自制大模型推理框架-第2次课程-内存和设备的管理

如何参加课程-《动手写大模型推理框架》

《动手自制大模型推理框架》课程目录：

《自制大模型推理框架》课程目录

《动手自制大模型推理框架》课程优势

1. 采用最新的C++ 20标准去写代码，统一、美观的代码风格，良好的错误处理；
2. 优秀的项目管理形式，我们采用CMake+Git的方式管理项目，接轨大厂；
3. 授人以渔，教大家怎么设计一个现代C++项目，同时教大家怎么用单元测试和Benchmark去测试验证自己的项目；
4. CPU和CUDA双后端实现，对时新的大模型（LLama3和Qwen系列）有非常好的支持。

视频

资源管理.mkv

评论（1）

重录后：重录后-内存的管理和设备类.mkv，声音清晰

可以任意选一

设计内存的分配和释放

设计需求

- 在不同的设备中，我们都会用到它的资源申请、资源释放等操作，我们的要求是在不同的设备下这些接口对外的参数都是一致的，这样一来就方便上方的调用者使用统一的接口进行调用。
- 另外你还需要对申请到的资源进行统一的管理，相当于你申请到的是一块裸指针，但是常常会忘记释放它，那么如何对它进行有效的管理呢，我们可以通过智能指针记录根据它的使用者，并对没有使用资源的自动释放。

我们将在下文中一个一个来解决这些问题，**我们将设备资源申请器抽象为一个父类，**叫做`DeviceAllocator`类，它有一个`allocate`接口，负责申请该类设备下的size大小个字节。有一个`release`接口，负责回收对该设备申请得到的资源指针`ptr`.完整代码见include\base\alloc.h 。

```
class DeviceAllocator {
 public:
  explicit DeviceAllocator(DeviceType device_type) :
device_type_(device_type) {}

  virtual DeviceType device_type() const { return device_type_; }

  // 需要释放的内存
  virtual void release(void* ptr) const = 0;

  // 需要申请的内存/显存字节数
  virtual void* allocate(size_t byte_size) const = 0;

  virtual void memcpy(const void* src_ptr, void* dest_ptr, size_t
byte_size,
                      MemcpyKind memcpy_kind =
MemcpyKind::kMemcpyCPU2CPU, void* stream = nullptr,
                      bool need_sync = false) const;

  virtual void memset_zero(void* ptr, size_t byte_size, void*
stream, bool need_sync = false);

 private:
  DeviceType device_type_ = DeviceType::kDeviceUnknown;
};
```

其中`device_type`表示该资源申请类所属的设备类型，因为`DeviceAllocator`是一个基类，所以此处的值是Unknown，而在派生类中该变量将具体指明当前设备的类型。拷贝时需要提供额外的参数，包括拷贝的方向，是否需要同步等待拷贝结束need_sync等。allocate方法和release方法的含义看代码上方的注释。

CPU内存分配器的实现

```
class CPUDeviceAllocator : public DeviceAllocator {
 public:
  explicit CPUDeviceAllocator();

```

```
    void* allocate(size_t size) const override;

    void release(void* ptr) const override;

    void memcpy(const void* src_ptr, void* dest_ptr, size_t size)
const override;
};
```

请同学们想一下，CPU上的内存资源分配无非是使用 `malloc` ，所以我们在 `allocate` 接口里封装的就是 `malloc` 方法，所以在 `release` 中封装的就是 `free` 方法，用于交还分配好的内存。

`GPU` 上的内存分配器我们也实现的，无非就是 `GPUDeviceAllocator` 的 `allocate` 方法封装了N卡中的 `cudamalloc` 用于申请显存，两类设备类上实现了同一个父类的接口方法。CudaAllocator子类在（https://github.com/zjhellofss/KuiperLLama/releases/）代码中的 `source/base/alloc_cu.cpp` 。

```
void* CPUDeviceAllocator::allocate(size_t byte_size) const {
  if (!byte_size) {
    return nullptr;
  }

  void* data = malloc(byte_size);
  return data;
}

void* CUDADeviceAllocator::allocate(size_t byte_size) const {
  if (!byte_size) {
    return nullptr;
  }
  void* ptr = nullptr;
  cudaError_t err = cudaMalloc(&ptr, byte_size);
  CHECK_EQ(err, cudaSuccess);
  return ptr;
}
```

Q: 当我们使用内存分配器得到的结果还是以指针形式返回的，那么在之后的过程中便存在一定的可能性使得使用 `Allocator` 申请得到的指针忘记释放，导致了内存泄露的问题，或者在释放置空后错误使用，引起程序奔溃的问题。

奔溃

王丹 2月2日 21:45

崩溃

对内存/显存的拷贝

对内存和显存的拷贝我们实现在DeviceAllocator这个父类中：

```
void DeviceAllocator::memcpy(const void* src_ptr, void* dest_ptr,
size_t byte_size,
                             MemcpyKind memcpy_kind, void*
stream, bool need_sync) const {
  CHECK_NE(src_ptr, nullptr);
  CHECK_NE(dest_ptr, nullptr);
  if (!byte_size) {
    return;
  }

  cudaStream_t stream_ = nullptr;
  if (stream) {
```

```
      stream_ = static_cast<CUstream_st*>(stream);
    }
    if (memcpy_kind == MemcpyKind::kMemcpyCPU2CPU) {
      std::memcpy(dest_ptr, src_ptr, byte_size);
    } else if (memcpy_kind == MemcpyKind::kMemcpyCPU2CUDA) {
      if (!stream_) {
        // 将位于src_ptr起始地址的cpu地址拷贝到dest_ptr显存中
        cudaMemcpy(dest_ptr, src_ptr, byte_size,
cudaMemcpyHostToDevice);
      } else {
      // 将位于src_ptr起始地址的cpu地址拷贝到dest_ptr显存中
        cudaMemcpyAsync(dest_ptr, src_ptr, byte_size,
cudaMemcpyHostToDevice, stream_);
      }
    }
    ...
    if (need_sync) {
      cudaDeviceSynchronize();
    }
  }
```

如果拷贝的方向是内存到内存，也就是memcpy_kind是CPU2CPU，我们将直接调用memcpy方法。如果是从CPU拷贝到显存，我们将根据是否需要异步拷贝，分别调用cudaMemcpy和cudaMemcpyAsync，如果需要在显存拷贝后进行同步（need_sync），我们可以额外调用cudaDeviceSynchronize()进行同步。

对内存的管理

为了解决A处说到的可能导致的内存泄露问题，我们设计了一个 `Buffer` 类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，以下是它的实现，具体代码见include/base/buffer.h 。

类内变量

1. `byte_size_` 这块内存的大小，以字节数作为单位。
2. `ptr_` 这块内存的地址，主要有两种来源， 一种是外部直接赋值得到的， Buffer不需要对它进行管理，和它的关系是借用，不负责它的生命周期管理，这种情况下对应下方 `use_external` 的值置为 `true` 。
3. 另外一种是需要Buffer对这块内存进行管理的，所以 `use_external` 值为false，**表示需要对它的生命周期进行管理，也就是没人使用该Buffer的时候会自动将 `ptr_` 指向的地址用对应类型的 `Allocator` 完成释放。**
4. `device_type_` ，表示Buffer中内存资源所属的设备类型
5. `allocator` Buffer对应设备类型的内存分配器，我们已经在上一节中说过，负责资源的释放、申请以及拷贝等，既可以是 `cpu allocator` 也可以是 `cuda allocator` .

```
namespace base {
class Buffer : public NoCopyable,
std::enable_shared_from_this<Buffer> {
 private:
  size_t byte_size_ = 0;
  void* ptr_ = nullptr;
  bool use_external_ = false; //是否拥有这块数据的所有权
  DeviceType device_type_ = DeviceType::kDeviceUnknown;
  std::shared_ptr<DeviceAllocator> allocator_;

```

```
    ...
    ...
};
```

我们结合 `Buffer` 类的构造和析构函数来看看 `use_external` 变量起到的作用。

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

从上面的代码可以看到，如果我们这里将 `use_external` 置为false，**表示当前Buffer拥有该内存**，表示这块资源需要Buffer进行管理，那么在Buffer对象释放的时候会调用对应 `allocator` 的释放方法，**自动释放这块内存。**

Allocator的释放函数

对于Cuda类型资源申请器的Release函数实现：

```
void CUDADeviceAllocator::release(void* ptr) const {
  if (ptr) {
    cudaFree(ptr);
  }
}
```

类内方法

1. 情况1，为Buffer**分配一块所属的内存**，使用内存分配器分配得到的内存指针是 `ptr_` ，并且将 `use_external` 置为false，表示在Buffer对象析构的时候也要将相关联的资源指针进行释放。

如果表示Buffer拥有当前的内存/显存，那么当我们在构造函数中指定use_external为false（表示当前Buffer拥有该对象）的时候，构造函数就自动调用allocate（上文说过的DeviceAllocator）方法完成资源的申请。

```
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;
  }
}

Buffer::Buffer(size_t byte_size, std::shared_ptr<DeviceAllocator>
allocator, void*      ptr, bool use_external)
```

```
16      : byte_size_(byte_size),
17        allocator_(allocator),
18        ptr_(ptr),
19        use_external_(use_external) {
20    if (!ptr_ && allocator_) {
21      device_type_ = allocator_->device_type();
22      use_external_ = false;
23      ptr_ = allocator_->allocate(byte_size);
24    }
25  }
```

2. 情况2，Buffer不申请资源，传入的指针指向并不属于它，因为不具有所属权，所以在退出的时候不需要对它释放。
3. 在情况2中，我们只需要使用Buffer buffer(32, nullptr, ptr, true)实例化，这表示我们将 `ptr` 指针赋值给Buffer，但是use_external=true变量表示buffer实例不需要负责去释放 `ptr` . **也不会去调用第5行之后的资源申请流程。**

```
1   Buffer::Buffer(size_t byte_size, std::shared_ptr<DeviceAllocator>
    allocator,
2                  void* ptr, bool use_external)
3       : byte_size_(byte_size), allocator_(allocator), ptr_(ptr),
    use_external_(use_external) {
4     if (!ptr_ && allocator_) {
5       device_type_ = allocator_->device_type();
6       use_external_ = false;
7       ptr_ = allocator_->allocate(byte_size);
8     }
9   }
```

3. 如果我们像1中说的那样，实例化的Buffer如果需要直接管理一片内存/显存，拥有所有权，则使用Buffer(32,allocator)，内存分配器会被赋值给该Buffer实例，等到构造函数内部调用buffer.allocate的时候会申请一块由该实例负责的内存区域，流程如下：
   a. auto buffer = Buffer(32, allocator);
   b. 随后在Buffer的构造函数中使用外部传入的allocator申请对应大小的显存/主存。
4. 这样一来就buffer就拥有了一块32字节大小的内存资源，等buffer被释放的时候 ，会连带着该内存资源一起释放。

动手练习

1. 请查看单元测试test/test_buffer.cpp，申请一块大小为32字节的内存资源。

```
1   TEST(test_buffer, allocate) {
2     using namespace base;
3     auto alloc = base::CPUDeviceAllocatorFactory::get_instance();
4     Buffer buffer(32, alloc);
5     CHECK_NE(buffer.ptr(), nullptr);
6   }
```

1. buffer在构造函数中调用 `alloc` 内存申请类申请了32个字节大小的内存资源，所以在 `buffer.ptr()` 返回内存的时候是非空的。
2. 请查看单元测试test/test_buffer.cpp，buffer不负责管理这块内存资源的情况。

```
TEST(test_buffer, use_external) {
  using namespace base;
  auto alloc = base::CPUDeviceAllocatorFactory::get_instance();
  float* ptr = new float[32];
  Buffer buffer(32, nullptr, ptr, true);
  CHECK_EQ(buffer.is_external(), true);
  delete[] ptr;
}
```

3. 查看Buffer的资源释放时机。allocate因为退出局部作用域后没有其他被引用的地方，所以它在局部花括号退出后就释放了，根据打印信息它会在HERE1和HERE2输出之间退出，而allocate2有外部引用，所以要等整个函数执行结束之后再退出。

```
TEST(test_buffer, allocate) {
  using namespace base;
  auto alloc = base::CPUDeviceAllocatorFactory::get_instance();
  {
    Buffer buffer(32, alloc);
    ASSERT_NE(buffer.ptr(), nullptr);
    LOG(INFO) << "HERE1";
  }
  LOG(INFO) << "HERE2";
}

TEST(test_buffer, allocate2) {
  using namespace base;
  auto alloc = base::CPUDeviceAllocatorFactory::get_instance();
  std::shared_ptr<Buffer> buffer;
  { buffer = std::make_shared<Buffer>(32, alloc); }
  LOG(INFO) << "HERE";
  ASSERT_NE(buffer->ptr(), nullptr);
}
```