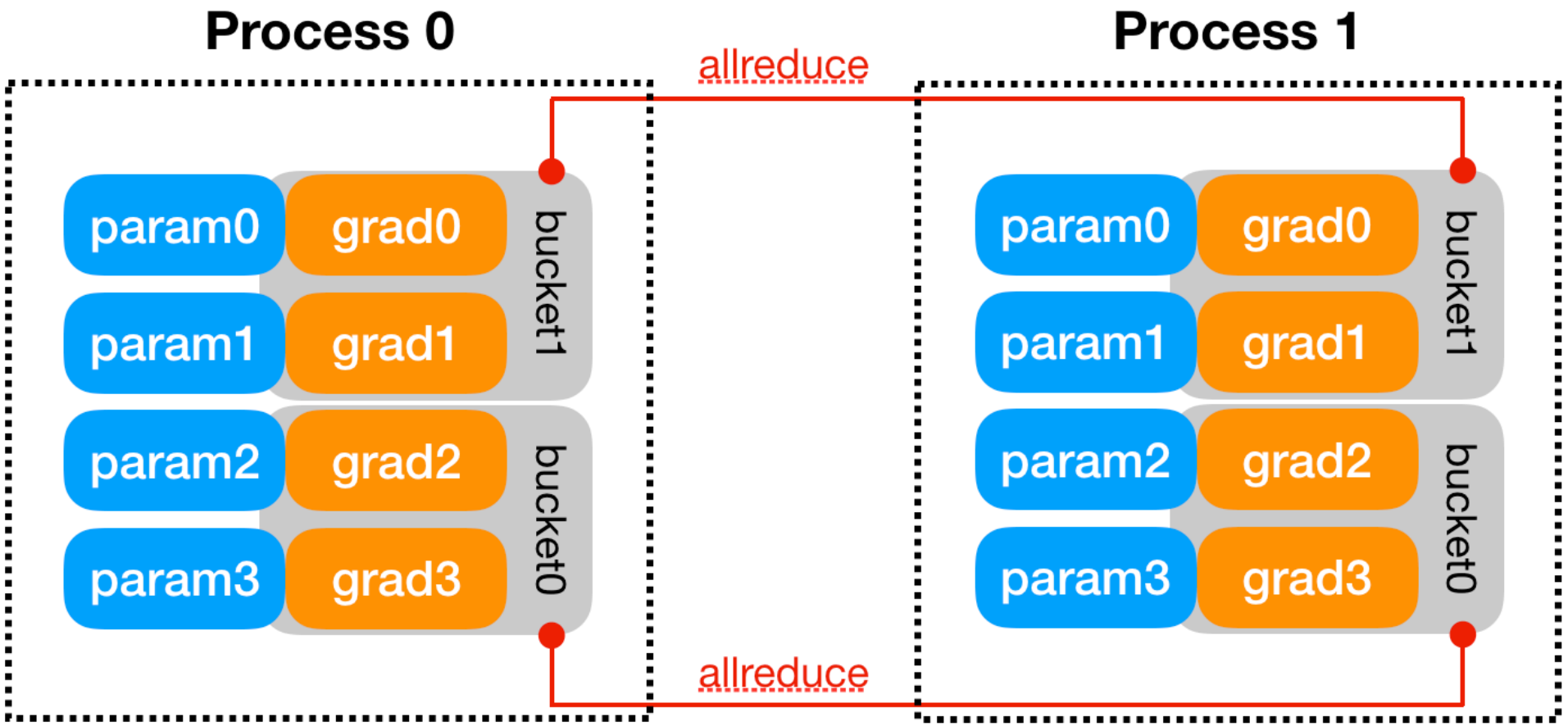
MindSpore分布式特性详解

目录

目录

数据并行


Process 0
Process 1
allreduce
allreduce
param0
grad0
param1
grad1
param2
grad2
param3
grad3
bucket1
bucket0
param0
grad0
param1
grad1
param2
grad2
param3
grad3
bucket1
bucket0

数据并行

数据并行过程:

- 每一张卡上放置相同的模型参数、梯度、优化器状态
- 不同的卡送入不同的数据训练
- 反向传播获得梯度后，进行AllReduce

数据并行的问题:

- 要求单卡可以放下模型
- 多卡训练时内存冗余

BERT的数据并行预训练

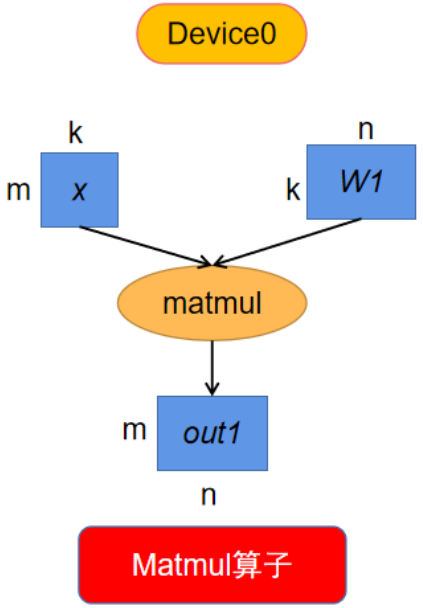
```
# 6. Pretrain
mean = _get_gradients_mean()
degree = _get_device_num()
grad_reducer = nn.DistributedGradReducer(optimizer.parameters, mean, degree)


def train_step(input_ids, input_mask, masked_lm_ids, masked_lm_positions, masked_lm_weights, \
               next_sentence_label, segment_ids):
    status = init_register()
    input_ids = ops.depend(input_ids, status)
    (total_loss, masked_lm_loss, next_sentence_loss), grads = grad_fn(input_ids, input_mask, segment_ids, \
                         masked_lm_ids, masked_lm_positions, masked_lm_weights, next_sentence_label)
    grads = clip_by_global_norm(grads, clip_norm=1.0)
    grads = grad_reducer(grads)
    status = all_finite(grads, status)
    if status:
        total_loss = loss_scaler.unscale(total_loss)
        grads = loss_scaler.unscale(grads)
        total_loss = ops.depend(total_loss, optimizer(grads))
    total_loss = ops.depend(total_loss, loss_scaler.adjust(status))
    return total_loss, masked_lm_loss, next_sentence_loss, status
```

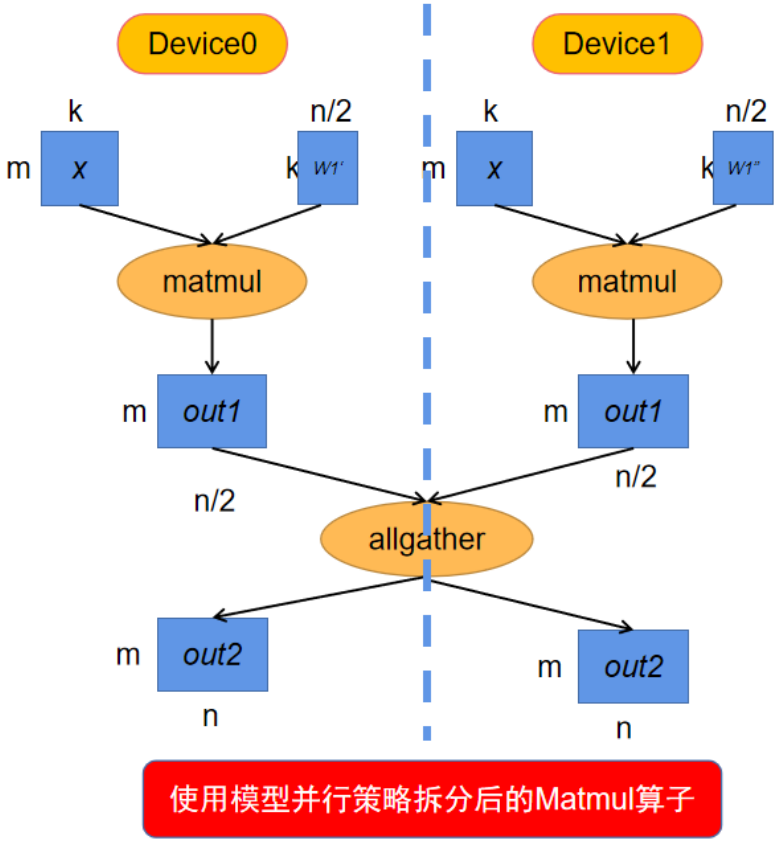
目录

模型并行

模型并行是算子层面的并行，它利用某些算子的特性将算子拆分到多个设备上进行计算。因此并不是网络中所有的算子都可以拆分计算，可以拆分的算子需满足如下特性：

- 可以并行计算的算子
- 算子其中一个输入来自于Parameter





MindSpore算子级并行

- MindSpore对每个算子独立建模，用户可以设置正向网络中每个算子的切分策略（对于未设置的算子，默认按数据并行进行切分）。
- 在构图阶段，框架将遍历正向图，根据算子的切分策略对每个算子及其输入张量进行切分建模，使得该算子的计算逻辑在切分前后保持数学等价。
- 框架内部使用Tensor Layout来表达输入输出张量在集群中的分布状态，Tensor Layout中包含了张量和设备间的映射关系，用户无需感知模型各切片在集群中如何分布，框架将自动调度分配。
- 框架还将遍历相邻算子间张量的Tensor Layout，如果前一个算子输出张量作为下一个算子的输入张量，且前一个算子输出张量的Tensor Layout与下一个算子输入张量的Tensor Layout不同，则需要在两个算子之间进行张量重排布（Tensor Redistribution）。
- 对于训练网络来说，框架处理完正向算子的分布式切分之后，依靠框架的自动微分能力，即能自动完成反向算子的分布式切分。

算子级并行示例

```
import mindspore.nn as nn
from mindspore import ops
import mindspore as ms

ms.set_auto_parallel_context(parallel_mode="semi_auto_parallel", device_num=4)

class DenseMatMulNet(nn.Cell):
    def __init__(self):
        super(DenseMutMulNet, self).__init__()
        self.matmul1 = ops.MatMul.shard(((4, 1), (1, 1)))
        self.matmul2 = ops.MatMul.shard(((1, 1), (1, 4)))
    def construct(self, x, w, v):
        y = self.matmul1(x, w)
        z = self.matmul2(y, v)
        return z
```

在右侧例子中，用户在4卡

计算两个连续的二维矩阵乘：

$$Z = (X * W) * V$$

- 第一个矩阵$Y = X * W$，用户想把X按行切4份（即数据并行）；
- 第二个矩阵乘$Z = Y * V$，用户想把V按列切4份（即模型并行）：

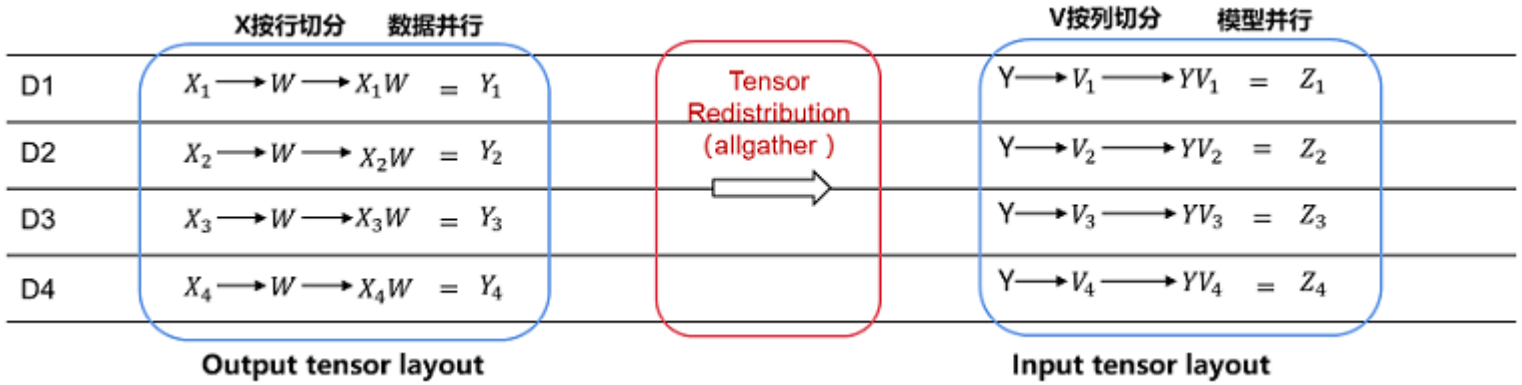
Matmul : (m, k) * (k, n) -> (m, n)

X: (batch_size, input_size) -> (4, 1)

W: (input _size, output_size) -> (1, 1)

算子级并行示例

- 由于第一个算子输出的Tensor Layout是第零维切分到集群，而第二个算子要求第一个输入Tensor在集群上复制。所以在图编译阶段，会自动识别两个算子输出/输入之间Tensor Layout的不同，从而自动推导出Tensor重排布的算法。而这个例子所需要的Tensor重排布是一个AllGather算子（注：MindSpore的AllGather算子会自动把多个输入Tensor在第零维进行合并）

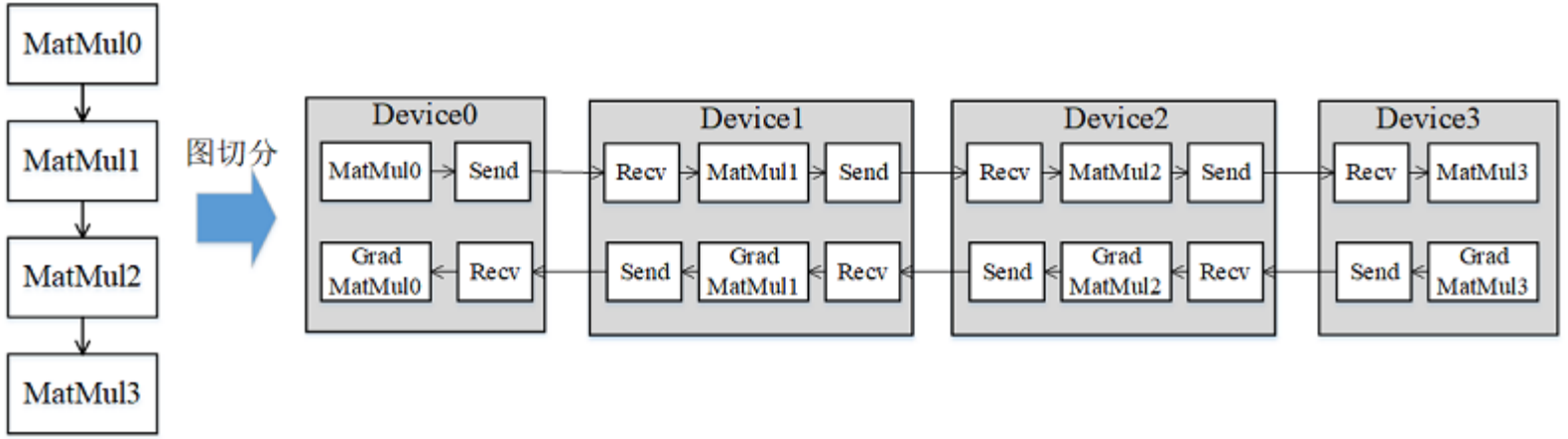


目录

流水线并行

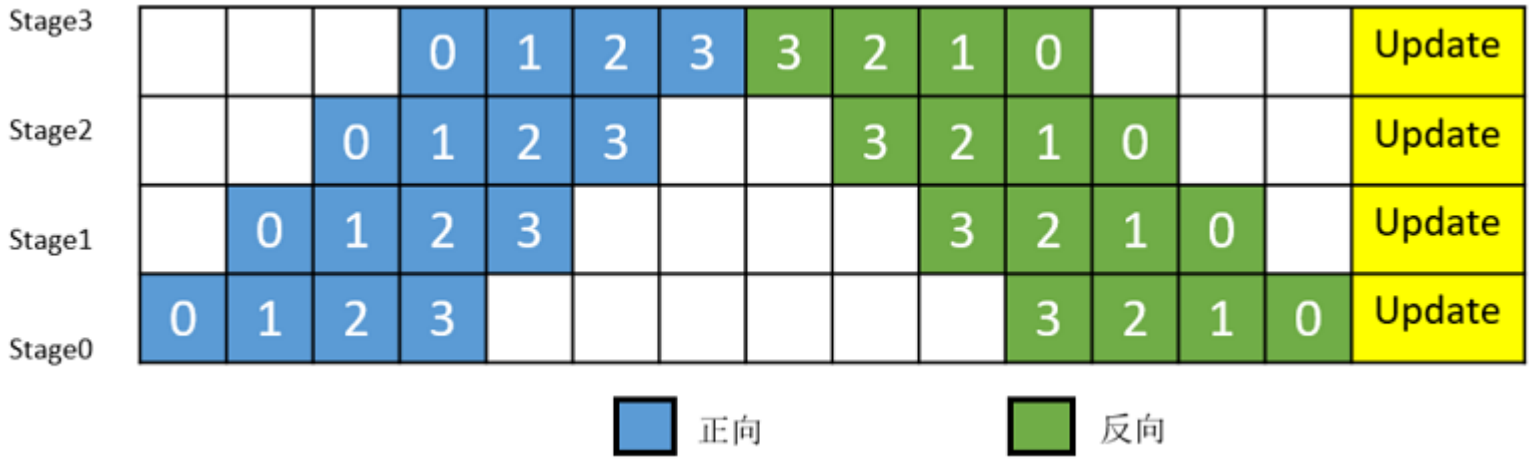
受server间通信带宽低的影响，传统数据并行叠加模型并行的这种混合并行模式的性能表现欠佳，需要引入流水线并行。流水线并行能够将模型在空间上按stage进行切分，每个stage只需执行网络的一部分，大大节省了内存开销，同时缩小了通信域，缩短了通信时间。

流水线（Pipeline）并行是将神经网络中的算子切分成多个阶段（Stage），再把阶段映射到不同的设备上，使得不同设备去计算神经网络的不同部分。



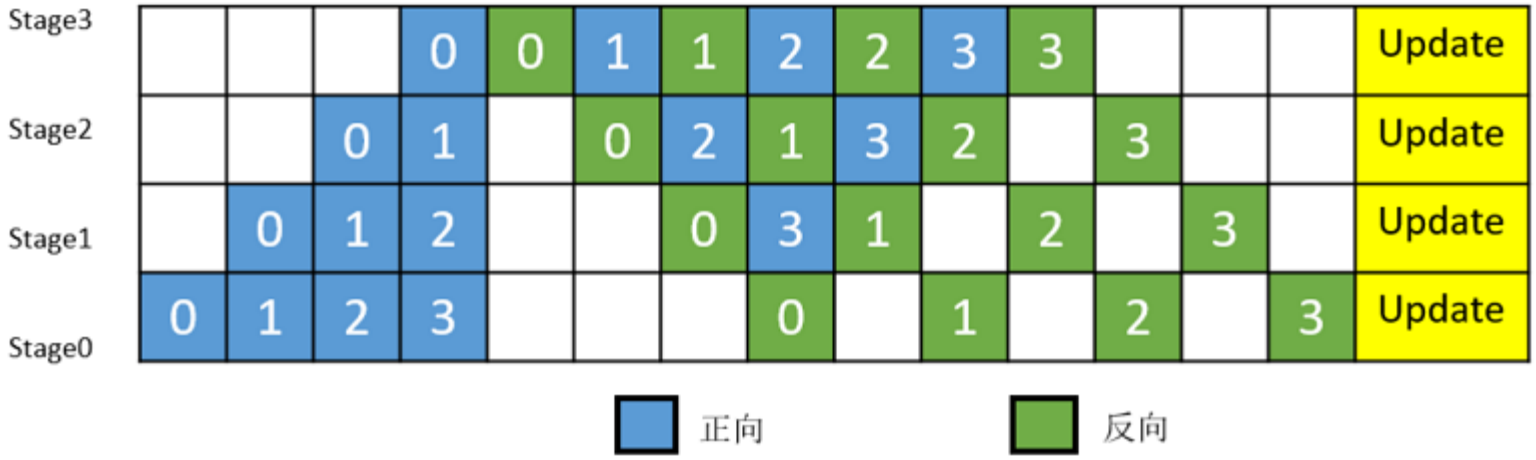
GPipe和Micro batch

简单地将模型切分到多设备上并不会带来性能的提升，因为模型的线性结构到时同一时刻只有一台设备在工作，而其它设备在等待，造成了资源的浪费。为了提升效率，流水线并行进一步将小批次(MiniBatch)切分成更细粒度的微批次(MicroBatch)，在微批次中采用流水线式的执行序，从而达到提升效率的目的。



1F1B

MindSpore的流水线并行实现中对执行序进行了调整，来达到更优的内存管理。如图3所示，在编号为0的MicroBatch的正向执行完后立即执行其反向，这样做使得编号为0的MicroBatch的中间结果的内存得以更早地（相较于图2）释放，进而确保内存使用的峰值比图2的方式更低。



流水线并行示例

流水线并行需要用户去定义并行的策略，通过调用pipeline_stage接口来指定每个layer要在哪个stage上去执行。

pipeline_stage接口的粒度为Cell。所有包含训练参数的Cell都需要配置pipeline_stage，并且pipeline_stage要按照网络执行的先后顺序，从小到大进行配置。

```
class ResNet(nn.Cell):
    """ResNet"""

    def __init__(self, block, num_classes=100, batch_size=32):
        """init"""
        super(ResNet, self).__init__()
        self.batch_size = batch_size
        self.num_classes = num_classes

        self.head = Head()
        self.layer1 = MakeLayer0(block, in_channels=64, out_channels=256, stride=1)
        self.layer2 = MakeLayer1(block, in_channels=256, out_channels=512, stride=2)
        self.layer3 = MakeLayer2(block, in_channels=512, out_channels=1024, stride=2)
        self.layer4 = MakeLayer3(block, in_channels=1024, out_channels=2048, stride=2)

        self.pool = ops.ReduceMean(keep_dims=True)
        self.squeeze = ops.Squeeze(axis=(2, 3))
        self.fc = fc_with_initialize(512 * block.expansion, num_classes)

        # pipeline parallel config
        self.head.pipeline_stage = 0
        self.layer1.pipeline_stage = 0
        self.layer2.pipeline_stage = 0
        self.layer3.pipeline_stage = 1
        self.layer4.pipeline_stage = 1
        self.fc.pipeline_stage = 1
```

目录

目录

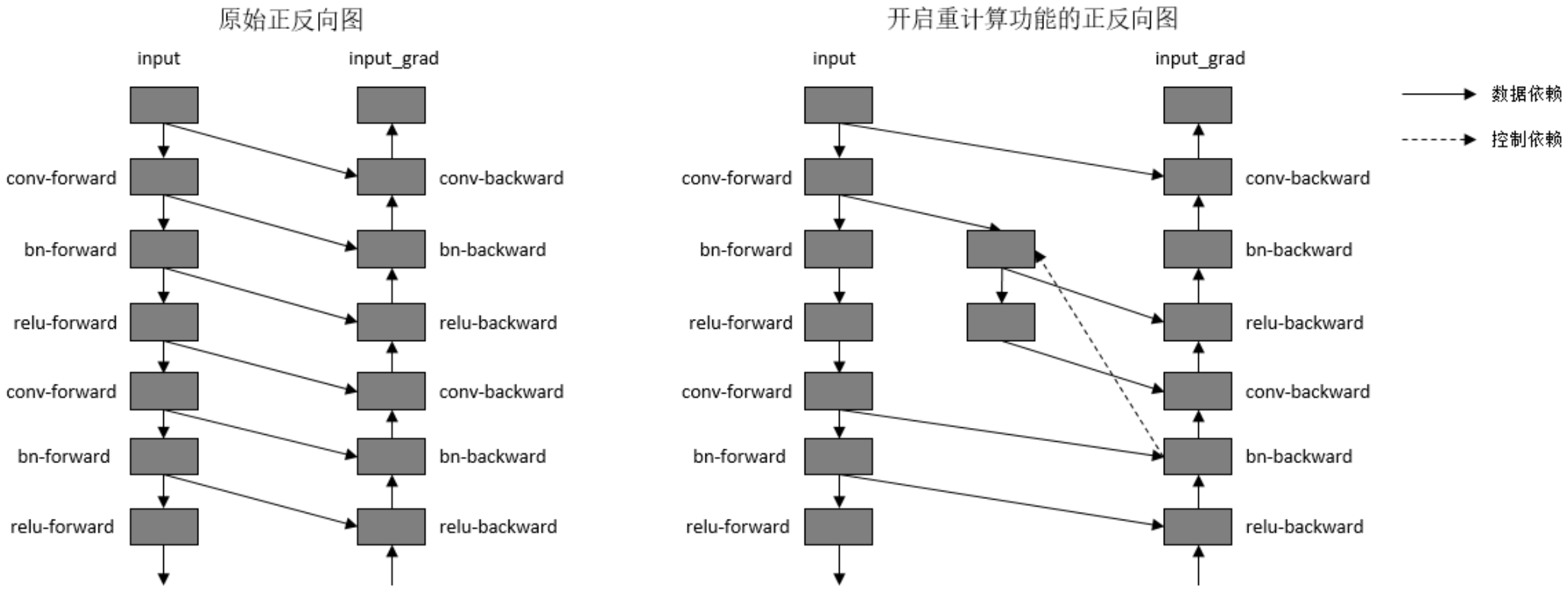
重计算(Recompute/Gradient checkpointing)

在计算某些反向算子时，需要用到一些正向算子的计算结果，导致这些正向算子的计算结果需要驻留在内存中，直到依赖它们的反向算子计算完，这些正向算子的计算结果占用的内存才会被复用。这一现象推高了训练的内存峰值，在大规模网络模型中尤为显著。如:

- Dropout
- Activations

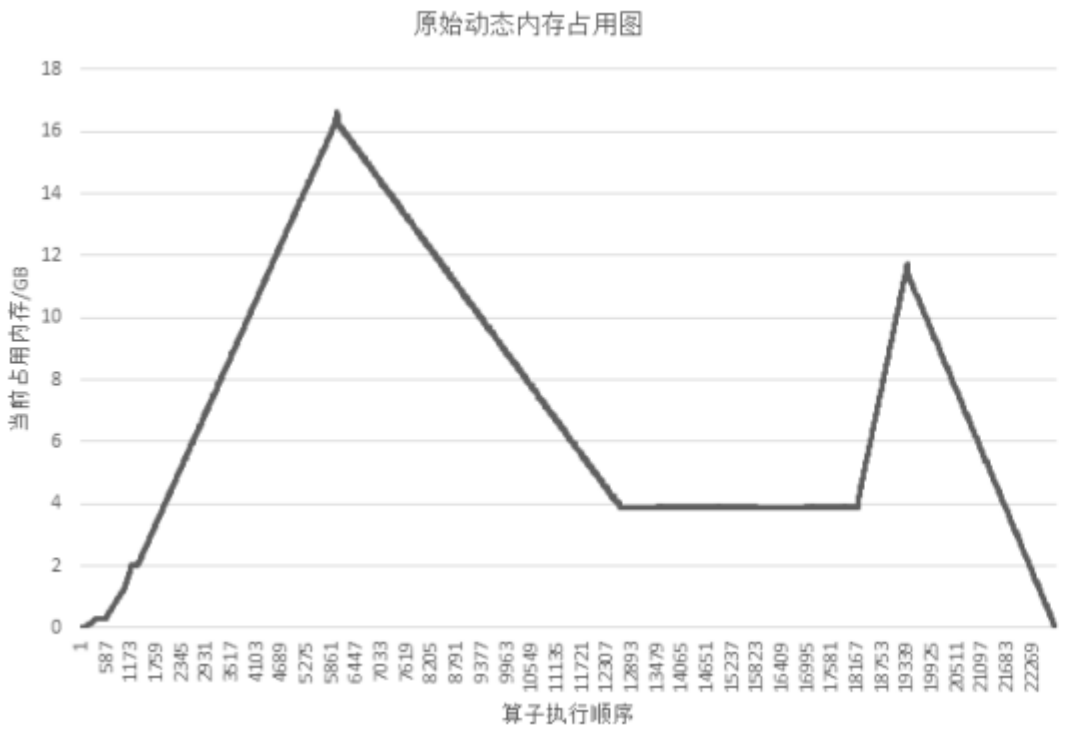
重计算——时间换空间

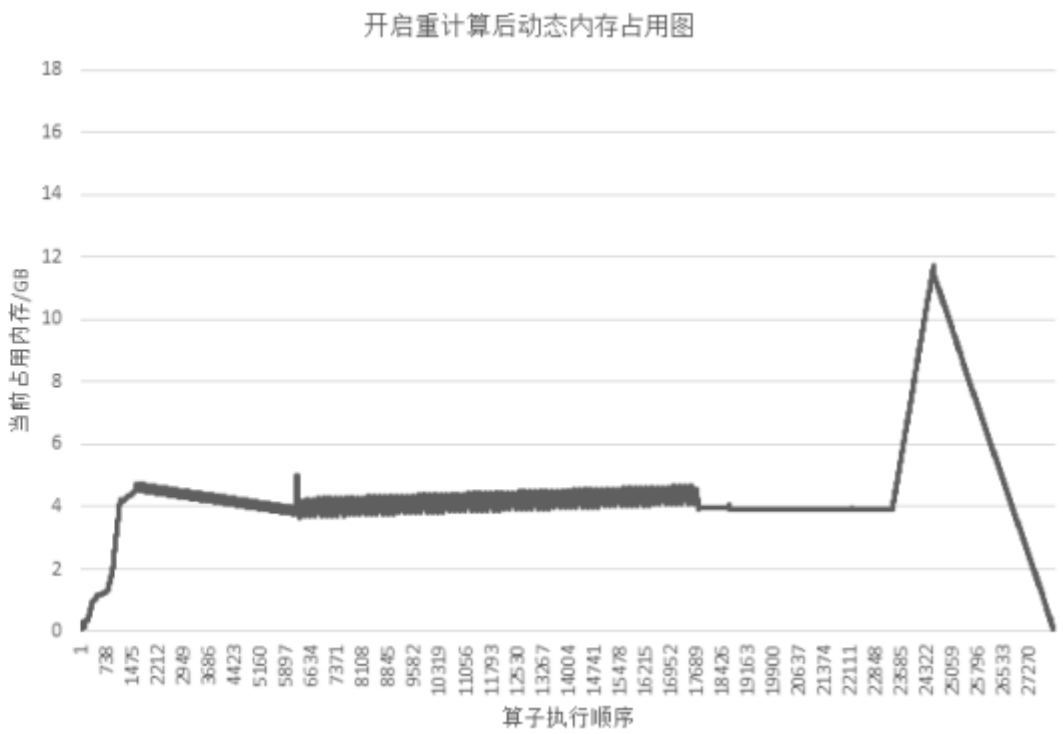
- 为了降低内存峰值，重计算技术可以不保存正向计算结果，让该内存可以被复用，然后在计算反向部分时，重新计算出正向结果。



重计算效果

以GPT3模型为例，设置策略为对每层layer对应的Cell设置为重计算，然后每层layer的输出算子设置为非重计算。72层GPT3网络开启重计算的效果如下图所示：





重计算使用方式

- 为了方便用户使用，MindSpore提供了针对单个算子和Cell设置的重计算接口。当用户调用Cell的重计算接口时，这个Cell里面的所有正向算子都会被设置为重计算。

```
class ResNet(nn.Cell):
    ...
    def __init__(self,
                 block,
                 layer_nums,
                 in_channels,
                 out_channels,
                 strides,
                 num_classes,
                 use_se=False,
                 res_base=False):
        super(ResNet, self).__init__()
        ...
        self.relu = ops.ReLU()
        self.relu.recompute()
        ...
```

目录

优化器并行

- 在进行数据并行训练时，模型的参数更新部分在各卡间存在冗余计算，优化器并行通过将优化器的计算量分散到数据并行维度的卡上，在大规模网络上（比如Bert、GPT）可以有效减少内存消耗并提升网络性能。
- 传统的数据并行模式将模型参数在每台设备上都有保有副本，把训练数据切分，在每次迭代后利用通信算子同步梯度信息，最后通过优化器计算对参数进行更新。数据并行虽然能够有效提升训练吞吐量，但并没有最大限度地利用机器资源。其中优化器会引入冗余内存和计算，消除这些冗余是需关注的优化点。

ZeRO (1-3)

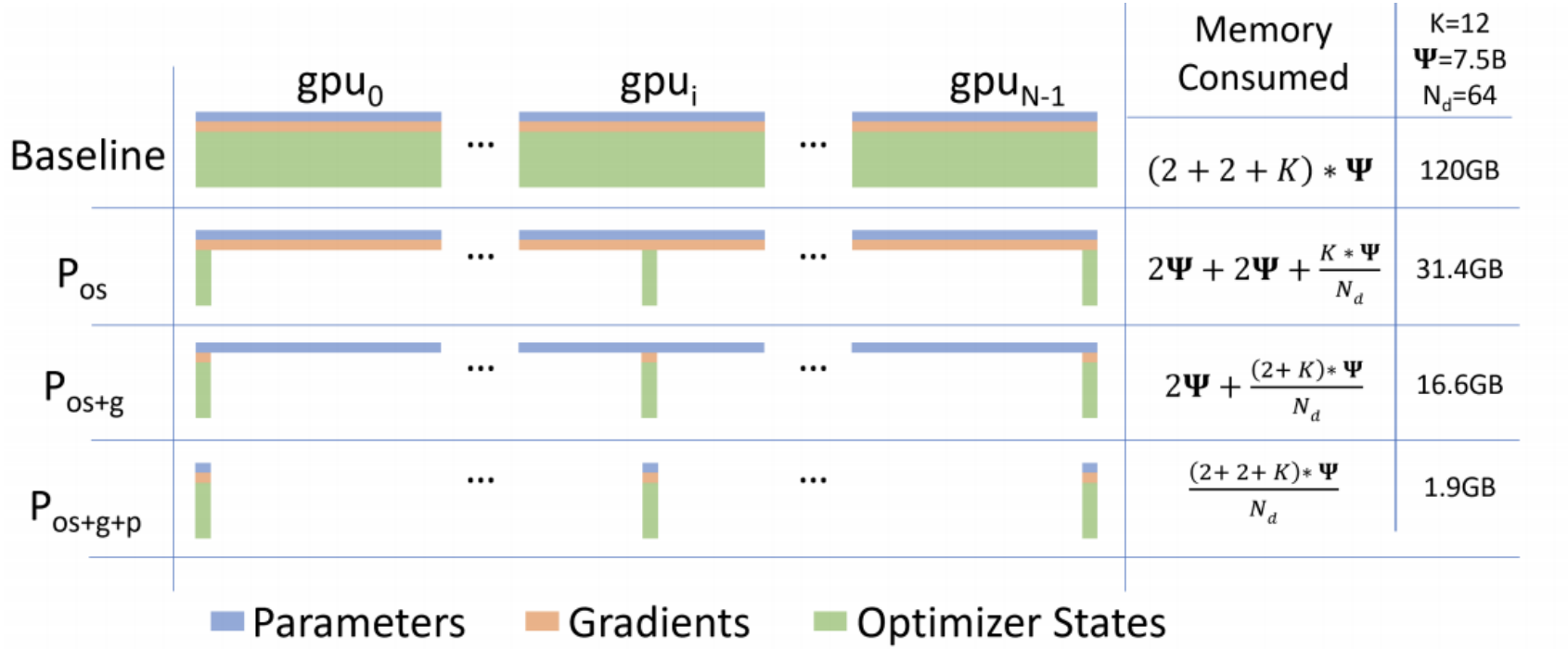


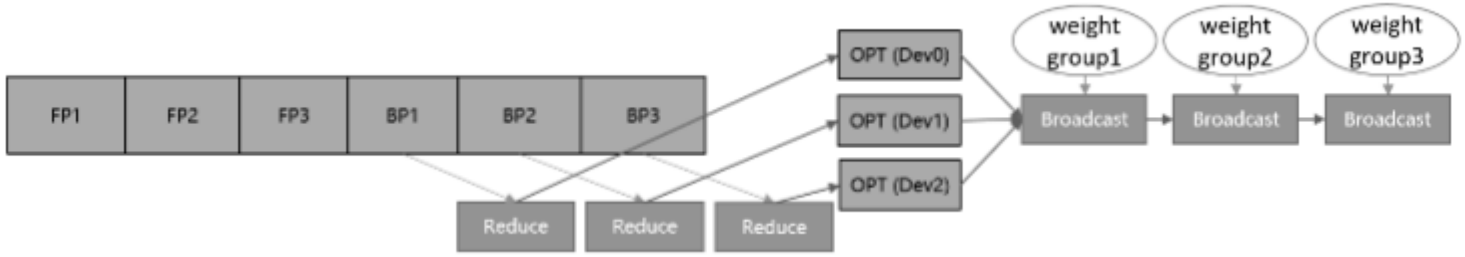
Figure 1: Comparing the per-device memory consumption of model states, with three stages of *ZeRO*-DP optimizations. Ψ denotes model size (number of parameters), K denotes the memory multiplier of optimizer states, and N_d denotes DP degree. In the example, we assume a model size of $\Psi = 7.5B$ and DP of $N_d = 64$ with $K = 12$ based on mixed-precision training with Adam optimizer.

ZeRO（1-3）

- 优化器状态切分p_{os}：切分优化器状态到各个计算卡中，在享有与普通数据并行相同通信量的情况下，可降低4倍的内存占用
- 添加梯度切分$p_{\mathrm{os}+g}$：在p_{os}的基础上，进一步将模型梯度切分到各个计算卡中，在享有与普通数据并行相同通信量的情况下，拥有8倍的内存降低能力
- 添加参数切分$p_{\mathrm{os}+g+p}$：在$p_{\mathrm{os}+g}$的基础上，将模型参数也切分到各个计算卡中，内存降低能力与并行数量 成线性比例，通信量大约有50%的增长

参数分组(Weights Grouping)

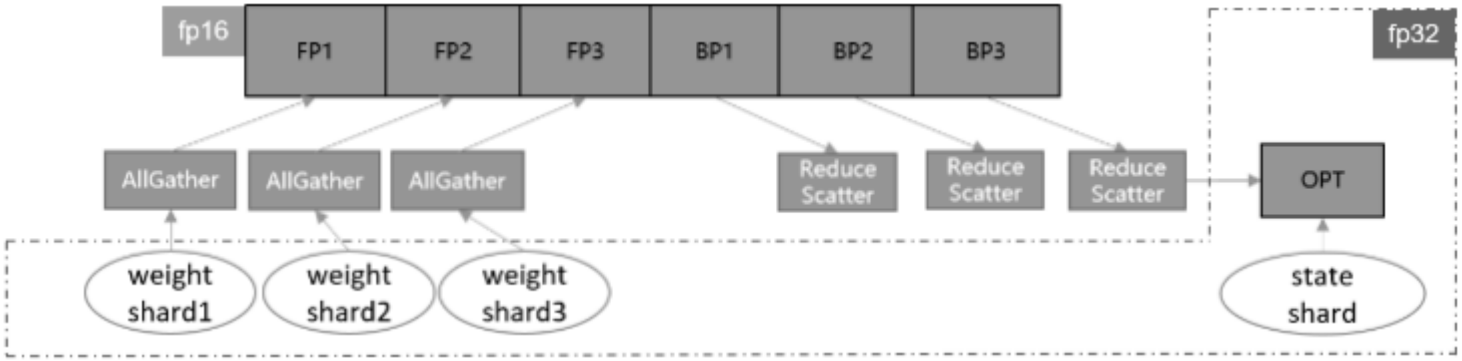
将参数和梯度分组放到不同卡上更新，再通过通信广播操作在设备间共享更新后的权值。该方案的内存和性能收益取决于参数比例最大的group。当参数均匀划分时，理论上的正收益是N-1/N的优化器运行时间和动态内存，以及N-1/N的优化器状态参数内存大小，其中N表示设备数。而引入的负收益是共享网络权重时带来的通信时间。



参数切分(Weights Sharding)

对参数做层内划分，对每一个参数及梯度根据设备号取其对应切片，各自更新后再调用通信聚合操作在设备间共享参数。这种方案的优点是天然支持负载均衡，即每张卡上参数量和计算量一致，缺点是对参数形状有整除设备数要求。该方案的理论收益与参数分组一致，为了扩大优势，框架做了如下几点改进。

- 对网络中的权重做切分，可以进一步减少静态内存。但这也需要将迭代末尾的共享权重操作移动到下一轮迭代的正向启动前执行，保证进入正反向运算的依旧是原始张量形状。
- 优化器并行运算带来的主要负收益是共享权重的通信时间，如果我们能够将其减少或隐藏，就可以带来性能上的提升。通信跨迭代执行的一个好处就是，可以通过对通信算子适当分组融合，将通信操作与正向网络交叠执行，从而尽可能隐藏通信耗时。通信耗时还与通信量有关，对于涉及混合精度的网络，如果能够使用fp16通信，通信量相比fp32将减少一半



优化器并行使用方式

开启优化器并行

在 `mindspore.set_auto_parallel_context` 中提供了 `enable_parallel_optimizer` 选项，将其配置为True后，即可使能优化器并行，默认对所有**占用内存不小于64KB**的参数进行优化器切分。

```
import mindspore as ms
ms.set_auto_parallel_context(enable_parallel_optimizer=True)
```

配置参数优化器并行

目录

MindSpore并行模式

- 数据并行：用户的网络参数规模在单卡上可以计算的情况下使用。这种模式会在每卡上复制相同的网络参数，训练时输入不同的训练数据，适合大部分用户使用。
- 半自动并行：用户的神经网络在单卡上无法计算，并且对切分的性能存在较大的需求。用户可以设置这种运行模式，手动指定每个算子的切分策略，达到较佳的训练性能。
- 自动并行：用户的神经网络在单卡上无法计算，但是不知道如何配置算子策略。用户启动这种模式，MindSpore会自动针对每个算子进行配置策略，适合想要并行训练但是不知道如何配置策略的用户。
- 混合并行：完全由用户自己设计并行训练的逻辑和实现，用户可以自己在网络中定义AllGather等通信算子。适合熟悉并行训练的用户。