5.序列并行

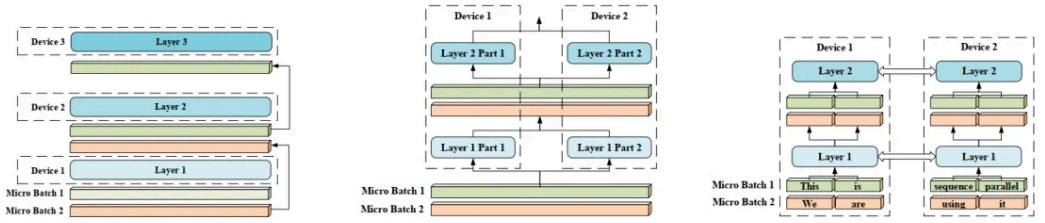
1.序列并行（Colossal-AI）

> Colossal-AI 发表的论文：Sequence Parallelism: Long Sequence Training from System Perspective，主要是**解决模型的输入长度(sequence length)限制。**

Colossal-AI 序列并行诞生的背景是 self-attention 的内存需求是输入长度（sequence length）的2次方。其复杂度为$O(n^2)$，其中，n 是序列长度。换言之，长序列数据将增加中间activation内存使用量，从而限制设备的训练能力。

而现有的工作侧重于从算法的角度降低时间和空间复杂度。因此，作者提出了序列并行，这是一种内存高效的并行方法，可以帮助我们打破输入序列长度限制，并在 GPU 上有效地训练更长的序列；同时，该方法与大多数现有的并行技术兼容（例如：数据并行、流水线并行和张量并行）。

更重要的是，不再需要单个设备来保存整个序列。 即在稀疏注意力的情况下，我们的序列并行使我们能够训练具有无限长序列的 Transformer。



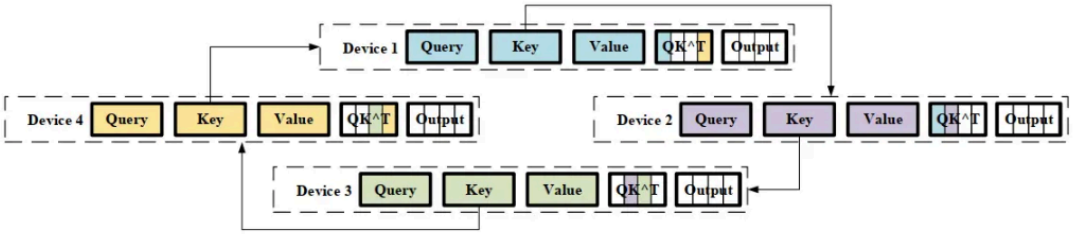
(a) Pipeline parallelism (b) Tensor parallelism (c) Sequence parallelism (Ours)

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.

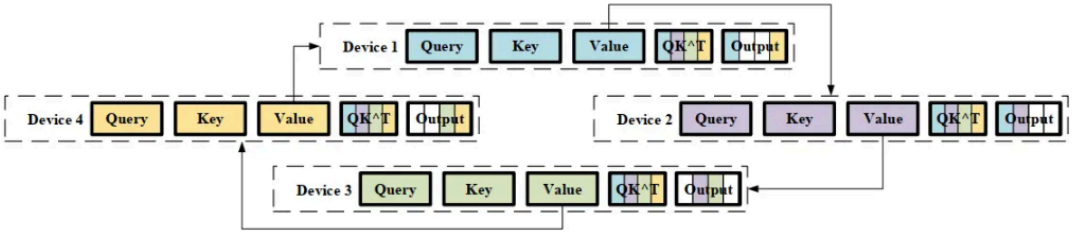
从图像中可以看到三种不同的并行方式：

1. **Pipeline Parallelism**（图 a）：在此并行方式下，模型的每一层被分配到不同的设备上，按顺序执行。在每个设备上进行微批次（Micro Batch）计算，然后在设备间传递数据，形成流水线结构。
2. **Tensor Parallelism**（图 b）：这种并行方法通过将每一层的张量分割成多个部分，并将这些部分分配到不同的设备上来实现并行。这使得每个设备只计算张量的部分内容，从而减少内存和计算的负担。
3. **Sequence Parallelism**（图 c）：这是序列并行的实现方式。在这种方式下，输入序列被拆分成多个部分，并且每个设备负责处理一个部分。需要注意的是，图中指出“Device 1 和 Device 2 共享相同的可训练参数”，这表明序列并行允许在不同设备之间共享模型的参数。这个特点使得设备不需要单独保存完整的序列，可以有效地训练超长的序列。

具体来说，**将输入序列分割成多个块，并将每个块输入到其相应的设备（即 GPU）中。**为了计算注意力输出，我们将环状通信与自注意力计算相结合，并提出了环自注意力（RSA）如下图所示。



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



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

Figure 2: Ring Self-Attention

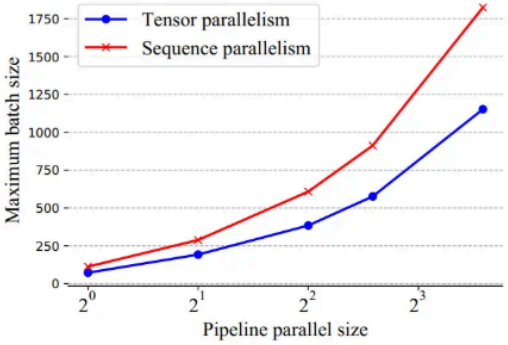
这张图展示的是 **环形自注意力（Ring Self-Attention）** 的实现方式。在自注意力计算中，输入序列的每个元素都会计算它与其他元素的关联（即注意力得分）。为了计算这些得分和生成输出，通常会使用查询（Query）、键（Key）和值（Value）这三种嵌入向量。

根据图中的描述，**环形自注意力**的核心概念是将查询、键和值在多个设备之间进行传递。

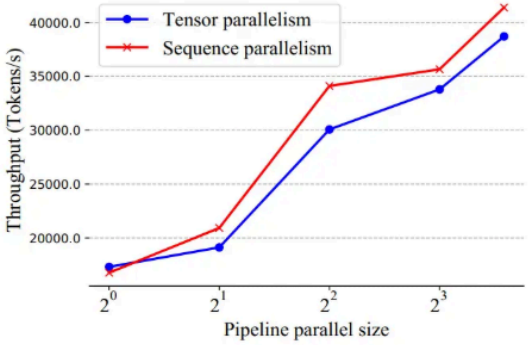
图解解析：

1. **(a) Transmitting key embeddings among devices to calculate attention scores**：
   - 在这一部分，图展示了如何将 **键（Key）** 嵌入向量在不同设备之间传输，用于计算 **注意力得分（Attention Scores）**。
   - 每个设备包含了不同的输入数据（Query、Key、Value），并且这些数据被传输到其他设备以计算注意力得分。图中的箭头表示了数据传输的方向和过程。
2. **(b) Transmitting value embeddings among devices to calculate the output of attention layers**：
   - 在这一部分，图展示了如何将 **值（Value）** 嵌入向量在设备之间传输，用于计算 **注意力层的输出。**
   - 同样，每个设备持有查询、键、值数据，而这些数据会通过设备间的传输进行协作计算，最终输出注意力结果。

实验表明，当按批量大小和序列长度进行缩放时，序列并行表现良好。

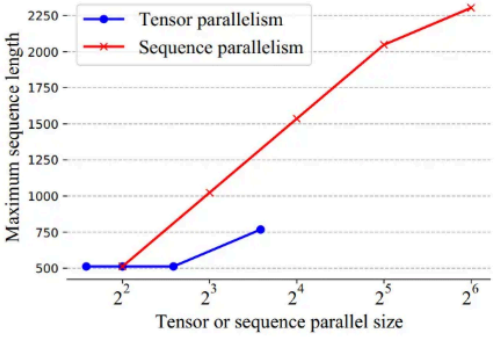


(a) Maximum batch size of BERT base scaling along pipeline parallel size

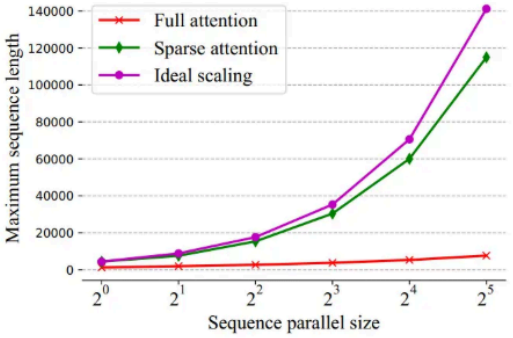


(b) Throughput of BERT base scaling along pipeline parallel size

Figure 4: Scaling with pipeline parallelism



(a) Maximum sequence length on BERT base



(b) Sequence length upper bound

Figure 5: Scaling with sequence length

当扩展到 64 个 NVIDIA P100 GPU 时，与张量并相比，该法分别实现了 13.7 倍和 3.0 倍的最大批量大小和序列长度。

通过稀疏注意力，序列可以处理具有超过 114K 个 Token 的序列，这比现有的在单个设备上保存整个序列的稀疏注意力运行长度超过 27 倍。

除此之外，与张量并行和流水线并行不同，序列并行不受超参数（例如： 注意力头数、层数）限制。 因此，只要序列长度能被序列并行大小整除，我们的序列并行就可以使用。

2.序列并行

> Megatron-LM 发表的论文：Reducing Activation Recomputation in Large Transformer Models， 主要是**减少模型显存**。

Megatron-LM 的初衷是**考虑通过其他方式分摊张量并行中无法分摊的显存**，因此提出了序列并行的方法。

虽然 Megatron-LM 引用了 Colossal-AI 的序列并行的这篇文章，但是这两者其实并不是一个东西。

Megatron-LM 只是借用了 Colossal-AI 把 Sequence 这个维度进行平均划分的思想。在 张量的基础上，**将 Transformer 层中的 LayerNorm 以及 Dropout 的输入按输入长度（Sequence Length）维度进行了切分，使得各个设备上面只需要做一部分的 Dropout 和 LayerNorm 即可。**

这样做的好处有：

1. LayerNorm 和 Dropout 的计算被平摊到了各个设备上，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设备上，进一步降低了显存开销。

在 Transformer 模型中，**LayerNorm** 和 **Dropout** 是两个非常重要的操作。它们在训练过程中对于模型的收敛性和性能非常关键，但它们也会带来显存开销。

- **LayerNorm**：在每一层的输入和输出上，进行归一化操作，通常是对每个序列的维度进行归一化，以保持每个神经元的输出有一个稳定的分布。这个操作会增加显存使用，因为需要存储中间计算结果（尤其是每个层的激活值）。
- **Dropout**：在训练过程中，Dropout 是一种常用的正则化技术，它通过随机将一部分神经元的输出置为零，减少模型对特定特征的依赖性。尽管它有助于防止过拟合，但它也会带来显存占用，因为需要存储 Dropout 的 mask（掩码）以及在计算过程中中间结果。

这些操作虽然有效，但在 **长序列**（长输入文本）情况下，显存需求会迅速增加，特别是在进行模型并行化时（例如，序列并行）。因此，如何在并行化的情况下减少这些操作带来的显存负担，是提高训练效率和可扩展性的关键。

Megatron-LM 的优化方式

Megatron-LM 在序列并行中的一个重要创新就是对 **LayerNorm** 和 **Dropout** 进行了优化，具体做法如下：

1. 按 Sequence Length 维度切分 LayerNorm 和 Dropout

在 Megatron-LM 中，LayerNorm 和 Dropout 操作的输入会根据 **输入序列的长度（Sequence Length）** 进行切分，分配到不同的设备上进行处理。这是一个非常巧妙的优化，因为它利用了 Colossal-AI 提出的序列并行思想，将序列长度维度平均划分给不同的设备。

假设有 n 个设备参与训练：

- **原本的做法**：在标准的 Transformer 中，LayerNorm 和 Dropout 是对整个输入序列（在每一层）进行操作的。这意味着每个设备需要存储和计算整个序列的所有激活值，包括 LayerNorm 和 Dropout 过程中的中间激活值。
- **切分后的做法**：Megatron-LM 切分了序列的长度，使得每个设备只处理部分序列的计算。具体来说，设备只需要做该部分序列的 **LayerNorm** 和 **Dropout** 计算。由于每个设备只处理部分数据，所以每个设备的显存负担大大降低。

2. 减少显存占用与计算资源浪费

切分 **LayerNorm** 和 **Dropout** 操作的好处有几个方面：

- **计算负担分摊**：LayerNorm 和 Dropout 的计算被分摊到各个设备上，不再由单个设备承担整个序列的计算任务。这有效减少了每个设备的计算负担，尤其是在输入序列长度非常长时。
- **显存使用优化**：由于每个设备只处理序列的一部分，涉及到的激活值和中间结果也会被切分，这样每个设备需要存储的激活值和梯度等中间结果的数量显著减少，降低了显存开销。特别是在多设备并行训练时，显存开销的降低尤为重要。

3. 进一步优化

这种切分方式不仅能减少显存占用，还能够提高训练的效率：

- **计算资源优化**：在传统的 Transformer 训练中，LayerNorm 和 Dropout 计算可能会成为瓶颈，因为它们需要对整个序列进行处理。而通过切分后，每个设备只处理自己负责的部分序列，能够提高设备间的计算效率，从而加速训练。
- **适应更长的序列**：在没有这种优化的情况下，长序列会导致显存压力很大，甚至无法在单个设备上进行训练。而通过切分，每个设备只需关注部分序列，使得可以处理更长的输入序列。

在 Megatron-LM 序列并行的这篇论文中，首先分析了 Transformer 模型运行时的显存占用情况。

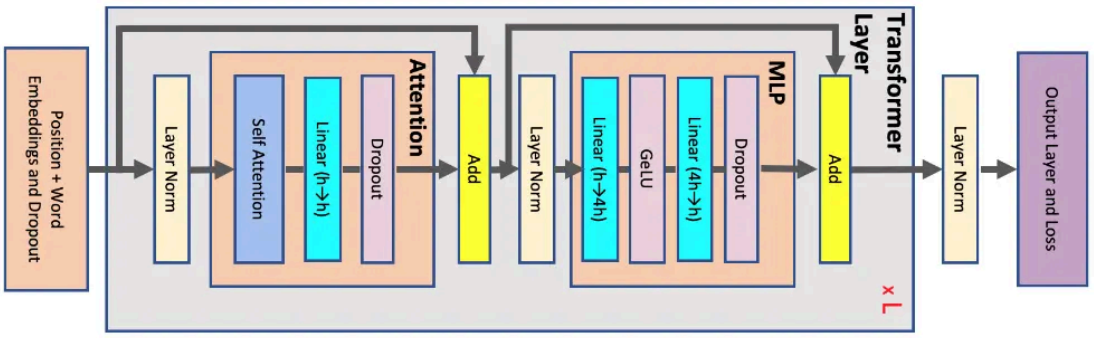


Figure 2: Transformer Architecture. Each gray block represents a single transformer layer that is replicated L times.

假设输入长度为 (s)，batch size 为 (b)，hidden dim 为 (h)，attention head 数量为 (a)，则每一层 Transformer（上图的灰色区域）的显存占用：

激活显存计算：在 Transformer 层中，显存的主要占用来源是激活值。这些激活值主要来自自注意力（Self-Attention）和前馈神经网络（MLP）计算，每一层 Transformer 需要存储以下内容：

1. **自注意力机制**：
   - 每个头的计算都会产生大小为$(s \times b \times h)$ 的激活值，因此自注意力的显存占用为 $(s \times b \times h \times 34)$（其中 34 是自注意力部分的常数因子，表示多个计算的组合）。
   - 另外，由于有 (a) 个 attention heads，显存占用还需要乘上 (5a)，这是因为每个注意力头的计算会产生独立的激活值。
2. **前馈网络（MLP）**：
   - 前馈网络同样会产生 $(s \times b \times h)$ 大小的激活值，随着前馈网络中多个线性层的计算，激活值的总量会增加。

因此，激活显存占用可以表示为：

$$\text{Activations memory per layer} = s \times b \times h \times (34 + 5a)$$

非共享的显存部分：当使用张量并行时，Transformer 层中的一些部分（如 LayerNorm 和 Dropout Mask）无法被分摊到多个设备。具体来说：

1. **LayerNorm**：
   - 每个 Transformer 层有两个 LayerNorm 操作，分别需要存储输入和输出，显存占用为 $(4 \times b \times s \times h)$。
2. **Dropout Mask**：
   - Dropout 操作需要存储二进制的 mask，因此显存占用为$(2 \times b \times s \times h)$。

这些部分不能通过张量并行分摊，因此它们占用的显存总和为：

$$\text{Non-shared memory} = 4 \times b \times s \times h + 2 \times b \times s \times h = 10 \times b \times s \times h$$

综上所述，每一层 Transformer 的显存占用计算如下：

$$\text{Activations memory per layer} = s \times b \times h \times (34 + 5a)$$

并且非共享部分的显存占用为 $(10 \times b \times s \times h)$。

> **Note**
>
> LayerNorm 操作：
>
> 1. **输入和输出激活**： LayerNorm 操作会对每一层的输入进行归一化，处理过程包括计算均值和方差，并基于这些值进行归一化。归一化的结果是经过修正后的输出。

2. **输入的存储**： 要计算 LayerNorm，我们需要保留输入的激活值。这些激活值大小为 $s \times b \times h$（即每个样本的输入序列长度、batch size 和 hidden dim）。
3. **输出的存储**： 归一化操作后的结果，即 LayerNorm 的输出，也需要存储。输出的大小与输入的激活值相同，同样为 $s \times b \times h$。
4. **均值和方差**： 在归一化过程中，我们还需要保存每个特征（即每个隐藏单元）的均值和方差，这通常占用 $2 \times b \times h$ 的显存（每个特征维度一个均值和一个方差）。

Dropout 操作：

对于 Dropout 操作，我们通常需要存储一个 **二进制掩码（mask）**，来决定哪些神经元将被“丢弃”（即被置为零）。每个掩码对应着输入数据的每个位置（即每个位置的神经元是否被丢弃）。

1. **Dropout Mask**： 在每一层 Dropout 操作中，我们需要为每个位置生成一个对应的二进制掩码。这个掩码的大小与输入数据相同，所以它的存储大小为 $b \times s \times h$。因此，**Dropout Mask** 的显存占用是 $2 \times b \times s \times h$，因为我们不仅需要存储掩码本身，还需要存储掩码中每个位置的状态（即它是否被丢弃）。这就导致了 Dropout Mask 占用的显存是 $2 \times b \times s \times h$。

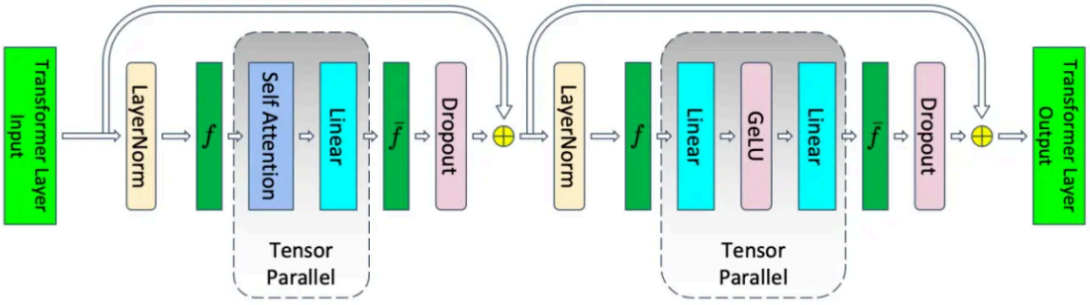


Figure 4: Transformer layer with tensor parallelism. f and $\bar{f}$ are conjugate. f is no operation in the forward pass and all-reduce in the backward pass. $\bar{f}$ is all-reduce in the forward pass and no operation in the backward pass.

根据图中的描述，**张量并行**将影响每个设备的显存占用。我们假设张量并行的大小为 t，这意味着输入数据在 t 个设备之间分割。假设张量并行大小为 t，每个设备每层 Transformer 的显存占用计算公式为：

$$\text{Activations memory per layer} = s \times b \times h \times (10 + \frac{24}{t} + 5a \times h \times t)$$

- s 是序列长度，b 是批次大小，h 是隐藏层维度，a 是注意力头的数量，t 是张量并行设备数。

下面开启张量并行以及序列并行，Transformer 层中的 LayerNorm 和 Dropout 块也会被切分，对 Tensor 在 Sequence 维度进行切分，切分数量等于张量并行大小

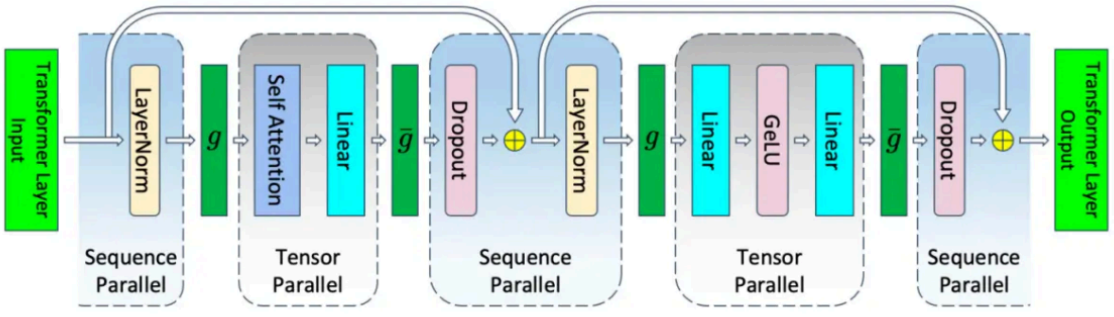


Figure 5: Transformer layer with tensor and sequence parallelism. g and $\bar{g}$ are conjugate. g is all-gather in the forward pass and reduce-scatter in the backward pass. $\bar{g}$ is reduce-scatter in forward pass and all-gather in backward pass.

每个设备每一层 Transformer 的显存占用为：

$$\text{Activations memory per layer} = s \times b \times h \times \left(\frac{10}{t} + \frac{24}{t} + \frac{5 \times a \times s}{t \times h}\right) = \frac{s \times b \times h}{t}\left(34 + \frac{5 \times a \times s}{h}\right)$$

做了额外的切分后，通信方式也会发生变化。

- **Transformer 层的张量并行通信** 是由正向传播的两个 **All-Reduce** 操作和反向传播的两个 **All-Reduce** 操作组成。
- 在 **序列并行** 中，由于对 **Sequence 维度** 进行了划分，**All-Reduce** 在这里不再合适。

为了在 **序列并行** 中收集在各个设备上进行计算产生的结果，需要插入 **All-Gather** 算子。

而为了使得 **张量并行** 所产生的结果可以传递到 **序列并行层**，需要插入 **Reduce-Scatter** 算子。

> **Note**
>
> **All-Gather** 是一种集体通信操作，它的作用是将所有参与设备的局部数据收集起来并传输到每个设备上，使得每个设备都能获得整个数据集。具体来说，在 **All-Gather** 操作中，每个设备持有一部分数据，然后将这些数据合并并发送到其他设备，每个设备最终会持有所有设备的数据。
>
> **All-Reduce** 是一种常用的集体通信操作，它的作用是将所有参与设备的局部数据进行“归约”操作（通常是加法、乘法等），然后将最终的结果广播到所有设备。简言之，在 **All-Reduce** 中，每个设备计算局部结果，并将这些局部结果合并，然后每个设备接收到合并后的结果。常见的应用场景是计算梯度累积和更新。
>
> 在 **序列并行** 的场景下，由于输入的序列已经被分割为多个部分，每个设备只负责处理其中一部分。如果使用 **All-Reduce** 操作，它会对每个设备的局部数据进行归约（如加法），然后将归约后的结果广播到所有设备。然而，序列并行中的数据并不是“归约”的数据，而是每个设备需要处理的不同部分。由于每个设备处理的是输入序列的不同部分，因此 **All-Reduce** 操作并不适合在序列并行中直接使用。

在下图中，g 代表的是前向传播的 **All-Gather** 操作，反向传播的 **Reduce-Scatter** 操作，$\overline{g}$ 则是相反的操作。

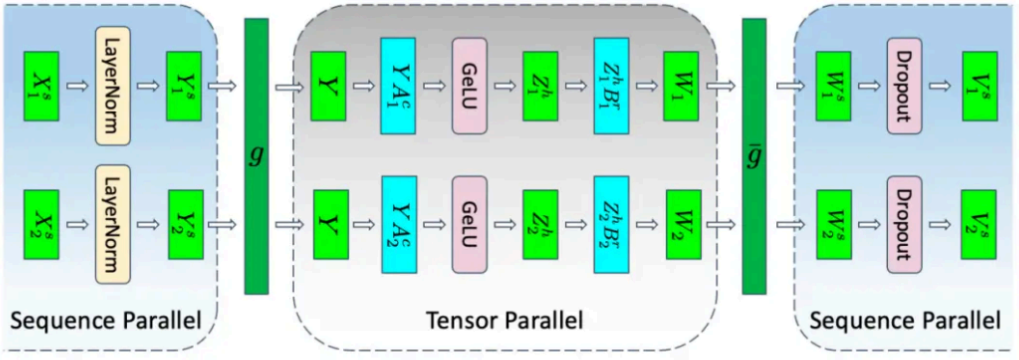


Figure 6: MLP layer with tensor and sequence parallelism. g and $\bar{g}$ are conjugate. g is all-gather in forward pass and reduce-scatter in backward pass. $\bar{g}$ is reduce-scatter in forward pass and all-gather in backward pass.

因此，我们可以清楚地看到，在 Megatron-LM 同时开启序列并行和模型并行时，每一个 Transformer 层完成一次前向传播和反向传播一共有 4 个 All-Gather 和 4 个 Reduce-Scatter 算子。乍一看，通信的操作比 Megatron-LM 仅开启张量并行多，但其实不然。因为，一个All-Reduce就相当于一个 Reduce-Scatter 和一个 All-Gather ，所以他们的总通信量是一样的。

通过添加序列并行并没有增加额外的通信开销，反而在后向传播代码的实现上，还把 Reduce-Scatter 和权重梯度的计算做了重叠，进一步减少了通信所占用的时间，使得提高设备的FLOPs Utilization成为了可能。

通过对Transformer层中所有Activation的消耗进行计算，发现在Transformer层里有一些操作是产生的激活值大，但计算量小。因此，就考虑干掉这一部分的激活值，通过选择性的进行激活重新计算（Selective Activation Recomputation）来进一步降低显存。与此同时，其他的激活值就通通保存，以节省重计算量。

通过对激活值的占比分析，序列并行降低了4成左右的激活值开销。选择性激活重新计算（selective activation recompute）也降低了4成左右的激活值开销。当两个特性都打开的时候，总共可以降低8成左右的激活值开销，尽管比全部激活值重计算的结果要稍高，但是在吞吐率上的提升还是非常的明显的。

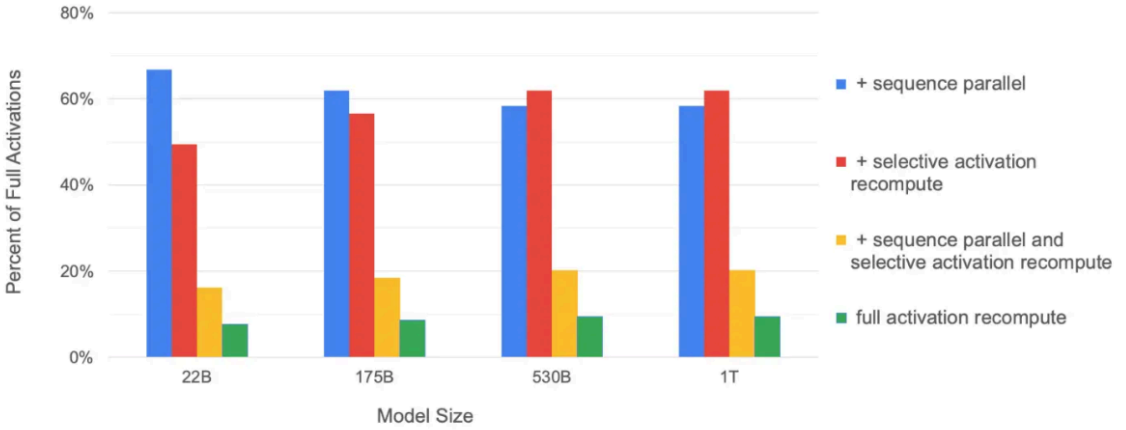


Figure 7: Percentage of required memory compared to the tensor-level parallel baseline. As the model size increases, both sequence parallelism and selective activation recomputation have similar memory savings and together they reduce the memory required by $\sim 5\times$.

3.Pytorch中的序列并行

上一篇张量并行的文章中提到 Pytorch 从 2.0.0 开始已经开始支持张量并行了。参考 Megatron-LM 的序列并行，目前在 Pytorch 中，也已经支持序列并行了，不过还没有 Release，具体示例如下所示：

```
# 通过设备网格根据给定的 world_size 创建分片计划
device_mesh = DeviceMesh("cuda", torch.arange(0, args.world_size))

# 创建模型并移动到GPU
model = ToyModel().cuda(rank)

# 为并行化模块创建优化器
LR = 0.25
optimizer = torch.optim.SGD(model.parameters(), lr=LR)

# 根据给定的并行风格并行化模块，这里指定为序列并行
model = parallelize_module(model, device_mesh, SequenceParallel())

# 对分片模块执行多次前向/后向传播和优化器对参数进行更新。
for _ in range(args.iter_nums):
    # 对于 SP，所有rank的输入可以不同。
    inp = torch.rand(20, 10).cuda(rank)
    output = model(inp)
    output.sum().backward()
    optimizer.step()Copy to clipboardErrorCopiedCopy to clipboardErrorCopied
```

4.总结

总的来说，Colossal-AI 的序列并行是为了打破单设备上序列长度的限制。而 Megatron-LM 的序列并行是在显存上面下了功夫，可以用更少的设备去运行大模型。除此之外，从文章细节里面可以看到，部分的计算的冗余被消除了，且重叠了一部分的通信，使得设备可以花更多的时间用于计算上面。虽然，Colossal-AI 和 Megatron-LM 都有序列并行，但是两者解决的问题、方法都不一样。除此之外，在Pytorch中，也已经支持序列并行了。