# FlashAttention2详解（性能比FlashAttention提升200%）

- 地址：arxiv.org/pdf/2307.08691.pdf
- 作者：普林斯顿；斯坦福

最新FlashDecoding++

Austin：【FlashAttention-V4，非官方】FlashDecoding++

FlashAttention-V1和V3版本详解：

Austin：FlashAttention图解（如何加速Attention）

Austin：FlashAttenion-V3: Flash Decoding详解

## 摘要

在过去几年中，如何扩展Transformer使之能够处理更长的序列一直是一个重要问题，因为这能提高Transformer语言建模性能和高分辨率图像理解能力，以及解锁代码、音频和视频生成等新应用。然而增加序列长度，注意力层是主要瓶颈，因为它的运行时间和内存会随序列长度的增加呈二次（平方）增加。FlashAttention利用GPU非匀称的存储器层次结构，实现了显著的内存节省（从平方增加转为线性增加）和计算加速（提速2-4倍），而且计算结果保持一致。但是，FlashAttention仍然不如优化的矩阵乘法（GEMM）操作快，只达到理论最大FLOPs/s的25-40%。作者观察到，这种低效是由于GPU对不同thread blocks和warps工作分配不是最优的，造成了利用率低和不必要的共享内存读写。因此，本文提出了FlashAttention-2以解决这些问题。

## 简介

如何扩展Transformer使之能够处理更长的序列一直是一个挑战，**因为其核心注意力层的运行时间和内存占用量随输入序列长度成二次增加。**我们希望能够打破2k序列长度限制，从而能够训练书籍、高分辨率图像和长视频。此外，写作等应用也需要模型能够处理长序列。过去一年中，业界推出了一些远超之前长度的语言模型：GPT-4为32k，MosaicML的MPT为65k，以及Anthropic的Claude为100k。

虽然相比标准Attention，FlashAttention快了2$_{4倍，节约了10}$20倍内存，但是离设备理论最大throughput和flops还差了很多。本文提出了FlashAttention-2，它具有更好的并行性和工作分区。实验结果显示，FlashAttention-2在正向传递中实现了约2倍的速度提升，达到了理论最大吞吐量的73%，在反向传递中达到了理论最大吞吐量的63%。在每个A100 GPU上的训练速度可达到225 TFLOPs/s。

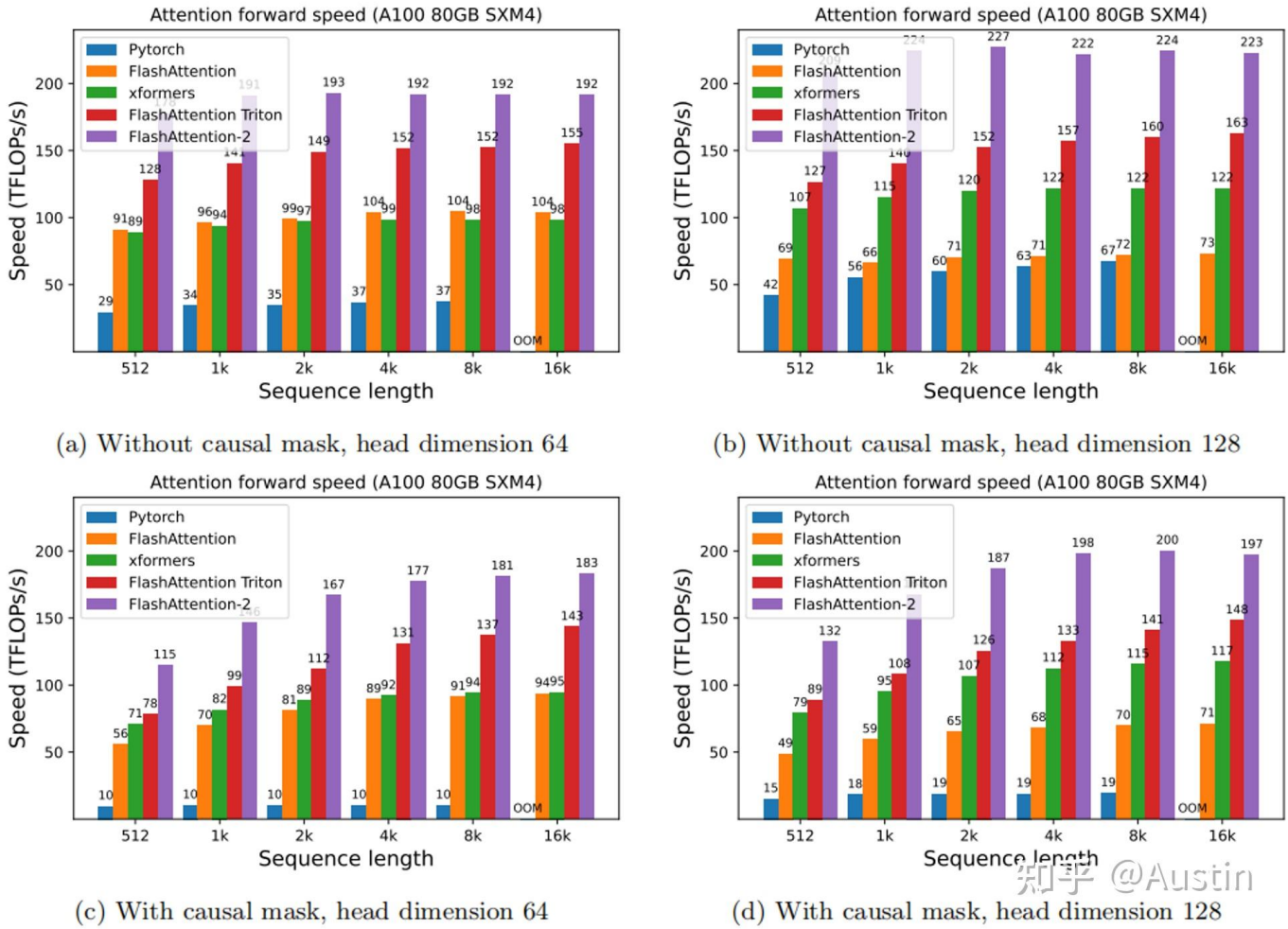
本文主要贡献和创新点为：

1. 减少了non-matmul FLOPs的数量（消除了原先频繁rescale）。虽然non-matmul FLOPs仅占总FLOPs的一小部分，但它们的执行时间较长，这是因为GPU有专用的矩阵乘法计算单元，其吞吐量高达非矩阵乘法吞吐量的16倍。因此，减少non-matmul FLOPs并尽可能多地执行matmul FLOPs非常重要。
2. 提出了在序列长度维度上并行化。该方法在输入序列很长（此时batch size通常很小）的情况下增加了GPU利用率。即使对于单个head，也在不同的thread block之间进行并行计算。
3. 在一个attention计算块内，将工作分配在一个thread block的不同warp上，以减少通信和共享内存读/写。

# 动机

为了解决这个问题，研究者们也提出了很多近似的attention算法，然而目前使用最多的还是标准attention。FlashAttention利用tiling、recomputation等技术显著提升了计算速度（提升了2$_{4倍}$），并且将内存占用从平方代价将为线性代价（节约了$_{10}$20倍内存）。虽然FlashAttention效果很好，但是仍然不如其他基本操作（如矩阵乘法）高效。例如，其前向推理仅达到GPU（A100）理论最大FLOPs/s的30-50%（下图）；反向传播更具挑战性，在A100上仅达到最大吞吐量的25-35%。相比之下，优化后的GEMM（矩阵乘法）可以达到最大吞吐量的80-90%。通过观察

分析，这种低效是由于GPU对不同thread blocks和warps工作分配不是最优的，造成了利用率低和不必要的共享内存读写。




(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



*Attention forward speed on A100 GPU. (Source: Figure 5 of the paper.)*

# 背景知识

下面介绍一些关于GPU的性能和计算特点，有关Attention和FlashAttention的详细内容请参考第一篇文章。

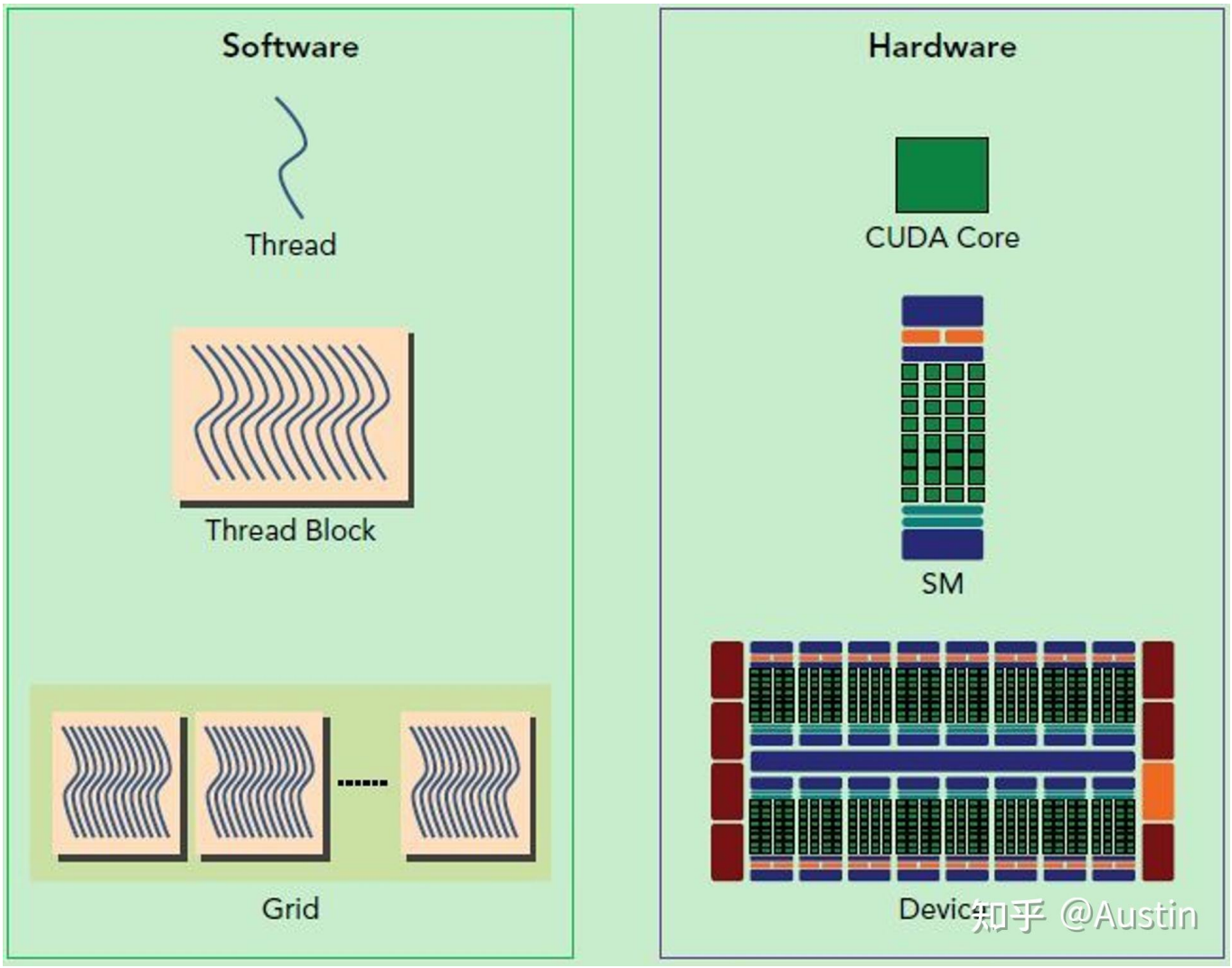
null

# GPU

**GPU performance characteristics.** GPU主要计算单元（如浮点运算单元）和内存层次结构。大多数现代GPU包含专用的低精度矩阵乘法单元（如Nvidia GPU的Tensor Core用于FP16/BF16矩阵乘法）。内存层次结构分为高带宽内存（High Bandwidth Memory, HBM）和片上SRAM（也称为shared memory）。以A100

GPU为例，它具有40-80GB的HBM，带宽为1.5-2.0TB/s，每个108个streaming multiprocessors共享的SRAM为192KB，带宽约为19TB/s。

> 这里忽略了L2缓存，因为不能直接被由程序员控制。



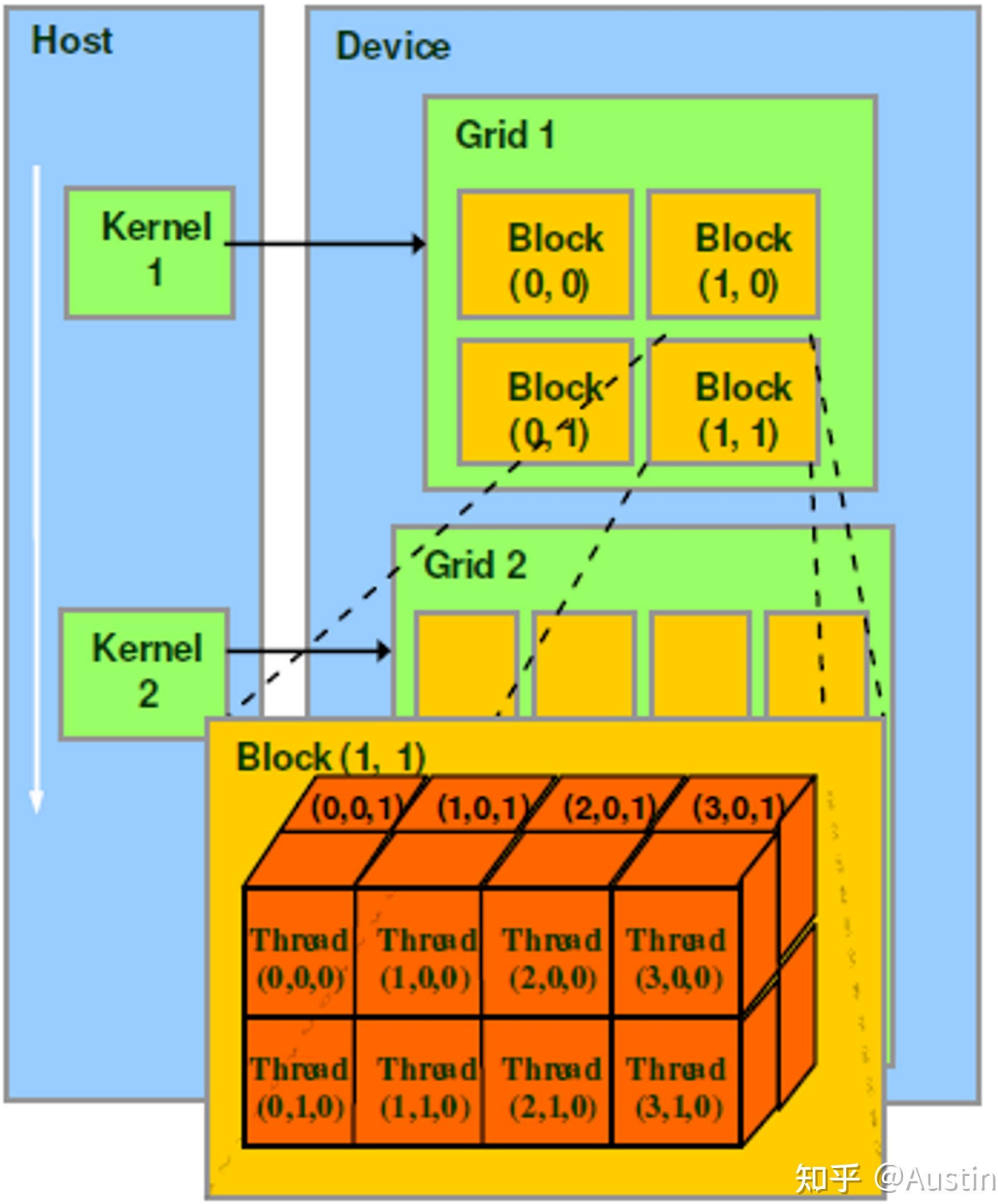

*CUDA的软件和硬件架构*

**从Hardware角度来看：**

- **Streaming Processor（SP）**：是最基本的处理单元，**从fermi架构开始被叫做CUDA core**。
- **Streaming MultiProcessor（SM）**：一个SM由多个CUDA core（SP）组成，每个SM在不同GPU架构上有不同数量的CUDA core，例如Pascal架构中一个SM有128个CUDA core。

SM还包括特殊运算单元(SFU)，共享内存(shared memory)，寄存器文件(Register File)和调度器(Warp Scheduler)等。register和shared memory是稀缺资源，**这些有限的资源就使每个SM中** active warps**有非常严格的限制，也就限制了并行能力。**

**从Software（编程）角度来看：**




*CUDA软件示例*

- **thread：** 一个CUDA并行程序由多个thread来执行
- thread是最基本的执行单元（the basic unit of execution）。
- **warp：** 一个warp通常包含32个thread。每个warp中的thread可以同时执行相同的指令，从而实现SIMT（单指令多线程）并行。

- **warp是SM中最小的调度单位**（the smallest scheduling unit on an SM），一个SM可以同时处理多个warp
- **thread block：** 一个thread block可以包含多个warp，同一个block中的thread可以同步，也可以通过shared memory进行通信。
- **thread block是GPU执行的最小单位（the smallest unit of execution on the GPU）。**
- 一个warp中的threads必然在同一个block中，如果block所含thread数量不是warp大小的整数倍，那么多出的那个warp中会剩余一些inactive的thread。也就是说，即使warp的thread数量不足，硬件也会为warp凑足thread，只不过这些thread是inactive状态，但也会消耗SM资源。
- **grid：** 在GPU编程中，grid是一个由多个thread block组成的二维或三维数组。grid的大小取决于计算任务的规模和thread block的大小，通常根据计算任务的特点和GPU性能来进行调整。

**Hardware和Software的联系：**

SM采用的是Single-Instruction Multiple-Thread（SIMT，单指令多线程）架构，warp是最基本的执行单元，一个warp包含32个并行thread，这些thread**以不同数据资源执行相同的指令** 。

当一个kernel被执行时，grid中的thread block被分配到SM上，大量的thread可能被分到不同的SM上，但是**一个线程块的thread只能在一个SM上调度** ，SM一般可以调度多个block。每个thread拥有自己的程序计数器和状态寄存器，并且可以使用不同的数据来执行指令，从而实现并行计算，这就是所谓的Single Instruction Multiple Thread。

一个CUDA core可以执行一个thread，一个SM中的CUDA core会被分成几个warp，由warp scheduler负责调度。GPU规定warp中所有thread在同一周期执行相同的指令，尽管这些thread执行同一程序地址，但可能产生不同的行为，比如分支结构。**一个SM同时并发的warp是有限的** ，由于资源限制，SM要为每个block分配共享内存，也要为每个warp中的thread分配独立的寄存器，所以SM的配置会影响其所支持的block和warp并发数量。

**GPU执行模型小结：**

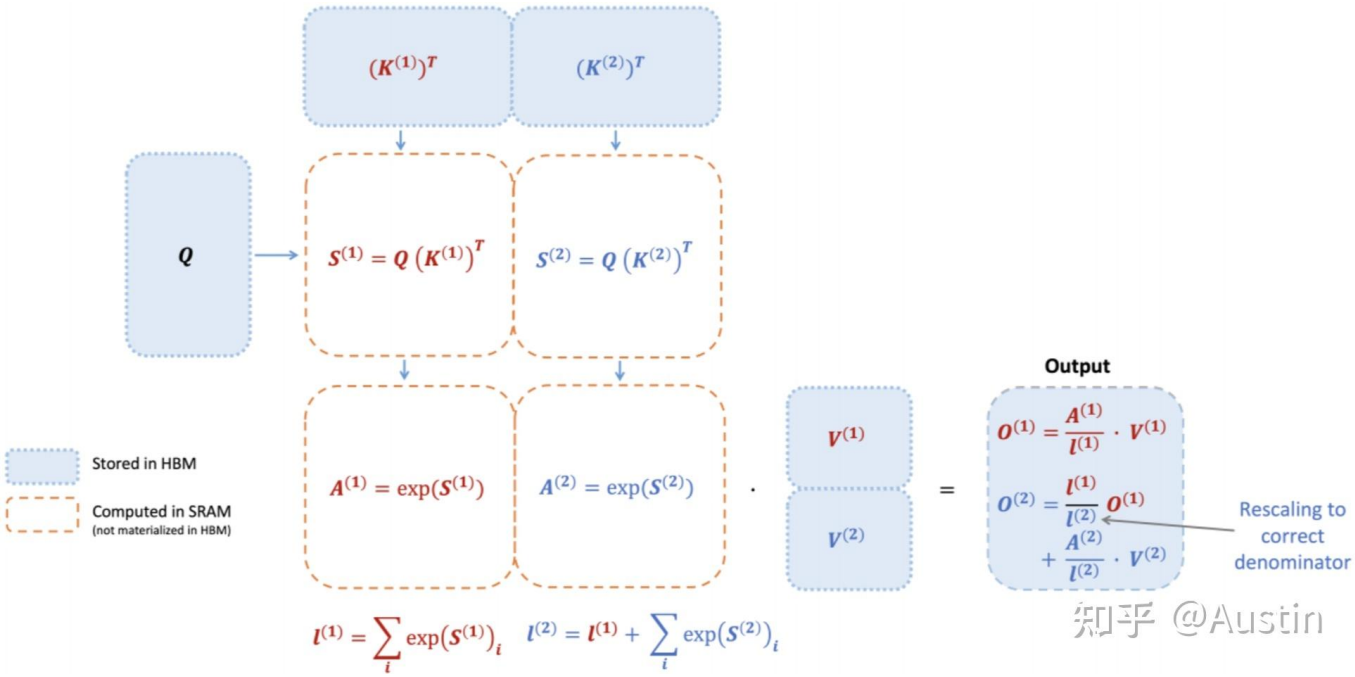
GPU有大量的threads用于执行操作（an operation，也称为a kernel）。这些thread组成了thread block，接着这些blocks被调度在SMs上运行。在每个thread block中，threads被组成了warps（32个threads为一组）。一个warp内的threads可以通过快速shuffle指令进行通信或者合作执行矩阵乘法。在每个thread block内部，warps可以通过读取/写入共享内存进行通信。每**个kernel从HBM加载数据到寄存器和SRAM中，进行计算，最后将结果写回HBM中。**

# FlashAttention

FlashAttention应用了tiling技术来减少内存访问，具体来说：

1. 从HBM中加载输入数据（K，Q，V）的一部分到SRAM中
2. 计算这部分数据的Attention结果
3. 更新输出到HBM，但是无需存储中间数据S和P

下图展示了一个示例：首先将K和V分成两部分（K1和K2，V1和V2，具体如何划分根据数据大小和GPU特性调整），根据K1和Q可以计算得到S1和A1，然后结合V1得到O1。接着计算第二部分，根据K2和Q可以计算得到S2和A2，然后结合V2得到O2。最后O2和O1一起得到Attention结果。



值得注意的是，输入数据K、Q、V是存储在HBM上的，中间结果S、A都不需要存储到HBM上。**通过这种方式，FlashAttention可以将内存开销降低到线性级别，并实现了2-4倍的加速，同时避免了对中间结果的频繁读写，从而提高了计算效率。**

# FlashAttention-2

经过铺垫，正式进入正文。我们先讲述FlashAttention-2对FlashAttention的改进，从而减少了非矩阵乘法运算（non-matmul）的FLOPs。然后说明如何将任务分配给不同的thread block进行并行计算，充分利用GPU资源。最后描述了如何在一个thread block内部分配任务给不同的warps，以减少访问共享内存次数。这些优化方案使得FlashAttention-2的性能提升了2-3倍。

# Algorithm

FlashAttention在FlashAttention算法基础上进行了调整，减少了非矩阵乘法运算（non-matmul）的FLOPs。这是因为现代GPU有针对matmul（GEMM）专用的计算单元（如Nvidia GPU上的Tensor Cores），效率很高。以A100 GPU为例，其FP16/BF16矩阵乘法的最大理论吞吐量为312 TFLOPs/s，但FP32非矩阵乘法仅有19.5 TFLOPs/s，即每个no-matmul FLOP比mat-mul FLOP昂贵16倍。为了确保高吞吐量（例如超过最大理论TFLOPs/s的50%），我们希望尽可能将时间花在matmul FLOPs上。

## Forward pass

通常实现Softmax算子为了数值稳定性（因为指数增长太快，数值会过大甚至溢出），会减去最大值：

$$\mathrm{softmax}(x) = \frac{e^{x_i}}{\sum e^{x_j}} = \frac{e^{x_i - x_{max}}}{\sum e^{x_j - x_{max}}} \tag{1}$$

这样带来的代价就是要对$x$遍历3次。

为了减少non-matmul FLOPs，本文在FlashAttention基础上做了两点改进：

1. 在计算局部attention时，先不考虑softmax的分母$\sum e^{x_i}$，即$\ell^{(i+1)}=e^{m^{(i)}-m^{(i+1)}}\ell^{(i)}+\text{rowsum}\left(e^{\mathbf{S}^{(i+1)}-m^{(i+1)}}\right)$，例如计算$\mathbf{O}^{(1)}$时去除了$\text{diag}\left(\ell^{(1)}\right)^{-1}$：

FlashAttention：$\mathbf{O}^{(1)}=\tilde{\mathbf{P}}(1)\mathbf{V}^{(1)}=\text{diag}\left(\ell^{(1)}\right)^{-1}e^{\mathbf{S}^{(1)}-m^{(1)}}\mathbf{V}^{(1)}$

FlashAttention-2：$\mathbf{O}^{(1)}=e^{\mathbf{S}^{(1)}-m^{(1)}}\mathbf{V}^{(1)}$

2. 由于去除了$\text{diag}\left(\ell^{(i)}\right)^{-1}$，更新$\mathbf{O}^{(i+1)}$时不需要rescale$\ell^{(i)}/\ell^{(i+1)}$，但是得弥补之前局部max值，例如示例中：

FlashAttention：
$$\mathbf{O}^{(2)}=\text{diag}\left(\ell^{(1)}/\ell^{(2)}\right)^{-1}\mathbf{O}^{(1)}+\text{diag}\left(\ell^{(2)}\right)^{-1}e^{\mathbf{S}^{(2)}-m^{(2)}}\mathbf{V}^{(2)}$$

FlashAttention-2：
$$\tilde{\mathbf{O}}^{(2)}=\text{diag}\left(e^{m^{(1)}-m^{(2)}}\right)\tilde{\mathbf{O}}^{(1)}+e^{\mathbf{S}^{(2)}-m^{(2)}}\mathbf{V}^{(2)}=e^{s^{(1)}-m}\mathbf{V}^{(1)}+e^{s^{(2)}-m}\mathbf{V}^{(2}$$

3. 由于更新$\mathbf{O}^{(i+1)}$未进行rescale，最后一步时需要将$\tilde{\mathbf{O}}^{(\text{last})}$乘以$\text{diag}\left(\ell^{(\text{last})}\right)^{-1}$来得到正确的输出，例如示例中：

FlashAttention-2：$\mathbf{O}=\text{diag}\left(\ell^{(2)}\right)^{-1}\tilde{\mathbf{O}}^{(2)}$

**简单示例的FlashAttention完整计算步骤（红色部分表示V1和V2区别）：**

$$
\begin{aligned}
m^{(1)}&=\text{rowmax}\left(\mathbf{S}^{(1)}\right)\in\mathbb{R}^{B_r}\\
\ell^{(1)}&=\text{rowsum}\left(e^{\mathbf{S}^{(1)}-m^{(1)}}\right)\in\mathbb{R}^{B_r}\\
\tilde{\mathbf{P}}^{(1)}&=\text{diag}\left(\ell^{(1)}\right)^{-1}e^{\mathbf{S}^{(1)}-m^{(1)}}\in\mathbb{R}^{B_r\times B_c}\\
\mathbf{O}^{(1)}&=\tilde{\mathbf{P}}(1)\mathbf{V}^{(1)}=\text{diag}\left(\ell^{(1)}\right)^{-1}e^{\mathbf{S}^{(1)}-m^{(1)}}\mathbf{V}^{(1)}\in\mathbb{R}^{B_r\times d}\\
m^{(2)}&=\max\left(m^{(1)},\text{rowmax}\left(\mathbf{S}^{(2)}\right)\right)=m\\
\ell^{(2)}&=e^{m^{(1)}-m^{(2)}}\ell^{(1)}+\text{rowsum}\left(e^{\mathbf{S}^{(2)}-m^{(2)}}\right)=\text{rowsum}\left(e^{\mathbf{S}^{(1)}-m}\right)+\text{rowsum}\left(e^{\mathbf{S}^{(2)}-m}\right)=\ell\\
\tilde{\mathbf{P}}^{(2)}&=\text{diag}\left(\ell^{(2)}\right)^{-1}e^{\mathbf{S}^{(2)}-m^{(2)}}\\
\mathbf{O}^{(2)}&=\text{diag}\left(\ell^{(1)}/\ell^{(2)}\right)^{-1}\mathbf{O}^{(1)}+\tilde{\mathbf{P}}^{(2)}\mathbf{V}^{(2)}=\text{diag}\left(\ell^{(2)}\right)^{-1}e^{s^{(1)}-m}\mathbf{V}^{(1)}+\text{diag}\left(\ell^{(2)}\right)^{-1}e^{s^{(2)}-m}\mathbf{V}^{(2)}=\mathbf{O}
\end{aligned}
$$



**FlashAttention-2的完整计算步骤（红色部分表示V1和V2区别）：**

1. 论文中多了$\tilde{\mathbf{P}}^{(2)}=\operatorname{diag}\left(\ell^{(2)}\right)^{-1} e^{\mathbf{S}^{(2)}-m^{(2)}}$，应该是笔误，这里我删除了！
2. 论文中Algorithm 1伪代码中$\tilde{\mathbf{P}}_i^{(j)}=\exp(\mathbf{S}_i^{(j)}-m_i^{(j)})$是没有问题的！

$$
\begin{aligned}
m^{(1)} &= \operatorname{rowmax}\left(\mathbf{S}^{(1)}\right) \in \mathbb{R}^{B_r} \\
\ell^{(1)} &= \operatorname{rowsum}\left(e^{\mathbf{S}^{(1)}-m^{(1)}}\right) \in \mathbb{R}^{B_r} \\
\mathbf{O}^{(1)} &= e^{\mathbf{S}^{(1)}-m^{(1)}}\mathbf{V}^{(1)} \in \mathbb{R}^{B_r \times d} \\
m^{(2)} &= \max\left(m^{(1)}, \operatorname{rowmax}\left(\mathbf{S}^{(2)}\right)\right) = m \\
\ell^{(2)} &= e^{m^{(1)}-m^{(2)}}\ell^{(1)} + \operatorname{rowsum}\left(e^{\mathbf{S}^{(2)}-m^{(2)}}\right) = \operatorname{rowsum}\left(e^{\mathbf{S}^{(1)}-m}\right) + \operatorname{rowsum}\left(e^{\mathbf{S}^{(2)}-m}\right) = \ell \\
\tilde{\mathbf{O}}^{(2)} &= \operatorname{diag}\left(e^{m^{(1)}-m^{(2)}}\right)\tilde{\mathbf{O}}^{(1)} + e^{\mathbf{S}^{(2)}-m^{(2)}}\mathbf{V}^{(2)} = e^{s^{(1)}-m}\mathbf{V}^{(1)} + e^{s^{(2)}-m}\mathbf{V}^{(2)} \\
\mathbf{O}^{(2)} &= \operatorname{diag}\left(\ell^{(2)}\right)^{-1}\tilde{\mathbf{O}}^{(2)} = \mathbf{O}
\end{aligned}
$$



有了上面分析和之前对[FlashAttention的讲解]，再看下面伪代码就没什么问题了。

**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 $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ 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.
2: Divide the output $\mathbf{O} \in \mathbb{R}^{N \times d}$ into $T_r$ blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, and divide the logsumexp $L$ into $T_r$ blocks $L_i, \ldots, L_{T_r}$ 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 $m_i^{(j)} = \max(m_i^{(j-1)}, \operatorname{rowmax}(\mathbf{S}_i^{(j)})) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - m_i^{(j)}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\ell_i^{(j)} = e^{m_i^{j-1} - m_i^{(j)}} \ell_i^{(j-1)} + \operatorname{rowsum}(\tilde{\mathbf{P}}_i^{(j)}) \in \mathbb{R}^{B_r}$.
10: On chip, compute $\mathbf{O}_i^{(j)} = \operatorname{diag}(e^{m_i^{(j-1)} - m_i^{(j)}})\mathbf{O}_i^{(j-1)} + \tilde{\mathbf{P}}_i^{(j)}\mathbf{V}_j$.
11: **end for**
12: On chip, compute $\mathbf{O}_i = \operatorname{diag}(\ell_i^{(T_c)})^{-1}\mathbf{O}_i^{(T_c)}$.
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 $L$.



Causal masking是attention的一个常见操作，特别是在自回归语言建模中，需要对注意力矩阵S应用因果掩码（即任何S ，其中 > 的条目都设置为−∞）。

1. 由于FlashAttention和FlashAttention-2已经通过块操作来实现，对于所有列索引都大于行索引的块（大约占总块数的一半），我们可以跳过该块的计算。这比没有应用因果掩码的注意力计算速度提高了1.7-1.8倍。

2. 不需要对那些行索引严格小于列索引的块应用因果掩码。这意味着对于每一行，我们只需要对1个块应用因果掩码。

# Parallelism

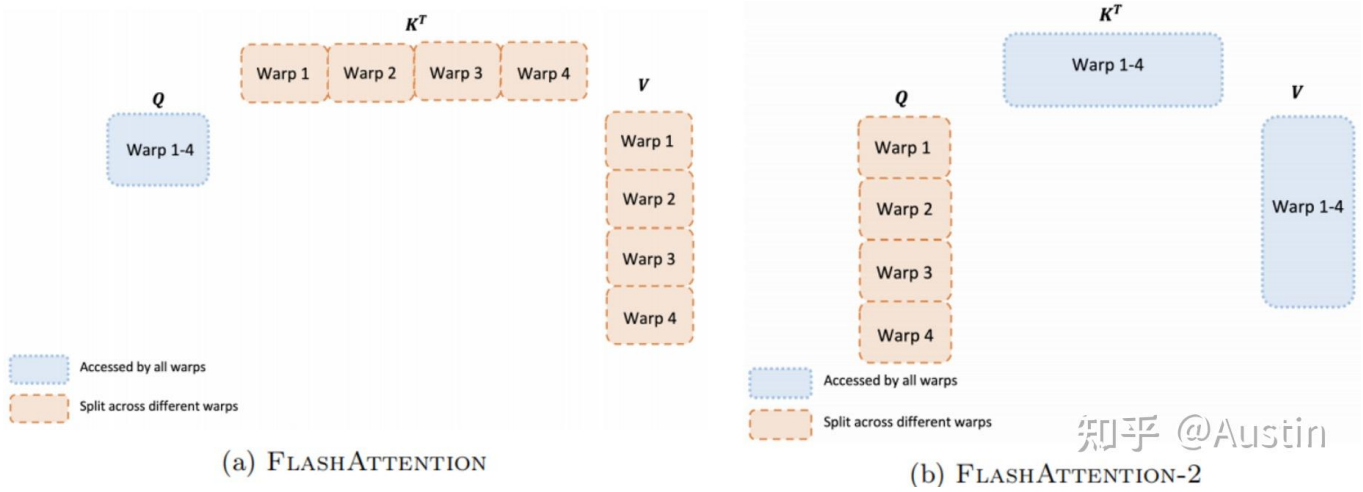
FlashAttention在batch和heads两个维度上进行了并行化：使用一个thread block来处理一个attention head，总共需要thread block的数量等于batch size × number of heads。每个block被调到到一个SM上运行，例如A100 GPU上有108个SMs。当block数量很大时（例如≥80），这种调度方式是高效的，因为几乎可以有效利用GPU上所有计算资源。

但是在处理长序列输入时，由于内存限制，通常会减小batch size和head数量，这样并行化成都就降低了。因此，FlashAttention-2还在序列长度这一维度上进行并行化，显著提升了计算速度。此外，当batch size和head数量较小时，在序列长度上增加并行性有助于提高GPU占用率。

**Forward pass.** FlashAttention算法有两个循环，$K, V$在外循环$j$，$Q, O$在内循环$i$。FlashAttention-2将$Q$移到了外循环$i$，$K, V$移到了内循环$j$，**由于改进了算法使得warps之间不再需要相互通信去处理 $Q_i$，所以外循环可以放在不同的thread block上。这个交换的优化方法是由Phil Tillet在Triton[17]提出并实现的。**

# Work Partitioning Between Warps

上一节讨论了如何分配thread block，然而在每个thread block内部，我们也需要决定如何在不同的warp之间分配工作。我们通常在每个thread block中使用4或8个warp，如下图所示。




*Work partitioning between different warps in the forward pass*

**FlashAttention forward pass.** 如下图所示，外循环对$K, V$在输入序列$N$上遍历，内循环对$Q$在$N$上遍历。对于每个block，FlashAttention将$K$和$V$分别分为4个warp，并且所有warp都可以访问$Q$。$K$的warp乘以$Q$得到$S$的一部分$S_{ij}$，然后$S_{ij}$经过局部softmax后还需要乘以$V$的一部分得到$O_i$。然而，每次外循环$j$++都需要更新一遍$O_i$（对上一次$O_i$先rescale再加上当前值），这就导致每个warp需要从HBM频繁读写$Q_i$以累加出总结果。这种方式被称为“split-K”方案，是非常低效的，因为所有warp都需要从HBM频繁读写中间结果（$Q_i, O_i, m_i, \ell_i$）。

> 论文中原话是”However, this is inefficient since all warps need to write their intermediate results out to**shared memory** , synchronize, then add up the intermediate results.”，说的是shared memory而非HBM，但是结合下图黄色框部分推断，我认为是HBM。

**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 $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ 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.

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.

**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 $\mathbf{O}$.



**FlashAttention-2 forward pass.** 如下图所示，FlashAttention-2将$Q$移到了外循环$i$，$K, V$移到了内循环$j$，并将$Q$分为4个warp，所有warp都可以访问$K$和$V$。**这样做的好处是，原来FlashAttention每次内循环 $i$++会导致 $O_i$也变换（而$O_i$需要通过HBM读写），现在每次内循环 $j$++处理的都是 $O_i$，此时 $O_i$是存储在SRAM上的，代价远小于HBM。**

**Algorithm 1** FLASHATTENTION-2 forward pass

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

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.

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.

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

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

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

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

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

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

    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), $\ell_i^{(j)} = e^{m_i^{j-1} - m_i^{(j)}} \ell_i^{(j-1)} + \text{rowsum}(\tilde{\mathbf{P}}_i^{(j)}) \in \mathbb{R}^{B_r}$.

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

  **end for**

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

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

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

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

**end for**

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

# References


- 理解CUDA中的thread,block,grid和warp - 三七和酒的文章 - 知乎


> 本文使用 Zhihu On VSCode创作并发布