分布式训练框架 Megatron ——1. Tensor并行

基础

1. 先看看 torch 基础的分布式教程，主要包括
    a. 数据并行（DP）：DISTRIBUTED DATA PARALLEL
    b. 模型 Pipeline 并行（PP）（单机多卡）：SINGLE-MACHINE MODEL PARALLEL BEST PRACTICES
    c. 混合并行（DP + PP）：Combining DDP with Model Parallelism
    d. torch 分布式通信 API：WRITING DISTRIBUTED APPLICATIONS WITH PYTORCH

这两篇博客是我对上面官方教程的一些重点内容的总结和测试，有兴趣也可以看一下。

torch分布式训练笔记

torch分布式通信基础

Megatron

1. 模型并行

先看下 Megatron 的论文，里面有对 tensor 并行的描述和设计。

Training Multi-Billion Parameter Language Models Using Model Parallelism

广义上来说，tensor 并行和 Pipeline 并行都属于模型并行，因为这两种方式都把模型切分到不同的 GPU 上了，用论文里的术语来说，就是 Multi-GPUs model。论文中用了两个不同的术语来区分，分别是 **intra-layer model parallel** 和 **inter-layer model parallel**

简单来说，Pipleline 并行，是把模型不同层切分到不同的 gpu 上，tensor 并行，就是把同一层切分到不同的 gpu，所以这两种方式是正交的（orthogonal）。这也说明两种并行方式可以一起使用。

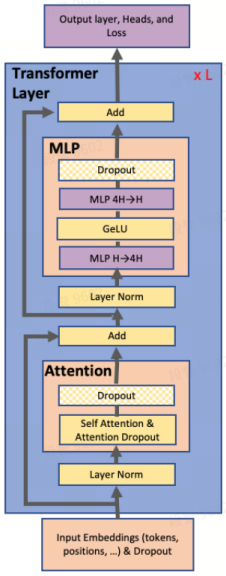
2. 源码解读

以下代码基于最新的 master 分支， git tag：3c1606b9d4b0f529028f710590beee732fb3c4b1（2023.7.10）

模型构造（Bert）

首先需要构造一个分布式的 Bert 模型，也就是 multi-gpus model。Bert 是基于 Transformer 结构的 encoder-only 模型，主要结构如下，（Attenstion + FFN） FFN 就是最简单的三层神经网络，也叫多

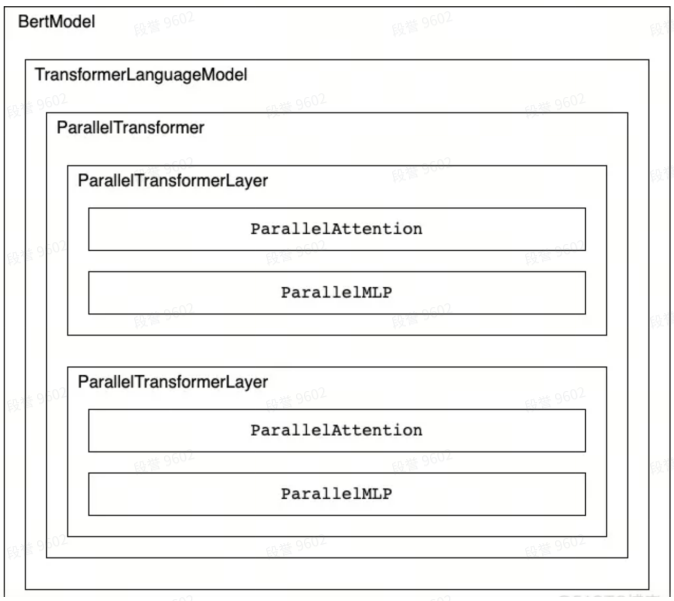
层感知机（MLP）



Bert模型结构

我们需要切分这个模型。（严格来说，切分这个词不太合适，我最初看到这个词时，脑海里的想法就是，先在 cpu 上定义一个完整的模型，然后切，把切好的每部分都拷贝到不同的卡上，实际上并不是这样，而是直接定义了一个“分布式”模型，在每张卡上都定义了模型的不同部分）

Megatron 源码的 Bert 模型定义结构如下：



上图还有些不完整，实际上 **ParrallelAttenntion** 和 **ParallelMLP** 里面还包含了 **ColumnParallelLinear** 和**RowParallelLinear**。现在从里到外，逐个阅读。

先来说说张量并行，简单的说就是把一个 GEMM，放在不同的 GPU 上算。论文里给出了该怎么切，怎么算的方法。

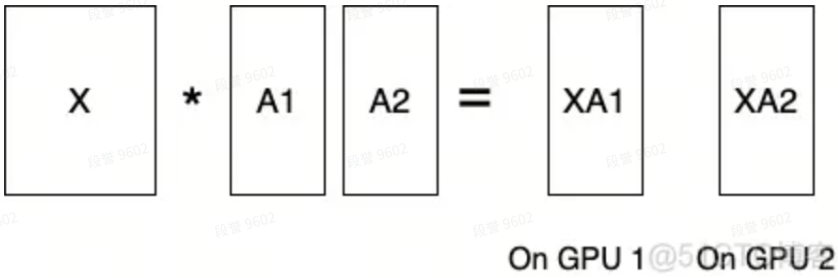
一个 GEMM 的运算公式为： **Y = XA + b**

假设现在只考虑有两张卡，我们需要把参数 **W** 切分到不同的卡上。

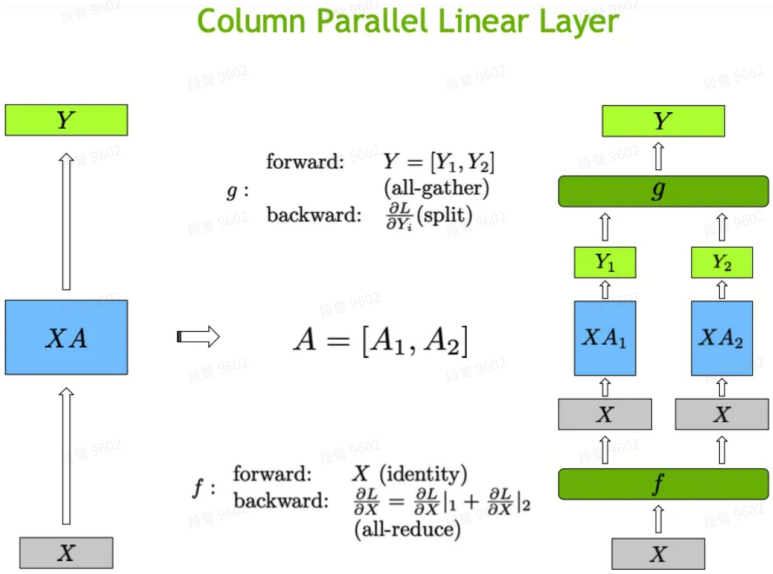
ColumnParallelLinear

切分一：把 A 按列切分，如下：

$$Y = XA = X[A_1, A_2] = [XA_1, XA_2]$$



我们在两张卡上各得到了部分结果，然后通过 all-gather 操作便可得到最后的结果，整体流程如下图所示：



f，g 都表示一种“操作”。

f函数，前向forward，因为每个gpu都是直接拿到X，所以forward就是identity function，相当于分发X到不同的gpu；而backward的时候，每个gpu上的关于X的梯度，会相加得到新的关于X的梯度，即需要的是pytorch的all-reduce函数。

g函数，前向forward是需要实现对XW1和XW2的按照最后一个维度的拼接，即out=[XW1, XW2]；反向backward时，直观上看反向应该是 gather 的反向操作 split。类似于对 $\frac{\partial L}{\partial Y}$ 进行切分，得到每个gpu上都有一份：$\frac{\partial L}{\partial Y_i}$。

记下来我们细读下 **RowParallelLinear** 这个类的代码，内容都在代码注释里。

初始化

初始化代码之中主要是用切分的信息来初始化权重，相当于不同卡上都只定义了权重的一部分！核心代码如下：

```
def __init__(self, input_size, output_size, *,
                 config: ModelParallelConfig,
                 init_method: Callable,
                 bias=True, gather_output=False, stride=1,
                 keep_master_weight_for_test=False,
                 skip_bias_add=False,
                 skip_weight_param_allocation: bool=False):
        super(ColumnParallelLinear, self).__init__()

        # Keep input parameters
        self.input_size = input_size # W的第一个维度
        self.output_size = output_size # W的第二个维度
        self.gather_output = gather_output
        # Divide the weight matrix along the last dimension.
        world_size = get_tensor_model_parallel_world_size() # 当前进程所在TP组的总进程数
        # 每块GPU上维护的hidden_size的大小，等于 原hidden_zize // TP组总进程数
        self.output_size_per_partition = divide(output_size, world_size) # 获得本子模型应输出size
        self.skip_bias_add = skip_bias_add
        self.config = config

        # Parameters.
        # Note: torch.nn.functional.linear performs XA^T + b and as a result
        # we allocate the transpose.
        # Initialize weight.
        if not skip_weight_param_allocation:
            # 用切分的size初始化权重
            if config.use_cpu_initialization: # CPU上初始化
                self.weight = Parameter(torch.empty(self.output_size_per_partition,
                                                    self.input_size,
                                                    dtype=config.params_dtype))
                if config.perform_initialization:
                    self.master_weight = _initialize_affine_weight_cpu(
                        self.weight, self.output_size, self.input_size,
                        self.output_size_per_partition, 0, init_method,
                        stride=stride, return_master_weight=keep_master_weight_for_test)
            else:
                self.weight = Parameter(torch.empty( # GPU上初始化
                    self.output_size_per_partition, self.input_size,
                    device=torch.cuda.current_device(), dtype=config.params_dtype))
                if config.perform_initialization: # 使用初始化方法 init_method，对 weight 做初始化
```

```
                    _initialize_affine_weight_gpu(self.weight, init_method,
                                                  partition_dim=0,
stride=stride)
        else:
            self.weight = None
```

forward

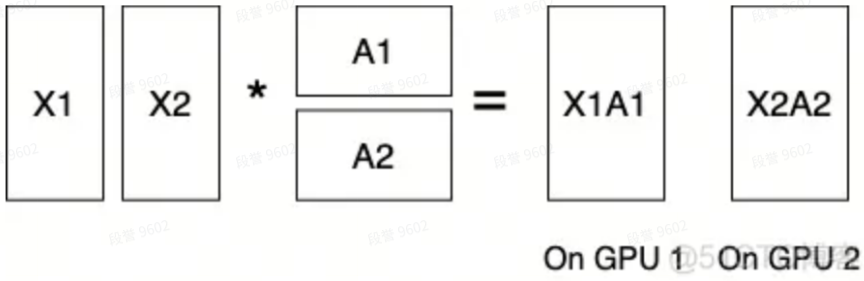
```
def forward(self,
                input_: torch.Tensor,
                weight: Optional[torch.Tensor] = None):
        bias = self.bias if not self.skip_bias_add else None

        if self.async_tensor_model_parallel_allreduce or \
                self.sequence_parallel:
            input_parallel = input_
        else:
            input_parallel = copy_to_tensor_model_parallel_region(input_) # 相当于定义列切割中的 f 算子，其做了前向copy操作，同时构建了后向 all-reduce。
        # Matrix multiply.
        output_parallel = self._forward_impl(
            input=input_parallel,
            weight=weight,
            bias=bias,
            gradient_accumulation_fusion=self.gradient_accumulation_fusion,
            async_grad_allreduce=self.async_tensor_model_parallel_allreduce,
            sequence_parallel=self.sequence_parallel
        )
        if self.gather_output:
            # All-gather across the partitions.
            assert not self.sequence_parallel
            output = gather_from_tensor_model_parallel_region(output_parallel) #相当于定义列切割中的 g 算子， 前向传播时候做 all-gather，同时构建反向传播时的scatter
        else:
            output = output_parallel
        output_bias = self.bias if self.skip_bias_add else None
        return output, output_bias
```

RowParallelLinear

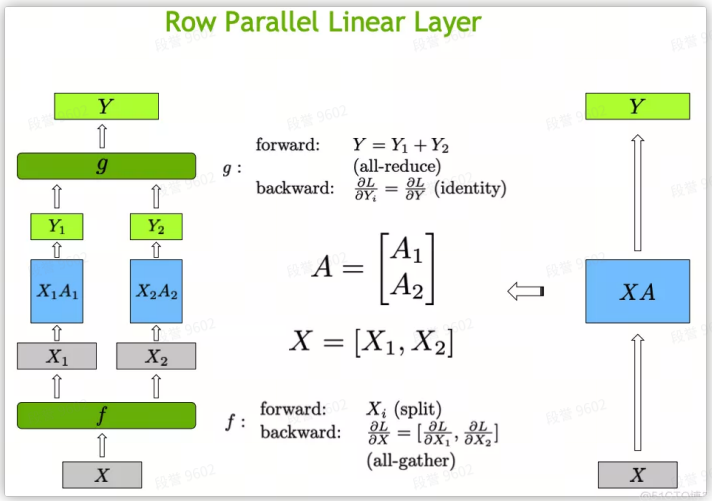
切法二： W 按行切分

把 A 按照行分割成两部分。为了保证运算，同时我们也把 X 按照列来分割为两部分，这里 X1 的最后一个维度等于A1 最前的一个维度，理论上是：

$$XA = [X_1 \quad X_2] \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = X_1A_1 + X_2A_2 = Y_1 + Y_2 = Y$$



注意 A1和A2在两张不同的卡上，所以 Y1+Y2 是 All-reduce 操作，整体流程如下图所示：



根据上面的切分方式，前向计算时，f 需要按列切分（split） x，所以是 scatter， g 则是 all-reduce（每个gpu上的运算结果相加到一起，会涉及到gpu的wait，即一个gpu运行完毕之后，需要等待）。

反向计算时：f 是 scatter，直观考虑，反向应该是切割的反操作，即拼接，我们就是要按照最后一列对“梯度”进行拼接，所以反向是 gather，同时梯度信息要同步到所有的机器上，所以是 all-gather。

g 是 all-reduce， 本质就是加法操作，故 $\frac{\partial L}{\partial Y_i} = \frac{\partial L}{\partial Y}$ ，所以 g 的方向操作直接拷贝梯度即可，两块GPU就可以独立计算各自权重的梯度了。（原封不动的传递给每个gpu，继续反向传播的过程）

初始化：

初始化比较简单，就是切分权重 A， 每张卡上只 hold 权重 A 的一部分。

Forward

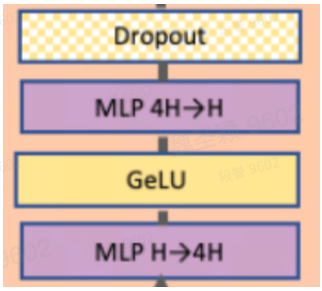
forward 代码之中，主要是实施了 f 和 g 的forward操作，同时把 f 和 g 的backward 操作搭建起来，具体如下：

```python
def forward(self, input_):
        # 该函数要求： 输入的张量已经被分割到每个GPU，输出张量是all-reduce之后的整体
        # Set up backprop all-reduce.
        if self.input_is_parallel: # 是否已经是split的输入
            input_parallel = input_ # 到达这里，因为已经split，所以直接就接了输入，不会scatter
        else:
            assert not self.sequence_parallel
            # 独立 row parallel 线性层到这里，会进行前向切分和后向拼接
            input_parallel = scatter_to_tensor_model_parallel_region(input_) # 否则就需要通过分布式 scatter api 在 TP 组内部切
        # Matrix multiply.（进行矩阵乘法运算）
        output_parallel = self._forward_impl(
            input=input_parallel,
            weight=self.weight,
            bias=None,
            gradient_accumulation_fusion=self.gradient_accumulation_fusion,
            async_grad_allreduce=False,
            sequence_parallel=False,
        )

        # All-reduce across all the partitions.
        # 进行前向all-reduce操作，这样每个GPU之上都是完整的最新结果，同时搭建了后向的identity操作。
        if self.sequence_parallel: # 暂时不考虑 sequence_parallel
            output_ = reduce_scatter_to_sequence_parallel_region(output_parallel)
        else:
            output_ = reduce_from_tensor_model_parallel_region(output_parallel)
        if not self.skip_bias_add:
            output = output_ + self.bias if self.bias is not None else output_
            output_bias = None
        else:
            output = output_
            output_bias = self.bias
        return output, output_bias
```

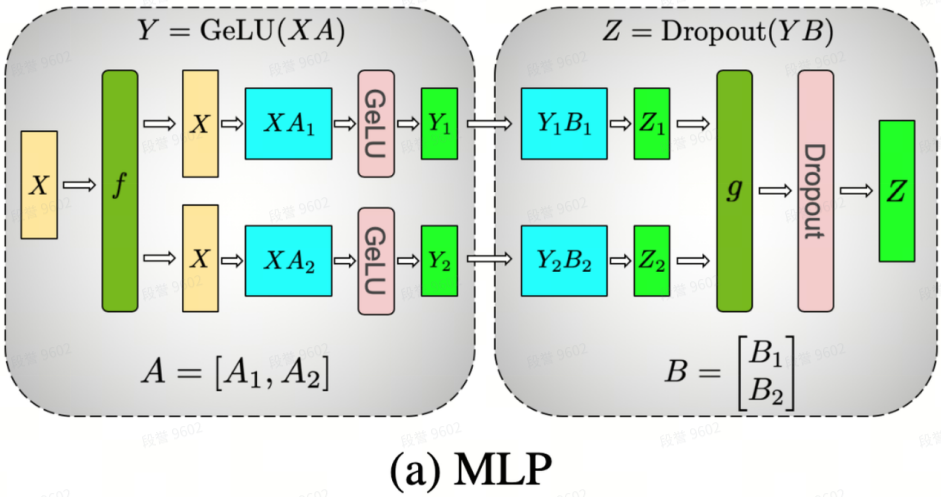
f 和 **g** 对应代码中的 **scatter_to_tensor_model_parallel_region，reduce_from_tensor_model_parallel_region**

ParallelMLP

上面构建了“分布式矩阵乘法”，接下来就可以构建 transformer 模型中的 MLP 层了。



显然，里面有两个矩阵乘法，根据论文中的切法，流程如下：



(a) MLP

MLP的概要图，f和g是两个关键的函数，其中f来自“纵刀流”的f；而g来自“横刀流”的g

第一个矩阵乘法按列切，第二个矩阵乘法按行切，至于为什么这么切，论文里有很详细的解释（为了减少 GPU 之间的同步，提高 GPU 的利用率）。下面的这个类的核心代码（相比源码我做了删减，保留了核心流程代码）。

```
class ParallelMLP(MegatronModule):
    """MLP.
    MLP will take the input with h hidden state, project it to 4*h
    hidden dimension, perform nonlinear transformation, and project the
    state back into h hidden dimension.
    """

    def __init__(self, config):
        super(ParallelMLP, self).__init__()
        # Project to 4h. If using swiglu double the output width, see https://arxiv.org/pdf/2002.05202.pdf
        self.dense_h_to_4h = tensor_parallel.ColumnParallelLinear( # 第一个矩阵乘法按列切
            config.hidden_size,
            ffn_hidden_size,
            config=config,
            init_method=config.init_method,
            bias=self.add_bias,
            gather_output=False,
```

```
            skip_bias_add=True,
        )

        self.activation_func = F.gelu # 激活函数 gelu

        # Project back to h.
        self.dense_4h_to_h = tensor_parallel.RowParallelLinear( # 第二个矩阵乘法按行切
            config.ffn_hidden_size,
            config.hidden_size,
            config=config,
            init_method=config.output_layer_init_method,
            bias=self.add_bias,
            input_is_parallel=True
        )

    def forward(self, hidden_states):
        # [s, b, 4hp]
        intermediate_parallel, bias_parallel = self.dense_h_to_4h(hidden_states)

        intermediate_parallel = self.activation_func(intermediate_parallel)

        # [s, b, h]
        output, output_bias = self.dense_4h_to_h(intermediate_parallel)
        return output, output_bias
```

非常简单，只需要利用前面写好的两个“分布式”矩阵乘法类就行了。

ParallelAttention

接下来就是 Attention 了，准确说是 multi head attention，这是 transformer 结构最核心部分，也是代码中较难的一部分，因为参数比较多，需要细读。

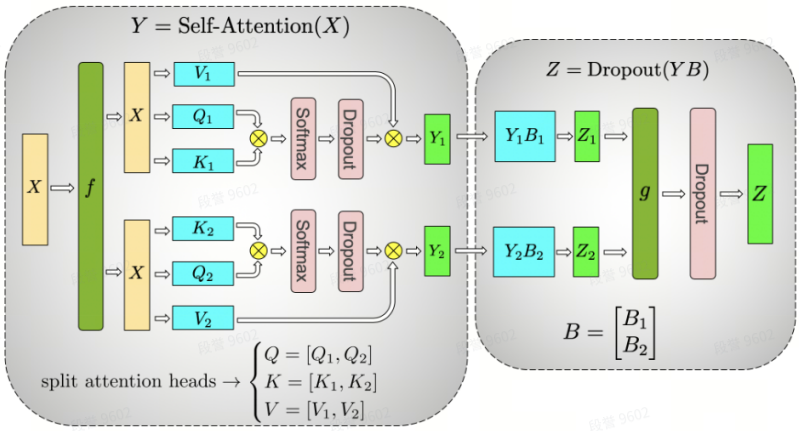
下面以 self-attentionn 为例讲解，cross-attention 类似。

> "这里说点题外话， hidden_size 给人的第一感觉就是三层神经网络中的隐藏层大小，但是在 transformer 中，一般表示的就是 embedding size 的大小。个人感觉的原因就是最直观上的原因，因为 tranformer 结构往往是多个 stack 在一起，tranformer 内部之间的输入输出shape 都是 [batch, seq, embedding]，所以就把 embeding size 叫做 hidden size。

ParallelAttention 的整体思路就是，把 W^{QKV} 这个矩阵按照列来切分，这个矩阵原始的 shape 为：[hidden_size, 3 * heads * d_q] 。按头切到不同的 GPU 上，也就是说，head 的数量，应该 >= GPU 的数量。

所以每个 GPU 上分得的shape 为：[hidden_size, 3 * heads / TP, d_p]。 TP表示当前 TP 进程组进程的个数。然后每个 GPU 上独立的进行 attention 计算，接着与一个矩阵相乘（该矩阵按行切），把多头的结果 project 到与输入一样的shape，最后做 reduce 操作。

如下图所示：



(b) Self-Attention

代码中， np表示： num_attention_heads_per_partition，即 heads / TP

hn表示：hidden_size_per_attention_head， 可以理解为新的 hidden_size, 毕竟transformer 模型不直接处理 x，而是处理 project 之后的 x，即 q k v，这样做的目的是提高模型的拟和能力（可以猛戳这篇博客了解一下

transformer中的Q,K,V到底是什么？），所以 hidden_size_per_attention_head = kv_channels

hp = hn * np

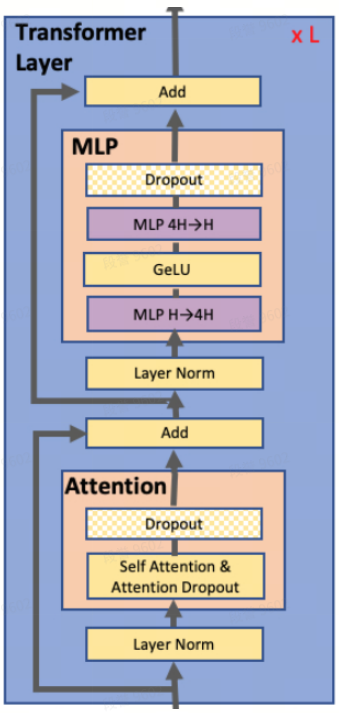
接下来看详细代码，注意 megatron 中，attention 的输入被处理成了 [sq, b, h]。在self attention 中，这里 sq，sk == seq（sequence len），核心代码如下：

```
def forward(self, hidden_states, attention_mask,
                encoder_output=None, inference_params=None,
                rotary_pos_emb=None):
        # hidden_states: [sq, b, h]
        # =====================
        # Query, Key, and Value
        # =====================

        # Attention heads [sq, b, h] --> [sq, b, (np * 3 * hn)]
        mixed_x_layer, _ = self.query_key_value(hidden_states) # 每个 GPU 上计算 qkv

        # [sq, b, (np * 3 * hn)] --> [sq, b, np, 3 * hn]
        new_tensor_shape = mixed_x_layer.size()[:-1] + \
            (self.num_attention_heads_per_partition,
             3 * self.hidden_size_per_attention_head)
```

```
16          mixed_x_layer = mixed_x_layer.view(*new_tensor_shape)
17
18          # [sq, b, np, 3 * hn] --> 3 [sq, b, np, hn]
19          (query_layer,
20           key_layer,
21           value_layer) =
    tensor_parallel.split_tensor_along_last_dim(mixed_x_layer, 3)
22          # ==================================
23          # core attention computation
24          # ==================================
25          q, k, v = [rearrange(x, 's b ... -> b s ...').contiguous()
26                     for x in (query_layer, key_layer, value_layer)]
27          context_layer = self.core_attention( # 完成的是 attention 的核心计算
28                      query_layer, key_layer, value_layer, attention_mask)
29
30          # =================
31          # Output. [sq, b, h]
32          # =================
33
34          output, bias = self.dense(context_layer) # 把多头的结果在 project 到
    hidden size
35
36          return output, bias
```

ParallelTransformerLayer

离胜利只剩一步了，接下来就是 ParalleTransformer，这里只以最简单的 encoder 结构来对照代码，核心代码很简单，和下面结构完全对应：



```python
def forward(self, hidden_states, attention_mask,
                encoder_output=None, enc_dec_attn_mask=None,
                retriever_input=None,
                retriever_output=None,
                retriever_attn_mask=None,
                inference_params=None,
                rotary_pos_emb=None):
        # hidden_states: [s, b, h]
        # Layer norm at the beginning of the transformer layer.
        layernorm_output = self.input_layernorm(hidden_states)

        # Self attention.
        attention_output, attention_bias = \
            self.self_attention(
                layernorm_output,
                attention_mask,
                inference_params=inference_params,
                rotary_pos_emb=rotary_pos_emb)

        # dropout + Residual connection.
        if self.apply_residual_connection_post_layernorm:
            residual = layernorm_output
        else:
            residual = hidden_states

        out = torch.nn.functional.dropout(attention_output + attention_bias,
                                          p=self.hidden_dropout,
                                          training=self.training)
        layernorm_input = residual + self.drop_path(out)

        # Layer norm post the self attention.
        layernorm_output = self.post_attention_layernorm(layernorm_input)

        # MLP.
        mlp_output, mlp_bias = self.mlp(layernorm_output)

        # droupout + Second residual connection.
        if self.apply_residual_connection_post_layernorm:
            residual = layernorm_output
        else:
            residual = layernorm_input

        out = torch.nn.functional.dropout(mlp_output,
                                          p=self.hidden_dropout,
                                          training=self.training)
        output = residual + self.drop_path(out)

```

```
48          return output
```

ParallelTransformer

最后就是 ParallelTransformer，由于模型的种类很多，encoder-only， decoder-only，encoder-decoder之类的，所以这个类做了很多封装，但实际核心代码就是重复上面的 ParallelTransformerLayer 这一层。

```
1   def build_layer(layer_number):
2       return ParallelTransformerLayer(
3           config,
4           layer_number,
5           layer_type=current_layer_type,
6           self_attn_mask_type=self_attn_mask_type,
7           drop_path_rate=self.drop_path_rates[layer_number - 1])
8
9   self.layers = torch.nn.ModuleList(
10                  [build_layer(i + 1 + offset) for i in range(self.num_layers)])
```

参考

Training Multi-Billion Parameter Language Models Using Model Parallelism

Megatron论文和代码