lightllm代码解读——之模型推理

robindu 关注

LLM（大模型）技术的发展日新月异，不停有新东西涌现。LLM的推理部署技术也吸引了很多关注，TGI/vLLM/lightLLM相对来说是这部分受到比较多关注的工作，在LLM部署上能够更加高效地利用GPU的计算能力，平衡访存与计算。lightLLM作为一个纯python的推理框架，在整体服务流程与模型推理中几乎全部在python中完成撰写，代码结构清晰，相对也更易于二次开发。本文希望对在lightLLM官方文档的基础上做进一步的代码解读，帮助大家了解其设计理念。大家在阅读本文前也可以先阅读lightllm特性对框架整体做大致了解。

一、概览

在lightLLM的README中有介绍框架的主要features，其中与模型推理强相关的有：Nopad、FlashAttention、Tensor Parallel、Int8KV Cache、Token Attention、Dynamic Batch。其中Token Attention与Dynamic Batch又与模型推理部分之外的调度部分相关。本文主要介绍模型推理相关的代码结构，前4个features会在其中涉及到。

模型相关的介绍可以参考官方文档“docs/AddNewModel_CN.md”，这个文档以bloom为例介绍如何增加一个自定义的模型backend，开头列出了模型相关的代码结构设计（lightllm/common/basemodel）。将一个模型的相关代码拆分成layer_weights、layer_infer、triton_kernel三个部分。

layer_weights主要负责加载各个模型的参数文件，将命令空间⁺统一到框架中。layer_infer则负责接收输入和模型参数，完成每一层的运算。triton_kernel使用openai⁺的triton计算库，用pythonic的语法实现底层cuda kernel功能。

basemodel目录下主要设计了整体结构，但layer_infer和layer_weights中基类方法的具体实现并不涉及。triton_kernel下实现了基本的矩阵乘法⁺（包含int8、int4功能）。

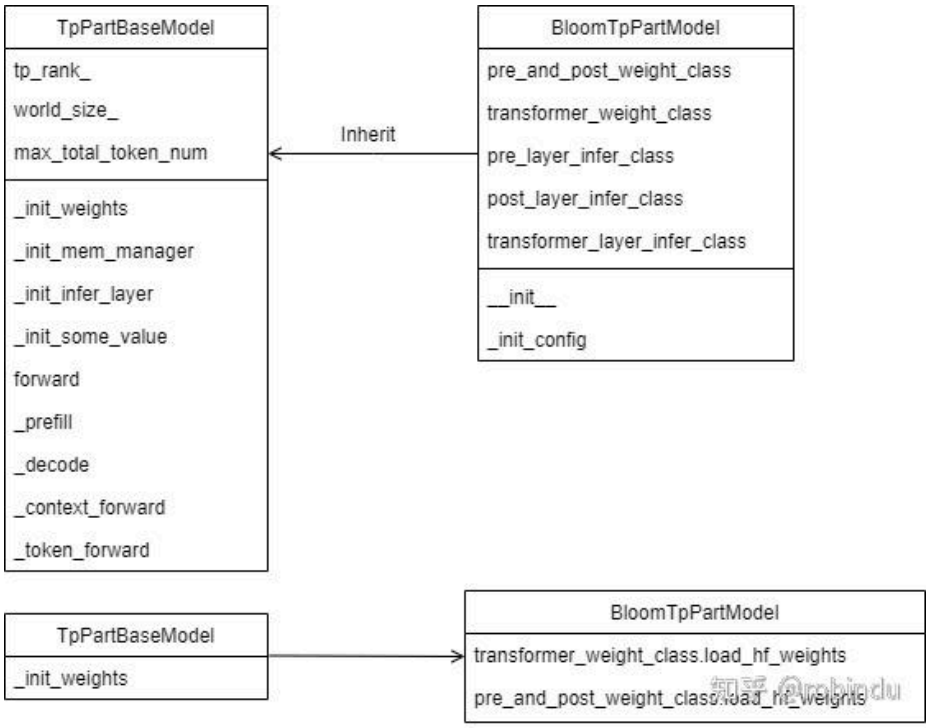
```
├── basemodel.py    # 模型框架类
├── infer_struct.py # 推理用的状态类
├── __init__.py
├── layer_infer # 推理层的基类实现
│   ├── base_layer_infer.py
│   ├── __init__.py
│   ├── post_layer_infer.py
│   ├── pre_layer_infer.py
│   ├── template # 推理层的模板实现，继承实现模板可以减少开发量和重复代码
│   │   ├── __init__.py
│   │   ├── post_layer_infer_template.py
│   │   ├── pre_layer_infer_template.py
│   │   └── transformer_layer_infer_template.py
│   └── transformer_layer_infer.py
├── layer_weights # 权重基类的实现
│   ├── base_layer_weight.py
│   ├── hf_load_utils.py
│   ├── __init__.py
│   ├── pre_and_post_layer_weight.py
│   └── transformer_layer_weight.py
└── triton_kernel # 一些公共使用的 triton kernel 算子
    ├── apply_penalty.py
    ├── destindex_copy_kv.py
    ├── quantize_gemm_int8.py
    ├── dequantize_gemm_int8.py
    ├── dequantize_gemm_int4.py
    └── __init__.py
```

二、模型推理代码

"lightllm/models"中是框架支持的模型列表，及每个模型的推理核心代码实现。这里以bloom模型为例一探究竟。

模型基类解读

首先看模型的主题类，bloom中的模型主题类继承至TpPartBaseModel基类，bloom中的代码主要给基类中的一些方法赋以实际值，如“pre_and_post_weight_class = BloomPreAndPostLayerWeight”。总体的调度逻辑在TpPartBaseModel基类中。



模型参数初始化

模型首先从初始化开始，第一步就是从huggingface格式的模型文件中读取参数，统一命名空间后，一一拷贝给bloom模型实例。该部分的最核心代码应该是bloom目录下BloomPreAndPostLayerWeight、BloomTransformerLayerWeight中load_hf_weights实现，这里的代码会真正创建出模型的参数实体。如“self.wte_weight_ = self._cuda(weights["word_embeddings.weight"][split_vob_size * self.tp_rank_: split_vob_size * (self.tp_rank_ + 1), :])”中，会将hf weights中embedding的权重载入，并赋值给self.wte_weight_。

同时在一些llm中该部分的参数量会相对较大，需要分拆到多张GPU上，相对应的参数分拆也会在这一步完成。既然代码中有tensor parallel相关的实现，并在初始化时就需要考虑，那我们也要看一下模型是如何在多卡上完成初始化的。这部分可以参考test/model/model_infer.py中的test_model_inference函数，初始化时就是通过for循环+进程来完成拆分后的多个子模型初始化。

```
# test/model/model_infer.py
def test_model_inference(world_size, model_dir, model_class, batch_size, input_len, ou
    ans_queue = Queue()
    workers = []
    for rank_id in range(world_size):
        proc = multiprocessing.Process(target=tppart_model_infer, args=(rank_id, world
        proc.start()
        workers.append(proc)

    for proc in workers:
        proc.join()

    assert not ans_queue.empty()
```

```
    while not ans_queue.empty():
        assert ans_queue.get()
    return
```

在模型结构上，lightllm在实现时将模型按照transformer layer和pre_and_post layer拆分，其中pre_and_post layer包括word embedding、输出层等，transformer layer则包括层中的自注意力、FFN、LN等。

模型前向运算

模型前向的实现则主要都在各模型下的layer_infer目录，bloom模型包括三个核心的layer_infer脚本，分别对应三种层。

```
├── layer_infer # 推理层的实现
│   ├── __init__.py
│   ├── post_layer_infer.py
│   ├── pre_layer_infer.py
│   └── transformer_layer_infer.py
```

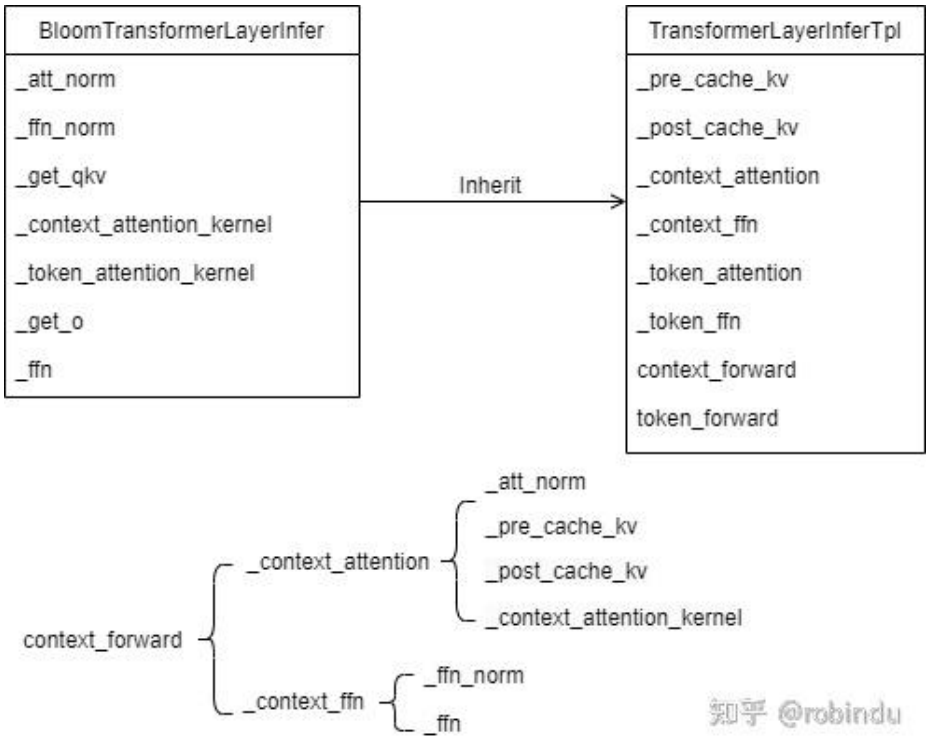
其中每一种层的前向有两个核心函数context_forward和token_forward，在模型类的forward方法中，根据is_prefill标志位决定使用contextcontext_forward和token_forward。阅读代码后可知context_forward会使用上下文context一起做一次前向，而token_forward则使用逐一token追加的方式逐次调用，这在test/model_model_infer.py中可以窥探得知，在使用token_forward时使用循环调用，标记seq起始index的参数会逐步加1。这在实际场景中应该对应于第一次模型调用时能一次性得到用户的prompt+问题，这部分直接可见到多个token，使用context_forward，然后接下来每次预测一个新的token，使用token_forward循环完成一次次调用。

PreLayer

在lightllm/models/bloom/layer_infer/pre_layer_infer.py中也可以简单的看到forward相关计算功能的实现。在pre_layer中主要完成两个计算，一个是word_embedding计算，一个是layernorm计算。embedding部分直接使用pytorch的内置函数实现input_embdings = torch.embedding(layer_weight.wte_weight_, tmp_input_ids, padding_idx=-1)，涉及到的多卡拆分使用dist.all_reduce完成。至于layernorm的计算则使用triton kenrel完成实现更高效的计算。

TransformerLayer

更重要且复杂的部分在transformer layer，这里按照算法组件拆分成attention和ffn两个子模块。这一层的实现在基类与子类中均有重要的实现，如下图所示。



在基类TransformerLayerInferTpl中，主要实现的功能有：组合attention和ffn功能模块按序计算、并管理kv缓存。其中基类中管理kv缓存的功能是比较重要的，因为LLM在自回归推理时为避免历史信息重复计算，会将自注意力机制[+]中的k/v历史值缓存起来。基类中通过_pre_cache_kv与_post_cache_kv来进行attention计算前kv值的获取与新kv值的缓存。具体如何高效管理放在后续章节讲解。

PostLayer

postlayer的实现主要在lightllm/models/bloom/layer_infer/post_layer_infer.py中，值得注意的是因为postlayer是做输出预测的层，这里只有token_forward实现，没有context_forward实现，因为用户提交的文本部分不需要对应预测只，预测是只针对answer部分的。

TritonKernel

此前介绍的代码主要是构建框架结构，具体实现功能包含内存管理、基础矩阵[+]乘操作（torch.addmm等实现），但一些复算子集合如attention和layernorm的具体计算实现是不涉及的。这部分的算子可以通过计算顺序重规划、算子融合[+]提升访存与运算效率，这部分的实现主要使用openai triton库实现。

在bloom目录下实现了attention和layernorm相关的代码，我们先看一下layernorm.py中是如何实现的。

```
├── triton_kernel # 推理层的实现
│   ├── __init__.py
│   ├── layernorm.py
│   ├── token_flashattention_nopad.py
│   ├── context_flashattention_nopad.py
│   └── ...
```

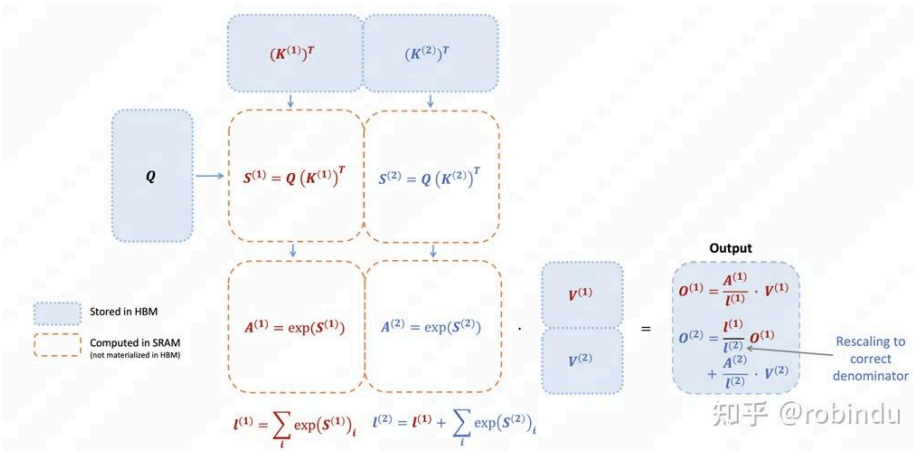
LayerNorm

这里对layernorm的实现与triton官网上的实现是一样的。做layernorm计算时，首先要沿着C维度计算mean和var，然后减均值除方差。这里代码上分为三段，理解了一段另外两段就自然也理解了。如果将输入视作一个二维矩阵，举例一个（BT，C）维度的输入矩阵会被划分到<grid, block_size>形状的硬件资源上做并行计算。每次计算一行，对应于X += row * stride每次会取第row行的起始index，每行共有N个元素，这N个元素每次可以由BLOCK_SIZE个线程并行处理，所以每行需要range(0, N, BLOCK_SIZE)个循环来完成。

```
row = tl.program_id(0)
Y += row * stride
X += row * stride
# Compute mean
mean = 0
_mean = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
for off in range(0, N, BLOCK_SIZE): # N > BLOCK_SIZE
  cols = off + tl.arange(0, BLOCK_SIZE)
  a = tl.load(X + cols, mask=cols < N, other=0.).to(tl.float32)
  _mean += a
  mean = tl.sum(_mean, axis=0) / N
```

ContextFlashAttention

ContextFlashAttention的计算复杂，对应的核心代码有几十行。对应于下图的分块计算流程，核心思想是q@kT，以及softmax(q@kT)的中间结果不写到HBM中，减少内存访问次数。该部分的代码解读后续看是否单独开一章节，nopad的特性也是在这部分的triton kernel中体现，在load数据与计算时，可以通过mask特性规避无效计算。



三、总结

对着LLM的发展，新的工程技术也在突飞猛进。而Nvidia的代表工具栈tensort在Transformer模型大放异彩的这段时间里反而迭代地不够迅速，而AI编译器等技术的发展衍生出了新的可行路线。这从lightllm库中可见一二，其中模型推理的相关代码完全舍弃了onnx，tensorrt+等开源库。转而使用pytorch来做内存、算子的组合管理，使用triton来完成底层kernel的实现，使用huggingface transformers的模型仓库资源加载模型权重。

随着LLM对算力的巨大要求，Nvidia凭借着强大的软硬件优势也取得了高速的发展。但也可以看到在软件层面也有一些新型方案在快速发展，以python动态思想为核心的方式，在简洁度、代码易编写、灵活性等层面正在获得更大的优势。

四、参考文献

[1] LightLLM：纯Python超轻量高性能LLM推理框架 | AiBard123| ai工具网址导航,ai最新产品

[2] github.com/ModelTC/ligh...

[3] github.com/ModelTC/ligh...

[4] tridao.me/publications/...

其他系列文章链接

第一弹：lightllm代码解读——模型推理

第二弹：lightllm代码解读番外篇——triton kernel撰写