ZOMI 大模型：分布式训练

张量并行

Tensor Parallel

大模型业务全流程

数据 & 模型算法

2. 数据处理

- 开源数据
- 数据预处理
- 向量数据库

3. 模型算法

- LLM模型架构
- 多模态一切皆Tokens

模型训练 & 微调

4. 模型训练

- 混合精度
- 梯度检查
- 梯度累积
- …
- 5. 分布式并行
- 训练集群稳定性

6. 模型微调

- 全参微调
- 低参微调
- 指令微调

模型验证 & 推理部署

7. 模型验证

- 下游任务
- 测评标准
- benchmark

8. 推理与智能体

- 量化压缩
- 推理加速
- 9. Agent智能体

1. AI 集群建设

- 计算、存储、网络
- AI 集群机房建设
- AI 集群上线与运维

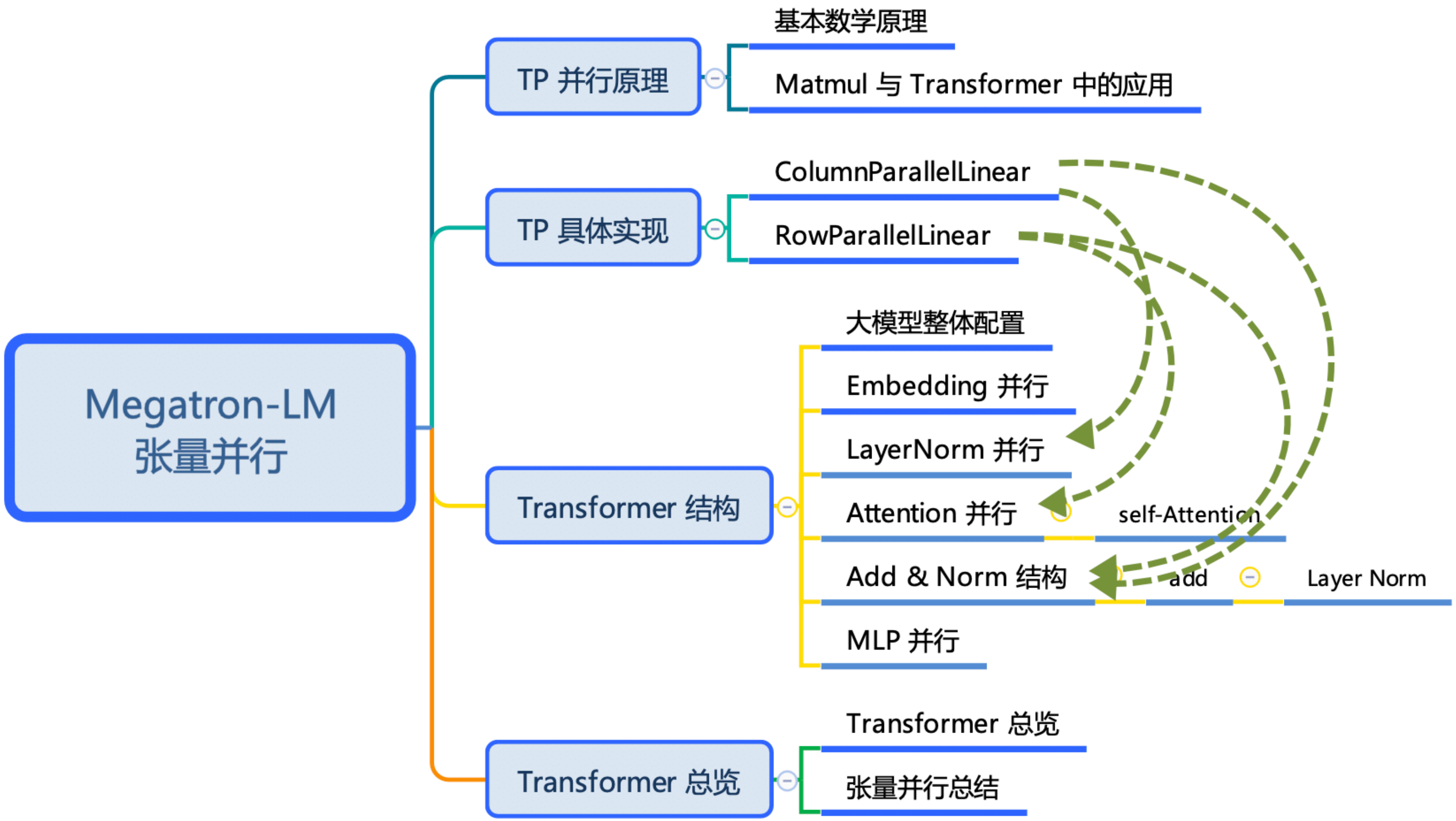
集群算力准备

大模型系列 – 分布式训练加速

- ## 具体内容

1. **分布式加速库：**
   - 业界常用分布式加速库 & 作用
2. **DeepSpeed 特性：**
   - 基本概念 - 整体框架 – Zero-1/2/3 – ZeRO-Offload – ZeRO-Infinity
3. **Megatron 特性：**
   1. 总体介绍 – 整体流程 – 并行配置 – DP – TP – PP

思维导图



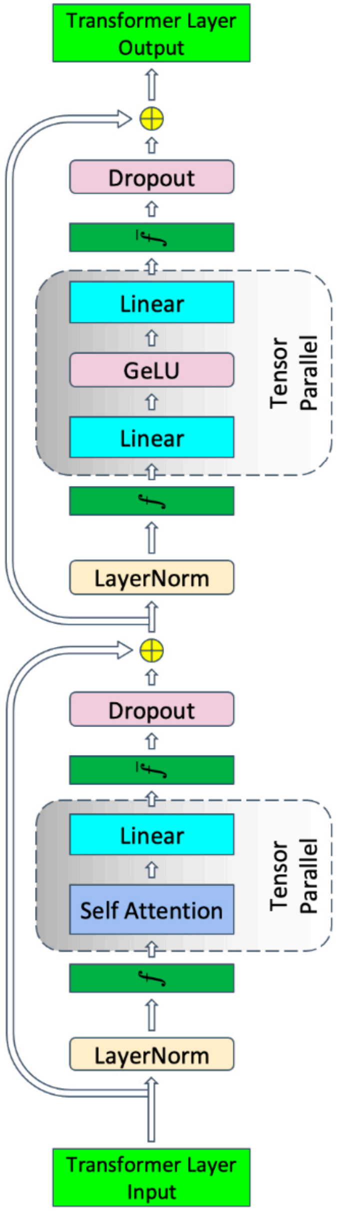
Megatron-LM 张量并行
TP 并行原理
基本数学原理
Matmul 与 Transformer 中的应用
TP 具体实现
ColumnParallelLinear
RowParallelLinear
Transformer 结构
大模型整体配置
Embedding 并行
LayerNorm 并行
Attention 并行
self-Attention
Add & Norm 结构
add
Layer Norm
MLP 并行
Transformer 总览
Transformer 总览
张量并行总结

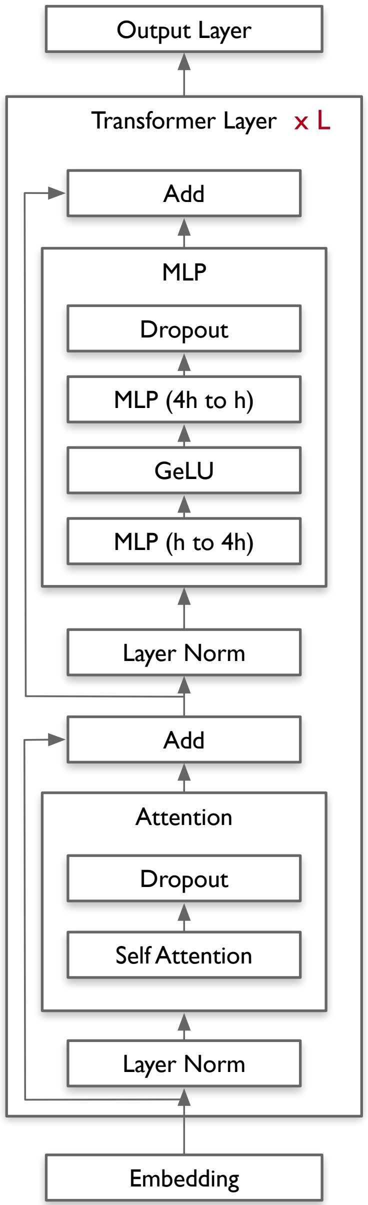
Megatron-LM 01

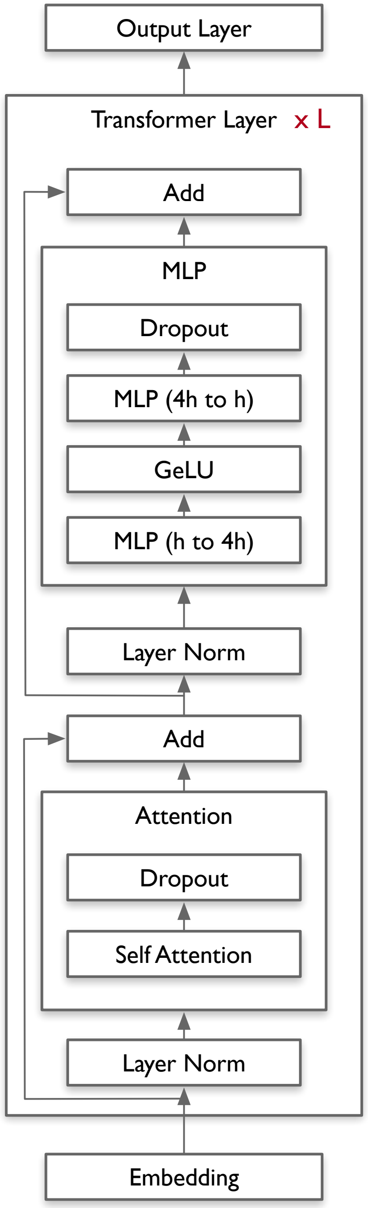
大模型整体配置

配置规范

- h: hidden size
- n: number of attention heads
- p: number of model parallel partitions
- np: n/p (attention heads / mp)
- hp: h/p (hidden size / mp)
- hn: h/n (hidden size / attention heads)
- b: batch size
- s: sequence length
- L: number of layers
- Transformer 输入 Shape [s, b, h]，输出 Shape [s, b, h]







模型配置

```python
if __name__ == "__main__":

    # Temporary for transition to core datasets
    train_valid_test_datasets_provider.is_distributed = True

    pretrain(train_valid_test_datasets_provider,
             model_provider,
             ModelType.encoder_or_decoder,
             forward_step,
             args_defaults={'tokenizer_type': 'GPT2BPETokenizer'})
```

```python
def model_provider(pre_process=True, post_process=True) -> Union[GPTModel, megatron.legacy.moc
    """Builds the model.

    Returns:
        Union[GPTModel, megatron.legacy.model.GPTModel]: The returned model
    """
    model = GPTModel(
        config=config,
        transformer_layer_spec=transformer_layer_spec,
        vocab_size=args.padded_vocab_size,
        max_sequence_length=args.max_position_embeddings,
        pre_process=pre_process,
        post_process=post_process,
        fp16_lm_cross_entropy=args.fp16_lm_cross_entropy,
        parallel_output=True,
        share_embeddings_and_output_weights=not args.untie_embeddings_and_output_weights,
        position_embedding_type=args.position_embedding_type,
        rotary_percent=args.rotary_percent,
    )

    return model
```

```python
class GPTModel(LanguageModule):
    """GPT Transformer language model.…

    def __init__(
        self,
        config: TransformerConfig,
        transformer_layer_spec: ModuleSpec,
        vocab_size: int,
        max_sequence_length: int,
        pre_process: bool = True,
        post_process: bool = True,
        fp16_lm_cross_entropy: bool = False,
        parallel_output: bool = True,
        share_embeddings_and_output_weights: bool = False,
        position_embedding_type: Literal['learned_absolute', 'rope'] = 'learned_a
        rotary_percent: float = 1.0,
        rotary_base: int = 10000,
        seq_len_interpolation_factor: Optional[float] = None,
    ) -> None:
```

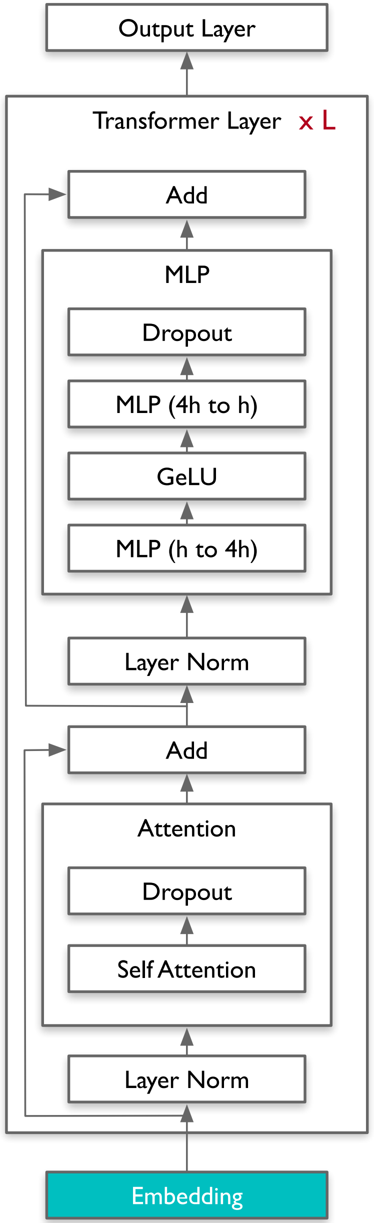
megatron > core > models > gpt > gpt_model.py > ...

- pretrain_gpt.py main函数实例 GPTModel 类
- GPTModel 类继承 LanguageModule 类并开始构建 GPT 模型

```bash
# Runs the "345M" parameter model
export CUDA_DEVICE_MAX_CONNECTIONS=1

CHECKPOINT_PATH=<Specify path>
VOCAB_FILE=<Specify path to file>/gpt2-vocab.json
MERGE_FILE=<Specify path to file>/gpt2-merges.txt
DATA_PATH=<Specify path and file prefix>_text_document

GPT_ARGS="
    --tensor-model-parallel-size 2 \
    --pipeline-model-parallel-size 1 \
    --num-layers 24 \
    --hidden-size 1024 \
    --num-attention-heads 16 \
    --seq-length 1024 \
    --max-position-embeddings 1024 \
    --voc 50000 \
    --micro-batch-size 4 \
    --global-batch-size 8 \
    --lr 0.00015 \
    --train-iters 500000 \
    --lr-decay-iters 320000 \
    --lr-decay-style cosine \
```

Megatron-LM 02

Embedding 并行



Embedding 并行

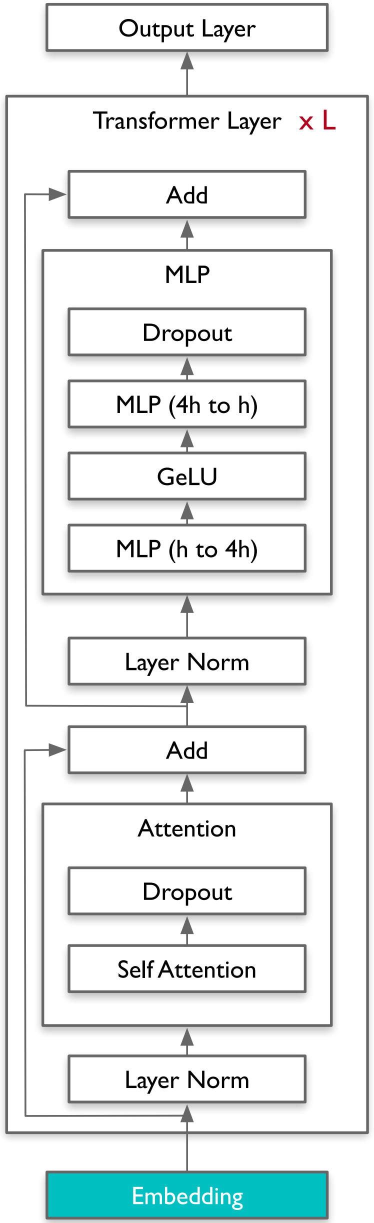
```python
class GPTModel(LanguageModule):
    def __init__(
        # These 2 attributes are needed for TensorRT-LLM export.
        self.max_position_embeddings = max_sequence_length
        self.rotary_percent = rotary_percent

        if self.pre_process:
            self.embedding = LanguageModelEmbedding(
                config=self.config,
                vocab_size=self.vocab_size,
                max_sequence_length=self.max_sequence_length,
                position_embedding_type=position_embedding_type,
            )
```

```python
class LanguageModelEmbedding(MegatronModule):
    def __init__(

        self.config: TransformerConfig = config
        self.vocab_size: int = vocab_size
        self.max_sequence_length: int = max_sequence_length
        self.add_position_embedding: bool = position_embedding_type == 'lea
        self.num_tokentypes = num_tokentypes

        # Word embeddings (parallel).
        self.word_embeddings = tensor_parallel.VocabParallelEmbedding(
            num_embeddings=self.vocab_size,
            embedding_dim=self.config.hidden_size,
            init_method=self.config.init_method,
            config=self.config,
        )
```

```python
class LanguageModelEmbedding(MegatronModule):
    def forward(self, input_ids: Tensor, position_ids: Tensor, tokentype_ids: i
        """Forward pass of the embedding module.

        Args:
            input_ids (Tensor): The input tokens
            position_ids (Tensor): The position id's used to calculate position
            tokentype_ids (int): The token type ids. Used when args.bert_binary

        Returns:
            Tensor: The output embeddings
        """
        word_embeddings = self.word_embeddings(input_ids)
        if self.add_position_embedding:
            position_embeddings = self.position_embeddings(position_ids)
            embeddings = word_embeddings + position_embeddings
        else:
            embeddings = word_embeddings
```

- GPTModel 类实例化 LanguageModelEmbedding 函数
- TransformerLanguageModel 类先构建 Embedding 层

```bash
# Runs the "345M" parameter model
export CUDA_DEVICE_MAX_CONNECTIONS=1

CHECKPOINT_PATH=<Specify path>
VOCAB_FILE=<Specify path to file>/gpt2-vocab.json
MERGE_FILE=<Specify path to file>/gpt2-merges.txt
DATA_PATH=<Specify path and file prefix>_text_document

GPT_ARGS="
    --tensor-model-parallel-size 2 \
    --pipeline-model-parallel-size 1 \
    --num-layers 24 \
    --hidden-size 1024 \
    --num-attention-heads 16 \
    --seq-length 1024 \
    --max-position-embeddings 1024 \
    --voc 50000 \
    --micro-batch-size 4 \
    --global-batch-size 8 \
    --lr 0.00015 \
    --train-iters 500000 \
    --lr-decay-iters 320000 \
    --lr-decay-style cosine \
```

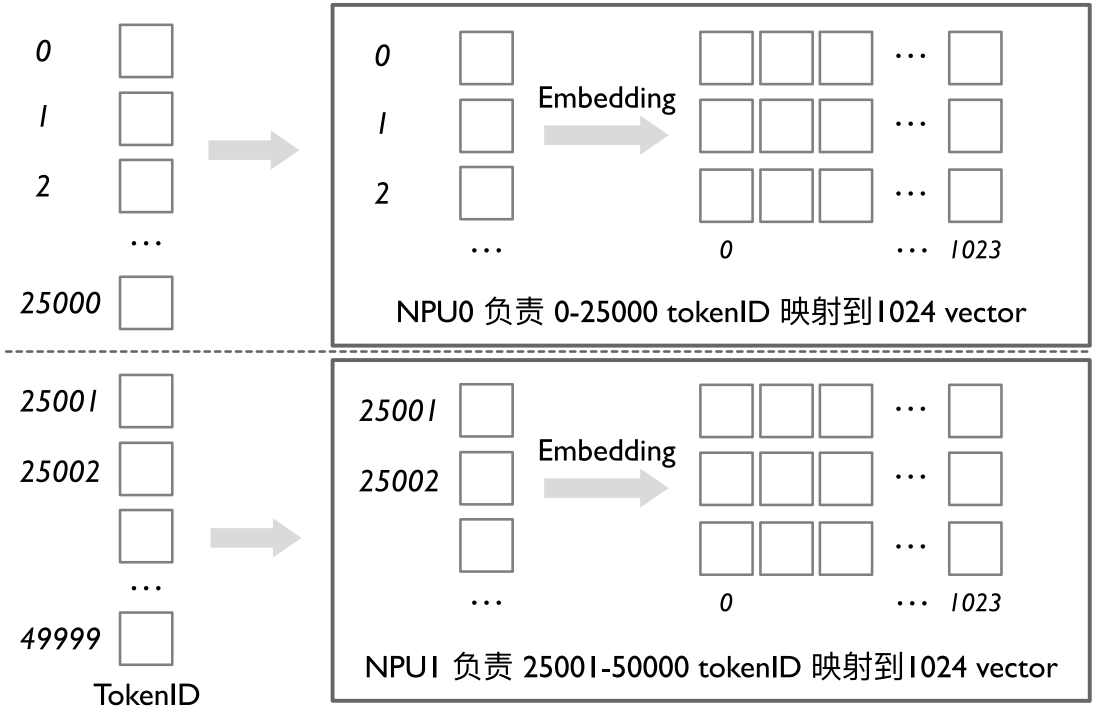


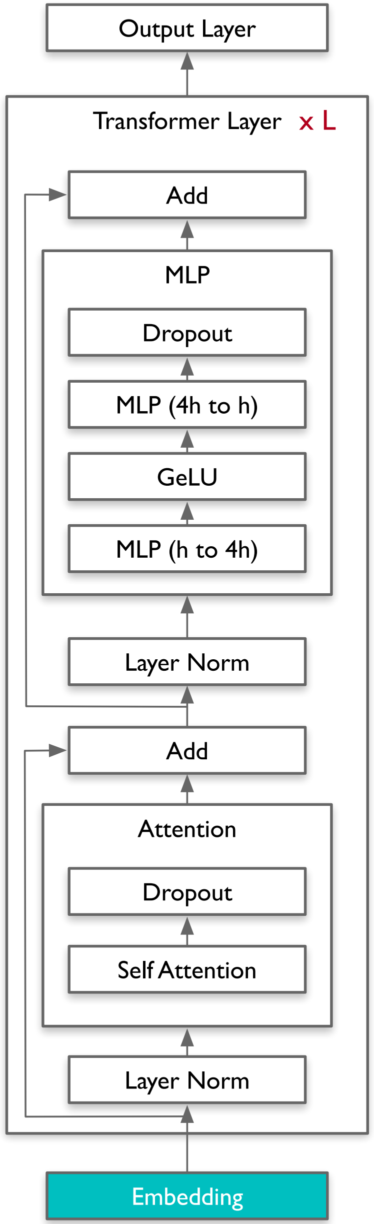
Embedding 并行

megatron > core > tensor_parallel > layers.py > VocabParallelEmbedding > __init__

```python
class VocabParallelEmbedding(torch.nn.Module):
    """Embedding parallelized in the vocabulary dimension.…

    def __init__(
        self,
        num_embeddings: int,
        embedding_dim: int,
        *,
        init_method: Callable,
        config: ModelParallelConfig,
    ):
        super(VocabParallelEmbedding, self).__init__()
        # Keep the input dimensions.
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim
        self.tensor_model_parallel_size = get_tensor_model_parallel_world_siz
        # Divide the weight matrix along the vocaburaly dimension.
        (
            self.vocab_start_index,
            self.vocab_end_index,
        ) = VocabUtility.vocab_range_from_global_vocab_size(
            self.num_embeddings, get_tensor_model_parallel_rank(), self.tenso
        )
        self.num_embeddings_per_partition = self.vocab_end_index - self.vocab
```

- Embedding 对输入 token sequence 执行 Embedding 并行
  - tp_size：2
  - vocal_size：50000
  - embeddiing_dim：1024
- 实现 50000+ 个 Tokens dict，每个token映射到1024向量的Embedding





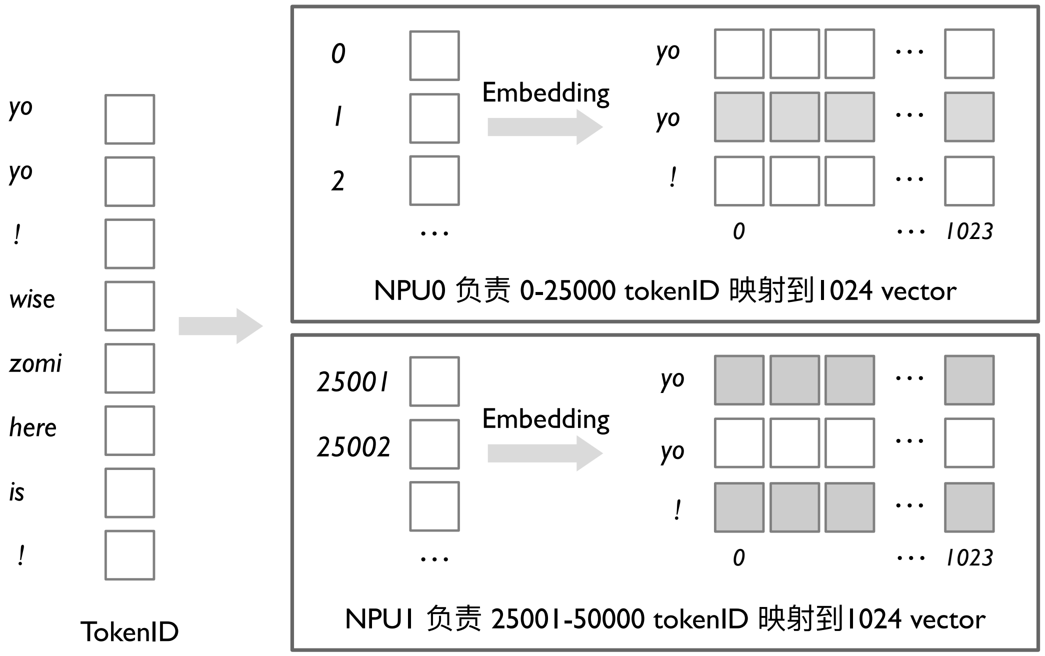
Embedding 并行

megatron > core > tensor_parallel > layers.py > VocabParallelEmbedding > __init__

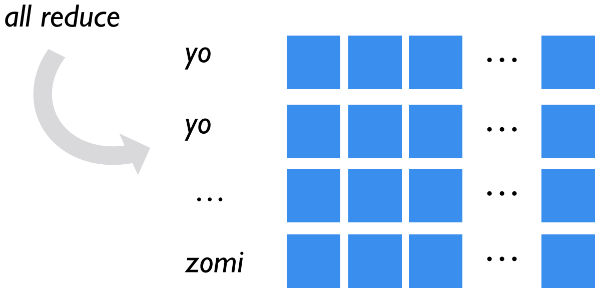
```python
class VocabParallelEmbedding(torch.nn.Module):
    """Embedding parallelized in the vocabulary dimension.…

    def __init__(
        self,
        num_embeddings: int,
        embedding_dim: int,
        *,
        init_method: Callable,
        config: ModelParallelConfig,
    ):
        super(VocabParallelEmbedding, self).__init__()
        # Keep the input dimensions.
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim
        self.tensor_model_parallel_size = get_tensor_model_parallel_world_siz
        # Divide the weight matrix along the vocaburaly dimension.
        (
            self.vocab_start_index,
            self.vocab_end_index,
        ) = VocabUtility.vocab_range_from_global_vocab_size(
            self.num_embeddings, get_tensor_model_parallel_rank(), self.tenso
        )
        self.num_embeddings_per_partition = self.vocab_end_index - self.vocab_
```

```python
class VocabParallelEmbedding(torch.nn.Module):

    def forward(self, input_):
        if self.tensor_model_parallel_size > 1:
            # Build the mask.
            input_mask = (input_ < self.vocab_start_index) | (input_ >= self.vocab_
            # Mask the input.
            masked_input = input_.clone() - self.vocab_start_index
            masked_input[input_mask] = 0
        else:
            masked_input = input_
        # Get the embeddings.
        output_parallel = self.weight[masked_input]
        # Mask the output embedding.
        if self.tensor_model_parallel_size > 1:
            output_parallel[input_mask, :] = 0.0
        # Reduce across all the model parallel GPUs.
        output = reduce_from_tensor_model_parallel_region(output_parallel)
        return output
```

- 2个 NPU Embedding 并行前向执行计算

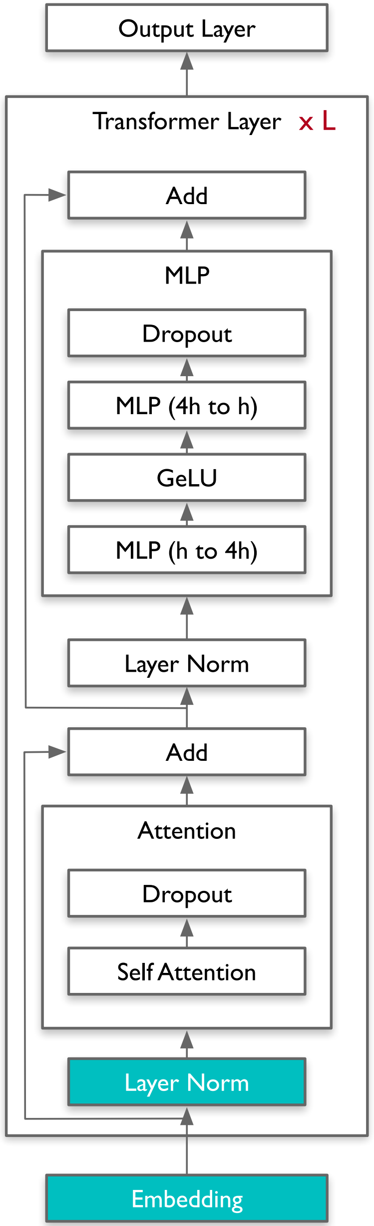


- NPU0 和 NPU1 对output_parallel 执行 all reduce
- 生成完整的input embedding shape[s, b, h]



Megatron-LM 03

LayerNorm 并行



LayerNorm 并行

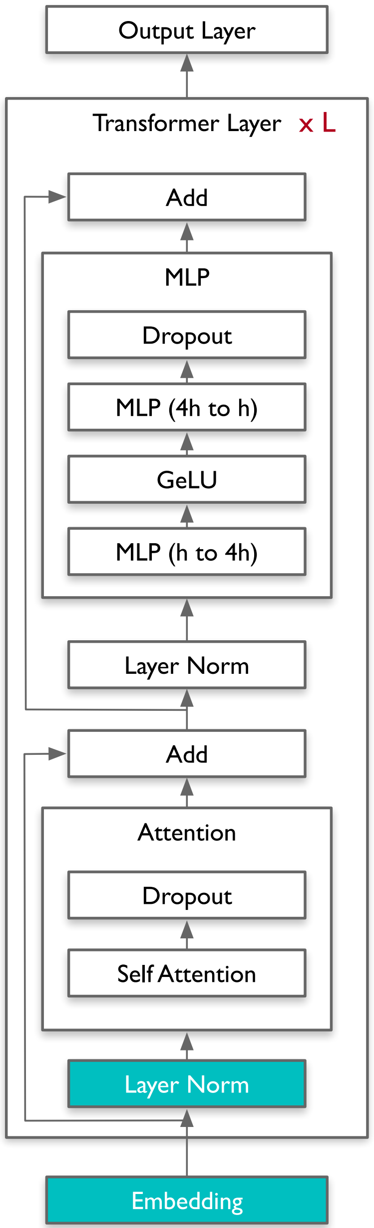
```python
class GPTModel(LanguageModule):
    def __init__(
        if self.pre_process:
            self.embedding = LanguageModelEmbedding(
                config=self.config,
                vocab_size=self.vocab_size,
                max_sequence_length=self.max_sequence_length,
                position_embedding_type=position_embedding_type,
            )

        if self.position_embedding_type == 'rope':
            self.rotary_pos_emb = RotaryEmbedding(
                kv_channels=self.config.kv_channels,
                rotary_percent=rotary_percent,
                rotary_interleaved=self.config.rotary_interleaved,
                seq_len_interpolation_factor=seq_len_interpolation
                rotary_base=rotary_base,
            )

        # Transformer.
        self.decoder = TransformerBlock(
            config=self.config,
            spec=transformer_layer_spec,
            pre_process=self.pre_process,
            post_process=self.post_process,
        )
```

```python
class TransformerBlock(MegatronModule):
    """Transformer class."""

    def __init__(
        self,
        config: TransformerConfig,
        spec: Union[TransformerBlockSubmodules, ModuleSpec],
        post_layer_norm: bool = True,
        pre_process: bool = True,
        post_process: bool = True,
    ):
        super().__init__(config=config)

        self.submodules = _get_block_submodules(config, spec)
        self.post_layer_norm = post_layer_norm
        self.pre_process = pre_process
        self.post_process = post_process
```

- GPTModel 实例化 TransformerBlock，输入shape [s, b, h]
- 设置 Layer-Norm 的位置属于 post or pre

```bash
# Runs the "345M" parameter model
export CUDA_DEVICE_MAX_CONNECTIONS=1

CHECKPOINT_PATH=<Specify path>
VOCAB_FILE=<Specify path to file>/gpt2-vocab.json
MERGE_FILE=<Specify path to file>/gpt2-merges.txt
DATA_PATH=<Specify path and file prefix>_text_document

GPT_ARGS="
    --tensor-model-parallel-size 2 \
    --pipeline-model-parallel-size 1 \
    --num-layers 24 \
    --hidden-size 1024 \
    --num-attention-heads 16 \
    --seq-length 1024 \
    --max-position-embeddings 1024 \
    --voc 50000 \
    --micro-batch-size 4 \
    --global-batch-size 8 \
    --lr 0.00015 \
    --train-iters 500000 \
    --lr-decay-iters 320000 \
    --lr-decay-style cosine \
```

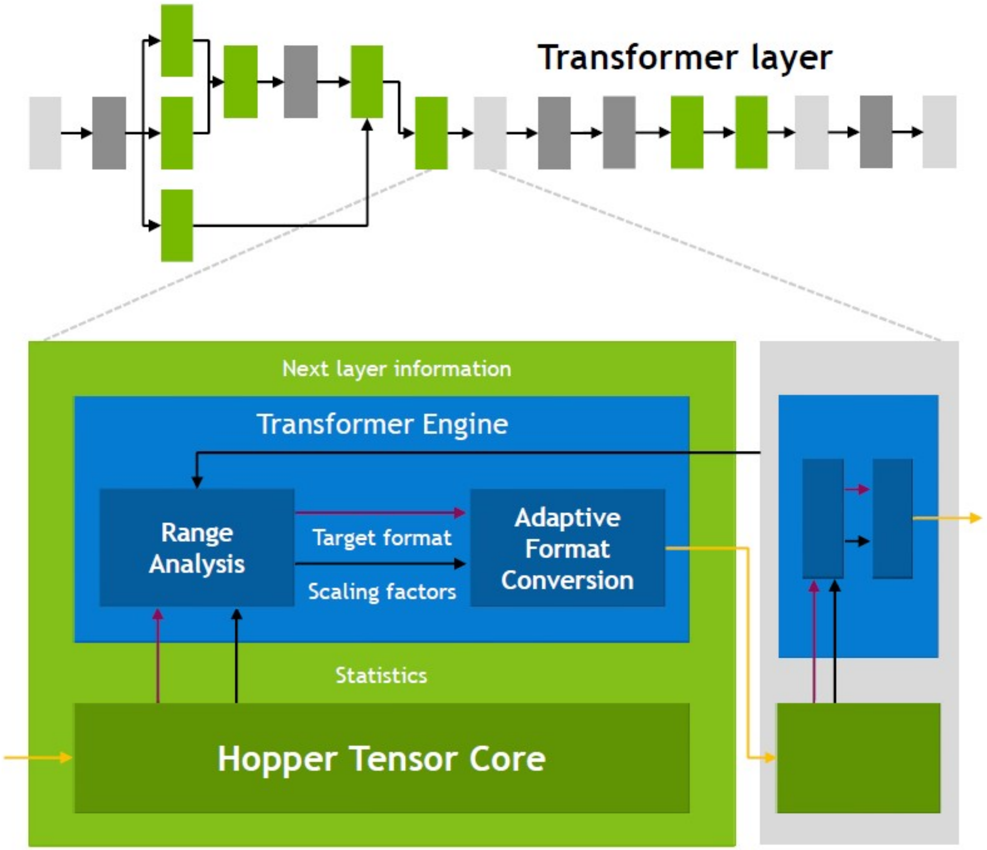
LayerNorm 并行




```python
class TransformerBlock(MegatronModule):
    def _build_layers(self):

        if self.post_process and self.post_layer_norm:
            # Final layer norm before output.
            self.final_layernorm = TENorm(
                config=self.config,
                hidden_size=self.config.hidden_size,
                eps=self.config.layernorm_epsilon,
            )

    def _get_layer(self, layer_number: int):
        return self.layers[layer_number]
```

```python
class TENorm:
    """
    A conditional wrapper to initialize an instance of Transformer-Engine's
    `LayerNorm` or `RMSNorm` based on input
    """

    # TODO should we ditch normalization config and just use spec to choose LayerNor
    def __new__(
        cls, config: TransformerConfig, hidden_size: int, eps: float = 1e-5,
    ):
        if config.normalization == "LayerNorm":
            instance = te.pytorch.LayerNorm(
                hidden_size=hidden_size,
                eps=eps,
                sequence_parallel=config.sequence_parallel,
                zero_centered_gamma=config.layernorm_zero_centered_gamma,
                **_get_extra_te_kwargs(config),
            )
        elif config.normalization == "RMSNorm":
            assert hasattr(
                te.pytorch, "RMSNorm"
            ), "Transformer-Engine >= v0.11 required to use this feature"
            instance = te.pytorch.RMSNorm(
                hidden_size=hidden_size,
                eps=eps,
                sequence_parallel=config.sequence_parallel,
                zero_centered_gamma=config.layernorm_zero_centered_gamma,
                **_get_extra_te_kwargs(config),
            )
        else:
            raise Exception('Only LayerNorm and RMSNorm are curently supported')

        return instance
```

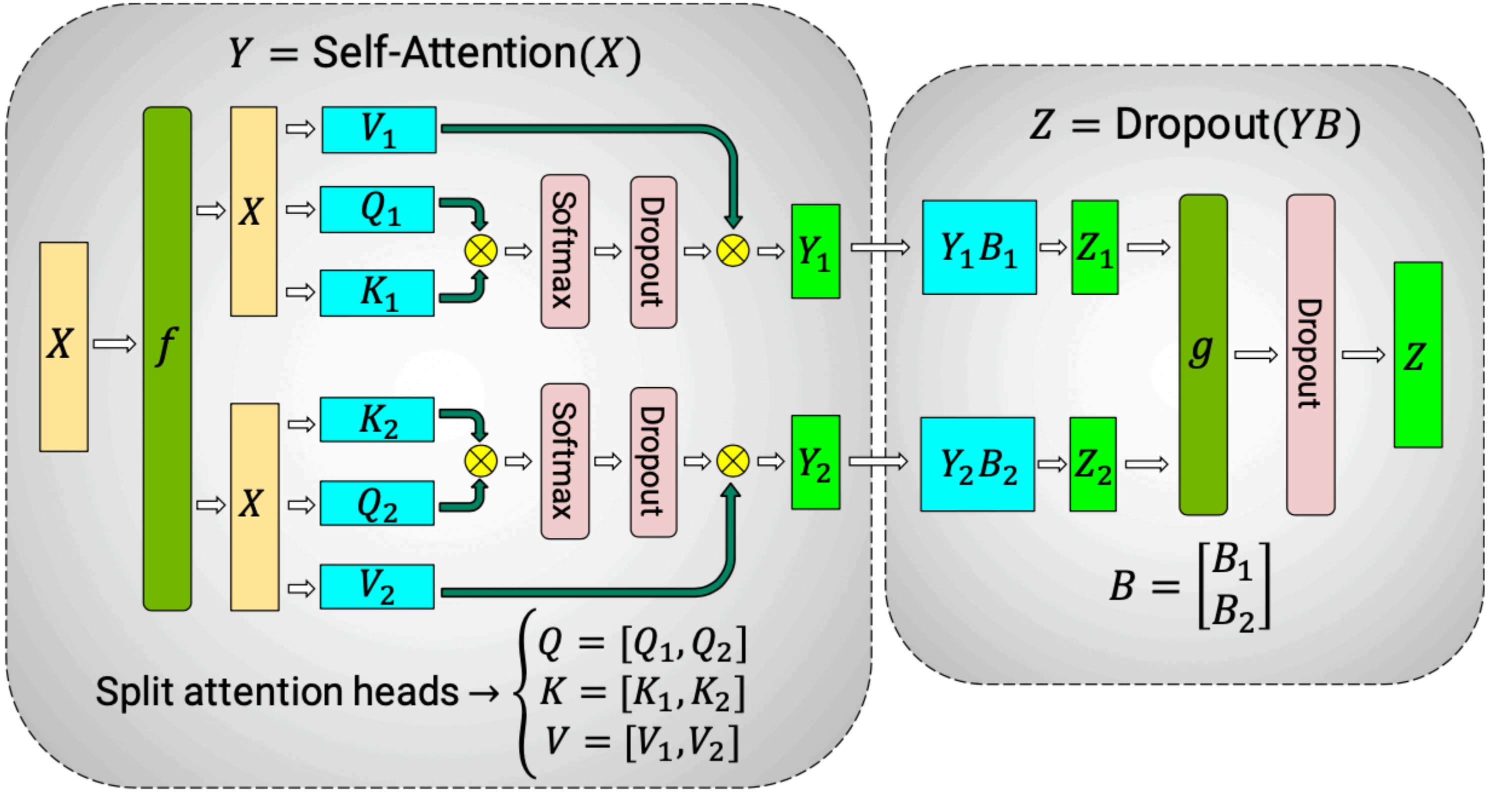
- Layer Norm 在没有开启 SP 的时候不需要并行
- 实际调用 TE 的 Nrom 算子进行加速



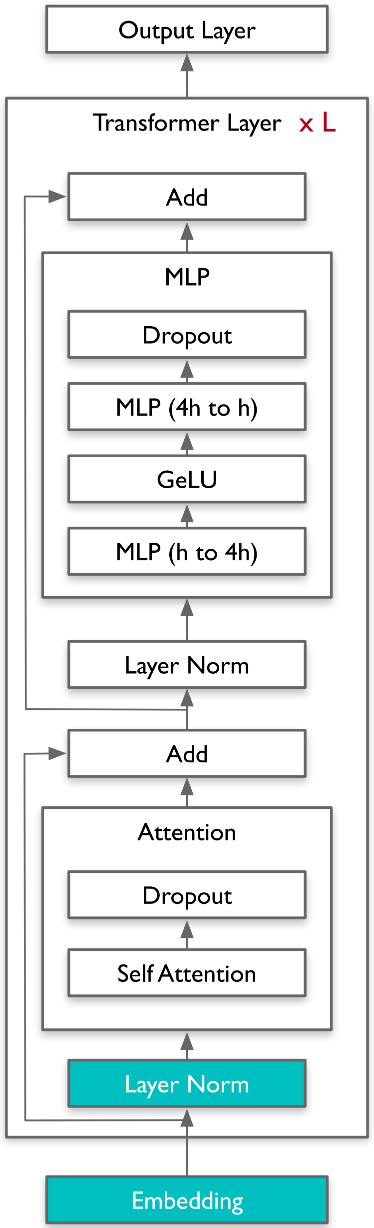
Megatron-LM 04

Attention 并行

Attention 并行



(b) Self-Attention.



Attention 并行

megatron > core > models > gpt > gpt_layer_specs.py > get_gpt_layer_local_spec

```python
def get_gpt_layer_local_spec(
    num_experts: int = None, moe_grouped_gemm: bool = False, qk_layer
) -> ModuleSpec:
    mlp = _get_mlp_module_spec(
        use_te=False, num_experts=num_experts, moe_grouped_gemm=moe_g
    )
    return ModuleSpec(
        module=TransformerLayer,
        submodules=TransformerLayerSubmodules(
            input_layernorm=FusedLayerNorm,
            self_attention=ModuleSpec(
                module=SelfAttention,
                params={"attn_mask_type": AttnMaskType.causal},
                submodules=SelfAttentionSubmodules(
                    linear_qkv=ColumnParallelLinear,
                    core_attention=DotProductAttention,
                    linear_proj=RowParallelLinear,
                    q_layernorm=FusedLayerNorm if qk_layernorm else I
                    k_layernorm=FusedLayerNorm if qk_layernorm else I
                ),
            ),
            self_attn_bda=get_bias_dropout_add,
            pre_mlp_layernorm=FusedLayerNorm,
            mlp=mlp,
            mlp_bda=get_bias_dropout_add,
```

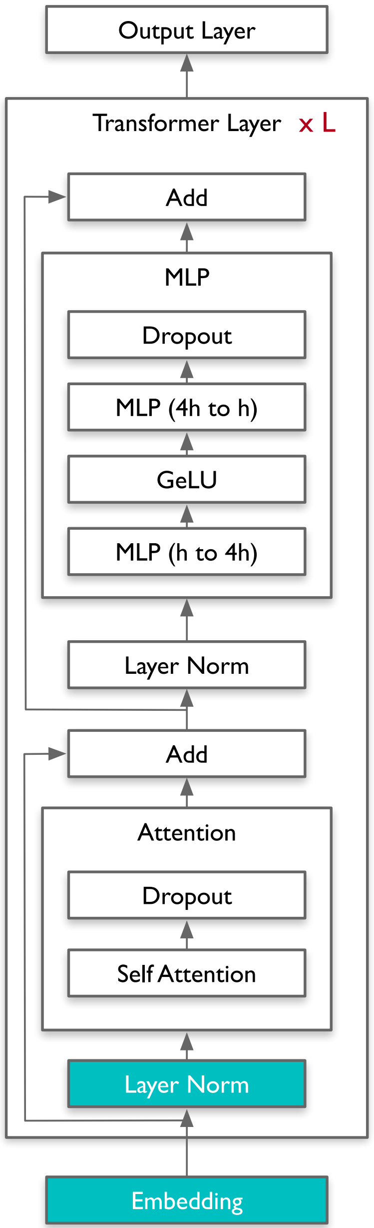
```python
class SelfAttention(Attention):
    """Self-attention layer class…

    def __init__(
        self,
        config: TransformerConfig,
        submodules: SelfAttentionSubmodules,
        layer_number: int,
        attn_mask_type=AttnMaskType.padding,
    ):
```

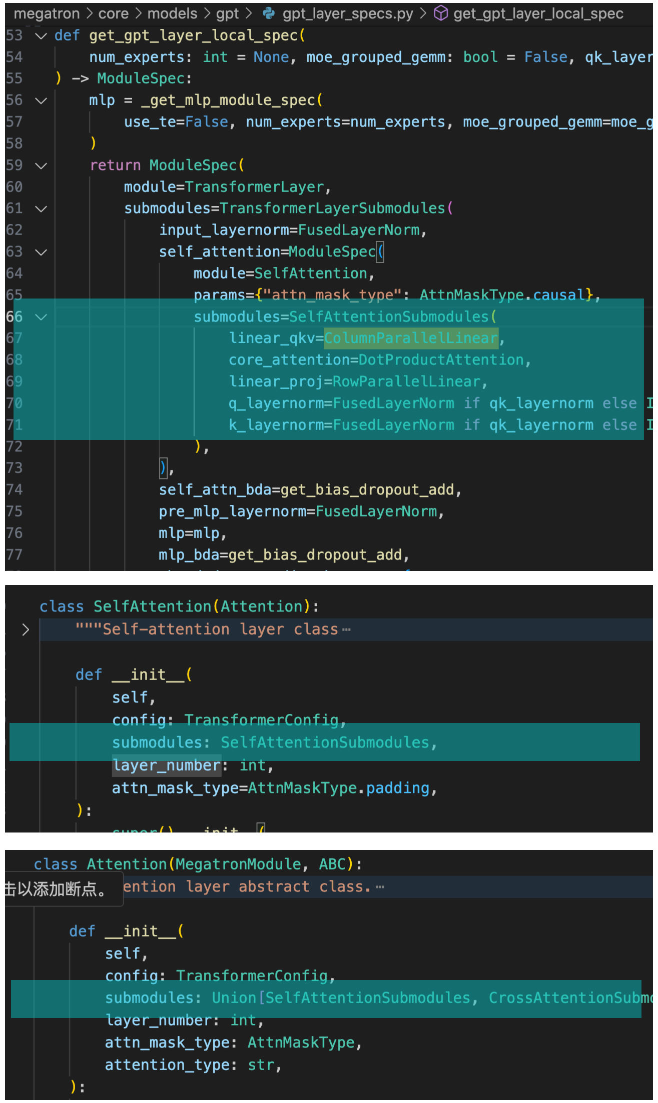
```python
class Attention(MegatronModule, ABC):
击以添加断点。ention layer abstract class.…

    def __init__(
        self,
        config: TransformerConfig,
        submodules: Union[SelfAttentionSubmodules, CrossAttentionSubm
        layer_number: int,
        attn_mask_type: AttnMaskType,
        attention_type: str,
    ):
```

- Attention 并行主要依赖模型参数
- 通过 get_gpt_layers_local_spec 注册模型结构
- SelfAttetnionSubmodules 类实现 SelfAttention

```bash
# Runs the "345M" parameter model
export CUDA_DEVICE_MAX_CONNECTIONS=1

CHECKPOINT_PATH=<Specify path>
VOCAB_FILE=<Specify path to file>/gpt2-vocab.json
MERGE_FILE=<Specify path to file>/gpt2-merges.txt
DATA_PATH=<Specify path and file prefix>_text_document

GPT_ARGS="
    --tensor-model-parallel-size 2 \
    --pipeline-model-parallel-size 1 \
    --num-layers 24 \
    --hidden-size 1024 \
    --num-attention-heads 16 \
    --seq-length 1024 \
    --max-position-embeddings 1024 \
    --voc 50000 \
    --micro-batch-size 4 \
    --global-batch-size 8 \
    --lr 0.00015 \
    --train-iters 500000 \
    --lr-decay-iters 320000 \
    --lr-decay-style cosine \
```

LayerNorm 并行





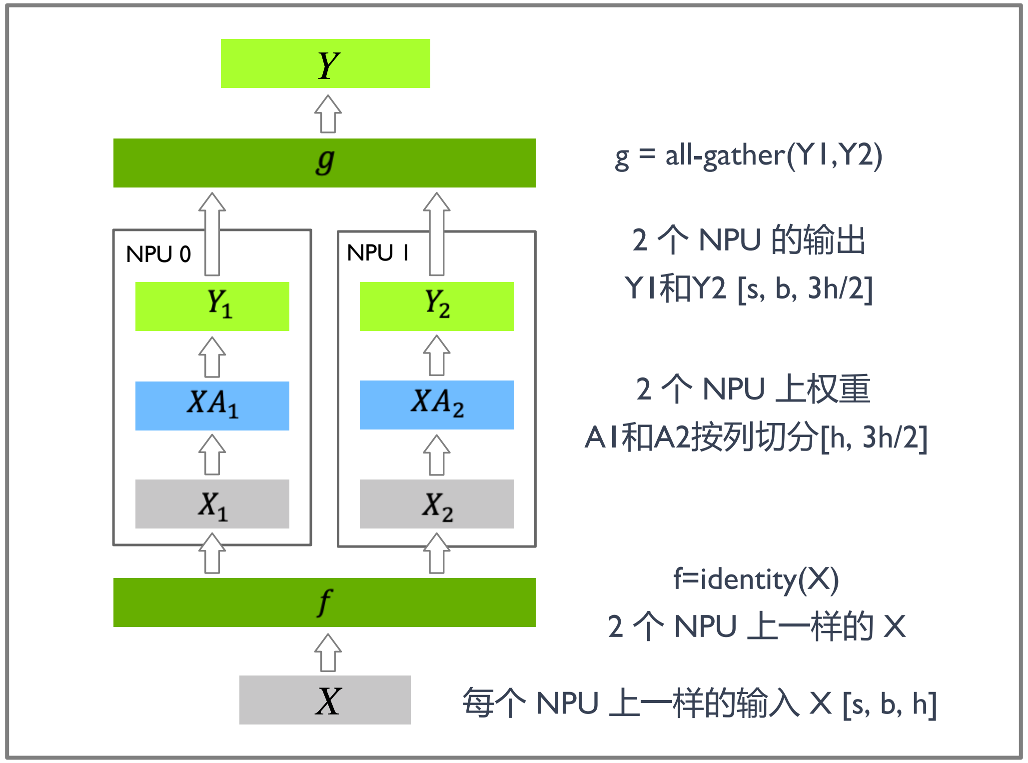
megatron > core > models > gpt > gpt_layer_specs.py > get_gpt_layer_local_spec

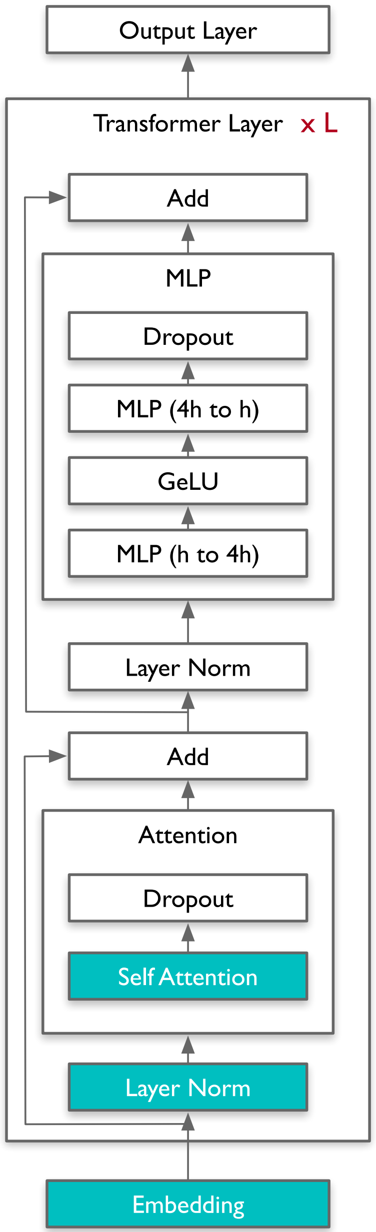
```python
def get_gpt_layer_local_spec(
    num_experts: int = None, moe_grouped_gemm: bool = False, qk_layer
) -> ModuleSpec:
    mlp = _get_mlp_module_spec(
        use_te=False, num_experts=num_experts, moe_grouped_gemm=moe_g
    )
    return ModuleSpec(
        module=TransformerLayer,
        submodules=TransformerLayerSubmodules(
            input_layernorm=FusedLayerNorm,
            self_attention=ModuleSpec(
                module=SelfAttention,
                params={"attn_mask_type": AttnMaskType.causal},
                submodules=SelfAttentionSubmodules(
                    linear_qkv=ColumnParallelLinear,
                    core_attention=DotProductAttention,
                    linear_proj=RowParallelLinear,
                    q_layernorm=FusedLayerNorm if qk_layernorm else I
                    k_layernorm=FusedLayerNorm if qk_layernorm else I
                ),
            ),
            self_attn_bda=get_bias_dropout_add,
            pre_mlp_layernorm=FusedLayerNorm,
            mlp=mlp,
            mlp_bda=get_bias_dropout_add,
```

```python
class SelfAttention(Attention):
    """Self-attention layer class…

    def __init__(
        self,
        config: TransformerConfig,
        submodules: SelfAttentionSubmodules,
        layer_number: int,
        attn_mask_type=AttnMaskType.padding,
    ):
```

```python
class Attention(MegatronModule, ABC):
    """Attention layer abstract class.…

    def __init__(
        self,
        config: TransformerConfig,
        submodules: Union[SelfAttentionSubmodules, CrossAttentionSubm
        layer_number: int,
        attn_mask_type: AttnMaskType,
        attention_type: str,
    ):
```

- ColumnParallelLinear 输入 [s, b, h] → [s, b, 3h]
- 将输出分割为 Q/K/V 3个矩阵后执行 Self Attetion 计算



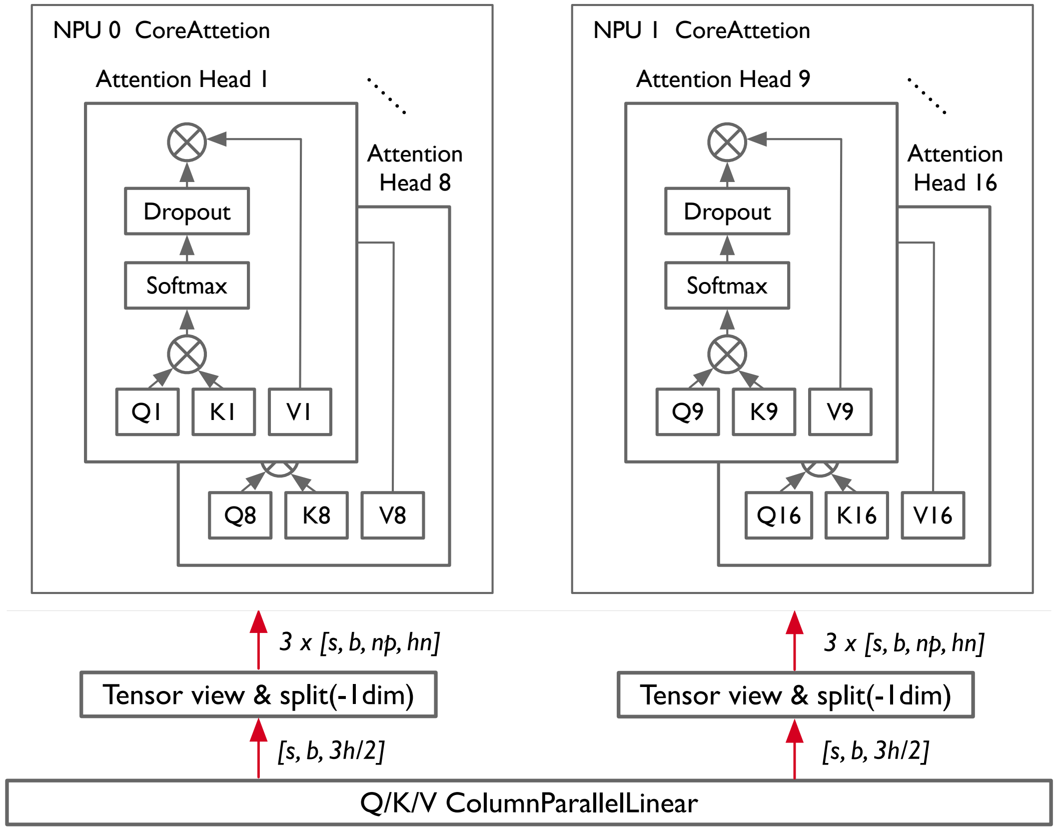


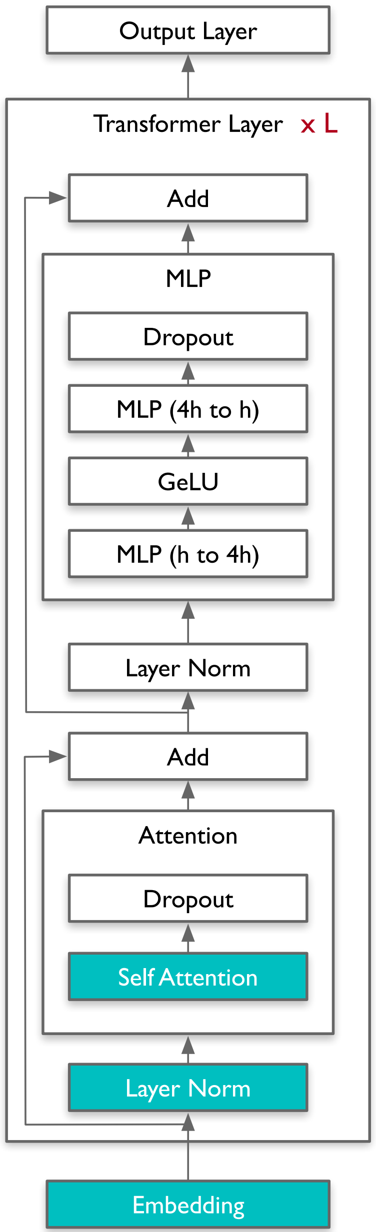
Attention 并行

```python
class SelfAttention(Attention):
    def get_query_key_value_tensors(self, hidden_states, key_value_states=None):
        if SplitAlongDim is not None:

            # [sq, b, ng, (np/ng + 2) * hn] --> [sq, b, ng, np/ng * hn], [sq, b, ng, hn], [
            (query, key, value) = SplitAlongDim(mixed_qkv, 3, split_arg_list,)
        else:
            # [sq, b, ng, (np/ng + 2) * hn] --> [sq, b, ng, np/ng * hn], [sq, b, ng, hn], [
            (query, key, value) = torch.split(mixed_qkv, split_arg_list, dim=3,)

        # [sq, b, ng, np/ng * hn] -> [sq, b, np, hn]
        query = query.reshape(query.size(0), query.size(1), -1, self.hidden_size_per_attent

        query = self.q_layernorm(query)
        key = self.k_layernorm(key)
```

```python
class Attention(MegatronModule, ABC):
    def forward(
        # ====================
        # Query, Key, and Value
        # ====================
        # Get the query, key and value tensors based on the type of attention -
        # self or cross attn.
        query, key, value = self.get_query_key_value_tensors(hidden_states, key_value_stat
```

- 每 NPU 将 ColumnParallelLinear 输出分割为 Q/K/V 3个矩阵
- 对该 NPU 上所有的 Head 合并执行 Q/K/V Attention 计算
- 将 Attention 按 Head 切分（16个 Head 切分到2个 NPU）
- ColumnParallelLinear 2 个 NPU 输出 [s, b, 3h/2] 作为输入



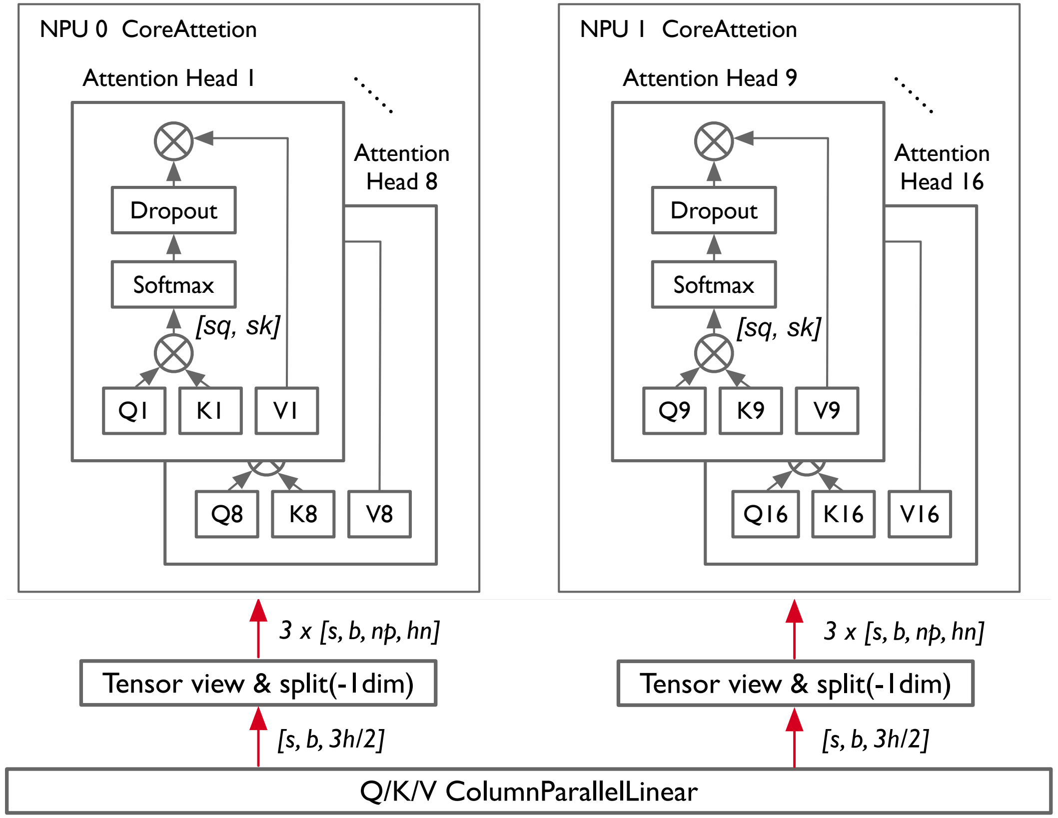


Attention 并行

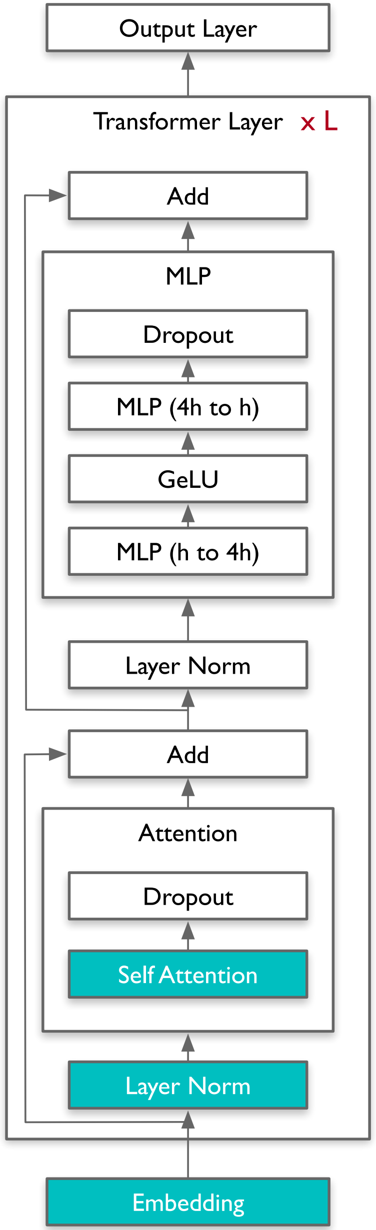
megatron > core > transformer > dot_product_attention.py > DotProductAttention > forward

```
class DotProductAttention(MegatronModule):
    def forward(
        # [b, np, sq, sk]
        output_size = (
            query.size(1),
            query.size(2),
            query.size(0),
            key.size(0),
        )

        # [sq, b, np, hn] -> [sq, b * np, hn]
        # This will be a simple view when doing normal attention, but in
        # the key and value tensors are repeated to match the queries so
        # to extract the queries.
        query = query.reshape(output_size[2], output_size[0] * output_siz
        # [sk, b, np, hn] -> [sk, b * np, hn]
        key = key.view(output_size[3], output_size[0] * output_size[1], -

        # preallocting input tensor: [b * np, sq, sk]
        matmul_input_buffer = parallel_state.get_global_memory_buffer().g
            (output_size[0] * output_size[1], output_size[2], output_size
        )

        # Raw attention scores. [b * np, sq, sk]
        matmul_result = torch.baddbmm(
            matmul_input_buffer,
            query.transpose(0, 1),  # [b * np, sq, hn]
```

- 利用 DotProuctAttention 每个 NPU 中所有的 Head 合并 Q&K [s, b, np, hn] 映射为 [s, b*np, hn]
- 计算 Q*K 得到每个 NPU 上的 Attention Scores [b*np, s, sk] 映射为 [b, np, s, sk]



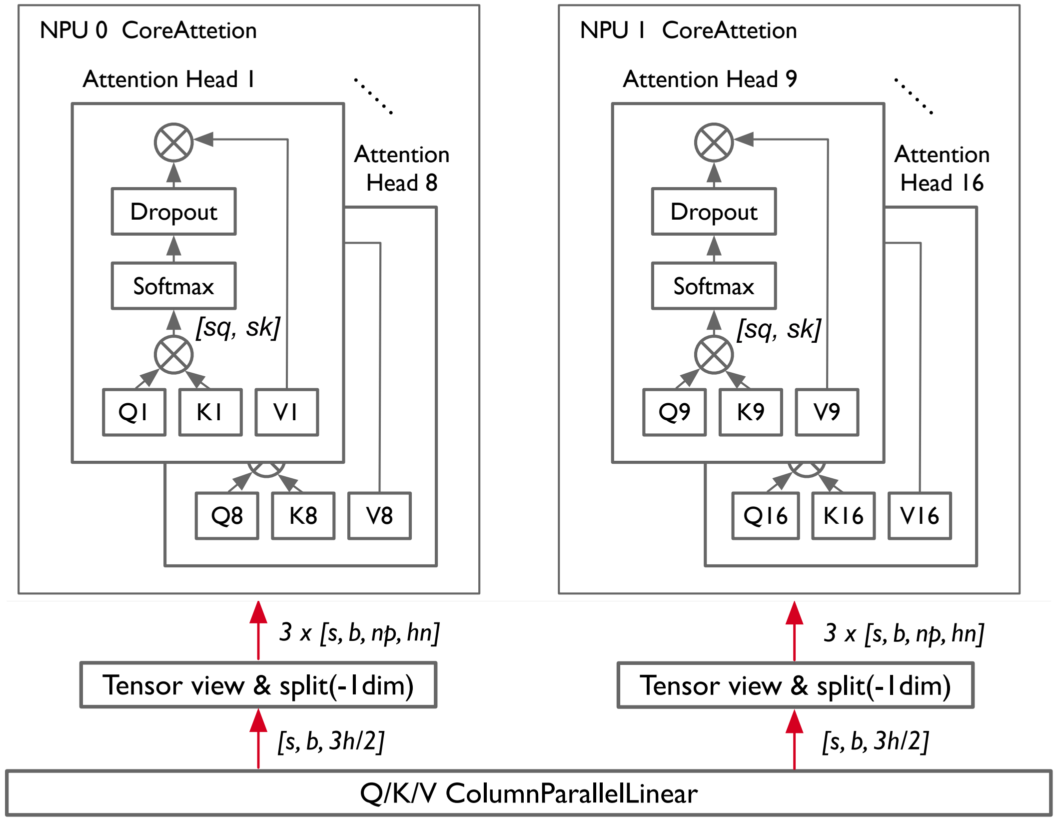
Attention 并行

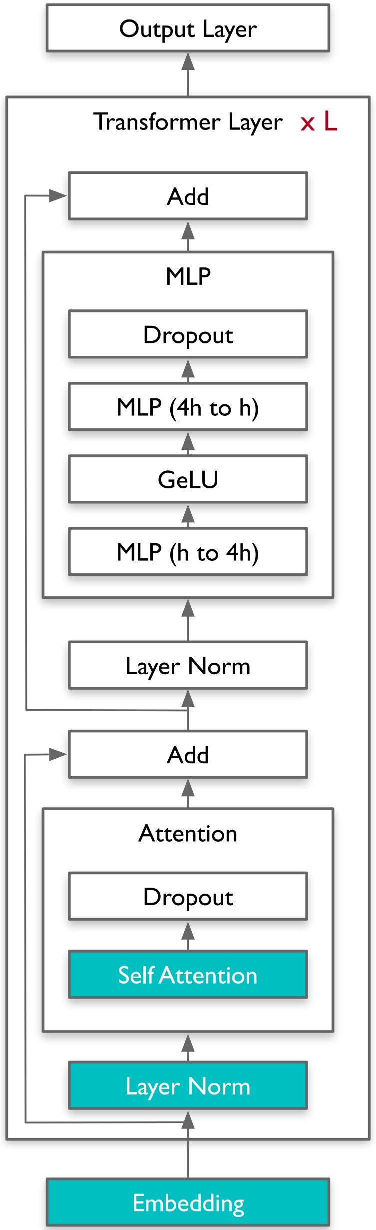


megatron > core > transformer > dot_product_attention.py > DotProductAttention > forward

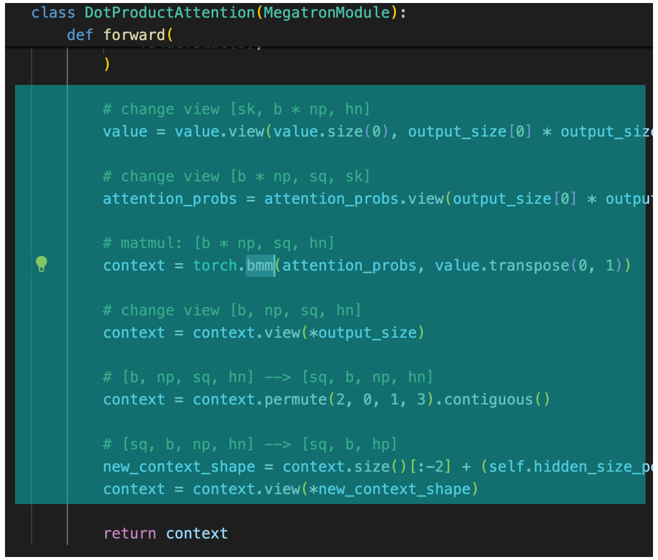
```python
class DotProductAttention(MegatronModule):
    def forward(

        # change view to [b, np, sq, sk]
        attention_scores = matmul_result.view(*output_size)

        # ===========================
        # Attention probs and dropout
        # ===========================

        # attention scores and attention mask [b, np, sq, sk]
        attention_probs: Tensor = self.scale_mask_softmax(attention_scores, a

        # This is actually dropping out entire tokens to attend to, which mig
        # seem a bit unusual, but is taken from the original Transformer pape

        if not self.config.sequence_parallel:
            with tensor_parallel.get_cuda_rng_tracker().fork():
                attention_probs = self.attention_dropout(attention_probs)
        else:
            attention_probs = self.attention_dropout(attention_probs)

        # =========================
        # Context layer. [sq, b, hp]
        # =========================
```

- 每个 NPU 上的 Attention Score [b, np, s, sk] 执行 scale_mask_softmax 和 dropout 得到 attention_probs [b, np, s, sk]
- V 映射 [n, np, s, hn] 后跟 attention_probs [b*np, s, sk] 执行计算 attention_probs*V



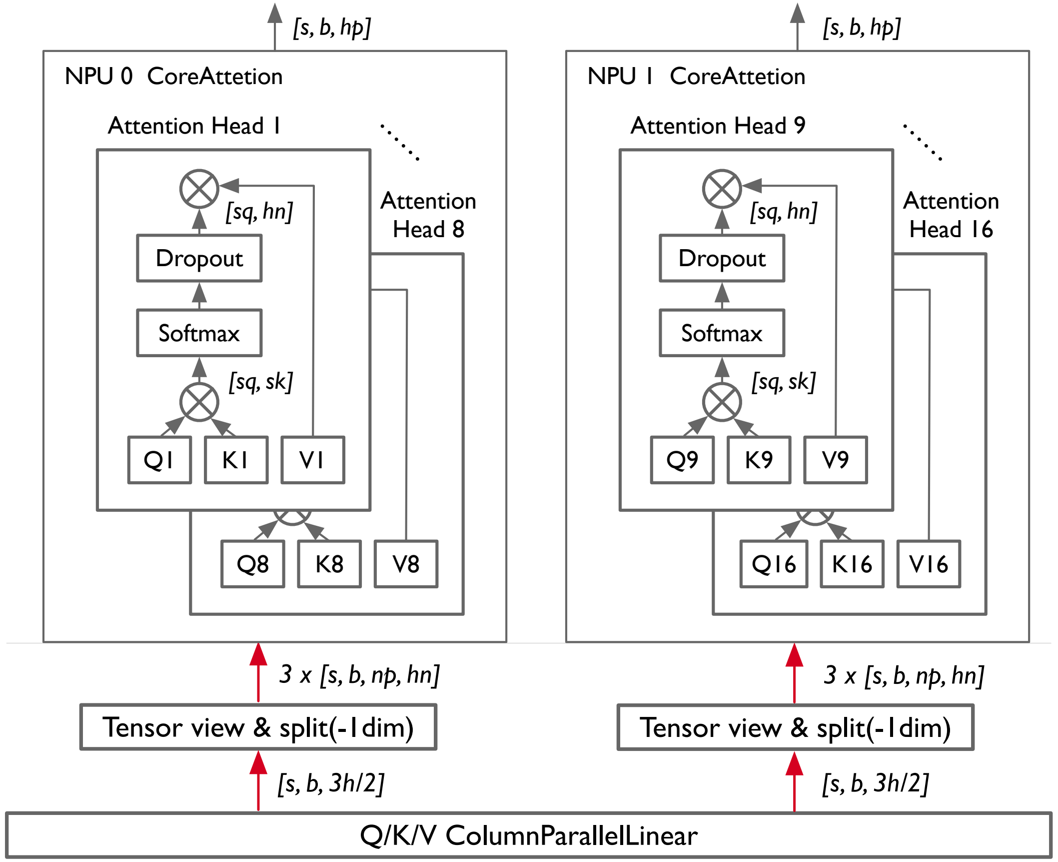


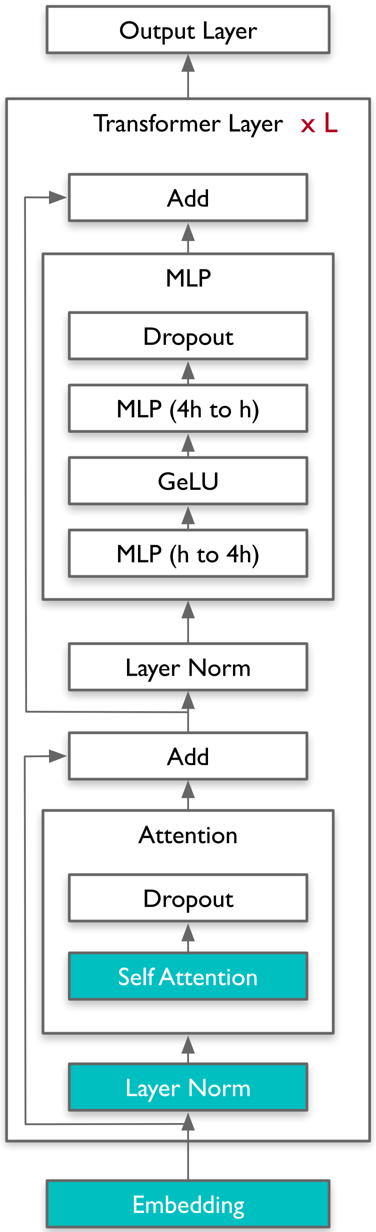
Attention 并行



```
class DotProductAttention(MegatronModule):
    def forward(
        )

        # change view [sk, b * np, hn]
        value = value.view(value.size(0), output_size[0] * output_siz

        # change view [b * np, sq, sk]
        attention_probs = attention_probs.view(output_size[0] * outpu

        # matmul: [b * np, sq, hn]
        context = torch.bmm(attention_probs, value.transpose(0, 1))

        # change view [b, np, sq, hn]
        context = context.view(*output_size)

        # [b, np, sq, hn] --> [sq, b, np, hn]
        context = context.permute(2, 0, 1, 3).contiguous()

        # [sq, b, np, hn] --> [sq, b, hp]
        new_context_shape = context.size()[:-2] + (self.hidden_size_p
        context = context.view(*new_context_shape)

        return context
```

- 计算出每个 NPU 上的 attention_probs * V，然后变换 Shape [s, b, np, hn] → [s, b, np, hn] → 最终得到每个 NPU 上的 Core Attetion 结果 [s, b, hp]

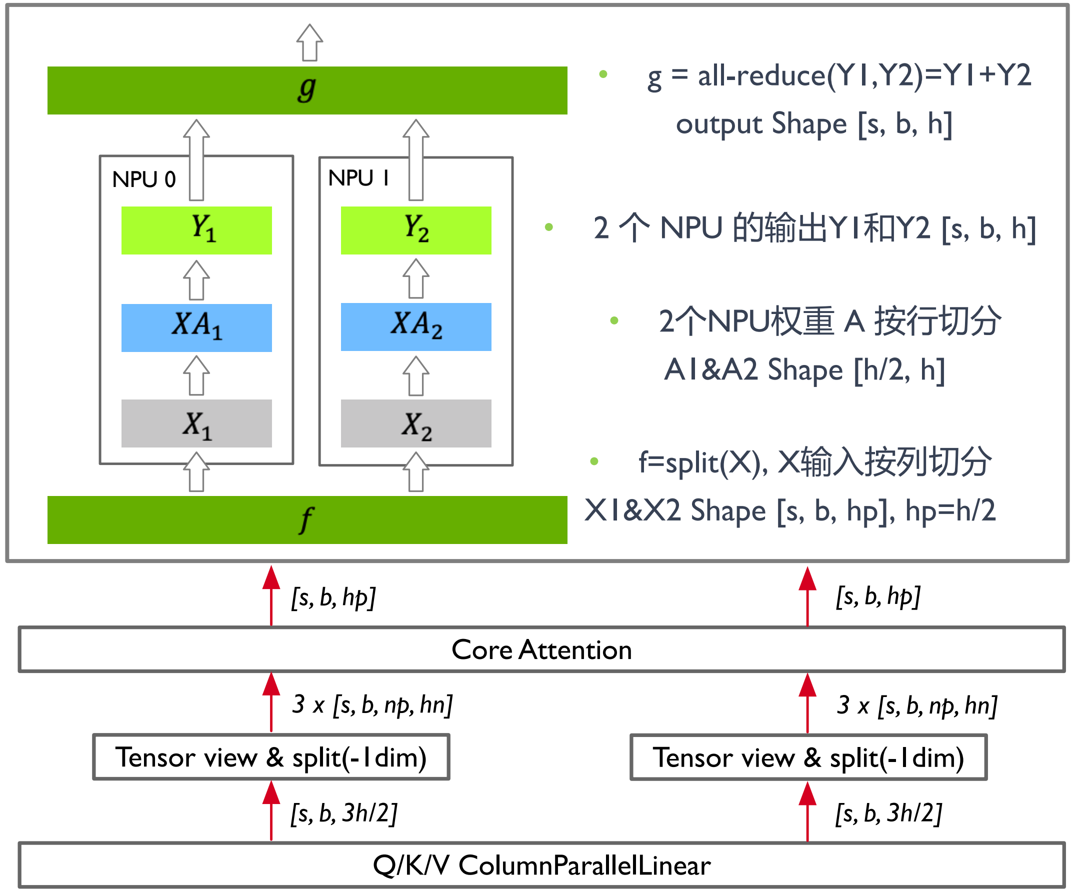




Attention 并行

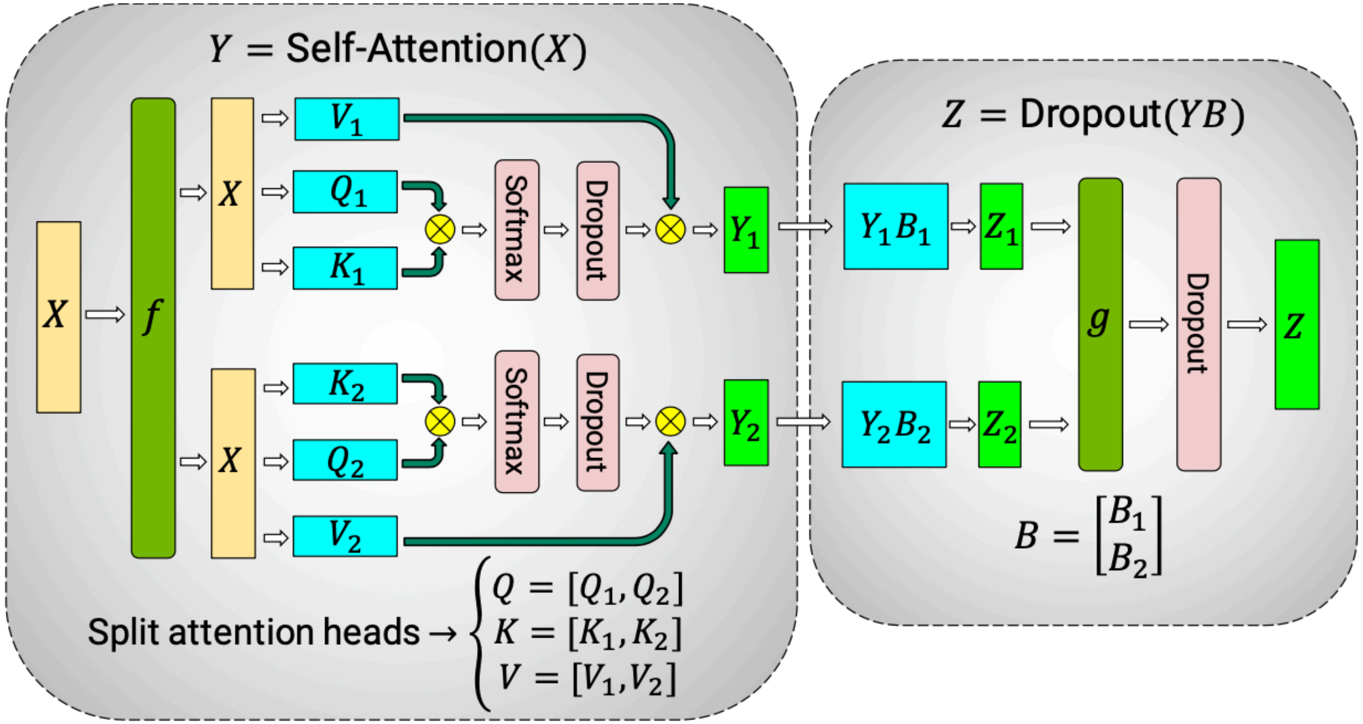
```
megatron > core > inference > gpt > model_specs.py > get_gpt_layer_ammo_spec
def get_gpt_layer_ammo_spec() -> ModuleSpec:
    """Mix the native spec with TENorm.

    This is essentially the native local spec except for the layernorm
    is using TENorm from Transformer-Engine. This TENorm supports both
    prevents the apex dependency.
    """
    return ModuleSpec(
        module=TransformerLayer,
        submodules=TransformerLayerSubmodules(
            input_layernorm=TENorm,
            self_attention=ModuleSpec(
                module=SelfAttention,
                params={"attn_mask_type": AttnMaskType.causal},
                submodules=SelfAttentionSubmodules(
                    linear_qkv=ColumnParallelLinear,
                    core_attention=DotProductAttention,
                    linear_proj=RowParallelLinear,
                ),
            ),
            self_attn_bda=get_bias_dropout_add,
            pre_mlp_layernorm=TENorm,
            mlp=ModuleSpec(
                module=MLP,
                submodules=MLPSubmodules(
                    linear_fc1=ColumnParallelLinear, linear_fc2=RowPar
```

- Self Attetion 中 RowParallelLinear Input [s, b, hp] → Output [s, b, h]



Attention 并行：行切分与列切分

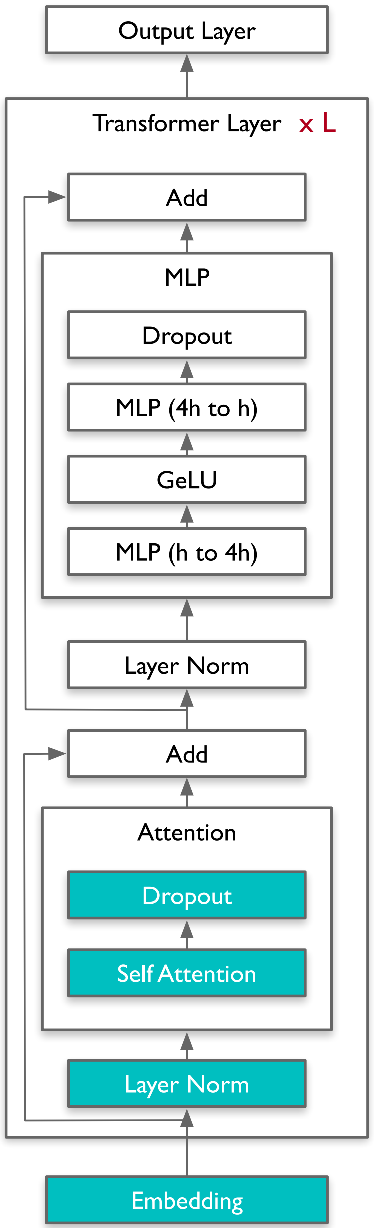
- ColumnParallelLinear 实现了 Attention 的前半部分或者考虑了这个线性层独立使用的情况；
- RowParallelLinear 实现了 Attention 的后半部分或者考虑了这个线性层独立使用的情况；



(b) Self-Attention.

Megatron-LM 05

Add & Norm 结构

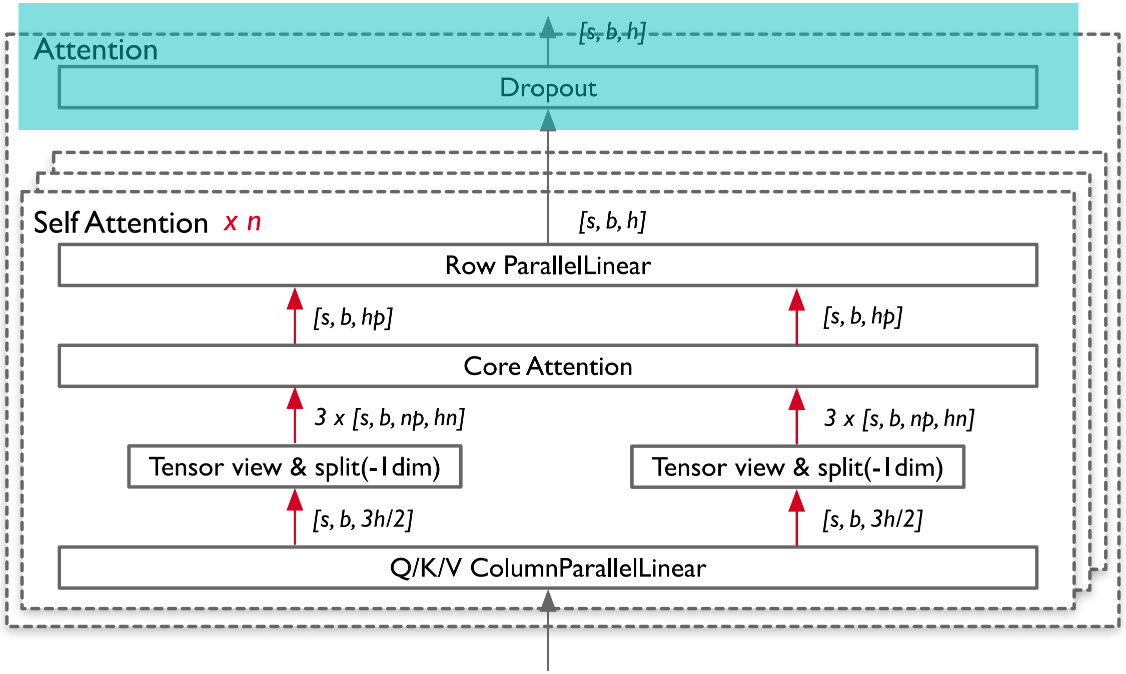


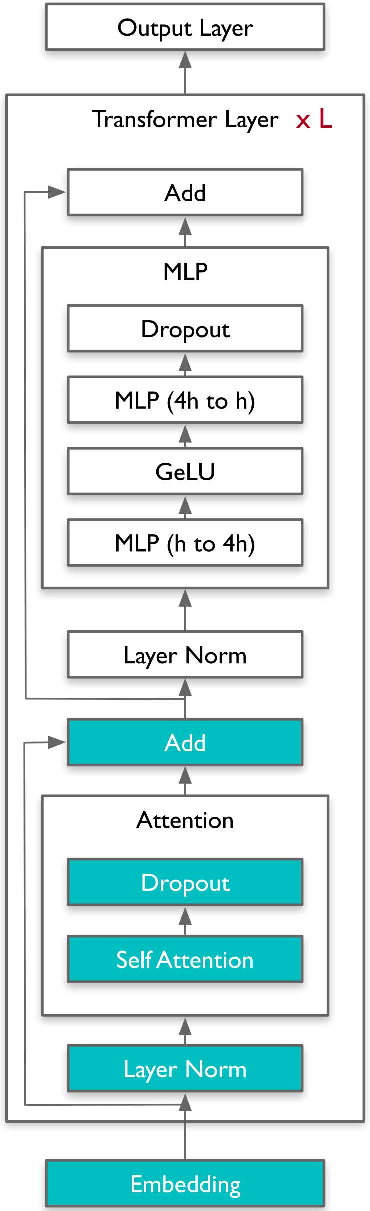
Attention 并行

```
def get_gpt_layer_ammo_spec() -> ModuleSpec:
    """Mix the native spec with TENorm.

    This is essentially the native local spec except for the layernorm
    is using TENorm from Transformer-Engine. This TENorm supports both
    prevents the apex dependency.
    """
    return ModuleSpec(
        module=TransformerLayer,
        submodules=TransformerLayerSubmodules(
            input_layernorm=TENorm,
            self_attention=ModuleSpec(
                module=SelfAttention,
                params={"attn_mask_type": AttnMaskType.causal},
                submodules=SelfAttentionSubmodules(
                    linear_qkv=ColumnParallelLinear,
                    core_attention=DotProductAttention,
                    linear_proj=RowParallelLinear,
                ),
            ),
            self_attn_bda=get_bias_dropout_add,
            pre_mlp_layernorm=TENorm,
            mlp=ModuleSpec(
                module=MLP,
                submodules=MLPSubmodules(
                    linear_fc1=ColumnParallelLinear, linear_fc2=RowPar
```

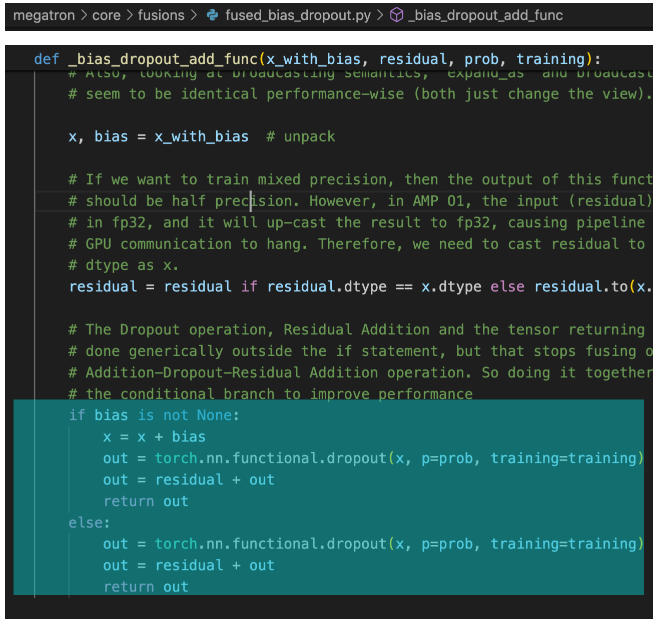
```
def get_bias_dropout_add(training, fused):
    if fused:
        # jit scripting for a nn.module (with dropout) is not
        # triggering the fusion kernel. For now, we use two
        # different nn.functional routines to account for varying
        # dropout semantics during training and inference phases.
        if training:
            return bias_dropout_add_fused_train
        else:
            return bias_dropout_add_fused_inference
    else:
        return bias_dropout_add_unfused(training)
```

- Self Attention 输出 [s, b, h] 作为 Dropout 输入
- Dropout 输出[s, b, h] 作为 Attention 的整体输出
- 实际代码使用 bias_dropout_add_fused 融合算子





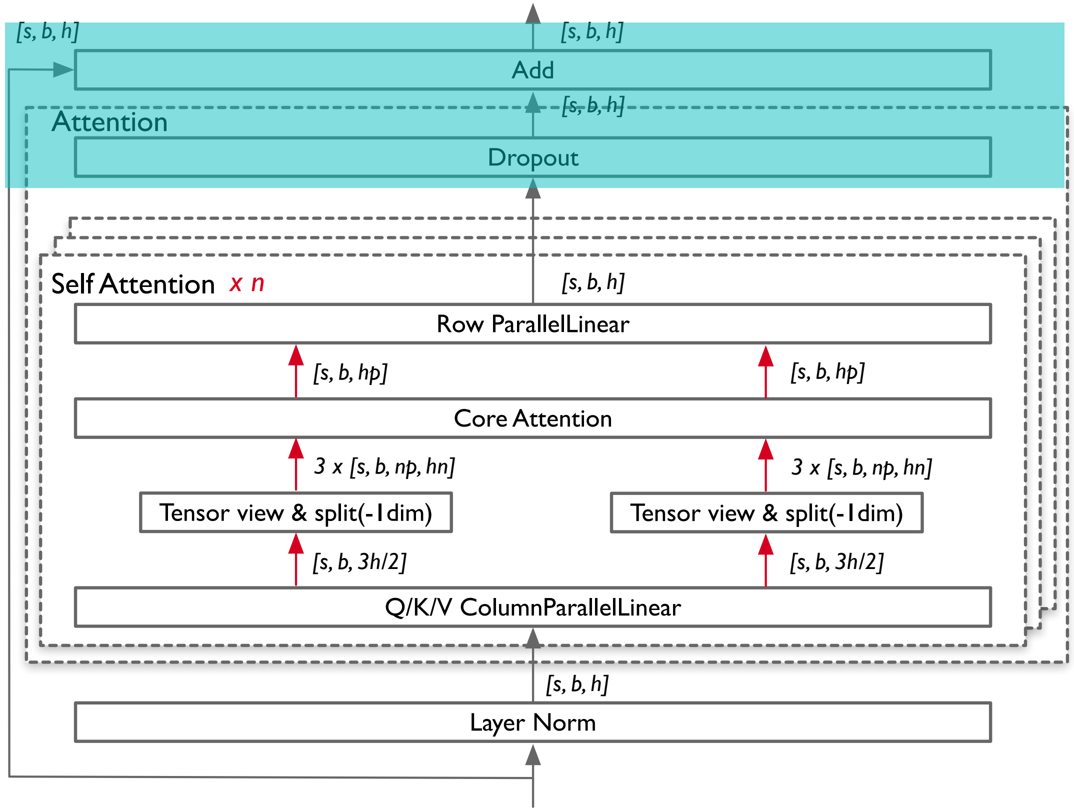
Add & Layer Norm



megatron > core > fusions > fused_bias_dropout.py > _bias_dropout_add_func

```python
def _bias_dropout_add_func(x_with_bias, residual, prob, training):
    # seem to be identical performance-wise (both just change the view).

    x, bias = x_with_bias  # unpack

    # If we want to train mixed precision, then the output of this funct
    # should be half precision. However, in AMP O1, the input (residual)
    # in fp32, and it will up-cast the result to fp32, causing pipeline
    # GPU communication to hang. Therefore, we need to cast residual to
    # dtype as x.
    residual = residual if residual.dtype == x.dtype else residual.to(x.

    # The Dropout operation, Residual Addition and the tensor returning
    # done generically outside the if statement, but that stops fusing o
    # Addition-Dropout-Residual Addition operation. So doing it together
    # the conditional branch to improve performance
    if bias is not None:
        x = x + bias
        out = torch.nn.functional.dropout(x, p=prob, training=training)
        out = residual + out
        return out
    else:
        out = torch.nn.functional.dropout(x, p=prob, training=training)
        out = residual + out
        return out
```

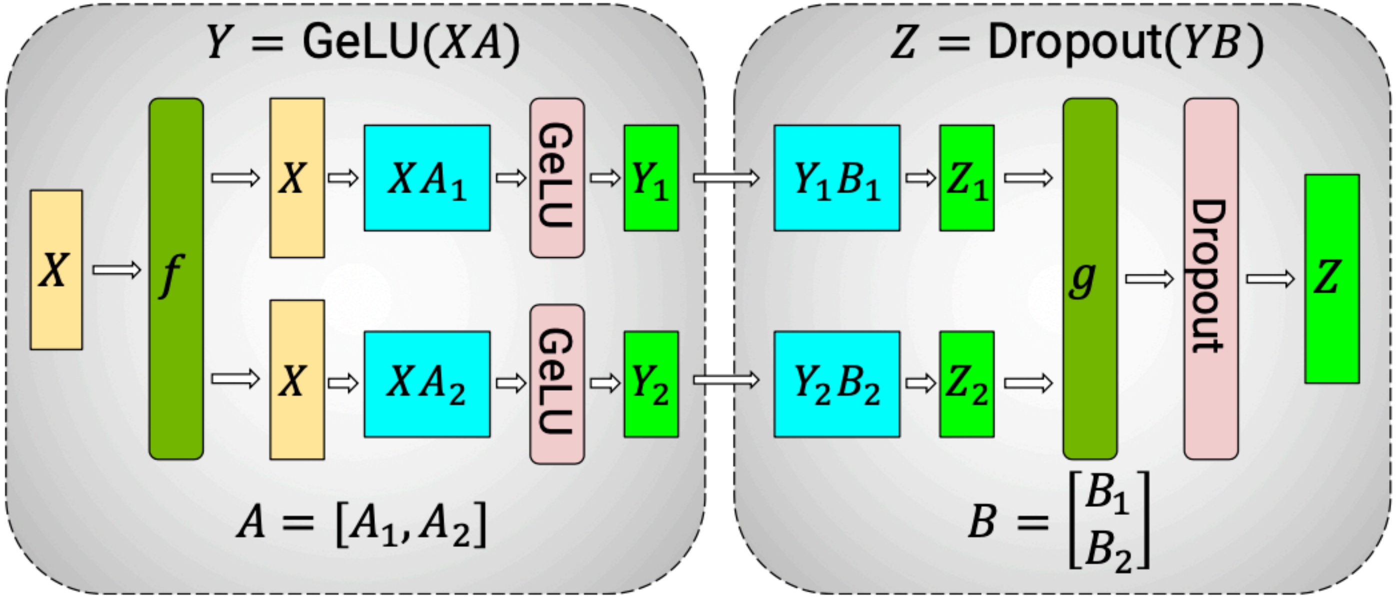
- 执行残差 residual 连接作为 Layer Norm 的输入 [s, b, h]



Megatron-LM 06

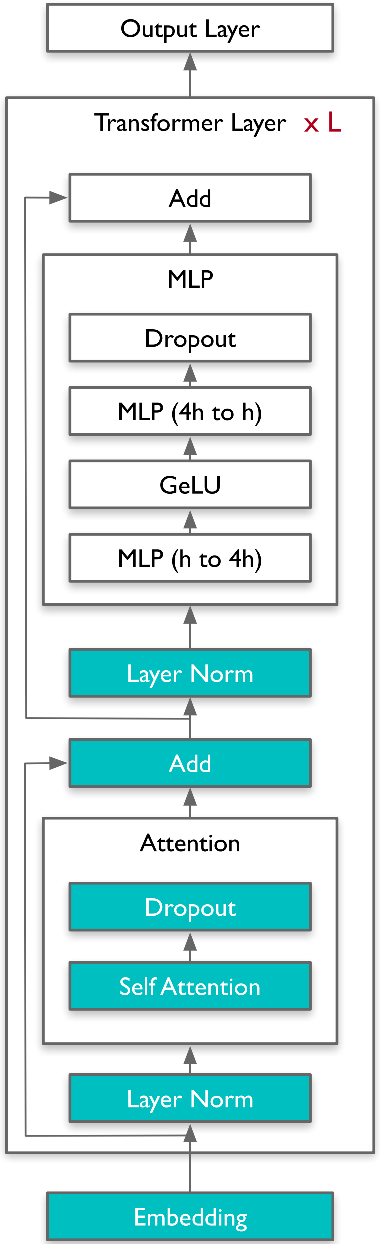
MLP 并行

MLP并行



$Y = \text{GeLU}(XA)$
$Z = \text{Dropout}(YB)$
X
f
X
XA_1
GeLU
Y_1
X
XA_2
GeLU
Y_2
Y_1B_1
Z_1
Y_2B_2
Z_2
g
Dropout
Z
$A = [A_1, A_2]$
$B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$

(a) MLP.



Add & Layer Norm

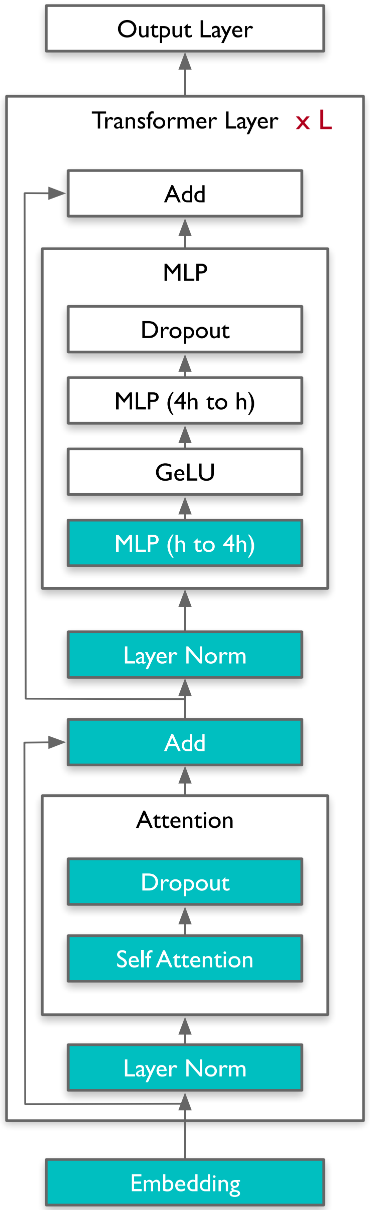
```python
def get_gpt_layer_ammo_spec() -> ModuleSpec:
                ),
            ),
            self_attn_bda=get_bias_dropout_add,
            pre_mlp_layernorm=TENorm,
            mlp=ModuleSpec(
                module=MLP,
                submodules=MLPSubmodules(
                    linear_fc1=ColumnParallelLinear, linear_fc2=RowParallelLinear,
                ),
            ),
            mlp_bda=get_bias_dropout_add,
            # Map TE-layernorm-fusion keys back
            sharded_state_dict_keys_map={
                'input_layernorm.': 'self_attention.linear_qkv.layer_norm_',
                'pre_mlp_layernorm.': 'mlp.linear_fc1.layer_norm_',
            },
        ),
    )
```

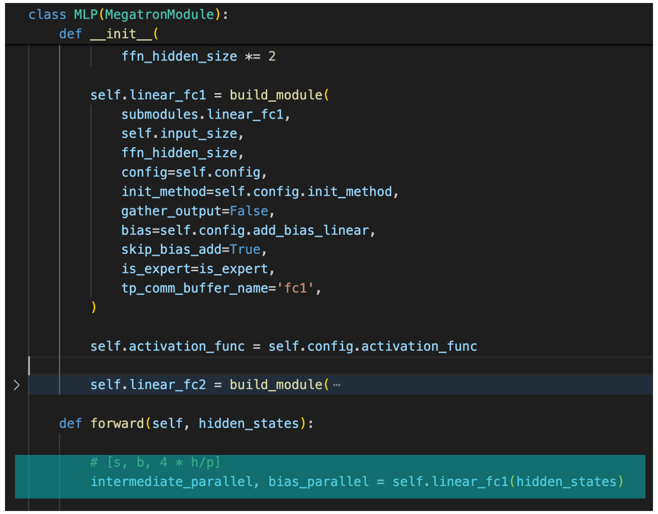
megatron > core > transformer > mlp.py > MLPSubmodules

```python
class MLP(MegatronModule):
    """
    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.

    Returns an output and a bias to be added to the output.
    If config.add_bias_linear is False, the bias returned is None.

    We use the following notation:
     h: hidden size
     p: number of tensor model parallel partitions
     b: batch size
     s: sequence length
    """

    def __init__(
        self,
        config: TransformerConfig,
        submodules: MLPSubmodules,
        is_expert: bool = False,
        input_size: int = None,
    ):
        super().__init__(config=config)
```

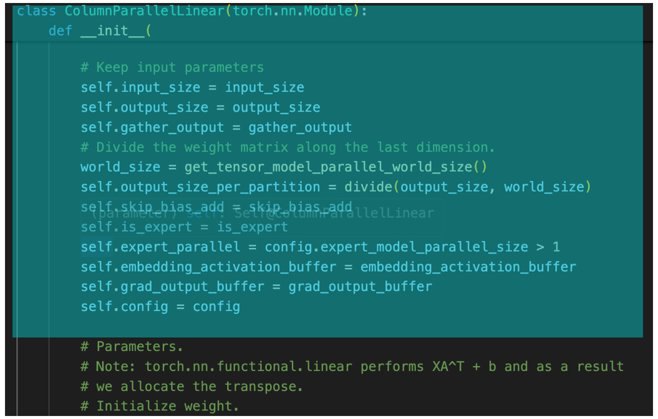
- 通过 MLPSubmodules 类注册到 get_gpt_layer_ammo_spec()
- 构建 NPU 并行计算的 PrarallelMLP，其中一个行并行一个列并行

MLP并行



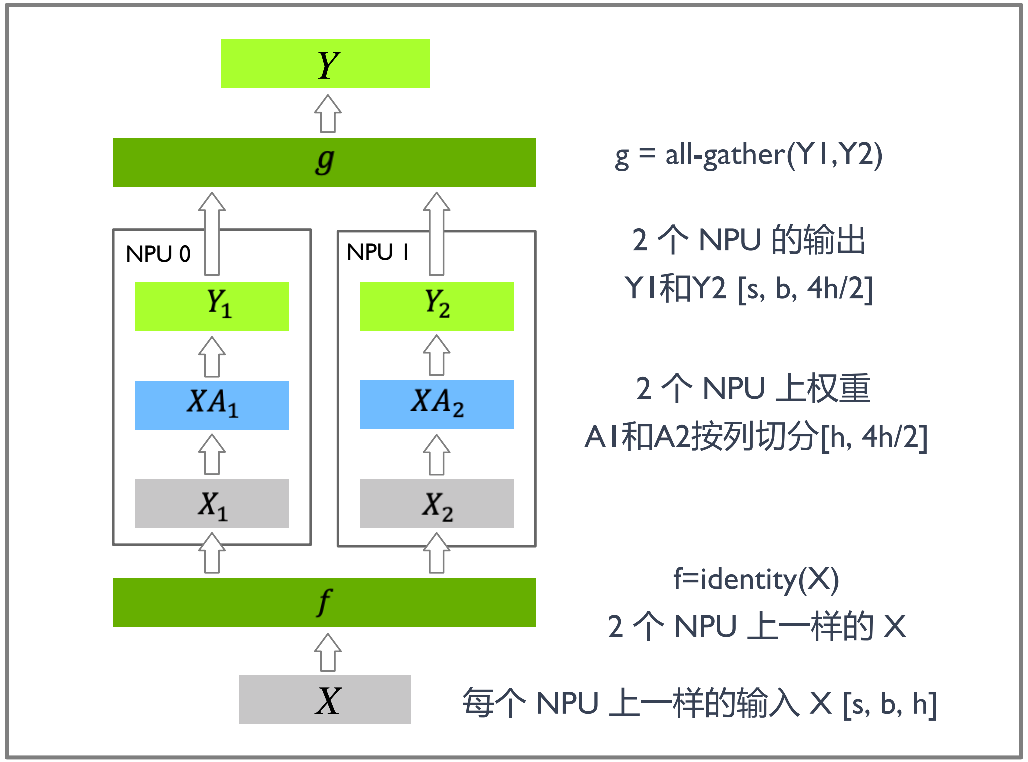


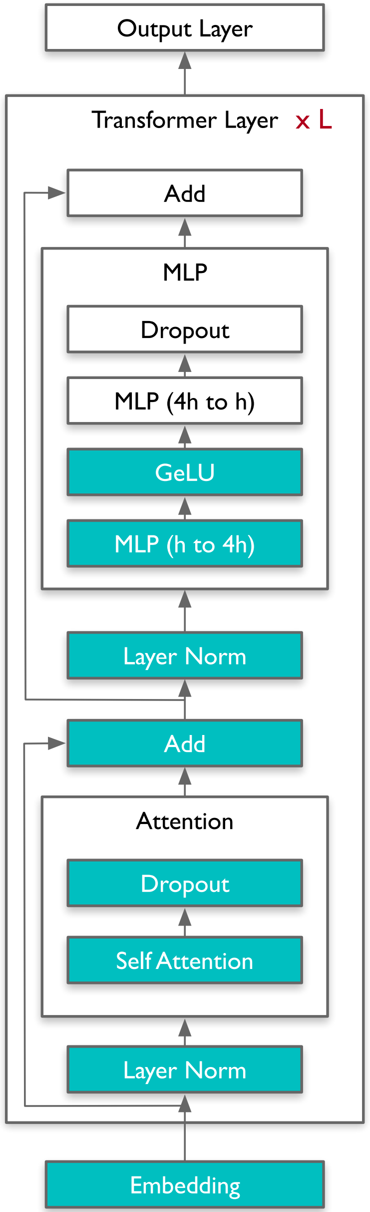
```python
class MLP(MegatronModule):
    def __init__(
            ffn_hidden_size *= 2

        self.linear_fc1 = build_module(
            submodules.linear_fc1,
            self.input_size,
            ffn_hidden_size,
            config=self.config,
            init_method=self.config.init_method,
            gather_output=False,
            bias=self.config.add_bias_linear,
            skip_bias_add=True,
            is_expert=is_expert,
            tp_comm_buffer_name='fc1',
        )

        self.activation_func = self.config.activation_func

        self.linear_fc2 = build_module(…

    def forward(self, hidden_states):

        # [s, b, 4 * h/p]
        intermediate_parallel, bias_parallel = self.linear_fc1(hidden_states)
```



```python
class ColumnParallelLinear(torch.nn.Module):
    def __init__(

        # Keep input parameters
        self.input_size = input_size
        self.output_size = output_size
        self.gather_output = gather_output
        # Divide the weight matrix along the last dimension.
        world_size = get_tensor_model_parallel_world_size()
        self.output_size_per_partition = divide(output_size, world_size)
        self.skip_bias_add = skip_bias_add
        self.is_expert = is_expert
        self.expert_parallel = config.expert_model_parallel_size > 1
        self.embedding_activation_buffer = embedding_activation_buffer
        self.grad_output_buffer = grad_output_buffer
        self.config = config

        # Parameters.
        # Note: torch.nn.functional.linear performs XA^T + b and as a result
        # we allocate the transpose.
        # Initialize weight.
```

- 构建 GPU 并行计算的 Parallel MLP， ColumnParallelLinear：
- 构建 MLP 输入 [s, b, h] → 输出 [s, b, 4h]

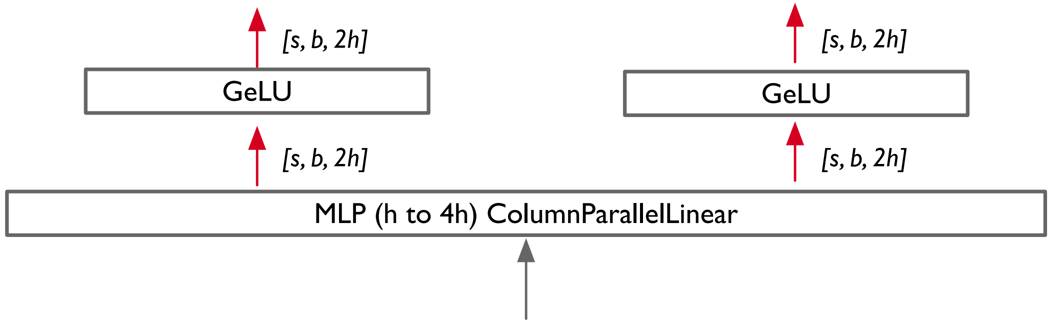




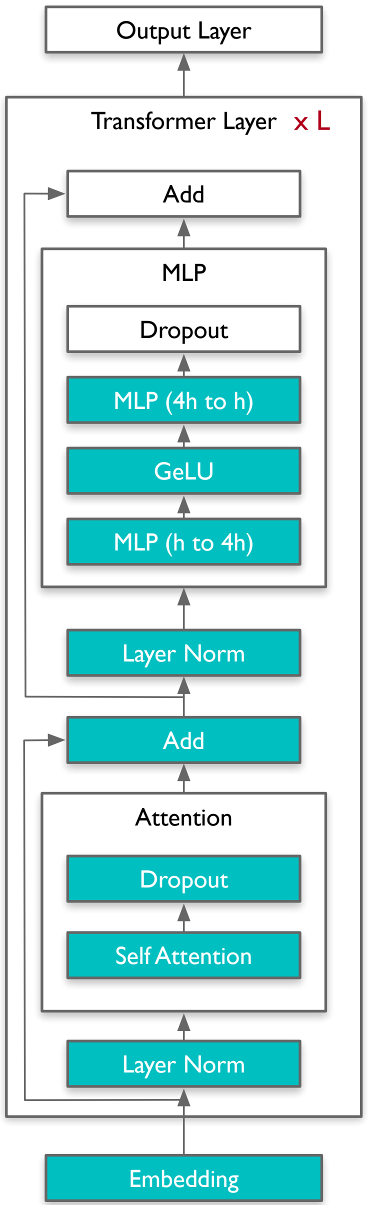
MLP并行

```
class MLP(MegatronModule):
    def forward(self, hidden_states):
        if self.config.bias_activation_fusion:
            if self.activation_func == F.gelu:
                if self.config.gated_linear_unit:
                    intermediate_parallel = bias_geglu_impl(intermediate_parallel, bia
                else:
                    assert self.config.add_bias_linear is True
                    intermediate_parallel = bias_gelu_impl(intermediate_parallel, bias
            elif self.activation_func == F.silu and self.config.gated_linear_unit:
                intermediate_parallel = bias_swiglu_impl(intermediate_parallel, bias_p
            else:
                raise ValueError("Only support fusion of gelu and swiglu")
        else:
            if bias_parallel is not None:
                intermediate_parallel = intermediate_parallel + bias_parallel
            if self.config.gated_linear_unit:

                def glu(x):
                    x = torch.chunk(x, 2, dim=-1)
                    return self.config.activation_func(x[0]) * x[1]

                intermediate_parallel = glu(intermediate_parallel)
            else:
                intermediate_parallel = self.activation_func(intermediate_parallel)
```

- 每个 NPU 对各自的输入 [s, b, 2h] 执行计算，输出 [s, b, 2h]



MLP并行

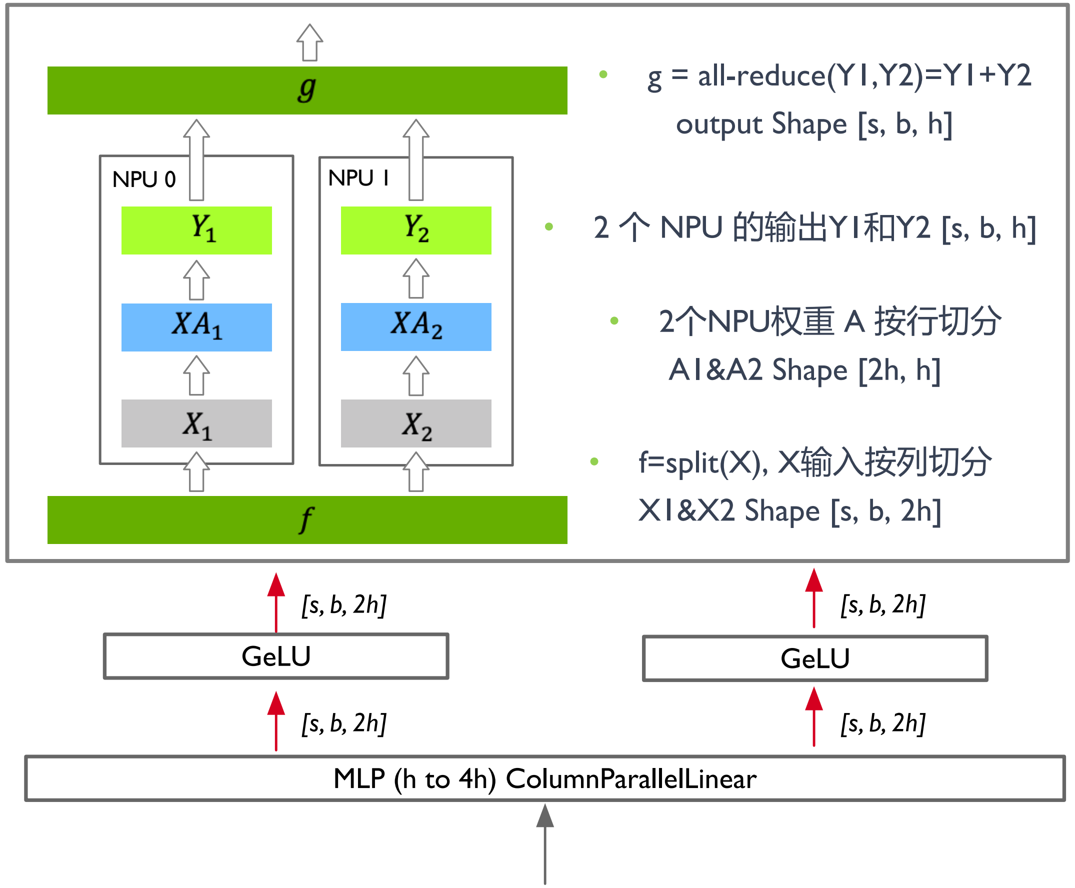



```
def get_gpt_layer_ammo_spec() -> ModuleSpec:
            self_attn_bda=get_bias_dropout_add,
            pre_mlp_layernorm=TENorm,
            mlp=ModuleSpec(
                module=MLP,
                submodules=MLPSubmodules(
                    linear_fc1=ColumnParallelLinear, linear_fc2=RowParallelLinear,
                ),
            ),
```

```
class MLP(MegatronModule):
    def forward(self, hidden_states):

        # [s, b, h]
        output, output_bias = self.linear_fc2(intermediate_parallel)

        return output, output_bias
```

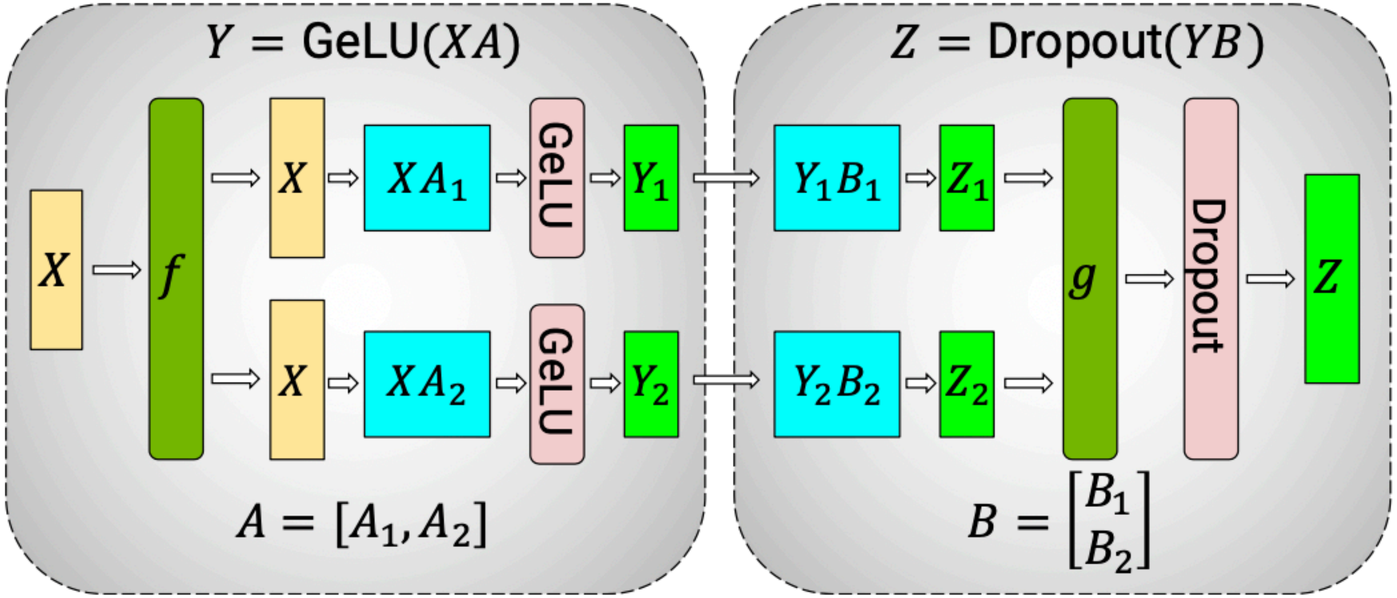
```
class MLP(MegatronModule):
    def __init__(

        self.linear_fc2 = build_module(
            submodules.linear_fc2,
            self.config.ffn_hidden_size,
            self.config.hidden_size,
            config=self.config,
            init_method=self.config.output_layer_init_method,
            bias=self.config.add_bias_linear,
            input_is_parallel=True,
            skip_bias_add=True,
            is_expert=is_expert,
            tp_comm_buffer_name='fc2',
        )
```

- MLP 4h→ h RowParallelLinear 输入 [s,b, 4h] 输出 [s, b, h]



MLP并行：行切分与列切分

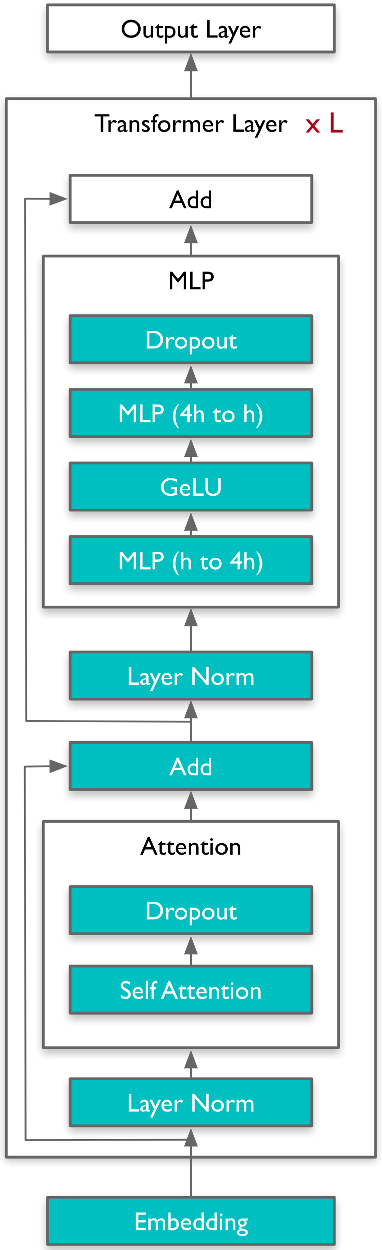
- ColumnParallelLinear 实现了 MLP 的前半部分或者考虑了这个线性层独立使用的情况。
- RowParallelLinear 实现了 MLP 的后半部分或者考虑了这个线性层独立使用的情况。



(a) MLP.

Megatron-LM 07

Add 结构

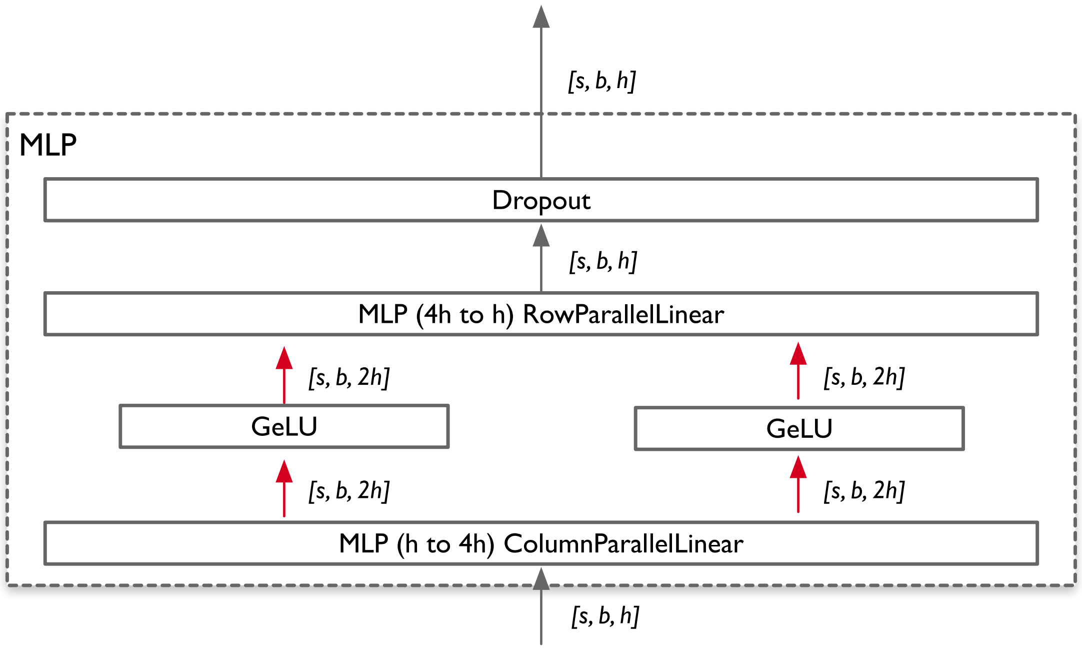


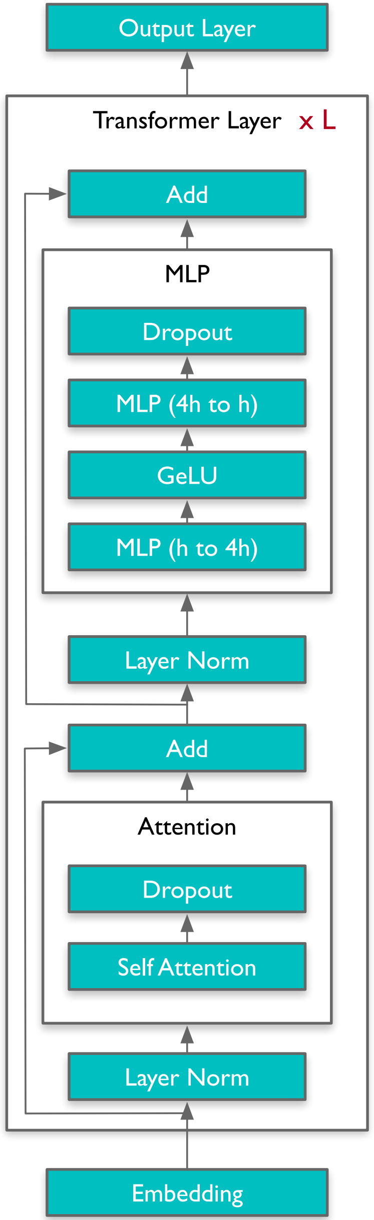
MLP并行

```python
def get_gpt_layer_ammo_spec() -> ModuleSpec:
            self_attn_bda=get_bias_dropout_add,
            pre_mlp_layernorm=TENorm,
            mlp=ModuleSpec(
                module=MLP,
                submodules=MLPSubmodules(
                    linear_fc1=ColumnParallelLinear, linear_fc2=RowParallelLinear,
                ),
            ),
            mlp_bda=get_bias_dropout_add,
            # Map TE-layernorm-fusion keys back
            sharded_state_dict_keys_map={
                'input_layernorm.': 'self_attention.linear_qkv.layer_norm_',
                'pre_mlp_layernorm.': 'mlp.linear_fc1.layer_norm_',
            },
        ),
    )
```

```python
def get_bias_dropout_add(training, fused):
    if fused:
        # jit scripting for a nn.module (with dropout) is not
        # triggering the fusion kernel. For now, we use two
        # different nn.functional routines to account for varying
        # dropout semantics during training and inference phases.
        if training:
            return bias_dropout_add_fused_train
        else:
            return bias_dropout_add_fused_inference
    else:
        return bias_dropout_add_unfused(training)
```

- MLP 4h→h RowParallelLinear 输出 [s, b, h]
- Dropout 输入输出shape [s, b, h]，得到 MLP 整体输出 [s, b, h]

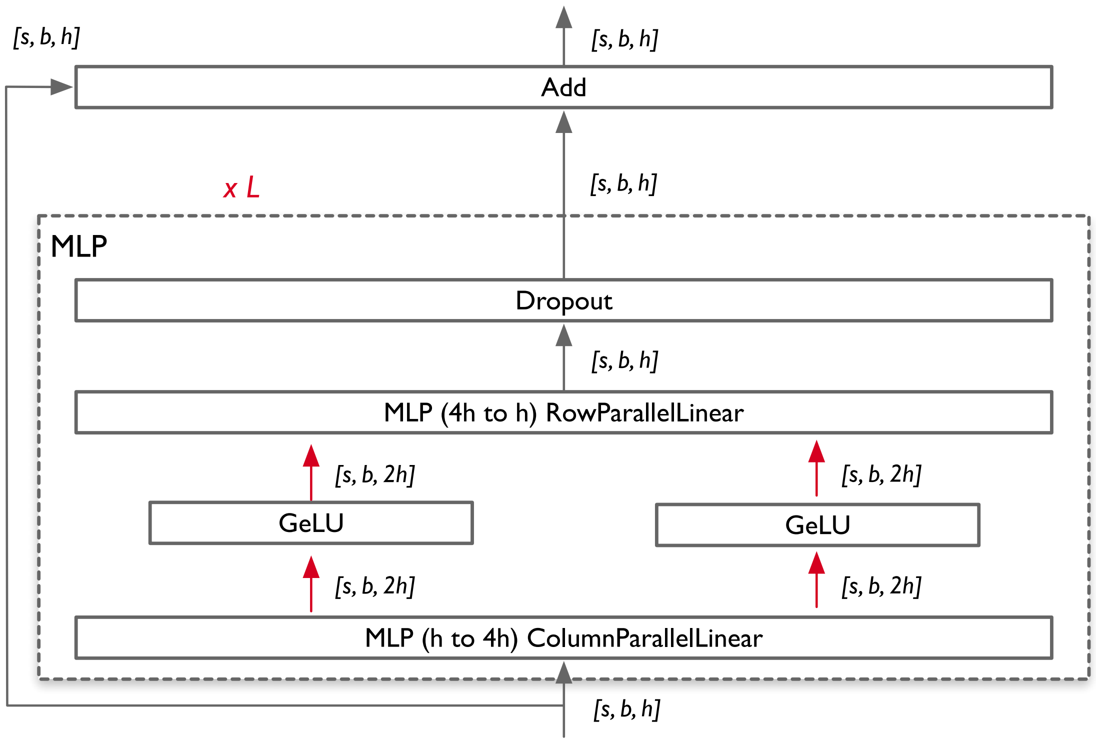




Add 结构

```
else:
    if mlp_bias is not None:
        mlp_output = mlp_output + mlp_bias
    out = torch.nn.functional.dropout(mlp_output,
                                      p=self.hidden_dropout,
                                      training=self.training)
    output = residual + self.drop_path(out)

if self.layer_type == LayerType.retro_decoder_with_retriever:
    return output, retriever_output
else:
    return output
```

- Add 将残差连接 Residual + MLP 输出 [s, b, h]
- 最终得到完整 Transformer Layer 输出 [s, b, h]
- 对于 L 层 Transformer Layer 重复 l 次
- 通信量：正反向各 2 次 all-reduce

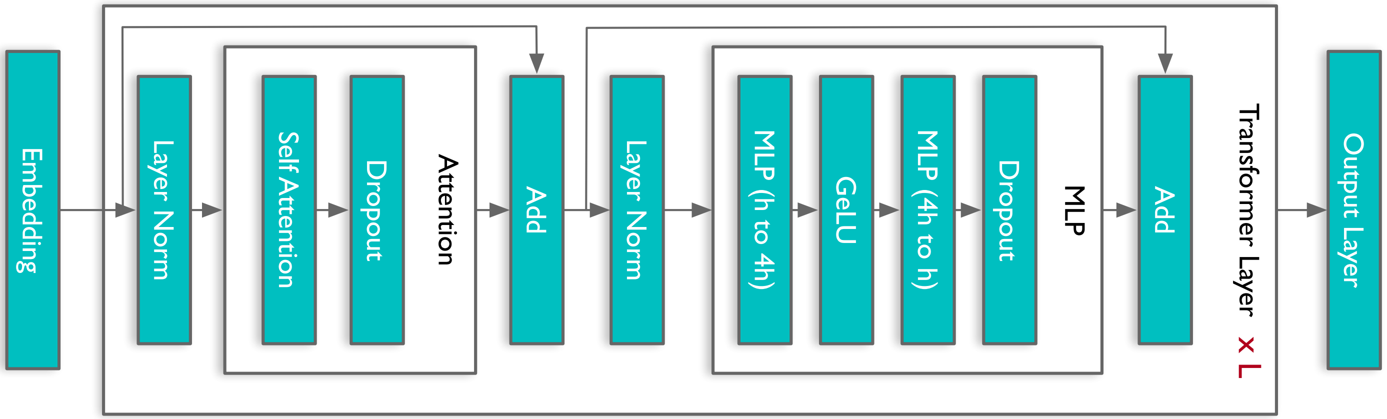


Megatron-LM 08

Transformer 总览

张量并行通信占用

- introduces two additional communication operations f and $\bar{f}$.



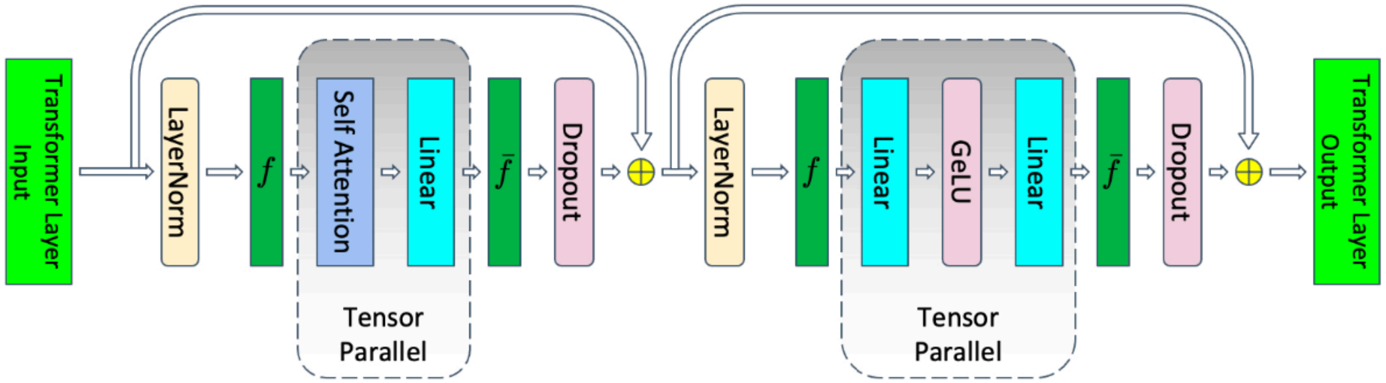
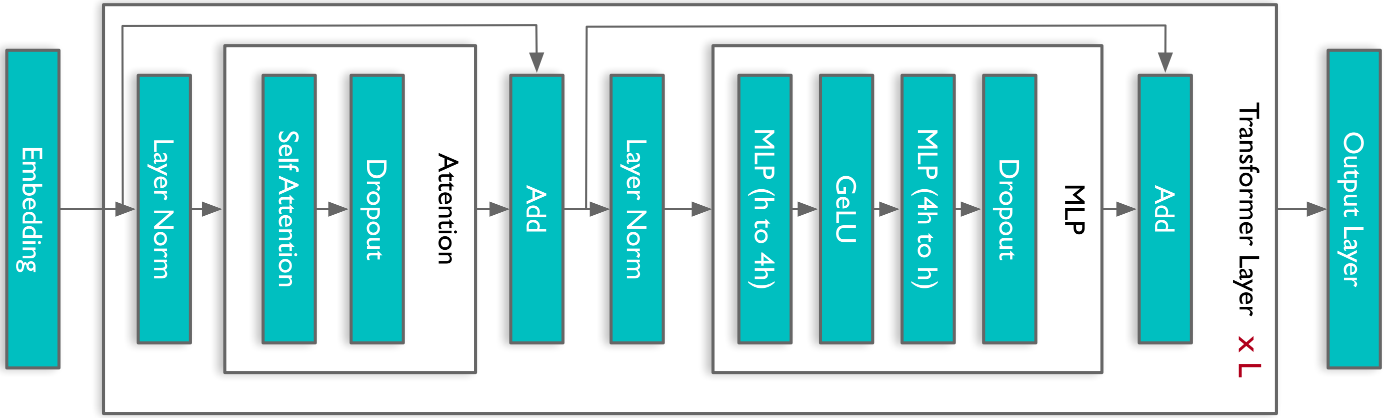


Figure 4: Transformer layer with tensor parallelism. f and $\bar{f}$ are conjugate. f is no operation in the

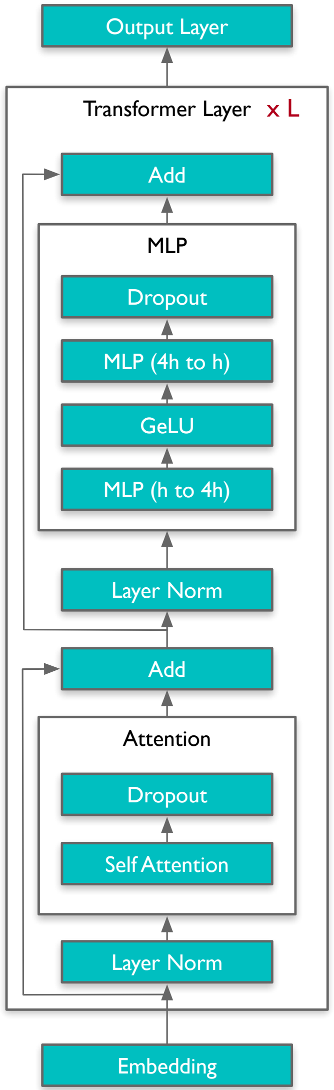
张量并行激活显存占用

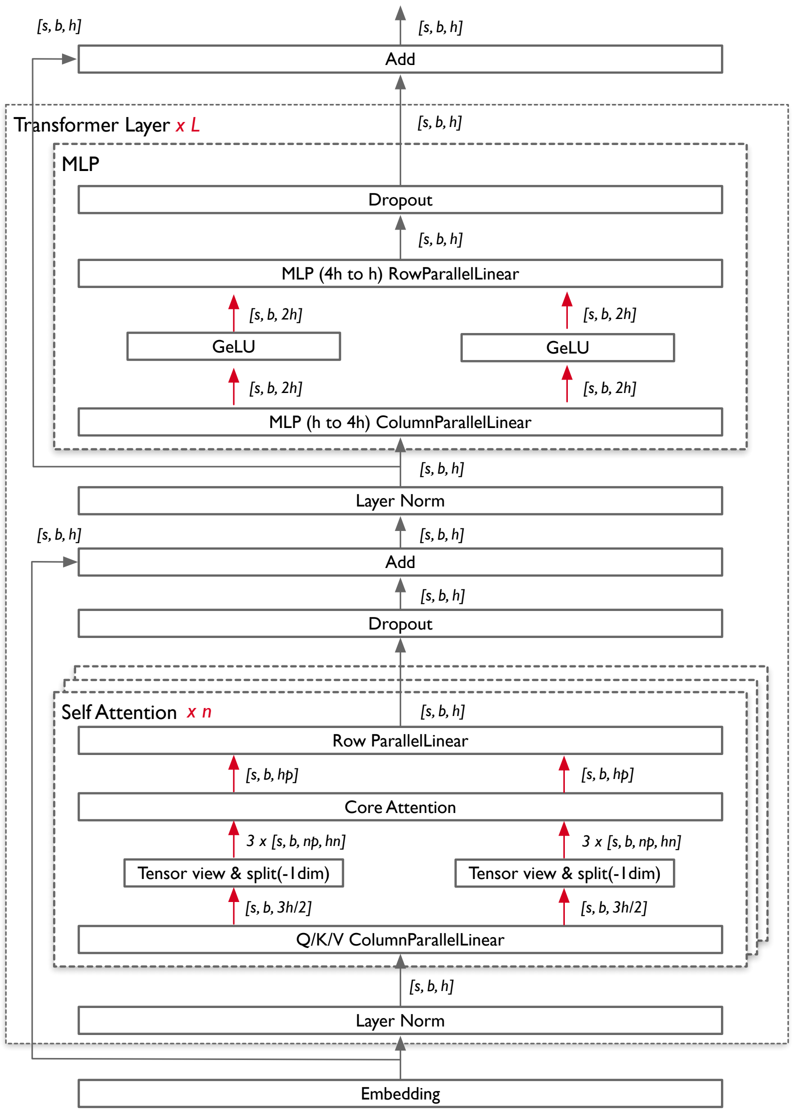
$$\text{Activations memory per layer} = sbh\left(34 + 5\frac{as}{h}\right)$$

$$\text{Activations memory per layer} = sbh\left(10 + \frac{24}{t} + 5\frac{as}{ht}\right)$$



张量并行总览


Output Layer
Transformer Layer x L
Add
MLP
Dropout
MLP (4h to h)
GeLU
MLP (h to 4h)
Layer Norm
Add
Attention
Dropout
Self Attention
Layer Norm
Embedding


[s, b, h]
[s, b, h]
Add
Transformer Layer x L
[s, b, h]
MLP
Dropout
[s, b, h]
MLP (4h to h) RowParallelLinear
[s, b, 2h]
[s, b, 2h]
GeLU
GeLU
[s, b, 2h]
[s, b, 2h]
MLP (h to 4h) ColumnParallelLinear
[s, b, h]
Layer Norm
[s, b, h]
[s, b, h]
Add
[s, b, h]
Dropout
Self Attention x n
[s, b, h]
Row ParallelLinear
[s, b, hp]
[s, b, hp]
Core Attention
3 x [s, b, np, hn]
3 x [s, b, np, hn]
Tensor view & split(-1dim)
Tensor view & split(-1dim)
[s, b, 3h/2]
[s, b, 3h/2]
Q/K/V ColumnParallelLinear
[s, b, h]
Layer Norm
Embedding

Thank you

把AI系统带入每个开发者、每个家庭、每个组织，构建万物互联的智能世界

Bring AI System to every person, home and organization for a fully connected, intelligent world.

ZOMI

Course chenzomi12.github.io

GitHub github.com/chenzomi12/DeepLearningSystem