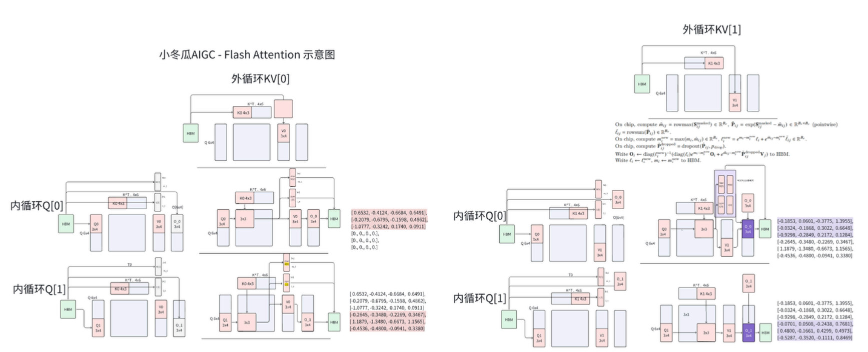


【手撕LLM-Flash Attention】从softmax说起，保姆级超长文！！

小冬瓜AIGC
原创课程➡️ 公众号：手撕LLM

> 我是小冬瓜AIGC，原创超长文知识分享，原创课程已帮助多名同学速成上岸LLM赛道。
> 研究方向：LLM、RL、RLHF、AIGC和Agent。

目录

导读：

本文已写有半年之久，个人建议的是，只要理解了算法的动机(memory efficient)，就知道为什么需要Flash Attention。 ***算法不建议在Tri Dao原论文上推， 忘掉原论文，step-by-step数学推导更容易通关***。

建议推导FA前掌握Online Softmax：

阅读完本篇畅读Flash Attention2:

1 论文解析

1.1 Flash Attention 背景

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

- `Transformers` 计算长文本时**平方时间复杂度**，很耗时很耗显存
- 有改进的方法降低计算复杂度，没太大用，原因**缓存存取很耗时**
- `FlashAttention` 优化了显存存取，**非优化计算复杂度**

Abstract

Transformers are slow and memory-hungry on long sequences, since the time and memory complexity of self-attention are quadratic in sequence length. Approximate attention methods have attempted to address this problem by trading off model quality to reduce the compute complexity, but often do not achieve wall-clock speedup. We argue that a missing principle is making attention algorithms *IO-aware*—accounting for reads and writes between levels of GPU memory. We propose FlashAttention, 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. We analyze the IO complexity

1.2 Flash Attention性能

Flash Attention是可以用于训练之中的，他实现了注意力Forward和Backward的加速计算。

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×)**

1.3 算法动机

- HBM（High Bandwidth Memory）比如A100-40GB版本里的 `HBM` 就是显存40GB
- `GPU` 计算实际工作在 SRAM（Static Random-Access Memory）
- `SRAM` 19TB/S比 `HBM` 1.5TB/S 计算速度快12.67倍，但只有20MB可以使用
- 目的： `SRAM <--> HBM` 为耗时瓶颈，传统 `Attention` 7次交换，目标在20MB的 `SRAM` 中不做过多内存交换实现 `Attention` 计算
- 对于 `NxN` 矩阵计算 `Attention` ，在 `SRAM` 不交换到 `HBM` ，加速 `7.6x`

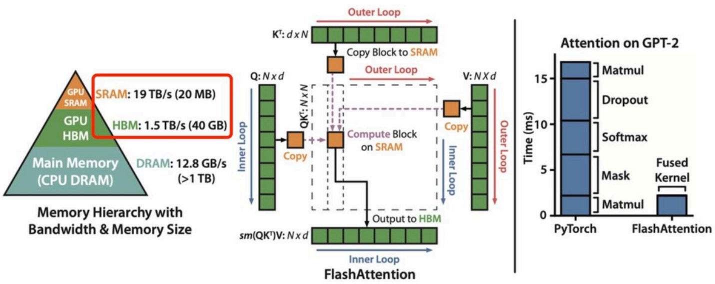
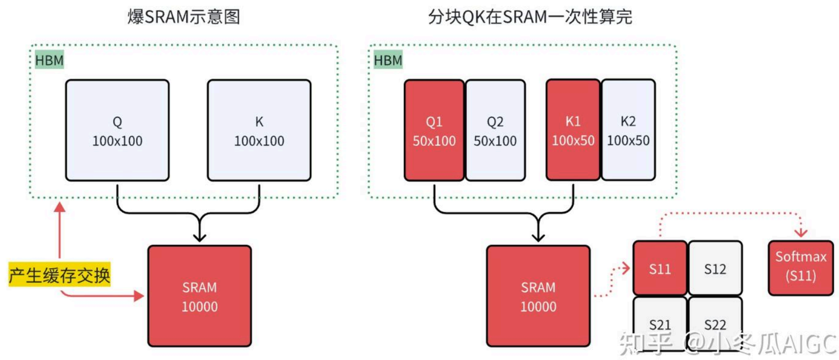


Figure 1: **Left:** FlashAttention uses tiling to prevent materialization of the large $N \times N$ attention matrix (dotted box) on (relatively) slow GPU HBM. In the outer loop (red arrows), FlashAttention loops through blocks of the **K** and **V** matrices and loads them to fast on-chip SRAM. In each block, FlashAttention loops over blocks of **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.

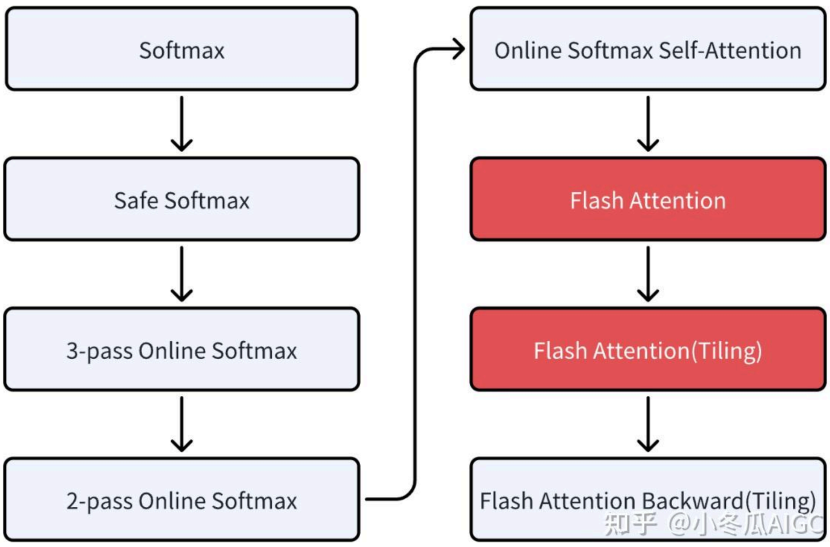
1.4 SRAM爆缓存示例

- 假设 `SRAM` 能计算 `1000` 个数据
- 单一向量 `Q[100,100]` ，`K[100,100]` ，此时需要加载 `2x100x100 = 20000` 个数据 > `10000` (`SRAM`)，此时计算将会出现对于HBM的write/read
- 将split `Q [100,100] -> {Q1[50, 100], Q2[50, 100]}`
- 将split `K [100,100] -> {K1[50, 100], K2[50, 100]}`
- 此时 `score_ij = Qi Kj^T` ， 在 `SRAM` 一次性算完块状的 `score`



2. Flash Attenion算法原理

2.1 Flash Attention Step-by-Step



2.2 Softmax

$$\begin{align} \text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \end{align}$$

```python
import torch
print('torch 手撕')
A = torch.randn(2, 6)
A_exp = torch.exp(A)
A_sum = torch.sum(A_exp, dim=1).unsqueeze(1)
P = A_exp / A_sum #广播
print(A)
print(P)
```

结果

```
tensor([[ 1.0668, -0.3969, -0.2226,  0.7207,  1.0509, -1.0740],
        [ 0.6774,  1.0916, -1.8402, -1.0806,  0.9309,  2.4612]])
tensor([[0.3016, 0.0698, 0.0831, 0.2133, 0.2968, 0.0355],
        [0.0999, 0.1512, 0.0081, 0.0172, 0.1288, 0.5948]])
```

2.3 Safe Softmax

原始 softmax 数值不稳定，改写成 Safe Softmax 版本

\begin{align} \text{softmax}(x_i) = \frac{e^{x_i - max(x)}}{\sum_{j=1}^{n} e^{x_j-max(x)}}
\end{align}

2.4 Online Softmax 3-pass/2-pass

> Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. CoRR, abs/1805.02867, 2018.

参考ref 实现**3次**循环的 `online softmax`：

\begin{align} for i\rightarrow 1,N \nonumber\\ m_i &\leftarrow \max(m_{i-1}, x_i) \\ for i\rightarrow 1,N \nonumber\\ d_i &\leftarrow d_{i-1}+e^{x_i- \color {red}{m_N}} \\ for i\rightarrow 1,N \nonumber\\ a_i &\leftarrow e^{x_i-m_N}/d_N \end{align}

将d'_i的计算改成***迭代形式***

\begin{align} d_i' &=\sum^i_{j=1}e^{x_j-m_i} \\ &=(\sum^{i-1}_{j=1}e^{x_j-m_i})+e^{x_i-m_i} \\ &= (\sum^{i-1}_{j=1}{e^{x_j-m_{i-1}}})e^{m_{i-1}-m_i}+e^{x_i-m_i}} \\ &=d'_{i-1} e^{m_{i-1}-m_{i}}+e^{x_i-m_i} \end{align}

此时得到***2-Pass Online Softmax***

Algorithm 3-pass online softmax

$$
\begin{aligned}
&m_0 = -\inf \\
&d_0 = 0.0 \\
&for\ i \rightarrow 1, N \\
&m_i \leftarrow \max(m_{i-1},\ x_i) \\
&for\ i \rightarrow 1, N \\
&d_i \leftarrow d_{i-1} + e^{x_i - m_N} \\
&for\ i \rightarrow 1, N \\
&a_i \leftarrow e^{x_i - m_N} / d_N
\end{aligned}
$$

Algorithm 2-pass online softmax

$$
\begin{aligned}
&m_0 = -\inf \\
&d_0 = 0.0 \\
&for\ i \rightarrow 1, N \\
&m_i \leftarrow \max(m_{i-1},\ x_i) \\
&d'_i \leftarrow d'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i} \\
&for\ i \rightarrow 1, N \\
&a_i \leftarrow e^{x_i - m_N} / d'_N
\end{aligned}
$$

知乎 @小冬瓜AIGC

此时2个循环搞定online softmax

```python
# online SoftMax 2-pass
import torch

N = 6
m = torch.tensor(-1000.0)
d = 0
x = torch.randn(N)
a = torch.zeros(N)
print('x:', x)

for i in range(N):
    m_pre = m
    m = torch.max(m, x[i])
    d = d * (m_pre - m).exp() + (x[i] - m).exp()

for i in range(N):
    a[i] = (x[i]-m).exp() / d

print('online softmax a:',a)
print(torch.sum(a))
```

结果

```
x: tensor([-1.0990,  0.1895,  0.3930,  1.5720,  1.0603, -0.7564])
online softmax a: tensor([0.0298, 0.1080, 0.1323, 0.4302, 0.2579, 0.0419])
tensor(1.)
```

2.5 Flash Attention By Online Softmax （tiling）

2.5.1 Algorithm Multi-pass Self-Attention

基于 2-pass online softmax 可以写出 2-pass 的 Self-Attention

\begin{align} for i\rightarrow 1,N \nonumber \\ x_i &\leftarrow Q[k,:]K^T[:,i] \\ m_i &\leftarrow \max(m_{i-1}, x_i) \\ d'_i &\leftarrow d'_{i-1}e^{m_{i-1}-m_i}+e^{x_i-m_i} \\ for i\rightarrow 1,N \nonumber \\ a_i &\leftarrow \frac{e^{x_i-m_N}}{d'_N} \\ o_i &\leftarrow o_{i-1}+a_i V[i,:] \end{align}

2.5.2 Algorithm Flash-Attention

进入 Flash Attention 正文！！！！！！！！！！！！！！！！！！！

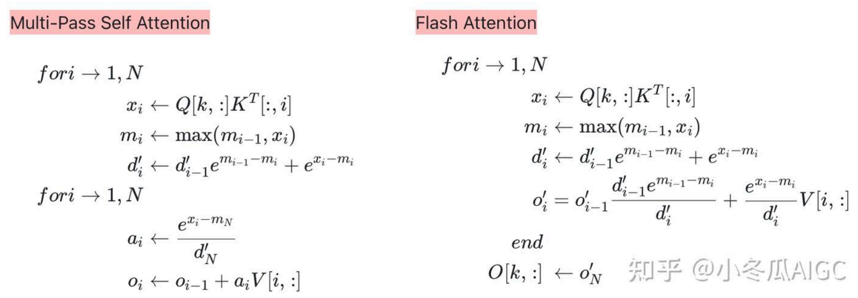
首先将系数d改成迭代形式

\begin{align} o_i &\leftarrow o_{i-1}+\frac{e^{x_i-m_N}}{d'_N} V[i,:] \\ o_i &:= \sum^i_{j=1}(\frac{e^{x_j-m_N}}{\color{red}{d'_N}} V[j,:])\\ o'_i &= (\sum^i_{j=1}\frac{e^{x_j-m_i}}{\color{red}{d'_i}} V[j,:]) \end{align}

将O_i改写成迭代形式

\begin{align} o'_i &= \sum^i_{j=1}\frac{e^{x_j-m_i}}{d'_i} V[j,:] \\ &= (\sum^{i-1}_{j=1}\frac{e^{x_j-m_i}}{d'_i}V[j,:]) + \frac{e^{x_i-m_i}}{d'_i}V[i,:] \\ &= (\sum^{i-1}_{j=1}\frac{e^{x_j-m_{i-1}}}{d'_{i-1}}\frac{e^{x_j-m_i}}{e^{x_j-m_{i-1}}} \frac{d'_{i-1}}{d'_i}V[j,:]) + \frac{e^{x_i-m_i}}{d'_i}V[i,:] \\ &= (\sum^{i-1}_{j=1}\frac{e^{x_j-m_{i-1}}}{d'_{i-1}}V[j,:])\frac{e^{x_j-m_i}}{e^{x_j-m_{i-1}}} \frac{d'_{i-1}}{d'_i} + \frac{e^{x_i-m_i}}{d'_i}V[i,:] \\ &= (\sum^{i-1}_{j=1}\frac{e^{x_j-m_{i-1}}}{d'_{i-1}}V[j,:]){e^{m_{i-1}-m_i}} \frac{d'_{i-1}}{d'_i} + \frac{e^{x_i-m_i}}{d'_i}V[i,:] \\ &= o'_{i-1} \frac{d'_{i-1}{e^{m_{i-1}-m_i}}}{d'_i} + \frac{e^{x_i-m_i}}{d'_i}V[i,:] \\ &= o'_{i-1} \frac{d'_{i-1}{e^{m_{i-1}-m_i}}}{d'_i} + e^{m_j-m_i} \frac{e^{x_i-m_j}}{d'_i}V[i,:] \end{align}

此时就得到 Flash Attention 的 ***one-pass 迭代形式***



2.5.3 Algorithm Flash-Attention(Tiling)

当有多条数据时可进一步改写，得到最终的 Flash Attention 形式，源码基于以下实现。

\begin{align} for i\rightarrow1 \#tiles \nonumber \\ x_i &\leftarrow Q[k,:]K^T[:,(i-1)b:ib] \nonumber\\ m_i^{local} &= \max_{j=1}^b(x_i[j]) \nonumber \\ m_i &\leftarrow \max(m_{i-1}, m_i^{local}) \nonumber \\ d'_i &\leftarrow d'_{i-1}e^{m_{i-1}-m_i}+\sum^b_{j=1}e^{x_i[j]-m_i} \nonumber \\ o'_{i} &= o'_{i-1} \frac{d'_{i-1}{e^{m_{i-1}-m_i}}}{d'_i} + \sum^b_{j=1}\frac{e^{x_i[j]-m_i}}{d'_i}V[j+(i-1)b,:] \nonumber \\ end \nonumber\\ O[k,:] &\leftarrow o'_{N} \nonumber \end{align}

注意：FlashAttention稍微多加一项

\begin{align} e^{m_{ij}-m_{new}}P_{ij}&=e^{m_{ij}-m_{new}} e^{x_i-m_{ij}} \\ &=e^{x_i-m_{new}} \end{align}

3 Flash Attention源码

论文具有详细的 `Flash Attention Forward` 流程， 本人基于此补充示意图和 简易版本代码

3.1 Flash Attention Forward

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

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.

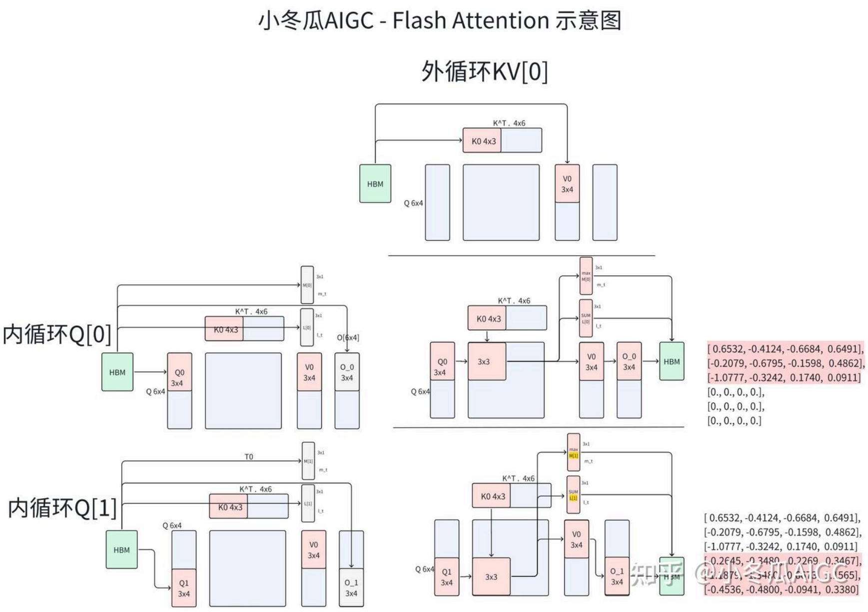
  end for

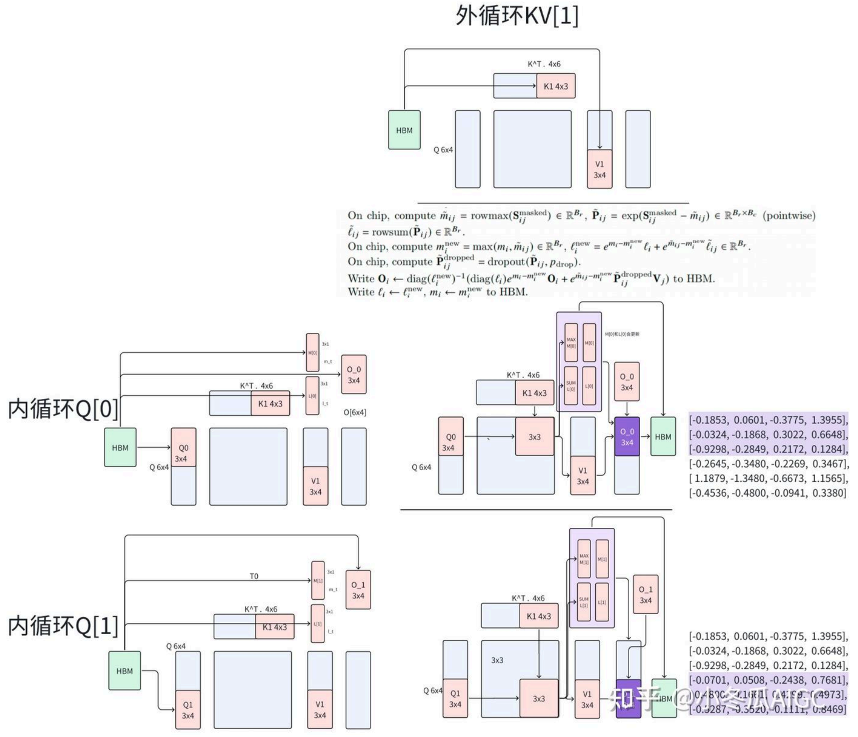
end for

Return $\mathbf{O}, \ell, m, \mathcal{R}$.

3.2 算法流程和HBM<->SRAM交换示意

学习 `Flash Attention` 一定要清楚数据的流转





3.3 手撕 FLash Attention

参考git: shreyansh26/FlashAttention-PyTorch ，

改写了纯 CPU 版本, 去除 scaled 、 mask 、 dropout 等代码，更加简洁

```
import torch

NEG_INF = -1e10  # -infinity
EPSILON = 1e-10

Q_LEN = 6
K_LEN = 6
Q_BLOCK_SIZE = 3 #
KV_BLOCK_SIZE = 3
Tr = Q_LEN // Q_BLOCK_SIZE
Tc = K_LEN // KV_BLOCK_SIZE

Q = torch.randn(1, 1, Q_LEN, 4, requires_grad=True).to(device='cpu')
K = torch.randn(1, 1, K_LEN, 4, requires_grad=True).to(device='cpu')
V = torch.randn(1, 1, K_LEN, 4, requires_grad=True).to(device='cpu')
O = torch.zeros_like(Q, requires_grad=True)
l = torch.zeros(Q.shape[:-1])[..., None]
m = torch.ones(Q.shape[:-1])[..., None] * NEG_INF

Q_BLOCKS = torch.split(Q, Q_BLOCK_SIZE, dim=2)
K_BLOCKS = torch.split(K, KV_BLOCK_SIZE, dim=2)
V_BLOCKS = torch.split(V, KV_BLOCK_SIZE, dim=2)
O_BLOCKS = list(torch.split(O, Q_BLOCK_SIZE, dim=2))
l_BLOCKS = list(torch.split(l, Q_BLOCK_SIZE, dim=2))
m_BLOCKS = list(torch.split(m, Q_BLOCK_SIZE, dim=2))

for j in range(Tc):
    Kj = K_BLOCKS[j]
    Vj = V_BLOCKS[j]
    for i in range(Tr):
        Qi = Q_BLOCKS[i]
        Oi = O_BLOCKS[i]
        li = l_BLOCKS[i]
        mi = m_BLOCKS[i]

        S_ij
```

```
        m_block_ij, _ = torch.max(S_ij, dim=-1, keepdims=True)
        P_ij = torch.exp(S_ij - m_block_ij)
        l_block_ij = torch.sum(P_ij, dim=-1, keepdims=True) + EPSILON
        mi_new = torch.maximum(m_block_ij, mi)
        P_ij_Vj = P_ij @ Vj

        li_new = torch.exp(mi - mi_new) * li  \
               + torch.exp(m_block_ij - mi_new) * l_block_ij

        O_BLOCKS[i] = (li / li_new) * torch.exp(mi - mi_new) * Oi \
                    +(torch.exp(m_block_ij - mi_new) / li_new) * P_ij_Vj
        print(f'-----------Attn : Q{i}xK{j}---------')
#         print(O_BLOCKS[i].shape)
        print(O_BLOCKS[0])
        print(O_BLOCKS[1])
        print('\n')

        l_BLOCKS[i] = li_new
        m_BLOCKS[i] = mi_new

O = torch.cat(O_BLOCKS, dim=2)
l = torch.cat(l_BLOCKS, dim=2)
m = torch.cat(m_BLOCKS, dim=2)
print(O)
```

结果

```
-----------Attn : Q0xK0---------
tensor([[[[-0.3828,  0.3858,  0.0073, -0.3497],
          [ 0.1703,  0.0784, -0.2246, -0.3114],
          [ 0.2711,  0.4141,  0.6492, -0.1008]]]], grad_fn=<AddBackward0>)
tensor([[[[0., 0., 0., 0.],
          [0., 0., 0., 0.],
          [0., 0., 0., 0.]]]], grad_fn=<SplitBackward0>)


-----------Attn : Q1xK0---------
tensor([[[[-0.3828,  0.3858,  0.0073, -0.3497],
          [ 0.1703,  0.0784, -0.2246, -0.3114],
          [ 0.2711,  0.4141,  0.6492, -0.1008]]]], grad_fn=<AddBackward0>)
tensor([[[[ 0.5309,  0.5167,  1.1180,  0.0456],
          [-0.1518, -0.0681, -0.8508, -0.5029],
          [ 0.3779,  0.3903,  0.6877, -0.0749]]]], grad_fn=<AddBackward0>)


-----------Attn : Q0xK1---------
tensor([[[[-0.3839,  0.3844,  0.0229, -0.3405],
          [-0.0877,  0.1931,  2.7363,  0.8162],
          [-0.1728,  0.3630,  1.7052,  0.7037]]]], grad_fn=<AddBackward0>)
tensor([[[[ 0.5309,  0.5167,  1.1180,  0.0456],
          [-0.1518, -0.0681, -0.8508, -0.5029],
          [ 0.3779,  0.3903,  0.6877, -0.0749]]]], grad_fn=<AddBackward0>)


-----------Attn : Q1xK1---------
tensor([[[[-0.3839,  0.3844,  0.0229, -0.3405],
          [-0.0877,  0.1931,  2.7363,  0.8162],
          [-0.1728,  0.3630,  1.7052,  0.7037]]]], grad_fn=<AddBackward0>)
tensor([[[[-0.1631,  0.2604,  3.6420,  1.1873],
          [-0.2228, -0.0282, -0.5896, -0.3328],
          [-0.2500,  0.3592,  1.4500,  0.7168]]]], grad_fn=<AddBackward0>)


tensor([[[[-0.3839,  0.3844,  0.0229, -0.3405],
          [-0.0877,  0.1931,  2.7363,  0.8162],
          [-
          [-
```