自制大模型推理框架-第13课-Sgemv算子的实现-cuda


自制大模型
推理框架
带你从零写一个支持LLama2/3推理
支持Cuda加速和int8量化的大模型框架
课程亮点
一、项目整体架构和设计
学习架构思维，我们将在这里学习算子、张量等关键数据结构的设计。
二、支持LLama2模型结构
对LLama模型的权重读取和构建，用KV Cache等机制支持加速推理。
三、模型的量化
为了减少显存的占用，我们会一起开发int8模型量化模块，包括量化模型导出和量化算子支持。
四、Cuda基础和算子实现
带你学Cuda并实战大模型算子的实现，每个算子都将讲解原理，从零手写。
五、用推理框架做点有趣的事情
文本生成，生成故事，多轮对话。
现价优惠 扫码马上报名

上节课的回顾

在上节的课程中，我们已经用模型权重文件去初始化各类型的带参数算子，除了实例化算子之外还将模型权重文件中的权重数据赋值给这些带参数的算子，并且两个后端算子的 `sgemv` 过程，并获得了一致的结果。

所以本节课我们就是要按按照这个思路，继续来实现 `sgemv` 的流程， `sgemv` 简单来说就是将一个权重乘以一个张量。

```
1   1  2  3          1
2   4  5  6          1   =(0,3,6)
3   7  8  9         -1
```

如同上图所示的那样， `weight` 是一个 `3x3` 的矩阵，输入 `input` 是一个3 x 1，我们要在这个算子要做的就是将 `weight` 和 `input` 进行相乘，得到一个 `3x1` 的输出 `output` .

如果手动计算这组输入就是第一步求得1，2，3和1，1，-1的内积，得到1+2-3 = 0，以此类推得到第二步的输出4 + 5 - 6 = 3，第三步的输出为7 + 8 - 9 = 6，最终的结果是(0，3，6)

视频

gpu实现.mkv

量化gemv算子的实现.mp4

选择Cuda后端的算子实现

就像上节课说的那样，我们会根据device_type来选择不同的算子实现：

```
1   MatmulKernel get_matmul_kernel(base::DeviceType device_type) {
2     if (device_type == base::DeviceType::kDeviceCPU) {
3       return matmul_kernel_cpu;
4     } else if (device_type == base::DeviceType::kDeviceCUDA) {
5       return matmul_kernel_cu; // 选择cuda上的实现作为计算方法
6     } else {
7       LOG(FATAL) << "Unknown device type for get an matmul kernel.";
8       return nullptr;
9     }
10  }
```

选择这里的第4行，并返回matmul_kernel_cu在Cuda上对gemv进行计算，计算一共有两个输入，一个是输入张量input，是N行1列的，另一个是权重张量weight是K×N列的，所以我们需要再matmul_kernel_cu中完成权重和输入向量的相乘操作。

Cuda Wrapper

在正式调用到核函数之前，是需要经过一个代理方法的，也就是我们刚才选中的matmul_kernel_cu方法。我们在这个代理方法中指定线程数和线程块的数量，另外也要对一些参数进行检查。

```
1  void matmul_kernel_cu(const tensor::Tensor& input, const tensor::Tensor&
   weight,
2                        const tensor::Tensor& output, const float scale, const
   CudaConfig* config) {
3    CHECK(config != nullptr);
4
5    CHECK(input.is_empty() == false && input.dims_size() <= 2);
6    CHECK(input.device_type() == base::DeviceType::kDeviceCUDA);
7
8    CHECK(weight.is_empty() == false && weight.dims_size() == 2);
9    CHECK(weight.device_type() == base::DeviceType::kDeviceCUDA);
10   const int32_t K = weight.get_dim(0);  // row
11   const int32_t M = weight.get_dim(1);  // col
12   // ...  K个线程块，每个128个线程
13   matmul_kernel_cu_fp32<128, 1><<<K, 128>>>(input.ptr<float>(),
   weight.ptr<float>(),
14                                             const_cast<float*>
   (output.ptr<float>()), M, K);
15 }
```

首先是检查了输入和权重是否为空，且设备类型是否为CUDA类型（因为我们这是一个Cuda算子的调用），K、M分别是权重的维度，这些信息都将作为核函数的辅助信息进行传递。**我们看到<<<K,128>>>这种方式实际上是在向核函数传递配置的线程块数量和线程数量，这里我们将用128个线程做一行内的规约，共有K个线程块每个线程块负责一行。**

我们在matmul_kernel_cu的最后调用matmul_kernel_fp32这个核函数启动计算

规约计算

在开始分析核函数的实现之前，我们来看看规约是怎么实现的。

1	2	3
4	5	6
7	8	9

假设以上有一个3 × 3的权重W，如上表格所示，此时的输入x为：

1

-1
2

我们要做的就是将权重的每行和输入向量进行规约，换句话说就是

1 * 1 + 2 * (-1) + 3 * 2
4 * 1 + 5 * (-1) + 6 * 2
7 * 1 + 8 * (-1) + 9 * 2

<<<K, 9>>>

我们举个例子，假设现在要处理的权重一行共有36个权重

<<<K, 9>>>表示现在一行一共有36个数据，为每一行配置了9个线程，每个线程处理4个数据

第一轮中：

以下过程为并行进行的。步长等于9

线程0对同一行第0，9，18，27位置上的权重和输入数据进行点乘求和。part_sum = in[0] * wei[0] + in[9] * wei[9] + ... + ...

线程1对同一行第1，10，19，28位置上的权重和输入数据进行点乘求和。 part_sum = in[1] * wei[1] + in[10] * wei[10] + ... + ...

线程2对同一行第2，11，20，29位置上的权重和输入数据进行点乘求和。

线程3对同一行第3，12，21，30位置上的权重和输入数据进行点乘求和。

线程4对同一行第4，13，22，31位置上的权重和输入数据进行点乘求和。

...

线程8对某一行第8，17，26，35位置上的权重和输入数据进行点乘求和。

第二轮中：

将线程0和线程4进行相加，并将结果放到线程0中暂存，所以线程0中含有0，9，18，27，4，13，22，31

将线程1和线程5进行相加，并将结果放到线程1中暂存，所以线程1中含有1，10，19，28，5，14，23，32

将线程2和线程6进行相加，并将结果放到线程2中暂存，所以线程2中还有2，11，20，29，6，15，24，33

第三轮中：

步长变为2

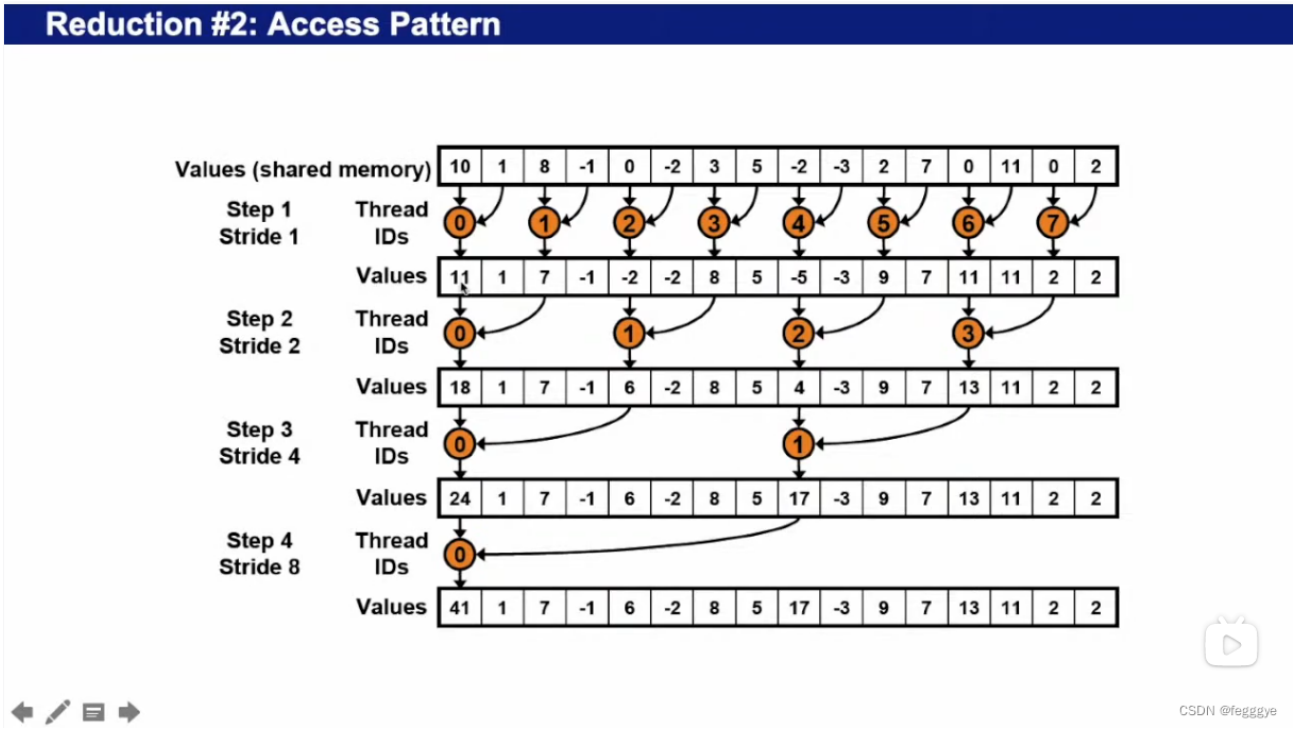
线程0+线程2

所以线程0中含有0，9，18，27，4，13，22，31，2，11，20，29，6，15，24，33的结果

第四轮中：步长等于1

0,1,2,3,4,...,35 结果都放在了线程0中

我们将线程0和线程2进行相加，并将结果再放到线程0中暂存，所以线程0中还有第一轮中线程0、2、4的结果，以此类推，直到多轮后线程0中包含了第一轮中线程0到线程9的局部和的总和，这样就可以得到最终的规约结果。我们给出一个参考的示意图，**图中稍有不同的是我们每个步骤中的步长是逐渐减小的，而图中的步长是逐步增大的，但是最后的结果不会发生变化，同样是将多组局部和集中于一个线程。**



Step等于1的时候每个方格子都代表一个线程，我们在每步的迭代中将一个位置线程上的局部和与step步之外的局部和再求和，每个线程在迭代过程中不断更新其局部和，最终得到整个计算过程中的累积结果。具体流程还可以辅助参考这两节内容，也是讲解Cuda中的Reduce实现和优化的。

第5次课程-RMSNorm算子的CUDA实现

第6次课程-Nsight compute的使用和算子的优化

算子的实现

我们在这里用到了Cub库中的BlockReduce，其本质是和上述的规约流程是一样的。我介绍一种常规实现，可以分为两个步骤：

1. BlockReduce会用到两次的WarpReduce，Warp是Cuda的最基础调度单位，一般是32个线程，而一个线程块(block)中可能会带有多个Warp。

2. float part_sum = BlockReduce(temp).Sum(sdata[tid]); 在一个线程中有这样的调用，我们假设现在一共有128个线程，我们设定编号从0...127，warp大小等于32，所以一共有4个warp。**一个线程块**计算一行的加权和
    - 首先第一个warp，也就是线程id从0到31，BlockReduce首先在一个warp内求和（使用的方法也是线程之间不同步长的规约）也就是将一个warp内的sdata的值进行求和，我们记作w1.

在一个计算配置中，每个行都由一个计算块（block）构成，而每个块又由多个warp组成。具体来说，

1. warp0包含了线程0到线程31，我们可以用t来代表这些线程。计算过程如下：sdata[0] = w[t0] * i[t0]，sdata[1] = w[t1] * i[t1]，依此类推，直到sdata[31] = w[t31] * i[t31]。通过规约操作，我们可以得到w1_part = sum(sdata[0] + sdata[1] + ⋯ + sdata[31])。
2. warp1包含了线程的范围是32到63，同样用t来表示，这些线程处理的是同一行中第31列到第63列的计算：w[t31] * i[t31]，w[t32] * i[t32]，依此类推，直到w[t63] * i[t63]。通过规约操作，我们可以得到w2_part。
3. warp2包含了线程64到线程95，它们负责同一行中第64列到第95列的计算：w[t64] * i[t64]，w[t65] * i[t65]，依此类推，直到w[t95] * i[t95]。通过规约操作，我们可以得到w3_part。
4. 接下来，我们还需要进行一次规约操作来得到w4_part。
5. 最后，完成这些warp内的规约后，**我们将w1_part、w2_part、w3_part和w4_part的值放入同一个warp中**，并将该warp的其他位置值置为0。最后，对该warp进行规约操作，以得到最终的结果。

我们找到了一个BlockReduce可能的实现：

```
// 在一个warp（32个线程里做规约，返回最大值）
__device__ __forceinline__ float warp_reduce_max(float val) {
  #pragma unroll
  for (int mask = kWarpSize >> 1; mask >= 1; mask >>= 1) {
    val = fmaxf(val, __shfl_xor_sync(0xffffffff, val, mask));
  }
  return val;
}
// 配置了K个线程块，一个线程块block，负责一行。block_reduce_sum就是负责一行的加权和
__device__ __forceinline__ float block_reduce_sum(float val) { // 对一行加权和的结果进行规约
  int warp = threadIdx.x / WARP_SIZE;
  int lane = threadIdx.x % WARP_SIZE;
  static __shared__ float shared[NUM_WARPS];

  // 分warp求出,warp_part1,warp_part2,warp_part3,warp_part4
  val = warp_reduce_sum<WARP_SIZE>(val);
  if (lane == 0)
     shared[warp] = val;
```

```
    __syncthreads();
    val = (lane < NUM_WARPS) ? shared[lane] : 0.0f;
    val = warp_reduce_sum<NUM_WARPS>(val);
    return val;
  }
```

1. 对BlockReduceSum的调用，softmax_v2中的多个线程同时调用block_reduce_sum，随后调用**warp_reduce_sum**并根据这些线程的warp归属求出属于每个warp的局部之和。就像上文所说的那样，如果一行共有128个数据，那么warp0中的局部和记录了数据0-31，warp1中的局部和记录了数据32-63，warp2中的局部和记录了数据64-95，每个warp内的计算方式就是以规约的方式进行的。
2. 我们再将warp0,warp1,warp2和warp3中的归约结果放到shared数组里，并将shared数组的其他位置置为0，由于shared数组的大小是32，**放满数据后我们再对其中的值进行分步长规约得到最终的结果（第21行）**，从此就可以得到整行的

我们来看一下cuda gemv的步骤：matmul_kernel_cu_fp32

1. 一个block可能会处理多行，从start_row行到end_row行，对每行的计算结果进行规约；

```
    __shared__ float sdata[THREAD_PER_BLOCK];
    unsigned int tid = threadIdx.x;
     // 找到当前线程块要处理的行
    int start_row = blockIdx.x * ROW_PER_BLOCK;
    int end_row = start_row + ROW_PER_BLOCK;
    if (start_row >= K) {
      return;
    }

    constexpr int pack_size = 4;
    const int pack_num = M / pack_size;
    const int pack_off = pack_size * pack_num;
```

2. 对一行权重和一行输入进行相乘后相加，我们采用了向量化读取的方式（这个前面的课程中有涉及，请大家自行回顾），**总的来说sdata数组中存放的是一行中每个位置的相乘值 weight[i] * input[i]**，我们下面就要将sdata[]数组进行规约求和，得到一行相乘后相加的结果。

1	**2**	**3**	**4**	5	6

```
  #pragma unroll
    for (int p = start_row; p < end_row; ++p) {
```

```
    sdata[tid] = 0;
    int row_offset = p * M;
    // 获取到输入和权重某一行的起始指针
    float4* input_float4_ptr = (float4*)input;
    float4* weight_float4_ptr = (float4*)(weight + row_offset);

#pragma unroll  // 将同一行中的权重和输入相乘并放入sdata数组中
    for (int i = tid; i < pack_num; i += blockDim.x) {
      float4 input_float4 = *(input_float4_ptr + i);
      float4 weight_float4 = *(weight_float4_ptr + i);
      // 同一行不同位置中输入和权重的相乘并相加，均记录在sdata中
      float part_sum = input_float4.x * weight_float4.x + \
       input_float4.y * weight_float4.y + input_float4.z * weight_float4.z \
       + input_float4.w * weight_float4.w;
      sdata[tid] += part_sum;
    }

    for (int i = pack_off + tid; i < M; i += blockDim.x) {
      sdata[tid] += input[i] * weight[row_offset + i];
    }
```

3. 同一个block内的线程进行同步，以确保同一行内每个位置中的数据都已经被写入到sdata了，随后再对sdata数组进行归约得到最终的结果，并写入到结果（input乘以weight一行的结果）的p行中。

```
__syncthreads();

using BlockReduce = cub::BlockReduce<float, THREAD_PER_BLOCK>;
__shared__ typename BlockReduce::TempStorage temp;
//得到一行的结果,blockreduce的原理上方已经讲过
float part_sum = BlockReduce(temp).Sum(sdata[tid]);
__syncthreads();

if (tid == 0) {
   output[p] = part_sum;
}
__syncthreads();
```

单元测试

```
TEST(test_matmul_cu, matmul_linear_course_cuda) {
  auto alloc_cu = base::CUDADeviceAllocatorFactory::get_instance();
```

```
3     auto alloc_cpu = base::CPUDeviceAllocatorFactory::get_instance();
4
5     tensor::Tensor input(base::DataType::kDataTypeFp32, 3, true, alloc_cpu);
6     tensor::Tensor weight(base::DataType::kDataTypeFp32, 3, 3, true, alloc_cpu);
7
8     input.index<float>(0) = float(1);
9     input.index<float>(1) = float(1);
10    input.index<float>(2) = float(-1);
11
12    for (int i = 1; i <= 9; ++i) {
13      weight.index<float>(i - 1) = float(i);
14    }
15
16    input.to_cuda();
17    weight.to_cuda();
18
19    tensor::Tensor out_cu(base::DataType::kDataTypeFp32, 3, true, alloc_cu);
20
21    kernel::get_matmul_kernel(base::DeviceType::kDeviceCUDA)(input, weight,
    out_cu, 1.f, nullptr);
22
23    tensor::Tensor out_cpu = out_cu.clone();
24
25    ASSERT_EQ(out_cpu.index<float>(0), 0);
26    ASSERT_EQ(out_cpu.index<float>(1), 3);
27    ASSERT_EQ(out_cpu.index<float>(2), 6);
28  }
```

在本节课中我们同样是用，输入是[1,1,-1]，权重是

1	2	3
4	5	6
7	8	9

得到的结果为0，3，6，符合我们在单元测试中的验证。

量化算子的实现

我们下面还将实现一个量化的GEMV算子，以下是对之前量化相关内容的回顾：下面将用宽泛的方式进行分析，在逐组量化中，假设我们有W个权重，我们将它分成G组，每组的权重个数是W/G个。在每组中我们都求得一个最大值，记作RMAX，共有group个。再举个实际的例子，**例如现在有4个权重数据，**分别是[1，3，5，-1]，每组的权重是两个。我们求出每组的最大值分别是3和5。

$$Scale = \frac{|r_{max}|}{|q_{max}|}$$

这里的量化系数是按组来求得的，随后我们再对输入数据进行量化，其中r表示原始浮点数据，**q表示量化后输出的整型数据。**

$$q = Round(\frac{r}{scale})$$

总结一下以上的流程：

- 首先将[1，3，5，-1] 4个权重分成两组，每组是2个fp32数据。
- 随后对每组求出一个最大值分别3，5，也就是等式中的rmax。
- 随后对各组求出scale , 当qmax = 127，scale**各组**分别等于3/127=0.02362, 5/127=0.03937
- 量化的值 round(1/0.02362) = 42（因为g该组的系数是0.039），round(5/0.03937) = 127

所以我们在经过量化之后，得到4个q值也是原来的4个浮点数经过量化后的int8值，还有group个scale值作为系数。如果要反量化的话，就有q×scale=r，其中r是原先的浮点数。q1,q2,q3,q4，每个组各自有一个系数。

cuda实现

```
template <int THREAD_PER_BLOCK, int ROW_PER_BLOCK>
__global__ void matmul_kernel_cu_fp32int8(const float* input, const int8_t* weight,
                                          const float* scales, const int32_t group_size, float* output, int M, int K) {
  __shared__ float sdata[THREAD_PER_BLOCK];
  unsigned int tid = threadIdx.x;

  int start_row = blockIdx.x * ROW_PER_BLOCK;
  int end_row = start_row + ROW_PER_BLOCK;
  if (start_row >= K) {
    return;
  }
  for (int p = start_row; p < end_row; ++p) {
    sdata[tid] = 0;
    for (int i = tid; i < M; i += THREAD_PER_BLOCK) {
      const int weight_idx = p * M + i; // weight_idx指向当前处理权重的位置
      const int group_idx = weight_idx / group_size; // 当前位置权重所属的组。
      // 一个组有一个系数，scales[group_idx], r = q*scales浮点权重值
      // r_weight = scales * static_cast<float>(weight[weight_idx] int8); 得到浮点权重
      // r_weight * input[i]
      sdata[tid] += input[i] * scales[group_idx] * static_cast<float>(weight[weight_idx]);
    }
```

```
22
23      // 0,....64 thead block 大小是4
24      // tid = 0 , 列0,4, 8, 12, 16
25      // tid = 1, 列1, 5, 9, 13, 17
```

除了权重参数经过量化后已转换为一组Int8值之外，其他参数和之前的非量化gemv算子均保持一致。以下是对量化gemv算子实现的分析：

1. 在1-2行中，tid表示当前线程在它线程块block内的编号。
2. 在8-9行中，**我们指定了当前线程块block需要计算的行数**，起始行是start_row，结束行是end_row；
3. 与之前的非量化gemv算子相同，我们需要对一行内的输入和权重进行规约，以得到一个累计值。正如第2点所述，一个线程块需要处理从start_row到end_row中的每一行，逐行将权重和改行输入进行归约操作。
4. 在15-19行中，**p表示当前正在处理的行号**，**weight_idx指向当前处理权重的位置**，也就是权重矩阵的第p行第i个位置。我们将同一线程处理的局部和记录在sdata[]数组中，就像上文说的那样，sdata[0]中存放了第0，THREAD_PER_BLOCK, 2×THREAD_PER_BLOCK,...位置的局部和。

   但是计算的过程中，**我们需要先将int8的输入乘以该组的系数scale得到反量化后的浮点输入再乘以权重**。group_size和量化权重导出时需要保持一致，weight_idx/group_size等于当前位置权重所属的组。

   以上就是量化和非量化GEMV算子在实现上的差异。

规约计算

在开始分析核函数的实现之前，我们来看看规约是怎么实现的。

1	2	3
4	5	6
7	8	9

假设以上有一个3 × 3的权重W，如上表格所示，此时的输入x为：

1
-1
2

我们要做的就是将权重的每行和输入向量进行规约，换句话说就是

1 * 1 + 2 * (-1) + 3 * 2
4 * 1 + 5 * (-1) + 6 * 2
7 * 1 + 8 * (-1) + 9 * 2