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

演讲者：陈驰水

日期：2025年7月17日

论文背景与前置知识

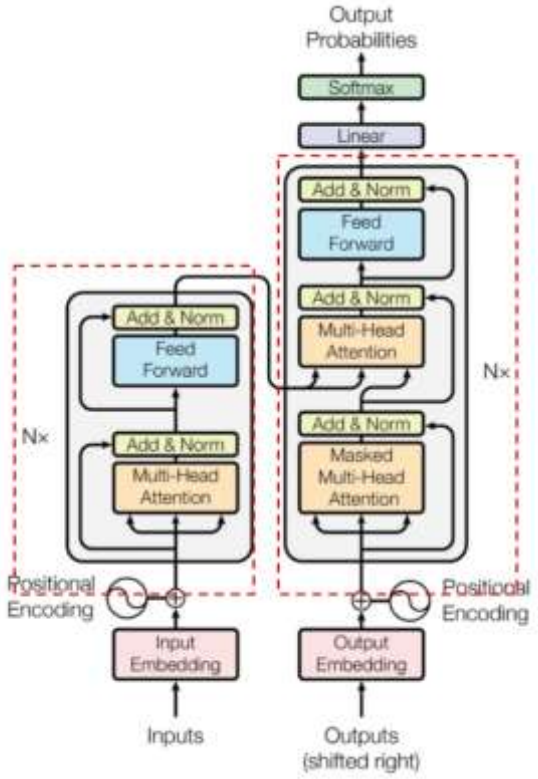
为什么需要 FlashAttention?
Self-Attention 机制?
GPU 内存层级结构是怎么样的?

为什么需要 FlashAttention？

Transformer 的注意力机制存在两个关键问题：

1. 每个注意力头需要计算 $\boldsymbol{QK^T}$，其计算和内存复杂度为 $\mathbf{O}(\boldsymbol{n^2})$

2. 内存带宽瓶颈：注意力虽在理论上计算量不大，但在实际运行中受限于 GPU 的 IO 带宽。大量中间张量tensor需要频繁地从显存（HBM）加载和写回，成为性能瓶颈。




```python
def forward(self,
            query:Tensor,
            key: Tensor,
            value: Tensor,
            mask:Optional[Tensor]=None):
    if mask is not None:
        mask = mask.unsqueeze(1)
    batch_size = query.size(0)

    # 1) Apply W^Q, W^K, W^V to generate new query, key, value
    query, key, value \
        = [proj_weight(x).view(batch_size, -1, self.num_heads, self.k_dim).transpose(1, 2)
           for proj_weight, x in zip(self.proj_weights, [query, key, value])] # -1 equals to seq_len

    # 2) Calculate attention score and the out
    out, self.attention_score = attention(query, key, value, mask=mask,
                                          dropout=self.dropout)

    # 3) "Concat" output
    out = out.transpose(1, 2).contiguous() \
          .view(batch_size, -1, self.num_heads * self.k_dim)

    # 4) Apply W^O to get the final output
    out = self.proj_weights[-1](out)

    return out
```

为什么需要 FlashAttention？

Transformer 的注意力机制存在两个关键问题：

1. 每个注意力头需要计算 QK^T，其计算和内存复杂度为 $O(n^2)$

2. 内存带宽瓶颈：注意力虽在理论上计算量不大，但在实际运行中受限于 GPU 的 IO 带宽。大量中间张量tensor需要频繁地从显存（HBM）加载和写回，成为性能瓶颈。

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

d_k是Q,K矩阵的列数，即向量维度

```python
def forward(self,
            query:Tensor,
            key: Tensor,
            value: Tensor,
            mask:Optional[Tensor]=None):
    if mask is not None:
        mask = mask.unsqueeze(1)
    batch_size = query.size(0)

    # 1) Apply W^Q, W^K, W^V to generate new query, key, value
    query, key, value \
        = [proj_weight(x).view(batch_size, -1, self.num_heads, self.k_dim).transpose(1, 2)
           for proj_weight, x in zip(self.proj_weights, [query, key, value])] # -1 equals to seq_len

    # 2) Calculate attention score and the out
    out, self.attention_score = attention(query, key, value, mask=mask,
                                          dropout=self.dropout)

    # 3) "Concat" output
    out = out.transpose(1, 2).contiguous() \
        .view(batch_size, -1, self.num_heads * self.k_dim)

    # 4) Apply W^O to get the final output
    out = self.proj_weights[-1](out)

    return out
```

GPU 内存层级结构是怎么样的？

访问速度寄存器（Register）：

最快、容量最小，每个线程私有

共享内存（SRAM）：

Block 内线程共享，速度较快

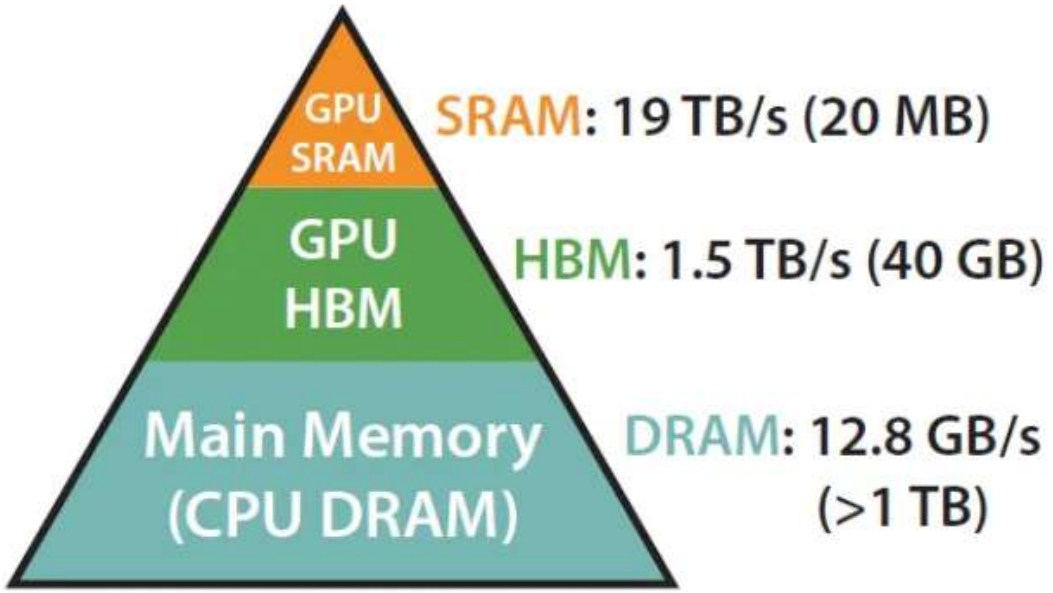
快全局内存（HBM）：

所有线程共享，容量大，但速度慢

主存（DRAM）

最慢，但容量最大

传统 Attention 实现中，大多数中间张量存在于 快全局内存（HBM）中，频繁 IO 降低了性能。



Memory Hierarchy with Bandwidth & Memory Size

论文贡献 FlashAttention

快速、内存高效、精确、IO感知

快速

本方法在训练速度上显著优于现有方案：在 BERT-large（序列长度 512）上，训练速度比 MLPerf 1.1 的记录快 15%；在 GPT-2（序列长度 1K）上，训练速度比 HuggingFace 和 Megatron-LM 的基线实现快了 3 倍；在 Long Range Arena（序列长度 1K 到 4K）任务中，训练速度提升了 2.4 倍。

内存高效

相比于传统注意力机制在序列长度上的二次内存复杂度 $\mathbf{O}(n^2)$，本方法显著降低了内存使用，达到了亚二次甚至线性的复杂度 $\mathbf{O}(n^1)$。

精确计算

与一些通过稀疏化或低秩矩阵等方式进行近似计算的注意力优化方法不同，本方法在保持高效的同时，计算结果与原始注意力机制完全一致，确保了模型精度不受影响。

I/O 感知

该方法在设计上充分考虑了硬件的内存访问和数据传输效率，相比传统注意力机制，对底层系统资源更加“敏感”，从而在实际运行中实现了更优的整体性能表现。

论文算法与创新点

IO-aware 的优化策略；将计算过程重构为“分块 + 融合”形式；
在寄存器和共享内存中完成全部计算；最大限度减少显存访问

原始attention的问题

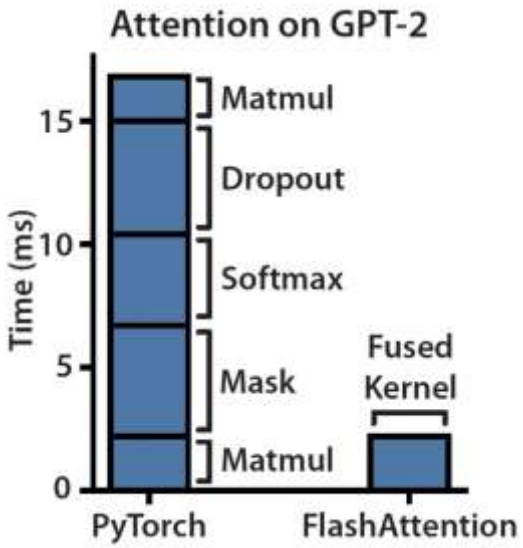
该算法的主要瓶颈在于：每次计算都需要将数据从 GPU 的高带宽内存（HBM）加载到片上高速缓存（如寄存器或 SRAM），计算完成后又需将结果写回 HBM。这种频繁的数据传输类似于 CPU 中寄存器与主内存之间的关系，极大限制了计算效率。因此，一个显而易见且有效的优化方向是：尽可能减少这种来回的数据移动。这正是所谓的“Kernel Fusion（核函数融合）”

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 $\mathbf{O}$.

从左侧性能分析栏可以观察到，尽管大部分的 FLOPs（每秒浮点运算次数） 都集中在矩阵乘法操作上，但实际消耗时间最多的却是 masking、softmax 和 dropout 等操作。这表明：虽然这些操作计算量较小，但由于其 IO 强度高或缺乏并行性，反而成为了性能瓶颈。



Flash-attention核心算法

Step 0：显存容量不是瓶颈高带宽内存（HBM）容量通常以 GB 为单位。例如，NVIDIA RTX 3090 拥有 24GB 显存，A100 则提供 40～80GB。因此，在显存中分配用于 Attention 计算的 Q、K 和 V 张量并不困难，不是主要限制因素。

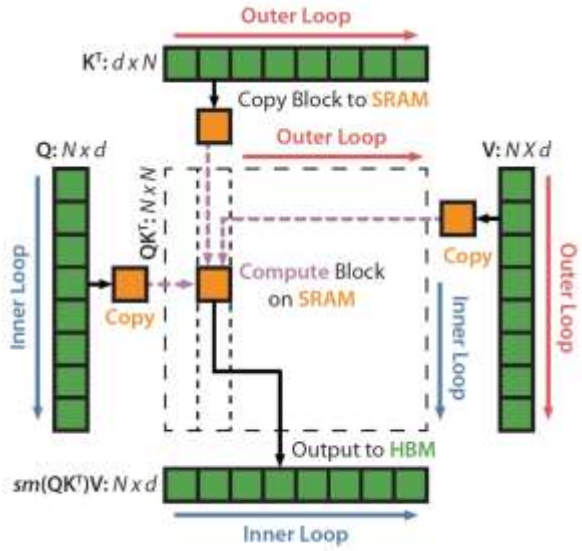
Step 1：确定块的大小为了优化内存访问和计算效率，将矩阵分成多个小块（tiles）进行处理。

Step 2：在开始计算之前，我们首先对中间变量进行初始化。输出矩阵 O 被初始化为全零矩阵，因为后续的计算将逐步将各个块的中间结果累加到其中，因此以 0 作为初始值是合理的。类似地，我们初始化一个向量 l，用于存储每一行 softmax 的归一化因子（即指数和），后续会在每个 block 计算中不断对其进行累加。

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 ℓ 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 $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1} (\mathrm{diag}(\ell_i) e^{m_i - m_i^{\mathrm{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}$, $m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.



Flash-attention核心算法

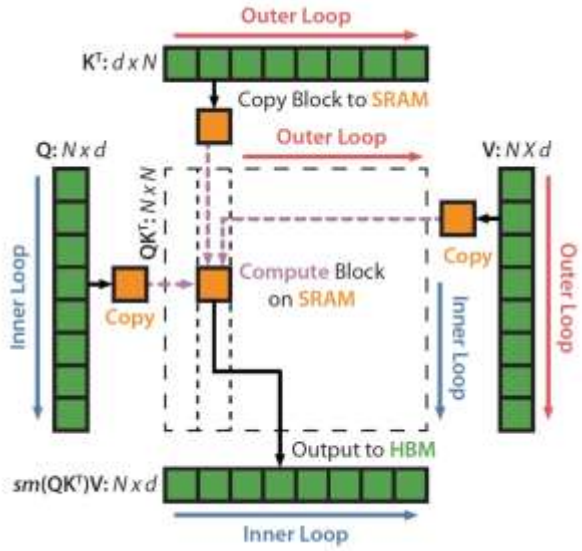
Step 3：将 Q、K、V 分块，Q 会按行方向被切分成多个大小为 Br×d 的小块，一共 Tr 个；K 和 V 会按列方向被切分成大小为 Bc×d 的小块，一共 Tc 个。每一次的 attention 计算，只会处理一个 Q 的小块和一个 K/V 的小块。Q 和转置后的 K 先进行乘法，得到注意力权重；再与对应的 V 块相乘，得到当前的输出块。最终，每个 Q 块经过所有 K/V 块的累加后，得到对应的输出部分。这样做可以避免一次性加载完整的 QKˆT 和 softmax 结果，降低显存压力。

Step 4：输出和中间变量也要分块为了配合前面对 Q 的分块，输出矩阵 O 也要按行切成多个 Br×d 的小块，用来存放每个 Q 块的计算结果。同时，辅助向量 l 和 m（长度为 N）也要按 Br 个一组分块，用来记录这些行的 softmax 累加值和当前最大值，方便在每个 Q 块内部更新。这样做可以让每个计算块都有对应的输出和中间状态，便于逐块处理。

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 ℓ 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 $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1} (\mathrm{diag}(\ell_i) e^{m_i - m_i^{\mathrm{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}$, $m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.



Flash-attention核心算法

Step 5：遍历每个 Q 块。依次遍历每一个 Q 的小块（共有 Tr 个），每个小块包含 Br 行。对于每个 Q 块，我们会取出对应的 O、l 和 m 的小块，用于存储当前块的中间输出、softmax 累加值和最大值。

Step 6：遍历所有 K/V 块，每次从中取出一个 K 块和对应的 V 块，和当前的 Q 块一起进行一次 attention 块计算。

计算注意力分数：将当前 Q 块与 K 块转置做矩阵乘法，得到一个大小为 Br×Bc 的注意力得分矩阵。

更新最大值 m：对当前行的得分进行比较，更新对应的最大值（用于 softmax 的稳定性处理）。

计算指数并归一化：使用更新后的最大值，对得分矩阵进行 softmax 近似计算（用累加方式维护 l）。

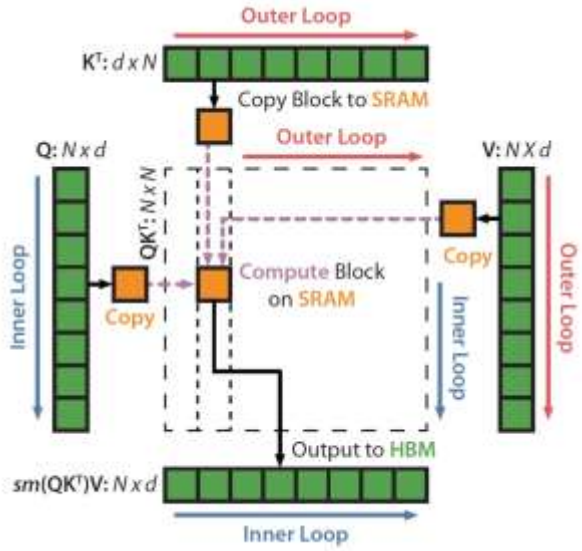
加权求和：将 softmax 权重与当前 V 块相乘，得到一个大小为 Br×d 的加权值，累加到对应的 O 块中。

Step 7：合并所有块，得到最终输出当遍历完所有的 Q 块，并分别完成了与所有 K/V 块的注意力计算后，每个输出块 O 都已累计完成。

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 ℓ 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 $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12: Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1} (\mathrm{diag}(\ell_i) e^{m_i - m_i^{\mathrm{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13: Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}$, $m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
14: **end for**
15: **end for**
16: Return $\mathbf{O}$.



感谢聆听，期待您的指导！

演讲者：陈驰水　　日期：2025年7月17日

FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning

演讲者：陈驰水

日期：2025年7月31日

论文背景与前置知识

并行性受限：v1 中 block-wise 的执行顺序限制了 GPU 多线程利用率。
可扩展性不足：在长序列或高 batch size 下表现不稳定。
代码复杂度高：难以扩展和维护。

Transformer 的注意力机制存在两个关键问题：

1. 每个注意力头需要计算 QK^T，其计算和内存复杂度为 $O(n^2)$

2. 内存带宽瓶颈：注意力虽在理论上计算量不大，但在实际运行中受限于 GPU 的 IO 带宽。大量中间张量tensor需要频繁地从显存（HBM）加载和写回，成为性能瓶颈。

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

d_k是Q,K矩阵的列数，即向量维度

```python
def forward(self,
            query:Tensor,
            key: Tensor,
            value: Tensor,
            mask:Optional[Tensor]=None):
    if mask is not None:
        mask = mask.unsqueeze(1)
    batch_size = query.size(0)

    # 1) Apply W^Q, W^K, W^V to generate new query, key, value
    query, key, value \
        = [proj_weight(x).view(batch_size, -1, self.num_heads, self.k_dim).transpose(1, 2)
           for proj_weight, x in zip(self.proj_weights, [query, key, value])] # -1 equals to seq_len

    # 2) Calculate attention score and the out
    out, self.attention_score = attention(query, key, value, mask=mask,
                                          dropout=self.dropout)

    # 3) "Concat" output
    out = out.transpose(1, 2).contiguous() \
        .view(batch_size, -1, self.num_heads * self.k_dim)

    # 4) Apply W^O to get the final output
    out = self.proj_weights[-1](out)

    return out
```

GPU 内存层级结构是怎么样的？

访问速度寄存器（Register）：

最快、容量最小，每个线程私有

共享内存（SRAM）：

Block 内线程共享，速度较快

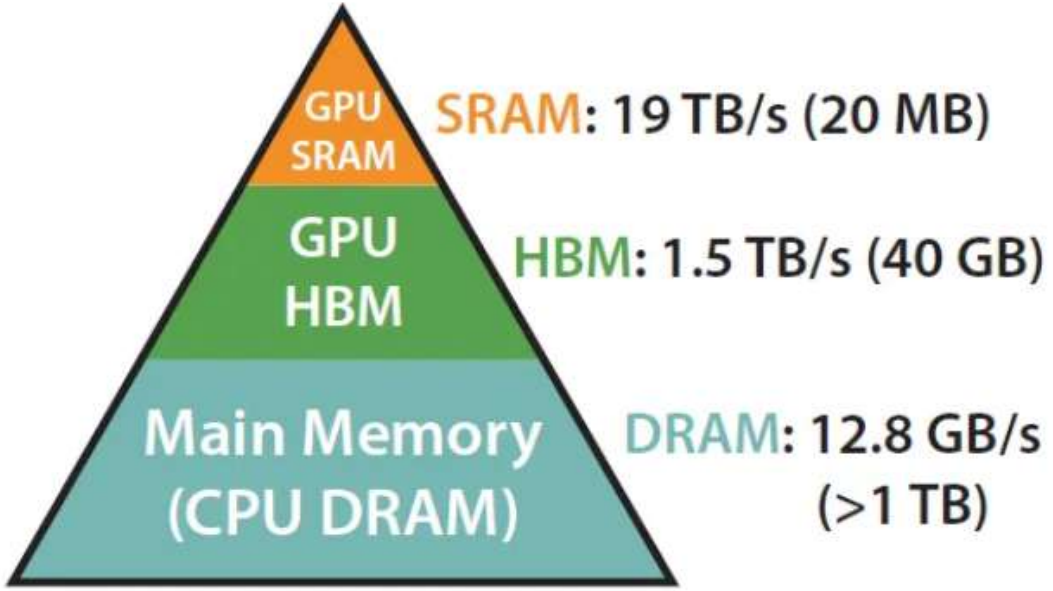
快全局内存（HBM）：

所有线程共享，容量大，但速度慢

主存（DRAM）

最慢，但容量最大

传统 Attention 实现中，大多数中间张量存在于 快全局内存（HBM）中，频繁 IO 降低了性能。

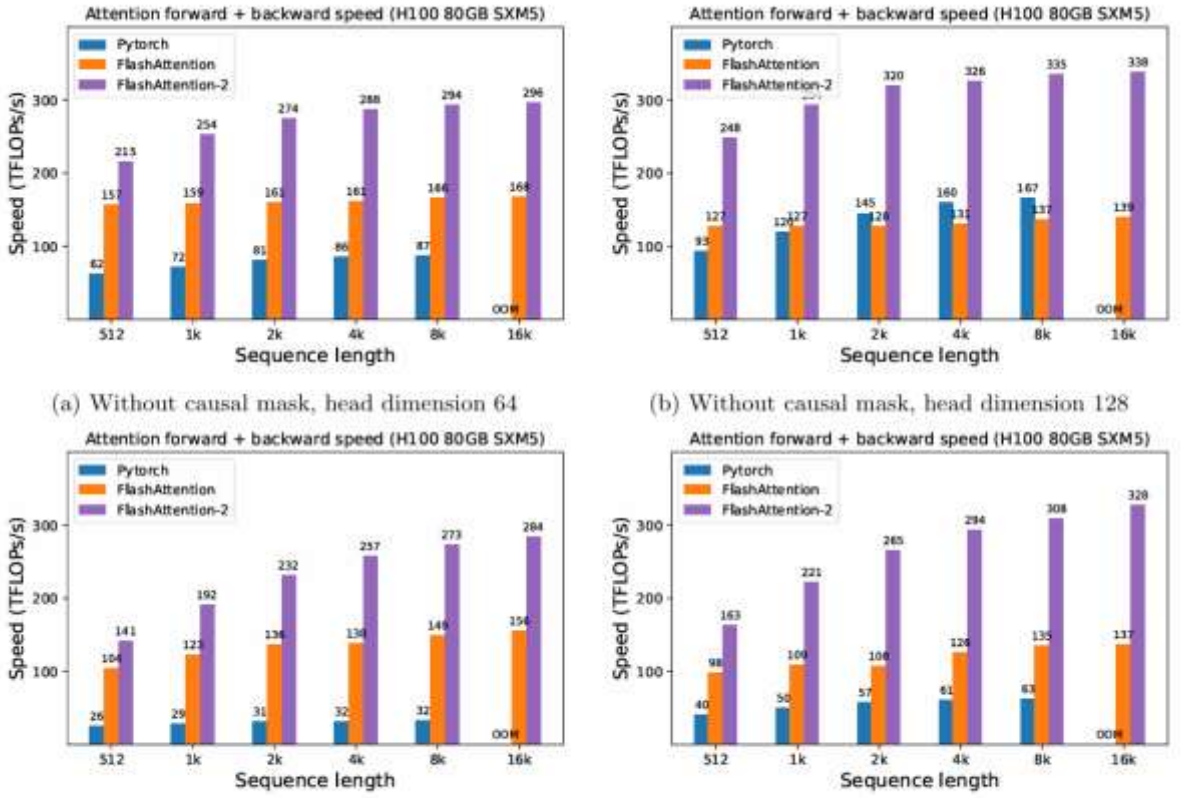


Memory Hierarchy with Bandwidth & Memory Size

FlashAttention v1 的问题

Flash Attention仍然不如其他基本操作(比如矩阵乘法)高效虽然Flash Attention已经比标准的注意力实现快2-4倍，但前向传播仅达到设备理论最大 FLOPs/s 的 30-50%，而反向传播更具挑战性，仅达到 A100 GPU 最大吞吐量的 25-35%

相比之下，优化的矩阵乘法可以达到理论最大设备吞吐量的 80-90% 。 通过仔细的分析，观察到Flash Attention在GPU上不同线程块和线程束之间的工作划分仍然不够优化，导致低占用率或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

Figure 7: Attention forward + backward speed on H100 GPU

论文贡献 FlashAttentionv2

减少非矩阵乘法的 FLOPs
提升 GPU 计算单元的利用率
优化线程块划分，提升并行度，减少线程间通信与共享内存读写

减少非矩阵乘法的 FLOPs

虽然这些非矩阵操作（如 softmax、dropout、归一化等）在总计算量中所占比例不高，但它们在 GPU 上的执行速度却远远慢于矩阵乘法操作。这是因为现代 GPU（如 Nvidia A100）配备了专门用于加速矩阵乘法的 Tensor Cores，可以在 FP16 或 BF16 精度下实现高达数百 TFLOPs/s 的吞吐率。FlashAttention-2 尽可能地将原本需要用标量操作完成的逻辑转化为张量级的矩阵乘法，将大量计算压力集中在 GPU 的强项——张量核心上，从而大幅提升整体执行效率。

提升 GPU 计算单元的利用率

算法对前向传播和反向传播的步骤进行了重构和精简，减少了那些无法触发张量核心的计算步骤，同时延迟某些归一化或缩放操作的执行，使得它们可以在矩阵乘法之后一次性完成，避免了中间频繁的内存访问和低效的标量操作。通过这种方式，FlashAttention-2 的前向传播性能在实测中可达到 A100 理论最大吞吐率的 73%，反向传播达到 63%，远高于 FlashAttention V1 的 25%-50%。

优化线程块划分，提升并行度，减少线程间通信与共享内存读写

在 FlashAttention V1 中，每个线程块通常只处理一个 attention head，而当 batch size 或 head 数较小时，这种划分方式会导致线程块总数不足，从而无法填满 GPU 的所有计算单元。FlashAttention-2 在此基础上引入了 跨序列维度的并行化，使得即使在长序列、小 batch 的场景下，也能维持高线程活跃度。

Metareview: 元评论:

This paper proposes an improved version of FlashAttention which is more efficient than the original one. Most reviewers agree that this is a strong paper, emphasizing clarity, engineering work, empirical validation, and demonstrated practical usefulness. Even though the novelty beyond the original FlashAttention is somewhat limited, the proposed method is already being adopted by the community and seems to be having real impact. I recommend acceptance.

本文提出了一个比原版 FlashAttention 更高效的改进版本。大多数评论者认为这是一篇优秀的论文，强调了其清晰性、工程化工作、实证验证以及实际应用价值。尽管其在原版 FlashAttention 基础上的创新性有限，但所提出的方法已被社区采用，并似乎产生了实际影响。我建议接收。

Justification For Why Not Higher Score:

为何不提高分数的理由:

Even though the proposed FlashAttention2 is already being impactful, there is not enough scientific novelty in this paper to justify (in my opinion) being highlighted as a spotlight paper.

尽管所提出的 FlashAttention2 已经产生了影响，但这篇论文的科学新颖性不足以（在我看来）成为焦点论文。

论文算法与创新点

向前传播；向后传播；并行化

原始attention的问题

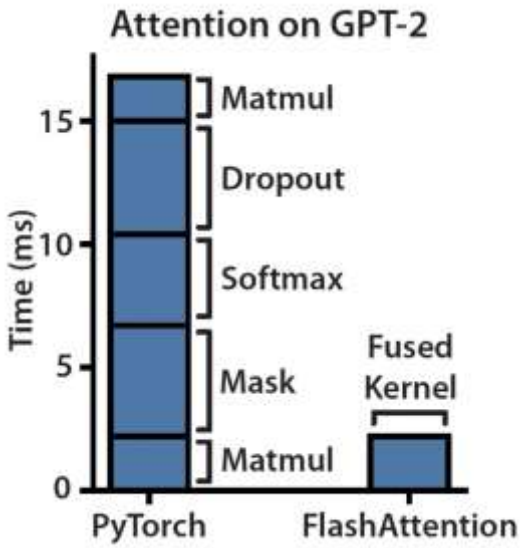
该算法的主要瓶颈在于：每次计算都需要将数据从 GPU 的高带宽内存（HBM）加载到片上高速缓存（如寄存器或 SRAM），计算完成后又需将结果写回 HBM。这种频繁的数据传输类似于 CPU 中寄存器与主内存之间的关系，极大限制了计算效率。因此，一个显而易见且有效的优化方向是：尽可能减少这种来回的数据移动。这正是所谓的“Kernel Fusion（核函数融合）”

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 $\mathbf{O}$.

从左侧性能分析栏可以观察到，尽管大部分的 FLOPs（每秒浮点运算次数） 都集中在矩阵乘法操作上，但实际消耗时间最多的却是 masking、softmax 和 dropout 等操作。这表明：虽然这些操作计算量较小，但由于其 IO 强度高或缺乏并行性，反而成为了性能瓶颈。



优化向前传播

FlashAttention V2 的聪明做法是：先不急着做归一化，而是先把所有注意力块都算完，再统一做一次归一化处理。这样可以少做很多重复的运算，节省计算时间。

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

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

第一行表示对第一个注意力块中的每一行取最大值，这一步是为了做 softmax 时增强数值稳定性，避免指数运算溢出。

第二行是对每一行的注意力分数减去最大值后再做指数运算，并将结果沿行求和，得到 softmax 的归一化因子。

第三行是将第一个块的未归一化的注意力权重乘以对应的值向量，得到第一个块的注意力输出，但此时还未除以归一化因子。

第四行的意思是我们要更新全局的最大值，比较第一个块的最大值和第二个块的行最大值，取更大的那个用于后续统一的 softmax 缩放。

第五行是将第一个块的归一化因子按比例缩放后与第二个块的新归一化因子相加，得到整体的 softmax 分母，也就是跨两个块的总归一化因子。

最后一行的表示将第一个块的输出按照新的最大值进行缩放，然后加上第二个块的输出，这样两个块的结果就被正确合并，并且可以得到最终的注意力输出。

FlashAttention V2 的做法是：

重计算 S 替代保存
只保存 m、l、O
完全片上计算
支持 dropout/mask
列方向并行

Algorithm 4 FlashAttention Backward Pass

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N \times d}$ in HBM, vectors $\ell, m \in \mathbb{R}^N$ in HBM, on-chip SRAM of size M, softmax scaling constant $\tau \in \mathbb{R}$, masking function MASK, dropout probability p_{drop}, pseudo-random number generator state $\mathcal{R}$ from the forward pass.

1. Set the pseudo-random number generator state to $\mathcal{R}$.
2. Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
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 $\mathbf{O}_1, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\mathbf{dO}$ into T_r blocks $\mathbf{dO}_1, \dots, \mathbf{dO}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_1, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \dots, m_{T_r}$ of size B_r each.
5. Initialize $\mathbf{dQ} = (0)_{N \times d}$ in HBM and divide it into T_r blocks $\mathbf{dQ}_1, \dots, \mathbf{dQ}_{T_r}$ of size $B_r \times d$ each. Initialize $\mathbf{dK} = (0)_{N \times d}, \mathbf{dV} = (0)_{N \times d}$ in HBM and divide $\mathbf{dK}, \mathbf{dV}$ in to T_c blocks $\mathbf{dK}_1, \dots, \mathbf{dK}_{T_c}$ and $\mathbf{dV}_1, \dots, \mathbf{dV}_{T_c}$, of size $B_c \times d$ each.
6. **for** $1 \le j \le T_c$ **do**
7. Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
8. Initialize $\mathbf{dK}_j = (0)_{B_c \times d}, \mathbf{dV}_j = (0)_{B_c \times d}$ on SRAM.
9. **for** $1 \le i \le T_r$ **do**
10. Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
11. On chip, compute $\mathbf{S}_{ij} = \tau \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
12. On chip, compute $\mathbf{S}_{ij}^{\text{masked}} = \text{MASK}(\mathbf{S}_{ij})$.
13. On chip, compute $\mathbf{P}_{ij} = \text{diag}(\ell_i)^{-1} \exp(\mathbf{S}_{ij}^{\text{masked}} - m_i) \in \mathbb{R}^{B_r \times B_c}$.
14. On chip, compute dropout mask $\mathbf{Z}_{ij} \in \mathbb{R}^{B_r \times B_c}$ where each entry has value $\frac{1}{1-p_{\text{drop}}}$ with probability $1 - p_{\text{drop}}$ and value 0 with probability p_{drop}.
15. On chip, compute $\mathbf{P}_{ij}^{\text{dropped}} = \mathbf{P}_{ij} \circ \mathbf{Z}_{ij}$ (pointwise multiply).
16. On chip, compute $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_{ij}^{\text{dropped}})^T \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$.
17. On chip, compute $\mathbf{dP}_{ij}^{\text{dropped}} = \mathbf{dO}_i \mathbf{V}_j^T \in \mathbb{R}^{B_r \times B_c}$.
18. On chip, compute $\mathbf{dP}_{ij} = \mathbf{dP}_{ij}^{\text{dropped}} \circ \mathbf{Z}_{ij}$ (pointwise multiply).
19. On chip, compute $D_i = \text{rowsum}(\mathbf{dO}_i \circ \mathbf{O}_i) \in \mathbb{R}^{B_r}$.
20. On chip, compute $\mathbf{dS}_{ij} = \mathbf{P}_{ij} \circ (\mathbf{dP}_{ij} - D_i) \in \mathbb{R}^{B_r \times B_c}$.
21. Write $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \tau \mathbf{dS}_{ij} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$ to HBM.
22. On chip, compute $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + \tau \mathbf{dS}_{ij}^T \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$.
23. **end for**
24. Write $\mathbf{dK}_j \leftarrow \mathbf{dK}_j, \mathbf{dV}_j \leftarrow \mathbf{dV}_j$ to HBM.
25. **end for**
26. Return $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.

算法2FlashAttention-2反向传播

要求：矩阵 $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N \times d}$在HBM中，向量 $L \in \mathbb{R}^N$在HBM中，块大小 B_c、B_r。

1: 将 Q分成 T_r个块 $=\left\lceil \frac{N}{B_r} \right\rceil$ 块 $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$，大小为 $B_r \times d$的每个块，并将 K，V分成 T_c个块$\left\lceil \frac{N}{B_c} \right\rceil$ 块 $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ 和 $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$，大小为 $B_c \times d$的每个块。

2: 将 O分成 T_r个块 $\mathbf{O}_i, \dots, \mathbf{O}_T$每个大小为 $B_r \times d$的块，将 dO分成 T_r个块 $\mathbf{dO}_1, \dots$，dO大小为 $B_r \times d$ 的每个块，并将 L分成 T_r个块 $L_i, \dots, L_T$大小为 B_r的每个块。

3: 初始化 $\mathbf{dQ} = (0)_{N \times d}$在HBM中，并将其分成 T_r块 $\mathbf{dQ}_1, \dots, \mathbf{dQ}_T$大小为 $B_r \times d$的每个。分割 $\mathbf{dK}, \mathbf{dV} \in \mathbb{R}^{N \times d}$ 分成 T_c 块 $\mathbf{dK}_1, \dots, \mathbf{dK}_{T_c}$ 和 $\mathbf{dV}_1, \dots, \mathbf{dV}_{T_c}$，大小为 $B_c \times d$的每个块。

4: 计算 $D = \text{rowsum}(\mathbf{dO} \circ \mathbf{O}) \in \mathbb{R}^d$(逐点相乘)，将 D写入HBM并将其分成 T_r块 $D_1, \dots, D_{T_r}$ 大小为 B_r的每个块。

5: 对于 $1 \le j \le T_c$执行

6: 从HBM加载 $\mathbf{K}_j, \mathbf{V}_j$到片上SRAM。

7: 初始化 $\mathbf{dK}_j = (0)_{B_c \times d}, \mathbf{dV}_j = (0)_{B_c \times d}$ 在SRAM上。

8: 对于 $1 \le i \le T_r$执行

9: 从HBM加载 $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, L_i, D_i$到片上SRAM。

10: 在芯片上，计算 $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}^{T}{}_j \in \mathbb{R}^{B_r \times B_c}$。

11: 在片上计算 $\mathbf{P}_i^{(j)} = \exp(\mathbf{S}_{ij} - L_i) \in \mathbb{R}^{B_r \times B_c}$。

12: 在芯片上，计算 $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_i^{(j)})^T \mathbf{dO}_i \in \mathbb{R}^{B_c \times d}$。

13: 在芯片上，计算 $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \mathbf{V}_j^T \in \mathbb{R}^{B_r \times B_c}$。

14: 在芯片上，计算 $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ (\mathbf{dP}_i^{(j)} - D_i) \in \mathbb{R}^{B_r \times B_c}$。

15: 从HBM加载$\mathbf{dQ}_i$到SRAM，然后在芯片上，更新$\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \mathbf{dS}_i^{(j)} \mathbf{K}_j \in \mathbb{R}^{B_r \times d}$，并写回HBM。

16: 在芯片上，计算$\mathbf{dK}_j \leftarrow \mathbf{dK}_j + \mathbf{dS}^{(j)}{}_i{}^T \mathbf{Q}_i \in \mathbb{R}^{B_c \times d}$。

17: 结束循环

18: 将$\mathbf{dK}_j$和$\mathbf{dV}_j$写回HBM。

19: 结束循环

20: 返回dQ，dK，dV。

优化反向传播

重计算 S 替代保存：

在传统注意力机制中，为了反向传播，需要保存正向计算得到的注意力分数矩阵 S 保存它将严重占用显存。FlashAttention V2 的做法是：不保存 S，而是在反向传播阶段重新计算 S。

只保存 m、l、O：

m：每一行的最大值（为了数值稳定）

ℓ：softmax 分母，即 $\sum \exp(S-m)$

O：最终注意力输出

完全片上计算：

所有关键计算步骤都在 GPU 的片上 SRAM中完成

支持 dropout/mask：

将 Dropout 和 Mask 的应用嵌入到 softmax 权重的计算流程中，而且在反向传播中也能高效支持

Dropout 的 mask 是在正向生成时记录的随机种子，在反向中用相同规则再采样

Mask 的处理通过对 S 的掩码操作和 softmax 归一化阶段融合完成

算法2FlashAttention-2反向传播

要求：矩阵 $\mathbf{Q},\mathbf{K},\mathbf{V},\mathbf{O},\mathbf{dO} \in \mathbb{R}^{N\times d}$在HBM中，向量 $L \in \mathbb{R}^{N}$在HBM中，块大小 B_c、B_r。

1: 将 Q分成 T_r个块 $=\left\lceil \frac{N}{B_r} \right\rceil$ 块 $\mathbf{Q}_1,\ldots,\mathbf{Q}_{T_r}$ 大小为 $B_r \times d$的每个块，并将 K，V分成 T_c个块$\left\lceil \frac{N}{B_r} \right\rceil$ 块 $\mathbf{K}_1,\ldots,\mathbf{K}_{T_c}$ 和 $\mathbf{V}_1,\ldots,\mathbf{V}_{T_c}$，大小为 $B_c \times d$的每个块。

2: 将 O分成 T_r个块 $\mathbf{O}_i,\ldots,\ldots,\ldots,\mathbf{O}_T$每个大小为 $B_r \times d$的块，将 dO分成 T_r个块 $\mathbf{dO}_1,\ldots,\mathbf{dO}_T$小为 $B_r \times d$ 的每个块，并将 L分成 T_r个块 $L_1,\ldots,L_T$大小为 B_r的每个块 .

3: 初始化 $\mathbf{dQ}=(0)_{N\times d}$在HBM中，并将其分成 T_r块 $\mathbf{dQ}_1,\ldots,\ldots,\ldots,\mathbf{dQ}_T$小为 $B_r \times d$的每个。分割 $\mathbf{dK},\mathbf{dV} \in \mathbb{R}^{N\times d}$ 分成 T_c 块 $\mathbf{dK}_1,\ldots,\mathbf{dK}_{T_c}$ 和 $\mathbf{dV}_1,\ldots,\mathbf{dV}_{T_c}$，大小为 $B_c \times d$的每个块。

4: 计算 $D=\mathrm{rowsum}(\mathbf{dO}\circ\mathbf{O}) \in \mathbb{R}^{d}$(逐点相乘)，将 D写入HBM并将其分成 T_r块 $D_1,\ldots,D_{T_r}$ 大小为 B_r的每个块 .

5: 对于 $1 \le j \le T_c$执行

6: 从HBM加载 $\mathbf{K}_j,\mathbf{V}_j$到片上SRAM。

7: 初始化 $\mathbf{dK}_j=(0)_{B_c\times d},\mathbf{dV}_j=(0)_{B_c\times d}$ 在SRAM上。

8: 对于 $1 \le i \le T_r$执行

9: 从HBM加载 $\mathbf{Q}_i,\mathbf{O}_i,\mathbf{dO}_i,\mathbf{dQ}_i,L_i,D_i$到片上SRAM。

10: 在芯片上，计算 $\mathbf{S}_i^{(j)}=\mathbf{Q}_i\mathbf{K}_j^T \in \mathbb{R}^{B_r\times B_c}$。

11: 在片上计算 $\mathbf{P}_i^{(j)}=\exp(\mathbf{S}_{ij}-L_i) \in \mathbb{R}^{B_r\times B_c}$。

12: 在芯片上，计算 $\mathbf{dV}_j \leftarrow \mathbf{dV}_j+(\mathbf{P}_i^{(j)})^T\mathbf{dO}_i \in \mathbb{R}^{B_c\times d}$。

13: 在芯片上，计算 $\mathbf{dP}_i^{(j)}=\mathbf{dO}_i\mathbf{V}_j^T \in \mathbb{R}^{B_r\times B_c}$。

14: 在芯片上，计算 $\mathbf{dS}_i^{(j)}=\mathbf{P}_i^{(j)}\circ(\mathbf{dP}_i^{(j)}-D_i) \in \mathbb{R}^{B_r\times B_c}$。

15: 从HBM加载$\mathbf{dQ}_i$到SRAM，然后在芯片上，更新$\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i+\mathbf{dS}_i^{(j)}\mathbf{K}_j \in \mathbb{R}^{B_r\times d}$，并写回HBM。

16: 在芯片上，计算$\mathbf{dK}_j \leftarrow \mathbf{dK}_j+\mathbf{dS}^{(j)}{}^T\mathbf{Q}_i \in \mathbb{R}^{B_c\times d}$。

17: 结束循环

18: 将$\mathbf{dK}_j$和$\mathbf{dV}_j$写回HBM。

19: 结束循环

20: 返回dQ，dK，dV。

lashAttention-2 采用了不同的并行化策略，其核心在于将并行方向从处理 K/V 的列切换到了处理 Q 的行。把 Q 分成多个块，每个 warp 只负责处理其中一部分 Q 的行，也就是每个 warp 独立负责一个或一组 query token 的 attention 输出计算。

在这种结构下，K^T 和 V 被所有 warps 共享访问，但因为是只读操作，并不会引起冲突或同步问题。注意力输出之间没有依赖关系，因此计算上是完全并行的。这一策略带来的好处非常明显。首先，每个 warp 计算的是不同 query 的 attention，不存在写冲突，也不需要 warp 间通信，因此并行度大大提高

FlashAttention-2 的并行化本质上是将计算重心从“所有 warp 共同处理一个 query 与多个 key 的匹配”转变为“每个 warp 独立处理一个 query 与所有 key 的匹配”，这种转变带来的结构性优化，使得 attention 模块在现代 GPU 上能够更好地释放硬件潜力，成为高效训练 Transformer 的关键技术之一。

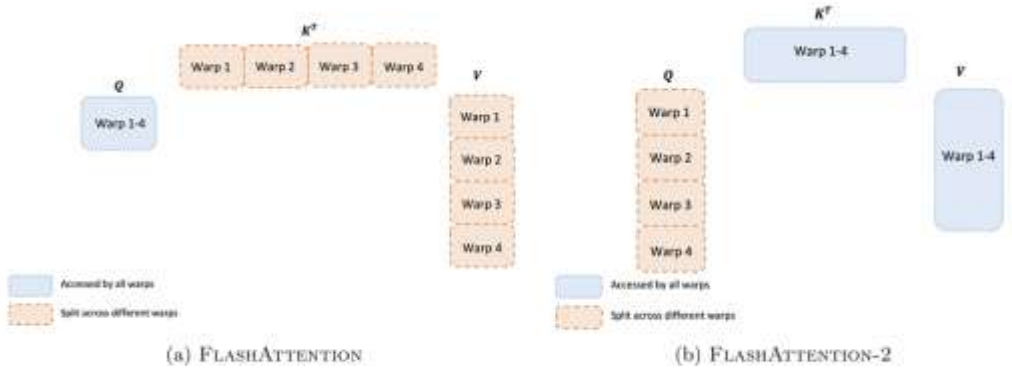


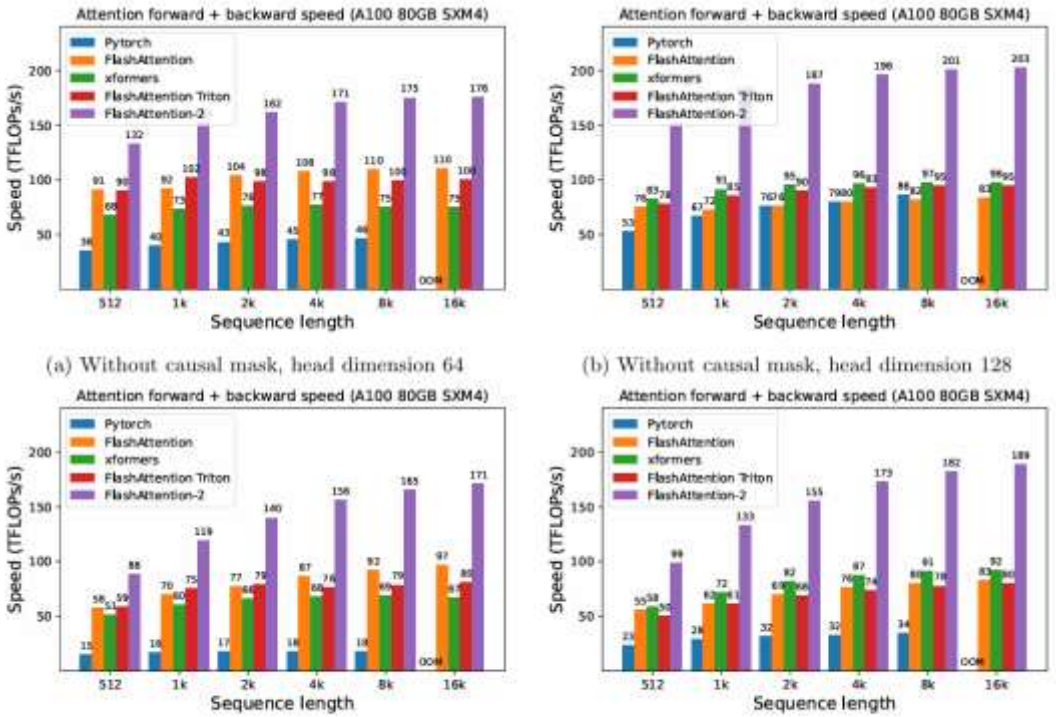
Figure 3: Work partitioning between different warps in the forward pass

论文实验

比 FlashAttention（v1）快约 2 倍
比 FlashAttention in Triton 快约 1.3-1.5 倍（前向），2 倍（反向）
比 PyTorch 原生实现最高快达 10 倍
在 16K 长序列时依然保持稳定运行，其他方法常常 OOM

实验一（注意力模块基准测试）

图 4 展示了在 NVIDIA A100 80GB SXM4 GPU 上，不同注意力实现的前向 + 反向传播速度，针对不同的设置（是否使用 causal mask、不同的 head dimension）和不同序列长度（从 512 到 16K）进行全面对比。
图 5 展示了在同样的设备上，不同注意力实现方法在仅前向传播（forward pass）阶段的性能表现。图中的对比方法包括 PyTorch 原生实现、FlashAttention v1、xFormers、FlashAttention Triton、FlashAttention-2v



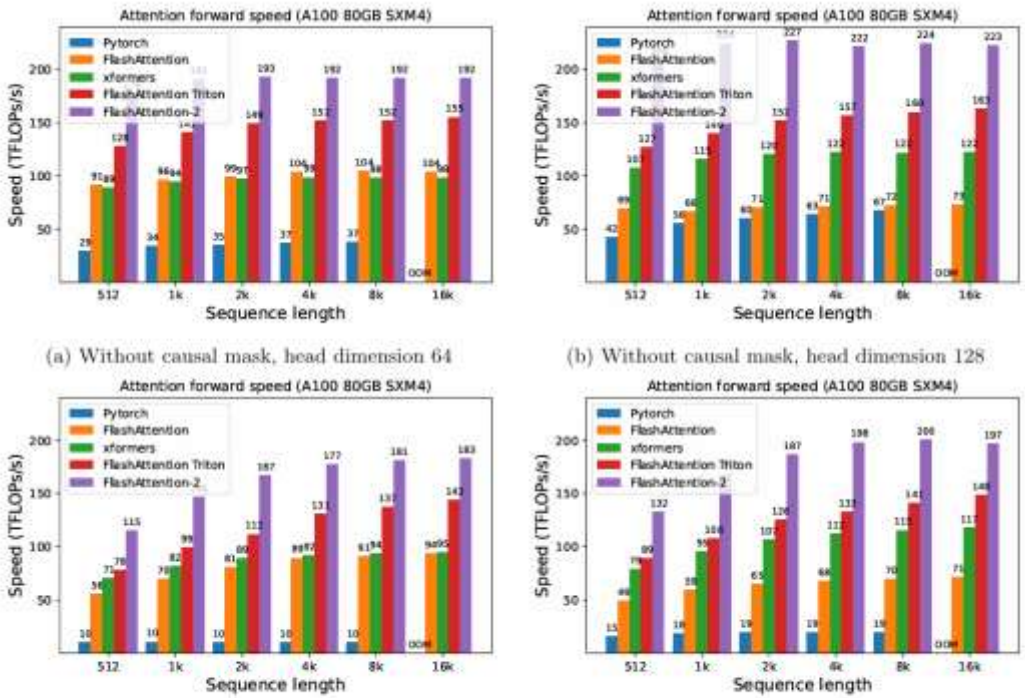
(a) Without causal mask, head dimension 64

(b) Without causal mask, head dimension 128

(c) With causal mask, head dimension 64

(d) With causal mask, head dimension 128

Figure 4: Attention forward + backward speed on A100 GPU



(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

Figure 5: Attention forward speed on A100 GPU

FlashAttention-2 显著提升了 GPT 模型的训练速度，尤其在长上下文（8K）设置下，相比传统实现可带来高达 3× 的加速，同时达到 A100 GPU 72% 的 FLOPs 利用率，几乎接近硬件性能上限，是当前最优的注意力计算方案之一。

Table 1: Training speed (TFLOPs/s/GPU) of GPT-style models on 8×A100 GPUs. FLASHATTENTION-2 reaches up to 225 TFLOPs/s (72% model FLOPs utilization). We compare against a baseline running without FLASHATTENTION.

Model	Without FLASHATTENTION	FLASHATTENTION	FLASHATTENTION-2
GPT3-1.3B 2k context	142 TFLOPs/s	189 TFLOPs/s	196 TFLOPs/s
GPT3-1.3B 8k context	72 TFLOPS/s	170 TFLOPs/s	220 TFLOPs/s
GPT3-2.7B 2k context	149 TFLOPs/s	189 TFLOPs/s	205 TFLOPs/s
GPT3-2.7B 8k context	80 TFLOPs/s	175 TFLOPs/s	225 TFLOPs/s