自制大模型推理框架-第4次课程-张量的设计与实现

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

第四章-张量的设计与实现

视频

第四章-张量.mkv

更新后的视频，画面、声音更清晰：

重新录制后的第四章.mkv

张量是一个多维数组，用于在推理的流程中管理，传递数据，同时也能配合第二次课程的Buffer类来自动管理内存或者显存资源。我们先来看一下Tensor类中的数个类内变量

```
class Tensor{
    ...
    ...
private:
  size_t size_ = 0;
  std::vector<int32_t> dims_;
  std::shared_ptr<base::Buffer> buffer_;
  base::DataType data_type_ = base::DataType::kDataTypeUnknown;
};
```

1. `size_` 是张量中数据的个数，比如张量后期要存储3个数据，分别为{1，2，3}，那么size的大小就等于3。
2. `dims` 就是张量的维度，比如有一个二维张量，且维度分别是{2, 3}，那么 `dim_` 记录的值就是{2,3}
3. `buffer_` 就是我们第二章中介绍过的， `Buffer` 类用来管理用分配器申请到的内存资源，在 `Buffer` 中我们用到 `RAII` 和智能指针技术。
4. `buffer->alloc` 会根据buffer中的设备类型去申请对应设备上的内存/显存资源，并**在buffer实例析构的时候**自动释放相关的内存/显存资源。

我们来回顾一下 `buffer->allocate` ，它会为 `Buffer` 内部申请一块资源，并将 `use_external` 置为false，表示这块资源需要进行管理。

```
bool Buffer::allocate() {
  if (allocator_ && byte_size_ != 0) {
    use_external_ = false;
    ptr_ = allocator_->allocate(byte_size_);
    if (!ptr_) {
      return false;
    } else {
      return true;
```

```
    }
  } else {
    return false;
  }
}
```

同理我们看 `Buffer` 的析构函数中，如果 `use_external` 等于false，表示这块资源不属于外部，而属于这块Buffer实例，所以这块资源需要被管理。

```
Buffer::~Buffer() {
  if (!use_external_) {
    if (ptr_ && allocator_) {
      allocator_->release(ptr_);
      ptr_ = nullptr;
    }
  }
}
```

为什么Tensor中有 `std::shared_ptr<Buffer>` 来保存数据呢，假设有以下的流程：

```
tensor::Tensor t1(3);
tensor::Tensor t2 = t1;
```

在第一行代码中， `Tensor t1` 申请了一个3 × sizeof(element)大小的内存，在第二行随后用 `t1` 对 `t2` 进行赋值，所以

```
std::shared_ptr<base::Buffer> buffer_;
```

所以t1和t2内部共享同一块Buffer，且这块Buffer的引用计数等于2，也就是说直到这两个张量 `t1` 和 `t2` 都销毁后才会触发Buffer的析构函数进行资源释放。

换句话说 `t1` 中的buffer是两个张量共有的，只有当使用这块 `buffer` 的 `tensor` 都销毁后才会被释放。

```
{
    tensor::Tensor t1(3);
    tensor::Tensor t2 = t1;
}
```

我们再举一个例子，当代码流程执行到括号外时，因为两个张量变量因为都是类内变量，所以会在第四行对两个张量都进行销毁。这是C++ RAII的内容，局部变量退出作用域后自动释放。

为张量分配内存/显存

我们来看一个Tensor的构造函数：

```
1   Tensor::Tensor(base::DataType data_type, int32_t dim0, int32_t dim1, bool
    need_alloc,
2                  std::shared_ptr<base::DeviceAllocator> alloc, void* ptr)
3       : data_type_(data_type) {
4     dims_.push_back(dim0);
5     dims_.push_back(dim1);
6     size_ = dim0 * dim1;
7     if (need_alloc && alloc) {
8       allocate(alloc);
9     }
10    ...
```

在构造函数中传入了数据类型 `data_type` ，张量维度 `dim_0` 和 `dim_1` ， `need_alloc` 表示是否需要用内存分配器 `alloc` 来分配内存/显存。 `size_` 是维度的乘积，也就是张量中数据的个数。

使用方式如下：

```
1    tensor::Tensor input_tokens(base::DataType::kDataTypeInt32,
2                                static_cast<int32_t>(max_seq_len), true,
    alloc_cpu);
```

我们在这里就申请了一个数据类型为 `int32` ，数据维度为 `max_seq_len` 并使用 `alloc_cpu` 内存分配器申请资源。

单元测试

```
1   TEST(test_tensor, init1) {
2     using namespace base;
3     auto alloc_cu = base::CPUDeviceAllocatorFactory::get_instance();
4
5     int32_t size = 32 * 151;
6     // 倒数第二个参数就是need_alloc
7     tensor::Tensor t1(base::DataType::kDataTypeFp32, size, true, alloc_cu);
8     ASSERT_EQ(t1.is_empty(), false);
```

```
}
```

如下，`alloc`和`need_alloc`参数有一个为空或者false，则张量为空。

```
TEST(test_tensor, init2) {
  using namespace base;
  auto alloc_cu = base::CPUDeviceAllocatorFactory::get_instance();

  int32_t size = 32 * 151;

  tensor::Tensor t1(base::DataType::kDataTypeFp32, size, false, alloc_cu);
  ASSERT_EQ(t1.is_empty(), true);
}
```

如何用已经有的内存赋值tensor

```
float* attn_ptr = nullptr;
cudaMallocManaged(reinterpret_cast<void**>(&attn_ptr),
                  config_->head_num_ * config_->seq_len_ * sizeof(float));
tensor::Tensor attn(base::DataType::kDataTypeFp32, config_->head_num_,
                    config_->seq_len_, false, alloc_cu, attn_ptr);
```

我们先是用`cudaMallocManaged`分配了一块显存`attn_ptr`，后续为了让张量`attn`可以自动管理这块显存，我们使用了构造函数的这一分支(第11行开始的），上方代码第4行我们除了将这块显存attn_ptr传入到Tensor的构造函数之外，还需要传入相关的维度**(config->head_num_, config->seq_len_)**以及显存的allocator。

```
Tensor::Tensor(base::DataType data_type, int32_t dim0, int32_t dim1, int32_t dim2,
               bool need_alloc, std::shared_ptr<base::DeviceAllocator> alloc, void* ptr)
    : data_type_(data_type) {
  dims_.push_back(dim0);
  dims_.push_back(dim1);
  dims_.push_back(dim2);
  size_ = dim0 * dim1 * dim2;
  if (need_alloc && alloc) {
    allocate(alloc);
  } else {
    if (ptr != nullptr) {
```

```
        CHECK(need_alloc == false)
            << "The need_alloc is is true when ptr parameter is not a null
    pointer.";
        if (!alloc) {
          // 如果传入一个空的allocator，表示该tensor不会对该显存进行管理
          std::shared_ptr<base::Buffer> buffer = std::make_shared<base::Buffer>(
              data_type_size(data_type) * size_, nullptr, ptr, true);
          this->buffer_ = buffer;
        } else {
          // 反之，如果传入一个非空的allocator，表示该tensor会对该显存进行管理，
          // 在Tensor生命周期结束后就会释放这块显存
          std::shared_ptr<base::Buffer> buffer = std::make_shared<base::Buffer>(
              data_type_size(data_type) * size_, alloc, ptr, false);
          this->buffer_ = buffer;
        }
      }
    }
  }
```

在这个分支中，如果传入了 `alloc` 参数不为空，则表示 `buffer` 对传入的 `ptr` 是管理关系，需要在所有对 `buffer` 有引用的所有 `tensor` 销毁后，自动释放 `ptr` 指针指向的内存/显存资源。

在以下的单元测试中，我们将已经有的指针对 `tensor` 赋值，可以看到tensor的值和ptr指向的内存地址和原始的指针都是一致的。

```
TEST(test_tensor, init3) {
  using namespace base;
  float* ptr = new float[32];
  ptr[0] = 31;
  tensor::Tensor t1(base::DataType::kDataTypeFp32, 32, false, nullptr, ptr);
  ASSERT_EQ(t1.is_empty(), false);
  ASSERT_EQ(t1.ptr<float>(), ptr);
  ASSERT_EQ(*t1.ptr<float>(), 31);
}
```

对张量数据的访问

```
template <typename T>
T* Tensor::ptr() {
  if (!buffer_) {
    return nullptr;
  }
  return reinterpret_cast<T*>(buffer_->ptr());
```

```
}

template <typename T>
T* Tensor::ptr(int64_t index) {
  CHECK(buffer_ != nullptr && buffer_->ptr() != nullptr)
      << "The data area buffer of this tensor is empty or it points to a null pointer.";
  return const_cast<T*>(reinterpret_cast<const T*>(buffer_->ptr())) + index;
}
```

`ptr()` 方法返回某个tensor中起始数据位置， `ptr(index)` 返回某个tensor中第index个的数据位置。模板参数类型T是将buffer中管理的原始指针转为相应的类型。**为什么要先将void*类型转换为T*类型呢?**

这是因为 `T*` 的步长和 `void*` 的步长是不一致的，比如我要访问 `ptr+1` 的位置，如果是直接将 `void*+1` 那么指针位置只会往后移动一个位置，如果是 `float*` ，那么指针的位置是往后移动4个位置的。

```
TEST(test_tensor, index) {
  using namespace base;
  float* ptr = new float[32];
  auto alloc_cu = base::CPUDeviceAllocatorFactory::get_instance();
  ptr[0] = 31;
  tensor::Tensor t1(base::DataType::kDataTypeFp32, 32, false, nullptr, ptr);
  void* p1 = t1.ptr<void>();
  p1 += 1;

  float* f1 = t1.ptr<float>();
  f1 += 1;
  ASSERT_NE(f1, p1);
  delete[] ptr;
}
```

可以看到在以上的单元测试中， `f1` 和 `p1` 在各自都加1后指针地址是不一致的。

```
template <typename T>
const T& Tensor::index(int64_t offset) const {
  CHECK_GE(offset, 0);
  CHECK_LT(offset, this->size());
  const T& val = *(reinterpret_cast<T*>(buffer_->ptr()) + offset);
  return val;
}
```

`index` 和 `ptr` 方法的不同就是访问对应位置指针的基础上，也就是 `buffer_->ptr()+offset` ，还取出了该地址的数据并返回。

辅助方法

例如现在有一个张量维度为{1，2，3}，如果我们要返回它某个维度，则可以使用：

```
int32_t Tensor::get_dim(int32_t idx) const {
  CHECK_GE(idx, 0);
  CHECK_LT(idx, this->dims_.size());
  return this->dims_.at(idx);
}
```

返回某个维度元素的步长，比如第一维上元素的步长为6 = 2×3，第二维上的元素步长为3，第三维上的步长为1.

```
std::vector<size_t> Tensor::strides() const {
  std::vector<size_t> strides;
  if (!dims_.empty()) {
    for (int32_t i = 0; i < dims_.size() - 1; ++i) {
      size_t stride = ReduceDimension(dims_.begin() + i + 1, dims_.end(), 1);
      strides.push_back(stride);
    }
    strides.push_back(1);
  }
  return strides;
}
```

`strides` 的计算方式如下，如果有dim1,dim2,...dim5的时候，在dim1维上的步长为dim2 × dim3 × dim4 ... × dim5，在dim2的时候步长等于dim3×...×dim5.

```
TEST(test_tensor, dims_stride) {
  using namespace base;
  auto alloc_cu = base::CPUDeviceAllocatorFactory::get_instance();

  tensor::Tensor t1(base::DataType::kDataTypeFp32, 32, 32, 3, true, alloc_cu);
  ASSERT_EQ(t1.is_empty(), false);
  ASSERT_EQ(t1.get_dim(0), 32);
  ASSERT_EQ(t1.get_dim(1), 32);
  ASSERT_EQ(t1.get_dim(2), 3);

  const auto& strides = t1.strides();
```

```
12      ASSERT_EQ(strides.at(0), 32 * 3);
13      ASSERT_EQ(strides.at(1), 3);
14      ASSERT_EQ(strides.at(2), 1);
15    }
```

在上面的单元测试中可以看到， `t1` 返回的三个维度分别为32、32和3，步长 `stride` 分别为96，3和1.