FlashAttention

Fast and Memory-Efficient Exact Attention with IO-Awareness

2024.07.19

HPC Lab

홍성준, 박지연, 김유나

Contents Table

1 Abstract

- ✓ Self-Attention의 시간 및 메모리 **복잡도가 시퀀스 길이에 대해 제곱**으로 비례하여,

  긴 시퀀스에서는 느리고 메모리를 많이 소모한다.

- ✓ GPU HBM와 GPU on-chip SRAM 간의 **메모리 Read & Write 횟수를 줄이기** 위해,

  GPU에서 **forward pass와 Backward pass**를 구현하고, **tiling**을 사용하는 FlashAttention을 제안한다.

- ✓ 또한, **Block-sparse Attention**으로 확장하여 빠른 Approximate Attention을 제공한다.

- ✓ Flash Attention 및 Block-sparse Attention은 Transformer의 더 **긴 Context**의 입력을 가능하게 한다.

✓ 이전 연구는, FLOP 감소에 집중하며 **메모리 접근(IO)에 발생하는 오버헤드를 무시**하는 경향이 있다.

✓ Attention Algorithm을 **IO-aware**로 만들기 위해,

빠른 GPU on-chip SRAM과 상대적으로 느린 GPU HBM 사이의 **Read & Write**를 고려해야 한다.

✓ Flash Attention의 주요 목표는 **Attention 행렬을 HBM에 읽고 쓰는 횟수를 줄이는 것**이다.

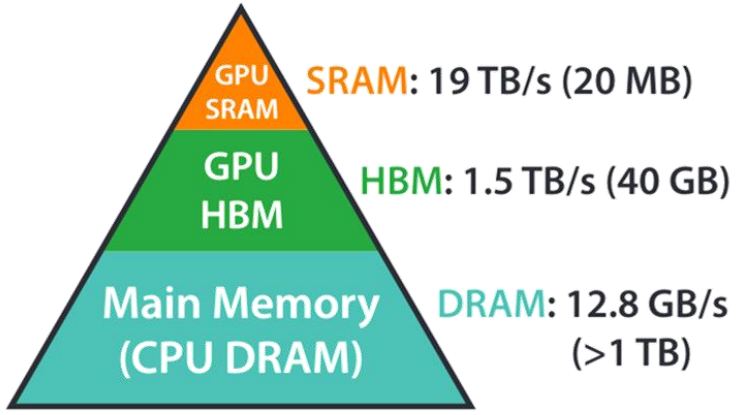
(1) 전체 입력에 접근하지 않고 Attention 계산을 재구성하여 입력을 **블록**으로 나누는 **Tiling**을 제안한다.

(2) **Softmax normalization factor**를 Forward pass에 저장하여

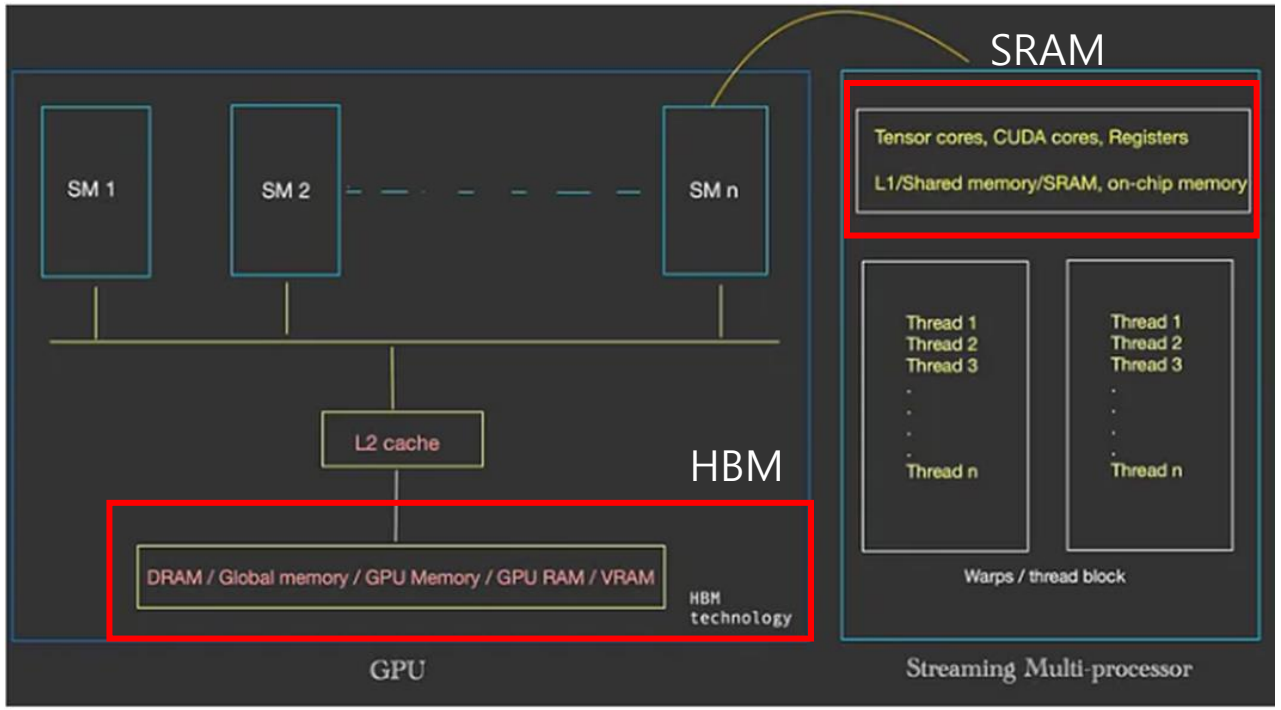
Backward pass에서는 SRAM에서 Attention을 빠르게 **Recomputation**할 수 있도록 한다.

Memory Hierarchy

- ✓ 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.



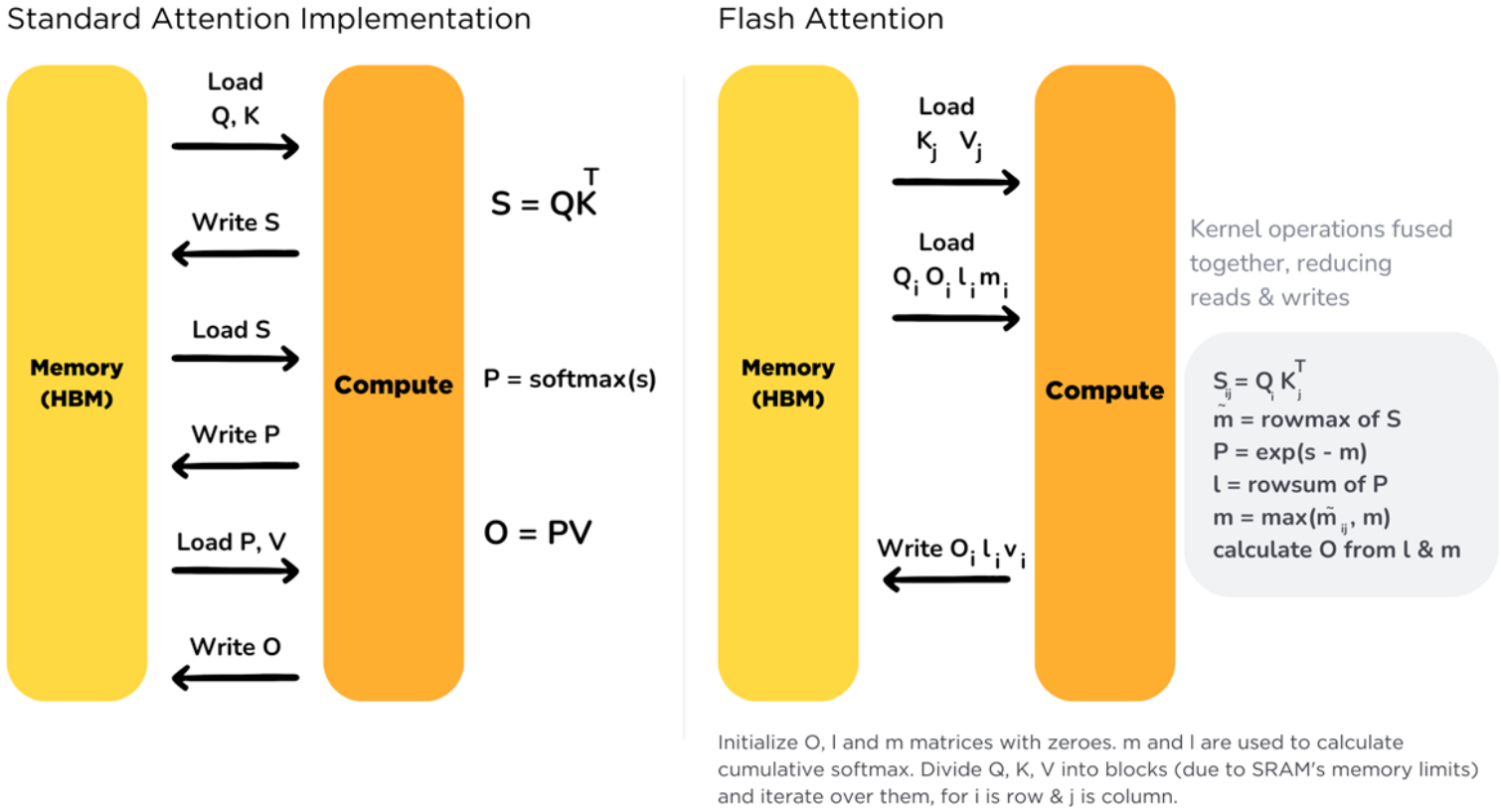
Memory Hierarchy with Bandwidth & Memory Size



Execution Model

✓ GPU는 대규모 스레드를 통해 연산(Kernel)을 실행한다.

✓ 각 커널은 **HBM에서** 레지스터 및 **SRAM으로 입력을 로드**하고, 계산을 수행한 후 **HBM에 출력을 쓴다**.



Standard Attention

✓ 표준 Attention은 행렬 S와 P(중간 값)를 HBM에 저장해야 하므로, O(N^2)의 메모리가 필요하다.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \text{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d},$$

S = Attention Score, P = Attention Weight, O = Attention Output

N: Sequence Length, d: head dimension

Algorithm 0 Standard Attention Implementation

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

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \text{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

FlashAttention

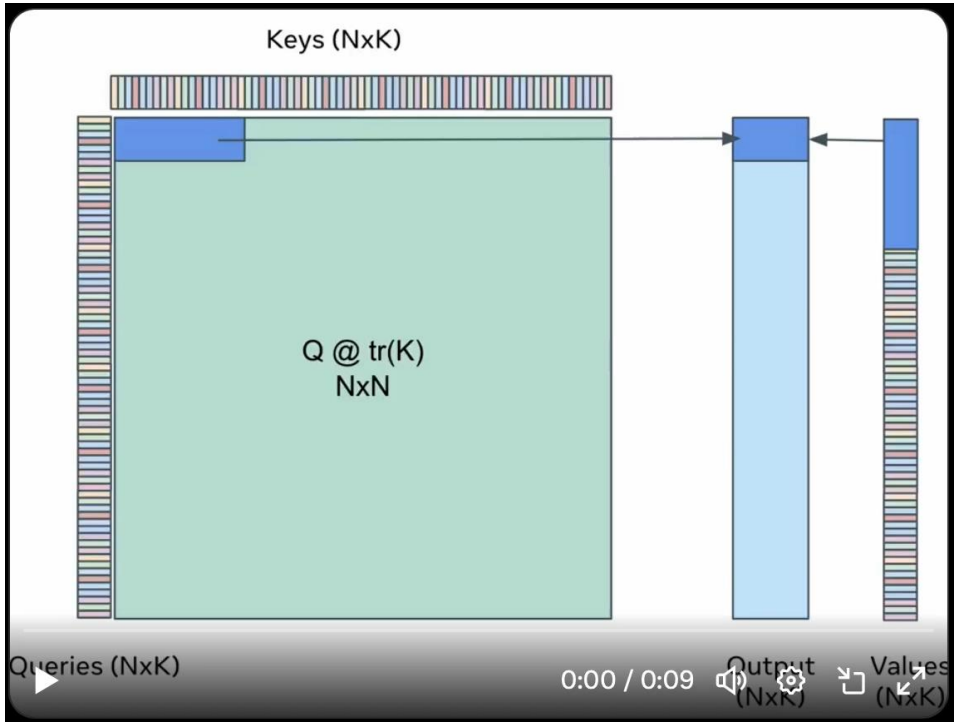
✓ Standard Attention은 HBM에 있는 Q, K, V $\in \mathbb{R}^{N \times d}$를 사용하여 Attention 연산의 중간 값을 계산하고 이를 HBM에 기록하는 과정에서 HBM 접근 횟수가 증가한다.

✓ FlashAttention은 HBM 접근 횟수를 줄이면서 정확한 Attention을 계산하기 위해서 **Tiling, Recomputation** 기법을 제안한다.

1. Q, K, V, Softmax Normalization Statistics(m, l), Attention Output(O)을 블록으로 나눔
2. 블록 별로 로드해서 Attention score 계산 (중간 값 저장 x)
3. Attention score로 m, l를 구한 후, Softmax 연산
4. O, m, l HBM 저장

Tiling

1. HBM에서 Q, K, V를 **블록** 단위로 SRAM에 로드
2. SRAM에서 Attention Output을 **블록** 단위로 계산



$$m(x) := \max_i \quad x_i, \quad f(x) := \begin{bmatrix} e^{x_1 - m(x)} & \dots & e^{x_B - m(x)} \end{bmatrix}$$

$$, \quad \ell(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

$$m(x) = m(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = \max(m(x^{(1)}), m(x^{(2)})),$$

$$f(x) = \begin{bmatrix} e^{m(x^{(1)}) - m(x)} f(x^{(1)}) & e^{m(x^{(2)}) - m(x)} f(x^{(2)}) \end{bmatrix},$$

$$\ell(x) = \ell(\begin{bmatrix} x^{(1)} & x^{(2)} \end{bmatrix}) = e^{m(x^{(1)}) - m(x)} \ell(x^{(1)}) + e^{m(x^{(2)}) - m(x)} \ell(x^{(2)}),$$

$$\text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

Recomputation

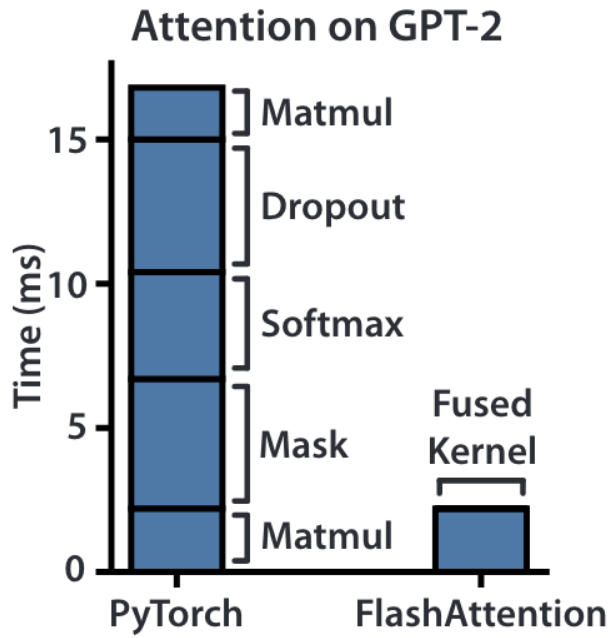
- ✓ Backward pass를 위한 중간 값인 $\boldsymbol{S}, \boldsymbol{P} \in \mathbb{R}^{N \times N}$ **를 저장하지 않는다**.
- ✓ 출력 O와 Softmax Normalization Statistics m, l을 저장함으로써,

  Backward pass에서 SRAM의 **Q, K, V 블록으로부터 S와 P를 Recomputation**할 수 있다.
- ✓ **더 많은 FLOP이 발생해도 HBM 접근이 줄어들어 Backward pass를 더 빠르게 한다.**

Kernel fusion

- ✓ **메모리 bound* 연산을 가속화**하기 위한 방법이다.
- ✓ 연산을 **하나의 Kernel**로 통합함으로써 메모리 액세스 시간과 Kernel 실행 시간을 감소하여 최적화한다.
- ✓ Standard Attention 모델 Train에서는, Attention 연산의 중간 값을 Backward pass를 위해 HBM에 저장해야 하므로, 이는 **Kernel fusion의 효과를 감소시킨다**.
- ✓ FlashAttention은 동일한 입력에 여러 연산이 적용되는 경우, 입력을 각 연산마다 여러 번 HBM에서 로드하는 대신 **한 번만 로드**하고 **중간 값을 저장하지 않는다**.

* 메모리 bound 작업에서는 메모리에서 데이터를 가져오고 쓰는 시간이 전체 작업의 속도를 결정짓는다.



Dropout - PRNG

✓ 원래 Dropout mask를 생성할 시, O(N^2)의 공간 복잡도가 발생한다.

✓ PRNG(Pseudo-random number generator)를 사용하여 0~1 사이의 값을 생성할 경우,

O(N)의 공간 복잡도가 발생하여 더 효율적이다.

1. Forward pass에서 생성되어 저장된 시드값 $\mathcal{R}$을 Backward pass에서 설정한다.
2. 시드값이 고정된 상태에서 0~1 사이의 값으로 Random Dropout mask 행렬을 생성한다.
3. dropout 확률보다 작을 경우, **0**

   dropout 확률보다 클 경우, $\frac{1}{(1-P_drop)}$으로 Dropout mask가 설정되고, Attention Weight에 곱해진다.

Algorithm 1. FlashAttention

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 $\mathbf{O}$.

Algorithm 1. FlashAttention

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.

1: SRAM의 사이즈인 M을 4d로 나누어 블럭 사이즈를 설정한다.

3: Q는 $T_r = \frac{N}{B_r}$ 개의 블럭 개수를 가지고, 각 블럭의 사이즈는 $B_r \times d$ 이다.

K와 V는 $T_c = \frac{N}{B_c}$ 개의 블럭 개수를 가지고, 각 블럭 사이즈는 $B_c \times d$ 이다.

6: K와 V의 j번 째 블럭을 HBM에서 SRAM으로 로드한다.

Algorithm 1. FlashAttention

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 $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: 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.
13: Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.

8: K와 V가 j번 째 블럭일 때, Q의 i번 째 블럭을 HBM에서 SRAM으로 로드한다.

9: Q의 i번 째 블럭과 K의 j번 째 블럭(Transpose)을 한 번에 행렬곱 한다.

행렬 곱한 중간 값(S, P)을 HBM에 write하는 과정이 없다.

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}.

Initialize the pseudo-random number generator state $\mathcal{R}$ and save to HBM.
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 $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
    On chip, compute $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$.
    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), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
    On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
    On chip, compute $\tilde{\mathbf{P}}_{ij}^{\text{dropped}} = \text{dropout}(\tilde{\mathbf{P}}_{ij}, p_{\text{drop}})$.
    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.
    Write $\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$ to HBM.
  end for
end for
Return $\mathbf{O}, \ell, m, \mathcal{R}$.

Algorithm 3. Standard Attention Backward pass

✓ Backward pass는 신경망을 학습시키는 과정에서 손실 함수의 기울기를 계산하여 가중치를 업데이트한다.

Algorithm 3 Standard Attention Backward Pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{dO} \in \mathbb{R}^{N \times d}$, $\mathbf{P} \in \mathbb{R}^{N \times N}$ in HBM.

1: Load $\mathbf{P}, \mathbf{dO}$ by blocks from HBM, compute $\mathbf{dV} = \mathbf{P}^\top \mathbf{dO} \in \mathbb{R}^{N \times d}$, write $\mathbf{dV}$ to HBM.
2: Load $\mathbf{dO}, \mathbf{V}$ by blocks from HBM, compute $\mathbf{dP} = \mathbf{dO}\mathbf{V}^\top \in \mathbb{R}^{N \times N}$, write $\mathbf{dP}$ to HBM.
3: Read $\mathbf{P}, \mathbf{dP}$ from HBM, compute $\mathbf{dS} \in \mathbb{R}^{N \times N}$ where $dS_{ij} = P_{ij}(dP_{ij} - \sum_l P_{il} dP_{il})$, write $\mathbf{dS}$ to HBM.
4: Load $\mathbf{dS}$ and $\mathbf{K}$ by blocks from HBM, compute $\mathbf{dQ} = \mathbf{dSK}$, write $\mathbf{dQ}$ to HBM.
5: Load $\mathbf{dS}$ and $\mathbf{Q}$ by blocks from HBM, compute $\mathbf{dK} = \mathbf{dS}^\top \mathbf{Q}$, write $\mathbf{dK}$ to HBM.
6: Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.

1: Attention Weight인 $\boldsymbol{P}$와 Attention Output gradient $\mathbf{dO}$를 로드하여 Value 행렬의 gradient $\mathbf{dV}$를 계산한다.

2: Attention Output gradient $\mathbf{dO}$와 Value 행렬 $\boldsymbol{V}$를 로드하여 $\mathbf{dP}$를 계산한다.

3: Attention Weight인 $\boldsymbol{P}$와 gradient $\mathbf{d}\boldsymbol{P}$를 사용하여 $\boldsymbol{dS}$를 계산한다.

4: Attention Score의 gradient $\mathbf{d}\boldsymbol{S}$와 Key 행렬 $\mathbf{K}$를 로드하여 Query 행렬의 gradient $\mathbf{dQ}$를 계산한다.

5: Attention Score의 gradient $\mathbf{d}\boldsymbol{S}$와 Query 행렬 $\mathbf{Q}$를 로드하여 Key 행렬의 gradient $\mathbf{dK}$를 계산한다.

Algorithm 4. FlashAttention Backward pass

Algorithm 4 FLASHATTENTION Backward Pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N \times d}$ in HBM, vectors $\ell, m \in \mathbb{R}^N$ in HBM, on-chip SRAM of size M, softmax scaling constant $\tau \in \mathbb{R}$, masking function MASK, dropout probability p_{drop}, pseudo-random number generator state $\mathcal{R}$ from the forward pass.

1: Set the pseudo-random number generator state to $\mathcal{R}$.
2: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.
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 $\mathbf{dO}$ into T_r blocks $\mathbf{dO}_i, \ldots, \mathbf{dO}_{T_r}$ 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: Initialize $\mathbf{dQ} = (0)_{N \times d}$ in HBM and divide it into T_r blocks $\mathbf{dQ}_1, \ldots, \mathbf{dQ}_{T_r}$ of size $B_r \times d$ each. Initialize $\mathbf{dK} = (0)_{N \times d}, \mathbf{dV} = (0)_{N \times d}$ in HBM and divide $\mathbf{dK}, \mathbf{dV}$ in to T_c blocks $\mathbf{dK}_1, \ldots, \mathbf{dK}_{T_c}$ and $\mathbf{dV}_1, \ldots, \mathbf{dV}_{T_c}$, of size $B_c \times d$ each.

✓ Standard Attention Backward pass와 다른 점은 블록 단위로 나누어 연산하는 **Tiling** 기법을 적용한다.

6: **for** $1 \le j \le T_c$ **do**
7: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
8: Initialize $\mathbf{d\tilde{K}}_j = (0)_{B_c \times d}, \mathbf{d\tilde{V}}_j = (0)_{B_c \times d}$ on SRAM.
9: **for** $1 \le i \le T_r$ **do**
10: Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
11: On chip, compute $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
12: On chip, compute $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$.
13: On chip, compute $\mathbf{P}_{ij} = \text{diag}(l_i)^{-1} \exp(\mathbf{S}_{ij}^{\text{masked}} - m_i) \in \mathbb{R}^{B_r \times B_c}$.
14: On chip, compute dropout mask $\mathbf{Z}_{ij} \in \mathbb{R}^{B_r \times B_c}$ where each entry has value $\frac{1}{1-p_{\text{drop}}}$ with probability $1 - p_{\text{drop}}$ and value 0 with probability p_{drop}.
15: On chip, compute $\mathbf{P}_{ij}^{\text{dropped}} = \mathbf{P}_{ij} \circ \mathbf{Z}_{ij}$ (pointwise multiply).
16: On chip, compute $\mathbf{d\tilde{V}}_j \leftarrow \mathbf{d\tilde{V}}_j + (\mathbf{P}_{ij}^{\text{dropped}})^\top \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$.
17: On chip, compute $\mathbf{dP}_{ij}^{\text{dropped}} = \mathbf{dO}_i \mathbf{V}_j^\top \in \mathbb{R}^{B_r \times B_c}$.
18: On chip, compute $\mathbf{dP}_{ij} = \mathbf{dP}_{ij}^{\text{dropped}} \circ \mathbf{Z}_{ij}$ (pointwise multiply).
19: On chip, compute $D_i = \text{rowsum}(\mathbf{dO}_i \circ \mathbf{O}_i) \in \mathbb{R}^{B_r}$.
20: On chip, compute $\mathbf{dS}_{ij} = \mathbf{P}_{ij} \circ (\mathbf{dP}_{ij} - D_i) \in \mathbb{R}^{B_r \times B_c}$.
21: Write $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \tau \mathbf{dS}_{ij} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$ to HBM.
22: On chip, compute $\mathbf{d\tilde{K}}_j \leftarrow \mathbf{d\tilde{K}}_j + \tau \mathbf{dS}_{ij}^\top \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$.
23: **end for**
24: Write $\mathbf{dK}_j \leftarrow \mathbf{d\tilde{K}}_j, \mathbf{dV}_j \leftarrow \mathbf{d\tilde{V}}_j$ to HBM.
25: **end for**
26: Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.

S = Attention Score

P = Attention Weight

16 ~ 26: Standard Attention Backward pass와 동일하다.

✓ S와 P는 HBM에 저장되지 않았기 때문에 SRAM에서 **Recomputation**한다.

IO Complexity of FlashAttention

✓ 시퀀스 길이를 N, 헤드 차원을 d, SRAM 크기를 M 일 경우,

Standard Attention은 $\boldsymbol{\theta(Nd + N^2)}$의 HBM 접근이 필요하다.

FlashAttention은 Standard Attention보다 훨씬 적은 $\boldsymbol{\theta\left(\frac{N^2d^2}{M}\right)}$의 HBM 접근이 필요하다.

The primary factor affecting runtime

✓ FlashAttention이 Backward pass에서 **Recomputation으로 인해**

Standard Attention보다 **더 많은 FLOP 수**를 가지고 있음에도 불구하고,

HBM 접근 수가 훨씬 적어 Attention 수행 시간이 훨씬 빠르다.

✓ 따라서, **HBM 접근 횟수**가 Attention 수행 시간을 결정하는 주요 요소이다.

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

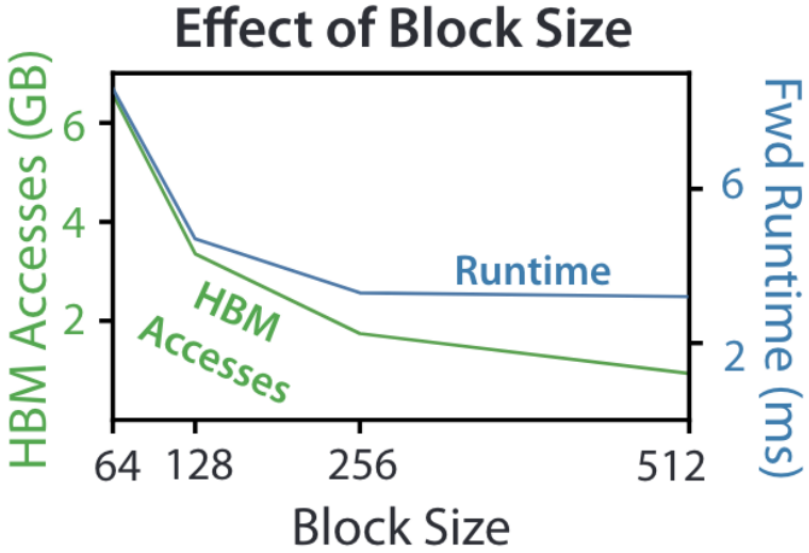
Option: GPT-2 medium, N(seq_length) = 1024, d = 64, head = 16, batch size = 64, A100 GPU

The primary factor affecting runtime

- ✓ FlashAttention의 **블록 크기**를 변경하여 HBM 접근 횟수와 Forward pass의 실행 시간을 측정했다.
- ✓ 블록 크기가 **증가**하면 **HBM 접근* 횟수가 줄어들고 실행 시간이 감소**한다.
- ✓ 블록 크기가 충분히 크면 (256 이상) 실행 시간은 산술 연산(FLOPs)에 의해 병목된다.
- ✓ 게다가, 더 큰 블록 크기는 작은 SRAM 크기에 맞지 않는다.

*HBM 접근

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



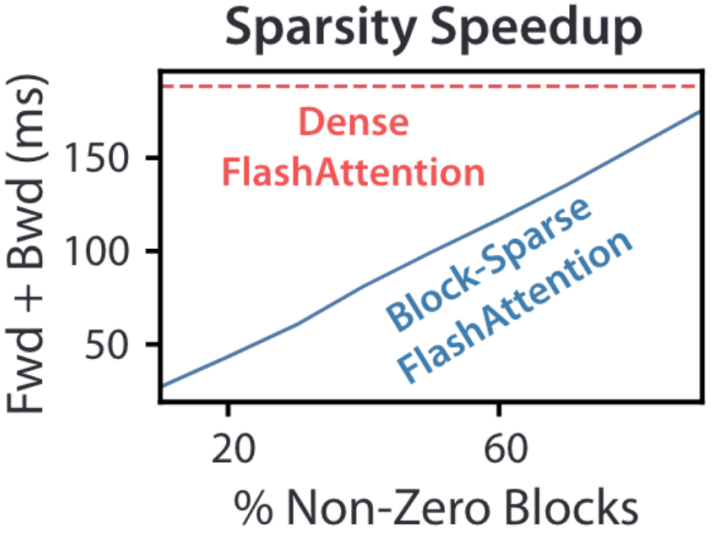
Option: GPT-2 medium, N = 1024, d = 64, head = 16, batch size = 64, A100 GPU

Block-Sparse FlashAttention

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 $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$, 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 $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ 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: **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 $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.



Option: N(seq_length) = 4k

✓ Mask Matrix $M \in \{0,1\}^{N \times N}$ 가 주어지고, **M = 0인 블럭(Mask)는 skip으로 처리되어 연산에서 제외된다.**

✓ Block-Sparse Attention은 $\theta\left(Nd + \frac{N^2 d^2}{M} s\right)$ (s: the fraction of nonzero block).

✓ Non-zero Block의 비율이 적을수록 Mask의 비율이 많기 때문에 Runtime이 적다.

Faster Models with FlashAttention

- ✓ FlashAttention을 사용하여 Wikipedia 데이터셋에서 BERT-large 모델을 훈련했다.
- ✓ MLPerf 1.1의 Training 시간 기록을 세운 Nvidia의 구현과 비교하였을 때 **Training time이 15% 감소했다**.

BERT Implementation	Training time (minutes)
Nvidia MLPerf 1.1 [58]	20.0 ± 1.5
FlashAttention (ours)	**17.4** ± 1.4

to reach the target accuracy of 72% on masked language modeling.
Averaged over 10 runs on 8xA100 GPUs.

Faster Models with FlashAttention

✓ FlashAttention을 사용하여 OpenWebtext 데이터셋에서 GPT-2에서 모델을 훈련했다.

✓ GPT-2 small과 medium은 HuggingFace 구현에 비해 **최대 3배**,

Megatron-LM에 비해 **최대 1.7배의 빠른 Training 속도**를 달성했다.

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×)**

Averaged over 10 runs on 8xA100 GPUs.

Faster Models with FlashAttention

✓ FlashAttention을 사용하여 Long-range Arena 벤치마크로 Vanilla Transformer에서 모델을 평가할 때, 각 Task는 1024에서 4096 사이의 다른 시퀀스 길이를 가진다.

✓ FlashAttention이 **2.4배**의 속도 향상을 달성했고, Block-sparse FlashAttention은 **2.8배**의 속도 향상을 달성했으며, 다른 Approximate Attention와 비교했을 때 가장 빠르다.

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×

Better Models with Longer Sequences

✓ FlashAttention은 GPT-2 small의 Context Length를 **4배**를 늘릴 수 있으며,

Megatron-LM보다 4배 더 긴 컨텍스트 길이에도 **30%** 더 빠르며, **0.7** 더 나은 perplexity를 달성했다.

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×)

Better Models with Longer Sequences

- ✓ FlashAttention을 사용하여 더 긴 시퀀스로 Transformer를 학습하면 MIMIC-III 및 ECtHR 데이터셋에서 성능이 향상된다.
- ✓ 사전 학습된 RoBERTa 모델의 **시퀀스 길이를 늘려** 성능 향상을 평가했다.
- ✓ MIMIC-III에서 시퀀스 길이 **16K가 512보다 4.3 포인트** 더 높고, ECtHR에서 시퀀스 길이 **8K가 512보다 8.5 포인트** 더 높다.

	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

Table 5: Long Document performance (micro F_1) at different sequence lengths using FlashAttention.

Better Models with Longer Sequences

- ✓ Path-X 및 Path-256 벤치마크는 긴 Context를 테스트하기 위해 설계된 LRA 벤치마크이다.
- ✓ 128×128(또는 256×256) 이미지에서 두 점이 연결되는 경로가 있는지 분류하는 작업이다.
- ✓ 이전 작업에서는 모든 Transformer 모델이 메모리가 부족하거나 불안정한 성능만 달성했다.
- ✓ FlashAttention은 Path-X에서 **61.4**의 정확도를 달성했다.

  추가적으로, Block-Sparse FlashAttention은 Transformers가 시퀀스 길이 64K로 확장되어 Path-256에서 **63.1**의 정확도를 달성했다.

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**

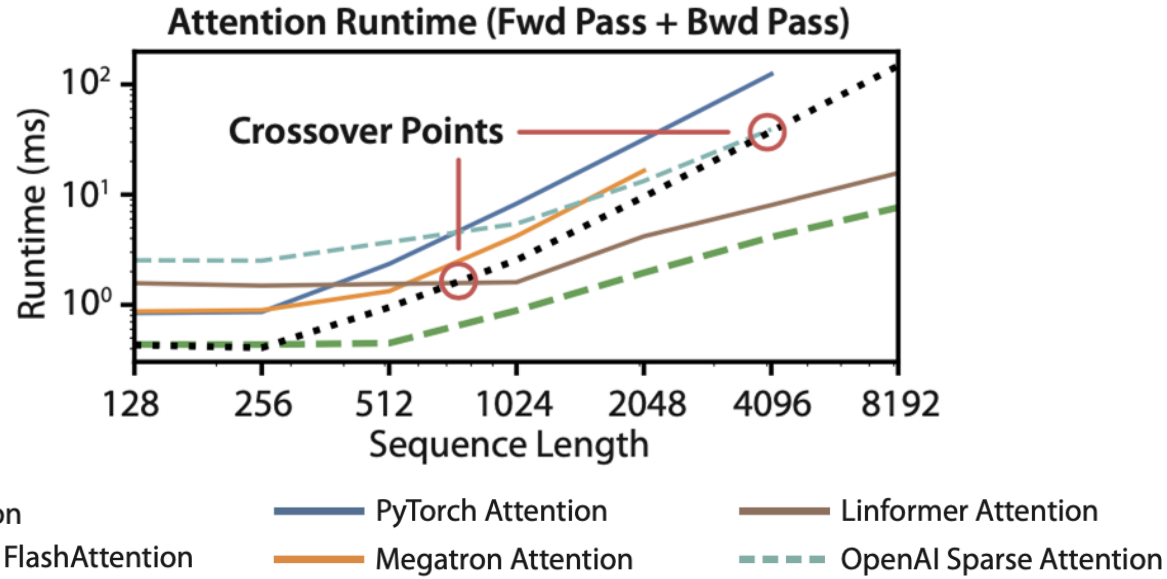
Benchmarking Attention - Runtime

✓ FlashAttention은 메모리 접근이 적기 때문에 짧은 시퀀스에서는

Approximate/Sparse Attention 어텐션보다도 빠르게 실행된다.

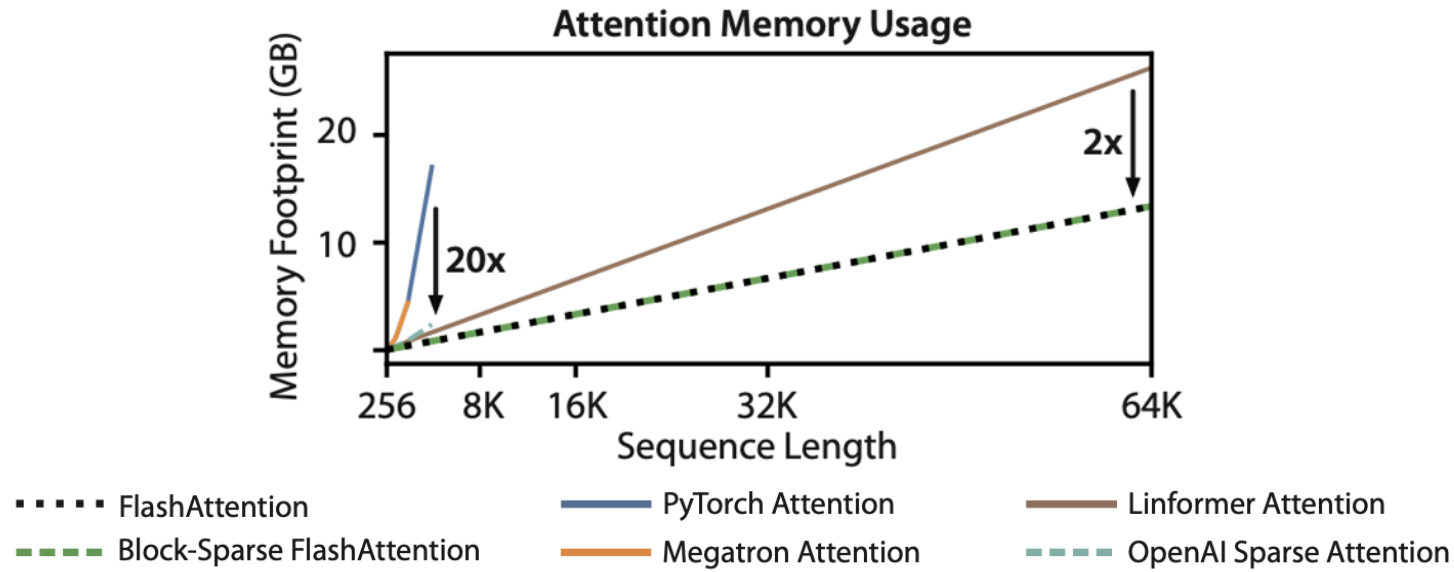
✓ Linformer Attention의 실행 시간은 시퀀스 길이가 512에서 1024 사이일 때 FlashAttention과 교차한다.

✓ 반면, **Block-Sparse FlashAttention은 모든 시퀀스 길이에 걸쳐 빠르다.**



Benchmarking Attention - Memory Footprint

- ✓ FlashAttention과 Block-Sparse FlashAttention은 **같은 메모리 사용량**을 가지며, 이는 **시퀀스 길이에 따라 선형적으로 증가한다.**
- ✓ Linformer를 제외한 모든 알고리즘은 A100 GPU에서 시퀀스 길이가 64K 이전에 메모리가 부족해지지만, FlashAttention은 Linformer보다도 **2배** 더 효율적이다.



Thank You