自制大模型推理框架-第13课-Sgemv算子的实现-cpu


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

视频

cpu实现.mkv

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3x1，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的乘积和，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)

Armadillo数学库学习

基本操作

我们在这节课要完成CPU后端上该算子的实现，CPU上我们用到了Armadillo数学库，先来看一下这个数学库的基础操作：

```
1   mat A(5, 5, fill::randu)
2   mat B(5, 5, fill::randu)
```

在这里我们用随机值初始化了2个5×5的矩阵，分别是矩阵A和矩阵B。

```
1   mat E = A + B;
2   mat F = A * B; // 矩阵乘法
3   mat G = A % B; // 矩阵逐点相乘
```

我们再在这里将两个矩阵相加，得到矩阵E；将矩阵A和矩阵B矩阵乘得到矩阵F；**这里要注意的是这里的%号，它是一种逐点相乘的操作，而不是取模操作，例如现有一个矩阵A、B它们的值分别是：**

```
1 2 3       1 2 3
4 5 6       4 5 6
7 8 9       7 8 9
```

那么这里的G矩阵，就是：

```
1  4  9
16 25 36
49 64 81
```

不难看出就是将两个矩阵的每个位置的值两两相乘，这就是%符号的作用。**实际的实现方式是重载了%这个运算符，下面是一种可能的实现：**

```cpp
#include <iostream>

class Demo {
public:
    Demo &operator%(const Demo &d) { // C++重载运算符
        v *= d.v;
        return *this;
    }

public:
    int v = 3;
};


int main() {
    Demo d1;
    d1.v = 5;

    Demo d2;
    d2.v = 6;

    d1 = d1 % d2;
    std::cout << d1.v << "\n";
    return 0;
}
```

从第5到第9行的代码中可以看出，我们重载了%号这个运算符，**让它完成的功能是相乘其本身和d实例**，当相乘得到结果之后再返回当前实例。`d1`实例中的值是5，`d2`实例中的值是6，我们现在做的

运算是 `d1 % d2` 应当返回两者相加后的结果，在这项运算中我们将两个Demo实例的类内变量v进行相乘。

复用内存

假如我们已经有一组输入，分别为

```
1  1 2 3 4 5 6
```

这组输入存放在**arr[6]**当中，而且此时我们又不想重新申请mat空间来存放这组数据，我们就可以用到armadillo的一个功能，就是内存复用。

该构造函数的定义为：

```
1     inline Mat(eT* aux_mem, const uword aux_n_rows,\
2                const uword aux_n_cols, const bool copy_aux_mem = true,\
3                const bool strict = false);
```

我们现在需要使用一组已知的浮点数数据来初始化一个矩阵（Mat）。为此，我们应该按照以下格式进行填充：（arr, 1, 6, false）。在这里，arr代表我们将使用arr数组来初始化这个矩阵；1和6分别代表这个矩阵的行数和列数；**参数 `copy_aux_mem` 设置为false，意味着我们直接使用arr数组存储空间直接来初始化矩阵，而不需要对arr数组进行额外的拷贝操作。**

Armadillo数据库中的列主序

列主序就是一种先列后行排列的方式，假设我们现在有一组数据为：

```
1  1 2 3 4 5 6 7 8
```

总计有8个数字，现在要将它排成一个4×2大小的矩阵，如果是行主序的方式则有：

```
1  1 2
2  3 4
3  5 6
4  7 8
```

可以看出，如果是行主序，我们就是以先后列的方式进行排列。如果是列主序的方式则有，方式就是先列后行。

```
1 5
2 6
3 7
4 8
```

内容测试

```
float arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
arma::fmat vec(arr, 4, 2, false);
std::cout << vec;
std::cout << "arr address: " << arr;
std::cout << "vec address: " << vec.mem;
```

输出内容

```
1.0000   5.0000
2.0000   6.0000
3.0000   7.0000
4.0000   8.0000
arr address: 0x7ffda53a4bc0
vec address: 0x7ffda53a4bc0
```

我们可以观察到，这个矩阵是一个具有4行2列的结构，并且它遵循的是列优先的排列方式。此外，值得注意的是，`vec`矩阵在复用后，其内存指针位置与原始的`arr`指针地址是保持一致的。在处理二维矩阵时，若行列的存储顺序不同，它们将呈现出一种转置的关系。

Matmul(Linear)算子的PyTorch实现

torch.nn.functional.linear

> torch.nn.functional.linear(*input*, *weight*, *bias=None*) → Tensor

Applies a linear transformation to the incoming data: $y = xA^T + b$.

This operation supports 2-D `weight` with sparse layout

在公式中变量x代表输入，A代表权重，而b代表偏置量。需要注意的是，在对A进行处理时，我们首先要对其进行转置，记作 A^T 。完成转置后，我们再将输入x与转置后的权重矩阵 A^T 进行矩阵乘法

运算。

因此，在开始计算之前，我们首先需要准备一组输入数据，我们将其命名为 `input` 。正如在之前的课程中所提到的，我们会将这组输入数据封装在一个Tensor类中。然而，需要注意的是，在使用 `Armadillo` 库时，由于其采用列主序的存储机制，可能会遇到一些需要注意的问题。假设我们的原始输入数据为 `input` ，我们可以将其表示为A。当我们将其存储到 `Armadillo` 的Mat结构中时，会发生一个转置的过程，使得原始的A变为 A^T 。

我们的实现

由于输入和权重在处理过程中会因主序问题而发生转置，当我们直接将权重A和输入x装入系统时，它们实际上会变为 A^T 和 x 。需要注意的是，由于x是一个N×1的向量，所以在将其数据装入时，不会出现因行列主序不同导致的转置现象。

假设权重A为如下：

```
1  1 2 3
2  4 5 6
```

输入x等于如下：

```
1  7 8 9
```

按照上图中Pytorch的公式，nn.Linear中 xA^T 计算方法为：

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

但是因为有行列主序计算的问题，存入到Armadillo矩阵后，会得到一个转置后的 A 矩阵，我们记作 B :

```
1  1  4
2  2  5
3  3  6
```

输入 x 仍然是

```
1  7 8 9
```

综上所述，我们如果想保持原有的计算结果，计算过程需要变为 xB ，随后我们从选算子实现开始讲解：

```
MatmulKernel get_matmul_kernel(base::DeviceType device_type) {
  if (device_type == base::DeviceType::kDeviceCPU) {
    return matmul_kernel_cpu;
  } else if (device_type == base::DeviceType::kDeviceCUDA) {
    return matmul_kernel_cu;
  } else {
    LOG(FATAL) << "Unknown device type for get an matmul kernel.";
    return nullptr;
  }
}
```

如果当前的设备后端是 `CPU` 的，也就是 `device_type` 是 `base::DeviceType::kDeviceCPU` 时候，我们就会选择 `matmul_kernel_cpu`

```
void matmul_kernel_cpu(const tensor::Tensor& input, const tensor::Tensor&
weight,
                       const tensor::Tensor& output, float scale,
                       const CudaConfig* config) {
  // ...
  // ... 省略参数检查的部分
  // ... ...

  const float* input_ptr = input.ptr<float>();
  const float* weight_ptr = weight.ptr<float>();
  const float* output_ptr = output.ptr<float>();

  const int32_t wei_dim0 = weight.get_dim(0);
  const int32_t wei_dim1 = weight.get_dim(1);

  arma::fmat input_mat(const_cast<float*>(input_ptr), in_dim1, in_dim0,
false, true);
  arma::fmat weight_mat(const_cast<float*>(weight_ptr), wei_dim1, wei_dim0,
false, true);
  arma::fmat output_mat(const_cast<float*>(output_ptr), in_dim1, wei_dim0,
false, true);
  output_mat = ((input_mat * weight_mat)) * scale;
}
```

首先我们获得输入、输出和权重指针，见代码第8-10行，随后再复用这些输入、输出和权重数据的空间创建armadillo中的矩阵实例，最后再将输入和权重相乘得到最后的结果。

单元测试

我们准备了一个输入，它的值分别为(1, 1, -1)，权重为

```
权重      输入
1 2 3
4 5 6   输入是(1,1,-1)
7 8 9

放入到Armadillo数学库后
由于列主序W会发生转置

                    1  4  7
输入是(1,1,-1)      2  5  8
                    3  6  9
```

我们将它们以 xA^T 的方式相乘，就像前文说的那样由于数学库的特殊性，所以会有列主序的问题，解决方法如上一节说的那样。

```
TEST(test_matmul_cu, matmul_linear_course) {
  auto alloc_cpu = base::CPUDeviceAllocatorFactory::get_instance();

  tensor::Tensor input(base::DataType::kDataTypeFp32, 3, true, alloc_cpu);
  tensor::Tensor weight(base::DataType::kDataTypeFp32, 3, 3, true, alloc_cpu);

  input.index<float>(0) = float(1);
  input.index<float>(1) = float(1);
  input.index<float>(2) = float(-1);

  for (int i = 1; i <= 9; ++i) {
    weight.index<float>(i - 1) = float(i);
  }
  tensor::Tensor input_cpu = input.clone();
  tensor::Tensor weight_cpu = weight.clone();

  input.to_cuda(nullptr);
  weight.to_cuda(nullptr);

  tensor::Tensor out_cpu(base::DataType::kDataTypeFp32, 3, true, alloc_cpu);

```

```
22    kernel::get_matmul_kernel(base::DeviceType::kDeviceCPU)(input_cpu,
    weight_cpu, out_cpu, 1.f,
23                                                            nullptr);
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  }
```

如果你想看Cuda上的实现，就扫描一开始的二维码来参加课程。