FlashAttention Feedback

Fast and Memory-Efficient Exact Attention with IO-Awareness

2024.07.25

HPC Lab

홍성준, 박지연, 김유나

Flash Attention에서 Load를 2번 하는 이유

✓ K_j, V_j를 매번 다른 블록 Q_i, O_i, l_i, m_i에 대한 연산을 수행하기 위함이다.

✓ Attention Output인 O도 Block 단위로 저장되고, 최종 O는 N x d이다.

Algorithm 1 FLASHATTENTION

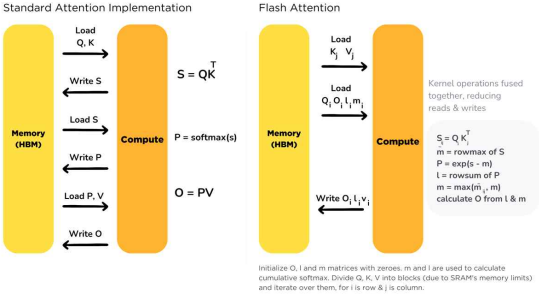
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.
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 $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ 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.
4: 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.
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.

S, P가 성능에 영향을 미치는가?

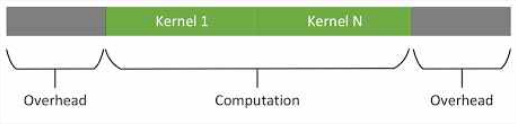
✓ Standard Attention에서는 Attention Output의 중간 값인 S와 P를 HBM에 Write하기 위해 $O(N^2)$의 메모리가 필요하다.

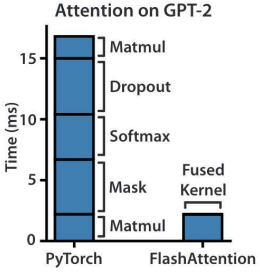
✓ FlashAttention에서는 S와 P를 HBM에 저장하지 않고 l과 m을 저장하기 위한 O(N)의 메모리가 필요하다.



Kernel Fusion

- ✓ Kernel Fusion은 연속된 연산 작업들을 단일 하드웨어로 통합하는 작업이다.
- ✓ (Matmul, Mask, Softmax, Dropout) 연산을 **하나의 Kernel**로 통합함으로써 메모리 액세스 시간을 감소시켜, Overhead를 줄인다.
- ✓ 따라서, Kernel Fusion에서 작은 커널 함수들을 단위로 결합하고 결합된 커널 함수를 GPU에서 실행하여 스케줄링 효율성을 높인다. 이 과정에서 GPU 스케줄러는 결합된 큰 커널을 단위로 처리하게 된다.





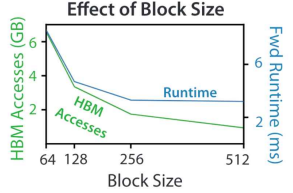
HBM Accesses 단위가 GB로 표현된 이유

- ✓ FlashAttention의 블록 크기를 변경하여 HBM 접근 횟수와 Forward pass의 Runtime을 측정했다.
- ✓ 블록 크기가 **증가**하면 **HBM 접근* 횟수가 줄어들고 Runtime이 감소**한다.
- ✓ 메모리 접근 횟수가 줄어들수록 데이터 중복 저장이 적어지고, 효율적으로 관리되므로 총 메모리 요구량도 함께 감소한다.

(the number of memory reads/writes). As mentioned in Section 2, 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 [66]

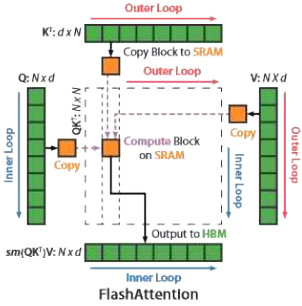
* HBM 접근

$$\theta\left(\frac{N^2d^2}{M}\right) = \theta\left(\frac{N^2d}{B_c}\right)$$



Block

- ✓ 파일 시스템에서 적용되는 I/O Block Size와 FlashAttention에서의 Block Size는 다른 개념이다.
- ✓ I/O Block Size는 저장 장치에서 데이터를 읽거나 쓸 때 사용하는 최소 단위이다.
- ✓ FlashAttention에서의 Block Size는 입력 데이터를 작은 블록 단위로 나누어 연산할 때 사용하는 단위이다.



Memory Usage

✓ Standard Attention에서는 $O(N^2)$의 메모리가 필요했지만, Forward pass는 O(N)의 메모리가 필요하다.

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

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

1. L_i : Softmax 정규화 상수를 N개 만큼 저장해야 하므로, O(N)의 메모리가 필요하다.

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

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

2. o_i : Attention output은 크기가 d인 벡터가 누적합으로 업데이트 되기 때문에 O(d)의 메모리가 필요하다.

Memory Usage

✓ Standard Attention에서는 O(N^2)의 메모리가 필요했지만, Backward pass는 O(N)의 메모리가 필요하다.

$$dv_j = \sum_i P_{ij} do_i = \sum_i \frac{e^{q_i^T k_j}}{L_i} do_i.$$

1. Compute dv_j for all j according to Eq. (3), which takes $O(d)$ extra memory.

1. v_j는 크기가 d인 벡터이다. 각 j에 대해 dv_j를 계산하기 위해 O(d)만큼의 메모리를 필요로 한다.

$$D_i = P_{i:}^T dP_{i:} = \sum_j \frac{e^{q_i^T k_j}}{L_i} do_i^T v_j = do_i^T \sum_j \frac{e^{q_i^T k_j}}{L_i} v_j = do_i^T o_i,$$

2. Compute D_i for all i according to Eq. (4), which takes $O(n)$ extra memory.

2. D_i를 계산할 때 필요한 메모리는 dP_i를 저장하는데 필요하다. dP_i는 크기가 N인 벡터이다.

따라서, 각 i에 대해 D_i를 계산하기 위해 O(N)만큼의 메모리가 필요하다.

Memory Usage

✓ Standard Attention에서는 O(N^2)의 메모리가 필요했지만, Backward에서 O(N)의 메모리가 필요하다.

$$dq_i = \sum_j dS_{ij} k_j = \sum_j P_{ij}(dP_{ij} - D_i)k_j = \sum_j \frac{e^{q_i^T k_j}}{L_i}(do_i^T v_j - D_i)k_j.$$

3. Compute dq_i for all i according to Eq. (5), which takes $O(d)$ extra memory.

3. q_j는 크기가 d인 벡터이다. 각 j에 대해 dq_j를 계산하기 위해 O(d)만큼의 메모리를 필요로 한다.

$$dk_j = \sum_i dS_{ij} q_i = \sum_i P_{ij}(dP_{ij} - D_i)q_i = \sum_i \frac{e^{q_i^T k_j}}{L_i}(do_i^T v_j - D_i)q_i.$$

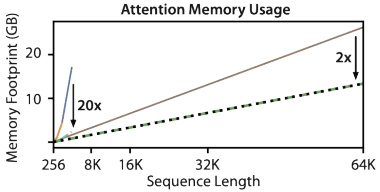
4. Compute dk_j for all j according to Eq. (6), which takes $O(d)$ extra memory.

4. k_j는 크기가 d인 벡터이다. 각 j에 대해 dk_j를 계산하기 위해 O(d)만큼의 메모리를 필요로 한다.

Block-Sparse FlashAttention

✓ Masking을 통해 특정 입력에 대한 Attention을 제한하면, Block-Sparse Attention은 non-mask Block만 처리하게 되어 메모리 접근 횟수를 줄이고, 전체 메모리 사용량을 감소시킨다.

✓ 아래의 실험에서는 **Masking을 하지 않고** 메모리 사용량을 비교하였기 때문에, Block-Sparse FlashAttention과 FlashAttention의 메모리 사용량이 동일하다.



Dropout	Masking	Pass	Table
No	No	Memory Usage (Combined)	Table 21

2 FlashAttention-2

Faster Attention with Better Parallelism and Work Partitioning

Contents Table

1 Abstract

- ✓ FlashAttention은 메모리를 크게 절약하고 Runtime 속도를 향상시켰지만,

  여전히 최적화된 행렬 곱셈(GEMM) 연산 만큼 빠르지 않기 때문에 FlashAttention-2를 제안한다.
- ✓ FlashAttention-2는 Non-Matmul FLOPs 수를 줄이기 위해 FlashAttention 알고리즘을 수정한다.
- ✓ 단일 head의 경우에도 다른 Thread Blocks 간의 Attention 계산을 병렬화하여 GPU 자원 활용을 높인다.
- ✓ 각 Thread Blocks 내에서 Warps 간의 작업을 분배하여 Shared Memory Read/Write를 줄인다.
- ✓ FlashAttention-2는 FlashAttention에 비해 약 2배의 속도 향상을 달성하며,

  A100에서 이론적 최대 FLOPs/s의 50-73%에 도달하고 GEMM 연산의 효율성에 가까워졌다.

FlashAttention

✓ FlashAttention은 Tiling, Recomputation을 활용하여 메모리 사용량을 시퀀스 길이에 대해 선형적으로 줄여주고 Runtime 속도를 크게 향상시켰다.

✓ 그러나 FlashAttention의 단점으로는 아래와 같다.

1. GPU는 행렬 곱셈을 가속화하기 때문에, Non-Matmul FLOPs는 전체 FLOPs의 작은 부분만 차지해도, 수행 시간이 더 오래 걸린다.
2. 시퀀스 길이가 증가함에 따라 FlashAttention은 다른 기본 연산 (예: 행렬 곱셈(GEMM)) 만큼 효율적이지 않다.
3. GPU의 다른 Thread blocks과 Warps 간의 비효율적인 작업 분할로 인해 낮은 점유율이나 불필요한 Shared Memory Read/Write가 발생한다.

FlashAttention-2

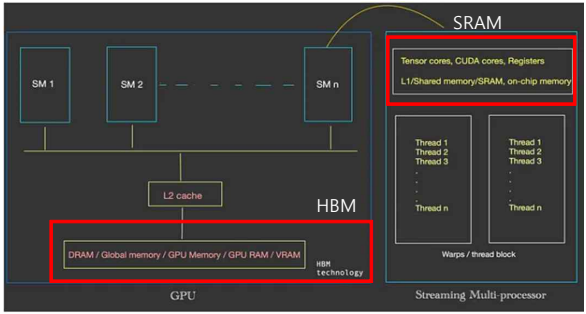
1. FlashAttention-2는 출력값을 변경하지 않으면서 Non-Matmul FLOPs 수를 줄이기 위해서 알고리즘을 수정하였다.

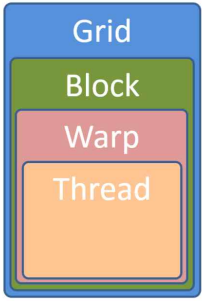
2. 시퀀스 길이, 배치 크기 및 헤드 수 차원에 따라 Forward pass와 Backward pass를 병렬화한다.

3. 하나의 Attention 블록 내에서도, Shared Memory (SRAM)의 Read/Write를 줄이기 위해 서로 다른 Warps 간의 작업을 분할한다.

Execution Model

- ✓ GPU는 많은 수의 Thread로 연산(Kernel)을 실행한다.
- ✓ 각 Thread Block 내에서 Thread는 Warps(32개의 thread 그룹)로 그룹화된다.
- ✓ Thread Block 내의 Warps는 Shared Memory Read/Write를 통해 통신할 수 있다.





Algorithm – Forward pass

✓ Non-Matmul FLOPs를 줄이기 위해 Softmax 연산을 수정한다.

$$\mathbf{O}^{(2)} = \mathrm{diag}(\ell^{(1)}/\ell^{(2)})^{-1}\mathbf{O}^{(1)} + \mathrm{diag}(\ell^{(2)})^{-1}e^{\mathbf{S}^{(2)}-m^{(2)}}\mathbf{V}^{(2)}. \quad \Rightarrow \quad \tilde{\mathbf{O}}^{(2)} = \mathrm{diag}(\ell^{(1)})^{-1}\mathbf{O}^{(1)} + e^{\mathbf{S}^{(2)}-m^{(2)}}\mathbf{V}^{(2)}.$$

1. Non-Matmul인 $diag(\ell^{(2)})^{-1}$로 Scale하지 않고,

   내부 루프 밖에서만 $\tilde{O}^{(last)}$를 $diag(\ell^{last})^{-1}$로 한 번 Scale하여 얻는다.

2. Backward pass에 사용될 Softmax Normalization Constants인 m^j과 ℓ^j 둘 다 저장하지 않고,

   $L^j = m^j + \log(\ell^j)$을 저장한다.

Algorithm – Forward pass

Algorithm 1 FlashAttention-2 forward pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, block sizes B_c, B_r.

1: 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.

2: Divide the output $\mathbf{O} \in \mathbb{R}^{N \times d}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, and divide the logsumexp L into T_r blocks $L_i, \dots, L_{T_r}$ of size B_r each.

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

4: Load $\mathbf{Q}_i$ from HBM to on-chip SRAM.

5: On chip, initialize $\mathbf{O}_i^{(0)} = (0)_{B_r \times d} \in \mathbb{R}^{B_r \times d}$, $\ell_i^{(0)} = (0)_{B_r} \in \mathbb{R}^{B_r}$, $m_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$.

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

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

8: On chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

9: On chip, compute $m_i^{(j)} = \max(m_i^{(j-1)}, \text{rowmax}(\mathbf{S}_i^{(j)})) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - m_i^{(j)}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\ell_i^{(j)} = e^{m_i^{(j-1)} - m_i^{(j)}} \ell_i^{(j-1)} + \text{rowsum}(\tilde{\mathbf{P}}_i^{(j)}) \in \mathbb{R}^{B_r}$.

10: On chip, compute $\mathbf{O}_i^{(j)} = \text{diag}(e^{m_i^{(j-1)} - m_i^{(j)}})^{-1} \mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}_j$.

11: **end for**

12: On chip, compute $\mathbf{O}_i = \text{diag}(\ell_i^{(T_c)})^{-1} \mathbf{O}_i^{(T_c)}$

13: On chip, compute $L_i = m_i^{(T_c)} + \log(\ell_i^{(T_c)})$.

14: Write $\mathbf{O}_i$ to HBM as the i-th block of $\mathbf{O}$.

15: Write L_i to HBM as the i-th block of L.

16: **end for**

17: Return the output $\mathbf{O}$ and the logsumexp L.

Forward pass – parallelism

- ✓ 기존 FlashAttention은 배치 크기와 헤드 수를 기준으로 병렬화를 수행한다.
- ✓ 각 Attention head를 하나의 thread block에서 처리했다. 총 Thread Block 개수 = Batch size * head
- ✓ 배치 크기와 헤드 수가 작을 경우 GPU 효율성이 떨어지기 때문에,

  FlashAttention-2는 시퀀스 길이에 대한 병렬을 추가하였다.
- ✓ 시퀀스 길이에 대한 병렬은 배치 크기와 헤드 수가 작을 때도 GPU의 점유율을 높여 속도를 향상시킨다.

Forward pass. We see that the outer loop (over sequence length) is embarrassingly parallel, and we schedule them on different thread blocks that do not need to communicate with each other. We also parallelize over the batch dimension and number of heads dimension, as done in FlashAttention. The increased parallelism over sequence length helps improve occupancy (fraction of GPU resources being used) when the batch size and number of heads are small, leading to speedup in this case.

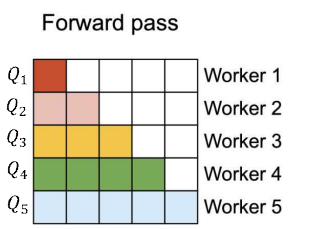
These ideas of swapping the order of the loop (outer loop over row blocks and inner loop over column blocks, instead of the other way round in the original FlashAttention paper), as well as parallelizing over the sequence length dimension were first suggested and implemented by Phil Tillet in the Triton [17] implementation.[8]

Forward pass – parallelism

✓ FlashAttention-2는 루프 순서(외부 루프와 내부 루프)를 swap함으로서

Q가 K,V를 거쳐 O를 업데이트한다.

✓ Q를 Block으로 나누어(시퀀스 길이에 대한 병렬) 각각 다른 Thread Block에 할당하여 병렬로 수행한다.



FlashAttention

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.

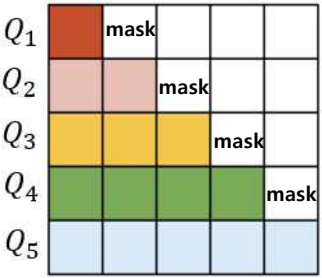
FlashAttention-2

3: **for** $1 \le i \le T_r$ **do**
4: Load $\mathbf{Q}_i$ from HBM to on-chip SRAM.
5: On chip, initialize $\mathbf{O}_i^{(0)} = (0)_{B_r \times d} \in \mathbb{R}^{B_r \times d}, \ell_i^{(0)} = (0)_{B_r} \in \mathbb{R}^{B_r}, m_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$.
6: **for** $1 \le j \le T_c$ **do**
7: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

Forward pass – Causal Mask

✓ 열 인덱스가 행 인덱스보다 큰 블록(Future Information)에 대해서는 해당 블록의 계산을 건너뛴다.

✓ 정사각형 행렬일 때, Causal Mask를 전부 적용할 필요가 없이 하나의 블록에만 Mask를 적용한다.



Backward pass

✓ Softmax 연산에서 Softmax Normalization Constants인 m과 l 대신 logsumexp L을 사용한다.

나머지 backward pass 과정은 FlashAttention과 유사하다.

Algorithm 2 FlashAttention-2 Backward Pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N \times d}$ in HBM, vector $L \in \mathbb{R}^N$ in HBM, block sizes B_c, B_r.

1: 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.

2: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\mathbf{dO}$ into T_r blocks $\mathbf{dO}_i, \dots, \mathbf{dO}_{T_r}$ of size $B_r \times d$ each, and divide L into T_r blocks $L_i, \dots, L_{T_r}$ of size B_r each.

3: Initialize $\mathbf{dQ} = (0)_{N \times d}$ in HBM and divide it into T_r blocks $\mathbf{dQ}_1, \dots, \mathbf{dQ}_{T_r}$ of size $B_r \times d$ each. Divide $\mathbf{dK}, \mathbf{dV} \in \mathbb{R}^{N \times d}$ in to T_c blocks $\mathbf{dK}_1, \dots, \mathbf{dK}_{T_c}$ and $\mathbf{dV}_1, \dots, \mathbf{dV}_{T_c}$, of size $B_c \times d$ each.

4: Compute $D = \text{rowsum}(\mathbf{dO} \circ \mathbf{O}) \in \mathbb{R}^d$ (pointwise multiply), write D to HBM and divide it into T_r blocks $D_1, \dots, D_{T_r}$ 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: Initialize $\mathbf{dK}_j = (0)_{B_c \times d}, \mathbf{dV}_j = (0)_{B_c \times d}$ on SRAM.

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

9: Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, L_i, D_i$ from HBM to on-chip SRAM.

10: On chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

11: On chip, compute $\mathbf{P}_i^{(j)} = \exp(\mathbf{S}_{ij} - L_i) \in \mathbb{R}^{B_r \times B_c}$.

12: On chip, compute $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_i^{(j)})^T \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$.

13: On chip, compute $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \mathbf{V}_j^T \in \mathbb{R}^{B_r \times B_c}$.

14: On chip, compute $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ (\mathbf{dP}_i^{(j)} - D_i) \in \mathbb{R}^{B_r \times B_c}$

15: Load $\mathbf{dQ}_i$ from HBM to SRAM, then on chip, update $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \mathbf{dS}_i^{(j)} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$, and write back to HBM.

16: On chip, compute $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + \mathbf{dS}_i^{(j)T} \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$.

17: **end for**

18: Write $\mathbf{dK}_j, \mathbf{dV}_j$ to HBM.

19: **end for**

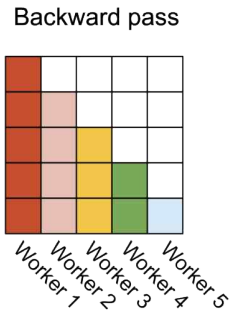
20: Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.

Backward pass

✓ 다른 Block이 같은 dQ를 업데이트 해야 하기 때문에, 이를 위해 Share Computation을 수행한다.

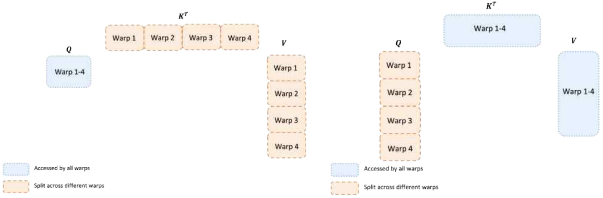
✓ 또한 시퀀스 길이 차원에서 병렬화한다.

5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: Initialize $\mathbf{dK}_j = (0)_{B_c \times d}, \mathbf{dV}_j = (0)_{B_c \times d}$ on SRAM.
8: **for** $1 \le i \le T_r$ **do**
9: Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, L_i, D_i$ from HBM to on-chip SRAM.
10: On chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
11: On chip, compute $\mathbf{P}_i^{(j)} = \exp(\mathbf{S}_{ij} - L_i) \in \mathbb{R}^{B_r \times B_c}$.
12: On chip, compute $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_i^{(j)})^\top \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$.
13: On chip, compute $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \mathbf{V}_j^\top \in \mathbb{R}^{B_r \times B_c}$.
14: On chip, compute $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ (\mathbf{dP}_i^{(j)} - D_i) \in \mathbb{R}^{B_r \times B_c}$.
15: Load $\mathbf{dQ}_i$ from HBM to SRAM, then on chip, update $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \mathbf{dS}_i^{(j)} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$, and write back to HBM.
16: On chip, compute $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + \mathbf{dS}_i^{(j)\top} \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$.
17: **end for**
18: Write $\mathbf{dK}_j, \mathbf{dV}_j$ to HBM.
19: **end for**



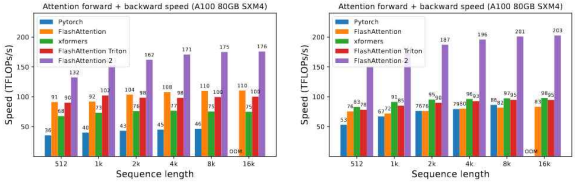
Work Partitioning Between Warps

- ✓ FlashAttention는 Q를 모든 Warp가 접근할 수 있도록 하고, K, V를 4개의 Warp로 분할한다.
- ✓ 이 경우 모든 Warp가 중간 결과를 Shared Memory에 기록해야 하고, 합산해야 하기 때문에 비효율적이다.
- ✓ FlashAttention-2는 Q를 4개의 Warp로 분할하고, K, V를 모든 Warp가 접근할 수 있도록 한다.
- ✓ 각 Warp가 행렬 곱셈을 한 후, 각 Warp는 자신이 공유하는 V와 곱셈을 수행하여 출력을 얻는다.
- ✓ 이 방식은 Warp 간 통신이 필요하지 않게 되어 Shared Memory의 Read/Write가 줄어 속도가 향상된다.



(a) FlashAttention (b) FlashAttention-2

Benchmarking Attention



(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

Benchmarking Attention

- ✓ GPT-3 모델의 Training Speed를 8xA100 GPU에서 측정한 결과이다.
- ✓ FlashAttention-2는 Without FlashAttention보다 속도가 최대 2.8배 향상했다.
- ✓ FlashAttention-2는 FlashAttention보다 속도가 최대 1.3배 향상했다.

Table 1: Training speed (TFLOPs/s/GPU) of GPT-style models on 8×A100 GPUs. FLASHATTENTION-2 reaches up to 225 TFLOPs/s (72% model FLOPs utilization). We compare against a baseline running without FLASHATTENTION.

Model	Without FLASHATTENTION	FLASHATTENTION	FLASHATTENTION-2
GPT3-1.3B 2k context	142 TFLOPs/s	189 TFLOPs/s	196 TFLOPs/s
GPT3-1.3B 8k context	72 TFLOPS/s	170 TFLOPs/s	220 TFLOPs/s
GPT3-2.7B 2k context	149 TFLOPs/s	189 TFLOPs/s	205 TFLOPs/s
GPT3-2.7B 8k context	80 TFLOPs/s	175 TFLOPs/s	225 TFLOPs/s


Thank You

LogSumExp

$$\begin{aligned}
\text{LogSumExp}(x_1 \ldots x_n) &= \log\left(\sum_{i=1}^{n} e^{x_i}\right) \\
&= \log\left(\sum_{i=1}^{n} e^{x_i - c} e^{c}\right) \\
&= \log\left(e^{c} \sum_{i=1}^{n} e^{x_i - c}\right) \\
&= \log\left(\sum_{i=1}^{n} e^{x_i - c}\right) + \log(e^{c}) \\
&= \log\left(\sum_{i=1}^{n} e^{x_i - c}\right) + c
\end{aligned}$$

FlashAttention

✓ FlashAttention은 메모리 I/O를 줄이기 위해 Tiling과 Recomputation을 제안한다.

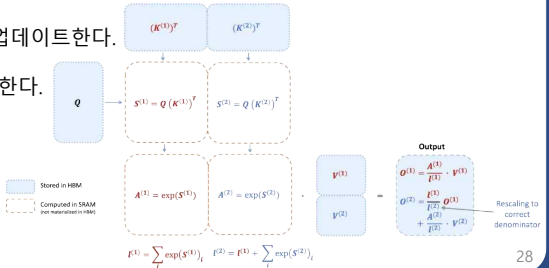
✓ Tiling 과정은 다음과 같은 순서로 수행된다.

1. HBM에서 SRAM으로 입력 block을 Load한다.

2. 해당 Block에 대해 Attention 계산한다.

3. 중간 값 S, P를 HBM에 기록하지 않고 출력을 업데이트한다.

✓ Recomputation은 각 Block의 출력을 rescaling한다.

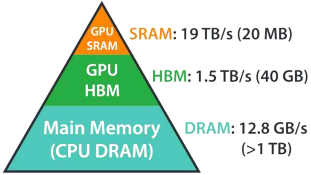


HBM, SRAM 접근 속도 차이

✓ GPU 메모리 계층은 크기와 속도가 다른 여러 형태의 메모리로 구성되며, 더 작은 메모리가 더 빠르다.

✓ SRAM은 on-chip 메모리로, L1 Cache/Shared 메모리 등을 지칭한다.

✓ 계산 속도가 메모리 속도에 비해 빨라짐에 따라, 연산은 점점 더 **HBM 접근에 의해 병목 현상**이 발생한다.



Memory Hierarchy with Bandwidth & Memory Size