知乎

lightllm代码解读——显存管理机制

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，LightLLM是其中受到比较多关注的一个。此前的文章中已经讲解了LightLLM的大多数特性，如模型推理的Nopad、Dynamic Batch、Token Attention等，但其中KV Cache的显存管理并没有详细的解读。因为LLM的大参数与流式部署特性，使得其对显存的耗费非常的大，如果不能高效地管理显存，也会导致整个计算系统的效率严重下滑。

一、概览

在LLM场景中模型的参数量非常的庞大，将模型的参数全部装载到GPU中已经会占据非常大的显存，如Fp16精度下一个7B的模型参数占据14GB显存，而一张Nvidia A10显卡的内存大小为24GB。这还仅仅是加载参数部分，模型在实际运算过程中还会对显存有额外的需求，这部分也是很大的比例。显存问题不仅仅只影响模型是否能运行起来，而且影响到模型运行的效率，计算情况下甚至会差别十倍百倍。

由于GPU是一种适合并行密集运算的芯片，使用中组batch等提升并行度的方法能提高GPU的利用率。组batch[+]在LLM场景中仍然是非常重要的技巧，甚至比小模型更为重要，这是因为如果使用批处理的方式可以在加载一次模型参数的情况下完成更多的运算，对访存带宽的利用率也会更高效。所以高效地利用显存，能更好地提升运算并行性，也能更好地平衡访存/计算，以提升机器利用率。

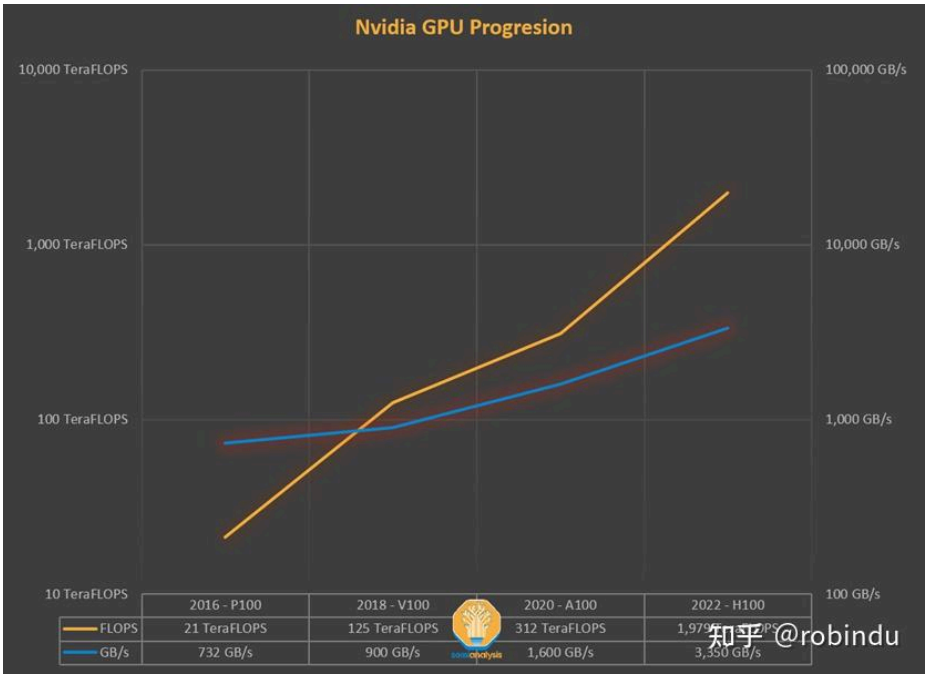

图1 16-20年GPU 计算与访存速度变化趋势

二、KV Cache

对于LLM中自注意力机制大家一般都有一定的熟悉度。Transformer模型在设计时使得每个token的embedding能够与其他所有token相互可见，并生成注意力矩阵[+]。LLM中引入casual特性，即使后面的token能见到前面的token，而前面的见不到后面的，在attention矩阵[+]中也即是对角阵的形式。具体的计算过程如下图所示。

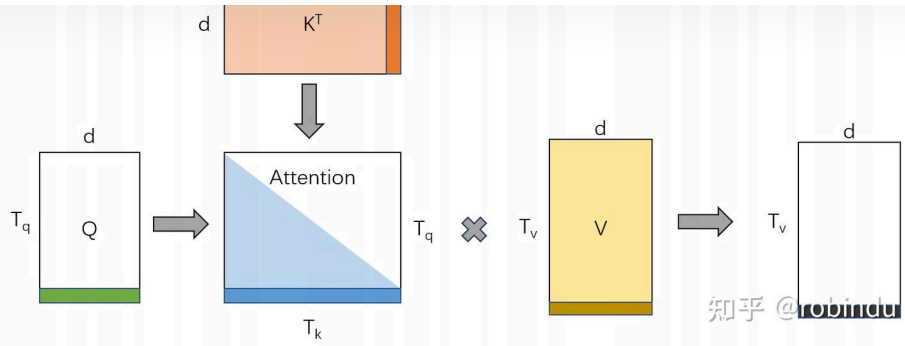

图2 LLM自注意力机制计算过程

在训练过程中因为所有的token是一次性载入的，所以只要在attention矩阵上蒙上一层mask遮蔽一般即可，但是在实际推理过程中，token是逐个生成的。当第n个token生成后，在自注意力层中参与计算的数据都随着新生成的token扩充尾部数数据。可以仔细观察这部分的计算流程，KV中的历史数据（对应于前n-1个token）是需要参与计算的，且该部分历史数据在后续的运算中是一直保持不变的。Q中只有对应最新token的数据才参与计算。所以为了避免重复计算KV历史部分应该被缓存在显存中，避免重复计算，也就是常说的KV Cache。

三、缓存管理策略

KV Cache对于每一个token所需要维护的数据量为n_head*head_dim*n_layer，对于一个13B的模型，每个token需要消耗1MB空间，如果同时对接上百路、token长度上千的并发请求，则该部分对GPU显存来说是一个很大的负担。那我们需要对这部分的显存需求做怎样的管理呢？试想一下，如果我们按照一般做法实现的话，对应的KV缓存在每路对应的线程里大概如下这样。

```
k_cache = torch.empty((num_tokens, n_head, head_dim, n_layer))
new_data = torch.empty((1, n_head, head_dim, n_layer))
k_cache = torch.cat((k_cache, new_data), dim=0)
# after computing done
del k_cache
```

当该路请求结束时，则对应的kv cache空间会被系统销毁回收。当一开始处理的请求不多时不会出现太大的问题，但当请求不同的新增销毁，该部分占用的庞大空间也会被频繁的申请然后销毁，这会产生许多许多内存碎片。所以像vLLM中提出的PagedAttention以及lightLLM中内存管理，都可以规避这样的碎片。核心理念就是先期申请一块大的内存空间，然后每次有内存需求时，会从中选取可用的部分使用，这样的话即使后期内存出现不连续的情况，kv cache也能高效地利用这些不连续的空间。而lightLLM的token attention概念使得在具体实现时更加容易。

更加具体的设计思路在官方文档LightLLM.md中也有提到，分为4个步骤：1）在模型初始化阶段，就根据设置的最大token数目计算出对应缓存空间，然后申请kv_buffer。这块空间在模型生命中期中是一直存在的，而其中哪里是使用过的则由另外建立的一个索引mem_state来标识；2）当有一路新请求时，会先检查kv_buffer中是否有可用的空间，然后为其分配；3）新token对应的缓存写到没被使用的空间上；4）请求完成后释放。

```
# 记录buffer空间使用状态的相关标识，size=max_total_token_num
self.mem_state = torch.ones((size,), dtype=torch.bool, device="cuda")    # 记录buffer使
self._mem_cum_sum = torch.empty((size,), dtype=torch.int32, device="cuda") # 是mem_st
self.indexes = torch.arange(0, size, dtype=torch.long, device="cuda")    # 用于选取索引的
self.can_use_mem_size = size     # 记录剩余空间

# 分配空间过程
torch.cumsum(self.mem_state, dim=0, dtype=torch.int32, out=self._mem_cum_sum)
select_index = torch.logical_and(self._mem_cum_sum <= need_size, self.mem_state == 1)
select_index = self.indexes[select_index]        # 换算为index下标
self.mem_state[select_index] = 0         # 更新占用状态
self.can_use_mem_size -= len(select_index)       # 更新剩余空间
```

介绍了缓存管理的策略，接下来让我们来深入代码看看具体是如何实现的。这里仍旧以bloom的实现为例分析，涉及到的核心代码主要在以下文件中。因为bloom的模型结构相对比较贴近标准transformer，所以一些实现直接继承模板类的实现。

```
lightllm/common/mem_manager.py  # 管理kv cache的全生命周期空间
lightllm/common/basemodel/layer_infer/basemodel.py      # 初始化mem_manger，管理一些基础
lightllm/common/basemodel/layer_infer/template/transformer_layer_infer_template.py
```

在mem_manger.py中实现的就是第三节中介绍的内容，初始化的buffer申请如下。可以看到根据层数申请最大token数目对应的缓存空间。另外值得注意的是第三节中分配空间代码分配的不一定是连续空间，在mem_manger.py中有分配连续空间的代码alloc_contiguous。

```
key_buffer = [torch.empty((size, head_num, head_dim), dtype=dtype, device="cuda") for
```

另外要看的是在base_model.py中，该部分我们重点看两个函数_init_mem_manager和_decode。在_init_mem_manager函数中特别简单，传入模型/系统能适配的最大token数，与一些模型基础信息即可，在内存管理器中会分配相对应的kv cache空间，也就是2*层数*token数*head_dim*n_head。

```
# lightllm/common/basemodel/basemodel.py
def _init_mem_manager(self):
  assert self.config["num_attention_heads"] % self.world_size_ == 0
  self.mem_manager = MemoryManager(self.max_total_token_num,
                                   dtype=torch.float16,
                                   head_num=self.config["num_attention_heads"] // self
                                   head_dim=self.config["n_embed"] // self.config["num
                                   layer_num=self.config["n_layer"])
  return
```

在_decode函数中一开始为模型运行阶段申请一个infer_state实例，用于汇总当次运行时所需的信息。值得注意的是将mem_manger也作为这个运行时临时变量的一个类内值，经过上述描述我们知道一个请求生命周期中产生的kv cache都存储在这个mem_manger管理的大块显存空间中。让我们暂时抛开代码，来直接思考_decode函数中需要完成那些操作呢?

这里涉及到kvcache应该分以下几步：1）将mem_manager中对应于当前batch的历史kv cache取出，或者找到index；2）向mem_manager申请当前需要的空间，应该是batchsize个token对应的空间；3）利用当前的token计算出新的cache_k和cache_v，然后写回到mem_manger中。

理清楚步骤后，我们在来结合实际代码看看是如何实现的。首先找到历史index，这部分在函数调用时由入参（b_loc, b_start_loc, b_seq_len）指定。然后申请出新的所需空间，空间位置记录在infer_state中。

```
# lightllm/common/basemodel/basemodel.py
infer_state = self.infer_state_class()
...
infer_state.mem_manager = self.mem_manager
# 为此次模型前向申请新的cache空间，因为n_head、had_dim、layer都是固定的，所以api都以token数目
# 因为是one token by token的调用，当次调用就是batch_size个token
# 返回的是申请空间的index，起始index和终止index
alloc_mem = self.mem_manager.alloc_contiguous(batch_size)
if alloc_mem is not None:
  infer_state.decode_is_contiguous = True
  infer_state.decode_mem_index = alloc_mem[0]
  infer_state.decode_mem_start = alloc_mem[1]
  infer_state.decode_mem_end = alloc_mem[2]
  # 这一句很重要，向上层追溯可知，这个变量是维护在model_rpc.py中一个InferBatch类型的变量中，记
  # 这里的decode_mem_index是一个batch_size长度的list。(b_loc是一个batch_size，max_seq_len
```

```
        # 处理无连续空间时的问题
        ....
```

然后代码在执行自注意力层时部分重要代码转向之前在模型推理部分提到的基类TransformerLayerInferTpl的方法，以_token_attention为例：

```
# lightllm/common/basemodel/layer_inder/template/transformer_layer_infer_template.py
def _token_attention(self, input_embding, infer_state: InferStateInfo, layer_weight):
  input1 = self._att_norm(input_embding, infer_state, layer_weight)
  # 将对应当前token的cache_k, cache_v空间明确取出，也就是上面一段代码alloc出的空间，方便进行下
  cache_k, cache_v = self._pre_cache_kv(infer_state, layer_weight)
  q = self._get_qkv(input1, cache_k, cache_v, infer_state, layer_weight)
  input1 = None
  # 将经过矩阵映射的qkv再全部在mem_manager中梳理清楚。主要应对此前申请不到连续空间的场景
  self._post_cache_kv(cache_k, cache_v, infer_state, layer_weight)
  o = self._token_attention_kernel(q, infer_state, layer_weight)
  ...
  return

# 之后就会转到bloom模型下自己的核心triton kernel
# lightllm/models/bloom/layer_infer/transformer_layer_infer.py
# 下面的调用就对应于图2所示的计算过程
def _token_attention_kernel(self, q, infer_state, layer_weight):
  o_tensor = torch.empty_like(q)
  token_attention_fwd(q.view(-1, self.tp_q_head_num_, self.head_dim_),
                      infer_state.mem_manager.key_buffer[self.layer_num_],
                      infer_state.mem_manager.value_buffer[self.layer_num_],
                      o_tensor.view(-1, self.tp_q_head_num_, self.head_dim_),
                      layer_weight.tp_alibi,
                      infer_state.b_loc,
                      infer_state.b_start_loc,
                      infer_state.b_seq_len,
                      infer_state.max_len_in_batch)
  return o_tensor
```

五、总结

官方文档LightLLM.md中已经阐释了如何管理kv cache的缓存原理，但是并没有介绍其如何融入到整个系统代码中。在整个系统中mem_manager部分的原理和实现都是非常清晰的，如果不考虑到动态batch、router等，其实融入模型推理也是简单的。但涉及到router与动态batch后，还要考虑到如何维护全生命周期的历史状态，以及如何不混乱，这部分的代码逻辑相对是复杂的，在阅读代码时需要耐心地自顶向下或者自底向上层层剥茧。

参考文献

[1] semianalysis.com/p/nvid...

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

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

第三弹：lightllm代码解读——Router