llama3-GQA&KVCache

llama3和llama2一样，使用了GQA以及KVCahce，来节省显存以及加速模型推理

由于使用了GQA，虽然每组多个query共享同一对KV，但是在计算时还是需要repeat KV来进行矩阵运算，因此定义函数repeat_kv如下：

repeat_kv:

```
def repeat_kv(x: torch.Tensor, n_rep: int) -> torch.Tensor:
    """torch.repeat_interleave(x, dim=2, repeats=n_rep)"""#等同于torch.repeat_interleave函数指定输入x的某一个维度，对他扩展repeats份。而torch.repeat函数是对整体进行repeat
    #对输入x的维度进行拆解
    bs, slen, n_kv_heads, head_dim = x.shape
    if n_rep == 1:#如果复制次数等于1则直接返回x
        return x
    return (#如果n_rep>1，要去对n_kv_heads进行repeat
        #先要再第四维进行扩维
        #x[:, :, :, None, :]，在某个位置加个none，表示在这一维进行扩维,维度大小为1，总的元素数不变
        x[:, :, :, None, :]
        #对扩维之后的x进行扩展，把原本第四维的1扩展成n_rep
        .expand(bs, slen, n_kv_heads, n_rep, head_dim)
        #然后在缩维，从五维缩成四维，第三维维度大小是n_kv_heads * n_rep，所以整个函数整体效果就是把kv的头数扩展了n_rep份
        .reshape(bs, slen, n_kv_heads * n_rep, head_dim)
    )
```

llama3的GQA和KVcache的使用体现在Attention类中：

在使用之前需要定义一些超参数：

class ModelArgs:

```
class ModelArgs:#模型的各种参数
    dim: int = 4096#模型维度
    n_layers: int = 32#层数
    n_heads: int = 32#总query头数
    n_kv_heads: Optional[int] = None #总kv头数
    vocab_size: int = -1  # 词典大小
    multiple_of: int = 256  #模型维度的倍数，默认为256，这可能用于确保模型的某些参数可以被这个数整除，以便于某些硬件优化。
    ffn_dim_multiplier: Optional[float] = None#前馈网络（Feed Forward Network）的维度乘数，默认为None，如果设置了这个值，它将用于计算前馈网络的维度。
    norm_eps: float = 1e-5#归一化时使用的epsilon值，默认为1e-5，用于防止除以零。

    max_batch_size: int = 32#最大batch_size
    max_seq_len: int = 2048#最长seq_len
```

class Attention:

```
class Attention(nn.Module):
    """GQA module."""
    def __init__(self, args: ModelArgs):#传入提前定义好的模型超参数
        super().__init__()
        #如果args中声明了kvheads则将其赋值给self.n_kv_heads（MQA或GQA），如果没有声明那么就让self.n_kv_heads等于args.n_heads（也就是MHA）
        #GQA和MQA可以减小KVcache中存储的数据量
        self.n_kv_heads = args.n_heads if args.n_kv_heads is None else args.n_kv_heads

        #模型太大则放到多张卡上存储,fs_init.get_model_parallel_world_size()返回当前分布式训练中参与训练的进程数量
        model_parallel_size = fs_init.get_model_parallel_world_size()
        #将头分到多张卡上，n_local_heads为每张卡上的query头数，实现并行化
        self.n_local_heads = args.n_heads // model_parallel_size
        #每张卡上的kv头数
        self.n_local_kv_heads = self.n_kv_heads // model_parallel_size

        #如果self.n_local_heads和self.n_local_kv_heads不相等，要把self.n_local_kv_heads复制self.n_rep份
        self.n_rep = self.n_local_heads // self.n_local_kv_heads
        #每个头的dim=模型维度/头数
        self.head_dim = args.dim // args.n_heads

        #定义可训练参数wq，wk,wv,wo，ColumnParallelLinear 是一种并行线性层，主要用于模型并行处理中。它的主要特点是将权重矩阵按列分割成多个子矩阵，并在不同的并行设备上进行计算
        self.wq = ColumnParallelLinear(
            args.dim,
            args.n_heads * self.head_dim,#从dim维映射到args.n_heads * self.head_dim维
            bias=False,
            #这通常用于模型并行的场景，其中每个设备处理一部分数据，并且最终结果不需要在单个设备上汇总。这样可以减少跨设备通信的开销，特别是在分布式训练或大规模并行计算中
            gather_output=False,#在使用模型并行时，gather_output 参数通常用于控制是否需要将分布在不同设备上计算的结果收集到一个设备上。
            init_method=lambda x: x,#创建一个线性层（nn.Linear）或其他类型的层时，这些层的权重默认会被初始化为随机值。如果你不提供任何初始化方法，或者提供一个像 lambda x: x 这样的不改变输入的 lambda 函数，那么权重将保持其随机初始化状态
        )
        self.wk = ColumnParallelLinear(
            args.dim,
            self.n_kv_heads * self.head_dim,
            bias=False,
            gather_output=False,
            init_method=lambda x: x,
        )
        self.wv = ColumnParallelLinear(
            args.dim,
            self.n_kv_heads * self.head_dim,
            bias=False,
            gather_output=False,
            init_method=lambda x: x,
        )
```

```
        #RowParallelLinear 是一种用于模型并行的线性层，它通过按行分割权重矩阵来实现并行计算
        self.wo = RowParallelLinear(
            args.n_heads * self.head_dim,
            args.dim,#从args.n_heads * self.head_dim维映射到args.dim维
            bias=False,
            input_is_parallel=True,
            init_method=lambda x: x,
        )
        #cache_k和cache_v都不参与训练，定义的大小为最大可能的大小(以下四者相乘)
        self.cache_k = torch.zeros(
            (
                args.max_batch_size,
                args.max_seq_len,
                self.n_local_kv_heads,
                self.head_dim,
            )
        ).cuda()#转成cuda的数据类型送到gpu上
        self.cache_v = torch.zeros(
            (
                args.max_batch_size,
                args.max_seq_len,
                self.n_local_kv_heads,
                self.head_dim,
            )
        ).cuda()

    def forward(
        self,
        x: torch.Tensor,#输入张量
        start_pos: int,#推理到第几个位置
        freqs_cis: torch.Tensor,#预计算的角度矩阵
        mask: Optional[torch.Tensor],#掩码矩阵
    ):

        #把x的shape拆解出来（batch_size,seq_len,dim）
        bsz, seqlen, _ = x.shape
        #把x映射到qkv上
        xq, xk, xv = self.wq(x), self.wk(x), self.wv(x)
        #本来没有头数这个维度（batch_size,seq_len,model_dim），通过view转换为(bsz,
seqlen, self.n_local_heads, self.head_dim)
        #view 函数允许你将一个张量的形状（shape）改变为任何其他形状，只要新形状的元素总数与原
始形状相同
        xq = xq.view(bsz, seqlen, self.n_local_heads, self.head_dim)
        xk = xk.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
        xv = xv.view(bsz, seqlen, self.n_local_kv_heads, self.head_dim)
        #对xq，xk进行RoPE
        xq, xk = apply_rotary_emb(xq, xk, freqs_cis=freqs_cis)
        #把cache_k和cache_v送到xq所在的设备上
        self.cache_k = self.cache_k.to(xq)
        self.cache_v = self.cache_v.to(xq)
        #把当前的kv缓存到cache_k和cache_v的新的位置上--当前推理到的位置，且每个batch对应位
置写入
        self.cache_k[:bsz, start_pos : start_pos + seqlen] = xk
        self.cache_v[:bsz, start_pos : start_pos + seqlen] = xv
        #算attention时，要用当前q和历史所有的kv进行运算，来获取上下文信息
        #从cache中取出来历史的kv
```

```
        keys = self.cache_k[:bsz, : start_pos + seqlen]
        values = self.cache_v[:bsz, : start_pos + seqlen]

        #如果n_kv_heads < n_heads，要对kv进行repeat，llama3中用的时GQA，所以要repeat
        #虽然GQA或者MQA都是一对kv对应多个q（这多个q共享同样的kv），但是在计算的时候还是要
repeat kv来进行一一计算
        keys = repeat_kv(keys, self.n_rep)  # repeat_kv之后形状变为(bs, cache_len +
seqlen, n_local_heads, head_dim)，cache_len + seqlen==历史kv+新的kv
        values = repeat_kv(values, self.n_rep)  # (bs, cache_len + seqlen,
n_local_heads, head_dim)
        #把当前query的头数维和seq_len维交换一下位置
        xq = xq.transpose(1, 2)  # (bs, n_local_heads, seqlen, head_dim)
        #kv也转一下
        keys = keys.transpose(1, 2) # (bs, n_local_heads, cache_len + seqlen,
head_dim)
        values = values.transpose(1, 2) # (bs, n_local_heads, cache_len + seqlen,
head_dim)
        #为了QK^T,keys.transpose(2, 3)对最后两维转置，然后按公式算注意力得分
        scores = torch.matmul(xq, keys.transpose(2, 3)) /
math.sqrt(self.head_dim)
        #如果由mask，还要进行mask
        if mask is not None:
            scores = scores + mask  # (bs, n_local_heads, seqlen, cache_len +
seqlen)
        #用的是普通softmax，进行按行求
        scores = F.softmax(scores.float(), dim=-1).type_as(xq)
        #乘上V
        output = torch.matmul(scores, values)  # 结果形状为(bs, n_local_heads,
seqlen, head_dim)
        #把输出转回3维，即把多个头的输出合并，先transpose把n_local_heads和head_dim放在最
后两维(bs, seqlen, n_local_heads, head_dim)
        #然后contiguous()，对其内存连续化(连续的内存布局可以提高内存访问的效率，并且view函数
要求张量在内存必须是连续的)，其实此处用reshape也行
        output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)
        #(batch_size,seq_len.model_dim)
        return self.wo(output)
```

使用位置：TransformerBlock中：

ps：用于构造tansformerblock，然后再用transformerblock去构造transformer层

```
self.attention = Attention(args)
```