复旦大学 25300180035 池裕涵

代码仓库：IO-AwareAttention

1. 项目概览

lec5 介绍了 PyTorch。lec6 讲了 GPU 内存层级（HBM vs SRAM）。lec7 讲了 Triton——用 Python 写 tile 级别的 GPU kernel。lec8 讲算子融合，把多个 kernel 压成一个。

这几节课串起来就是：怎么从软件层把程序加速到极致。于是我选了 FlashAttention 作为课程作业。

FlashAttention 是一个 IO-aware 的精确注意力算法，核心想法是把大的 N×N 注意力矩阵切块，在 GPU 的高速 SRAM 里逐块计算，而不是把中间结果写回慢速 HBM。这样它既保持了标准 attention 的数学精度，又把显存从 $O(N^2)$ 降到 $O(N)$，还跑得更快。对长序列来说，这是绕不开的基础设施——没有它，我们现在身边的大模型根本跑不动。

这个项目里，我从最朴素的 Pure Python 开始，依次用 C++、PyTorch 实现了标准 Attention 计算，最后用 Triton 写了 FlashAttention。实验结果很直观：光是把中间矩阵从 HBM 搬到 SRAM 里算，就能把显存和速度都拉开一个数量级的差距。IO 瓶颈比想象中严重得多。

我做了两个版本： - **Lite**：局部 Softmax，代码简单但 $N>64$ 时不准 - **Online**：加了 running max/sum，全局精确，只比 Lite 多 10 行

2. 文件说明

文件名字	文件内容
`flashattention_lite.py`	Triton kernel，局部 Softmax 版
`flashattention_online.py`	Triton kernel，Online Softmax 版
`cpp_attention.cpp`	C++ 版 Attention benchmark（Value* 指针风格）
`benchmark.py`	性能对比脚本,内含Pure Python 和 PyTorch Attention

3. 各实现版本的 Attention

3.1 Pure Python

```
attn_logits = [sum(q_h[j] * k_h[t][j] for j in range(head_dim))
               / head_dim**0.5 for t in range(len(k_h))]
```

```
attn_weights = softmax(attn_logits)
head_out = [sum(attn_weights[t] * v_h[t][j] for t in range(len(v_h)))
            for j in range(head_dim)]
```

每个 `*`、`+` 都要创建 Value 对象记录计算图，慢在对象创建而不是运算本身。

3.2 C++

```
// Value*
double scale_factor = 1.0 / sqrt((double)head_dim);
Value* scale_node = Value::make_new(scale_factor);
for(size_t t = 0; t < seq_len; ++t) {
    Value* dot = Value::make_new(0.0);
    for(size_t i = 0; i < q.data.size(); ++i)
        dot = Value::add(dot, Value::mul(q.data[i], keys[t].data[i]));
    scores.data[t] = Value::mul(dot, scale_node);
}
Vector weights = softmax(scores);
Vector output; output.data.resize(dim_embd, nullptr);
for(size_t i = 0; i < dim_embd; ++i) output.data[i] = Value::make_new(0.0);
for(size_t t = 0; t < seq_len; ++t) {
    Value* w = weights.data[t];
    for(size_t i = 0; i < dim_embd; ++i)
        output.data[i] = Value::add(output.data[i], Value::mul(w,
values[t].data[i]));
}
```

C++ 版，`Value*` 指针操作，编译器能做内联、循环展开。不过 CPU 单线程，N 大了还是慢。

3.3 PyTorch

```
wei = q @ k.transpose(-2, -1) * (head_dim ** -0.5)
wei = F.softmax(wei, dim=-1)
out = wei @ v
```

3 行搞定，cuBLAS 高度优化。但 `wei` 这个 $N \times N$ 矩阵必须写 HBM，这就是问题。

3.4 FlashAttention-Lite（flashattention_lite.py）

```
pid_m = tl.program_id(0)
qm_start = pid_m * BLOCK_M
offs_m = qm_start + tl.arange(0, BLOCK_M)
offs_d = tl.arange(0, BLOCK_D)
q_ptrs = Q_ptr + offs_m[:, None] * stride_qn + offs_d[None, :] * stride_qd
q_mask = (offs_m[:, None] < N) & (offs_d[None, :] < d)
Q_tile = tl.load(q_ptrs, mask=q_mask, other=0.0)
```

```
acc = tl.zeros([BLOCK_M, BLOCK_D], dtype=tl.float32)
num_k_blocks = tl.cdiv(N, BLOCK_N)
for j in range(num_k_blocks):
    kn_start = j * BLOCK_N
    offs_n = kn_start + tl.arange(0, BLOCK_N)
    k_ptrs = K_ptr + offs_n[:, None] * stride_kn + offs_d[None, :] * stride_kd
    k_mask = (offs_n[:, None] < N) & (offs_d[None, :] < d)
    K_tile = tl.load(k_ptrs, mask=k_mask, other=0.0)
    v_ptrs = V_ptr + offs_n[:, None] * stride_vn + offs_d[None, :] * stride_vd
    v_mask = (offs_n[:, None] < N) & (offs_d[None, :] < d)
    V_tile = tl.load(v_ptrs, mask=v_mask, other=0.0)
    S_ij = tl.dot(Q_tile, tl.trans(K_tile)) * scale
    S_ij = tl.where(offs_n[None, :] < N, S_ij, float('-inf'))
    m_ij = tl.max(S_ij, axis=1)[:, None]
    P_ij = tl.exp(S_ij - m_ij)
    l_ij = tl.sum(P_ij, axis=1)[:, None]
    P_ij = P_ij / l_ij
    acc += tl.dot(P_ij.to(V_tile.dtype), V_tile)
o_ptrs = O_ptr + offs_m[:, None] * stride_on + offs_d[None, :] * stride_od
o_mask = (offs_m[:, None] < N) & (offs_d[None, :] < d)
tl.store(o_ptrs, acc.to(O_ptr.dtype.element_ty), mask=o_mask)
```

和前面 3 个的区别：S 和 P 只存在于 SRAM，循环结束才写一次 HBM。 但是问题也会有，softmax是要全局的max，每个tile内部算得再快，结果有偏也是完蛋。

3.5 FlashAttention-Online（flashattention_online.py）

和 Lite 版的唯一区别在 Softmax 处理，用 Δ 标注：

```
pid_m = tl.program_id(0)
qm_start = pid_m * BLOCK_M
offs_m = qm_start + tl.arange(0, BLOCK_M)
offs_d = tl.arange(0, BLOCK_D)
q_ptrs = Q_ptr + offs_m[:, None] * stride_qn + offs_d[None, :] * stride_qd
q_mask = (offs_m[:, None] < N) & (offs_d[None, :] < d)
Q_tile = tl.load(q_ptrs, mask=q_mask, other=0.0)
acc = tl.zeros([BLOCK_M, BLOCK_D], dtype=tl.float32)
m_i = tl.full([BLOCK_M, 1], float('-inf'), dtype=tl.float32)  # Δ running max
l_i = tl.zeros([BLOCK_M, 1], dtype=tl.float32)                 # Δ running sum
num_k_blocks = tl.cdiv(N, BLOCK_N)
for j in range(num_k_blocks):
    kn_start = j * BLOCK_N
    offs_n = kn_start + tl.arange(0, BLOCK_N)
    k_ptrs = K_ptr + offs_n[:, None] * stride_kn + offs_d[None, :] * stride_kd
    k_mask = (offs_n[:, None] < N) & (offs_d[None, :] < d)
    K_tile = tl.load(k_ptrs, mask=k_mask, other=0.0)
    v_ptrs = V_ptr + offs_n[:, None] * stride_vn + offs_d[None, :] * stride_vd
    v_mask = (offs_n[:, None] < N) & (offs_d[None, :] < d)
    V_tile = tl.load(v_ptrs, mask=v_mask, other=0.0)
    S_ij = tl.dot(Q_tile, tl.trans(K_tile)) * scale
    S_ij = tl.where(offs_n[None, :] < N, S_ij, float('-inf'))
    m_ij = tl.max(S_ij, axis=1)[:, None]
```

```
    m_new = tl.maximum(m_i, m_ij)                                    # Δ 全局 max
    correction = tl.exp(m_i - m_new)                                  # Δ 修正因子
    P_ij = tl.exp(S_ij - m_new)
    l_i = l_i * correction + tl.sum(P_ij, axis=1)[:, None]       # Δ 全局 sum
    acc = acc * correction + tl.dot(P_ij.to(V_tile.dtype), V_tile)# Δ 修正累加器
    m_i = m_new
acc = acc / l_i                                                       # Δ 最终归一化
o_ptrs = O_ptr + offs_m[:, None] * stride_on + offs_d[None, :] * stride_od
o_mask = (offs_m[:, None] < N) & (offs_d[None, :] < d)
tl.store(o_ptrs, acc.to(O_ptr.dtype.element_ty), mask=o_mask)
```

为了解决刚刚提出的问题，我们需要引入一种“修正“，就是online算法。

Online 的解法：维护全局 max/sum，每来一个新 tile 用 `correction = exp(m_old - m_new)` 把之前的贡献缩放到新的全局 max 下，保证所有 tile 的 softmax 基准一致。

4. 实验结果

4.1 全栈对比（$d=16$）

N	PP(+Autograd)	PP(raw)	C++ (-O2)	PyTorch GPU	Flash-Lite
8	0.209 ms	0.018 ms	0.001 ms	0.054 ms	0.025 ms
16	0.446 ms	0.032 ms	0.005 ms	0.049 ms	0.034 ms
32	0.886 ms	0.061 ms	0.017 ms	0.054 ms	0.020 ms
64	1.551 ms	0.100 ms	0.063 ms	0.056 ms	0.036 ms

PP+AD 比 PP raw 慢 12-16 倍，因为每个 `*`，`+` 都创建 Value 对象。

4.2 GPU 大序列对比（$d=64$）

N	PyTorch	Flash	加速	PyTorch 显存	Flash 显存	显存节省
1024	0.081 ms	0.039 ms	2.1x	8.00 MB	0.25 MB	32x
2048	0.258 ms	0.158 ms	1.6x	32.00 MB	0.50 MB	64x
4096	1.668 ms	0.462 ms	3.6x	128.00 MB	1.00 MB	128x

FlashAttention 的显存基本不随 N 涨，因为不需要存 $N \times N$ 的中间矩阵。

4.3 Lite vs Online 正确性

N	Lite 误差	Online 误差
64	0.002	0.002
128	0.761	0.002
256	1.292	0.002
512	2.287	0.001
1024	3.491	0.001

Lite $N>64$ 就飘了，因为 softmax 只在 64 行上做归一化。Online 所有尺度误差都 <0.003，和 PyTorch 标准版一致。

4.4 计算强度（Arithmetic Intensity = FLOPs / HBM 访存字节数）

N	AI（标准）	AI（Flash）
1024	15.4	261
4096	16.1	1044

标准实现 AI 一直在 16，说明 GPU 大部分时间等数据搬运（Memory Bound）。Flash 的 AI 涨到 1044，是因为计算量一样但搬的数据少了 N 倍，所以每搬 1 字节能干更多活（Compute Bound）。

5. 总结

阶段	解决了什么	还剩什么问题
PP+AD → PP(raw)	去掉 Autograd 的 Value 对象开销（9-34x）	Python 还是慢
PP → C++	编译优化，比解释执行快（2-18x）	单线程 CPU
C++ → PyTorch	GPU 并行，cuBLAS 向量化	中间 $N \times N$ 矩阵占显存
PyTorch → Flash	IO-Aware 分块 + 算子融合，$O(N^2) \to O(N)$	—

这条优化链其实就是 lec5-8 的串联：

换言之,FlashAttention 其实就是这四节课串在一起的结果。