ZOMI 大模型：分布式训练

序列并行

Sequence 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
- bench mark

8. 推理与智能体

- 量化压缩
- 推理加速
- 9. Agent智能体

集群算力准备

1. AI 集群建设

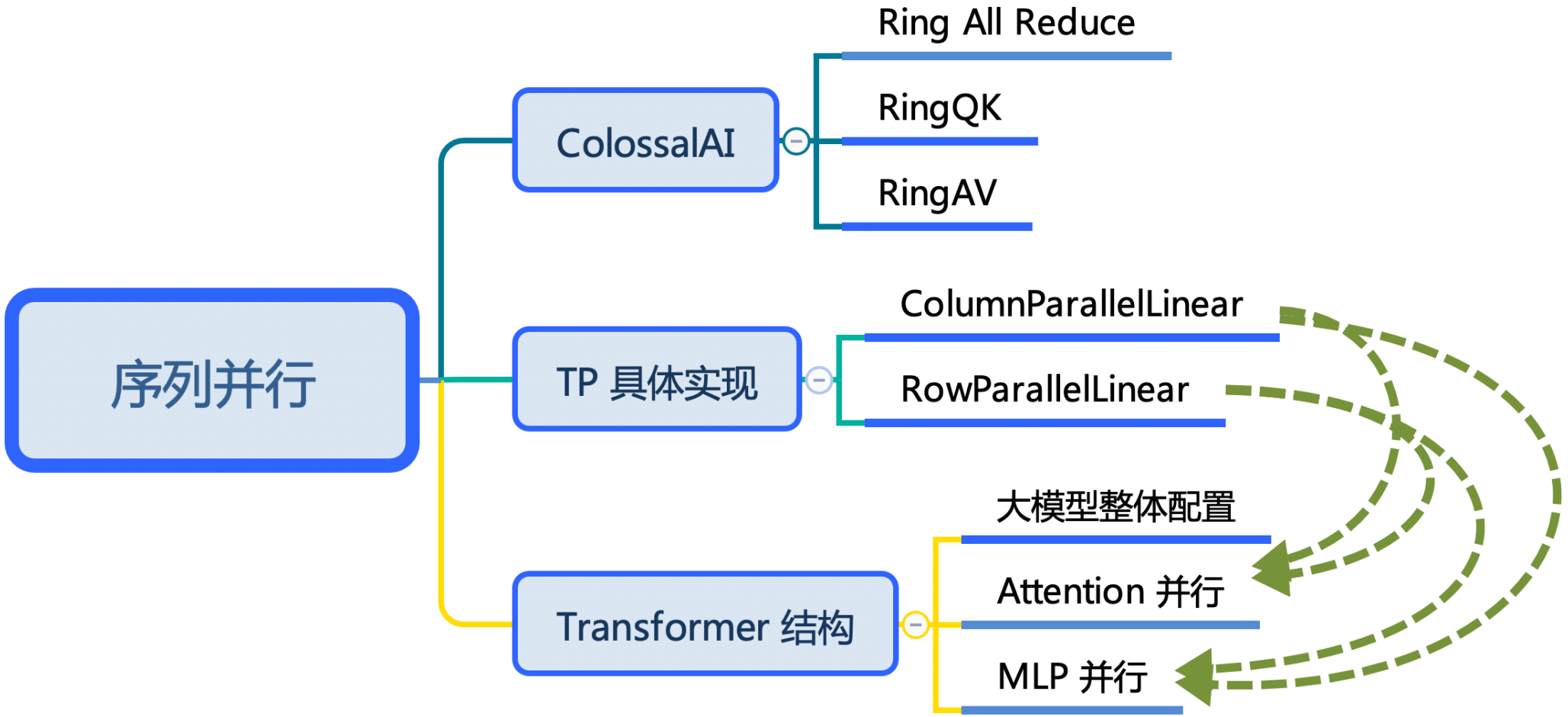
- 计算、存储、网络
- AI 集群机房建设
- AI 集群上线与运维

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 -- SP – PP

思维导图


序列并行
ColossalAI
Ring All Reduce
RingQK
RingAV
TP 具体实现
ColumnParallelLinear
RowParallelLinear
Transformer 结构
大模型整体配置
Attention 并行
MLP 并行

Colossal-AI 01

序列并行原理

ColossalAI序列并行

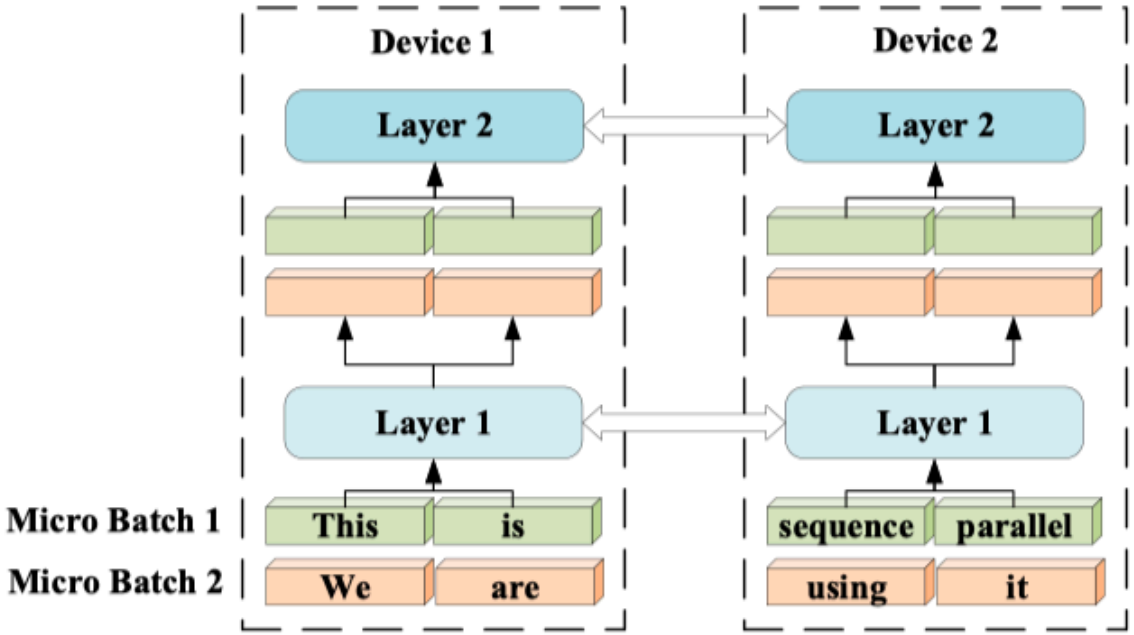
Sequence Parallelism: Long Sequence Training from System Perspective

Shenggui Li[1], **Fuzhao Xue**[1†] **Chaitanya Baranwal**[1] **Yongbin Li**[1] **Yang You**[1]
[1]Department of Computer Science, National University of Singapore

Abstract

Transformer achieves promising results on various tasks. However, self-attention suffers from quadratic memory requirements with respect to the sequence length. Existing work focuses on reducing time and space complexity from an algorithm perspective. In this work, we propose sequence parallelism, a memory-efficient parallelism method to help us break input sequence length limitation and train with longer sequences on GPUs efficiently. Our approach is compatible with most existing parallelisms (*e.g.,* data parallelism, pipeline parallelism and tensor parallelism), which means our sequence parallelism makes 4D parallelism possible. More importantly, we no longer require a single device to hold the whole sequence. That is, with sparse attention, our sequence parallelism enables us to train transformer with infinite long sequence. Specifically, we split the input sequence into multiple chunks and feed each chunk into its corresponding device (*i.e.,* GPU). To compute the attention output, we integrated ring-style communication with self-attention calculation and proposed Ring Self-Attention (RSA). Experiments show that sequence parallelism performs well when scaling with batch size and sequence length. Compared with tensor parallelism, our approach achieved 13.7× and 3.0× maximum batch size and sequence length respectively when scaling up to 64 NVIDIA P100 GPUs. With sparse attention, sequence can handle sequence with over 114K tokens, which is over 27× longer than existing sparse attention works holding the whole sequence on a single device.

- 实现上 ColossalAI 借鉴了 Ring-Allreduce 算法实现，通过 RingQK 和 RingAV 两种计算方式实现序列并行。

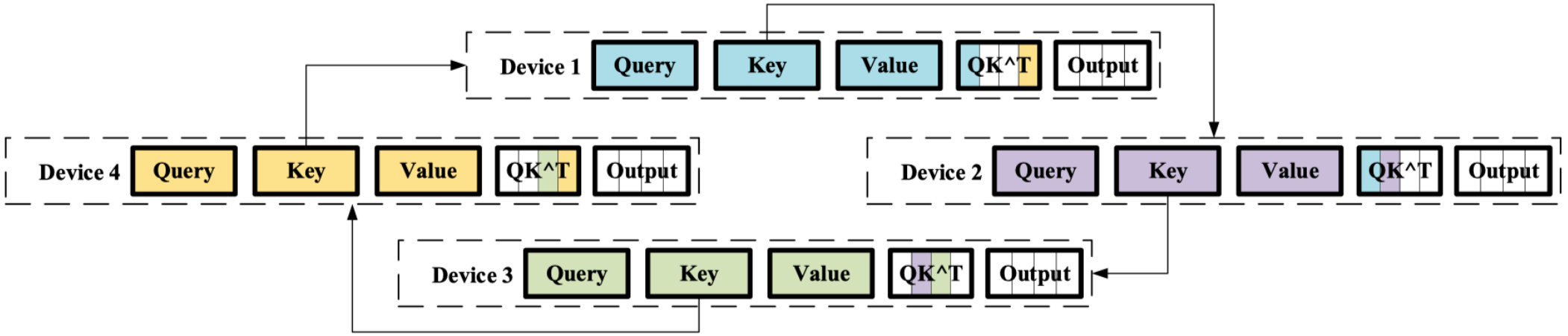


(c) Sequence parallelism (Ours)

Figure 1: The overall architecture of the proposed sequence parallelism and existing parallel approaches. For sequence parallelism, Device 1 and Device 2 share the same trainable parameters.

RingQK 过程

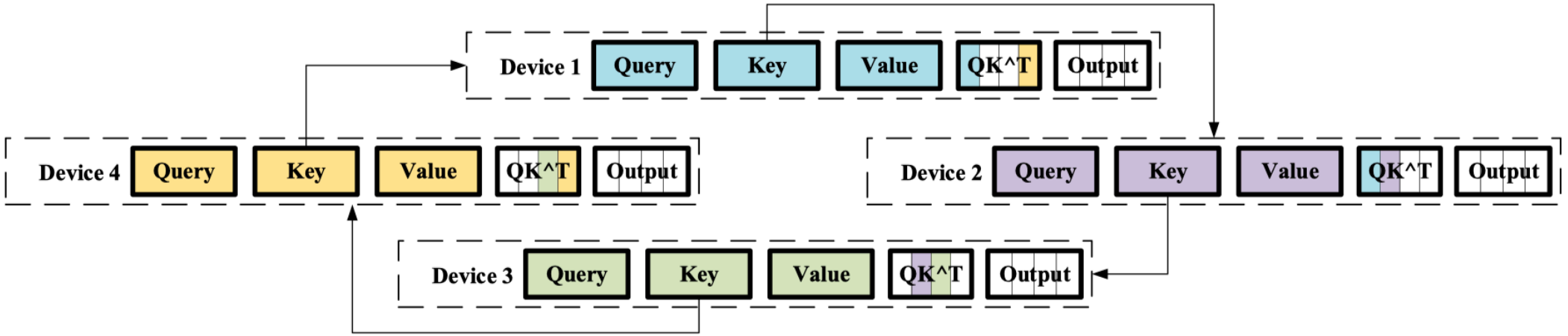
- 每 NPU 保存一份 sub Seq，但是 Q/K/V 的计算有需要和完整的 Seq 作为输入；
- 利用Ring All-Reduce 在每次通信 iter 时，互相传输各 NPU 的子序列数据。



(a) Transmitting key embeddings among devices to calculate attention scores

RingQK 过程

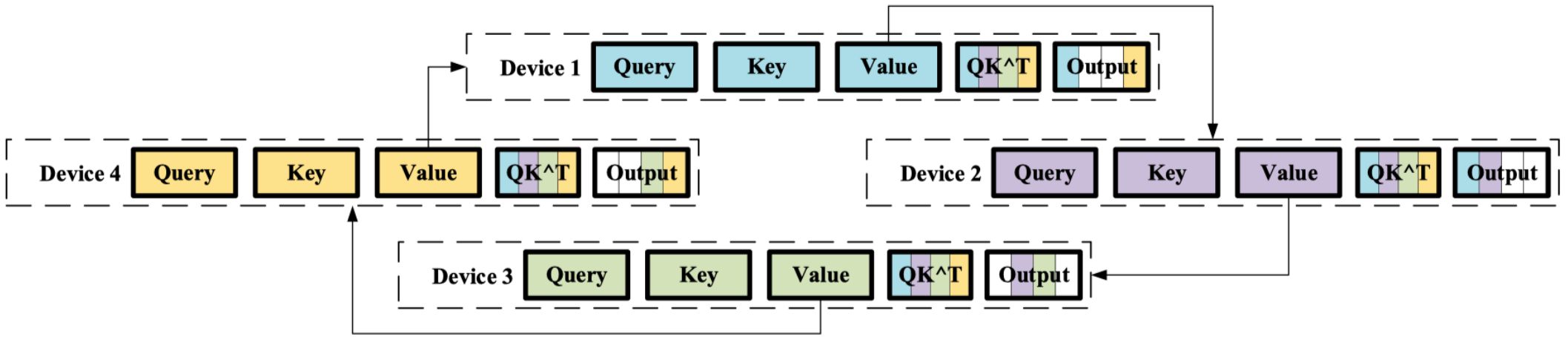
- NPU1 接收了 NPU4 的 Key，计算 NPU1 & NPU4 的 $QK^T{}_1$
- NPU2 接收了 NPU1 的 Key，计算 NPU2 & NPU1 的 $QK^T{}_2$
- ….
- N-1 个 iter 后，所有 NPU 都有完整 QK^T 结果，完成一次 RingQK 过程。



(a) Transmitting key embeddings among devices to calculate attention scores

RingAV 过程

- 接下来计算 Attention Scores，通过 Ring All-Reduce 算法计算每 NPU 的 Attention Prob * Value；
- 与 RingQK 相同的计算逻辑，每 NPU 都传输各自 Value，得到最终的输出；



(b) Transmitting value embeddings among devices to calculate the output of attention layers

MLP Block 内存计算方式

Table 1: MLP block memory usage comparison. M1 means the matrix before linear layer, and M2 is the trainable matrix of linear layer.

	GEMM	M1	M2	output	Memory
Tensor parallelism	1st linear	(B, L, H)	$(H, \frac{4H}{N})$	$(B, L, \frac{4H}{N})$	$\frac{32H^2}{N} + \frac{4BLH}{N} + BLH$
	2nd linear	$(B, L, \frac{4H}{N})$	$(\frac{4H}{N}, H)$	(B, L, H)	
Sequence parallelism	1st linear	$(B, \frac{L}{N}, H)$	$(H, 4H)$	$(B, \frac{L}{N}, 4H)$	$32H^2 + \frac{5BLH}{N}$
	2nd linear	$(B, \frac{L}{N}, 4H)$	$(4H, H)$	$(B, \frac{L}{N}, H)$	

Attention Block 内存计算方式

Table 2: Multi-head attention block memory usage comparison

	Operation	M1	M2	output	Memory
Tensor parallelism	$Q/K/V$	(B, L, H)	$(H, \frac{ZA}{N})$	$(B, \frac{Z}{N}, L, A)$	$\frac{16AZH}{N} + \frac{4BLZA}{N} + \frac{BZL^2}{N} + BLH$
	QK^T	$(B, \frac{Z}{N}, L, A)$	$(B, \frac{Z}{N}, L, A)$	$(B, \frac{Z}{N}, L, L)$	
	AV	$(B, \frac{Z}{N}, L, L)$	$(B, \frac{Z}{N}, L, A)$	$(B, \frac{Z}{N}, L, A)$	
	Linear	$(B, \frac{Z}{N}, L, A)$	$(\frac{AZ}{N}, H)$	(B, L, H)	
Sequence parallelism	$Q/K/V$	$(B, \frac{L}{N}, H)$	(H, AZ)	$(B, Z, \frac{L}{N}, A)$	$16AZH + \frac{4BZLA}{N} + \frac{BZL^2}{N} + \frac{BLH}{N}$
	Ring-QK^T	$(B, Z, \frac{L}{N}, A)$	$(B, Z, \frac{L}{N}, A)$	$(B, Z, \frac{L}{N}, L)$	
	Ring-AV	$(B, Z, \frac{L}{N}, L)$	$(B, Z, \frac{L}{N}, A)$	$(B, Z, \frac{L}{N}, A)$	
	Linear	$(B, Z, \frac{L}{N}, A)$	(AZ, H)	$(B, \frac{L}{N}, H)$	

问题在哪里？

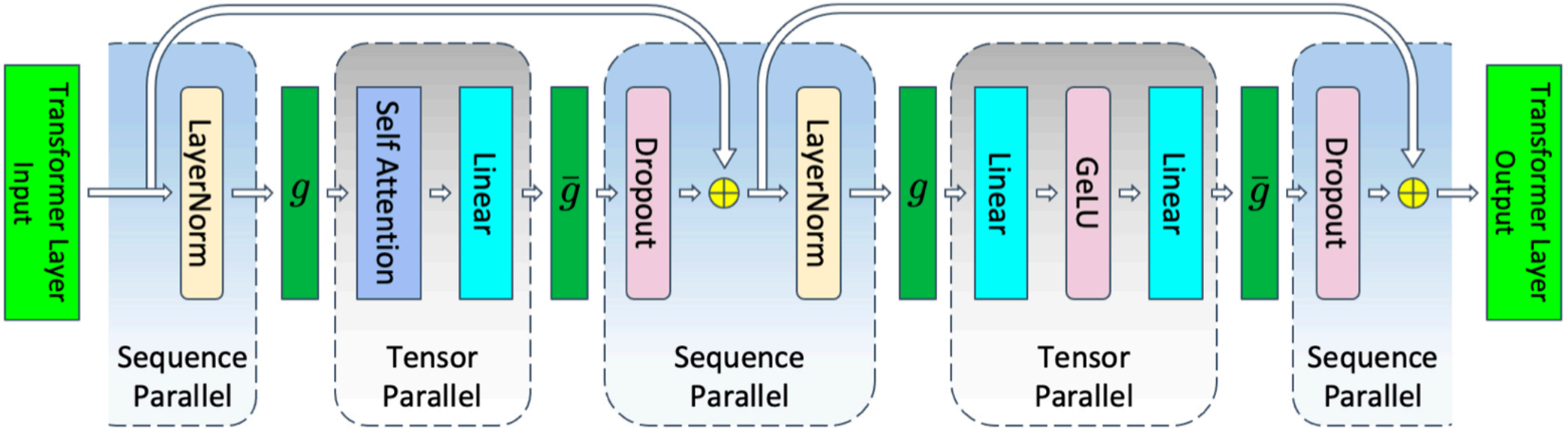
1. 序列并行不能与模型并行一起使用，原理上会引起冲突，那么应该在满足什么条件下才使用 ColossalAI 的系列并行呢？
2. 因为使用了 Ring All Reduce 算法，会引入更频繁的通信（包括前向和后向），对比 Megatron-LM 的序列并行 + 模型并行，哪种效果会更好呢？

Megatron-LM 02

序列并行原理

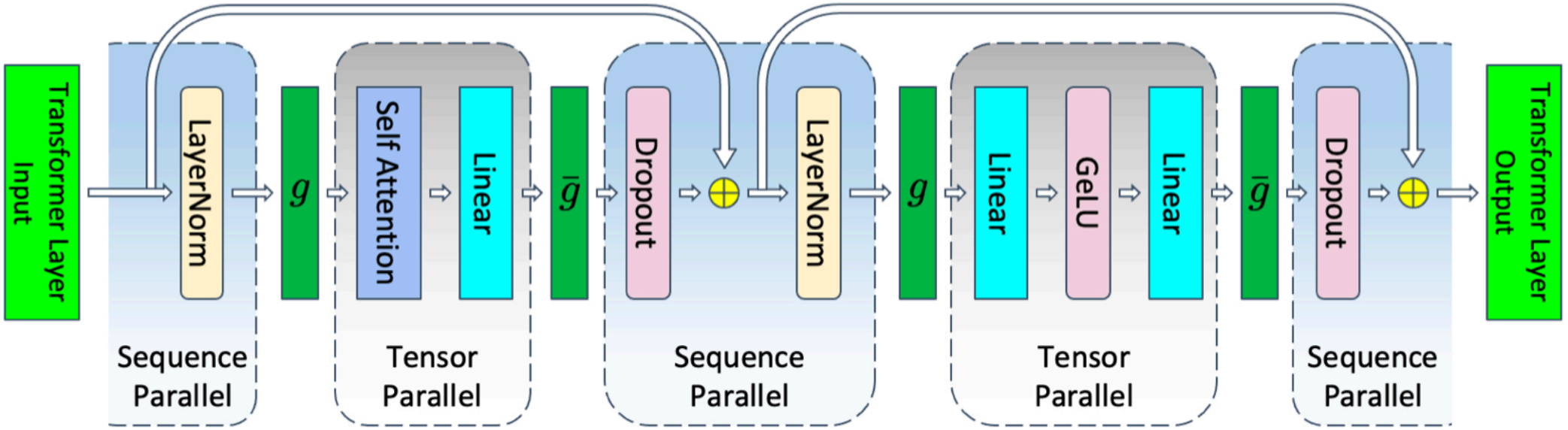
序列并行原理

1. LayerNorm 和 Dropout 的计算被平摊到了各个 NPU 上，减少了计算资源的浪费；
2. LayerNorm 和 Dropout 所产生的激活值也被平摊到了各个 NPU 上，进一步降低了显存开销；



序列并行原理：引入通信

- Megatron1/2：Transformer Core TP 由正向 2x All-reduce + 后向 2x All-reduce 组成;
- Megatron 3：对 TP+SP，2x All-reduce 不合适，为了收集在各 NPU SP 输出结果，需插入 All-Gather；为使 TP 产生结果可传入 SP 层，需插入 Reduce-Scatter 算子；其中，g 代表前向 All-Gather，反向reduce scatter，$\bar{g}$ 则是反之。因此 TP + SP 有 4x All-Gather + 4x Reduce-Scatter。



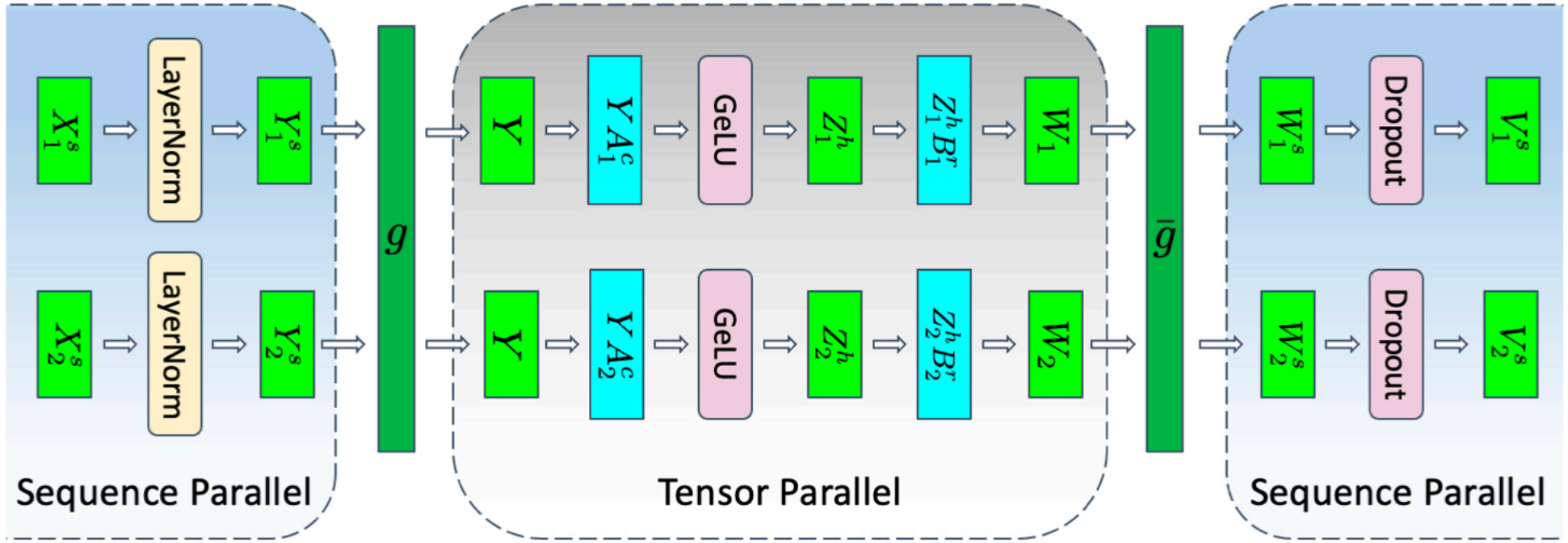
序列并行原理

We detail g and $\bar{g}$'s derivation using the MLP block. In the non-parallel form, as shown in Figure 2, the layer-norm followed by the MLP block can be formulated as:

$$
\begin{aligned}
Y &= \text{LayerNorm}(X), \\
Z &= \text{GeLU}(YA), \\
W &= ZB, \\
V &= \text{Dropout}(W),
\end{aligned}
$$

where X is input to the layer-norm with size $s \times b \times h$ and A and B are the weight matrices of the linear layers with size $h \times 4h$ and $4h \times h$, respectively. The combined tensor and sequence parallel form of the above operations is shown in Figure 6. The subscripts represent splitting among accelerators and superscripts depict the dimension along which the splitting is done. For example, X_1^s is the first accelerator's part of X that is split along the s dimension (sequence dimension) while Z_2^h is the second accelerator's part of Z that is split along the h dimension (hidden dimension).

序列并行原理


X_1^s
LayerNorm
Y_1^s
X_2^s
LayerNorm
Y_2^s
Sequence Parallel
g
Y
YA_1^c
GeLU
Z_1^h
$Z_1^h B_1^r$
W_1
Y
YA_2^c
GeLU
Z_2^h
$Z_2^h B_2^r$
W_2
Tensor Parallel
$\bar{g}$
W_1^s
Dropout
V_1^s
W_2^s
Dropout
V_2^s
Sequence Parallel

序列并行原理

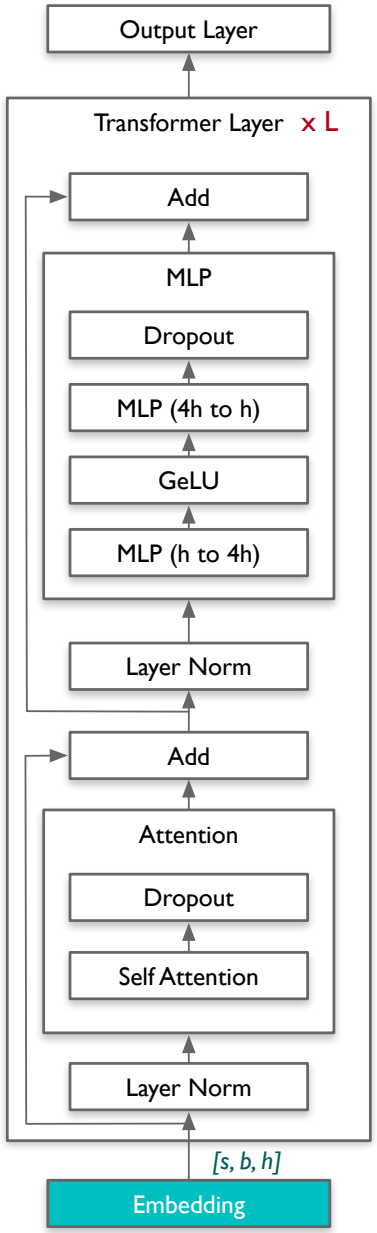
then parallelizing in the sequence dimension, we combine these two operations into a reduce-scatter operation. As a result, $\bar{g}$ can be a single reduce-scatter operation in the forward pass. Putting it all together, we arrive at:

$$
\begin{aligned}
[Y_1^s, Y_2^s] &= \text{LayerNorm}([X_1^s, X_2^s]), \\
Y &= g(Y_1^s, Y_2^s), \\
[Z_1^h, Z_2^h] &= [\text{GeLU}(YA_1^c),\ \text{GeLU}(YA_2^c)], \\
W_1 &= Z_1^h B_1^r \quad \text{and} \quad W_2 = Z_2^h B_2^r, \\
[W_1^s, W_2^s] &= \bar{g}(W_1, W_2), \\
[V_1^s, V_2^s] &= [\text{Dropout}(W_1^s),\ \text{Dropout}(W_2^s)].
\end{aligned}
\tag{3}
$$

If we follow a similar break-down for the backward pass, we find that g and $\bar{g}$ are conjugate of each other. g is an all-gather in the forward pass and a reduce-scatter in the backward pass, and $\bar{g}$ is a reduce-scatter in the forward pass and an all-gather in the backward pass. A similar breakdown done for the layer-norm followed by the attention part of the transformer layer arrives at Figure 5.

Megatron-LM 04

序列并行实现



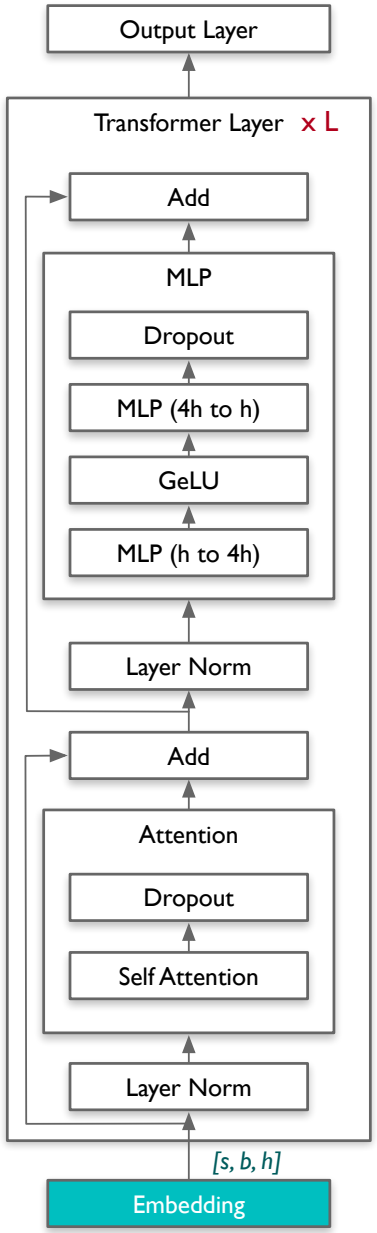
序列并行实现

```
    # disable sequence parallelism when tp=1
    # to avoid change in numerics when
    # sequence_parallelism is enabled.
    if args.tensor_model_parallel_size == 1:
        args.sequence_parallel = False
```

```
class LanguageModelEmbedding(MegatronModule):
    def forward(self, input_ids: Tensor, position_ids: Tensor, tokentype_ids: int = Non
        else:
            assert self.tokentype_embeddings is None

        # If the input flag for fp32 residual connection is set, convert for float.
        if self.config.fp32_residual_connection:
            embeddings = embeddings.float()

        # Dropout.
        if self.config.sequence_parallel:
            embeddings = tensor_parallel.scatter_to_sequence_parallel_region(embeddings
            # `scatter_to_sequence_parallel_region` returns a view, which prevents
            # the original tensor from being garbage collected. Clone to facilitate GC.
            # Has a small runtime cost (~0.5%).
            if self.config.clone_scatter_output_in_embedding:
                embeddings = embeddings.clone()
            with tensor_parallel.get_cuda_rng_tracker().fork():
                embeddings = self.embedding_dropout(embeddings)
        else:
            embeddings = self.embedding_dropout(embeddings)

        return embeddings
```

- seq/sq/sk = 输入 seq len， e.g. 2048
- np = 每个 NPU 上的 Head 数量， e.g. 8
- hn = 每个 Head 输出的 Hidden Size， e.g. 64
- hp = 每个 NPU 上 Attention 输出 hidden Size， e.g. 1024/2 = 512
- h = Tranformer Layer Hidden Size， e.g. 1024

```
GPT_ARGS="
    --tensor-model-parallel-size 2 \
    --pipeline-model-parallel-size 1 \
    -- sequence_parallel \
    --num-layers 24 \
    --hidden-size 1024 \
    --num-attention-heads 16 \
    --seq-length 1024 \
    --max-position-embeddings 1024 \
    --micro-batch-size 4 \
    --global-batch-size 8 \
    --lr 0.00015 \
    --train-iters 500000 \
    --lr-decay-iters 320000 \
    --lr-decay-style cosine \
    --min-lr 1.0e-5 \
    --weight-decay 1e-2 \
```



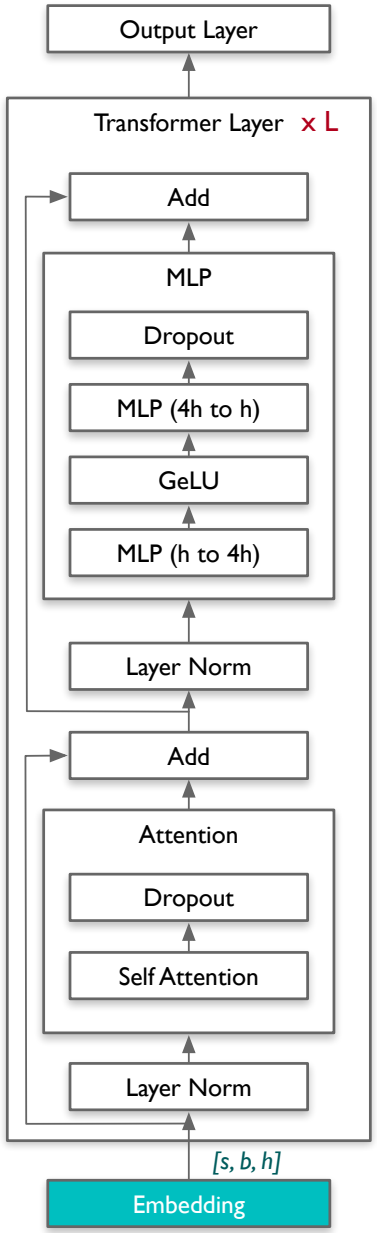
序列并行实现

```
    # disable sequence parallelism when tp=1
    # to avoid change in numerics when
    # sequence_parallelism is enabled.
    if args.tensor_model_parallel_size == 1:
        args.sequence_parallel = False
```

```
class LanguageModelEmbedding(MegatronModule):
    def forward(self, input_ids: Tensor, position_ids: Tensor, tokentype_ids: int = Non
        else:
            assert self.tokentype_embeddings is None

        # If the input flag for fp32 residual connection is set, convert for float.
        if self.config.fp32_residual_connection:
            embeddings = embeddings.float()

        # Dropout.
        if self.config.sequence_parallel:
            embeddings = tensor_parallel.scatter_to_sequence_parallel_region(embeddings
            # `scatter_to_sequence_parallel_region` returns a view, which prevents
            # the original tensor from being garbage collected. Clone to facilitate GC.
            # Has a small runtime cost (~0.5%).
            if self.config.clone_scatter_output_in_embedding:
                embeddings = embeddings.clone()
            with tensor_parallel.get_cuda_rng_tracker().fork():
                embeddings = self.embedding_dropout(embeddings)
        else:
            embeddings = self.embedding_dropout(embeddings)

        return embeddings
```

- 打开 sequence-parallel 配置
- Embeddings 对 input 执行 word embedding，输出 shape [s, b, h]
- 对word embeddings 进行 sequence parallel 并行

```
GPT_ARGS="
    --tensor-model-parallel-size 2 \
    --pipeline-model-parallel-size 1 \
    -- sequence_parallel \
    --num-layers 24 \
    --hidden-size 1024 \
    --num-attention-heads 16 \
    --seq-length 1024 \
    --max-position-embeddings 1024 \
    --micro-batch-size 4 \
    --global-batch-size 8 \
    --lr 0.00015 \
    --train-iters 500000 \
    --lr-decay-iters 320000 \
    --lr-decay-style cosine \
    --min-lr 1.0e-5 \
    --weight-decay 1e-2 \
```



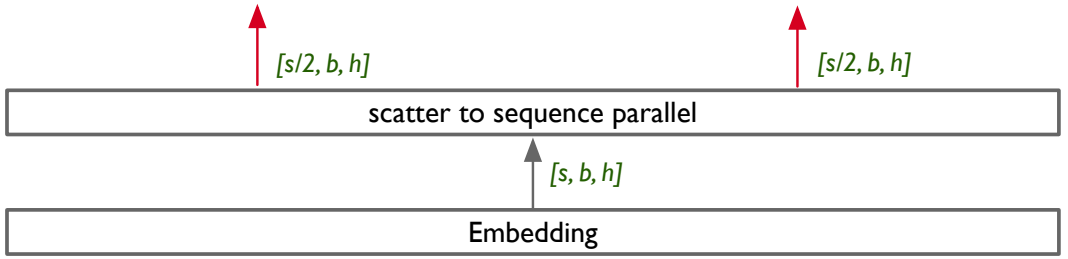
序列并行实现

```python
class LanguageModelEmbedding(MegatronModule):
    def forward(self, input_ids: Tensor, position_ids: Tensor, tokentype_ids: int = None
        # Dropout.
        if self.config.sequence_parallel:
            embeddings = tensor_parallel.scatter_to_sequence_parallel_region(embeddings)
            # `scatter_to_sequence_parallel_region` returns a view, which prevents
            # the original tensor from being garbage collected. Clone to facilitate GC.
            # Has a small runtime cost (~0.5%).
```

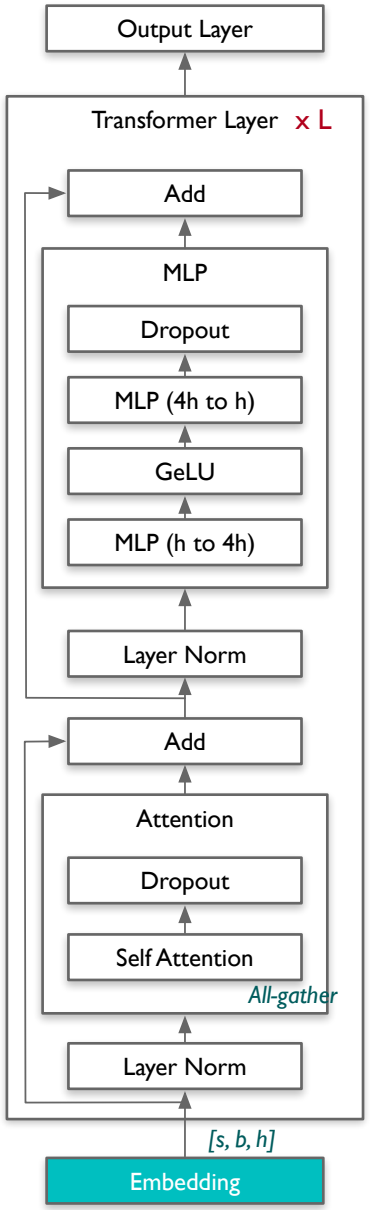
```python
class _ScatterToSequenceParallelRegion(torch.autograd.Function):
    """Split the input and keep only the corresponding chuck to the rank."""

    @staticmethod
    def symbolic(graph, input_):
        return _split_along_first_dim(input_)
```

```python
def _split_along_first_dim(input_):
    """Split the tensor along its first dimension and keep the
    corresponding slice."""

    world_size = get_tensor_model_parallel_world_size()
    # Bypass the function if we are using only 1 GPU.
    if world_size == 1:
        return input_

    # Split along first dimension.
    dim_size = input_.size()[0]
    assert (
        dim_size % world_size == 0
    ), "First dimension of the tensor should be divisible by tensor paralle
    local_dim_size = dim_size // world_size
    rank = get_tensor_model_parallel_rank()
    dim_offset = rank * local_dim_size

    output = input_[dim_offset : dim_offset + local_dim_size].contiguous()

    return output
```

- _split_along_first_dim() 函数实现按 seq 维度对输入 words_embeddings 执行 split
- 2 个 NPU 上分别输出 X1 X2 shape [s/2, b, h]，即在 s 维度进行切分



序列并行实现

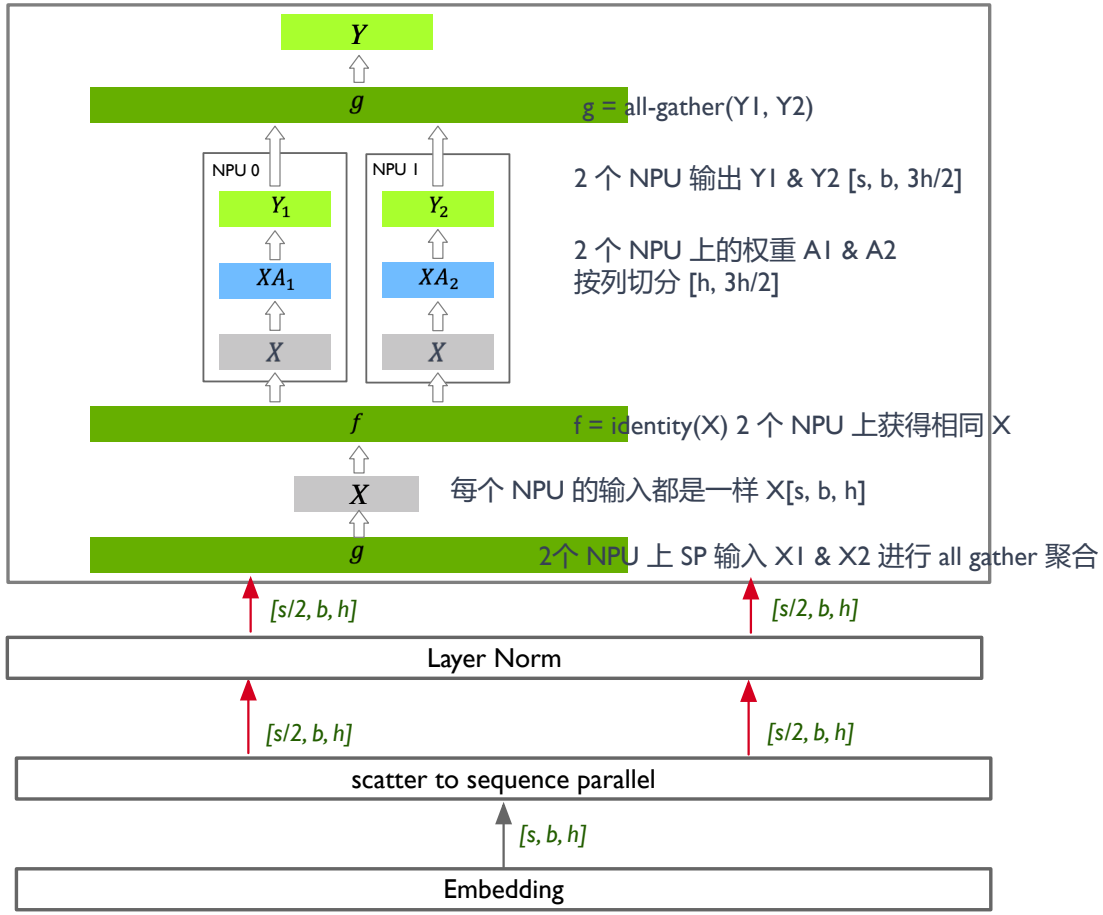



```python
class ColumnParallelLinear(torch.nn.Module):
    def __init__(

        if self.async_tensor_model_parallel_allreduce and self.sequence_parallel:
            raise RuntimeError(
                "`async_tensor_model_parallel_allreduce` and `sequence_parallel` "
                "cannot be enabled at the same time."
            )

        self._forward_impl = linear_with_grad_accumulation_and_async_allreduce
        self.explicit_expert_comm = self.is_expert and (
            self.sequence_parallel or self.expert_parallel
        )
```

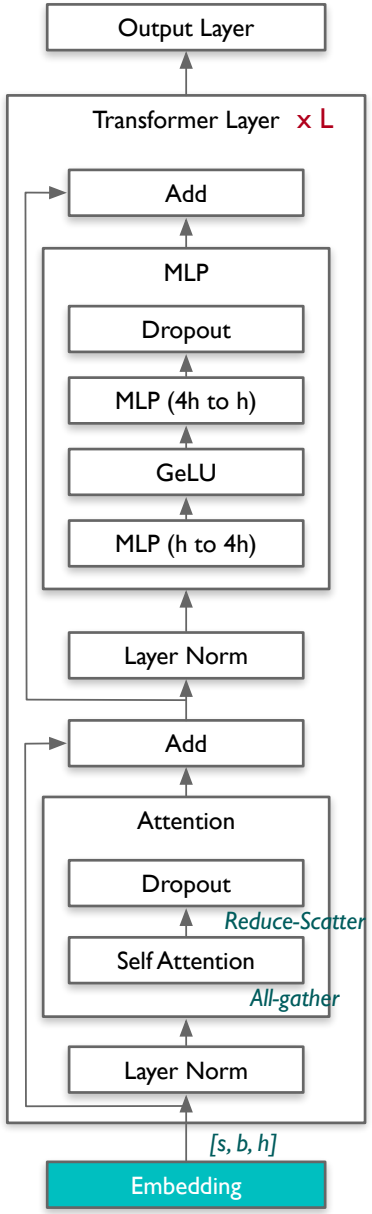
```python
class LinearWithGradAccumulationAndAsyncCommunication(torch.autograd.Function):
    def forward(

        if sequence_parallel:
            world_size = get_tensor_model_parallel_world_size()
            dim_size = list(input.size())
            dim_size[0] = dim_size[0] * world_size

            all_gather_buffer = get_global_memory_buffer().get_tensor(dim_size, input.d
            torch.distributed._all_gather_base(
                all_gather_buffer, input, group=get_tensor_model_parallel_group()
            )
            total_input = all_gather_buffer
        else:
            total_input = input

        output = torch.matmul(total_input, weight.t())
        if bias is not None:
            output = output + bias
        return output
```

```python
def _gather_along_first_dim(input_):
    """Gather tensors and concatinate along the first dimension."""

    world_size = get_tensor_model_parallel_world_size()
    # Bypass the function if we are using only 1 GPU.
    if world_size == 1:
        return input_

    dim_size = list(input_.size())
    dim_size[0] = dim_size[0] * world_size

    output = torch.empty(dim_size, dtype=input_.dtype, device=torch.cuda.current_devic
    torch.distributed._all_gather_base(
        output, input_.contiguous(), group=get_tensor_model_parallel_group()
    )

    return output
```

- ColumnParallelLinear 先将 SP 在 2 个 NPU 上分别输出 X1 X2 Shape [s/2, b, h] 执行 all-gather 合并成 Shape [s, b, h]
- 在 2 个 NPU 上并行计算输出 Y1 Y2 Shape [s, b, 3h/2]



序列并行实现

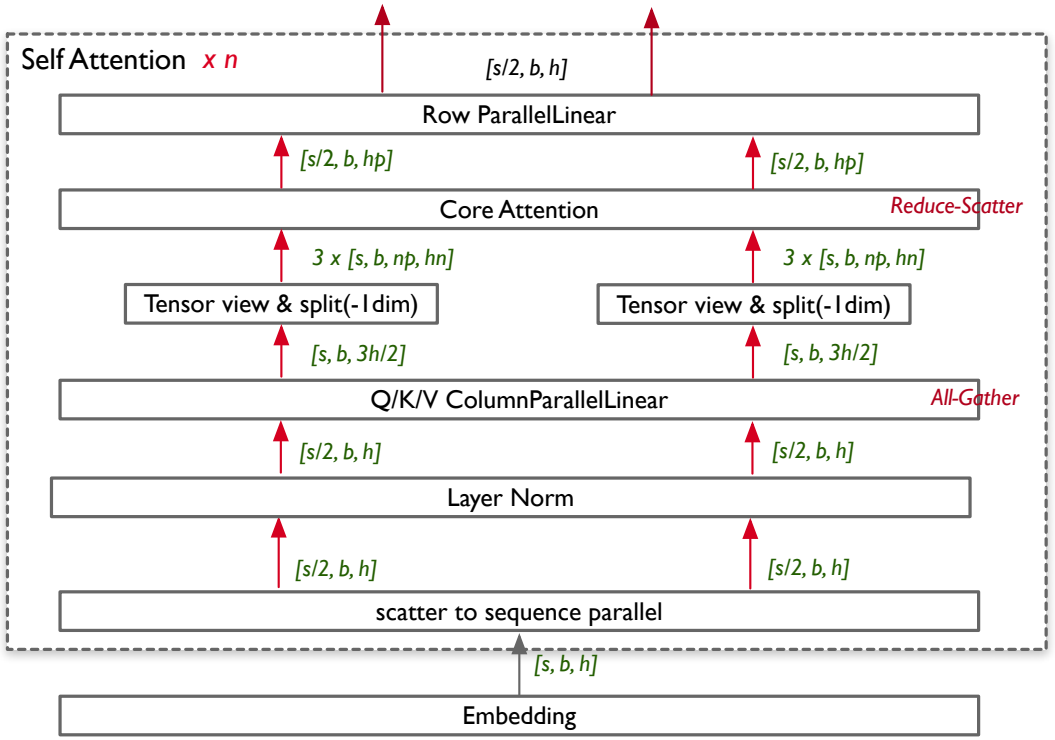


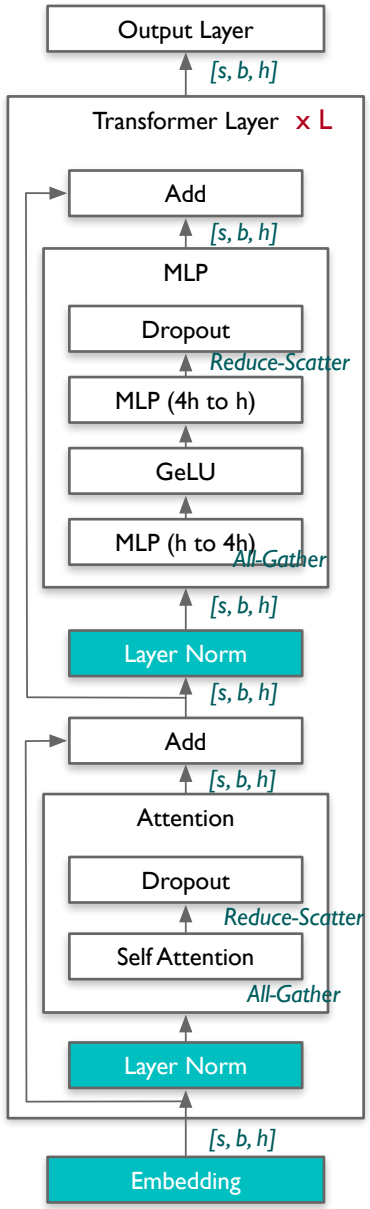

```python
class RowParallelLinear(torch.nn.Module):
    def __init__(
        self.input_size_per_partition = divide(input_size, world_size)
        self.skip_bias_add = skip_bias_add
        self.config = config
        self.is_expert = is_expert
        self.expert_parallel = config.expert_model_parallel_size > 1
        self.gradient_accumulation_fusion = config.gradient_accumulation_fusion
        self.sequence_parallel = config.sequence_parallel
        if self.sequence_parallel and not self.input_is_parallel:
            raise RuntimeError("To enable `sequence_parallel`, `input_is_parallel` mus
```

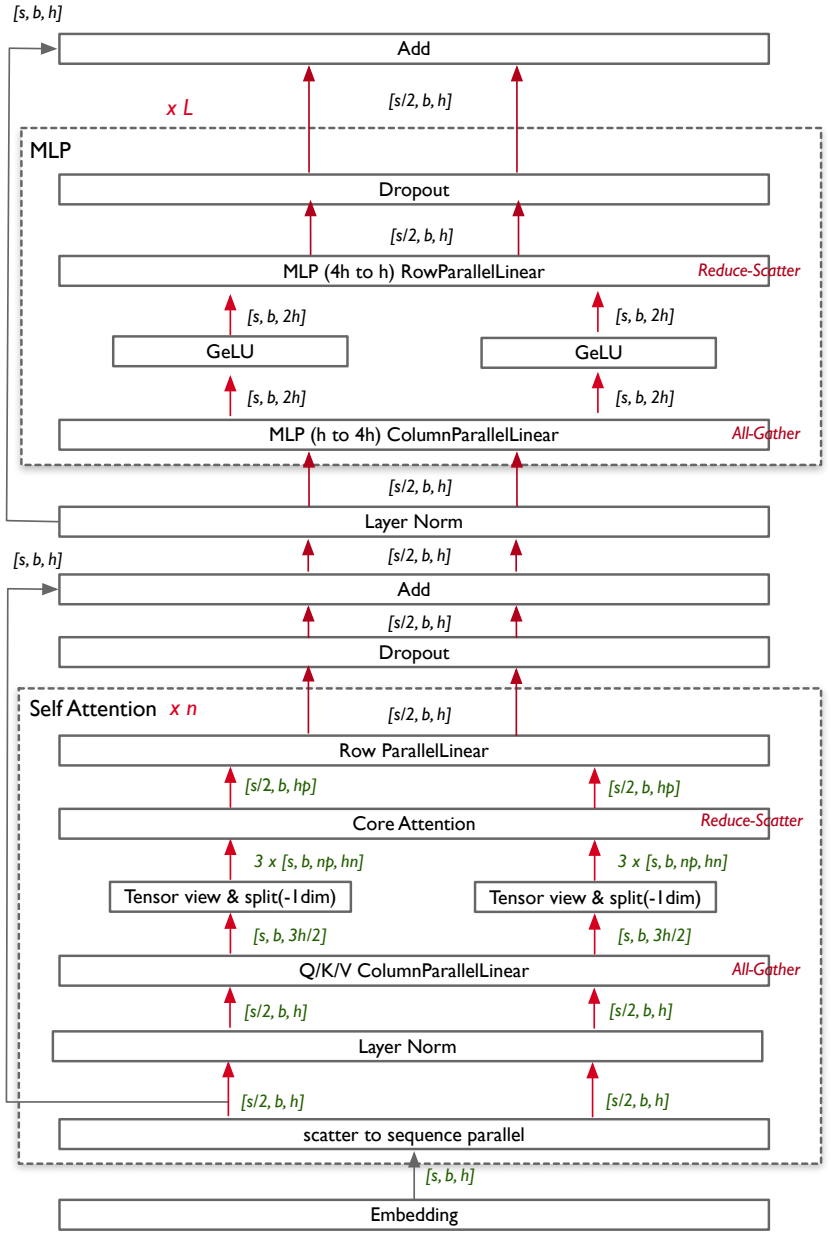
```python
    def forward(self, input_):

        # All-reduce across all the partitions.
        if self.explicit_expert_comm:
            assert self.skip_bias_add
            output_ = output_parallel
        elif self.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
```

```python
def _reduce_scatter_along_first_dim(input_):
    """Reduce-scatter the input tensor across model parallel group."""
    world_size = get_tensor_model_parallel_world_size()
    # Bypass the function if we are using only 1 GPU.
    if world_size == 1:
        return input_

    dim_size = list(input_.size())
    assert (
        dim_size[0] % world_size == 0
    ), "First dimension of the tensor should be divisible by tensor parallel size"

    dim_size[0] = dim_size[0] // world_size

    output = torch.empty(dim_size, dtype=input_.dtype, device=torch.cuda.current_device())
    torch.distributed._reduce_scatter_base(
        output, input_.contiguous(), group=get_tensor_model_parallel_group()
    )
    return output
```

- RowParallelLinear 先将 SP 在 2 个 NPU 上Y1 Y2 [s, b, h]
- 执行 reduce-scatter 输出 Y1 Y2 Shape [s/2, b, h]




Output Layer
[s, b, h]
Transformer Layer x L
Add
[s, b, h]
MLP
Dropout
Reduce-Scatter
MLP (4h to h)
GeLU
MLP (h to 4h)
All-Gather
[s, b, h]
Layer Norm
[s, b, h]
Add
[s, b, h]
Attention
Dropout
Reduce-Scatter
Self Attention
All-Gather
Layer Norm
[s, b, h]
Embedding


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