并行算法-Attention算子实现

说明：本文档部分图片来自知乎文章https://zhuanlan.zhihu.com/p/668888063

背景

Attention算子是Transformer模型的核心组件之一，主要用于处理序列数据。它通过计算输入序列中各个位置之间的相关性来生成输出序列。 其集体的计算公式是

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

Q,K,V是x输入经过三个线性层得到的查询（Query）、键（Key）和值（Value）矩阵，d_k是键的维度。

不同于以往的并行算法，Attention算子目前不仅受制于运算速度，还受制于内存带宽，以及空间复杂度，算力对高速显存的依赖需要我们改进算法，在计算速度和占用内存之间取得平衡。

加速算法-FlashAttention

FlashAttention是一种高效的Attention计算方法，主要通过以下方式加速：

1. online-softmax
2. 分块访存利用内存层次

online-softmax的历史演变

softmax的伪代码如下：

Algorithm 1 Naive softmax

1: $d_0 \leftarrow 0$
2: **for** $j \leftarrow 1, V$ **do**
3: $\quad d_j \leftarrow d_{j-1} + e^{x_j}$
4: **end for**
5: **for** $i \leftarrow 1, V$ **do**
6: $\quad y_i \leftarrow \frac{e^{x_i}}{d_V}$
7: **end for**

1. 从访存的角度考虑，原始的softmax对每个元素需要2次load,1次store操作；
2. 从数值稳定性上考虑，原始的naive softmax在计算过程中可能会出现数值溢出的问题，因此需要对其进行改进。

safe-softmax
safe-softmax通过减去输入向量的最大值来避免这种问题. safe-softmax的伪代码如下：

Algorithm 2 Safe softmax

$m_0 \leftarrow -\infty$
for $k \leftarrow 1, V$ **do**
  $m_k \leftarrow \max(m_{k-1}, x_k)$
end for
$d_0 \leftarrow 0$
for $j \leftarrow 1, V$ **do**
  $d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
end for
for $i \leftarrow 1, V$ **do**
  $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
end for

这次改进由于需要**计算max**,所以每个元素需要3次load,1次store操作。

onlinr-softmax最早由nvidia在2018年提出，主要利用了softmax中指数运算法则的特性，将softmax的前后依赖关系打破，允许在计算softmax的过程中进行并行计算。伪代码如下图：

Algorithm 3 Safe softmax with online normalizer calculation

$m_0 \leftarrow -\infty$
$d_0 \leftarrow 0$
for $j \leftarrow 1, V$ **do**
  $m_j \leftarrow \max(m_{j-1}, x_j)$
  $d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j}$
end for
for $i \leftarrow 1, V$ **do**
  $y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
end for

分块访存前向计算

GPU的访存根据访问速度从高速到低速有层次之分，从register, L1-cache/shared memory, L2-cache, L3-cache, HBM等。为了充分利用GPU的高速缓存，FlashAttention采用了分块访存的方式进行前向计算。

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 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.

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

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

  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. 再load Q

    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 $\mathbf{O}$.

通过分块，可以将Q,K,V矩阵分成多个小块，每次只计算一个小块的Attention，具体是将小块载入共享内存，然后在共享内存中进行计算。这样可以减少对HBM的访问次数，提高计算速度。

反向计算优化

从访存上分析，反向计算根本不需要从HBM获取中间变量，而是直接利用sram的分块Q,K进行recompute。因为工程上速度慢一点没有很大影响，但是recompute可以大大节省显存开销，所以可以设计更大的神经网络，最终的网络效果会更好。

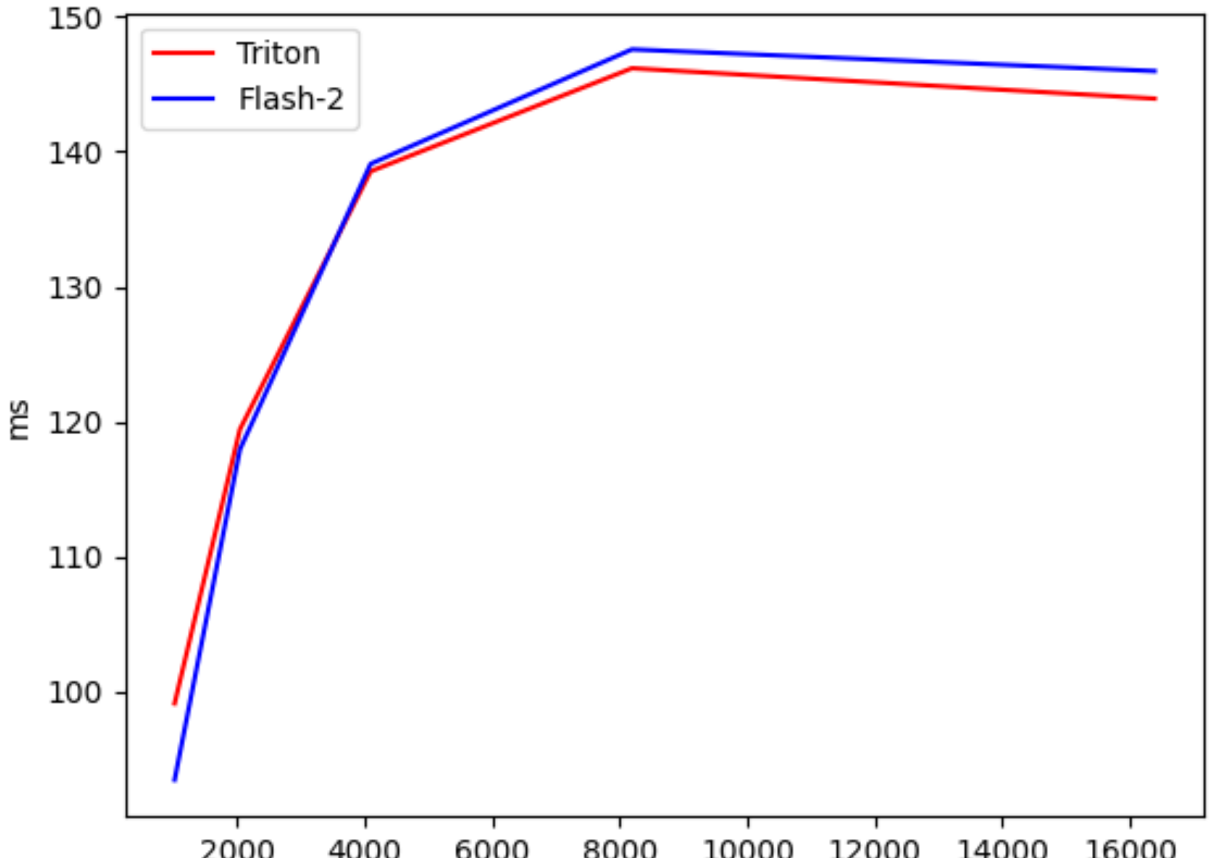
算法实现

使用RTX4090-24GB显卡作为实验平台，使用Triton编译器，以python语言实现算子，Jit动态编译将triton算子转化为cuda代码。Triton编译器使得python开发高性能算子成为可能，只需要对分块进行说明，就可以执行高效的分块并行计算。虽然在性能上不一定比得上C++实现，但在开发效率上有很大提升。

```
# 前向传播内核：FlashAttention 前向
@triton.jit
def _fwd_kernel(
    Q, K, V, sm_scale,              # 输入：q, k, v 和 softmax 缩放因子
    L, Out,                         # 输出：行和 log(sum(exp))，以及最终输出
    stride_qz, stride_qh, stride_qm, stride_qk,
    stride_kz, stride_kh, stride_kn, stride_kk,
    stride_vz, stride_vh, stride_vk, stride_vn,
    stride_oz, stride_oh, stride_om, stride_on,
    Z, H, N_CTX,                    # 批量、头数、序列长度
    BLOCK_M: tl.constexpr, BLOCK_DMODEL: tl.constexpr,
    BLOCK_N: tl.constexpr,
    IS_CAUSAL: tl.constexpr,
):
```

实验测试

首先是以4, 48, 4096, 32作为测试： 可以发现和cuda实现性能上没有什么区别，稍微落后于高性能的Flash-2 cuda实现。



backward的实现和flash-2的cuda实现更相近，在部分测试中甚至超过了Flash-2的cuda实现。