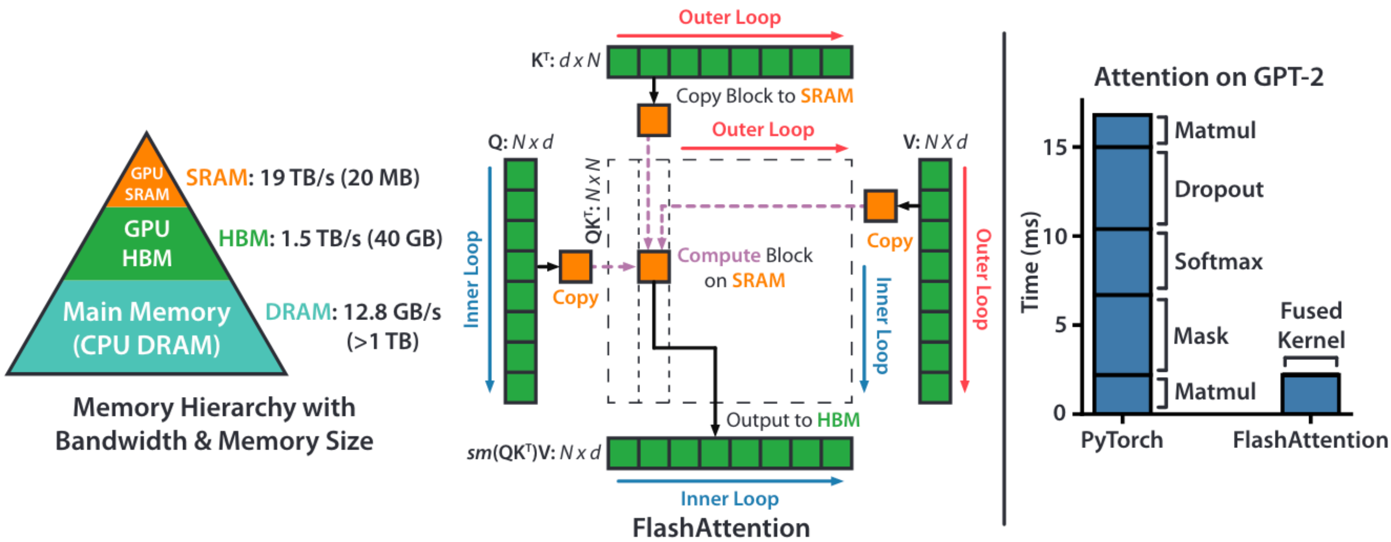
Flash attention-原理

自注意力模块的计算和空间复杂度为 $O(N^2)$，N为序列长度（假设 $Q, K, V \in R^{N\times d}$，batch大小为b, 注意力头数为a，计算 QK^T 的计算复杂度为 $2bN^2d$，用fp16存储 QK^T 的空间复杂度为 $2bN^2a$ ）。这导致在处理长序列时速度变慢且内存需求巨大，也限制了模型输入输出的序列长度。

SRAM的运算速度很快，但是空间远小于HBM显存。自注意力对应的空间需求巨大，对应的将数据从HBM搬运到SRAM也消耗了大量时间。FlashAttention提出优化算法来提高注意力模块的计算速度。



传统attention

例如在GPT3中，一般N=2048，d=128。对于传统算法， `S` 和 `P` 远大于Q K V O，在SRAM中放不下，需要在HBM中进行存储，计算时需要反复访问HBM，搬运数据浪费了大量时间。

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N\times N}, \quad \mathbf{P} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \quad \mathbf{O} = \mathbf{P}\mathbf{V} \in \mathbb{R}^{N\times d}, \tag{1}$$

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 $\mathbf{O}$.

需要把输入 $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ 从HBM中读取，并计算完毕后把输出 $\mathbf{O}$ 写入到HBM中。

第一步把 $\mathbf{Q}, \mathbf{K}$ 读取出来计算出 $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$，然后把 $\mathbf{S}$ 存回去，内存访问复杂度 $\Theta\left(Nd + N^2\right)$
。

第二步把 $\mathbf{S}$ 读取出来计算出 $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$，然后把 $\mathbf{P}$ 存回去，内存访问复杂度 $\Theta\left(N^2\right)$。

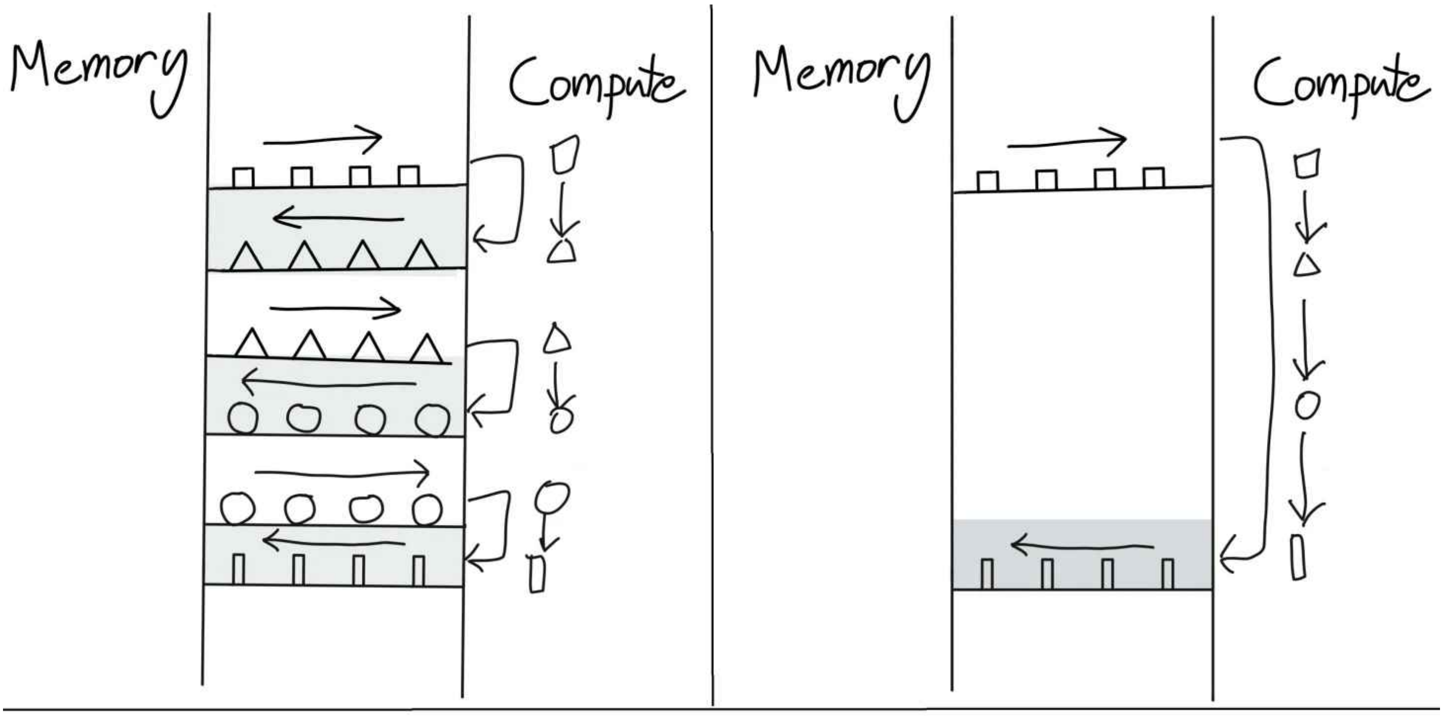
第三步把 $\mathbf{V}, \mathbf{P}$ 读取出来计算出 $\mathbf{O}=\mathbf{PV}$，然后计算出结果 $\mathbf{O}$，内存访问复杂度 $\Theta\left(Nd+N^2\right)$。

综上所述，整体的内存访问复杂度为 $\Theta\left(Nd+N^2\right)$。总结一下标准注意力面临两个问题：

1. 显存占用多，由于计算过程中存储完整注意力矩阵P和S，需要 $O(N^2)$ 的空间。
2. HBM读写次数多，需要传输的数据量大。

Flash attention

将从输入的 $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ 到输出 $\mathbf{O}$ 的整个过程进行融合，省去 $\mathbf{S}, \mathbf{P}$ 矩阵的存储开销。此外，为了将计算过程的结果完全存储在SRAM中，摆脱对HBM的依赖，采用分片（tiling）操作，每次进行部分计算，确保这些计算结果能在SRAM内进行交互，待得到对应的结果后再进行输出。总之，tiling分块计算使得我们可以用一个CUDA kernel来执行注意力的所有操作。从HBM中加载输入数据，在SRAM中执行所有的计算操作(矩阵乘法、mask、softmax、dropout、矩阵乘法)，再将计算结果写回到HBM中。



Operator Fusion Simplified

上图右面表示分块后的状态，由于分块使得SRAM能存储下所有中间态，因而可以在SRAM执行完所有计算操作后才写回到HBM中，降低了访问HBM的次数。具体如何分片参考 https://www.bilibili.com/video/BV1Zz4y1q7FX/，

对于float32和bfloat16来说，当 x≥89 时，就会变得很大甚至变成inf，故为了避免发生数值溢出的问题，需要对指数项减去最大值 $m(x)$，

$$m(x) := \max_i x_i, \quad f(x) := \left[\begin{array}{lll} e^{x_1-m(x)} & \ldots & e^{x_B-m(x)} \end{array}\right], \quad \ell(x) := \sum_i f(x)_i, \quad \operatorname{softmax}(x) := \frac{f(x)}{\ell(x)} \tag{2}$$

对于分块计算，矩阵乘法和逐点操作(scale，mask，dropout)的分块计算是容易实现的。需要注意的是，在进行softmax分块计算时，需要完整的一行作为输入数据，因为其分母需要对完整一行求和。当我们将输入进行分片后，无法对完整的行数据执行Softmax操作。我们可以通过如下所示方法来获得与完整行Softmax相同的结果，而无需使用近似操作。

$$m(x)=m\left(\left[x^{(1)} x^{(2)}\right]\right)=\max \left(m\left(x^{(1)}\right), m\left(x^{(2)}\right)\right), \quad f(x)=\left[\begin{array}{ll} e^{m\left(x^{(1)}\right)-m(x)} f\left(x^{(1)}\right) & e^{m\left(x^{(2)}\right)-m(x)} f\left(x^{(2)}\right) \end{array}\right],$$
$$\ell(x)=\ell\left(\left[x^{(1)} x^{(2)}\right]\right)=e^{m\left(x^{(1)}\right)-m(x)} \ell\left(x^{(1)}\right)+e^{m\left(x^{(2)}\right)-m(x)} \ell\left(x^{(2)}\right), \quad \operatorname{softmax}(x)=\frac{f(x)}{\ell(x)}. \quad (3$$

原理：参考https://www.zhihu.com/question/611236756/answer/3132304304

考虑一个向量 $x \in R^{2d}$ ，将其分为两块 $x=[x^{(1)}, x^{(2)}]$。采用分块计算，首先计算 x^1 ，

(9) $m(x^{(1)}) = \max([x_1^{(1)}, x_2^{(1)}, \ldots, x_B^{(1)}])$

(10) $f(x^{(1)}) = [e^{x_1^{(1)} - m(x^{(1)})}, \ldots, e^{x_B^{(1)} - m(x^{(1)})}]$

(11) $l(x^{(1)}) = \sum_i f(x^{(1)})_i$

(12) $\operatorname{softmax}(x^{(1)}) = \frac{f(x^{(1)})}{l(x^{(1)})}$

由此计算得到的 $softmax(x^{(1)})$ 并不是子向量 $x^{(1)}$ 的结果，因为（10）中的最大值应该是整个向量的最大值，（11）中的求和项应该是整个向量的求和值。

处理完 $x^{(1)}$ 后，显存中只保留 $m(x^{(1)})$ 和 $l(x^{(1)})$ ，节省内存开销，此外保存两个全局标量 m_{max} 和 l_{all} ，分别表示当前最大值和全局exp的求和项，暂时分别等于 $m(x^{(1)})$ 和 $l(x^{(1)})$。对于 $x^{(2)}$ 采用同样方式处理：

$$(13)\ m(x^{(2)}) = \max([x_1^{(2)}, x_2^{(2)}, \ldots, x_B^{(2)}])$$

$$(14)\ f(x^{(2)}) = [e^{x_1^{(2)} - m(x^{(2)})}, \ldots, e^{x_B^{(2)} - m(x^{(2)})}]$$

$$(15)\ l(x^{(2)}) = \sum_i f(x^{(2)})_i$$

$$(16)\ \mathrm{softmax}(x^{(2)}) = \frac{f(x^{(2)})}{l(x^{(2)})}$$

$softmax(x^{(2)})$ 也是局部的,利用 $x^{(2)}$ 更新 m_{max} 和 l_{all}：

$$(17)\ m_{max}^{new} = \max([m_{max}, m(x^{(2)})])$$

$$(18)\ l_{all}^{new} = e^{m_{max} - m_{max}^{new}} l_{all} + e^{m_{x^{(2)}} - m_{max}^{new}} l(x^{(2)})$$

为了消除局部性，就需要利用到新的 m_{max} 和 l_{all} 。对于 $l(x^{(2)})$ ，需要将其减去的 $m(x^{(2)})$ 替换成 m_{max} ，为此做如下变化：

$$(20)\quad \begin{aligned} l^{new}(x^{(2)}) &= l(x^{(2)}) \cdot e^{m(x^{(2)}) - m_{max}^{new}} \\ &= \sum_i e^{x_i^{(2)} - m(x^{(2)})} \cdot e^{m(x^{(2)}) - m_{max}^{new}} \\ &= \sum_i e^{x_i^{(2)} - m_{max}^{new}} \end{aligned}$$

总之，当需要把某个更新为“全局的”时，只要将其乘以一个项： $e^{m - m_{max}^{new}}$ ，其中 m表示当前l对应的最大值， m_{max}^{new} 表示当前最大值。同理也对 l_{all} 去除局部性。为了去除 $f(x^{(2)})$ 的局部性，也做相同的变化

$$
\begin{aligned}
f^{new}(x^{(2)}) &= f(x^{(2)}) \cdot e^{m(x^{(2)})-m_{max}^{new}} \\
&= [e^{x_1^{(2)}-m(x^{(2)})}, \ldots, e^{x_B^{(2)}-m(x^{(2)})}] \cdot e^{m(x^{(2)})-m_{max}^{new}} \\
&= [e^{x_1^{(2)}-m_{max}^{new}}, \ldots, e^{x_B^{(2)}-m_{max}^{new}}]
\end{aligned} \tag{21}
$$

因此，全局的 $softmax^{new}(x^{(2)})$ 就更新成了：

$$
softmax^{new}(x^{(2)}) = \frac{f^{new}(x^{(2)})}{l_{all}^{new}} = \frac{f(x^{(2)}) \cdot e^{m(x^{(2)}-m_{max}^{new})}}{l_{all}^{new}} = \frac{softmax(x^{(2)}) \cdot l(x^{(2)}) \cdot e^{m(x^{(2)}-m_{max}^{new})}}{l_{all}^{new}}
$$

将公式中的 $x^{(2)}$ 替换成 $x^{(1)}$ 就可以对 $x^{(1)}$ 的softmax进行更新。上述其实是一个增量计算的过程

1. 我们首先计算一个分块的局部softmax值，然后存储起来
2. 当处理完下一个分块时，可以根据此时的新的全局最大值和全局exp求和项来更新旧的softmax值，接着再处理下一个分块，然后再更新
3. 当处理完所有分块后，此时的所有分块的softmax值都是“全局的”

整个过程中不需要保存 $x^{(1)}$ 和 $x^{(2)}$ 在SRAM中，要知道softmax的矩阵S是 $N \times N$ 的，SRAM一次存储不下，如果每次增量计算都要重复加载之前的 $x^{(1)}$,则反而会增加HBM访问次数。