FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

FlashAttention, is an IO-aware exact attention algorithm that uses tiling to reduce the number of memory reads/writes between GPU high bandwidth memory (HBM) and GPU on-chip SRAM. The paper also shows that it requires fewer HBM accesses than standard attention and is optimal for a range of SRAM sizes.

Hardware Performance

Performance characteristics. Depending on the balance of computation and memory accesses, operations can be classified as either compute-bound or memory-bound. This is commonly measured by the
arithmetic intensity, which is the number of arithmetic operations per byte of memory access.

1. **Compute-bound:** the time taken by the operation is determined by how many arithmetic operations there are, while time accessing HBM is much smaller. Typical examples are matrix multiply with large inner dimension, and convolution with large number of channels.
2. **Memory-bound:** the time taken by the operation is determined by the number of memory accesses, while time spent in computation is much smaller. Examples include most other operations: elementwise (e.g., activation, dropout), and reduction (e.g., sum, softmax, batch norm, layer norm).

Kernel fusion : The most common approach to accelerate memory-bound operations is kernel fusion: if there are multiple operations applied to the same input, the input can be loaded once from HBM, instead of multiple times for each operation. Compilers can automatically fuse many elementwise operations. However, in the context of model training, the intermediate values still need to be written to HBM to save for the backward pass, reducing the effectiveness of naive kernel fusion.

Standard Attention Implementation

Given input sequences $Q, K, V \in \mathbb{R}^{N \times d}$ where N is the sequence length and d is the head dimension, we want to compute the attention output $O \in \mathbb{R}^{N \times d}$:

$$S = QK^T \in \mathbb{R}^{N \times N}, \quad P = \mathrm{softmax}(S) \in \mathbb{R}^{N \times N}, \quad O = PV \in \mathbb{R}^{N \times d}, \qquad \text{-> eqn (1)}$$

where softmax is applied row-wise.
Standard attention implementations materialize the matrices S and P to HBM, which takes $O(N^2)$ memory. Often $N \gg d$ (e.g., for GPT2, $N = 1024$ and $d = 64$). We describe the standard attention implementation in Algorithm 0. As some or most of the operations are memory-bound (e.g., softmax), the large number of memory accesses translates to slow wall-clock time. This problem is exacerbated by other elementwise operations applied to the attention matrix, such as masking applied to S or dropout applied to P. As a result, there have been many attempts to fuse several elementwise operations, such as fusing masking with softmax.

Algorithm 0 Standard Attention Implementation

Require: Matrices $Q, K, V \in \mathbb{R}^{N \times d}$ in HBM.
1: Load Q, K by blocks from HBM, compute $S = QK^T$, write S to HBM.
2: Read S from HBM, compute $P = \mathrm{softmax}(S)$, write P to HBM.
3: Load P and V by blocks from HBM, compute $O = PV$, write O to HBM. 4: Return O.

Flash Attention: Algorithm, Analysis, and Extensions

An Efficient Attention Algorithm With Tiling and Re-computation

Given the inputs $Q, K, V \in \mathbb{R}^{N \times d}$ in HBM, we aim to compute the attention output $O \in \mathbb{R}^{N \times d}$ and write it to HBM. Our goal is to reduce the amount of HBM accesses (to sub-quadratic in N).

We apply two established techniques (tiling, re-computation) to overcome the technical challenge of computing exact attention in sub-quadratic HBM accesses. We describe this in Algorithm 1. The main idea is that we split the inputs Q, K, V into blocks, load them from slow HBM to fast SRAM, then compute the attention output with respect to those blocks. By scaling the output of each block by the right normalization factor before adding them up, we get the correct result at the end.

Tiling. We compute attention by blocks. Softmax couples columns of K, so we decompose the large softmax with scaling. For numerical stability, the softmax of vector $x \in R^B$ is computed as:

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

For vectors $x^{(1)}, x^{(2)} \in \mathbb{R}^B$, we can decompose the softmax of the concatenated $x = \left[x^{(1)}\ x^{(2)}\right] \in \mathbb{R}^{2B}$ as:

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

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

Online Softmax

The generic formula for softmax is

$$\text{Softmax}(\{x_1, x_2, \dots x_n\}) = \left\{\frac{e^{xi}}{\sum_{j=1}^{N} e^{xj}}\right\}_{i=1}^{N}$$

Detail to add…..!

Algorithm : Softmax Original

m_i = -inf
for i = 1 to N do
$m_i = \max(m_i, x_i)$
$S_e = 0$
for i <- 1 to N do
$S_e = S_e + e^{(xi - mi)}$
for i =1 to N do
$x_i = e^{(xi-mi)} / S_e$

Algorithm : Online Softmax

m_{i-i} = -inf
$S_e = 0$
for i = 1 to N do
$m_i = \max(m_{i-1}, x_i)$
$S_e = S_e * e^{(mi-1 - mi)} + e^{(xi - mi)}$
for i =1 to N do
$xi = e(x_i - m_i)/S_e$

The algorithm proposes the decomposition of softmax for efficient tiling of Attention (eqn 1). The main challenge in making attention memory-efficient is the softmax that couples the columns of K (and columns of V). Our approach is to compute the softmax normalization constant separately to decouple the columns. This technique [2] has been used in the literature [3] to show that attention computation does not need quadratic *extra* memory (though the number of HBM accesses is still quadratic, resulting in slow run-time). This enables tiling the complete tiling of attention $A = (\text{Softmax}(Q*K^T)*V)/S$. This is similar to Online Softmax. If we keep track of some extra statistics ($m(x), \ell(x)$), we can compute softmax one block at a time.[2]

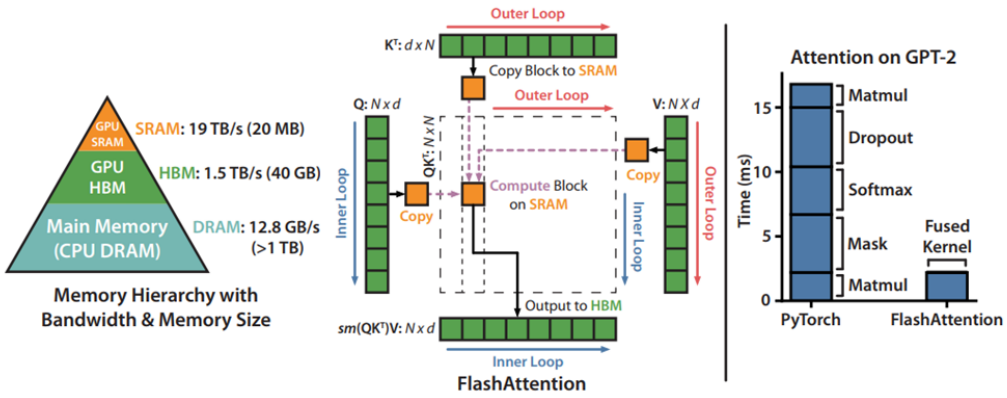

Figure 1: **Left:** FLASHATTENTION uses tiling to prevent materialization of the large $N \times N$ attention matrix (dotted box) on (relatively) slow GPU HBM. In the outer loop (red arrows), FLASHATTENTION loops through blocks of the $\mathbf{K}$ and $\mathbf{V}$ matrices and loads them to fast on-chip SRAM. In each block, FLASHATTENTION loops over blocks of $\mathbf{Q}$ matrix (blue arrows), loading them to SRAM, and writing the output of the attention computation back to HBM. **Right:** Speedup over the PyTorch implementation of attention on GPT-2. FLASHATTENTION does not read and write the large $N \times N$ attention matrix to HBM, resulting in an 7.6× speedup on the attention computation.

The generic algorithm with tiling and online softmax is given below (Algoritm1). It splits the inputs Q, K, V into blocks (Algorithm 1 line 3), compute the softmax values along with extra statistics (Algorithm 1 line 10), and combine the results (Algorithm 1 line 12). **Recomputation**. The goal is to not store $\mathcal{O}(N^2)$ intermediate values for the backward pass. The backward pass typically requires the matrices $S, P \in \mathbb{R}^{N \times N}$ to compute the gradients with respect to Q, K, V. However, by storing the output O and the softmax normalization statistics (m, ℓ), we can recompute the attention matrix S and P easily in the backward pass from blocks of Q, K, V in SRAM. This can be seen as a form of selective gradient checkpointing [10, 34]. While gradient checkpointing has been suggested to reduce the maximum amount of memory required [66], all implementations (that we know off) must trade speed for memory. In contrast, even with more FLOPs, our recomputation speeds up the backward pass due to reduced HBM accesses (Fig. 2). The full backward pass description is in Appendix B.

Implementation details: Kernel fusion. Tiling enables us to implement our algorithm in one CUDA kernel, loading input from HBM, performing all the computation steps (matrix multiply, softmax, optionally masking and dropout, matrix multiply), then write the result back to HBM (masking and dropout in Appendix B). This avoids repeatedly reading and writing of inputs and outputs from and to HBM.

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.

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

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.

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

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

for $1 \le j \le T_c$ **do**

  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

  for $1 \le i \le T_r$ **do**

    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.

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

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

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

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

    Write $\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$ to HBM.

  end for

end for

Return $\mathbf{O}$.

Theorem 1. *Algorithm 1 returns* $O = \text{softmax}(QK^T)V$ *with* $\mathcal{O}(N^2 d)$ *FLOPs and requires* $\mathcal{O}(N)$ *additional memory beyond inputs and output.*

A python torch Implementation without masking and dropout.

```
import torch
import math

# attention per sequence
# q: shape [context_length, qkv_heads, head_size]
# k: shape [context_length, qkv_heads, head_size]
# v: shape [context_length, qkv_heads, head_size]
# output: shape [q_heads, head_size]

def flash_attn(q, k, v, scale):
    O = torch.zeros((q.shape[0], v.shape[1]))
    for i in range(q.shape[0]):
      mi = float('-inf')
      mi1 = 0
      di1 = 0
      Oi = torch.zeros(v.shape[1])
      for j in range(k.shape[0]):
        xi = torch.dot(q[i,:], k[j,:])/scale
        mi = max(mi, xi)
        e1 = math.exp(mi1 - mi)
        e2 = math.exp(xi - mi)
        di = di1 * e1 + e2
        Oi = Oi*(di1*e1/di) + (e2/di) * v[j,:]
        mi1 = mi
        di1 = di
      O[i,:] = Oi;
    return O;
```

More code and details to add….

Analysis: IO Complexity of FlashAttention

The IO complexity analysis of FlashAttention, shows significant reduction in HBM accesses compared to standard attention. They also provide a lower bound, proving that no exact attention algorithm can asymptotically improve on HBM accesses over all SRAM sizes.

Theorem 2. *Let N be the sequence length, d be the head dimension, and M be size of SRAM with $d \le M \le Nd$. 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.*

For typical values of d (64-128) and M (around 100KB), d^2 is many times smaller than M, and thus FlashAttention requires many times fewer HBM accesses than standard implementation. This leads to both faster execution and lower memory footprint, which we validate in Section 4.3. The main idea of the proof is that given the SRAM size of M, we can load blocks of K, V of size $\Theta(M)$ each (Algorithm 1 line 6). For each block of K and V, we iterate over all blocks of Q (Algorithm 1 line 8) to compute the intermediate values, resulting in $\Theta(NdM^{-1})$ passes over Q. Each pass loads $\Theta(Nd)$ elements, which amounts to $\Theta(N^2 d^2 M^{-1})$ HBM accesses. We similarly prove that the backward pass of standard attention requires $\Theta(Nd + N^2)$ HBM accesses while the backward pass of FlashAttention requires $\Theta(N^2 d^2 M^{-1})$ HBM accesses (Appendix B). We prove a lower-bound: one cannot asymptotically improve on the number of HBM accesses for all values of M (the SRAM size) when computing exact attention.

Proposition 3. *Let N be the sequence length, d be the head dimension, and M be size of SRAM with $d \le M \le Nd$. There does not exist an algorithm to compute exact attention with $o(N^2 d^2 M^{-1})$ HBM accesses for all M in the range $[d, Nd]$.*

The proof relies on the fact that for $M = \Theta(Nd)$ any algorithm must perform $\Omega(N^2 d^2 M^{-1}) = \Omega(Nd)$ HBM accesses. This type of lower bound over a subrange of M is common in the streaming algorithms literature [88]. We leave proving parameterized complexity [27] lower bounds in terms of M as exciting future work.

We validate that the number of HBM accesses is the main determining factor of attention run-time. In Fig. 2 (left), we see that even though FlashAttention has higher FLOP count compared to standard attention (due to re-computation in the backward pass), it has much fewer HBM accesses, resulting in much faster runtime. In Fig. 2 (middle), we vary the block size B_c of FlashAttention, which results in different amounts of HBM accesses, and measure the runtime of the forward pass. As block size increases, the number of HBM accesses decreases (as we make fewer passes over the input), and runtime decreases. For large enough block size (beyond 256), the runtime is then bottlenecked by other factors (e.g., arithmetic operations). Moreover, larger block size will not fit into the small SRAM size.

Our implementation uses Apex's FMHA code (https://github.com/NVIDIA/apex/tree/master/apex/ contrib/csrc/fmha) as a starting point.

Algorithm Details

We first derive the forward and backward passes of attention and show that they can be computed in a memory-efficient manner (requiring extra memory linear instead of quadratic in the sequence length). Though they reduce the amount of extra memory required, naively they still incur quadratic HBM accesses, resulting in slower execution speed. We describe the FlashAttention algorithm to implement both the forward and the backward passes on GPUs that reduces HBM accesses, leading to both faster runtime and smaller memory footprint.

Memory-efficient forward pass

The main challenge in making attention memory-efficient is the softmax that couples the columns of K (and columns of V). Our approach is to compute the softmax normalization constant separately to decouple the columns. This technique [2] has been used in the literature [3] to show that attention computation does not need quadratic *extra* memory (though the number of HBM accesses is still quadratic, resulting in slow run-time). For simplicity, we omit here the max-shifting step during softmax. Recall that given input sequences Q, K, V $\in \mathbb{R}^{N \times d}$, we want to compute the attention output O $\in \mathbb{R}^{N \times d}$:

$$S = QK^T \in \mathbb{R}^{N \times N}, \quad P = \mathrm{softmax}(S) \in \mathbb{R}^{N \times N}, \quad O = PV \in \mathbb{R}^{N \times d}.$$

We have that $S_{ij} = q_i^T k_j$ where q_i and k_j are the i-th and j-th columns of Q and K respectively. Define the normalization constants of softmax:

$$L_i = \sum_j e^{q_i^T k_j}. \tag{1}$$

Let v_j be the j-th column of $\mathbf{V}$, then the i-th columns of the output is

$$o_i = P_{i:}\mathbf{V} = \sum_j P_{ij} v_j = \sum_j \frac{e^{q_i^T k_j}}{L_i} v_j. \tag{2}$$

We see that once L_i is computed, we can compute o_i without extra memory by repeatedly summing $\frac{e^{q_i^T k_j}}{L_i} v_j$. Therefore the forward pass can be computed with $O(n)$ extra memory:

1. Compute L_i for all i according to Eq. (1), which takes $O(n)$ extra memory.
2. Compute o_i for all i according to Eq. (2), which takes $O(d)$ extra memory.

The full algorithm below contains all the steps.

Flash Attention: Forward Pass

Given input sequences **Q**, **K**, **V** 2 RN×d, we want to compute the attention output **O** 2 RN×d:

$$\mathbf{S} = \tau \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{S}^{\text{masked}} = \text{MASK}(S) \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \text{softmax}(\mathbf{S}^{\text{masked}}) \in \mathbb{R}^{N\times N},$$
$$\mathbf{P}^{\text{dropped}} = \text{dropout}(\mathbf{P}, p_{\text{drop}}), \quad \mathbf{O} = \mathbf{P}^{\text{dropped}}\mathbf{V} \in \mathbb{R}^{N\times d},$$

where $\tau \in$ R is some softmax scaling (typically $\frac{1}{\sqrt{d}}$), mask is some masking function that sets some entries of the input to $-\infty$ and keep other entries the same (e.g., key padding mask when sequences in the batch don't have the same lengths and are padded), and dropout(x, p) applies dropout to x elementwise (i.e., output $\frac{1}{1-p}x$ with probability $1 - p$ and output 0 with probability p for each element x).

The full algorithm is in Algorithm 2. We save the output O, the softmax statistics ℓ and m, and the pseudo-random number generator state R for the backward pass.

Algorithm 2 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}.

1: Initialize the pseudo-random number generator state $\mathcal{R}$ and save to HBM.
2: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.
3: 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.
4: 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.
5: 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.
6: **for** $1 \le j \le T_c$ **do**
7: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
8: **for** $1 \le i \le T_r$ **do**
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 for**
18: **end for**
19: Return $\mathbf{O}, \ell, m, \mathcal{R}$.

Comparison with Rabe and Staats [3]

Here is some similarities and differences between our FlashAttention algorithm and the algorithm of Rabe and Staats [3].

Conceptually, both FlashAttention and Rabe and Staats [3] operate on blocks of the attention matrix using the well-established technique of tiling (or softmax scaling) [2]. To reduce the memory footprint, both methods avoid storing the large attention matrix in the forward pass and recompute it in the backward pass.

The first major difference is that Rabe and Staats [3] focuses on the reducing the total memory footprint (maximum amount of GPU memory required) while FlashAttention focuses on reducing memory accesses (the number of memory reads/writes). The amount of memory access is the primary determining factor of runtime. Reducing memory accesses also necessarily reduces the total amount of memory required (e.g., if an operation incurs A memory accesses, then its total memory requirement is at most A). As a result, FlashAttention is faster than standard attention (2-4×) while Rabe and Staats [3] is around the same speed or slightly slower than standard attention. In terms of total memory required, both methods offer substantial memory saving.

The second difference between the two methods is the way information is summarized from each block to pass to the next block. Rabe and Staats [3] summarizes each block with its temporary output along with the softmax normalization statistics. At the end of the forward pass, the temporary outputs of all the blocks are combined using the statistics to produce the final output. FlashAttention instead incrementally updates the output (Algorithm 1 line 12) after processing each block, so only one copy of the output is needed (instead of K copies for K blocks). This means that FlashAttention has smaller total memory requirement compared to Rabe and Staats [3].

The final major difference is the way the backward pass is computed. Rabe and Staats [3] uses gradient checkpointing to recompute the attention matrix and the temporary output of each block. FlashAttention instead simplifies the backward pass analytically. It only recomputes the attention matrix and does not recompute the temporary output of each block. This reduces the memory requirement for the backward pass and yields speedup.

Limitations and Future Directions

We discuss limitations of our approach and future directions. Related work is given in Appendix A.

Compiling to CUDA. Our current approach to building IO-aware implementations of attention requires writing a new CUDA kernel for each new attention implementation. This requires writing the attention algorithm in a considerably lower-level language than PyTorch and requires significant engineering effort. Implementations may also not be transferrable across GPU architectures. These limitations suggest the need for a method that supports writing attention algorithms in a high-level language (e.g., PyTorch), and compiling to IO-aware implementations in CUDA.

IO-Aware Deep Learning. We believe that the IO-aware approach can extend beyond attention. Attention is the most memory-intensive computation in Transformers, but every layer in a deep network touches GPU HBM. We hope our work inspires IO-aware implementations of additional modules..

Multi-GPU IO-Aware Methods. Our IO-aware implementation of attention is optimal within constants for computing attention on a single GPU. However, the attention computation may be parallelizable across multiple GPUs [72]. Using multiple GPUs adds an additional layer to IO analysis—accounting for data transfer between GPUs..

2. Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.
3. Markus N Rabe and Charles Staats. Self-attention does not need $O(n^2)$ memory. *arXiv preprint arXiv:2112.05682*, 2021