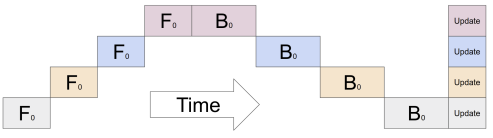
大模型与分布式技术之Megatron-LM

1、并行化技术总概：

① 流水线并行（PP）：

先提一句模型并行：存在单卡时，无法将大模型容纳下，一个解决办法是将模型分成不同的层，每一层都能放到一张卡上进行训练。此时模型做一次forward与backward的形式如下：



其中下标表示batch编号，这里只有一个batch，因此下标都是0。每一行表示一个GPU。每一列表示timestep。

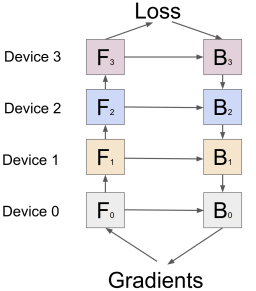
这张图的含义是：在卡0上做完一次forward，然后将卡0上最后一层的输入传给卡1，继续做forward，直到四张卡都做完forward后，再依次做backward。等把四张卡上的backward全部做完后，最后一个时刻统一更新每一层的梯度。

如此确实能够训练模型，这样操作会带来两个问题：

（1）卡的利用率太低。假设有张卡，而单张卡上做一次forward和backward的时间为：.则带来的空余率就是：

当K越大，即卡的数量越多时，空置的比例接近1，即卡的资源都被浪费掉了。

（2）产生大量的计算中间结果，从而占据内存。

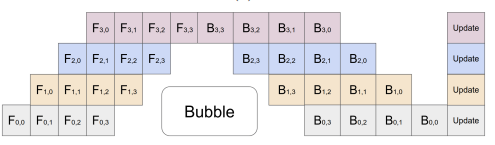


在做backward计算梯度的过程中，需要用到每一层的中间结果z。假设模型有L层，每一层的宽度为d，则对于每张卡，不考虑其参数本身的存储，额外的空间复杂度为，其中N为mini-batch的大小。从这个复杂度可以看出，随着模型的增大，N，L，d三者的增加可能会平滑掉K增加带来的GPU内存收益。

应对思路：Gpipe中给出的解法：

（1）切分micro-batch：

流水线并行的核心思想是：在模型并行的基础上，进一步引入数据并行的办法，即把原先的数据再划分成若干个batch，送入卡中进行训练。未划分前的数据，叫mini-batch.在mini-batch上再划分的数据，叫micro-batch.



其中，第一个下标表示卡的编号，第二个下标表示micro-batch编号。假设将mini-batch划分为M个，则流水线并行下，bubble的时间复杂度为：.Gpipe通过实验证明，当时，bubble产生的空转时间占比对最终训练时长影响是微小的，可以忽略不计。

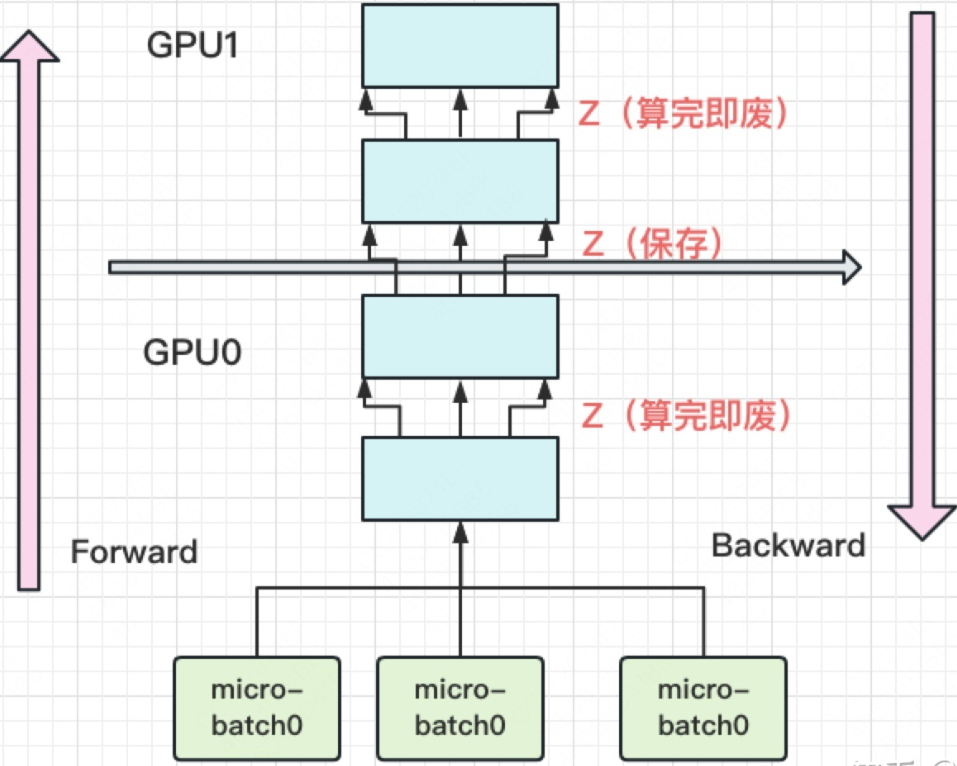
将batch切好，并逐一送入GPU的过程，就像一个流水生产线一样（类似于CPU里的流水线），因此也被称为Pipeline Parallelism(PP).

为了使气泡时间分数小，我们需要𝑚>>𝑝。然而，对于如此大的𝑚，这种方法的内存占用很高，因为它需要在训练迭代的整个生命周期内为所有𝑚微批保存隐藏的中间激活(或者使用激活重新计算时每个管道阶段的输入激活)

(2)active checkpoint(re-materialization)

解决了卡的空置问题，提升了卡计算的整体效率。接下来，就要解决卡的内存问题了。前文说过，随着模型的增加，每块卡中存储的中间结果也会越大。对此，Gpipe采用了一种非常简单粗暴但有效的办法：用时间换空间，在论文里，这种方法被命名为re-materalization，也称为active checkpoint。

具体来说就是，出了某些variable缓存之外几乎不保存中间结果，forward结束即丢弃，等到backward时候再计算一次forward（也叫重计算）。

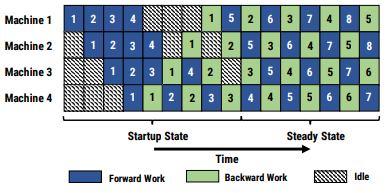


每张卡上，只保存来自上一张的最后一层输入z，其余的中间结果算完就废。等到backward的时候再由保存下来的z重新进行forward来算出。

现在来计算每块GPU峰值时刻的内存：每块GPU峰值时刻存储大小 = 每块GPU上的输入数据大小 + 每块GPU在forward过程中的中间结果大小，即为.其中N为初始mini-batch的大小。作为对比，在没有重计算与划分时，GPU的空间复杂度为。

最后补充一点，在micro-batch的划分下，Batch Normalization会收到影响，解决的方式是采用mini-batch的均值与方差进行计算，同时追踪全部mini-batch的移动平均与方差，以便在测试阶段进行使用。Layer Normalization不受影响。

PipeDream中给出的解法：1F1B



1F1B使得每个stage的反向计算可以尽早得到调度执行，因此可以及时释放对应的内存，减少内存占用。但此方式存在两个问题：

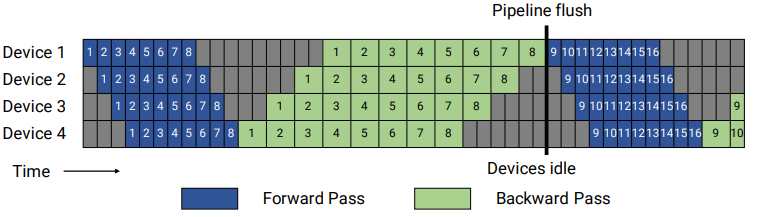
同一个mini-batch的前向和后向weight不一致（如上图的Machine 1的mini-batch 5的正反向）

要解决此问题，需要为每个mini-batch引入weight副本，同一个mini-batch的前向和反向使用相同的weight副本。此方式解决了weight不一致的问题，但引入了多份副本，又增加了内存占用。

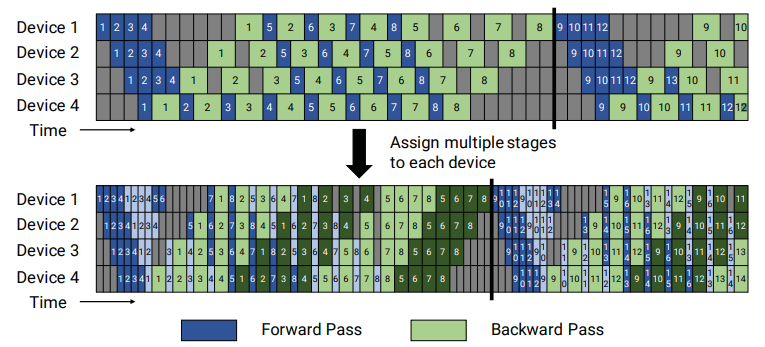
同一个mini-batch的不同stage的参数不一致（mini-batch 5的正向在Machine 1和2上）；

要解决此问题，需要引入Vertical Sync机制，每个mini-batch在进入Pipeline开始使用最新的weight，并且将信息传递给后续stage，所有stage使用相同的weight副本。

Megatron-LM中给出的解法：PipeLine-Flush，并且进一步增加interleaved PipeLine来优化Bublle Time。



这里是GPipe的方式，每一个batch被分成了8个micro-batch，首先执行批处理中所有微批的正向传递，然后执行所有微批的向后传递。优化器是分级的，权重参数在管道刷新时更新，以确保严格的优化器语义，从而导致设备闲置和管道气泡。



这里一个batch被分成了4个micro-batch，上图是PipeDream中提出的1F1B的方式，下图是Megatron-LM中提出的增加了interleave的方式。

采用PipeDream-Flush调度，在这个调度过程中，首先进入一个热身阶段，worker执行不同数量的向前传递，如图(顶部)所示。该计划将运行中的微批数量(向后传递未完成且需要维持激活的微批数量)限制在管道的深度，而不是一个批中的微批数量。在热身阶段之后，每个worker进入稳定状态，worker进行一次向前传递，然后进行一次向后传递(简称1F1B)。最后，在批处理结束时，完成所有剩余的飞行微批的向后传递。在气泡中花费的时间与这个新调度相同，但未完成的正向传递的数量最多是PipeDream-Flush调度的管道阶段数量。因此，这个调度要求为𝑝或更少的微批次(与GPipe调度的𝑚微批次相比)存储激活。因此，当𝑚>>𝑝时，PipeDream-Flush比GPipe更节省内存。

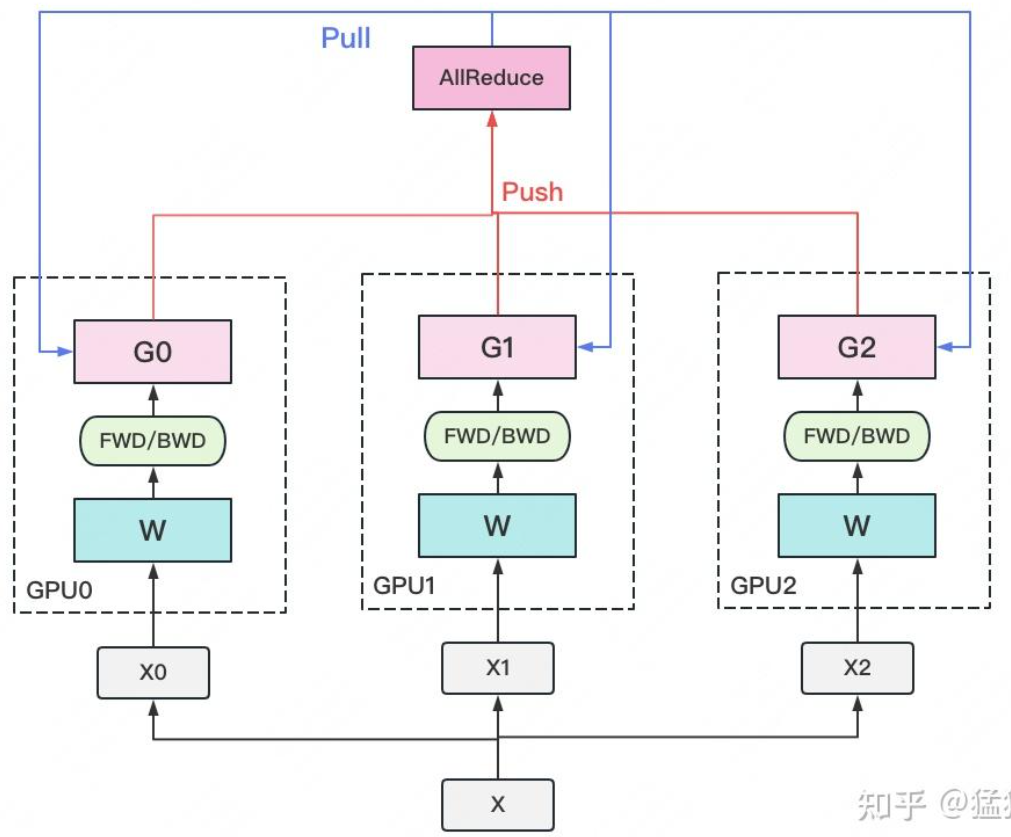
为了减少管道气泡的大小，每张卡可以为多个层子集(称为模型块)执行计算，而不是单个连续的层集。例如，如果每个设备之前有4层(即设备1有1 - 4层，设备2有5 - 8层，等等)，可以让每张卡为两个模型块执行计算(每个模型块有2层)，即设备1有1、2、9、10层；设备2具有第3、4、11、12层等等。该方案为管道中的每个设备分配了多个管道级(每个管道级的计算量比以前少)。和以前一样，我们可以使用该调度的“全部向前，全部向后”版本，但这将占用大量内存(与𝑚成正比)。相反，我们开发了一个交错调度，它适应了之前的内存高效1F1B调度。这个新的调度如图（底部）所示，它要求批处理中的微批数量是管道并行度的整数倍(管道中的设备数量)。例如，对于4个设备，批次中的微批数量必须是4的倍数。采用interleaved的方式即每张卡被指派多个chunk，可以设其大小为v（图中为2），深色的表示第一个chunk，浅色的表示第二个chunk。气泡的尺寸变得更小了（在interleaved之间管道刷新变得更快了）。interleaved Pipeline将气泡时间降低了v倍，同时也引进了v倍的通信量。

② 数据并行（DP）：

数据并行的核心思想是：在各张卡上都拷贝一份完整模型，各自吃一份数据，算一份梯度，最后对梯度进行累加来更新整体模型。理念不复杂，但到了大模型场景，巨大的存储和卡间的通讯量，就是系统设计要考虑的重点了。将递进介绍三种主流数据并行的实现方式：

* DP（Data Parallelism）：最早的数据并行模式，一般采用参数服务器(Parameters Server)这一编程框架。实际中多用于单机多卡
* DDP（Distributed Data Parallelism）：分布式数据并行，采用Ring AllReduce的通讯方式，实际中多用于多机场景
* ZeRO：零冗余优化器。由微软推出并应用于其DeepSpeed框架中。严格来讲ZeRO采用数据并行+张量并行的方式，旨在降低存储。

（1）数据并行（DP）



一个经典数据并行的过程如下：

* 1. 若干块计算GPU，如图中GPU0~GPU2；1块梯度收集GPU，如图中AllReduce操作所在GPU。
  2. 在每块计算GPU上都拷贝一份完整的模型参数。
  3. 把一份数据X（例如一个batch）均匀分给不同的计算GPU。
  4. 每块计算GPU做一轮FWD和BWD后，算得一份梯度G。
  5. 每块计算GPU将自己的梯度push给梯度收集GPU，做聚合操作。这里的聚合操作一般指梯度累加。当然也支持用户自定义。
  6. 梯度收集GPU聚合完毕后，计算GPU从它那pull下完整的梯度结果，用于更新模型参数W。更新完毕后，计算GPU上的模型参数依然保持一致。

聚合再下发梯度的操作，称为AllReduce。

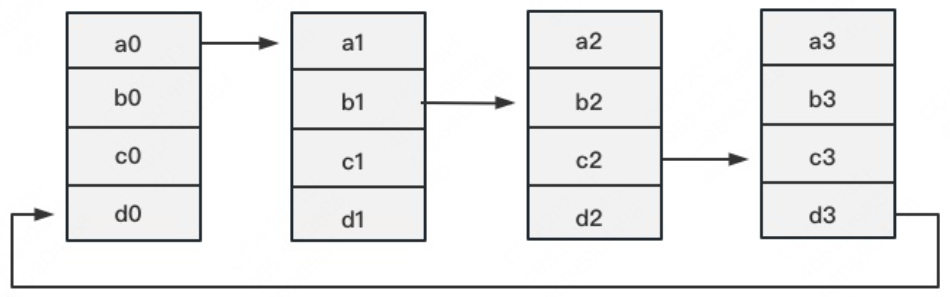
需要注意的是，DP框架中参数服务器容易成为通信的瓶颈，所以一般会考虑延迟1的asynchronous update

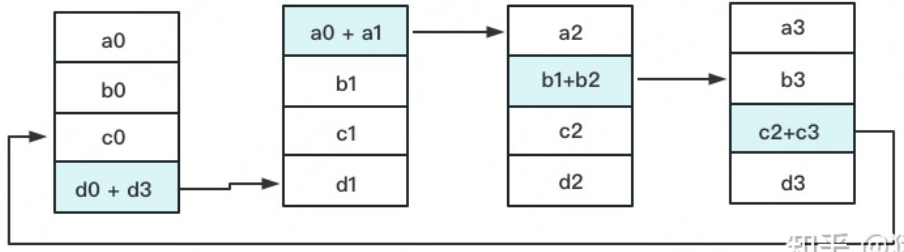
（2）分布式数据并行（DDP）：

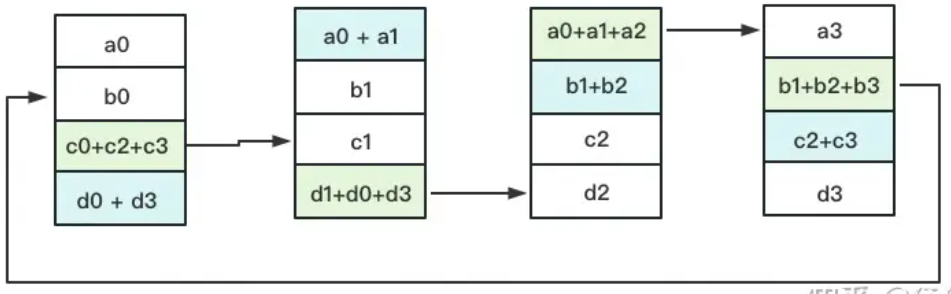
DP一般用于单机多卡场景。因此，DDP作为一种更通用的解决方案出现了，既能多机，也能单机。DDP首先要解决的就是通讯问题：将Server上的通讯压力均衡转到各个Worker上。实现这一点后，可以进一步去Server，留Worker。此时不使用AllReduce而是采取Ring-AllReduce = Reduce-Scatter + All-Gather

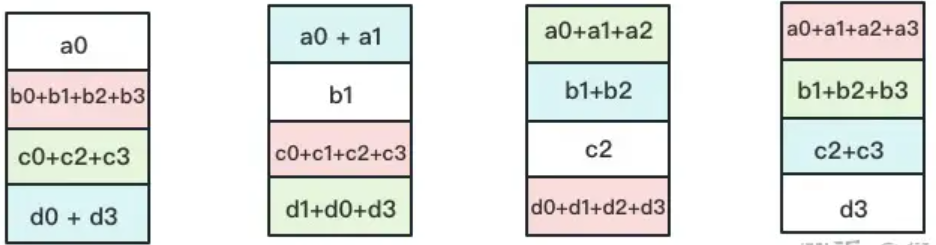
Reduce-Scatter：

定义网络拓扑关系，使得每张卡只和其相邻的两张卡通讯，每次发送对应位置的数据进行累加，每一次累加更新都形成一个拓扑环，因此被称为Ring.





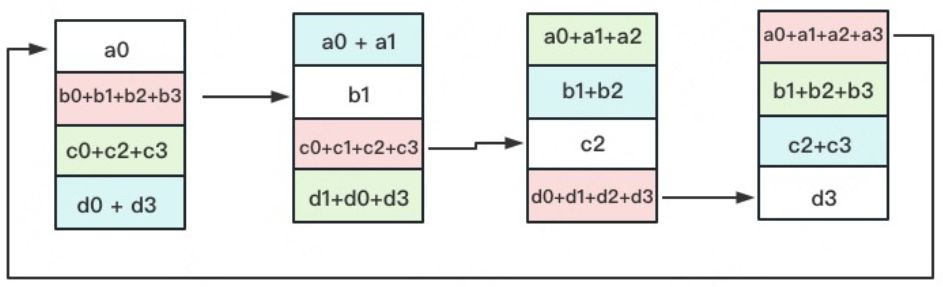


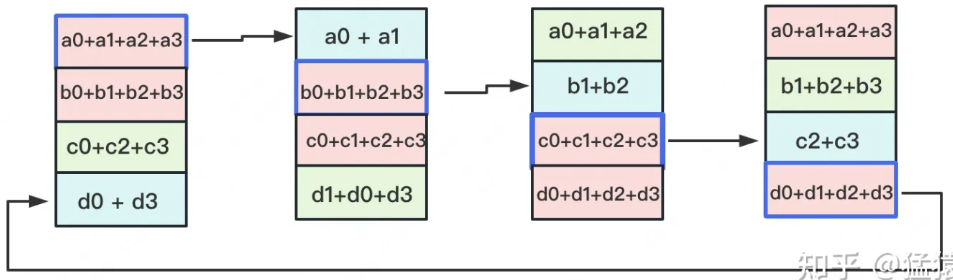


3次更新之后，每张卡上都有一块数据拥有了对应位置完整的聚合（图中红色）。此时，Reduce-Scatter阶段结束。进入All-Gather阶段。目标是把红色块的数据广播到其余GPU对应的位置上。

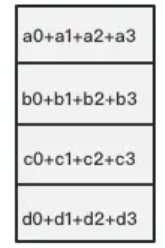
All-Gather：

依然按照“相邻GPU对应位置进行通讯”的原则，但对应位置数据不再做相加，而是直接替换。All-Gather以红色块作为起点。





以此类推，同样经过3轮迭代后，使得每块GPU上都汇总到了完整的数据，变成如下形式：



通讯开销分析：

假设模型参数W的大小为，卡的数量为N，则梯度大小也为，每个梯度块的大小为，对单卡来说（只算其send通讯量）：

Reduce-Scatter阶段，通讯量为：

All-Gather阶段，通讯量为：

单卡总通讯量为2.随着N的增大，可以近似为2。全卡总通讯量为2N.

对DP来说，它的Server承载的通讯量是N，Workers为N，全卡总通讯量依然为2N.虽然通讯量相同，但搬运相同数据量的时间却不一定相同。DDP把通讯量均衡负载到了每一时刻的每个Worker上，而DP仅让Server通信。

（3）ZeRO（零冗余优化器）：

先看存储消耗分类，两大类——模型状态与剩余状态：

模型状态：优化器状态、梯度、参数

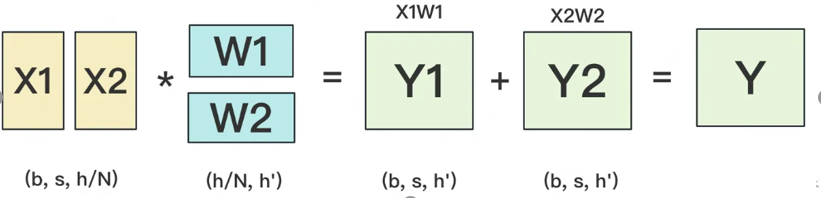
剩余状态：激活值、临时存储、不可用的碎片

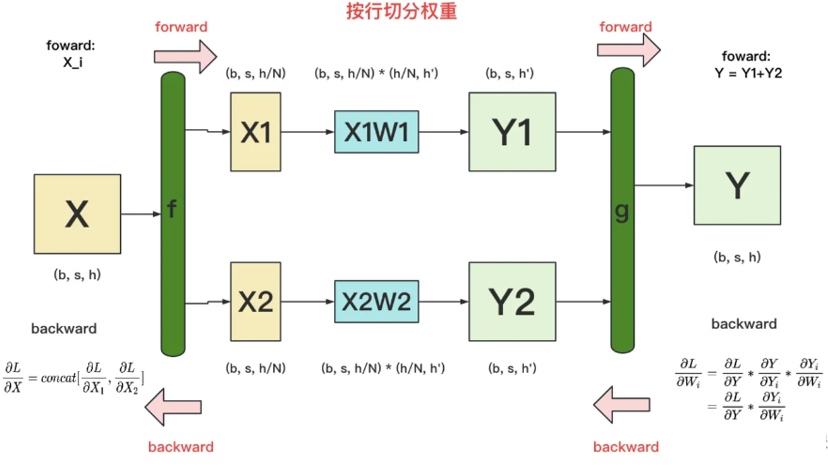
③ Tensor Parallel 张量并行：

基本思想就是将模型的参数进行拆分，放到不同的卡上进行独立计算，之后再进行聚合。

考虑两种分割方式：行切割与列切割

（1）行切割：



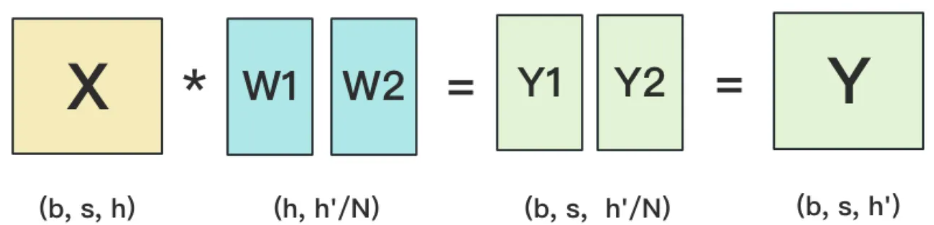


f与g是两个共轭算子：

f：forward中按列split输入；backward中all-gather梯度，代码定义在class \_ScatterToModelParallelRegion

g:forward中AllReduce输出；backward中直接输出梯度，无需做任何通讯（因为经过g的forward，每张卡上已经拥有Yi与Y，则根据图中的backward公式可知，每张卡可独立计算梯度）。代码定义在class \_ReduceFromModelParallelRegion

（2）列切割：



f：forward中，直接copy输入；在backward中对梯度做AllReduce。在代码里定义为class \_CopyToModelParallelRegion.

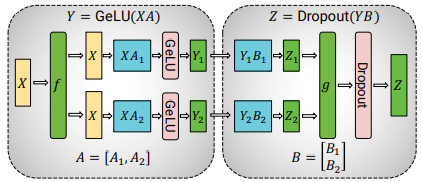
g：forward中，all-gather输出；在backward中对梯度做split（每张卡经过all-gather已有完整的Y了，因此以Y为起点计算梯度之后，沿着列做split就能得到Y1与Y2的梯度）。在代码里定义为class \_GatherFromModelParallelRegion

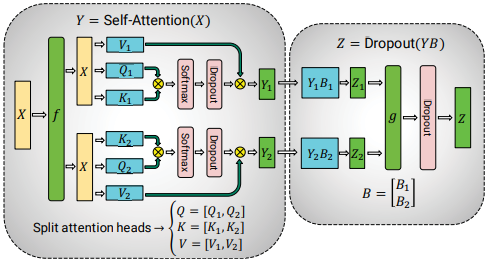
f的forward计算：把输入X拷贝到两张卡上，每张卡即可独立做forward计算。

g的forward计算：每张卡上的forward的计算完毕，取得Z1和Z2后，卡间做一次AllReduce，相加结果产生Z。

g的backward计算：只需要把拷贝到两张卡上，U就能各自独立做梯度计算。

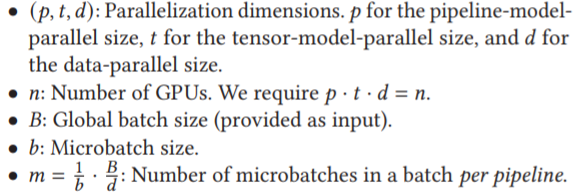
f的forward计算：当当前层的梯度计算完毕，需要传递到下一层继续做梯度计算时，需要求得。则此时两块GPU做一次AllReduce，把各自的梯度和相加即可。





f 是正向传递中的恒等算子，反向传递中的allreduce算子；g则相反。

为什么对A采用列切割，对B采用行切割呢？这样设计的原因是，尽量保证各张卡上的计算相互独立，减少通讯量。对A来说，需要做一次GELU的计算，而GELU函数是非线形的，也就意味着，如果对A采用行切割，必须在做GELU前，做一次AllReduce，这样就会产生额外通讯量。但是如果对A采用列切割，那每块GPU就可以继续独立计算了。一旦确认好A做列切割，那么也就相应定好B需要做行切割了。



不同卡之间的通信量也受𝑝和𝑡的值的影响。管道模型的并行性提供了更便宜的点对点通信。另一方面，张量模型的并行性使用all-reduce通信(在向前和向后传递中各有两个all-reduce操作)。

使用管道并行，每个微批的每对连续设备(无论是向前传递还是向后传递)之间需要执行的通信总量为bsh，其中𝑠是序列长度，而h是隐藏大小。

在张量模型并行性下，在每一层的前向和后向传递中，总尺寸bsh的张量需要在𝑡模型副本之间分别进行两次all-reduce，导致对每个device上的每层的每个微批总通信为。每个设备通常有多个层，每个微批的每个设备的张量并行通信总量为。是每个pipeline中的模型层数。

图解大模型一：<https://zhuanlan.zhihu.com/p/629121480>

图解大模型二：<https://zhuanlan.zhihu.com/p/634377071>

deepspeed使能：<https://www.deepspeed.ai/tutorials/megatron/>

计算logits：

首先，在使用\_VocabParallelCrossEntropy 计算交叉熵前，需要计算logit。这时调用parallel\_lm\_logits 函数，将模型最后一层的输出X（这个X已经在TP组内AllReduce了），乘上当前进程上维护的输入层WE的转置（输入层和输出层共用一套embedding），得到当前进程的logit Y\_i，同时我们选择不对输出logit做AllReduce。

你可能会有一个疑惑：在Transformer中，输出层会额外训练一个线性矩阵，来计算logit；为什么在gpt中，可以用输入层WE的转置来代替这个线性矩阵？

这个问题的答案，对理解Megatron交叉熵计算也至关重要。可将X\*WE^T结果理解成“X与WE间的相似度”，例如对Y1来说，它的第一行中的每个logit，表示第一个token与词表里每个词的相似度。

注意到每个进程上只维护部分WE。例如，假设词表共有10个单词，WE1维护前5个单词，WE2维护后5个单词。因此再严格来说：对Y1，它的第一行中的每个logit，表示第一个token与词表中前5个词的相似度；对Y2，它的第一行中的每个logit，表示第一个token与词表中后5个词的相似度。

计算交叉熵

知道了logit的含义，来看交叉熵计算。

首先做了一系列求max的计算，得到基于全局的max(logit)，再将orig\_logit - max(logit)，得到处理后的结果。这步理解起来不难，主要目的是为了防止计算溢出。

接下来，就是基于logit算loss了。

每个进程上都有一份(b, s)维度的真值，它表示每个token的真值是哪个词（词用id表示）。基于这份真值，在Y\_i上找出真值位置的logit。例如：seq\_length = 3，即我们需要对3个token去做预测，假设前两个token的真值在第1个进程所维护的WE1中，最后一个token的真值在第2个进程所维护的WE2中。那么我们去Y1的前两行里，取出真值位置的logit，这个logit表示“token与真值的相似度”，去Y2的最后一行里做同样操作。

这样，就能得到L1和L2，和真值位置不对应的地方，统一填充0。随后对L1和L2做AllReduce，得到L。L中的每行表示“token与真值间的相似度"

现在，回来对Y1和Y2的每一行求sum(e^logit)，得到e1和e2。将e1和e2做AllReduce，得到e。e中的每行表示“token和词表中所有词相似度的总和”

期望（token和词表中所有词相似度的总和-token与真值间的相似度) /token和词表中所有词相似度的总和这个值最小，这个差值就是最终的loss。

*# 列切割中的f与g*

**class** **\_CopyToModelParallelRegion**(torch**.**autograd**.**Function):

"""Pass the input to the model parallel region."""

*# 列切割下的f算子*

*# forward：copy输入*

*# backward：对梯度做AllReduce*

@staticmethod

**def** **symbolic**(graph, input\_):

**return** input\_

@staticmethod

**def** **forward**(ctx, input\_):

**return** input\_

@staticmethod

**def** **backward**(ctx, grad\_output):

**return** \_reduce(grad\_output)

**class** **\_GatherFromModelParallelRegion**(torch**.**autograd**.**Function):

"""Gather the input from model parallel region and concatinate."""

*# 列切割中的g算子*

*# forward：All-Gather输出*

*# backward：对梯度，沿着列方向做split*

@staticmethod

**def** **symbolic**(graph, input\_):

**return** \_gather(input\_)

@staticmethod

**def** **forward**(ctx, input\_):

**return** \_gather(input\_)

@staticmethod

**def** **backward**(ctx, grad\_output):

**return** \_split(grad\_output)

*# 行切割中的f和g算子*

**class** **\_ScatterToModelParallelRegion**(torch**.**autograd**.**Function):

"""Split the input and keep only the corresponding chuck to the rank."""

*# 行切割中的f算子*

*# forward：沿列split输入*

*# backward：all-gather梯度*

@staticmethod

**def** **symbolic**(graph, input\_):

**return** \_split(input\_)

@staticmethod

**def** **forward**(ctx, input\_):

**return** \_split(input\_)

@staticmethod

**def** **backward**(ctx, grad\_output):

**return** \_gather(grad\_output)

**class** **\_ReduceFromModelParallelRegion**(torch**.**autograd**.**Function):

"""All-reduce the input from the model parallel region."""

*# 行切割中的g算子*

*# forward：AllReduce输出*

*# backward：正常计算梯度，GPU间无需做任何通讯*

@staticmethod

**def** **symbolic**(graph, input\_):

**return** \_reduce(input\_)

@staticmethod

**def** **forward**(ctx, input\_):

**return** \_reduce(input\_)

@staticmethod

**def** **backward**(ctx, grad\_output):

**return** grad\_output

模型分析

① 模型架构：

现在LLM提出的一种模型架构是decoder-only：理论上是因为encoder的双向注意力会存在低秩问题，这可能会削弱模型的表达能力，就生成式任务而言，引入双向注意力并无实质性好处。decoder-only模型在没有任何tuning数据的情况下、zero-shot表现最好，而encoder-decoder则需要在一定量的标注数据上做multitask finetuning才能激发最佳性能。

decoder-only在参数量不太大时就更具有更强的zero-shot性能、更匹配主流的自监督训练范式；而在大参数量的加持下，具有了涌现能力（emergent abilities）后、可以匹敌encoder-decoder做finetuning的效果；在In Context的环境下、又能更好地做few-shot任务。

encoder-decoder模型的火爆更多是依赖于在特定标注数据上的训练。encoder-decoder有两个特点可能会使得它在以下两类任务上有优势：

* encoder的多样性。许多多模态工作(BLIP、ALBEF、SimVLM)上可以看见encoder-decoder的影子，因为encoder可以用来encode多种模态的信息，而decoder-only的多模态工作相对较少（微软近日的visual-ChatGPT更多偏向模型级联pipeline）。
* Deep Encoder+Shallow Decoder的推理优势。Encoder-decoder架构本来在计算效率上（以FLOPs衡量）就是优于其他LM的，且工业界目前常使用Deep Encoder+Shallow Decoder的组合，由于encoder本身并行度高，这类encoder-decoder的infer速度远超大型的decoder-only。在有较多标注数据的任务上，encoder-decoder还是具有成本优势的。

② BLOOM模型：Decoder-Only：与原始的transformer架构相比，优化点在于：

（1）AliBi位置编码：ALiBi位置嵌入不是向嵌入层添加位置信息，而是根据键和查询的距离直接衰减注意力分数。最初的动机是其推断更长序列的能力，但后来发现即使在原始序列长度下，它也能带来更平滑的训练和更好的下游性能，效果好于Rotory Embedding与both learned

（2）Embedding之后加入LayerNorm

（3）其多头注意力就是在标准的注意力多头上加上ALiBi（单个注意力进行拼接）

模型结构总述如下：

* Decoder-Only architecture
* Layer Normalization applied to word embedding layer
* AliBipositional embedding，add GeLUa activation function
* 176B parameters:

70 layers, 112 attention heads

Hidden layers are 14336-dimensional

Sequence length of 2048tokens used

Objective function: Cross Entropy with mean reduction