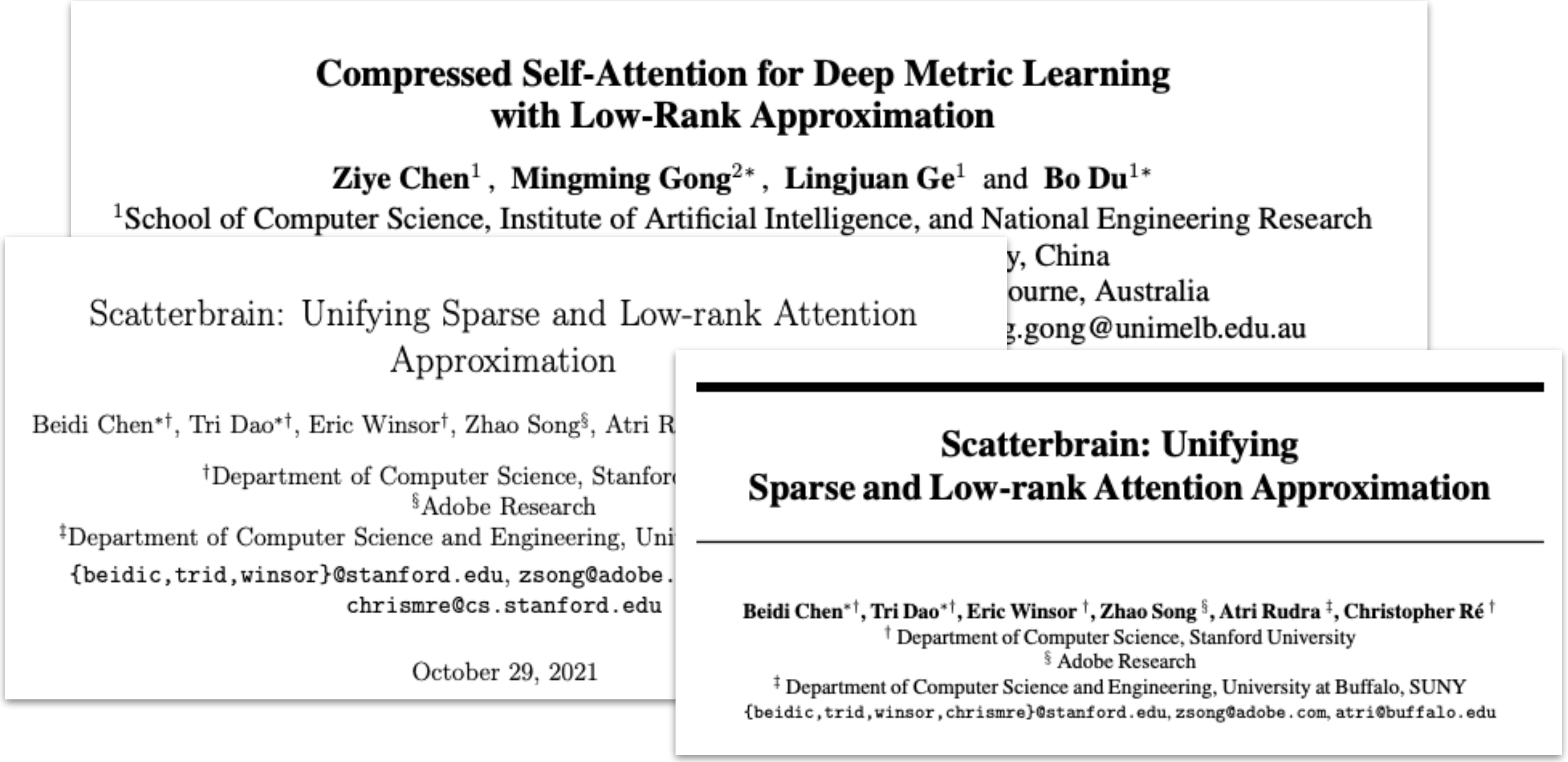
# FlashAttention

**[DIYA NLP Team]**

# FlashAttention: Fast and Memory-Efficient Extract Attention with IO-Awareness

FlashAttention is a new algorithm to speed up attention and reduce its memory footprint wo any approximation.



Compressed Self-Attention for Deep Metric Learning with Low-Rank Approximation

Ziye Chen[1], Mingming Gong[2*], Lingjuan Ge[1] and Bo Du[1*]

[1]School of Computer Science, Institute of Artificial Intelligence, and National Engineering Research

y, China

ourne, Australia

.gong@unimelb.edu.au

Scatterbrain: Unifying Sparse and Low-rank Attention Approximation

Beidi Chen*†, Tri Dao*†, Eric Winsor†, Zhao Song§, Atri R

†Department of Computer Science, Stanfor

§Adobe Research

‡Department of Computer Science and Engineering, Uni

{beidic,trid,winsor}@stanford.edu, zsong@adobe.

chrismre@cs.stanford.edu

October 29, 2021

Scatterbrain: Unifying Sparse and Low-rank Attention Approximation

Beidi Chen*†, Tri Dao*†, Eric Winsor†, Zhao Song§, Atri Rudra‡, Christopher Ré†

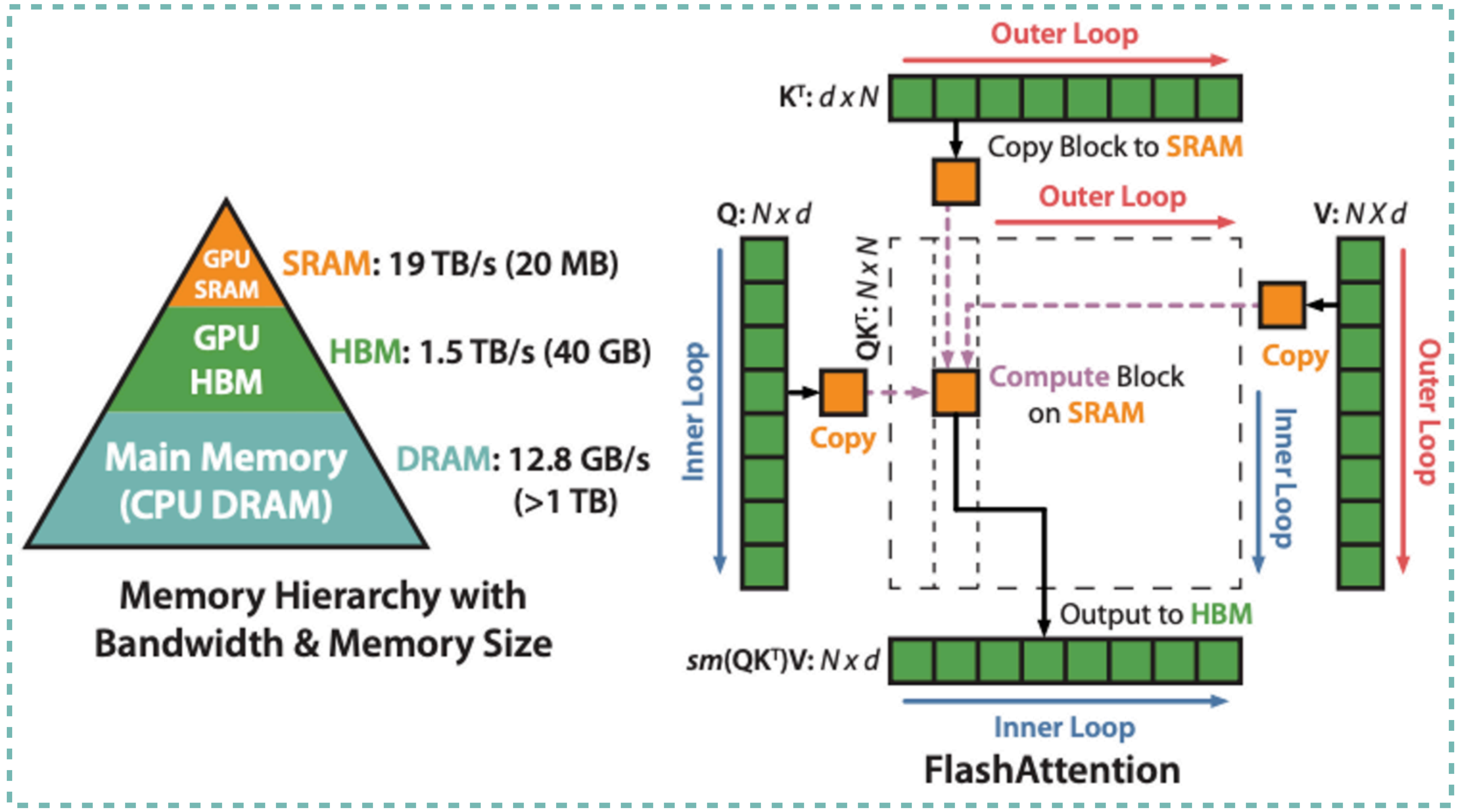
† Department of Computer Science, Stanford University

§ Adobe Research

‡ Department of Computer Science and Engineering, University at Buffalo, SUNY

{beidic,trid,winsor,chrismre}@stanford.edu, zsong@adobe.com, atri@buffalo.edu

Sparse / Low-rank Attention approximation




IO-aware Attention

IO-aware: 서로 다른 수준의 빠른 메모리와 느린 메모리(예: 빠른 GPU on-chip SRAM과 상대적으로 느린 HBM)에 대한 읽기 및 쓰기를 신중하게 고려하는 원칙

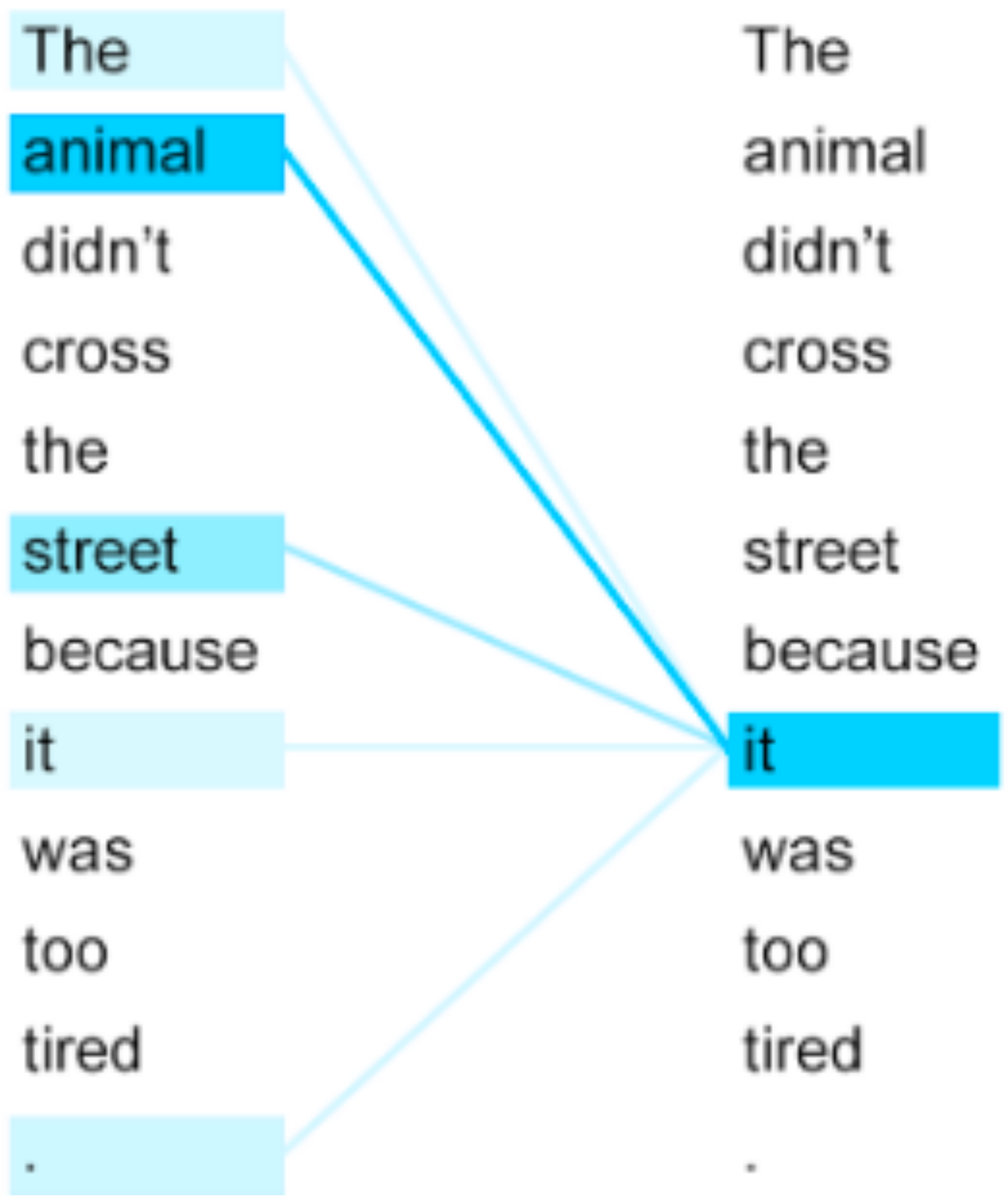
**본 논문에서는 standard attention algorithms에서 IO-aware가 누락되어 있다고 주장**

# Self-Attention

# Attention과 Self-Attention이란?


**Attention (관심법)**

1. 주어진 Query에 대해서 모든 Key와의 유사도를 각각 구함
2. 구해낸 이 유사도를 가중치로 하여 Kye와 맵핑되어있는 각각의 Value에 반영해줌
3. 그리고 유사도가 반영된 Value를 모두 가중합하여 리턴

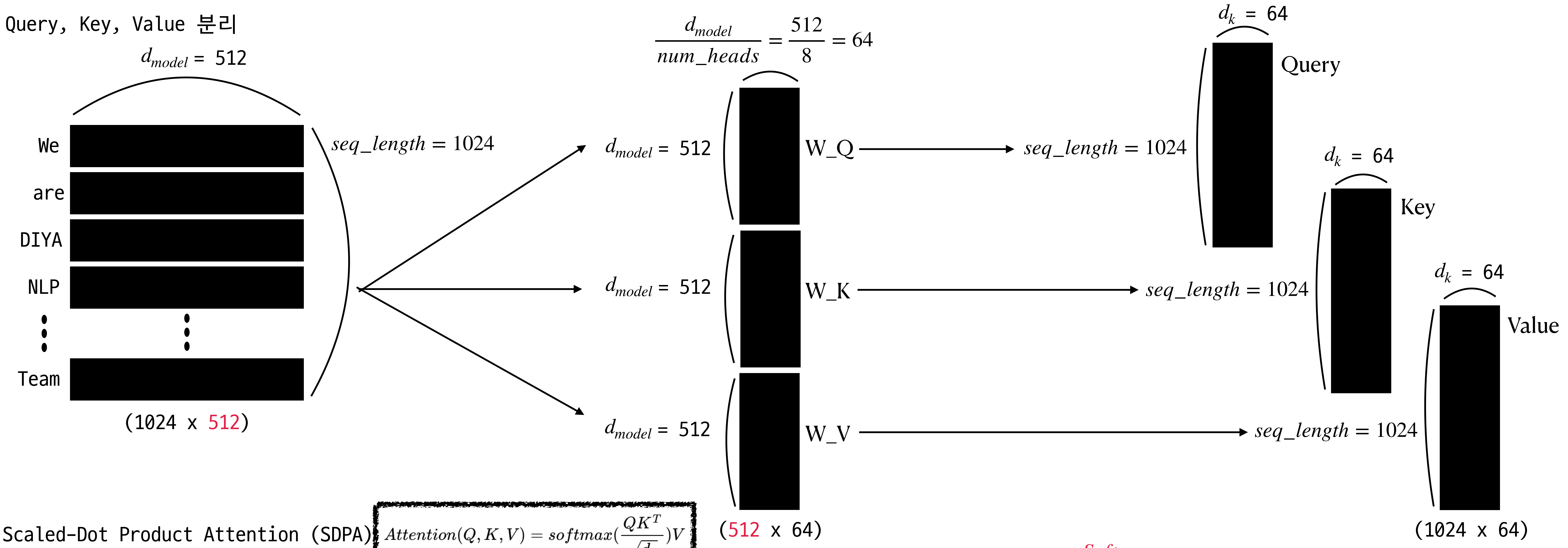



**Self-Attention**

Query, Key, Value를 이용해서 각 단어가 문장 속에서 지닌 전체적인 의미를 파악해보자

1. Attention : https://codingopera.tistory.com/41
2. Self-Attention : https://wikidocs.net/31379

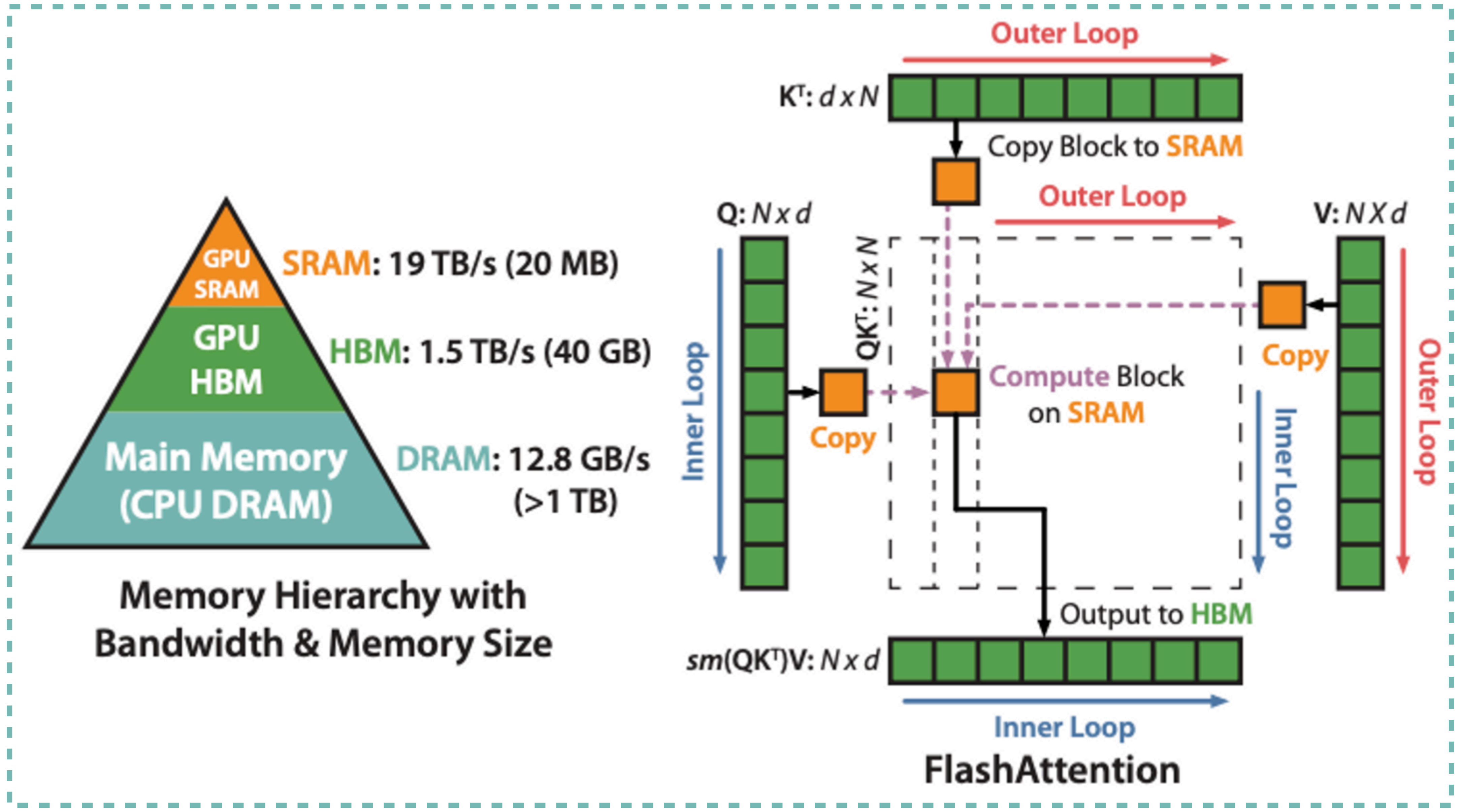
## 1. Query, Key, Value 분리

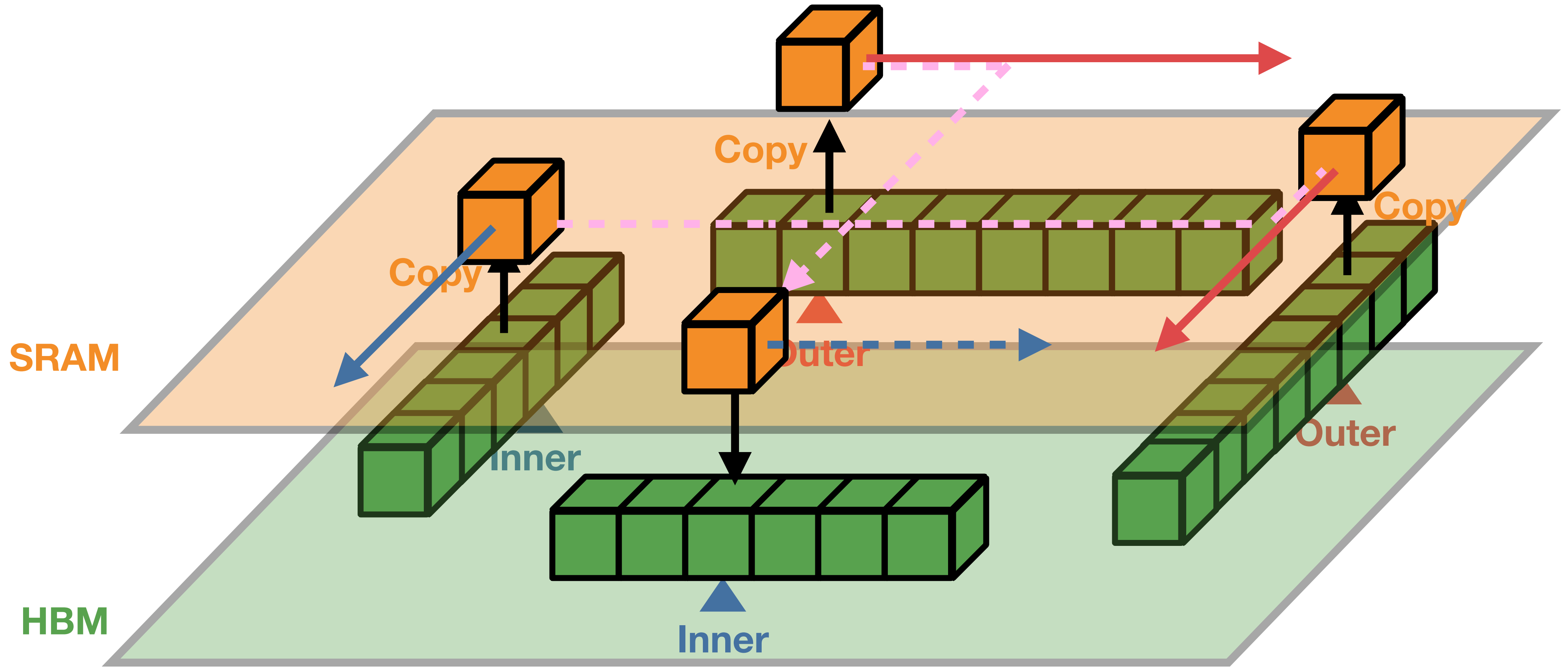



## 2. Scaled-Dot Product Attention (SDPA)

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

# FlashAttention: Fast and Memory-Efficient Extract Attention with IO-Awareness




i) We restructure the attention computation to **split the input into blocks and make several passes over input blocks**, thus incrementally performing the softmax reduction (also known as tiling).

ii) We **store the softmax normalization factor** from the forward pass to quickly recompute attention on-chip in the backward pass, which is faster than the standard approach of reading the intermediate attention matrix from HBM.

# FlashAttention: Fast and Memory-Efficient Extract Attention with IO-Awareness

## Softmax reduction

기존 Softmax: $\sigma(x) = \dfrac{e^{x_i}}{\sum_{j=1}^{K} e^{x_i}}$

$m(x) := \max_i x_i, \; f(x) := [e^{x_1 - m(x)} \ldots e^{x_B - m(x)}], \; l(x) := \sum_i f(x)_i, \; \text{softmax} := \frac{f(x)}{l(x)}$

↑ 값이 너무 커졌을 때 overflow를 방지하기 위해서 max값을 빼줌

하지만 Tiling에서 쪼개서 연산을 하기 때문에 분모의 $\sum_i^K e^{x_i}$ 부분을 계산할 수 없음

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

$$m(x) = m([x^{(1)}, x^{(2)}]) = \max(m(x^{(1)}), m(x^{(2)})),$$

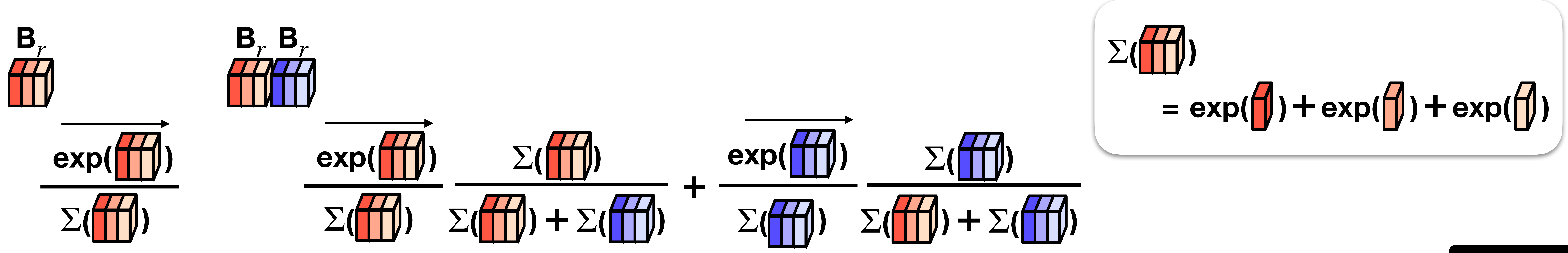
$$f(x) = [e^{x^{(1)} - m(x)} f(x^{(1)}) \; e^{x^{(2)} - m(x)} f(x^{(2)})],$$

$$l(x) = l([x^{(1)}, x^{(2)}]) = e^{x^{(1)} - m(x)} l(x^{(1)}) + e^{x^{(2)} - m(x)} l(x^{(2)}),$$

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

그래서 2개의 Block이 반복하여 들어가면서 Softmax 연산

Therefore if we keep track of some extra statistics $(m(x), \ell(x))$, we can compute softmax one block at a time.



# FlashAttention: Fast and Memory-Efficient Extract Attention with IO-Awareness

Matrices $Q, K, V \in \mathbb{R}^{N \times d}$, SRAM의 크기 M, block 크기 $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$ 설정

HBM에 $O = (0)_{N \times d} \in \mathbb{R}^{N \times d}, l = (0)_N \in \mathbb{R}^N, m = (-\inf)_N \in \mathbb{R}^N$ 미리 초기화

이후 Q는 $T_r = \lceil \frac{N}{B_r} \rceil$의 크기만큼$(B_r, \times d)$, $K, V$는 $T_c = \lceil \frac{N}{B_c} \rceil$만큼$(B_c, \times d)$ block으로 나눈다.

$O, l, m$ 도 $Q$ 와 같은 block 크기로 나눈다.

$1\ to\ T_c\ (j)$ 반복 →쪼개진 k, v에 대해서 모든 q 벡터 iteration

HBM에서 $K_j, V_j$ SRAM으로 이동

$1\ to\ T_r\ (i)$ 반복

$Q_i, O_i, l_i, m_i$ SRAM으로 이동

$S_{ij} = Q_i K_j^T \in \mathbb{R}^{B_r \times B_c}$

$\tilde{m}_{ij} = \text{rowmax}(S_{ij}),\ \tilde{P}_{ij} = \exp(S_{ij} - \tilde{m}_{ij})$ (pointwise), $\tilde{l}_{ij} = \text{rowsum}(\tilde{P}_{ij})$

$m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r},\ l_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} l_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{l}_{ij} \in \mathbb{R}^{B_r}$

$O_i \leftarrow \text{diag}(l_i^{\text{new}})^{-1}(\text{diag}(l_i) e^{m_i - m_i^{\text{new}}} O_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{P}_{ij} V_j)$

$\text{diag}(s)a$: vector s를 행렬 a와 elementwise하게 곱할 수 있음 (block 단위 softmax 연산)

$l_i \leftarrow l_i^{\text{new}},\ m_i \leftarrow m_i^{\text{new}}$

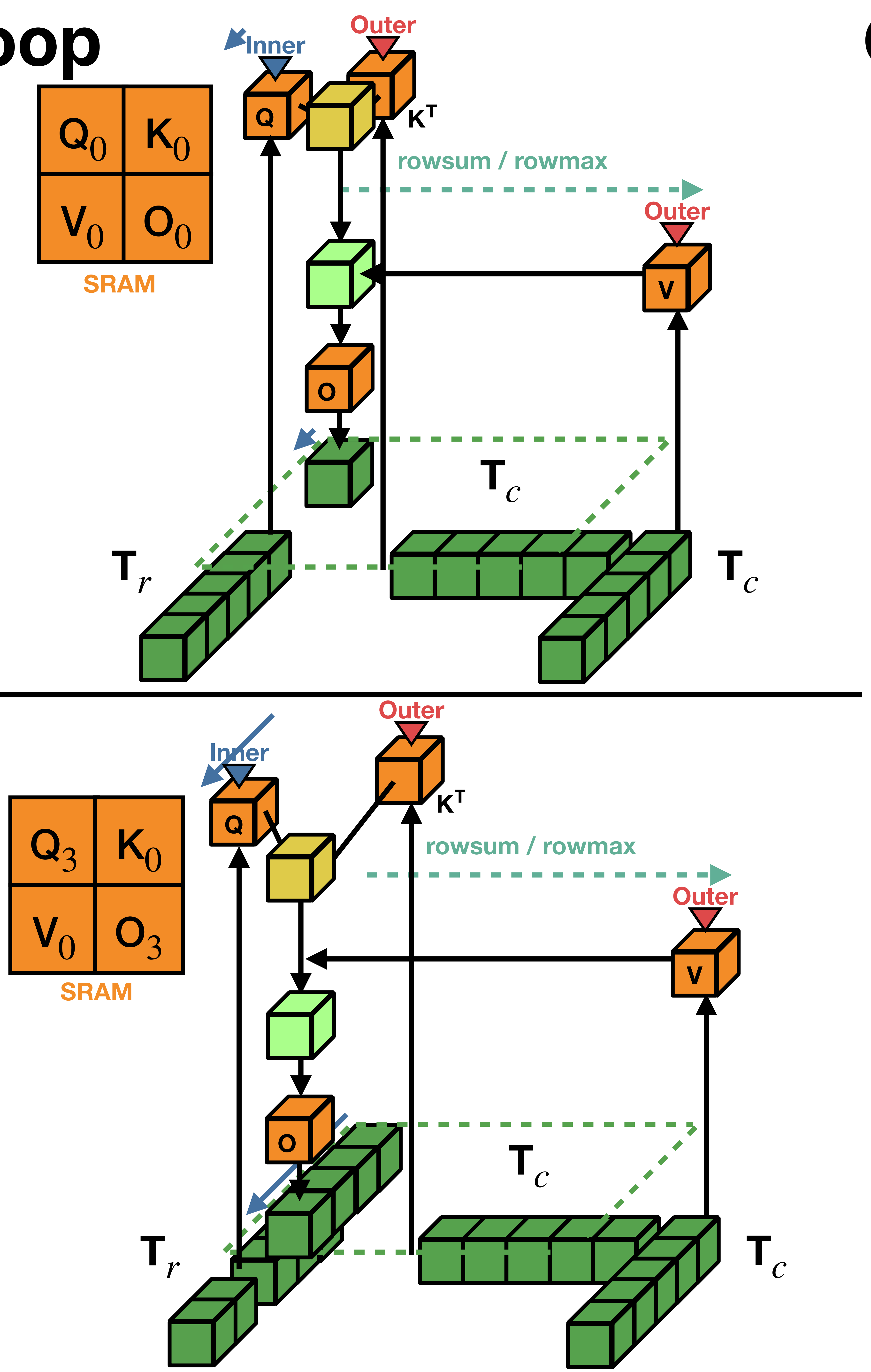
return $O$

**Algorithm 1** FlashAttention

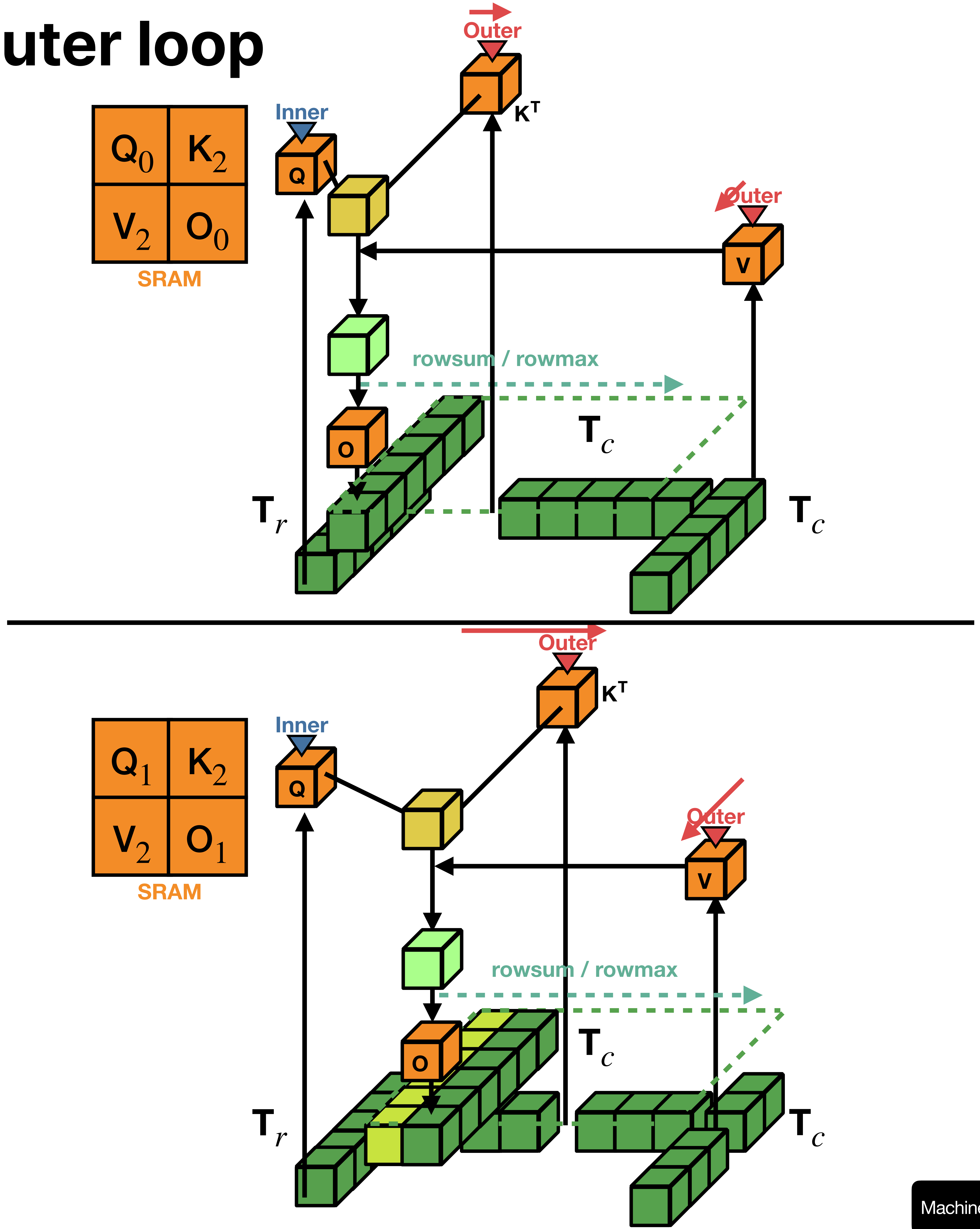
**Require:** Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size $M$.

Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
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 $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ 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 $\ell$ 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 $\mathbf{O}$.

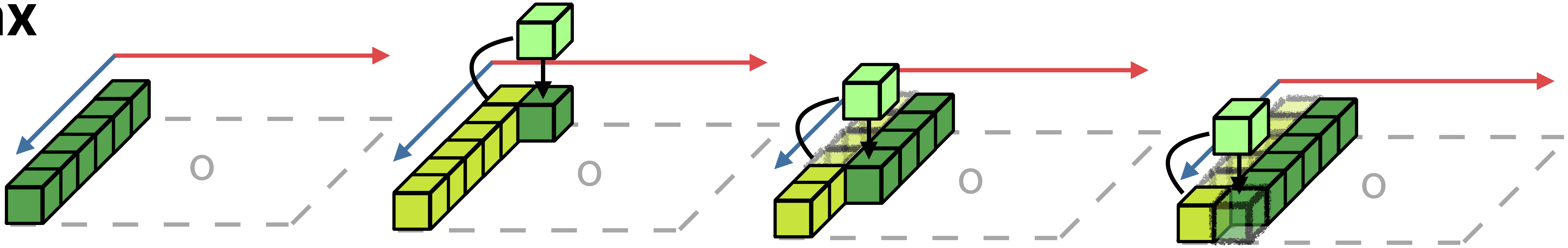
# Inner loop

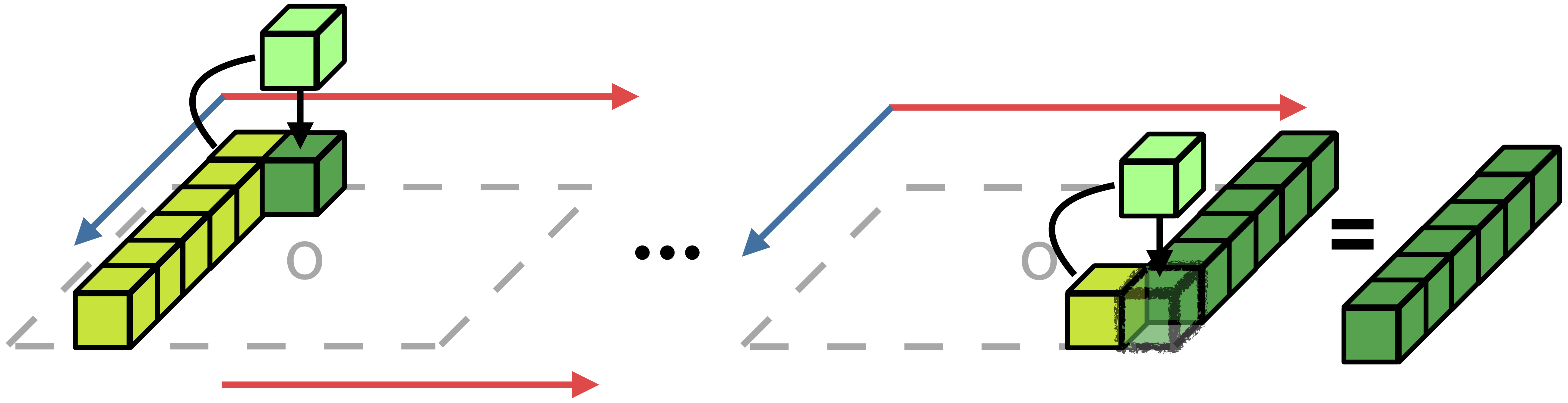



# Outer loop



# Softmax



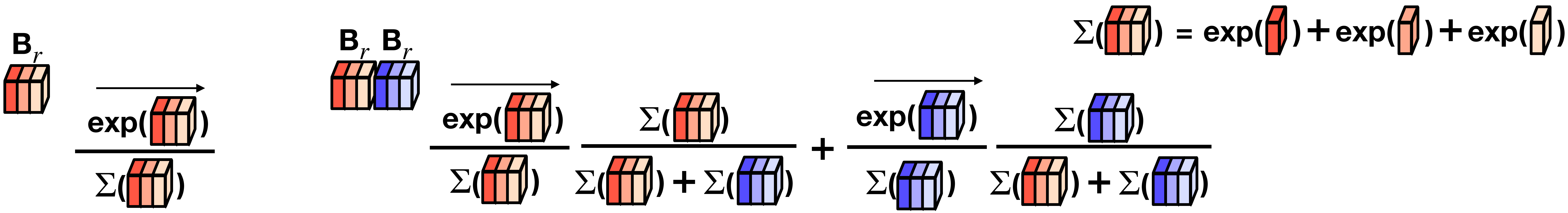


$$m(x) = m([x^{(1)}, x^{(2)}]) = \max(m(x^{(1)}), m(x^{(2)})),$$

$$f(x) = [e^{x^{(1)} - m(x)} f(x^{(1)}) \;\; e^{x^{(2)} - m(x)} f(x^{(2)})],$$

$$l(x) = l([x^{(1)}, x^{(2)}]) = e^{x^{(1)} - m(x)} l(x^{(1)}) + e^{x^{(2)} - m(x)} l(x^{(2)}),$$

$$\mathrm{softmax}(x) = \frac{f(x)}{l(x)}. \; =$$



# FlashAttention: Fast and Memory-Efficient Extract Attention with IO-Awareness

Matrices $Q, K, V \in \mathbb{R}^{N\times d}$, SRAM의 크기 M, block 크기 $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$ 설정

HBM에 $O = (0)_{N\times d} \in \mathbb{R}^{N\times d}, l = (0)_N \in \mathbb{R}^N, m = (-\inf)_N \in \mathbb{R}^N$ 미리 초기화

이후 Q는 $T_r = \lceil \frac{N}{B_r} \rceil$의 크기만큼$(B_r, \times d)$, $K, V$는 $T_c = \lceil \frac{N}{B_c} \rceil$만큼$(B_c, \times d)$ block으로 나눈다.

$O, l, m$ 도 $Q$ 와 같은 block 크기로 나눈다.

$1\ to\ T_c\ (j)$ 반복 →쪼개진 k, v에 대해서 모든 q 벡터 iteration

HBM에서 $K_j, V_j$ SRAM으로 이동

$1\ to\ T_r\ (i)$ 반복

$Q_i, O_i, l_i, m_i$ SRAM으로 이동

$S_{ij} = Q_i K_j^T \in \mathbb{R}^{B_r \times B_c}$

$\tilde{m}_{ij} = \text{rowmax}(S_{ij}),\ \tilde{P}_{ij} = \exp(S_{ij} - \tilde{m}_{ij})$ (pointwise), $\tilde{l}_{ij} = \text{rowsum}(\tilde{P}_{ij})$

$m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r},\ l_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} l_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{l}_{ij} \in \mathbb{R}^{B_r}$

$O_i \leftarrow \text{diag}(l_i^{\text{new}})^{-1}(\text{diag}(l_i) e^{m_i - m_i^{\text{new}}} O_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{P}_{ij} V_j)$

diag$(s)a$: vector s를 행렬 a와 elementwise하게 곱할 수 있음 (block 단위 softmax 연산)

$l_i \leftarrow l_i^{\text{new}},\ m_i \leftarrow m_i^{\text{new}}$

return $O$

**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 $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
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 $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ 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 $\ell$ 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 $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (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 $\mathbf{O}$.

$\lceil (M/4d) \rceil$인 이유: Q, K, V vector 가 d-dimension 벡터이고, 출력차원을 d로 결합하기 때문에 q, k, v, o (4개)로 SRAM을 최대한 사용할 수 있다.

$$
\begin{aligned}
\mathbf{O}^{(j+1)} &= \text{diag}(\ell^{(j+1)})^{-1}(\text{diag}(\ell^{(j)}) e^{m^{(j)} - m^{(j+1)}} \mathbf{O}^{(j)} + e^{\tilde{m} - m^{(j+1)}} \exp(\mathbf{S}_{j:j+1} - \tilde{m}) \mathbf{V}_{j:j+1}) \\
&= \text{diag}(\ell^{(j+1)})^{-1}(\text{diag}(\ell^{(j)}) e^{m^{(j)} - m^{(j+1)}} \mathbf{P}_{:,:j} \mathbf{V}_{:j} + e^{-m^{(j+1)}} \exp(\mathbf{S}_{j:j+1}) \mathbf{V}_{j:j+1}) \\
&= \text{diag}(\ell^{(j+1)})^{-1}(\text{diag}(\ell^{(j)}) e^{m^{(j)} - m^{(j+1)}} \text{diag}(\ell^{(j)}) \exp(\mathbf{S}_{:,:j} - m^{(j)}) \mathbf{V}_{:j} + e^{-m^{(j+1)}} \exp(\mathbf{S}_{j:j+1}) \mathbf{V}_{j:j+1}) \\
&= \text{diag}(\ell^{(j+1)})^{-1}(e^{-m^{(j+1)}} \exp(\mathbf{S}_{:,:j}) \mathbf{V}_{:j} + e^{-m^{(j+1)}} \exp(\mathbf{S}_{j:j+1}) \mathbf{V}_{j:j+1}) \\
&= \text{diag}(\ell^{(j+1)})^{-1}(\exp(\mathbf{S}_{:,:j} - m^{(j+1)}) \mathbf{V}_{:j} + \exp(\mathbf{S}_{j:j+1} - m^{(j+1)}) \mathbf{V}_{j:j+1}) \\
&= \text{diag}(\ell^{(j+1)})^{-1} \left( \exp \left( \begin{bmatrix} \mathbf{S}_{:,:j} & \mathbf{S}_{j:j+1} \end{bmatrix} - m^{(j+1)} \right) \right) \begin{bmatrix} \mathbf{V}_{:j} \\ \mathbf{V}_{j:j+1} \end{bmatrix} \\
&= \text{softmax}(\mathbf{S}_{:j+1}) \mathbf{V}_{:j+1}.
\end{aligned}
$$

# FlashAttention: Fast and Memory-Efficient Extract Attention with IO-Awareness

## Complexity

Standard Attention

$$\Omega(Nd + N^2)$$

$\Omega(Nd + N^2)$ 인 이유: Attention 메커니즘을 구현하는데 필요한 HBM access (I/O 복잡도)를 나타낸 것

('Ω' (빅 오메가, Big Omega) 표기법: 어떤 알고리즘이 특정 입력 크기에 대해 하한(최선))

$\Theta(Nd)$는 $Q, K, V \in \mathbb{R}^{N \times d}$ 차원이기 때문에 각 입력을 읽고 쓰기 위해서는 $Nd$ 개의 데이터가 필요

$\Theta(N^2)$는 $N$개의 $Q, K$를 내적하여 생성된 행렬 $S \in \mathbb{R}^{N \times N}$ 차원이기 때문에 각 $NN$ 개의 데이터가 필요

FlashAttention에서는 불필요!

Flash Attention

$O(N^2 d^2 M^{-1})$ , $d$: head dimension, M: Size of SRAM

$K_j, V_j$ block의 크기는 $B_c \times d$이기 때문에 $B_c$를 올리는 비용은

$$B_c d = O(M) \Leftrightarrow B_c = O(\frac{M}{d}).$$

$Q_j$ block의 크기는 $B_r \times d$이기 때문에 $B_r$를 올리는 비용은

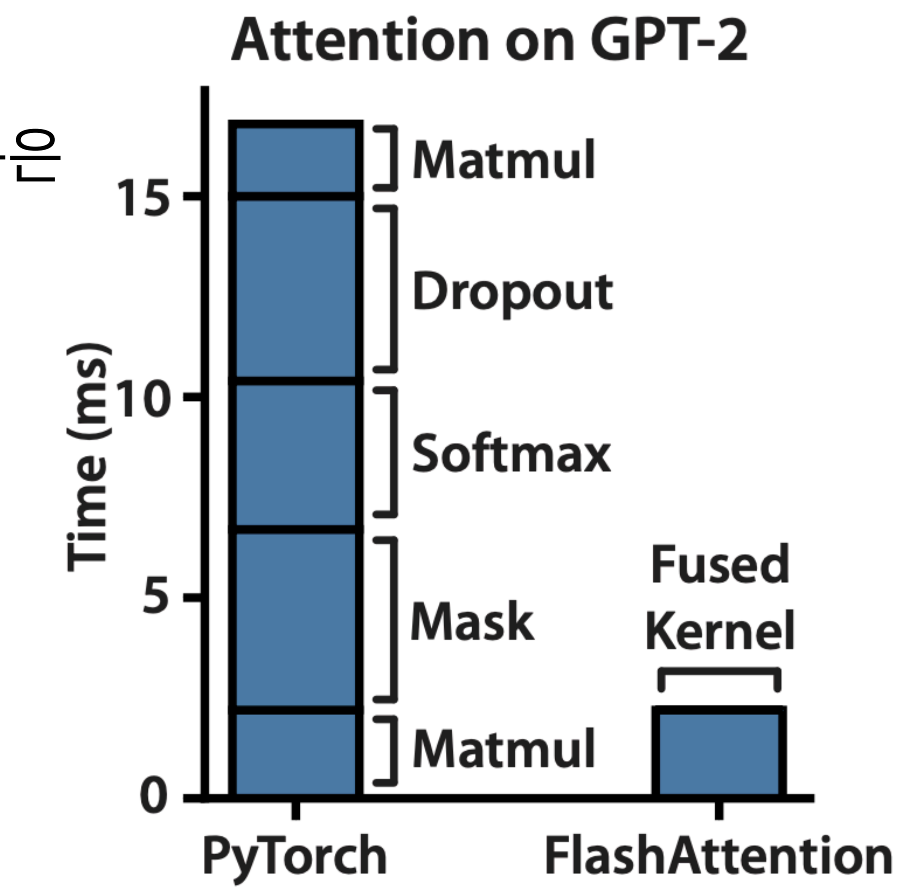
$$B_r d = O(M) \Leftrightarrow B_r = O(\frac{M}{d}).$$

$$B_c = \Theta(\frac{M}{d}),\ B_r = \Theta(\min(\frac{M}{d}, \frac{M}{B_c})) = \Theta(\min(\frac{M}{d}, d))$$

$$T_c = \frac{N}{B_c} = \Theta(\frac{Nd}{M})$$

$$\Theta(NdT_c) = \Theta(\frac{N^2 d^2}{M}).$$

Q,K 연산 곱하기 Tiling

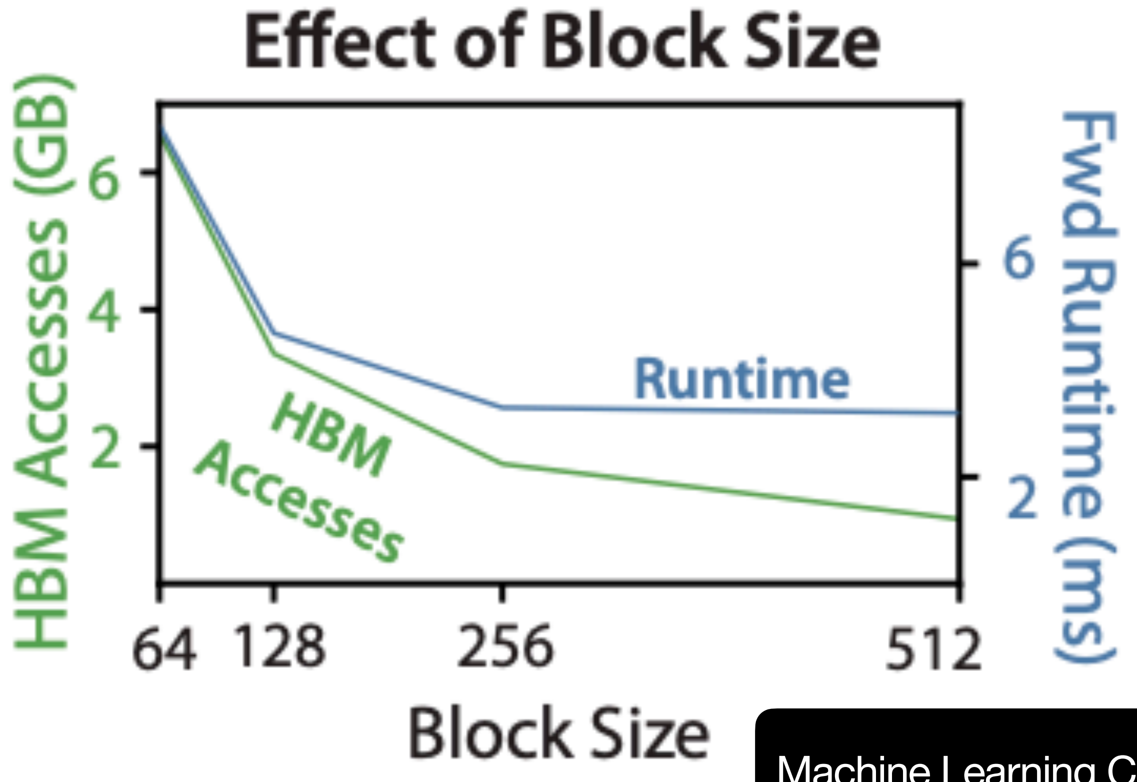



| Model implementations | OpenWebText (ppl) | Training time (speedup) |
|---|---|---|
| GPT-2 small - Huggingface [87] | 18.2 | 9.5 days (1.0×) |
| GPT-2 small - Megatron-LM [77] | 18.2 | 4.7 days (2.0×) |
| GPT-2 small - FlashAttention | 18.2 | **2.7 days (3.5×)** |
| GPT-2 medium - Huggingface [87] | 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×)** |

Huggingface, Megatron(Parallel) GPT-2보다 훨씬 더 빠른 속도를 보여줌

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

FlashAttention은 Standard Attention에 비해 속도, 용량, 시간 모두 절약




$B_c$의 크기에 따라 크게 감소하다가 256개 부터는 병목현상 발생

# Implementation

**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 $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
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 $T_r = \lceil \frac{N}{B_r} \rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \lceil \frac{N}{B_c} \rceil$ 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 $\ell$ 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 $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\text{new}})^{-1}(\mathrm{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ 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 $\mathbf{O}$.

Q = ( N, d )  K = ( N, d )  V = ( N, d )  O = ( N, d )

Tr = [ N / Br ], Tc = [ N / Bc ]

Q = ( Tr, Br, d )  K = ( Tc, Bc, d )  V = ( Tc, Bc, d )  O = ( Tr, Br, d )

**Loop in Tc:**
K, V = ( Bc, d ) each

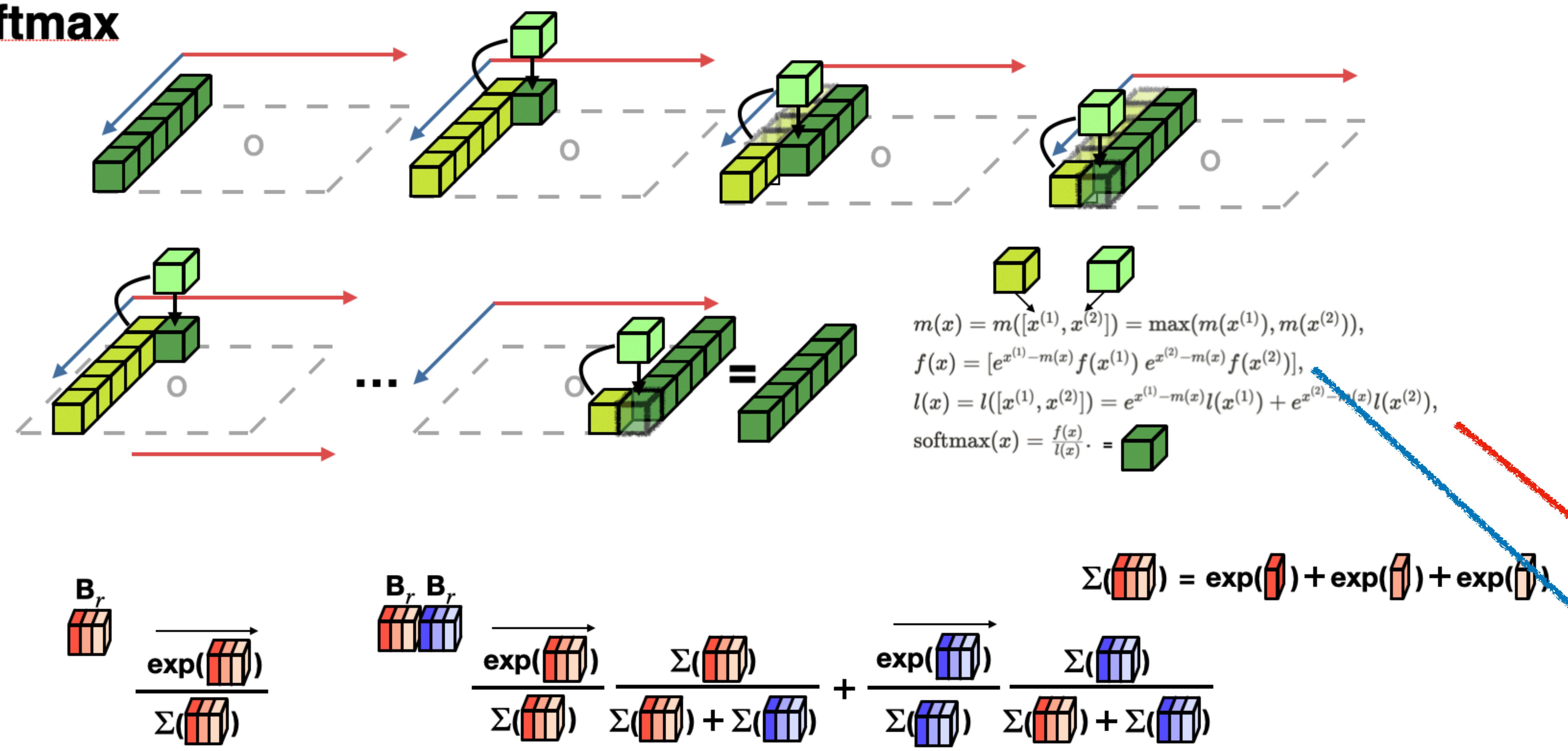
**Loop in Tr:**
Q, O = ( Br, d ) each
S_ij = ( Br, Bc )
mhat_ij = Br,  Phat_ij = ( Br, Bc ),  lhat_ij =Br

{Line 12 : O overwrite => new O which is ( Br, d )}

Br —Row-wise→ ( Br ——→ Br → ( Br, d )  Br → ( Br, Bc ) ( Bc, d ) )

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

# Softmax




```python
# 5. Loop in Tc
for j in range(T_c):
    # 6. Load K_j, V_j from HBM to SRAM
    K_j, V_j = K_blocks[j], V_blocks[j]

    # 7. Loop in Tr
    for i in range(T_r):
        # 8. Load Q i, O i, ℓ i, m i from HBM to on-chip SRAM.
        # Q_i, O_i, l_i, m_i = Q_blocks[i], O_tiles[i], l_tiles[i], m_tiles[i]
        Q_i, O_i, l_i, m_i = Q_blocks[i], O_tiles[i], l_tiles[i], m_tiles[i]

        # 9. Compute Sij = Q_iK^T_j which return Br X Bc
        S_ij = np.dot(Q_i, K_j.T)  # Orginally on SRAM
        # Proof of shape
        # print(f"S_ij.shape:{S_ij.shape} vs B_r X B_c : {B_r,B_c}")

        # 10. Compute i) mhat_ij = rowmax(S_ij) return B_r,
        #            ii) P_ij = e^(S_ij - mhat_ij) return B_r X B_c,
        #           iii) lhat_ij = rowsum(P_ij) return B_r

        mhat_ij = np.max(S_ij, axis=1)  # max per column
        P_ij = np.exp(S_ij - mhat_ij[:, np.newaxis])
        lhat_ij = np.sum(P_ij, axis=1)

        # 11. Compute i) mnew_i = max(m_i, mhat_ij) return B_r,
        #            ii) lnew_i = exp(m_i - mnew_i)l_i + exp(mhat_ij-mnew_i)lhat_ij return B_r
        mnew_i = np.maximum(m_i, mhat_ij)
        lnew_i = np.exp(m_i - mnew_i) * l_i + np.exp(mhat_ij - mnew_i) * lhat_ij

        # 12. O_i = diag(lnew_i)^-1 * (diag(l_i)exp(m_i - mnew_i)*O_i + exp(mhat_ij-mnew_i)lhat_ij) return B_r
        # TODO
        functionF = (l_i * np.exp(m_i - mnew_i))[:,np.newaxis] * O_i + (np.exp(mhat_ij - mnew_i)[:,np.newaxis]) * P_ij @ V_j
        functionL = lnew_i[:, np.newaxis]
        O_tiles[i] = functionF/functionL
        # 13. override
        l_tiles[i], m_tiles[i] = lnew_i, mnew_i
# 14. O = concatenate(O_1, O_2, ... , O_T_r) return N X d
out = np.concatenate(O_tiles, axis=0)
```