第5次课程-RMSNorm算子的CUDA实现

视频

RMSNorm算子的CUDA实现.mkv

重置之后的视频，声音更清晰

重置后的第五次课程.mp4

算子的计算公式

我们先来看看RMSNorm算子的计算公式。

$$scale = \frac{1}{d}\sum_{i=1}^{d} x_i^2$$
$$rsqrt = \frac{1}{\sqrt{scale + eps}}$$
$$y = x \cdot rsqrt \cdot w$$

其中d是输入的x向量的维度，w是权重需要在计算1，2两式完成时对原结果进行逐点相乘。x是算子计算的输入，y是算子计算的输出。

算子的调用接口

```
1    typedef void (*RMSNormKernel)(const tensor::Tensor& input, const
    tensor::Tensor& weight,  const tensor::Tensor& output, void* stream);
```

以上代码见本节课程附属代码kuiper/source/op/kernels/kernels_interface.h中，在此我们先定义了函数指针类型，要求该计算rmsnorm的函数有一个输入 `input` ，一个输出 `output` ，权重 `weight` ，另外还需要有Cuda流作为参数进行传递。

补充说明：

在CUDA中，一个流是一个序列化的任务队列，它允许开发者将一系列的CUDA操作（如内存复制、内核执行等）放入一个队列中，然后按顺序执行这些操作，而不需要等待前面的操作完全完成。

CPU上的算子实现

```
1   void rmsnorm_kernel_cpu(const tensor::Tensor& input, const tensor::Tensor&
    weight, const tensor::Tensor& output, void* stream = nullptr);
```

可以看到我们在CPU上的算子定义**符合以上函数指针类型 `RMSNormKernel` 的要求**，也就是如上所述的具有一个输入、输出以及权重。但是值得一提的是，CPU上的stream指针参数恒为空，因为这是只有在 `GPU` 上才有的机制。以下是CPU上算子的具体实现：

```
1   void rmsnorm_kernel_cpu(const tensor::Tensor& input, const tensor::Tensor&
    weight,
2                           const tensor::Tensor& output, void* stream) {
3
4     float* in_ptr = const_cast<float*>(input.ptr<float>());
5     float* out_ptr = const_cast<float*>(output.ptr<float>());
```

```
6
7     int size = static_cast<int32_t>(input.size());
8     float sum = 0.f;
9     for (int i = 0; i < size; ++i) {
10      float input_value = input.index<float>(i);
11      sum += input_value * input_value;
12    }
13    const float eps = 1e-5f;
14    float mean = sum / float(size) + eps;
15
16    const float rsqrt = 1.f / std::sqrt(mean);
17    for (int i = 0; i < size; ++i) {
18      *(out_ptr + i) = weight.index<float>(i) * (rsqrt * (*(in_ptr + i)));
19    }
20  }
```

在以上的代码中，`in_ptr` 和 `out_ptr` 分别是该算子输入输出数据指向的指针。在第9行到第12行的代码中，我们先计算输入数据的平方和得到结果 `sum_value`，也就上文公式中的 $\sum_{i=1}^{d} x_i^2$ 随后再计算它的平均值mean，也就是公式中的 $\frac{1}{d}\sum_{i=1}^{d} x_i^2$

第17行算得rsqrt值，也就是公式中的 $\frac{1}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}}$ ，随后再在18行开始的代码中，我们计算该值对输入数据的逐点加权，也就是 `rsqrt` 乘以 `input[i]` ，再逐元素乘以权重得到最终的结果。

CUDA上的算子实现

我们知道Cuda的编程模型 `SIMT` 由于Thread和Block组成，一个Block中有多个Thread负责计算。我以工地施工班组作为一个例子，一个工地有多个班组，也就是多个Block。每个班组(Block)内又有多名工人，每个工人也就是Thread负责具体的工作（比如砌墙、搬砖等）。来看官方和正式一点的解释，我们将对这些概念进行解读：

1. Thread：
    - Thread 是 CUDA 中最基本的执行单元。
    - 在 GPU 上执行的 CUDA 程序会启动大量的线程，这些线程可以在不同的数据上并行执行相同的指令。一个线程可以访问自己的局部数据，同时也可以访问分配给其所在线程块内所有线程的共享数据。

如果现在要处理的数据有 1024 个，应该明确划分数据的方式是通过线程索引和数据总量的关系来确定，例如每个线程可以处理的数据量为 。

2. Block：
    - Block 是由多个线程组成的集合，这些线程在执行时可以被协作地调度和执行。

- Blocks 是网格（Grid）的组成部分，它们以多维的形式组织。
- 同一个块中的线程可以通过共享内存快速交换数据，并且可以同步它们的执行。

在 CUDA 编程中，共享内存是一种特殊的内存，它被同一线程块中的所有线程共享。共享内存是一种快速的、位于芯片上的内存，其访问速度远高于全局内存，但容量有限。

使用共享内存的典型场景如矩阵乘法，每个块负责计算矩阵的一部分结果时，可以将输入矩阵的子块加载到共享内存中，同一块中的线程在计算时可从共享内存快速读取数据，而非全局内存。如果一个块内的线程要访问同一个块内其他线程写入的数据，那么在访问之前需要进行同步操作，以确保数据的正确性。共享内存在硬件上是一小块存储空间，速度比外围全局显存快。

3. Warp：
   - Warp 是 GPU 执行指令的最小调度单元，它由 32 个线程组成。
   - 在执行时，warp 中的所有线程在同一时间步执行相同的指令，但是每个线程可能会处理不同的数据。
   - 这种单指令多线程（SIMT）的模型允许高效的并行处理，因为多个线程可以同时执行相同的操作，减少了控制逻辑的开销。
   - 当 warp 中的任何一个线程需要分支（例如，由于 if-else 语句）时，所有的 32 个线程都会执行分支的所有路径，但是不参与的线程会被屏蔽，以避免执行无效的计算。

Cuda实现的RMS核函数

```
1  static __global__ void row_rmsnorm_f32(const float* in, const float* wei,
   float* out, const int size, const float eps) {
2    const int tid = threadIdx.x; // 线程 id 多个线程是同时进来的。 block 0当中有线程
   0，block1当中也有线程0
3    // tid 的范围等于0到blockDim-1
4    const int lane_id = tid % warpSize; // 这个线程id在warp内的编号
5
6    float sum = 0.0f;
7      // lane_id从0到31都有，是同时执行的
8    for (int i = lane_id; i < size; i += warpSize) {
9      sum += in[i] * in[i];
10     // lane_id=0: in[0]*int[0] + in[32]*in[32] + in[64]*in[64]
11     // lane_id=1: in[1]*in[1] + in[33]*in[33]
12     // lane_id=2: in[2]*in[2]+ in[34]*in[34]
13     // ..
14     // lane_id = 31
15   }
16   // 会有一个局部和，lane_id不同局部和也不同。不同线程在此处的sum是不同的，因为是并行执
   行的。
17   // 下一步就要把局部和，变成全局和
```

和普通 C++ 函数相比，Cuda 核函数 row_rmsnorm_f32 多了一个**__global__**修饰符。在第二行中我们获取到了当前的线程号 tid 以及该线程所属的 warp 内编号。因为 warpSize 等于 32，所以 lane_id 是线程在 warp 内的相对编号，取值范围是 0 到 31。即 lane_id = tid % warpSize。

我们配置了512 个线程，tid 从 0...511 同时进核函数执行，由于一个 warp 有 32 个线程，所以通过 tid % warpSize 可以得到一个 warp 内的编号，也就是lane_id。由于SIMT编程模型中多个线程是同时执行的，所以有lane_id为0-31的线程同时进入第6-8的循环中，我们不难想象有如下的几个循环在不同的线程中同时在进行，以下为伪代码：多个线程同时进入，并行执行。

```
1   for i = 0(lane id); i < size; i += 32;
2       sum += in[i] * in[i]
```

```
1   for i = 1(lane id); i < size; i += 32;
2       sum += in[i] * in[i]
```

```
1   for i = 2(lane id); i < size; i += 32;
2       sum += in[i] * in[i]
```

这是一个warp内的多个线程在并行执行，第一个线程在处理第0，32，64，96，...，size号数据。第二个线程在处理第1，33，65，97，... 号数据。第三个线程在处理第2，34，66，98，号数据，以此类推。随后为了算得总和，我们需要将所有线程的计算结果**局部sum进行求和**，因为原先的sum只是每个线程负责的部分数据的总和，我们还需要将它相加得到实际的总和。共有32个局部和，随后我们要对32个局部和求和得到总和。

```
1   static __global__ void row_rmsnorm_f32(const float* in, const float* wei,
    float* out,  const int size, const float eps) {
2     ...
3     ...
4     using WarpReduce = cub::WarpReduce<float, 32>;
5     __shared__ typename WarpReduce::TempStorage temp;
6     __shared__ float shared_val;
7     sum = WarpReduce(temp).Reduce(sum, cub::Sum());
```

其中 `sum` 是多个局部和的总和，我们来大概了解一下 `warpReduce` 的原理。

```
1   template<const int kWarpSize = WARP_SIZE>
2   __device__ __forceinline__ float warp_reduce_sum(float val) {
```

```
  #pragma unroll
  for (int mask = kWarpSize >> 1; mask >= 1; mask >>= 1) {
    val += __shfl_xor_sync(0xffffffff, val, mask);
  }
  return val;
}
// 所有的局部和加起来。第一轮循环进入的线程id是0...32，mask = 16
// 线程0进来的时候，0+16
// 线程1进来的时候，1+17

// 线程16，16+31
```

这个函数使用模板参数 `kWarpSize`（默认为 `WARP_SIZE`，通常为 32）实现了在 warp 内进行求和操作的功能。通过循环和 `__shfl_xor_sync` 函数，实现了高效的并行求和。

1. 在第一轮循环中，由于 `mask = kWarpSize >> 1`，此时 `mask = 16`。满足循环条件的线程会进行求和操作，例如线程 0 和线程 16 进行求和，线程 1 和线程 17 进行求和，以此类推。这样，线程 0 中保存了线程 0 和线程 16 的加和结果。
2. 在第二轮循环中，`kWarpSize` 变成 8。此时，满足条件的线程继续进行求和操作。线程 0 将和线程 8 求和，线程 1 将和线程 9 求和，以此类推。当本轮循环结束时，线程 0 中保留了线程 0、16、8、24 的局部和。
3. 在第三轮循环中，`kWarpSize` 变成 4。同样，满足条件的线程进行求和操作。线程 0 将和线程 4 求和，线程 1 将和线程 5 求和（这里注意不是线程 9，避免产生误解），以此类推。当本轮循环结束时，线程 0 中保留了线程 0、16、8、24、4、20、12、28 的局部和。
4. 直到最后一轮循环时，`kWarpSize` 等于 1。此时，线程 0 和线程 1 上的局部和相加，最终线程 0 得到了 32 个线程数据的全局和，即线程 0 上保存了来自线程 0、1、2、3……31 的所有局部和。

总之，这个函数通过多次循环和 warp 内线程的数据交换，高效地实现了对多个局部和的求和操作，得到了 warp 内所有线程数据的全局和，我们继续来看 `rmsnorm` 的cuda实现：

```
  static __global__ void row_rmsnorm_f32(const float* in, const float* wei, float* out,  const int size, const float eps) {
      const float scale = rsqrtf(sum / static_cast<float>(size) + eps);
      for (int i = lane_id; i < size; i += warpSize) {
        out[i] = scale * in[i] * wei[i];
      }
  }
```

其中，sum 为前一步所求出的全局和。依据公式，我们可求得 scale 的值。在这个 warp 内存在多个线程。第一个线程在 for 循环中处理第 0、32、64、96…… 直至 size 个数据；第二个线程处理第

1、33、65、97…… 个数据；第三个线程处理第 2、34、98 个数据，依此类推。这些线程之间互不重叠，将每个数据先乘以 scale，再乘以权重 weight，从而得到最终的结果。

下面，我们综合起来看一下 rmsnorm 算子在 CUDA 上的实现：

```
1    static __global__ void row_rmsnorm_f32(const float* in, const float* wei,
   float* out,  const int size, const float eps) {
2        const int tid = threadIdx.x; // 线程 id 多个线程是同时进来的。
3        const int lane_id = tid % warpSize; // 这个线程id在warp内的编号
4
5        float sum = 0.0f;
6          // lane_id从0到31都有，是同时执行的
7        for (int i = lane_id; i < size; i += warpSize) {
8          sum += in[i] * in[i];
9        }
10       // 根据局部和求出sum的全局和
11       using WarpReduce = cub::WarpReduce<float, 32>;
12       __shared__ typename WarpReduce::TempStorage temp;
13       __shared__ float shared_val;
14       sum = WarpReduce(temp).Reduce(sum, cub::Sum());
15
16       const float scale = rsqrtf(sum / static_cast<float>(size) + eps);
17       for (int i = lane_id; i < size; i += warpSize) {
18         out[i] = scale * in[i] * wei[i];
19       }
20   }
```

如何启动rms核函数

具体代码请见 中，以下是如何启动核函数的代码：

```
1  constexpr int threads_num = 128;
2  if (stream) {
3    cudaStream_t stream_ = static_cast<cudaStream_t>(stream);
4    row_rmsnorm_f32<<<1, threads_num, 0, stream_>>>(in_ptr, wei_ptr, out_ptr,
   size, eps);
5  } else {
6    row_rmsnorm_f32<<<1, threads_num>>>(in_ptr, wei_ptr, out_ptr, size, eps);
7  }
```

`<<<1, threads_num>>>` 这里的三个尖括号是Cuda语言特殊的语法，它表示在本次计算中我使用了1个block，然后这一个block中使用了 `thread_num` 个线程。

结果对比

见单元测试test/test_op/test_cu_rmsnorm.cpp，请看视频演示