FlashAttention

賴俊宇

2024/01/26

相关链接

- GitHub 仓库（仓库中包含 V1、V2 的论文）
- HuggingFace
- From Online Softmax to FlashAttention（个人强烈推荐）
- FlashAttention V1 的推导细节（个人推荐）
- FlashAttention V1、V2 差异总结（个人推荐）

FlashAttention

- Installation
- GPU Basics
- FlashAttention V1
- FlashAttention V2
- Other

Note

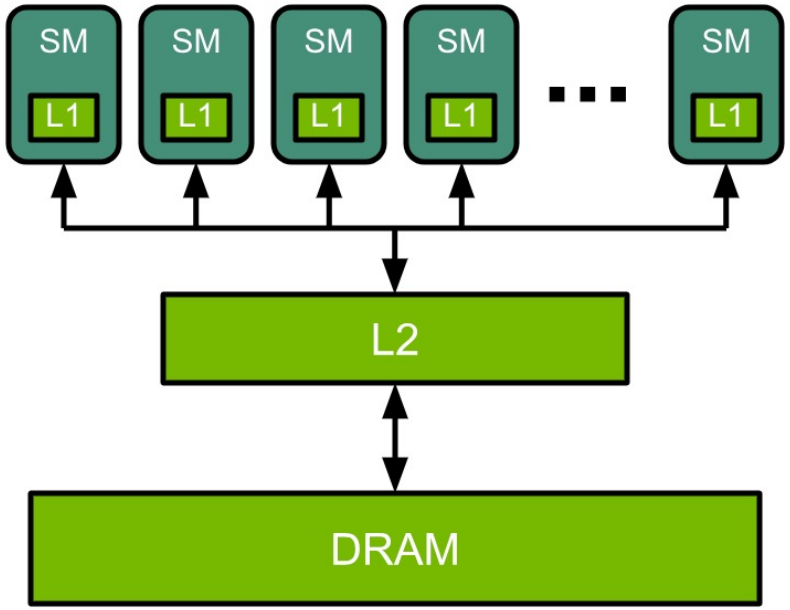
- 一定要先浏览一遍 GitHub 仓库中的 Installation and features
- 安装过程中会使用 ninja 做编译，一定要注意设置 MAX_JOBS 环境变量，防止机器内存被快速用完
- 编译过程比较慢，这是正常的
- FlashAttention 目前仅支持 Ampere、Ada、Hopper 架构的 GPU，以 91 机器为例，只有 A6000 支持，TITAN 不支持
- FlashAttention 仅支持 fp16 和 bf16 两种数据类型

FlashAttention

- Installation
- GPU Basics
- FlashAttention V1
- FlashAttention V2
- Other

GPU Architecture

- 从 high level 的角度看，GPU 的组件包括：Streaming Multiprocessors、on-chip L2 cache、high-bandwidth DRAM
- 其中计算指令通过 SM 执行，数据和代码会从 DRAM 缓存到 cache 上
- 以 A100 为例，包含 108 个 SM、40MB 的 L2 cache、80G 的 DRAM



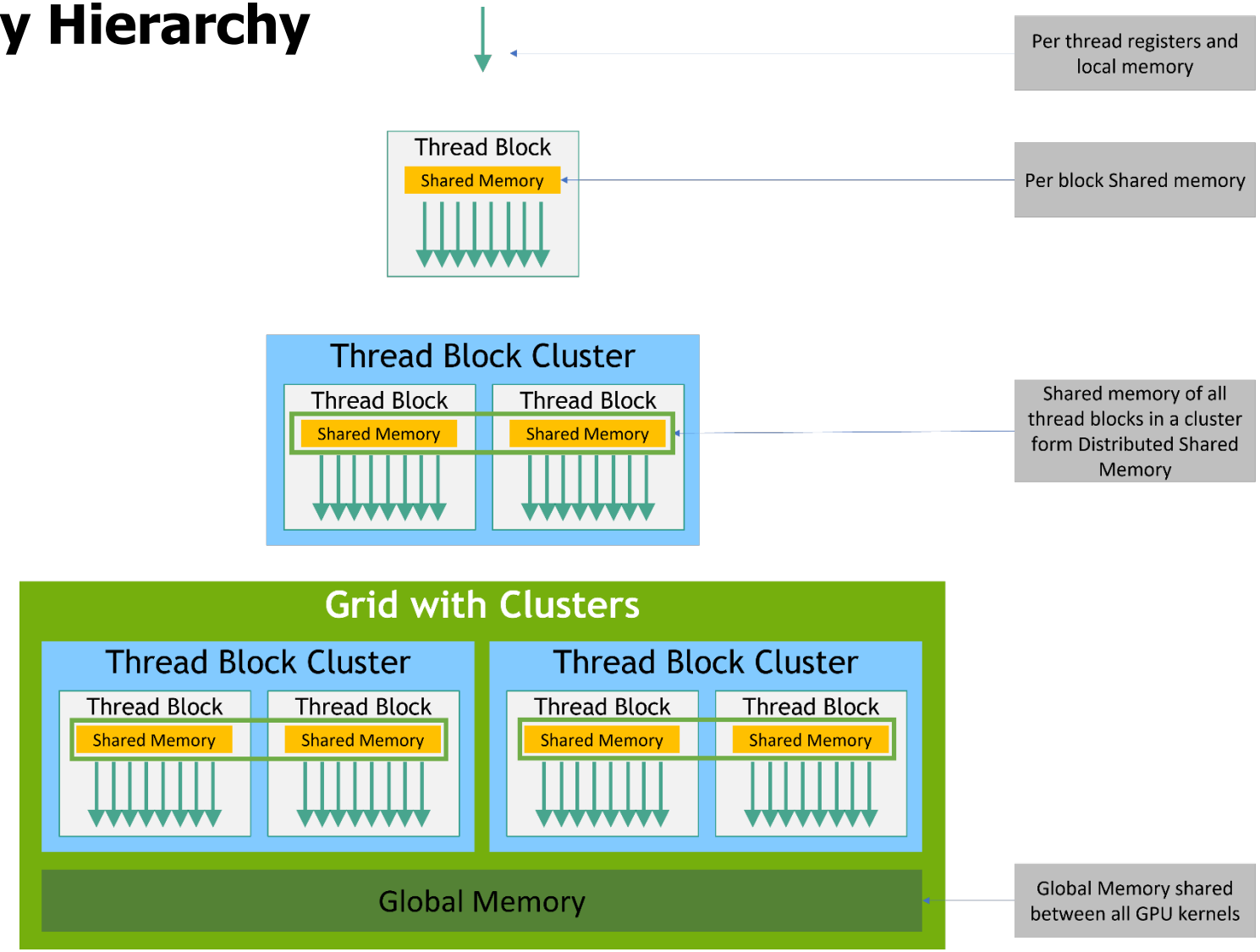
Streaming Multiprocessors（SM）

- Streaming Multiprocessors（SM）：GPU 内部的数据处理单元，每个 SM 有自己的执行流，可以类比为多核 CPU 中的一个核，只是 GPU 的一个核能运行多个线程
- 一个 SM 的构成：
  - 多个 CUDA Core，用于做数学运算
  - 若干 special function units，用于特殊的计算场景
  - 几个 warp scheduler
- 此外，一个 SM 还拥有：
  - 一个 read-only constant cache
  - 一个统一的 data cache 和 shared memory，大小根据具体的设备而不同，大概是一百多到两百多 KB，shared memory 的大小可配置，配置完后剩余的存储空间就作为 L1 cache

Thread Hierarchy

- 多个线程被组织成一个 block，在执行过程中，同一个 block 内的线程会被放在一个 SM 上执行，因此同一个 block 中的线程会共享 L1，一个 block 中最多包含 1024 个线程
- 多个 block 会被组织成一个 grid，一个 grid 中包含多少 block 由具体的数据规模决定
- 一方面来说，我们可以让一次计算尽可能使用多个 block 来提高并行度；另一方面，我们也可以让一个 SM 并发执行多个计算任务的 block
- 从硬件执行的角度来说，SM 会把一个 block 中的线程再分成 32 个为一组，称为 warp，一个 warp 上的线程会执行完全一样的指令，所以效率最高的情况是 warp 中的线程执行路径完全相同；而当出现分支的情况下，可能会导致部分线程提前执行完指令，进而导致当前的 GPU core 空闲

Memory Hierarchy



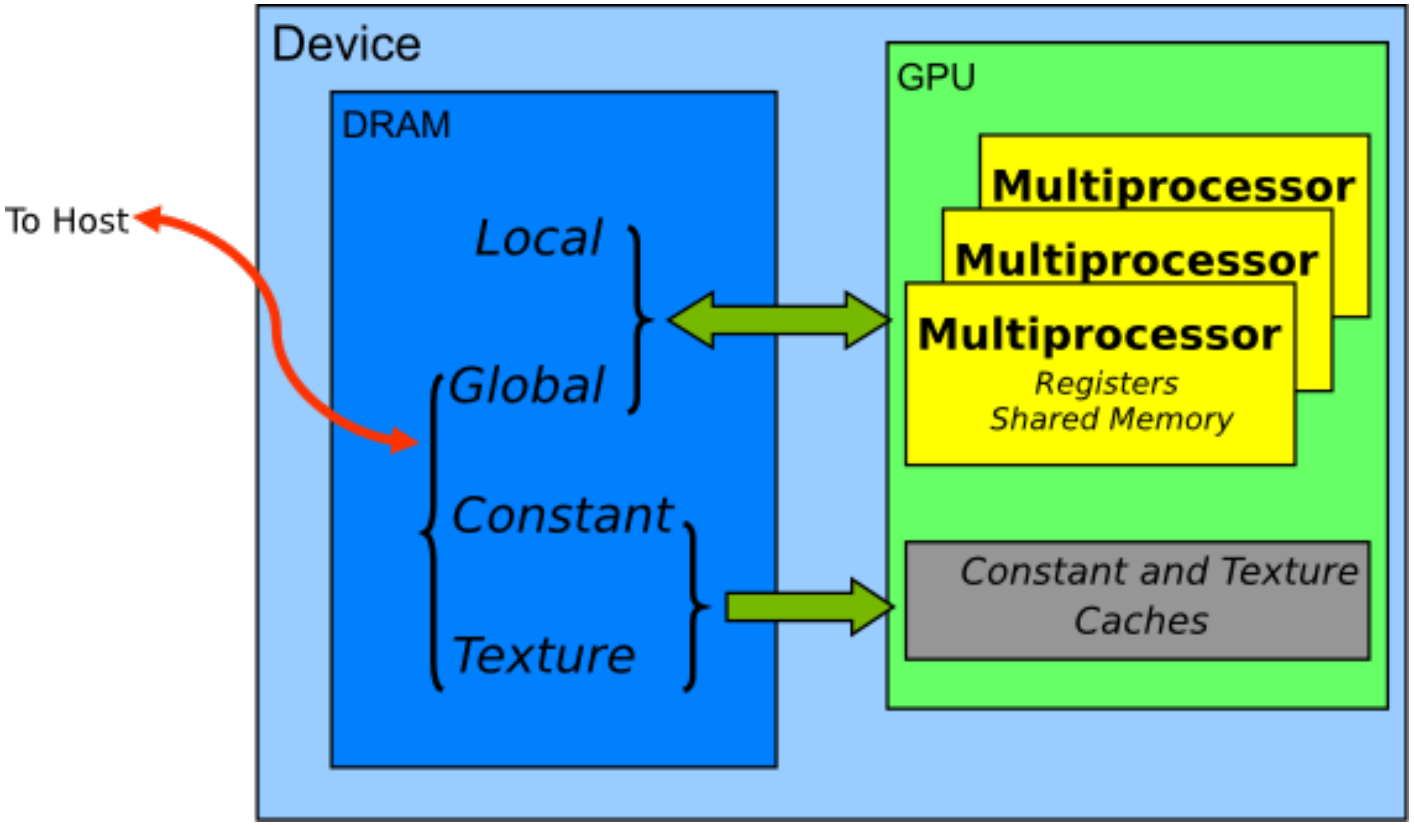
GPU Basics

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

Memory Hierarchy



Memory 补充说明

- on-chip memory：包括 register 和 shared memory，所有的 on-chip memory 都是 SRAM
- off-chip memory：包括 global、local、constants、texture memory，所有的 off-chip memory 都是 DRAM
- Global Memory 中访问的数据总是会被缓存到 L2 中，当满足一些更严格的条件时会进一步被缓存到 L1 中
- GPU DRAM 的大小 = off-chip memory 的大小 = "显存"
- High Bandwidth Memory（HBM）：可以认为指的就是 DRAM
- 之前提到过，L1 cache 和 shared memory 共享一块 on-chip memory，所以我们可以认为这两者的访问速度相同。注意，cache 是程序员无法控制的，但 shared memory 可以

FlashAttention

- Installation
- GPU Basics
- FlashAttention V1
- FlashAttention V2
- Other

Basic Info

- 效果：FlashAttention 可以加速 Attention Layer 在训练和推理过程中的计算速度，并且保证计算结果准确
- 出发点：Transformer 架构的计算时间开销大
- 原理：减少存储访问开销，这与绝大数减少计算量的方法的原理是不一样的

标准 Self Attention

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 $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, 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 $\mathbf{O}$.

- N 表示序列长度， d 表示 head dimension
- 在这个过程中，一共包含了 8 次需要访问 HBM 的操作
  - 第 1 行：读 Q、K，写 S
  - 第 2 行：读 S，写 P
  - 第 3 行：读 P、V，写 O
- HBM 访问成本： $\boldsymbol{O}(Nd + N^2)$

优化维度

Algorithm 0 Standard Attention Implementation

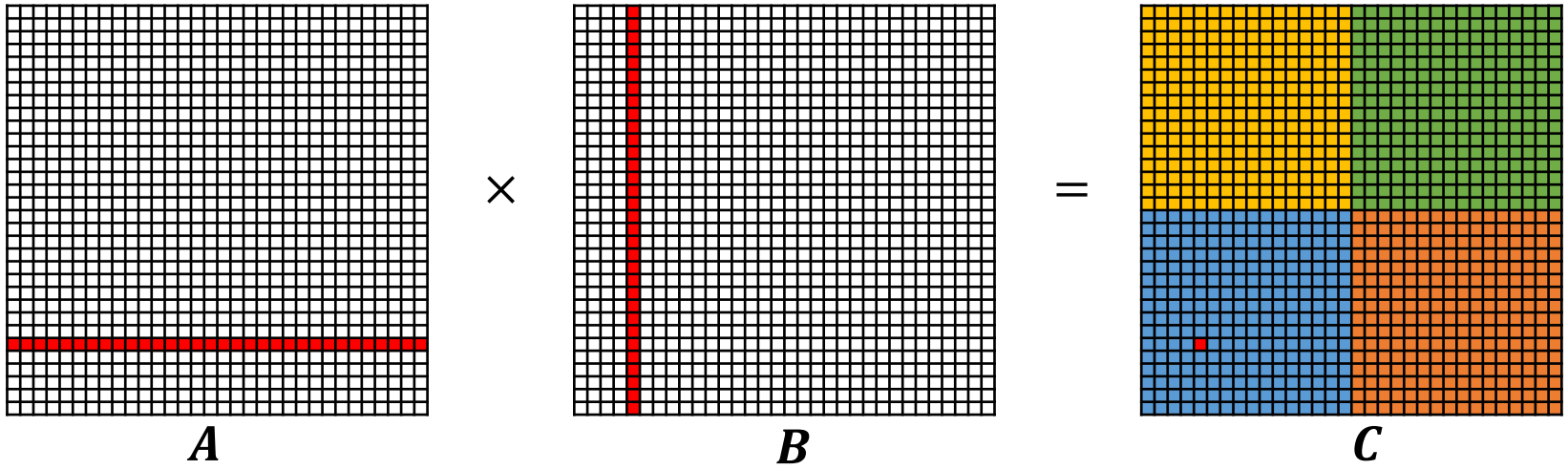
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

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 $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, 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 $\mathbf{O}$.

- 一种思路是：减少每一步中实际访问 HBM（global memory）的次数
- 或者：调整算法步骤，减少整体流程上访问 HBM 的次数

从 block 出发思考问题

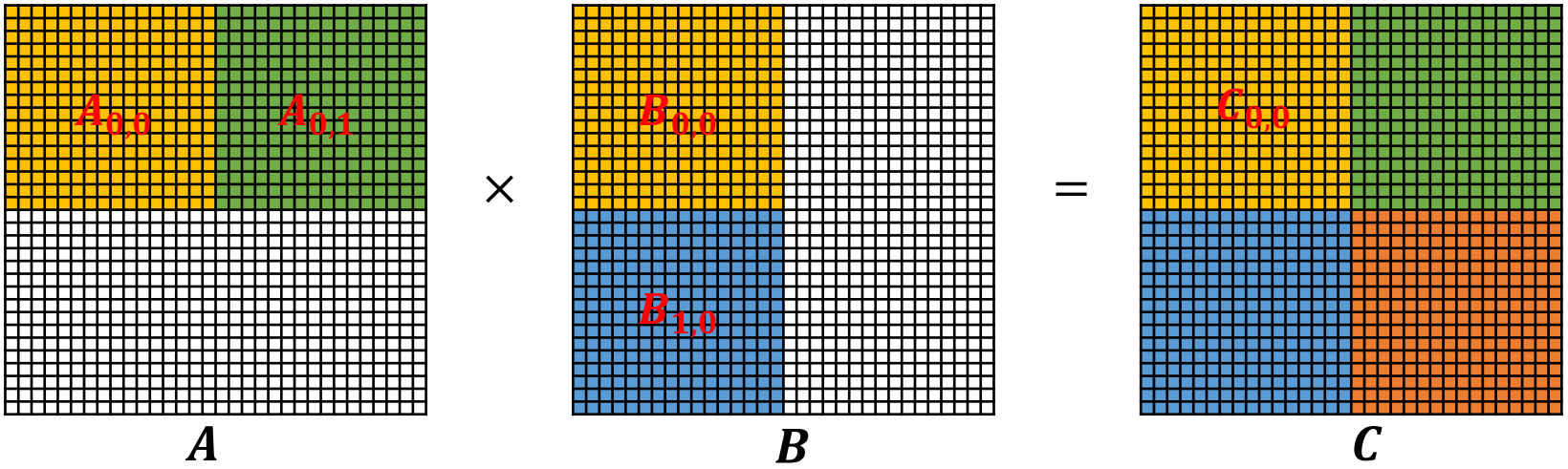
- 以矩阵乘法 $C = A \times B$ 为例，在实际的计算过程中，线程会被组织成 block，再交由 SM 执行
- 以 C 为 32*32 的矩阵，block 为 16*16 为例，一个 naive 的实现方法：



- C 中每个位置的计算需要访问 global memory 2*32 次，总共就是 2*32*32*32 次

Tiling 技术

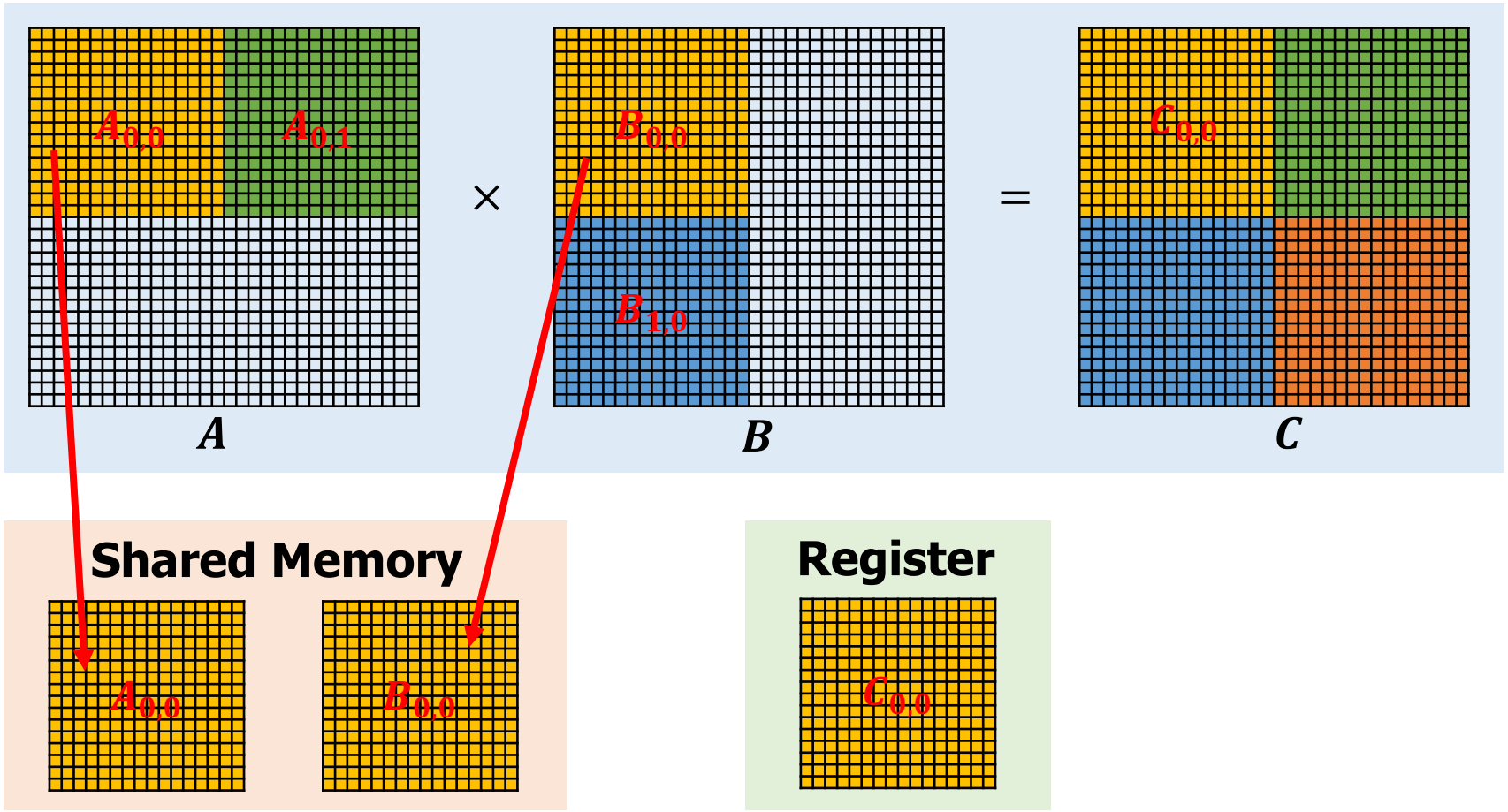
- 在 naive 的实现方法中，我们并没有考虑利用 shared memory，而 Tiling 技术就是通过利用 shared memory 减少 global memory 的访问
- Tiling 技术图示：



- $A_{0,0} \times B_{0,0} + A_{0,1} \times B_{1,0} = C_{0,0}$
- $A_{0,0}$ 和 $B_{0,0}$ 可以同时存储在 shared memory 上，$C_{0,0}$ 中的每个元素的值存储在 register 上

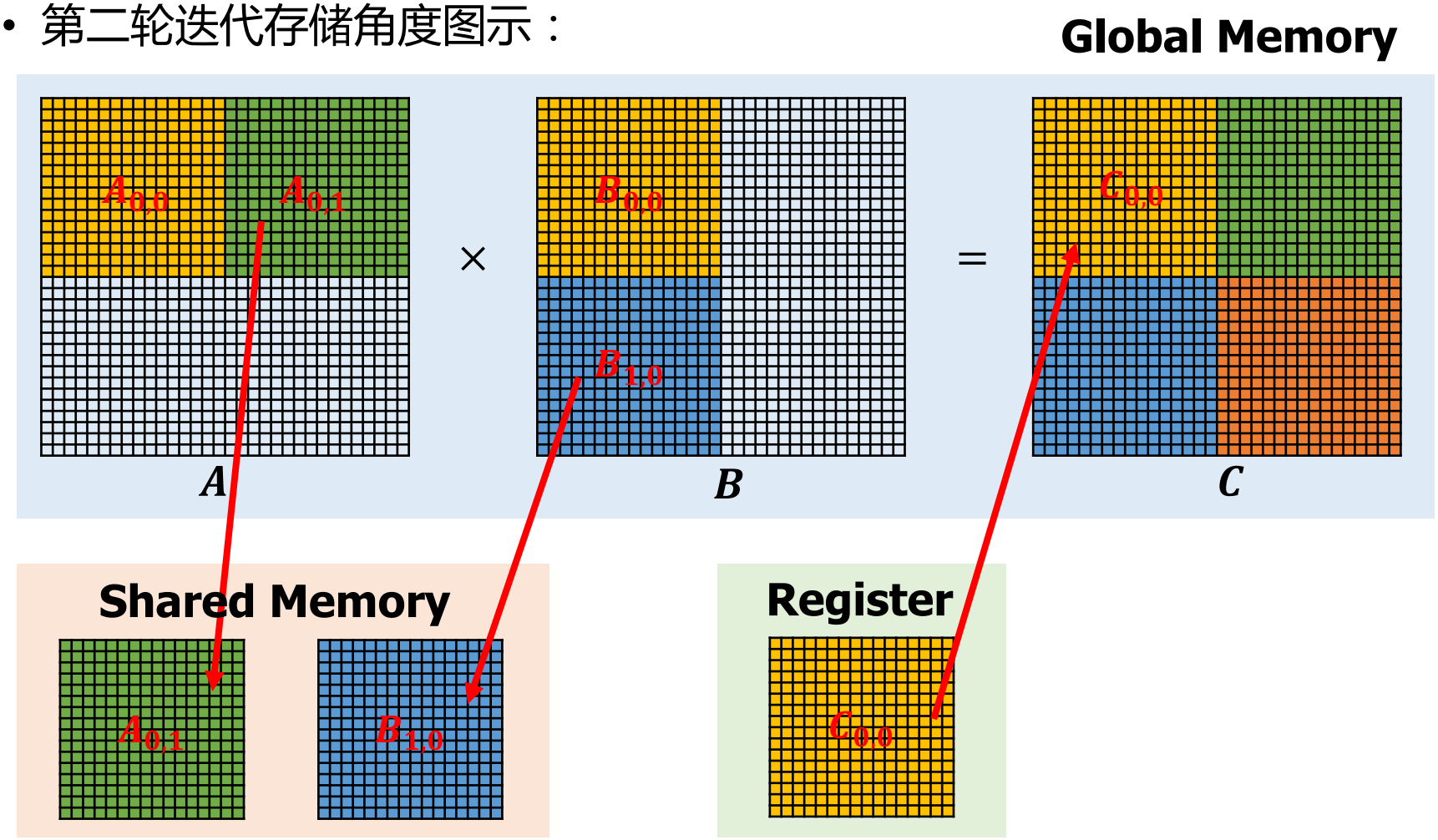
Tiling 技术 (cont'd)

- 第一轮迭代存储角度图示：



Tiling 技术 (cont'd)

- 第二轮迭代存储角度图示：

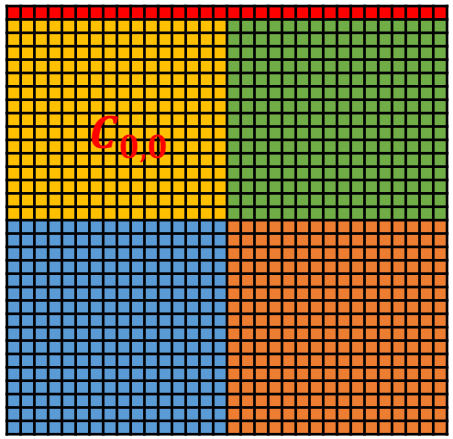


Tiling 技术 (cont'd)

- 总计算量保持不变
- 但是总的 global memory 的访问次数大大降低，我们算出 C 矩阵四分之一的结果时，访问了 16*16*4 次 global memory，那么总共将访问 16*16*4*4 次，一共 4096 次；而之前 naive 的方法访问了 65536 次，减少为了原来的 1/16
- 通过调整 block 的大小，我们还可以进一步改变 global memory 的访问次数

Unfortunately

- Tiling 技术虽然可以用于矩阵乘法，但是不能直接用于 Attention 的计算
- 在 Attention Layer 的计算中，存在一次 softmax 操作，这个操作是 row-wise 的



- 在仅计算出 $C_{0,0}$ 的情况下，无法计算 softmax 的值，因为 softmax 的值还依赖于 $C_{0,1}$
- 因此 Tiling 技术仅仅减少了标准 Attention 算法中矩阵乘法的实际 global memory 访问次数，但是并没有从整体上改变标准 Attention 算法的流程

Safe Softmax

- Softmax 的公式：

$$\text{softmax}(\{x_1, \ldots, x_N\}) = \left\{ \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \right\}_{i=1}^{N}$$

- 为了防止指数爆炸问题，在实际计算的时候会采用 Safe Softmax：

$$\frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} = \frac{e^{x_i - m}}{\sum_{j=1}^{N} e^{x_j - m}}$$

- 一般来说，上述公式中 $m = \max_{j=1}^{N}(x_j)$，这样能保证指数项一定小于等于 0

一个迭代式的 Safe Softmax 的算法（V1）

NOTATIONS

$\{m_i\}$: $\max_{j=1}^{i}\{x_j\}$, with initial value $m_0 = -\infty$.

$\{d_i\}$: $\sum_{j=1}^{i} e^{x_j - m_N}$, with initial value $d_0 = 0$, d_N is the denominator of safe softmax.

$\{a_i\}$: the final softmax value.

BODY

for $i \leftarrow 1, N$ **do**

$$m_i \leftarrow \max\left(m_{i-1}, x_i\right)$$

end

for $i \leftarrow 1, N$ **do**

$$d_i \leftarrow d_{i-1} + e^{x_i - m_N}$$

end

for $i \leftarrow 1, N$ **do**

$$a_i \leftarrow \frac{e^{x_i - m_N}}{d_N}$$

end

Online Softmax（V2）

- 优化思路：消除 d_i 对 m_N 的依赖

$$
\begin{aligned}
d'_i &= \sum_{j=1}^{i} e^{x_j - m_i} \\
&= \left(\sum_{j=1}^{i-1} e^{x_j - m_i} \right) + e^{x_i - m_i} \\
&= \left(\sum_{j=1}^{i-1} e^{x_j - m_{i-1}} \right) e^{m_{i-1} - m_i} + e^{x_i - m_i} \\
&= d'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i}
\end{aligned}
$$

---

for $i \leftarrow 1, N$ **do**

$$
\begin{aligned}
m_i &\leftarrow \max(m_{i-1}, x_i) \\
d'_i &\leftarrow d'_{i-1} e^{m_{i-1} - m_i} + e^{x_i - m_i}
\end{aligned}
$$

end
for $i \leftarrow 1, N$ **do**

$$
a_i \leftarrow \frac{e^{x_i - m_N}}{d'_N}
$$

end

V2 版本的算法

Again, Unfortunately

- 这个优化对于 softmax 操作来说已经到头了，我们不可能在一次循环中把 softmax 的结果计算出来
- 原因：向量中的每个元素都是独立的，不可能在没有遍历到后续元素的情况下，确定当前元素最终的 softmax 值

But

- Attention Layer 的最终目的并不是为了计算 softmax，而是 softmax 以后的还需要乘以矩阵 V

一个 2-pass 的 Self Attention 的算法（V1）

NOTATIONS

$Q[k,:]$: the k-th row vector of Q matrix.

$K^T[:,i]$: the i-th column vector of K^T matrix.

$O[k,:]$: the k-th row of output O matrix.

$V[i,:]$: the i-th row of V matrix.

$\{\boldsymbol{o}_i\}$: $\sum_{j=1}^{i} a_j V[j,:]$, a row vector storing partial aggregation result $A[k,:i] \times V[:i,:]$

BODY

for $i \leftarrow 1, N$ **do**

$$
\begin{aligned}
x_i &\leftarrow Q[k,:]\, K^T[:,i] \\
m_i &\leftarrow \max(m_{i-1}, x_i) \\
d'_i &\leftarrow d'_{i-1} e^{m_{i-1}-m_i} + e^{x_i - m_i}
\end{aligned}
$$

end

for $i \leftarrow 1, N$ **do**

$$
\begin{aligned}
a_i &\leftarrow \frac{e^{x_i - m_N}}{d'_N} \\
\boldsymbol{o}_i &\leftarrow \boldsymbol{o}_{i-1} + a_i V[i,:]
\end{aligned}
$$

end

$$O[k,:] \leftarrow \boldsymbol{o}_N$$

仅考虑输出矩阵 O 的第 k 行

改良版的 1-pass 算法（V2）

$$
\begin{aligned}
\boldsymbol{o}'_i &= \sum_{j=1}^{i} \frac{e^{x_j - m_i}}{d'_i} V[j,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_i}}{d'_i} V[j,:] \right) + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{d'_{i-1}} \frac{e^{x_j - m_i}}{e^{x_j - m_{i-1}}} \frac{d'_{i-1}}{d'_i} V[j,:] \right) + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \left(\sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{d'_{i-1}} V[j,:] \right) \frac{d'_{i-1}}{d'_i} e^{m_{i-1} - m_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:] \\
&= \boldsymbol{o}'_{i-1} \frac{d'_{i-1} e^{m_{i-1} - m_i}}{d'_i} + \frac{e^{x_i - m_i}}{d'_i} V[i,:]
\end{aligned}
$$

改良版的 1-pass 算法（V2）（cont'd）

for $i \leftarrow 1, N$ **do**

$$
\begin{aligned}
x_i &\leftarrow Q[k,:]\,K^T[:,i] \\
m_i &\leftarrow \max(m_{i-1}, x_i) \\
d'_i &\leftarrow d'_{i-1}\,e^{m_{i-1}-m_i} + e^{x_i-m_i} \\
\boldsymbol{o}'_i &\leftarrow \boldsymbol{o}'_{i-1}\frac{d'_{i-1}\,e^{m_{i-1}-m_i}}{d'_i} + \frac{e^{x_i-m_i}}{d'_i}V[i,:]
\end{aligned}
$$

end

$$O[k,:] \leftarrow \boldsymbol{o}'_N$$

- 我们会惊喜地发现，虽然 softmax 无法用 1-pass 的方式解决，但是 Self Attention 的计算可以
- 这个 1-pass 的 Self Attention 的算法就可以看作 FlashAttention V1 的原型

FlashAttention V1

- FlashAttention 在实现时，还考虑到了 Tiling 技术

NEW NOTATIONS

b: the block size of the tile

#tiles: number of tiles in the row, $N = b \times$ #tiles.

$\boldsymbol{x}_i$: a vector storing the $Q[k]\,K^T$ value of the i-th tile $[(i-1)\,b{:}\,i\,b]$.

$m_i^{(\text{local})}$: the local maximum value inside $\boldsymbol{x}_i$.

BODY

for $i \leftarrow 1, \#\text{tiles}$ **do**

$$\boldsymbol{x}_i \leftarrow Q[k,:]\,K^T[:,(i-1)\,b{:}\,i\,b]$$

一次计算 K^T 的多列

$$m_i^{(\text{local})} = \max_{j=1}^{b}(\boldsymbol{x}_i[j])$$

$$m_i \leftarrow \max\left(m_{i-1}, m_i^{(\text{local})}\right)$$

$$d_i' \leftarrow d_{i-1}'\,e^{m_{i-1}-m_i} + \sum_{j=1}^{b} e^{\boldsymbol{x}_i[j]-m_i}$$

$\boldsymbol{x}_i[j]$ 指的是向量 $\boldsymbol{x}_i$ 的第 j 列

$$\boldsymbol{o}_i' \leftarrow \boldsymbol{o}_{i-1}'\frac{d_{i-1}'\,e^{m_{i-1}-m_i}}{d_i'} + \sum_{j=1}^{b}\frac{e^{\boldsymbol{x}_i[j]-m_i}}{d_i'}V[j+(i-1)\,b,:]$$

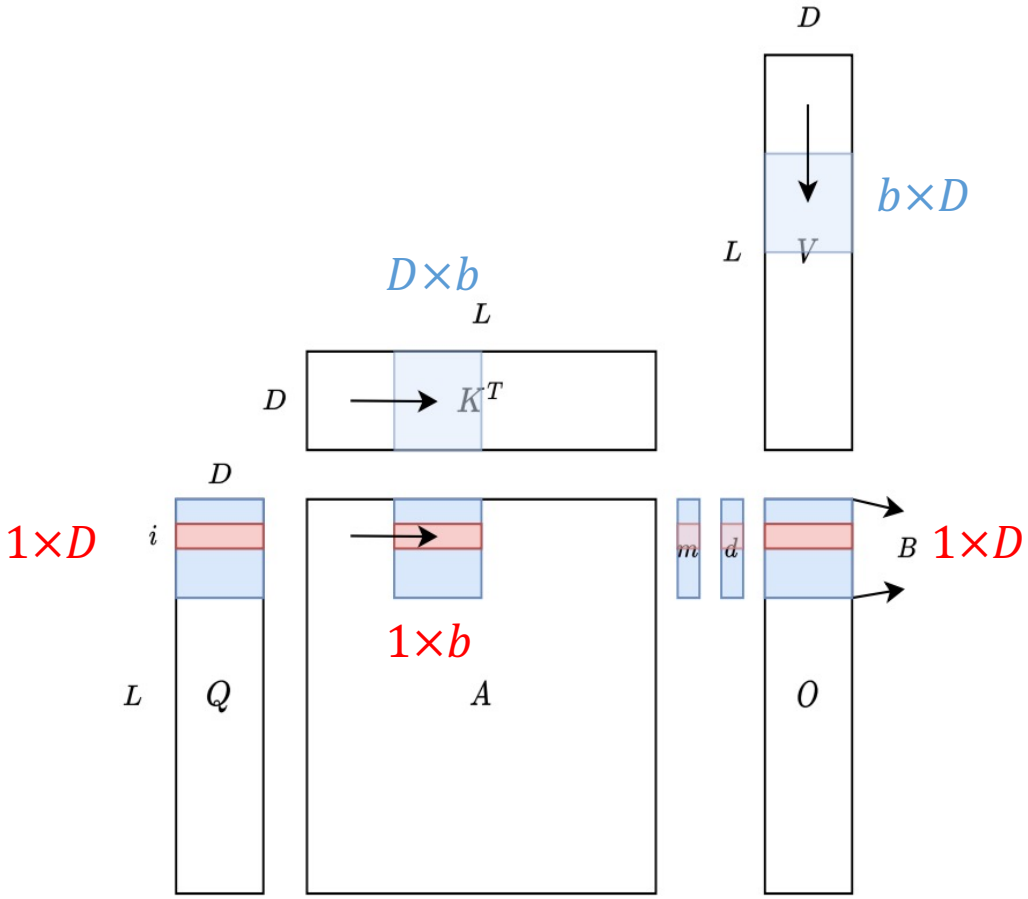
这里计算得到的结果是一个 $1\times D$ 的向量

end

$$O[k,:] \leftarrow \boldsymbol{o}_{N/b}'$$

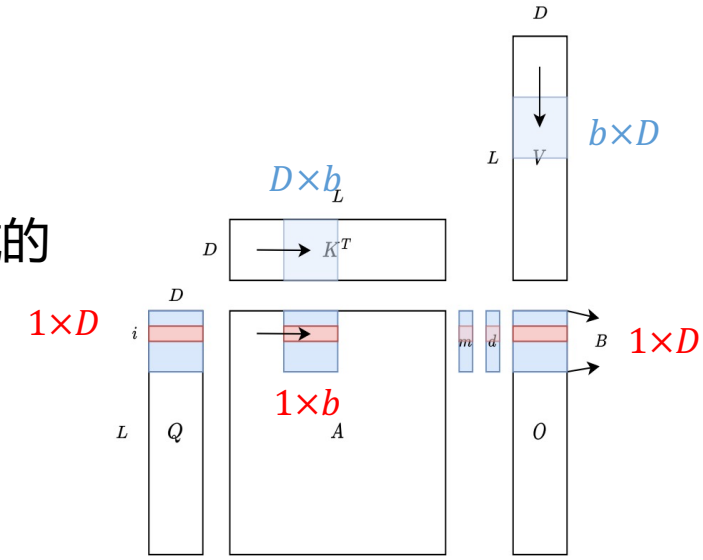
FlashAttention V1

- 如下图所示，其中蓝色的部分表示当前存储在 shared memory 中的部分



补充说明

- FlashAttention 原文中的算法描述和这里存在一些差异，在原文中：
  - B_c 指的是图中的 b
  - B_r 指的是图中矩阵 Q 中蓝色区域的行数
  - 算法描述的形式是针对整个矩阵 Q 中蓝色区域的
  - 外层循环是矩阵 K, V，内层循环是矩阵 Q, O
  - d 指的是图中的 D
  - N 指的是图中的 L



- FlashAttention 的实现是不唯一的，事实上，很多实现都没有完全采用原始论文中的方法，会有一定程度的调整
- 此处仅讨论了 forward 的情况，backward 的情况类似

原文算法描述

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

分块

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.

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.

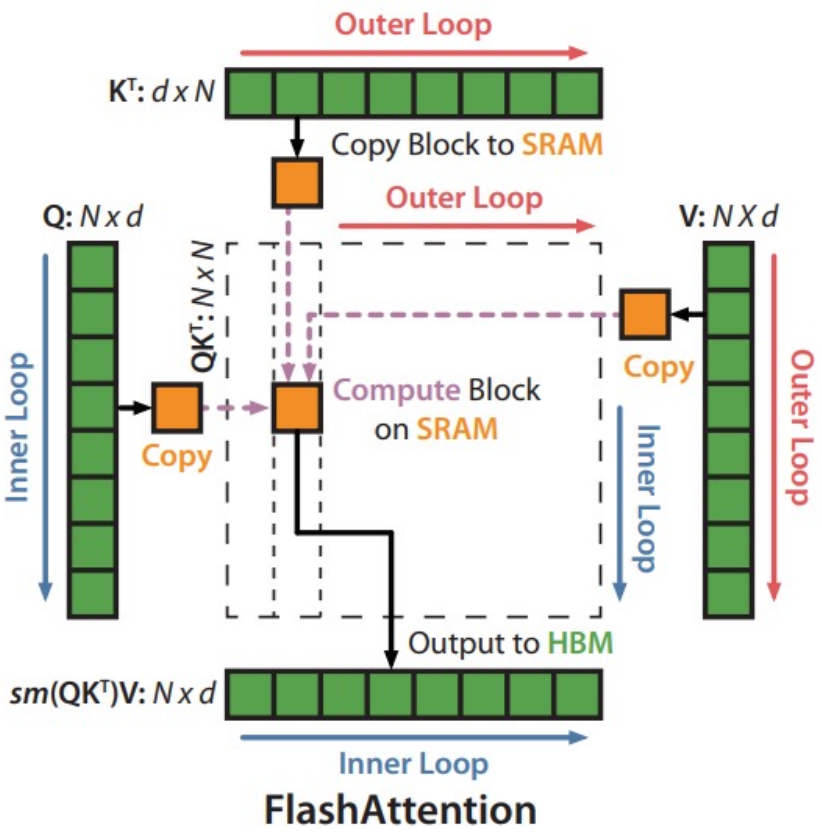
O_i 和 Q_i 是对应的，位置一样

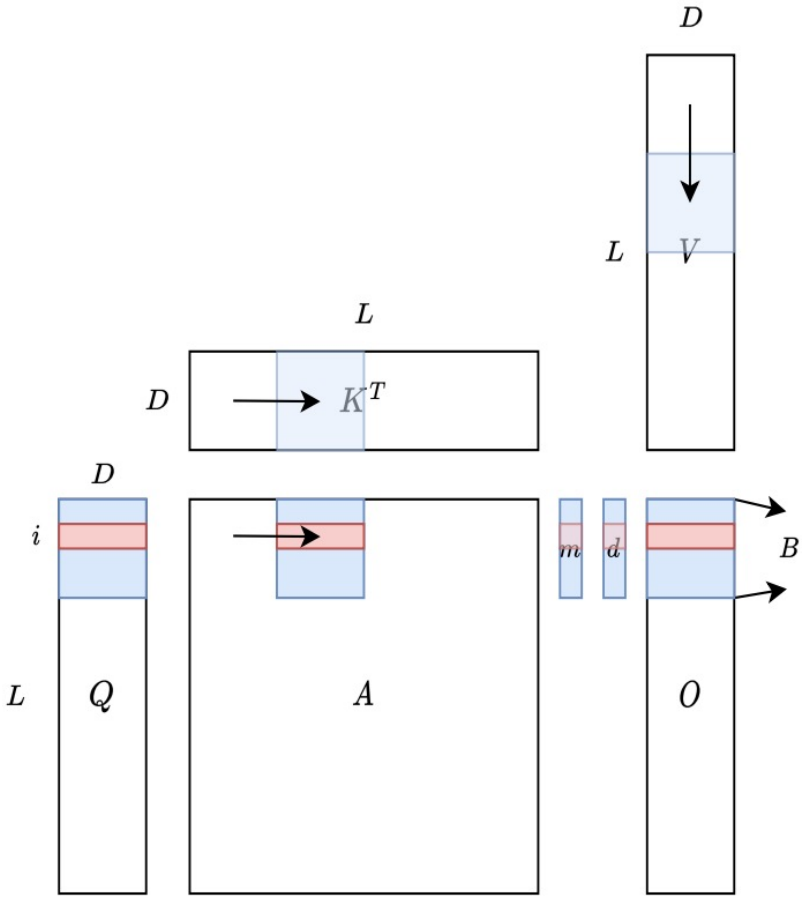
14: **end for**

15: **end for**

16: Return $\mathbf{O}$.

原文算法图和此处的图的比较



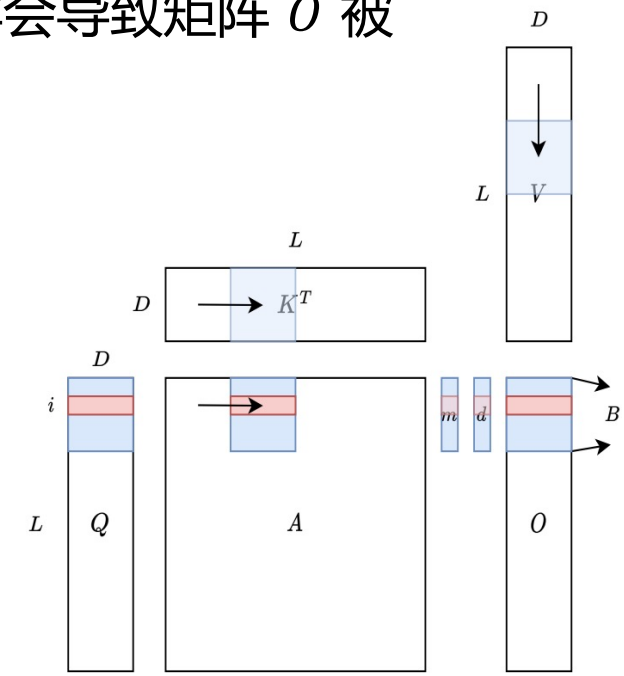


FlashAttention

- Installation
- GPU Basics
- FlashAttention V1
- FlashAttention V2
- Other

改进一：调整内外循环

- FlashAttention V1 中采用了一个不直觉的外层循环矩阵 K, V，内层循环矩阵 Q, O 的方式，这个思路其实很奇怪，因为这样会导致矩阵 O 被额外加载



- 事实上，在 FlashAttention V2 出来之前，很多 FlashAttention 的实现就修改了这个循环顺序

改进二：减少了非矩阵乘法的运算次数

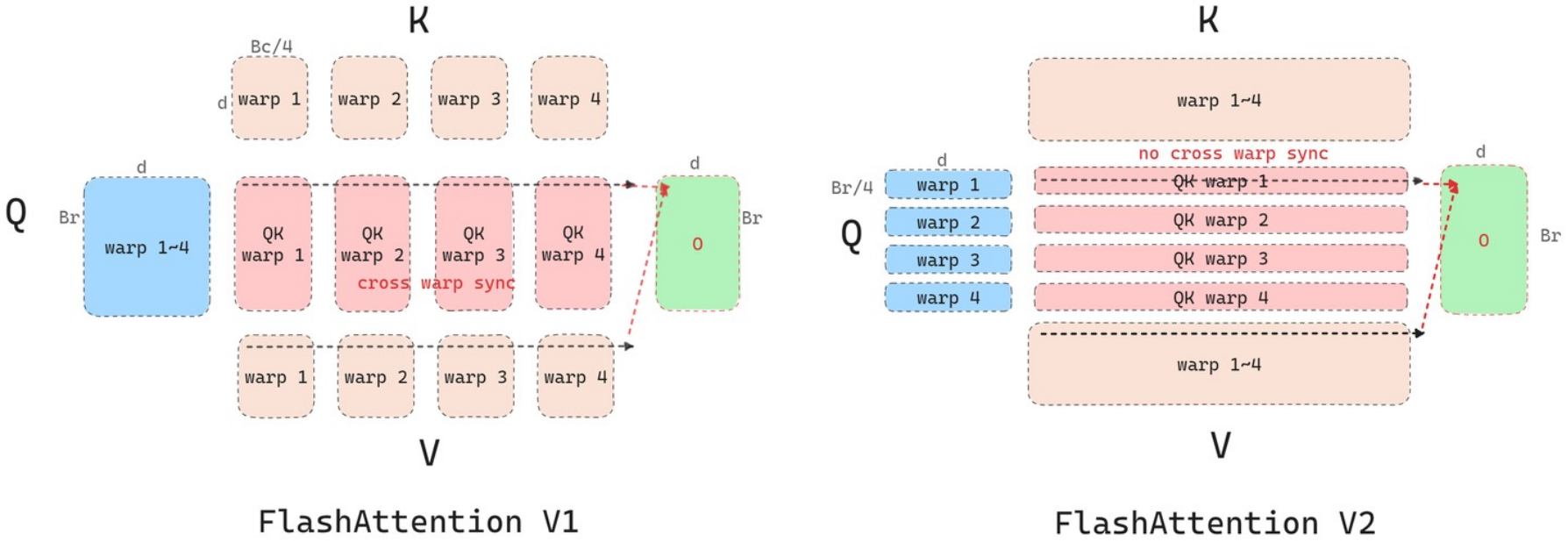
- 原因：现代 GPU 对矩阵乘法有专门的硬件优化，矩阵乘法的 FLOP 是非矩阵乘法的 FLOP 的 16 倍左右
- 具体实现上，FlashAttention V1 每轮迭代都有一个 rescale 操作，即：

$$
\begin{aligned}
&\textbf{for } i \leftarrow 1, N \textbf{ do} \\
&\qquad x_i \leftarrow Q[k,:]\,K^T[:,i] \\
&\qquad m_i \leftarrow \max(m_{i-1}, x_i) \\
&\qquad d'_i \leftarrow d'_{i-1} e^{m_{i-1}-m_i} + e^{x_i-m_i} \\
&\qquad \boldsymbol{o}'_i \leftarrow \boldsymbol{o}'_{i-1} \frac{d'_{i-1} e^{m_{i-1}-m_i}}{d'_i} + \frac{e^{x_i-m_i}}{d'_i} V[i,:] \quad \text{rescale} \\
&\textbf{end} \\
&O[k,:] \leftarrow \boldsymbol{o}'_N
\end{aligned}
$$

- 在 V2 中，不再在每轮迭代中都除以 d'_i，而是等循环体结束以后，对计算得到的 $\boldsymbol{o}'_N$ 统一除以 d'_N

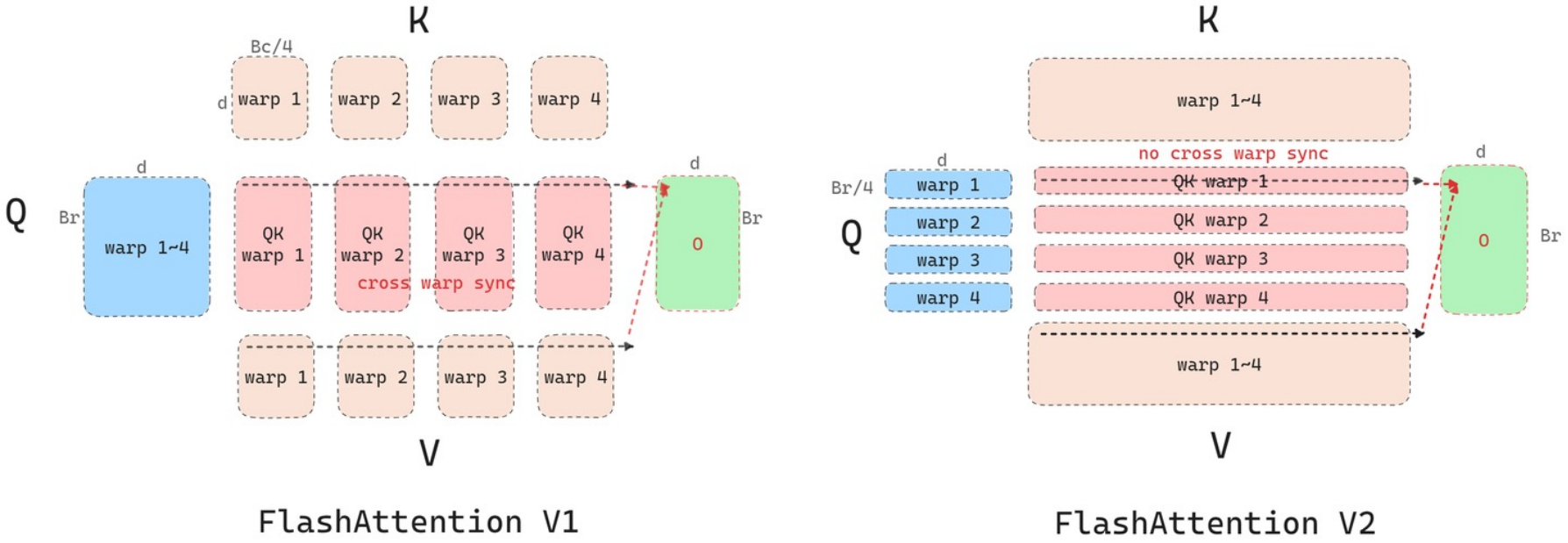
改进三：Warp Level 并行度

- 假设一个 block 实际上会被 SM 划分成 4 个 warp，在 V1 版本中，矩阵 K, V 的 block 会被划分成 4 个 warp，每个 warp 计算 $\boldsymbol{Q}_i\boldsymbol{K}_j^T$ 后会得到一个 $B_r \times \frac{B_c}{4}$ 的矩阵，需要 4 个 warp 全部计算完以后，把四个矩阵排成一行（下图中 V1 版本红色的四个矩阵），才能计算 $\boldsymbol{Q}_i\boldsymbol{K}_j^T$ 真正的值，这个过程中存在 warp 之间的通信



改进三：Warp Level 并行度（cont'd）

- 在 V2 版本中，矩阵 Q 的 block 会被划分成 4 个 warp，这种情况下每个 warp 计算出来的结果就是一个 $\frac{B_r}{4} \times B_c$ 的矩阵，这个矩阵已经包含了 $\boldsymbol{Q}_i \boldsymbol{K}_j^T$ 中完整的 $\frac{B_r}{4}$ 行，所以整个计算就只需要在 warp 内部进行，不需要进行 warp 之间的通信



FlashAttention

- Installation
- GPU Basics
- FlashAttention V1
- FlashAttention V2
- Other

FlashAttention 目前最方便的使用途径

- 使用官方库 flash_attn，可以通过 pip 直接安装，这种方法如果需要做一些逻辑上的修改（例如加 mask），学习和 Debug 的成本较高
- 使用 Triton Language 中的实现，实际性能也非常好
  - Triton Language 将在下一次讨论班中介绍 ☺

Thanks！

賴俊宇

2024/01/26