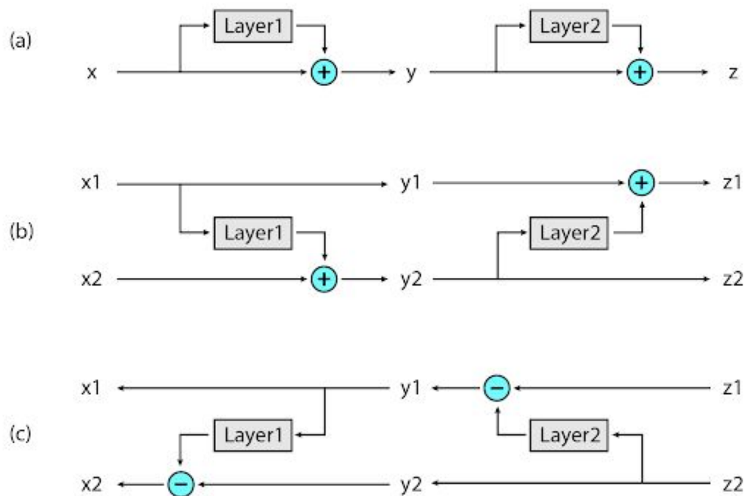
Related Works: Sparse Approximation

Reformer: The Efficient Transformer[1]

1. Locality Sensitive Hashing (LSH): reduce the complexity of attending over long sequences
2. Reversible Residual Layers: use available memory more efficiently

[1] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *The International Conference on Machine Learning (ICML)*, 2020.

Related Works: Sparse Approximation



Goal: recompute the input of each layer on-demand during back-propagation, rather than storing it in memory.

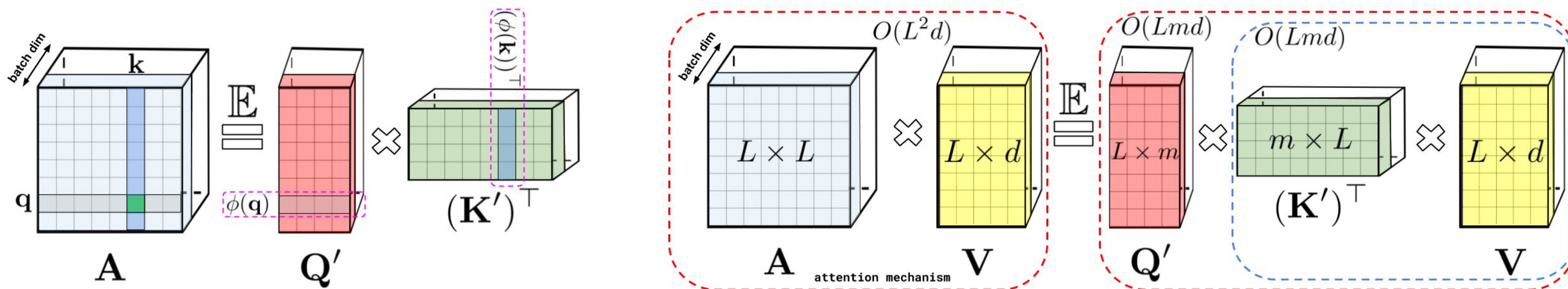
Reversible layers:

- Activations from the last layer are used to recover activations from any intermediate layer
    - In a typical residual network, each layer in the stack keeps adding to vectors that pass through the network.
- Reversible layers, instead, have two sets of activations for each layer.
    - One follows the standard procedure and is progressively updated from one layer to the next
    - The other captures only the changes to the first. Thus, to run the network in reverse, one simply subtracts the activations applied at each layer.

Related Works: Low-rank Approximation

Rethinking Attention with **Performers** [1]

- Framework implementation algorithm: Fast Attention via Matrix Associativity (FAVOR+)
- *Any* attention matrix can be *effectively* approximated in downstream Transformer-applications using random features.



[1] Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. In *International Conference on Learning Representations (ICLR)*, 2020.

Related Works ← Problems

Didn't display wall-clock speedup. And they want to trade performance for speedup.

Why?

They only focused on FLOP reduction, and ignored IO (ie. memory access) overheads.

Let's first look at the evolving process before *FlashAttention*…

Evolving Process

- For each attention head, we need to store the attention matrix:
- $\left[softmax\left(v_i^T Q_r K_r^T v_j + p_{i,j}^r\right)_{j\in[n]}\right]_{i\in[n]}$
- Let's just consider one row:
  - $softmax\left(v_i^T Q_r K_r^T v_j + p_{i,j}^r\right)_{j\in[n]}$
- Key idea of Flash-Attention:
  - We store for every r and j, this takes memory .
  - We do not store the full softmax matrix, we will "compute them on the fly" to save memory.
  - Reduce memory usage from n to d/m

Evolving Process

Main Memory Usage:

- For each attention head, we need to store the $n \times n$ attention matrix:
- $\left[softmax\left(v_i^T Q_r K_r^T v_j + p_{i,j}^r \right)_{j \in [n]} \right]_{i \in [n]}$
- Let's just consider one row:
  - $softmax\left(v_i^T Q_r K_r^T v_j + p_{i,j}^r \right)_{j \in [n]}$
- Key idea of Flash-Attention:
  - We store $K_r^T v_j$, $Q_r^T v_j$ for every r and j, this takes memory $d \times n$.
  - We do not store the full softmax matrix, we will "compute them on the fly" to save memory.
  - Reduce memory usage from n to d/m

Evolving Process

- Consider $O = \sum_{i \in [n]} y_i \times softmax(x)_i$
- Where for each x_i, y_i, we need computation time d/m to retrieve it.
- Stupid-Attention computation:
  - For i in range(n):
    - Compute $norm_factor = norm_factor + \exp(x_i)$.
      if x_i is really large-> overflow! Due to floating point accuracy,
    - Compute $O = O + y_i \exp(x_i)$
  - Return $O/norm_factor$
- This only requires memory $O(M)$, where $M = \frac{d}{m}$ is the dimension of y_i
- But this have floating point accuracy issue

Evolving Process: From Stupid Attention (v2) to Flash Attention

- Why is Stupid Attention Stupid?
- Floating Point accuracy. We can not compute $\sum \exp(x_i)$ accurately! No such accuracy.
- Stupid Attention V2:
  - Go through i, compute the max of x_i as $m(x)$
  - For i in range(n):
    - Compute $norm_factor = norm_factor + \exp(x_i - m(x))$.
    - Compute $O = O + y_i \exp(x_i - m(x))$
    - Solve floating point issue-> one term in summation that is one
  - Return $O/norm_factor$
  - This extract x_i twice, one from max, one from else->computation cost
- But then we need to compute x_i twice, unless we store it in the memory…
- But we don't want to put x_i in memory
- But we solve the floating point issue

Evolving Process: From Stupid Attention (v3) to Flash Attention

- Stupid Attention V3 is an upgrade of stupid attention v2, where we only compute x_i once and maintain the correct floating-point accuracy. Maintain running max instead of actual max
- Each time subtract running max, not actual max
- For i in range(n):
  - Compute $m_{new}(x) = \max(\, m(x),\ x_i)$
  - Compute $norm = \exp\big(m(x) - m_{new}(x)\big) norm + \exp\big(x_i\ - m_{new}(x)\big)$.
  - Compute $O\ = \exp(m(x) - m_{new}(x))O\ + y_i \exp(x_i\ - m_{new}(x))$
  - Update $m(x)\ =\ m_{new}(x)$
- Output O/norm.
- Same computation cost as naïve version
- Memory usage-> d/m
- But still use for loop! Not utilizing fast matrix multiplication

Evolving Process: Flash Attention

- Flash attention is a little bit more involved than the previous slides.
- It divides the computation in chunks of R
- Extract $x(m)$ in blocks instead in for loops! Speed really fast!!
- For i in range($n//R$): -> for loop go through blocks
    - Compute self attention in naïve way in each block
    - Compute the softmax for x[iR:iR +R] using the fastest way, which uses memory R. Then compute
        - $O_i = \sum_{i \in [iR, iR+R)} y_j \times softmax(x[iR : iR + R])_j$
        - (only store this O_i in SRAM).
    - Store the max of x[j] for j in [iR, iR + R) in memory as m[I].
    - Store the normalization factor of the softmax (after subtracting the max) of $x[iR : iR + R]$ in memory as norm[i].
    - Update $m_{new}(x) = \max(m(x),\ m[i])$
    - Update $O\ = O\exp\big(m(x) - m_{new}(x)\big) + \exp\big(m[i] -\ m_{new}(x)\big)O_i \times norm[i]$
    - Update $norm = \exp\big(m(x) - m_{new}(x)\big)norm + \mathrm{norm}\big[\mathrm{i}\big] \times \exp\big(m[i]\ - m_{new}(x)\big)$.
    - Update $m(x)\ =\ m_{new}(x)$

Authors’ Hypothesis

IO-aware attention algorithm can bridge the gap.

- Carefully account for reads/writes to various levels of fast/slow memory.
- Common python DL interfaces (Pytorch, Tensorflow, etc.) don’t allow fine-grained memory access control.

Goal: Avoid reading and writing the attention matrix to and from HBM.

- Compute the softmax reduction without access to the whole input
- Not store the large intermediate attention matrix for the backward pass.

FlashAttention: a new attention algorithm, computes exact attention, access memory far less.

Both performance and computation increase in comparison to related works.

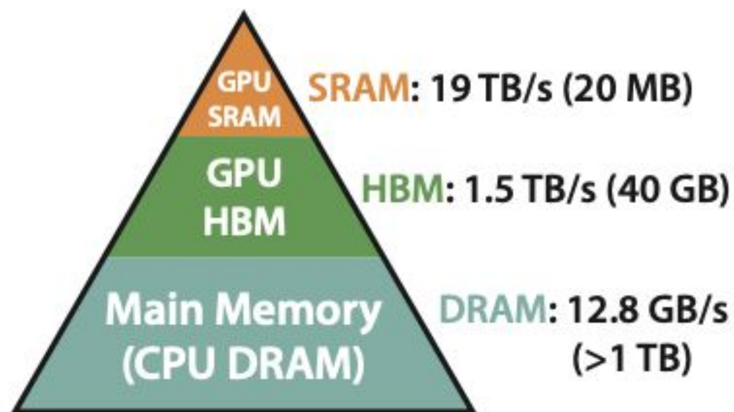
FlashAttention: Reduce HBM R/W via Computing by Blocks

Goal 1: Compute softmax without access to full input
Goal 2: Compute backward without the large attention matrix from forward

Solution 1: Tiling. Restructure the algorithm to load block by block from HBM to SRAM to compute attention.

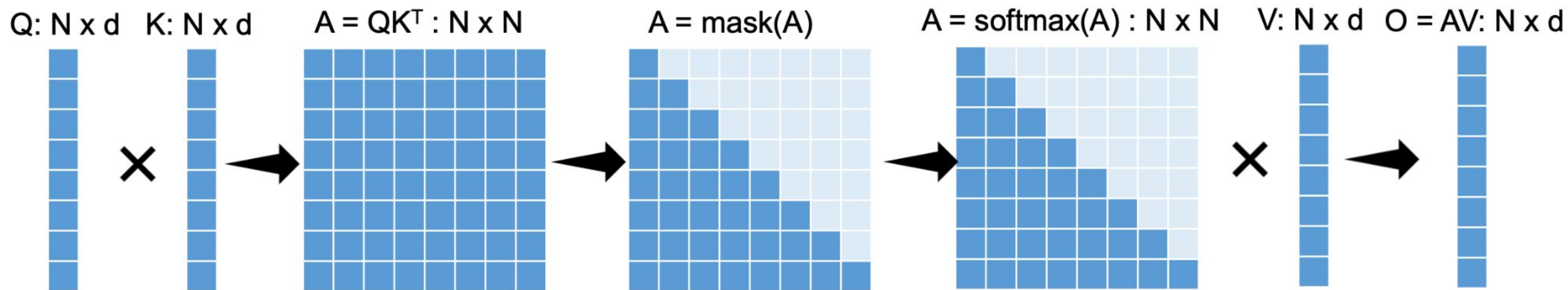
Solution 2: Recomputation. Without storing the attention matrix in forward, recompute in backward.

HBM vs SRAM: Memory Hierarchy



Memory Hierarchy with
Bandwidth & Memory Size

Recall: Attention $\mathbf{O} = \mathrm{Dropout}(\mathrm{Softmax}(\mathrm{Mask}(\mathbf{QK}^\top)))\mathbf{V}$



$$m(x) := \max_i \quad x_i, \quad f(x) := \left[e^{x_1 - m(x)} \quad \ldots \quad e^{x_B - m(x)}\right], \quad \ell(x) := \sum_i f(x)_i, \quad \mathrm{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

Tiling: dynamically compute softmax

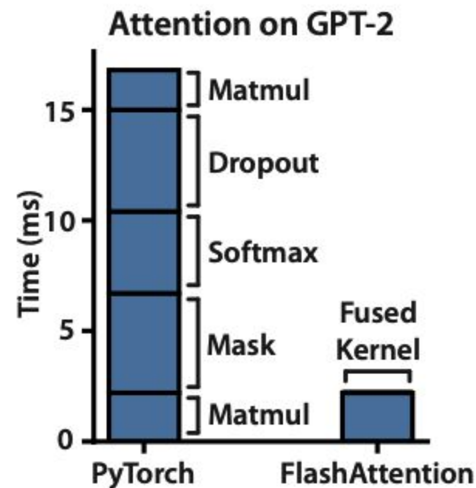
$$m(x) = m([x^{(1)}\ x^{(2)}]) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = \left[e^{m(x^{(1)})-m(x)} f(x^{(1)}) \quad e^{m(x^{(2)})-m(x)} f(x^{(2)})\right]$$

$$\ell(x) = \ell([x^{(1)}\ x^{(2)}]) = e^{m(x^{(1)})-m(x)} \ell(x^{(1)}) + e^{m(x^{(2)})-m(x)} \ell(x^{(2)}), \quad \mathrm{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

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

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

*alpha and beta are scaling factor of the denominator of softmax



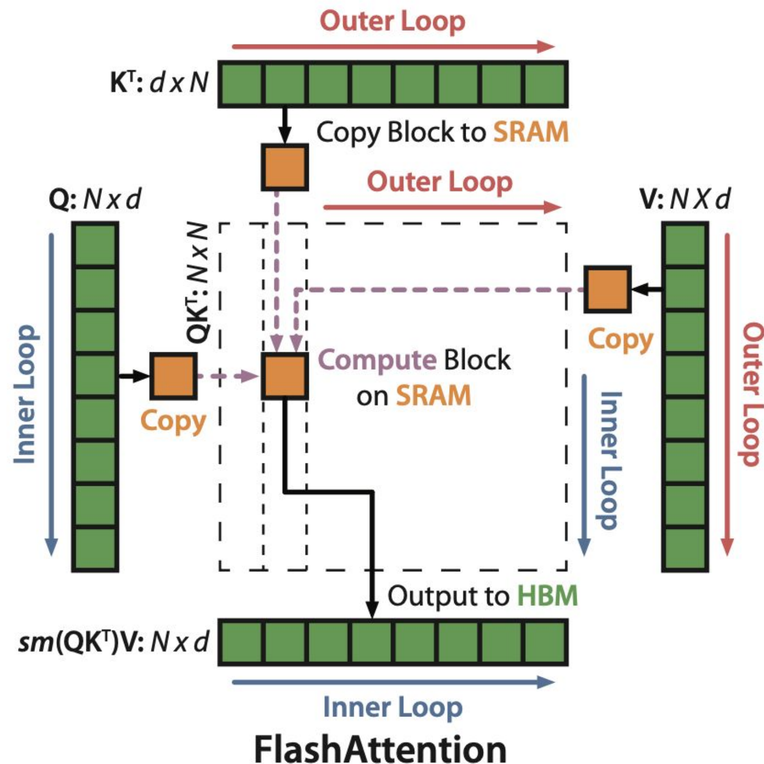
Tiling

Decompose Large Softmax into smaller ones by Scaling

1. Load inputs by blocks from global to shared memory (HBM to SRAM)
2. On chip, compute attention output wrt the block
3. Update output in device memory (HBM) by scaling

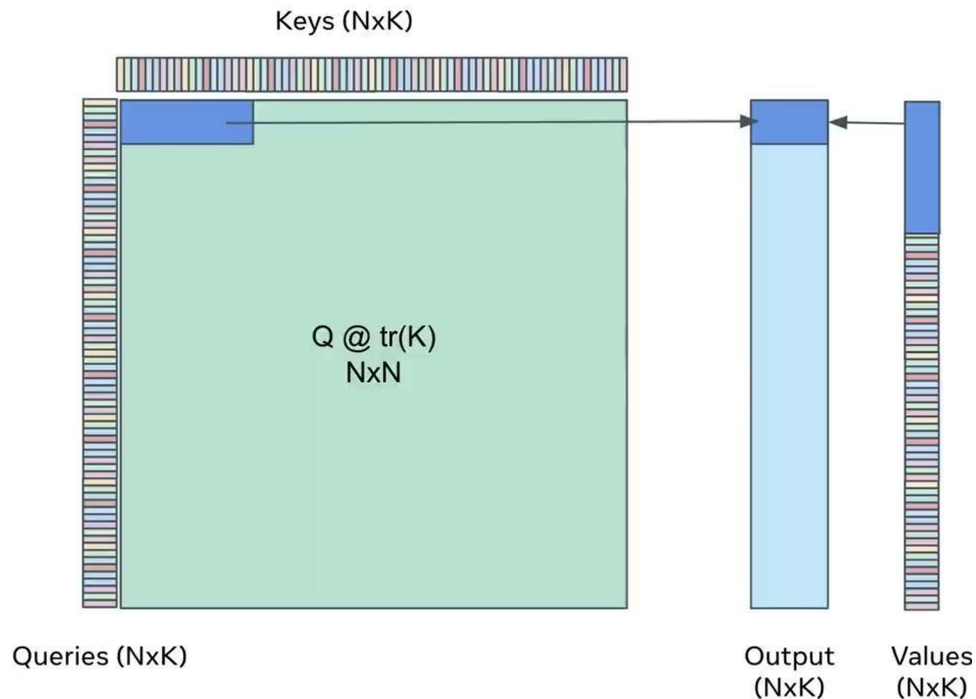
$$\mathrm{softmax}([A_1, A_2]) = [\alpha \times \mathrm{softmax}(A_1), \beta \times \mathrm{softmax}(A_2)]$$

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



Tiling

1. Load inputs by blocks from global to shared memory (HBM to SRAM)
2. On chip, compute attention output wrt the block
3. Update output in device memory (HBM) by scaling



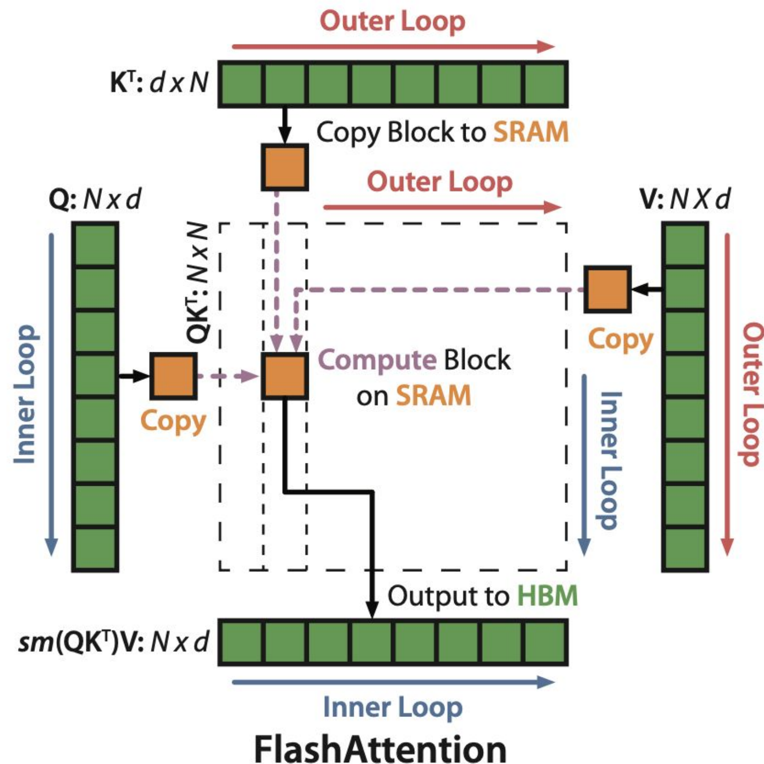
Cr: Francisco Massa

Recomputation in backward pass

By storing softmax normalization factors from forward (size N),
quickly recompute attention in the backward from inputs in shared memory (SRAM)

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

💡 Speed up backward pass with increased FLOPs 💡

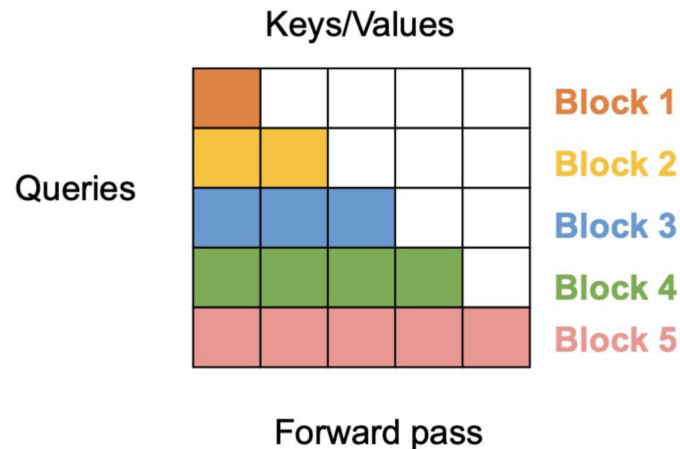


Threadblock-level Parallelism

Partition of FlashAttention across thread blocks:

Eg. A100 with 108 SMMs

- Step 1: assign different heads to different thread blocks (16-64 heads)
- Step 2: assign different queries to different thread blocks



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.

1: Set block sizes $B_c = \left\lceil \frac{M}{4d} \right\rceil, B_r = \min\left(\left\lceil \frac{M}{4d} \right\rceil, d\right)$.

2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

3: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.

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 $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

10: On chip, compute $\tilde{m}_{ij} = \mathrm{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

11: On chip, compute $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.

12: Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1} (\mathrm{diag}(\ell_i) e^{m_i - m_i^{\mathrm{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.

13: Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}$, $m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.

14: **end for**

15: **end for**

16: Return $\mathbf{O}$.

Block-Sparse FlashAttention

- The algorithm is almost identical to Algorithm 1 except we skip zero blocks
- Given a predefined block sparsity mask $\mathbf{M} \in \{0,1\}^{N/B_r \times N/B_c}$, we can adapt Algorithm 1 to only compute the nonzero blocks of the attention matrix.

Given inputs $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ and a mask matrix $\tilde{\mathbf{M}} \in \{0,1\}^{N \times N}$, we want to compute:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S} \odot \mathbb{1}_{\tilde{\mathbf{M}}}) \in \mathbb{R}^{N \times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N \times d},$$

where $(\mathbf{S} \odot \mathbb{1}_{\tilde{\mathbf{M}}})_{kl} = \mathbf{S}_{kl}$ if $\tilde{\mathbf{M}}_{kl} = 1$ and $-\infty$ if $\mathbf{M}_{kl} = 0$. We require $\tilde{\mathbf{M}}$ to have block form: for some block sizes B_r, B_c, for all k, l, $\tilde{\mathbf{M}}_{k,l} = \mathbf{M}_{ij}$ with $i = \lfloor k/B_r \rfloor$, $j = \lfloor l/B_c \rfloor$ for some $\mathbf{M} \in \{0,1\}^{N/B_r \times N/B_c}$.

Algorithm 5 Block-Sparse FlashAttention Forward Pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M, softmax scaling constant $\tau \in \mathbb{R}$, masking function MASK, dropout probability p_{drop}, block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$, block sparsity mask $M \in \{0,1\}^{N/B_r \times N/B_c}$..

1: Initialize the pseudo-random number generator state $\mathcal{R}$ and save to HBM.
2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
3: Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.
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: **if** $M_{ij} \neq 0$ **then**
9: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
10: On chip, compute $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
11: On chip, compute $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$.
12: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}^{\text{masked}}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij}^{\text{masked}} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
13: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
14: On chip, compute $\tilde{\mathbf{P}}_{ij}^{\text{dropped}} = \text{dropout}(\tilde{\mathbf{P}}_{ij}, p_{\text{drop}})$.
15: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij}^{\text{dropped}} \mathbf{V}_j)$ to HBM.
16: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
17: **end if**
18: **end for**
19: **end for**
20: Return $\mathbf{O}, \ell, m, \mathcal{R}$.

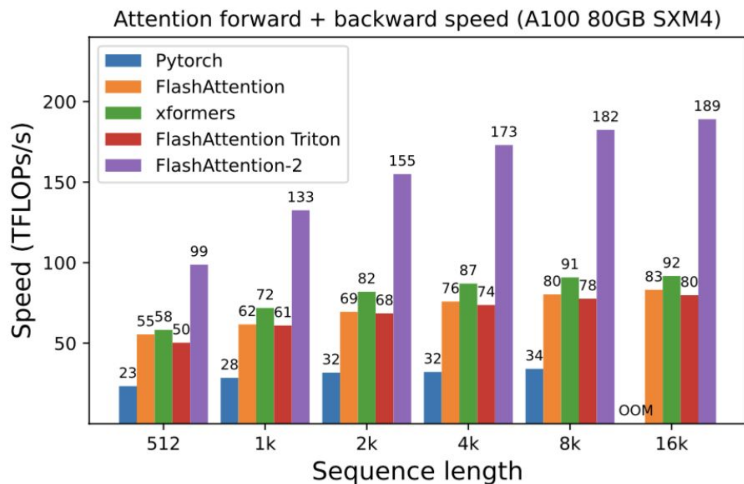
Further Improvements for Block-Sparse FlashAttention

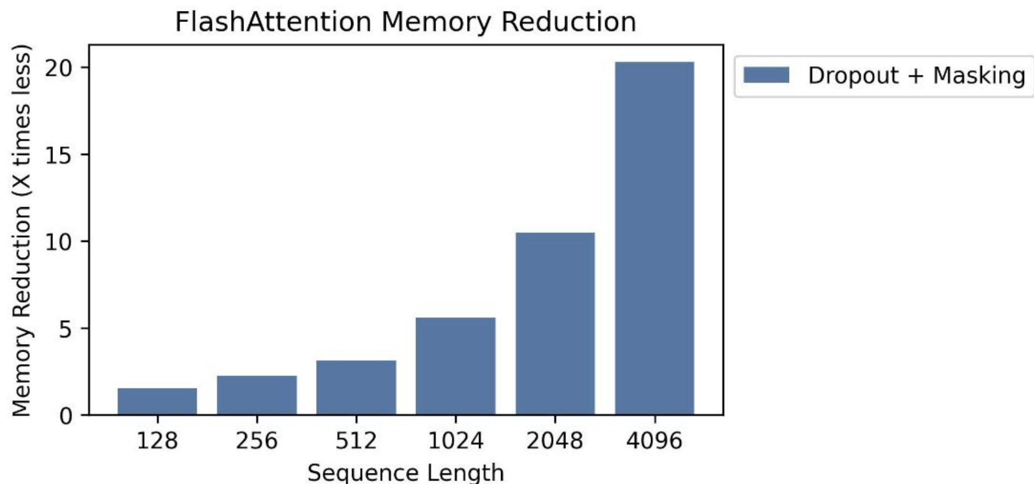
- 2-4x faster than FlashAttention
- Scale up to sequence length of 64K
- Better IO complexity than FlashAttention by a factor proportional to sparsity ratio

Pros

Speedup: faster end-to-end training of transformers

Memory savings: Memory linear in sequence length, longer sequences, higher-quality transformers





Evaluation and Experiment

End to End training results (***Speedup***):

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

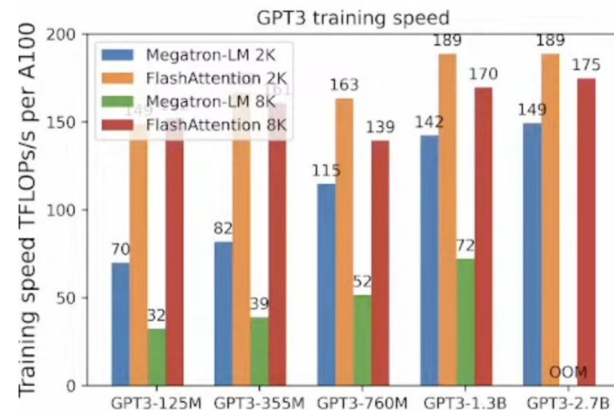
MLPerf: (high optimized) standard benchmark for training speed.

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

Evaluation and Experiment

End to End training results (***Speedup and Performance***):

Model implementations	OpenWebText (ppl)	Training time (speedup)
GPT-2 small - Huggingface [87]	18.2	9.5 days (1.0×)
GPT-2 small - Megatron-LM [77]	18.2	4.7 days (2.0×)
GPT-2 small - FLASHATTENTION	18.2	**2.7 days (3.5×)**
GPT-2 medium - Huggingface [87]	14.2	21.0 days (1.0×)
GPT-2 medium - Megatron-LM [77]	14.3	11.5 days (1.8×)
GPT-2 medium - FLASHATTENTION	14.3	**6.9 days (3.0×)**



FlashAttention on GPT-3: GPT-2 small and medium using FlashAttention achieve up to 3× speed up compared to Huggingface implementation and up to 1.7× compared to Megatron-LM. Training time reported on 8×A100s GPUs. (Faster Training, Longer Context)

Evaluation and Experiment

End to End training results (***Speedup and Performance***):

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×

Long-Range Arena Benchmark: Compare vanilla Transformer (with either standard implementation or FlashAttention. Each task has a different sequence length varying between 1024 and 4096.

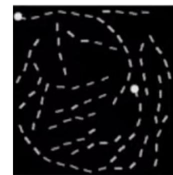
Evaluation and Experiment

End to End training results (***Speedup and Performance***):

Model implementations	Context length	OpenWebText (ppl)	Training time (speedup)
GPT-2 small - Megatron-LM	1k	18.2	4.7 days (1.0×)
GPT-2 small - FlashAttention	1k	18.2	**2.7 days (1.7×)**
GPT-2 small - FlashAttention	2k	17.6	3.0 days (1.6×)
GPT-2 small - FlashAttention	4k	**17.5**	3.6 days (1.3×)

Long Context: The runtime and memory-efficiency of FlashAttention increase the context length of GPT-2 by 4× while still running faster than the optimized implementation from Megatron-LM. (GPT-2 with FlashAttention and context length 4K is still 30% faster than GPT-2 from Megatron with context length 1K, while achieving 0.7 better perplexity.)

Evaluation and Experiment



End to End training results (***Performance***):

	512	1024	2048	4096	8192	16384
MIMIC-III [47]	52.8	50.7	51.7	54.6	56.4	**57.1**
ECtHR [6]	72.2	74.3	77.1	78.6	**80.7**	79.2

Model	Path-X	Path-256
Transformer	✗	✗
Linformer [84]	✗	✗
Linear Attention [50]	✗	✗
Performer [12]	✗	✗
Local Attention [80]	✗	✗
Reformer [51]	✗	✗
SMYRF [19]	✗	✗
FlashAttention	**61.4**	✗
Block-sparse FlashAttention	56.0	**63.1**

Why long text?: Long Document performance (micro F1) at different sequence lengths using FlashAttention.

Path-X, Path-256: Transformer model that can achieve non-random performance on Path-X and Path-256. Other models can only do random guess. Path-X tell whether two dots are connected, using pixels as inputs. Require sequence length 16K/ 64K for Path-X/ Path-256.

Future Direction

1. **Multi-GPU IO-Aware Methods:** While the current IO-aware attention implementation is optimal for single GPU usage, attention computation can be parallelized across multiple GPUs. However, this introduces complexities in IO analysis, particularly in accounting for data transfer between GPUs. There's a potential for future research to explore IO-aware methodologies in multi-GPU.

2. **Extension of IO-Aware Approach:** Beyond attention, the IO-aware methodology can be applied to other modules in deep learning. While attention is the most memory-intensive computation in Transformers, every layer interacts with GPU HBM.