第21次课程-用课程大模型推理框架跑LLama3和Qwen2.5

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

第21次课程.mp4

理论基础

这节课主要是面向 Demo（演示）的。在前面的课程已经完成所有算子的实现，以及张量、计算图等部分的设计之后，我们就要开始编写整体代码，以演示课程中大模型推理框架的整体推理功能。首先，我们来介绍一下从用户输入的 prompt 文本到最终整体输出的过程。

1. 假设用户输入 “How are you?”，那么大模型推理框架首先会把这个句子切分成多个词元组合。在这种情况下，输入句子会被拆分成 [How, are, you, ?] 共 4 个词元。**假设每个词元对应的向量长度为 512，那么编码后的词元向量就是 4×512。**

prompt阶段

2. 首先进入 prompt（提示）阶段，prompt 阶段会按逐个步骤（step）处理用户输入。**当 step 等于 1 时，我们将输入指定为词元向量矩阵的第一行，其维度为 1×512。**需要知道的是，prompt 阶段不会对产生的输出进行采样，而是直接选取原始输入中的词元作为输出。

   例如，现在步骤 step 等于 1，那么本轮的输出就直接从用户输入词元中选取，选择 are。既然 prompt 阶段的输出会被直接丢弃而不做任何处理，那我们为什么还要设置这个阶段呢？目的是为了构建自注意力模块中的 key（键）和 value（值）矩阵。

   自制大模型推理框架-第15次课程-KVCache的原理和实现

3. 当 step 等于 0 并且进入自注意力模块时，输入特征input乘以 Wk 矩阵会得到 K0，输入特征乘以 Wv 矩阵会得到 V0，随后将它们存放到 kv-cache 空间对应的位置中。**当 prompt 阶段结束时，kv-cache 中已经存储了（v0，v1，v2，v3..... 以及 k0，k1，k2，k3.....）**

以上就是prompt阶段的意义，主要是生成k,v向量用于在之后的generate阶段用于求出新单词和用户输入的promt阶段的单词的求出关联分数。

generate阶段

4. 在 prompt 阶段结束后进入 generate 阶段时，**generate阶段的输入 Input 乘以 wq 得到对应的 query 向量**，然后该 query 向量与 Key 矩阵相乘（Key 矩阵是由前面 prompt 的几个步骤中的 k0，k1，k2，k3 等拼接而成的），在 generate 步骤中，我们将得到的 query 与整个 Key 矩阵进行矩阵相乘，借此得到本步骤输入与之前 prompt 阶段句子中几个词元的关系，进而得到 scores 矩阵。

   简单而言，prompt 阶段虽无实际输出，但会记录下每个步骤中输入所产生的 key 和 value 向量。待下一个步骤(generate)的输入到来后，新输入得到的 query 向量会与前几个阶段的 key 和 value 向量相乘，得到 scores 矩阵。例如在自然语言处理任务的句子里，每个单词对应的 query 与其他单词对应的 key 相乘所得分数，反映出当前单词与句子中其他单词的关联程度。

如果scores(分数)较高，就表明它们在语义等方面存在较强的联系。之后，在此处得到的 scores 矩阵会通过将注意力分数与 value（值）矩阵做进一步计算（通常是加权求和操作），从而能够综合不同位置的特征信息，生成更符合上下文需求的输出表示。

5. 总的来说，在 prompt 阶段结束之后，便进入 generate 阶段逐个生成新的词向量。就本例的情况而言，下一个生成的单词可能是 “Im”，**然后我们再对这个新生成的单词进行编码以得到对应的嵌入向量，并将其拼接到 “How are you” 这几个单词对应的嵌入向量里，**接着再依据 “How are you? Im” 这几个词元对应的嵌入向量来生成下一个最有可能的单词。

Im --> 311

311查询得到1x512维度的向量

分段总结

在大模型推理的初始阶段，用户指定一句输入，比如在上例中，这句输入为 “How are you?”。根据嵌入式词表，我们会将这句输入转换为一个矩阵，其维度为 4×512。这里的 “4” 表示词元的个数，“512” 则是每个词元对应向量的维度。

在prompt 阶段，我们把这组矩阵输入到自注意力模块。自注意力模块中的 Wk 矩阵与输入相乘可得到 key 矩阵，Wv 权重矩阵与输入相乘可得到 value 矩阵。**接着，我们将 key 和 value 矩阵存放到 kv cache 中。**这样**在 generate 阶段**需要产生新词的时候，当前词元对应的输入特征与 wq 权重矩阵相乘得到 query，随后 query 可以与过往阶段的 key 矩阵进行矩阵乘法运算得到注意力分数 score，并对 value 矩阵进行加权处理。

模型加载流程

见项目demo路径下的main.cpp

```
int main(int argc, char* argv[]) {
  if (argc != 3) {
    LOG(INFO) << "Usage: ./demo checkpoint path tokenizer path";
    return -1;
  }
  const char* checkpoint_path = argv[1];  // e.g. out/model.bin
```

```
  const char* tokenizer_path = argv[2];

  model::LLama2Model model(base::TokenizerType::kEncodeSpe, tokenizer_path,
    checkpoint_path, false);
  auto init_status = model.init(base::DeviceType::kDeviceCUDA);
  if (!init_status) {
    LOG(FATAL) << "The model init failed, the error code is: " <<
init_status.get_err_code();
  }
```

在以上代码的第 2 至 7 行中，我们首先传入两个参数，分别是分词器的路径（记作tokenizer_path）和模型的路径（记作checkpoint_path），随后再用这两个变量去初始化对应的模型 model。

```
LLama2Model::LLama2Model(base::TokenizerType tokenizer_type, std::string
token_path,
                         std::string model_path, bool is_quant_model)
    : Model(tokenizer_type, base::ModelType::kModelTypeLLama2,
std::move(token_path),
            std::move(model_path), is_quant_model) {
 }
```

模型构造函数中的参数分别用于指定分词器的类型、分词器的路径、模型的路径以及是否为量化模型。随后便是对模型实例的初始化，**即调用 init 方法**。init方法中的步骤其实我们分开已经讲过了，这里我们将它连接起来再串讲一遍。**这里调用init方法的流程请结合代码一起学习，视频也是这样的。**

总体流程

> 请结合视频一起来阅读一下模型初始化的整体流程

```
base::Status LLama2Model::init(base::DeviceType device_type) {
  using namespace base;
  if (token_path_.empty()) {
    return error::PathNotValid(token_path_);
  }
  if (device_type == base::DeviceType::kDeviceCPU && is_quant_model_) {
    return error::InternalError("The cpu device do not support int8 quant
model.");
  }

  // 初始化设备
  device_type_ = device_type;
  if (device_type == DeviceType::kDeviceCUDA) {
    cudaSetDevice(0);
```

```
14        cuda_config_ = std::make_shared<kernel::CudaConfig>();
15        cudaStreamCreate(&cuda_config_->stream);
16        cudaError_t err = cudaGetLastError();
17        if (err != cudaSuccess) {
18          return error::InternalError("The cuda hanle create failed.");
19        }
20      }
21
22      // 映射模型权重
23      Status read_status = gen_model_from_file();
24      if (!read_status) {
25        return read_status;
26      }
27      init_mem(); // 为算子分配显存空间
28      if (device_type_ == base::DeviceType::kDeviceCPU) {
29        kernel::sin_cos_cache_calc_cpu(config_->head_size_, config_->seq_len_,
30
    get_buffer(ModelBufferType::kSinCache).ptr<float>(),
31
    get_buffer(ModelBufferType::kCosCache).ptr<float>());
32      } else {
33        CHECK_NE(cuda_config_, nullptr);
34        kernel::sin_cos_cache_calc_cu(config_->head_size_, config_->seq_len_,
35                                      get_buffer(ModelBufferType::kSinCache),
36                                      get_buffer(ModelBufferType::kCosCache),
    cuda_config_->stream);
37      }
38
39      sampler_ = std::make_unique<sampler::ArgmaxSampler>(device_type_);
40      return error::Success();
41    }
```

总体流程可以大致分为以下的几个部分：

1. 初始化cuda设备，绑定cuda流；
2. 加载模型权重文件并映射到进程内存空间中；
3. 创建算子，并从模型权重文件中加载权重；
4. 创建并绑定算子对应的输出显存空间，将算子中的权重加载到cuda中。

初始化设备

在模型的init方法中是对Cuda设备进行初始化，包括指定设备，创建Cuda流，这里的调用栈就是model.init()。

```
  device_type_ = device_type;
  if (device_type == DeviceType::kDeviceCUDA) {
    cudaSetDevice(0);
    cuda_config_ = std::make_shared<kernel::CudaConfig>();
    cudaStreamCreate(&cuda_config_->stream);
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
      return error::InternalError("The cuda hanle create failed.");
    }
  }
```

加载并映射模型

加载并映射模型的流程同样是通过init方法中调用的，**gen_model_from_file**中用mmap模型映射的方法打开模型文件，在gen_model_from_file方法中会做以下的几步，首先用mmap方法将打开后的模型文件映射到对应的进程空间，见gen_model_from_file中的第6行。这里的调用栈是model.init---> gen_model_from_file。

经过内存映射后的模型文件可以以内存偏移的方式进行访问。随后再利用访问到的权重数据依次创建算子，创建算子的流程调用见gen_model_from_file中的第11行。可以回顾 第10课-自制大模型推理框架-mmap映射模型文件

```
base::Status Model::gen_model_from_file() {
  using namespace base;
  config_ = std::make_unique<TransformerConfig>();

  // mmap
  auto mmap_status = read_model_file();
  if (!mmap_status) {
    LOG(ERROR) << "Handle model file " << model_path_ << " failed!";
    return mmap_status;
  }
  auto layer_create_status = create_layers();
  if (!layer_create_status) {
    LOG(ERROR) << "Create layers for the model file " << model_path_ << "
failed!";
    return layer_create_status;
  }

  return error::Success();
}
```

创建算子

在完成上一步的加载操作并把模型映射至进程空间以后，接下来便要着手初始化算子。此前的课程中已经对算子的初始化流程予以讲解，烦请大家自行回顾。其总体流程为：**依照 LLama 模型里的算子排列情况创建算子，接着索引到模型全量文件中相应的权重数据位置，利用这些权重来完成对模型的初始化。**当创建下一个算子时，再将权重的索引位置向后偏移。第11课-自制大模型推理框架-算子层的创建和权重的载入 到这一步，**调用栈为model.init-->gen_model_from_file-->create_layers。**

```
1   base::Status LLama2Model::create_layers() {
2     using namespace base;
3     if (!llama_layers_) {
4       llama_layers_ = std::make_unique<LLama2Layers>();
5     }
6
7     if (!is_quant_model_) {
8       create_param_layers();
9     } else {
10      create_param_quant_layers();
11    }
12    create_nonparam_layers();
13
14    if (!llama_layers_->embedding_layer_) {
15      return error::InternalError("Create the embedding layer for the llama
    model failed!");
16    }
17
18    if (llama_layers_->rmsnorm_layers_.size() != 2 * config_->layer_num_ + 1) {
19      return error::InternalError("Create the rmsnorm layers for the llama
    model failed!");
20    }
```

创建每个算子的输出显存空间

模型初始化方法 `init` 会调用 `init_mem` 方法，而 `init_mem` 方法会创建与算子相关的显存输出空间，以供后续使用。我们会依据类型将输出显存空间插入（`insert buffer`）到一个键值对集合中，**待算子执行需要时再进行取用。**这里的调用栈是 `model.init --> init_mem()`，调用见下方代码的28行。随后，**我们会在算子执行的时候将输出空间取出，随后会在算子的计算过程中使用。**

此外，`init_mem` 方法还会把上一步创建算子中的所有权重和输入输出张量同步到显存中，它使用的是 `to_cuda` 方法。`to_cuda` 方法会调用每个算子类同名的 `to_cuda` 方法，在这个方法中，会将该算子的输入和输出张量中的数据上传到显存空间中，**若存在权重，也会将权重张量同步到显存中，见下方代码的第16-18行。**

```cpp
void Layer::to_cuda() {
    for (auto& input : inputs_) {
        if (!input.is_empty()) {
            input.to_cuda(cuda_config_ ? cuda_config_->stream : nullptr);
        }
    }
    for (auto& output : outputs_) {
        if (!output.is_empty()) {
            output.to_cuda(cuda_config_ ? cuda_config_->stream : nullptr);
        }
    }
}

void LayerParam::to_cuda() {
    Layer::to_cuda();
    for (auto& weight : weights_) {
        weight.to_cuda(cuda_config_ ? cuda_config_->stream : nullptr);
    }
    if (!scales_.is_empty()) {
        scales_.to_cuda(cuda_config_ ? cuda_config_->stream : nullptr);
    }
}

void LLama2Model::init_mem() {

    if (device_type_ == base::DeviceType::kDeviceCUDA) {
        CHECK_NE(cuda_config_, nullptr);
        llama_layers_->to_cuda(cuda_config_);
    }

    CHECK(insert_buffer(ModelBufferType::kSinCache, sin_cache));
    CHECK(insert_buffer(ModelBufferType::kCosCache, cos_cache));

    CHECK(insert_buffer(ModelBufferType::kInputTokens, input_tokens));
    CHECK(insert_buffer(ModelBufferType::kInputEmbeddings, input_embeddings));
}
```

模型的Demo代码

在上一步中，我们已经加载好了一个模型，包括：

1. 初始化cuda设备，绑定cuda流；
2. 加载模型权重文件并映射到进程内存空间中；
3. 创建算子，并从模型权重文件中加载权重；
4. 创建并绑定算子对应的输出显存空间，将算子中的权重加载到cuda中。

等以上的几个步骤。

```
int32_t generate(const model::LLama2Model& model, const std::string& sentence, int total_steps,
                 bool need_output = false) {
  auto tokens = model.encode(sentence);
  int32_t prompt_len = tokens.size();
  LOG_IF(FATAL, tokens.empty()) << "The tokens is empty.";

  int32_t pos = 0;
  int32_t next = -1;
  bool is_prompt = true;
  const auto& prompt_embedding = model.embedding(tokens);
  tensor::Tensor pos_tensor = model.get_buffer(model::ModelBufferType::kInputPos);

  std::vector<int32_t> words;
  while (pos < total_steps) {
    pos_tensor.index<int32_t>(0) = pos;
    if (pos < prompt_len - 1) {
      tensor::Tensor input = model.fill_input(pos_tensor, prompt_embedding, is_prompt);
      model.predict(input, pos_tensor, is_prompt, next);
    } else {
      is_prompt = false;
      tokens = std::vector<int32_t>{next};
      const auto& token_embedding = model.embedding(tokens);
      tensor::Tensor input = model.fill_input(pos_tensor, token_embedding, is_prompt);
      model.predict(input, pos_tensor, is_prompt, next);
    }
    if (model.is_sentence_ending(next)) {
      break;
    }
    if (is_prompt) {
      next = tokens.at(pos + 1);
      words.push_back(next);
    } else {
      words.push_back(next);
    }

    pos += 1;
  }
  if (need_output) {
    printf("%s ", model.decode(words).data());
    fflush(stdout);
```

```
41      }
42      return std::min(pos, total_steps);
43  }
```

Demo可以分为以下的几个流程：

1. 把输入进行编码并转换为一组向量。sentence 是用户输入的提示词，首先将其切分为单词，进而转换为单词的编码数组 tokens，随后再把单词编码数组映射为向量prompt_embedding，即嵌入式词表中这几个单词所对应的向量；

```
1   auto tokens = model.encode(sentence);
2   int32_t prompt_len = tokens.size();
3   LOG_IF(FATAL, tokens.empty()) << "The tokens is empty.";
4
5   int32_t pos = 0;
6   int32_t next = -1;
7   bool is_prompt = true;
8   const auto& prompt_embedding = model.embedding(tokens);
```

2. 在prompt阶段，我们用输入词元对应的嵌入权重来填充模型的输入，**在fill_input方法中，我们根据当前自回归的步数(pos)选择填充哪一个单词的嵌入权重作为输入。**

```
1   while (pos < total_steps) {
2       pos_tensor.index<int32_t>(0) = pos;
3       if (pos < prompt_len - 1) {
4           // 通过pos来选择当前阶段填充哪个单词的嵌入权重作为输入
5           tensor::Tensor input = model.fill_input(pos_tensor, prompt_embedding,
    is_prompt);
6           model.predict(input, pos_tensor, is_prompt, next);
7       }
```

3. 在generate阶段同理，我们将上一个阶段模型预测单词的词元编码为嵌入权重作为输入。

```
1   is_prompt = false;
2   tokens = std::vector<int32_t>{next};
3   // 将上一步预测的单词编码为权重
4   const auto& token_embedding = model.embedding(tokens);
5   // 填充
6   tensor::Tensor input = model.fill_input(pos_tensor, token_embedding,
    is_prompt);
7   model.predict(input, pos_tensor, is_prompt, next);
```

4. 在步骤 2 和 3 中，**新一轮的输出预测都放置在 next变量中并返回，之后我们需要将这里的输出记录下来**，以供后续编码使用。在此过程中，还需要判断输出的next是否为句子结束的标记词，如果是，则退出；另外一种退出情况是自循环的次数达到了一定的轮数。

```
if (model.is_sentence_ending(next)) {
    break;
}
if (is_prompt) {
    next = tokens.at(pos + 1);
    words.push_back(next);
} else {
    words.push_back(next);
}

pos += 1;
```

5. 最后，我们来看看fill_input方法的实现，了解怎么通过当前的迭代轮数和输入的编码向量来对模型进行赋值：

```
tensor::Tensor LLama2Model::fill_input(const tensor::Tensor& pos_tensor,
                                       const op::EmbeddingOutput&
embedding_output,
                                       bool is_prompt) const {
  const int32_t pos = pos_tensor.index<int32_t>(0);
  auto [input_tokens, input_embeddings, input_token_num] = embedding_output;

  int32_t index = 0;
  if (is_prompt) {
    index = pos;
  }
  std::shared_ptr<base::Buffer> input_emb_buffer =
      std::make_shared<base::Buffer>(config_->dim_ * sizeof(float), nullptr,
                                     input_embeddings.ptr<float>(index *
config_->dim_), true);

  tensor::Tensor input(base::DataType::kDataTypeFp32, config_->dim_);
  input.assign(input_emb_buffer);
  input.set_device_type(device_type_);
  return input;
}
```

我们知道，每个单词向量的维度是 dim。如果要索引到第 index 个单词的向量，就需要用 index×dim 进行索引，就像 fill_input 第 11 到 12 行所示的那样。

我们根据**迭代的轮数**指定当前的嵌入权重的偏移，为 `index×config->dim`，随后再将它赋值给一个输入张量 input。但是如果是 generate 阶段，这里的 index 一般为 0，因为每次只处理一个单词。如果是 prefill 阶段，因为用户输入是一长串的，所以我们每次迭代只处理其中之一，**所以 index 为当前迭代的轮数 pos。**

Demo演示

> 在这里我要提醒大家，需注意正确选择编解码算法，同时要确保量化模型选项也被正确选择。

qwen2.5演示

1. 首先需要从hugging face上下载模型

```
export HF_ENDPOINT=https://hf-mirror.com
pip3 install huggingface-cli
huggingface-cli download --resume-download Qwen/Qwen2.5-0.5B --local-dir Qwen/Qwen2.5-0.5B --local-dir-use-symlinks False
```

2. 导出模型

```
python3 tools/export_qwen2.py Qwen2.5-0.5B.bin --hf=Qwen/Qwen2.5-0.5B
```

导出后得到的模型为Qwen2.5-0.5B.bin

3. 支持Qwen2.5的编译方式

```
mkdir build
cd build
# 开启 USE_CPM 选项，自动下载第三方依赖，前提是需要网络畅通
cmake -DUSE_CPM=ON -DQWEN2_SUPPORT=ON .
make -j16
```

这里需要注意，如果要手动安装 re2 和 abseil 库，需要以动态的形式对它们进行编译安装。

4. 启动demo

```
/home/fss/code/KuiperLLama/cmake-build-release-system/demo/qwen_infer Qwen2.5-0.5B.bin Qwen/Qwen2.5-0.5B/tokenizer.json
Generating...
今天，我们来聊聊“如何在职场上保持积极心态”。
首先，我们需要认识到，职场上的人际关系是复杂多样的，每个人都有自己的性格特点和价值观。因此，保持积极心态需要我们学会适应不同的文化背景和工作环境，同时也要学会尊重他人的观点和意见。
其次，我们需要学会积极地面对挑战和困难。在职场上，我们经常会遇到各种各样的问题和挑战，这些挑战可能来自于工作、人际关系、家庭等方面。因此，我们需要学会积极地面对这些挑战，积极地寻找解决问题的方法和途径。
最后，我们需要学会保持乐观的心态。在职场上
steps:128

duration:1.170372

steps/s:109.366899
```

llama3.2演示

同上