DP特性详解 &MindNLP数据并行实操

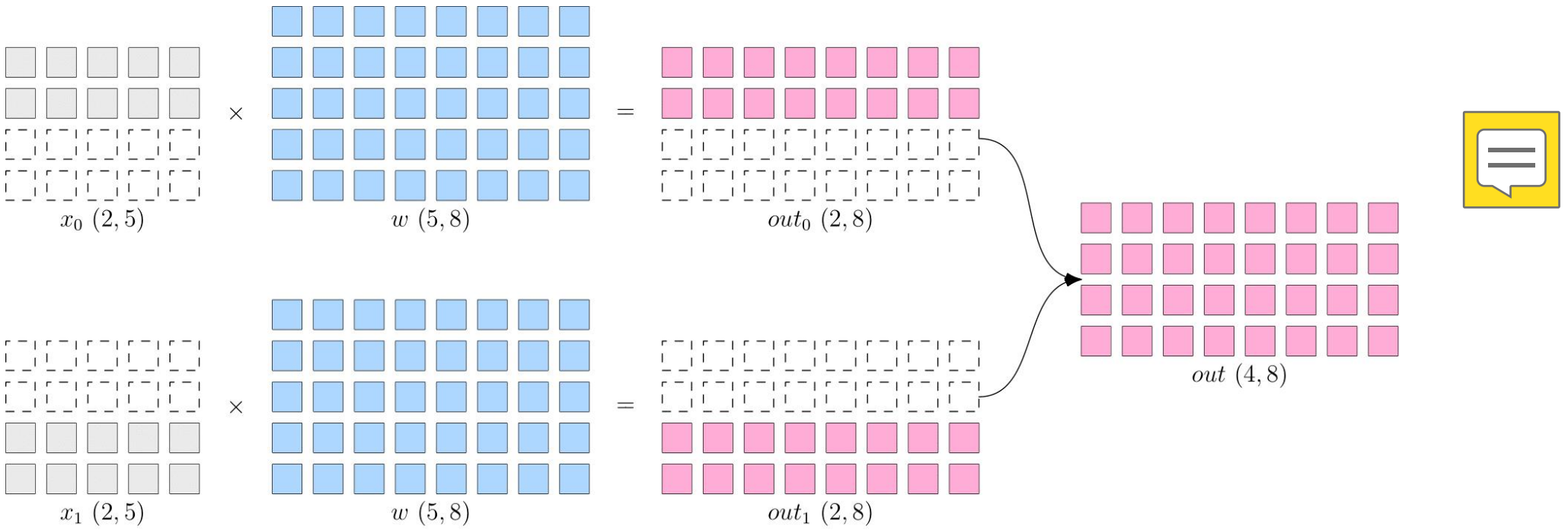
吕昱峰

目录

- 数据并行
  - AllReduce
  - Ring Allreduce
  - MindSpore数据并行
  - MindNLP数据并行实现解析
- MindNLP数据并行实操
- 作业
- 答疑

```
### MindSpore 支持的分布式计算架构

### 1. **数据并行 (Data Parallelism)**

#### **案例**：
- **ResNet50 分布式训练**：在多个 GPU 上进行 ResNet50 的分布式训练，使用 AllReduce 进行梯度同步。
- **BERT 模型训练**：在多个节点上进行 BERT 模型的分布式训练，使用 AllReduce 或 PS 架构进行梯度同步。

### 2. **模型并行 (Model Parallelism)**

#### **案例**：
- **GPT-3 模型训练**：在多个 GPU 上进行 GPT-3 模型的分布式训练，使用模型并行将不同层分配到不同的 GPU 上。
- **T5 模型训练**：在多个节点上进行 T5 模型的分布式训练，使用模型并行将编码器和解码器分别分配到不同的节点上。

### 3. **Pipeline 并行 (Pipeline Parallelism)**

#### **案例**：
- **Transformer 模型训练**：在多个 GPU 上进行 Transformer 模型的分布式训练，使用 Pipeline 并行将不同层分配到不同的 GPU 上。
- **语音识别模型训练**：在多个节点上进行语音识别模型的分布式训练，使用 Pipeline 并行将编码器和解码器分别分配到不同的节点上。

### 4. **混合并行 (Hybrid Parallelism)**

#### **案例**：
- **MoE (Mixture of Experts) 模型训练**：在多个 GPU 上进行 MoE 模型的分布式训练，使用混合并行将不同专家分配到不同的 GPU 上，并使用数据并行处理不同的数据子集。
- **大规模 NLP 模型训练**：在多个节点上进行大规模 NLP 模型的分布式训练，使用混合并行将编码器和解码器分别分配到不同的节点上，并使用数据并行处理不同的数据子集。

### 5. **弹性训练 (Elastic Training)**

#### **案例**：
- **云平台上的分布式训练**：在云平台上进行分布式训练，使用弹性训练功能动态调整训练集群的规模，并处理节点故障。
- **大规模 NLP 模型训练**：在大规模集群上进行 NLP 模型的分布式训练，使用弹性训练功能确保在节点故障时继续训练。
```

数据并行



数据并行（Data Parallelism，DP）的核心思想是将大规模的数据集分割成若干个较小的数据子集，并将这些子集分配到不同的 NPU 计算节点上，每个节点运行相同的模型副本，但处理不同的数据子集。

数据并行过程

输入数据切分

数据并行运行过程中，会通过两种方式切分输入数据。第一种方式，根据并行进程数进行划分，无需进
数据通信，每个进程只读取自身划分到的数据；第二种方式，数据读取由进程负责，数据读取后根据并行
进程数切分，再发送到对应进程中。

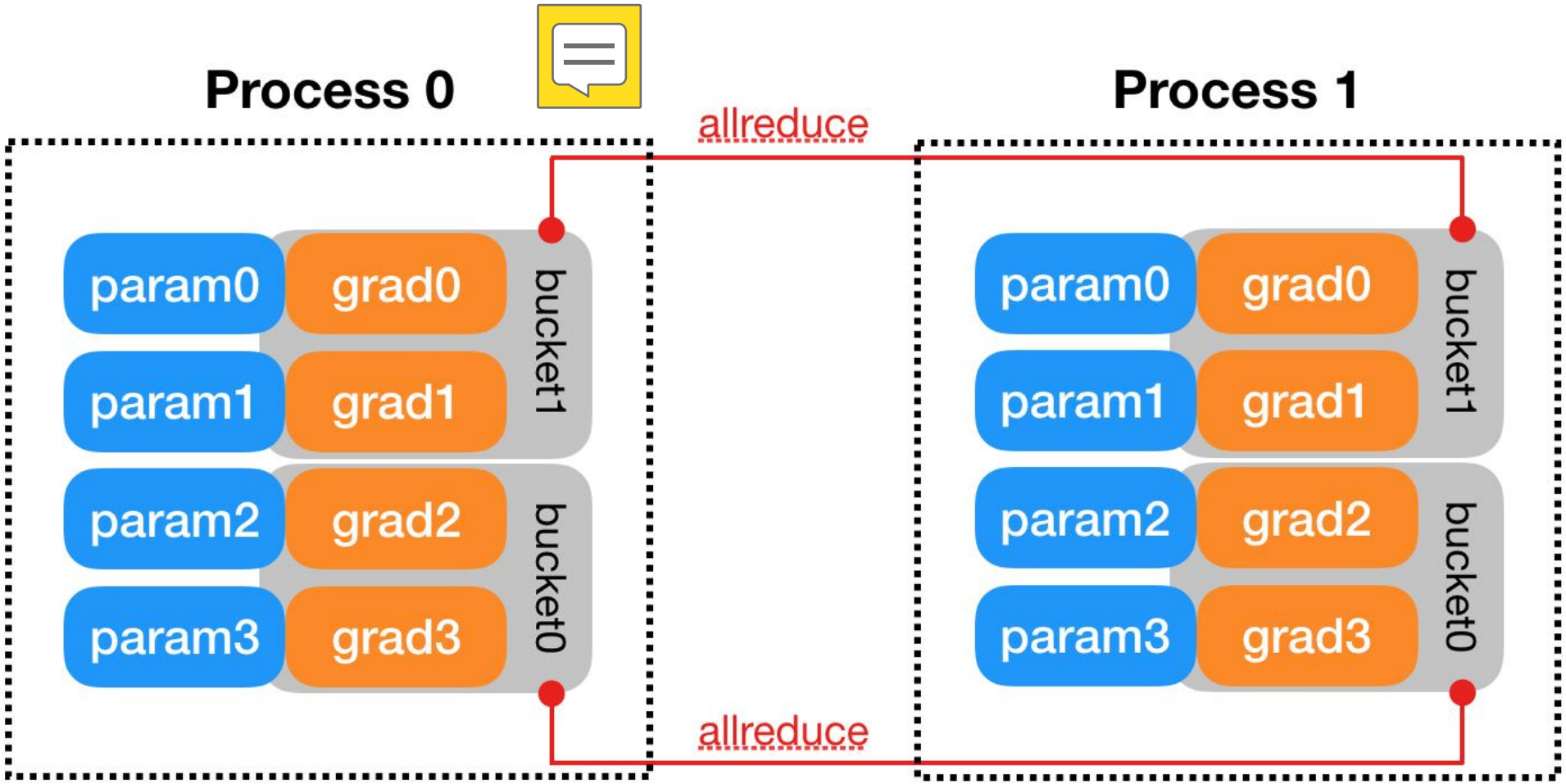
模型参数同步

数据并行过程中，需要在处理数据时保持模型参数同步。模型参数同步可以在初始时使用相同随机种子完
成，以相同的顺序进行初始化来实现。也可以通过某一进程初始化全部模型参数后，向其他进程广播模型
参数，实现同步。

参数更新

数据并行的参数更新是在输入数据切分和模型参数同步的步骤完成后进行的。更新前，每个进程的参数相
同；更新时，基于所有进程上的梯度同步得到的全局梯度也相同，所以实现在更新后每个进程得到的参数
也是相同的。

数据并行示意图(dp=2)

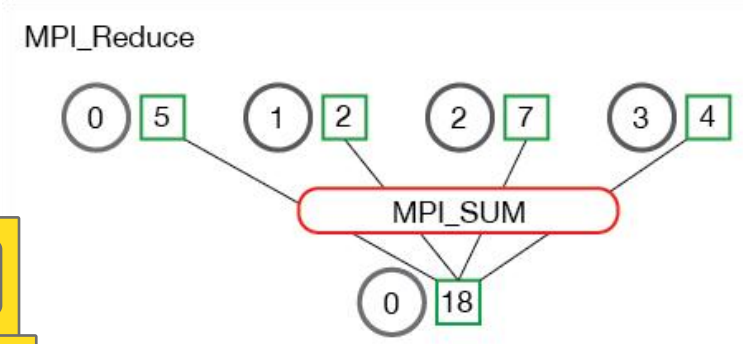

Process 0
Process 1
allreduce
param0
grad0
param1
grad1
param2
grad2
param3
grad3
bucket1
bucket0
allreduce

Reduce

归约简介

归约 是函数式编程中的经典概念。 数据归约包括通过函数将一组数字归约为较小的一组数字。 例如，假设我们有一个数字列表 `[1,2,3,4,5]` 。 用 sum 函数归约此数字列表将产生 `sum([1、2、3、4、5]) = 15` 。 类似地，乘法归约将产生 `multiply([1、2、3、4、5]) = 120` 。

就像您想象的那样，在一组分布式数字上应用归约函数可能非常麻烦。 随之而来的是，难以有效地实现非可交换的归约，即必须以设定顺序发生的缩减。 幸运的是，MPI 有一个方便的函数， `MPI_Reduce` ，它将处理程序员在并行程序中需要执行的几乎所有常见的归约操作。



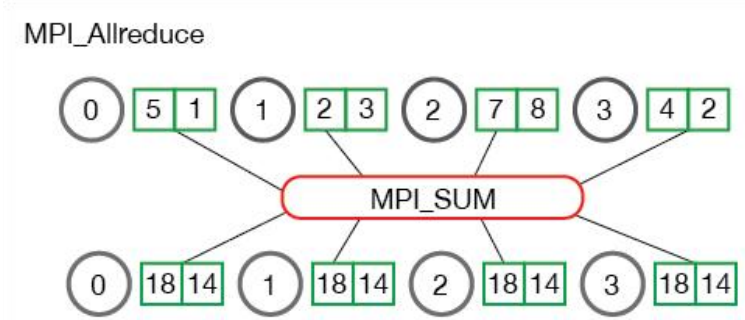
AllReduce

`MPI_Allreduce` 等效于先执行 `MPI_Reduce`，然后执行 `MPI_Bcast`。

MPI_Allreduce

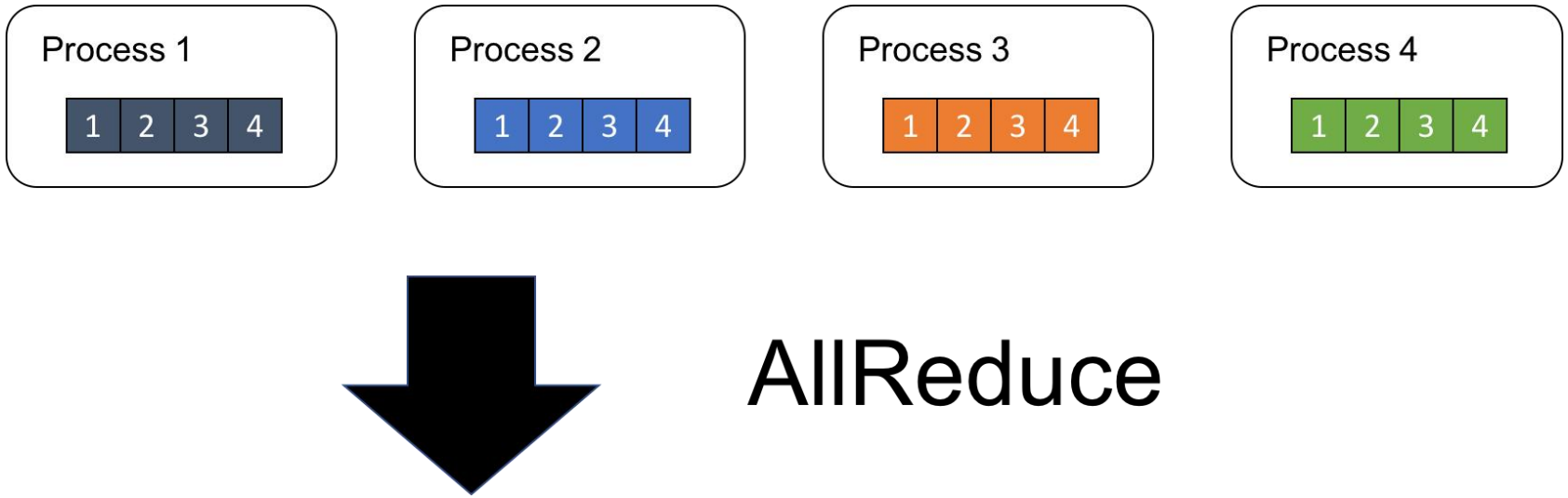
许多并行程序中，需要在所有进程而不是仅仅在根进程中访问归约的结果。 以与 `MPI_Gather` 相似的补充方式， `MPI_Allreduce` 将归约值并将结果分配给所有进程。 函数原型如下：

```
llreduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm communicator)
```



您可能已经注意到， `MPI_Allreduce` 与 `MPI_Reduce` 相同，不同之处在于它不需要根进程 ID（因为结果分配给所有进程）。 下图介绍了 `MPI_Allreduce` 的通信模式：

AllReduce


Process 1
1 2 3 4
Process 2
1 2 3 4
Process 3
1 2 3 4
Process 4
1 2 3 4
AllReduce
Process 1
1 2 3 4
Process 2
1 2 3 4
Process 3
1 2 3 4
Process 4
1 2 3 4

MindSpore AllReduce

```
>>> import numpy as np
>>> from mindspore.communication import init
>>> from mindspore.communication.comm_func import all_r
>>> from mindspore import Tensor
>>>
>>> init()
>>> input_t = Tensor(np.ones([2, 8]).astype(np.float32))
>>> output = all_reduce(input_tensor)
>>> print(output)
[[2. 2. 2. 2. 2. 2. 2. 2.]
[2. 2. 2. 2. 2. 2. 2. 2.]]
```

Before AllReduce

NPU 0:

```
[[1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]]
```

NPU 1:

```
[[1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]]
```

After AllReduce

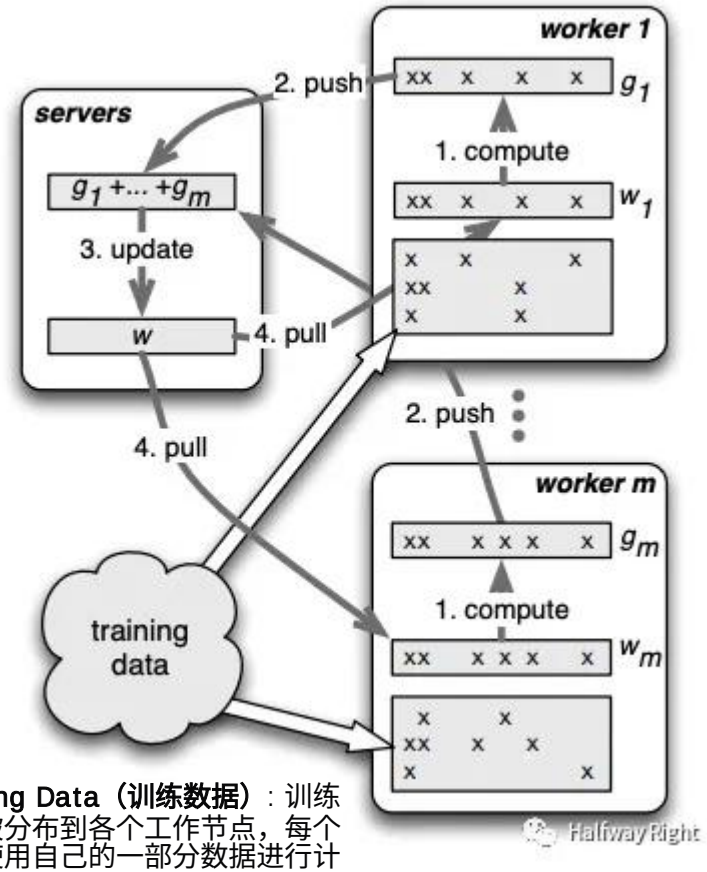
NPU 0:

```
[[2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]]
```

NPU 1:

```
[[2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]]
```

Parameter-Server



Training Data（训练数据）：训练数据被分布到各个工作节点，每个节点使用自己的一部分数据进行计算。

主要思想：所有node被分为server node和worker node

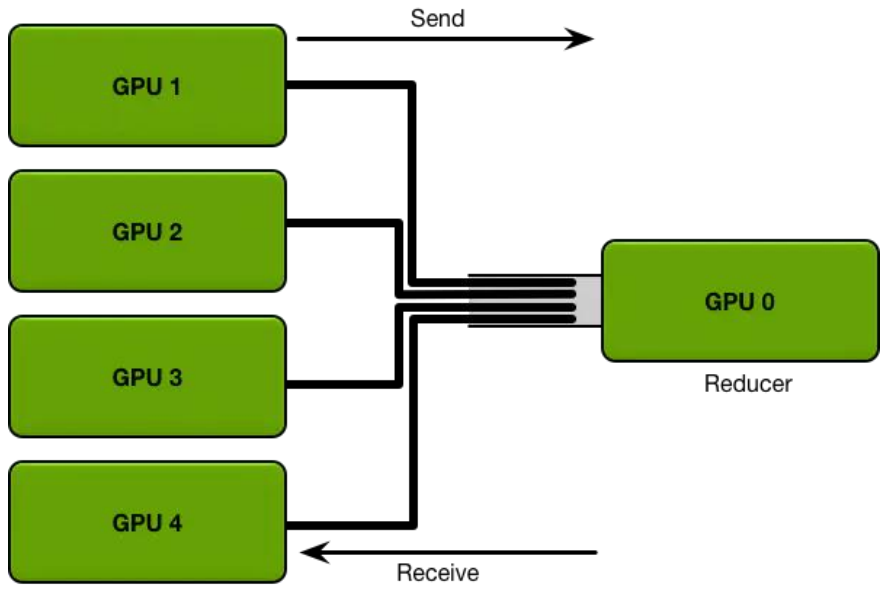
Server node：负责参数的存储和全局的聚合操作。

Worker node：负责计算。

计算过程：

1. 所有数据被划分到每个worker node上；
2. 每个worker node从server node中pull 模型的参数；
3. worker node用本地的数据进行计算得到本地的梯度；
4. 将梯度push到server node上；
5. server node进行allreduce得到全局的梯度；
6. 进行参数的更新；
7. 重复1-6。

Parameter-Server的问题

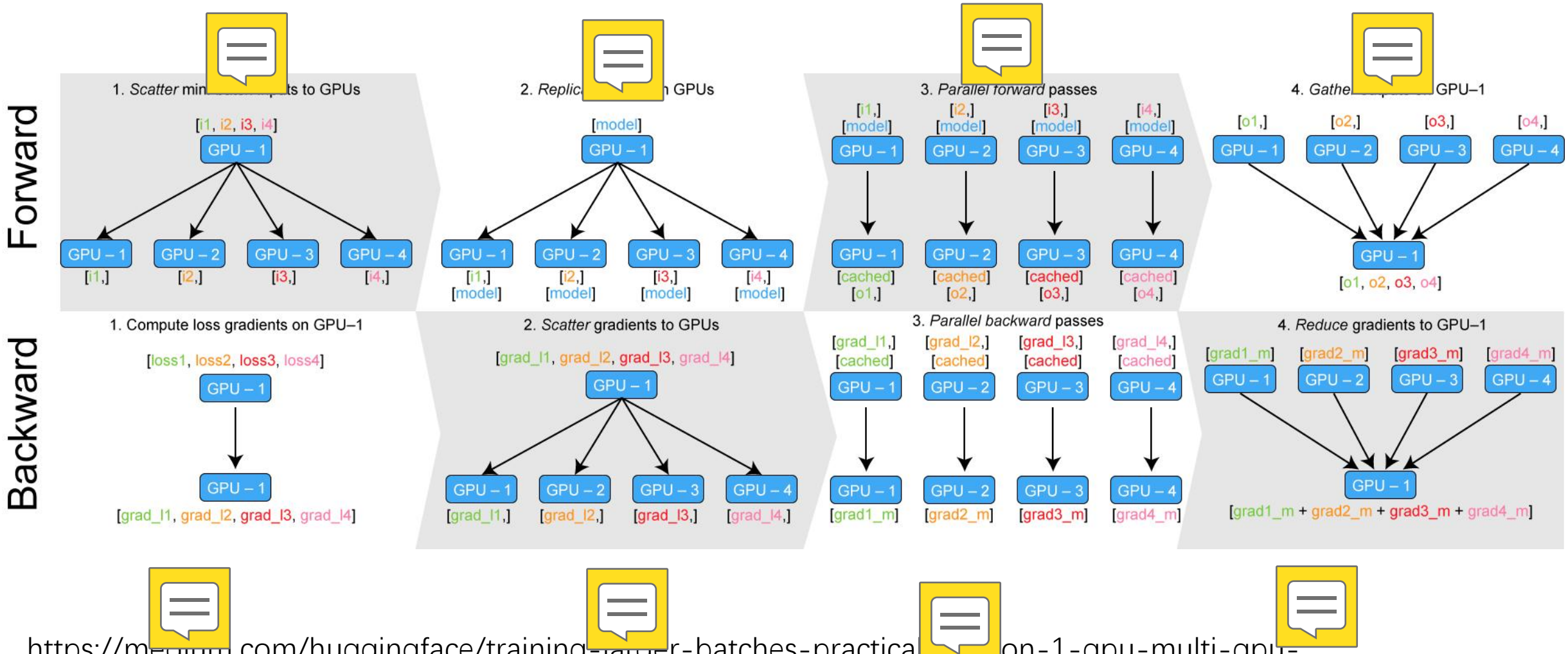


考虑一个简单的同步通信策略：

1. 假设有N张卡（N=5），其中GPU0作为Server，剩余4张卡作为Worker。
2. 将大小为K的训练数据分为N-1块，分给每张GPU。
3. 每个GPU计算得到local gradients。
4. N-1块GPU将计算所得的local gradients 发送给GPU 0。
5. GPU 0对所有的local gradients进行reduce得到全局的梯度，进行参数更新。
6. 将该新的模型参数返回给每块GPU。

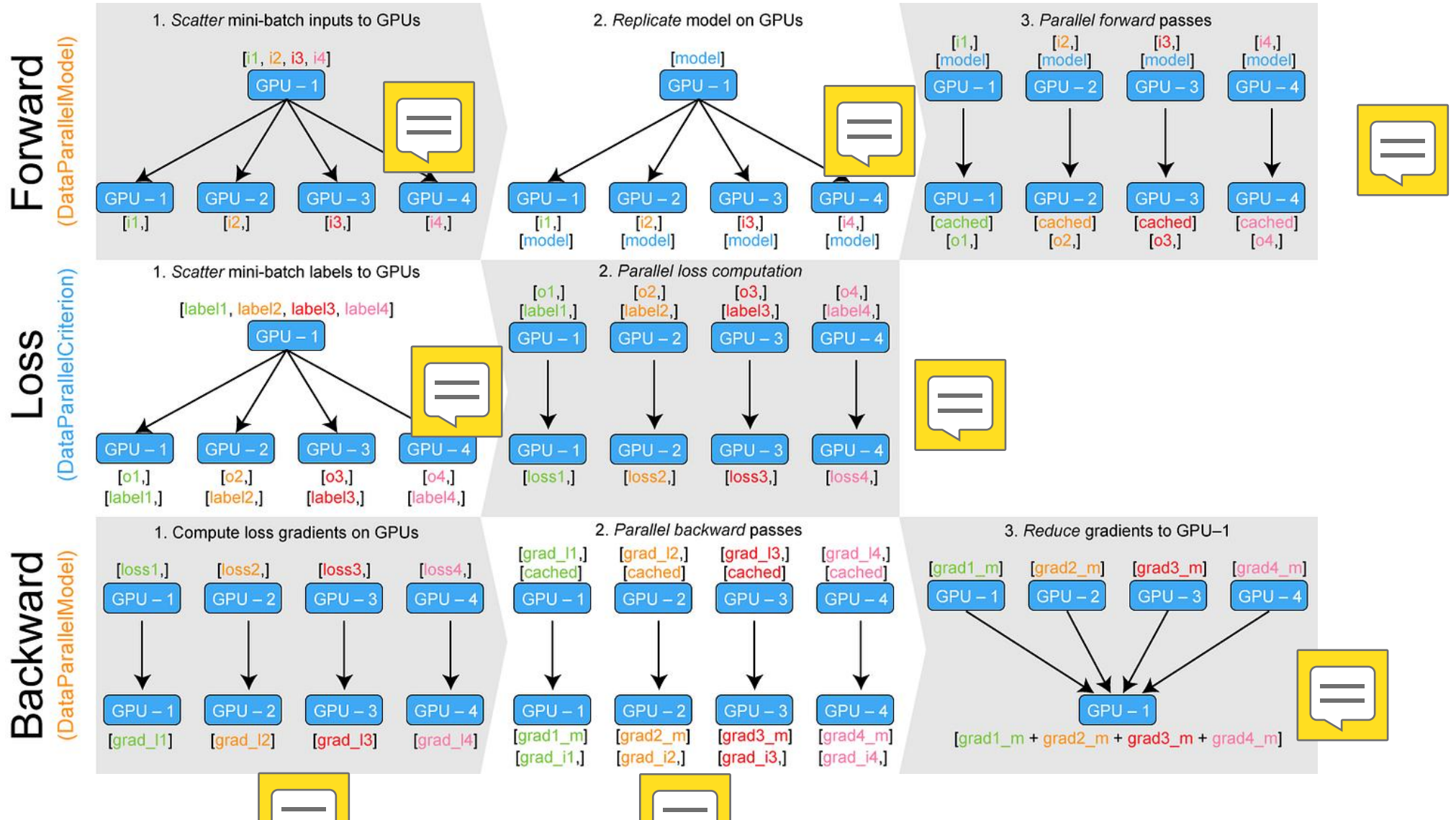
假设单个Worker到Server的通信开销为C。将local gradients梯度全部发送到GPU 0上的通信成本是 C * （N-1），由于受GPU 0的network bandwidth的影响，通信成本随着设备数的增加，而线性增长。

Pytorch DataParallel



https://medium.com/huggingface/training-larger-batches-practical-on-1-gpu-multi-gpu-distributed-setups-ec88c3e51255

Pytorch DataParallel(with criterion)



Pytorch DataParallel

问题：

1. 仍旧是Parameter-Server模式，性能差
2. 需要额外的GPU进行梯度聚合/GPU:0需要额外的显存

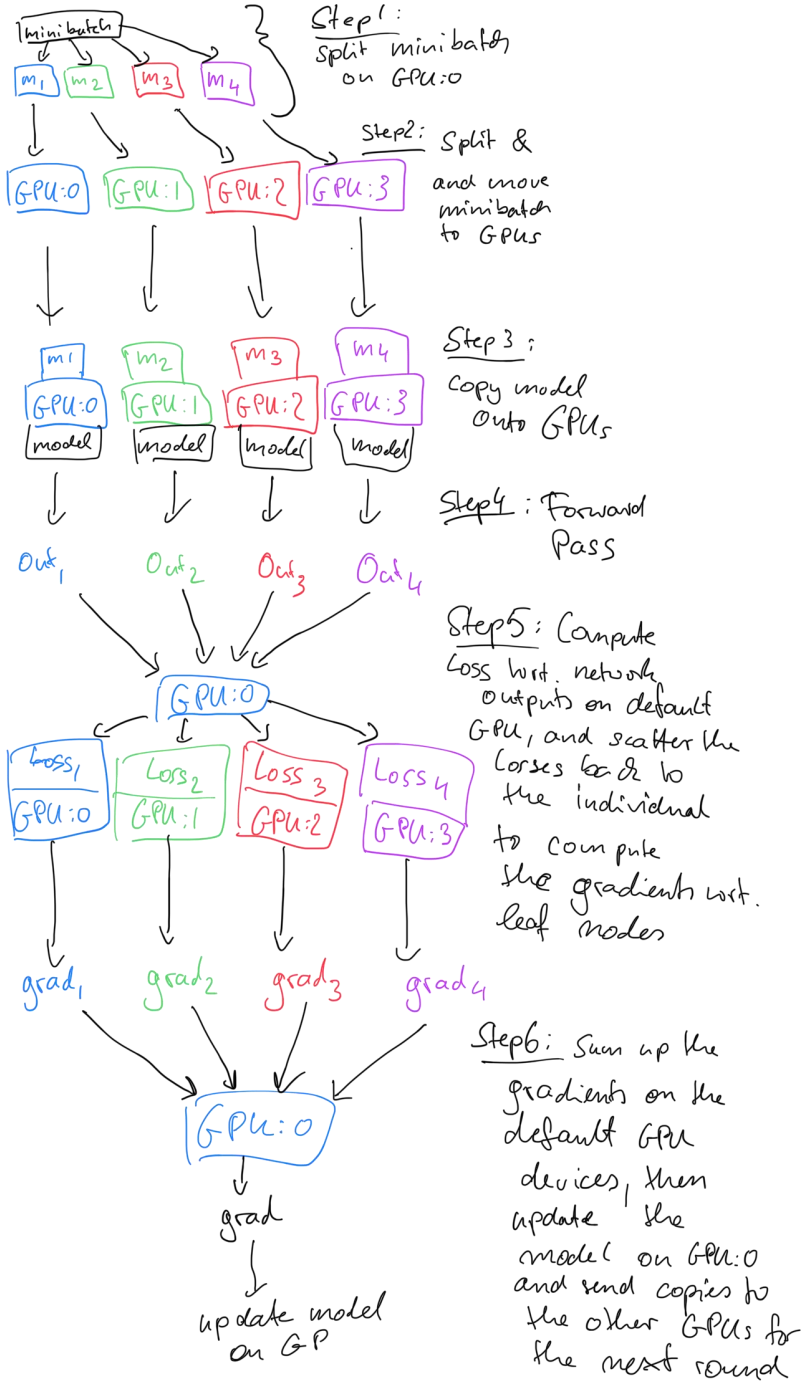
计算 梯度聚合

GPU0: 12G+12G

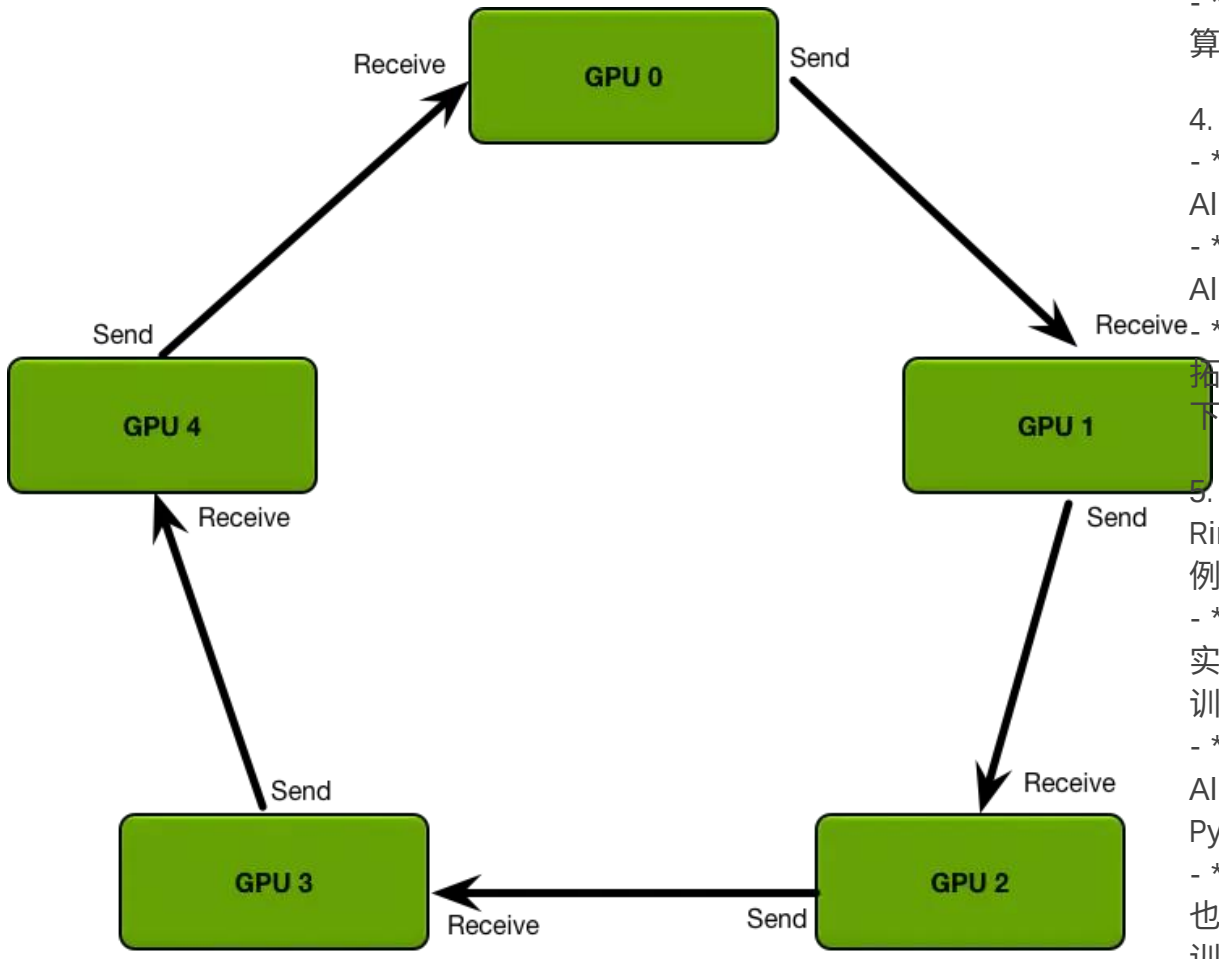
GPU1: 12G

GPU2: 12G

GPU3: 12G



Ring Allreduce

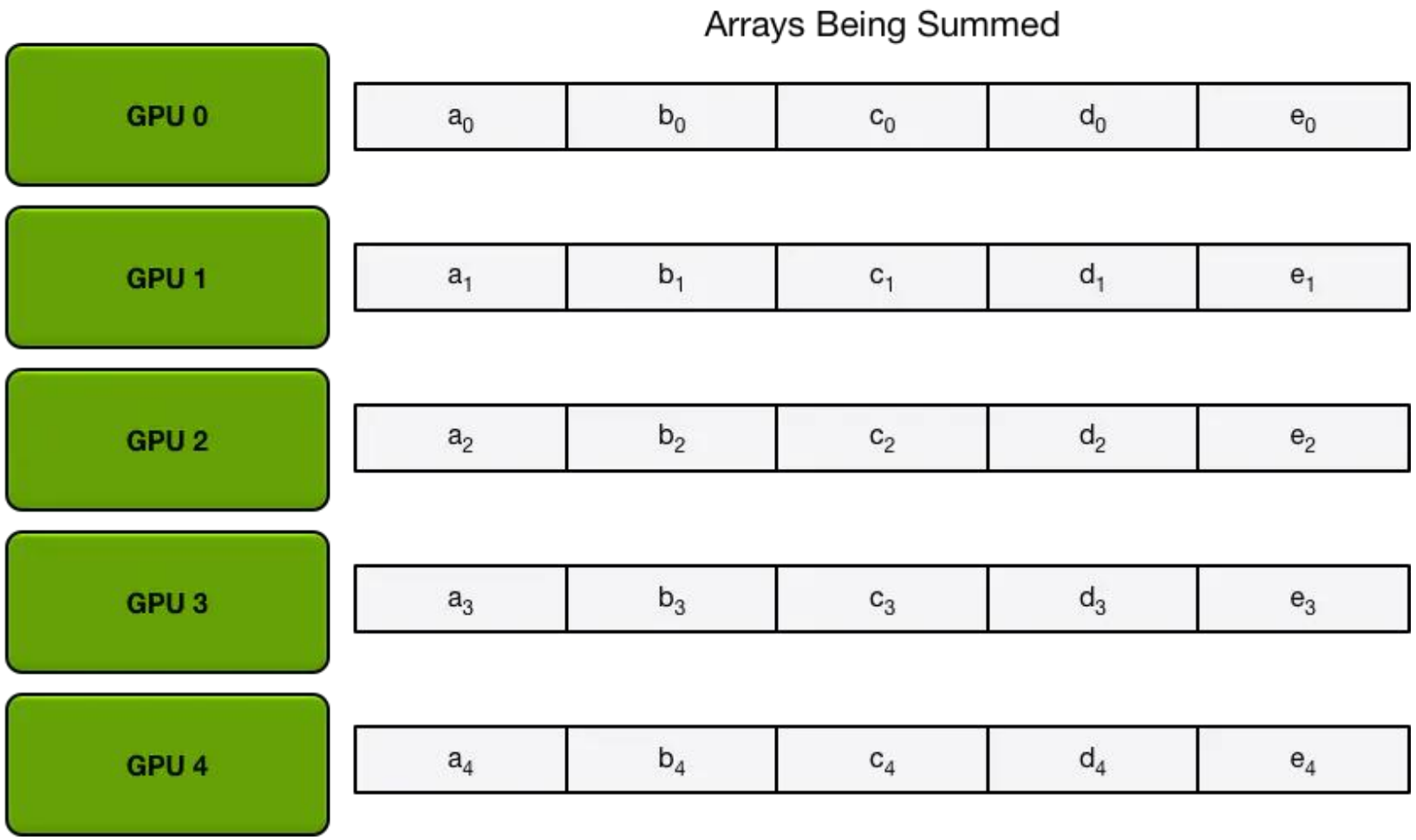


3. Ring Allreduce 的优势
- **通信负载均衡**：每个节点只与两个邻居节点通信，通信负载均匀分布，避免了单点瓶颈。
- **带宽利用率高**：通过分块和逐步传递的方式，Ring Allreduce 充分利用了网络带宽。
- **扩展性强**：随着节点数量的增加，Ring Allreduce 的通信开销增长较慢，适合大规模分布式训练。
- **无中心节点**：Ring Allreduce 是一种去中心化的算法，不需要参数服务器，避免了单点故障问题。

4. Ring Allreduce 的局限性
- **延迟较高**：由于数据需要逐步传递，Ring Allreduce 的延迟随着节点数量的增加而增加。
- **实现复杂**：相比传统的参数服务器架构，Ring Allreduce 的实现和调试更加复杂。
- **对网络拓扑敏感**：Ring Allreduce 的性能受网络拓扑结构的影响，网络延迟和带宽不均衡可能导致性能下降。

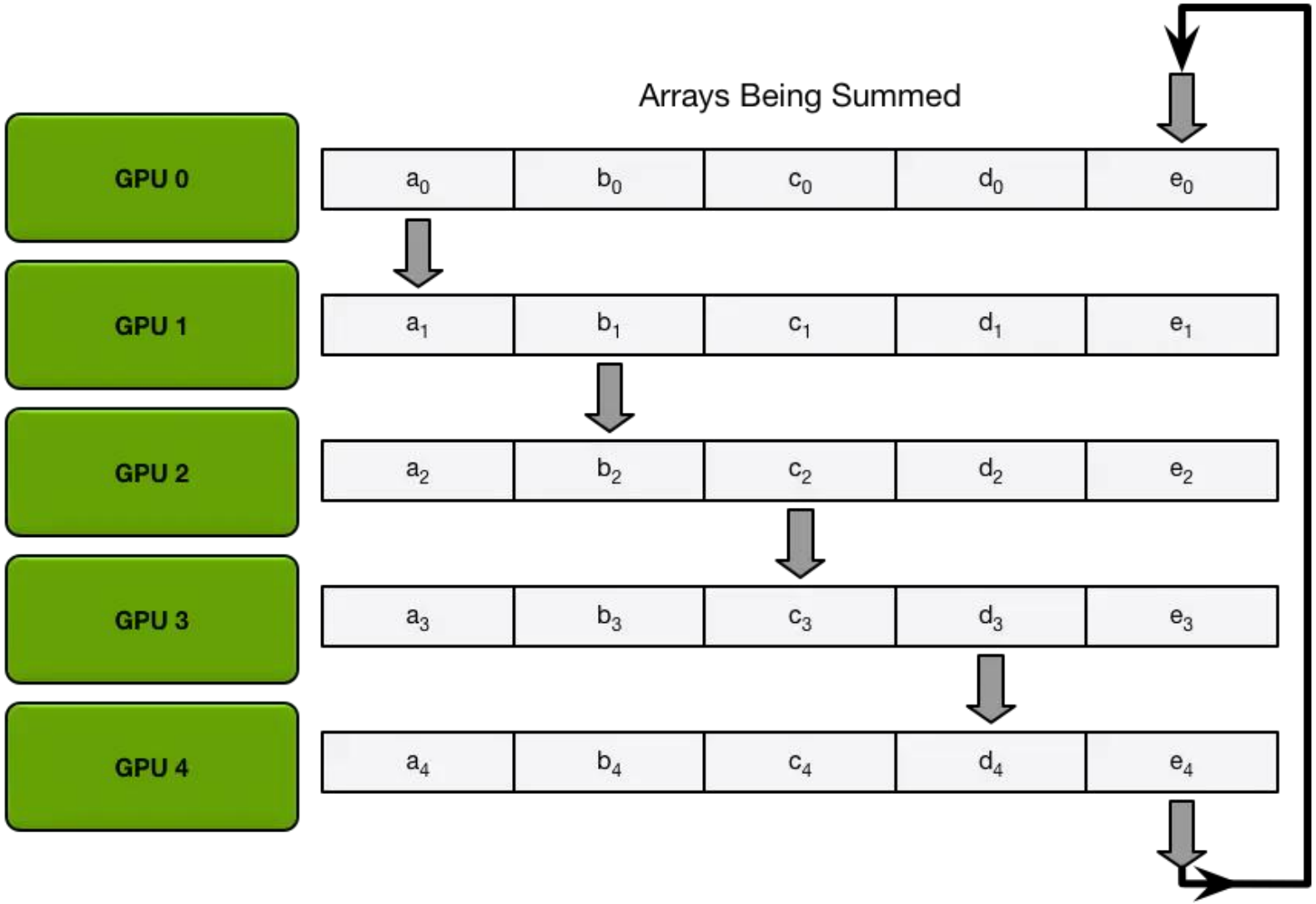
5. Ring Allreduce 的应用
Ring Allreduce 被广泛应用于分布式深度学习框架中，例如：
- **NVIDIA NCCL**：NVIDIA 的集合通信库（NCCL）实现了高效的 Ring Allreduce，支持多 GPU 和多节点训练。
- **Horovod**：Uber 开发的 Horovod 框架使用 Ring Allreduce 进行分布式训练，支持 TensorFlow、PyTorch 等深度学习框架。
- **PyTorch Distributed**：PyTorch 的分布式训练模块也支持 Ring Allreduce，用于高效的多 GPU 和多节点训练。

Scatter-Reduce

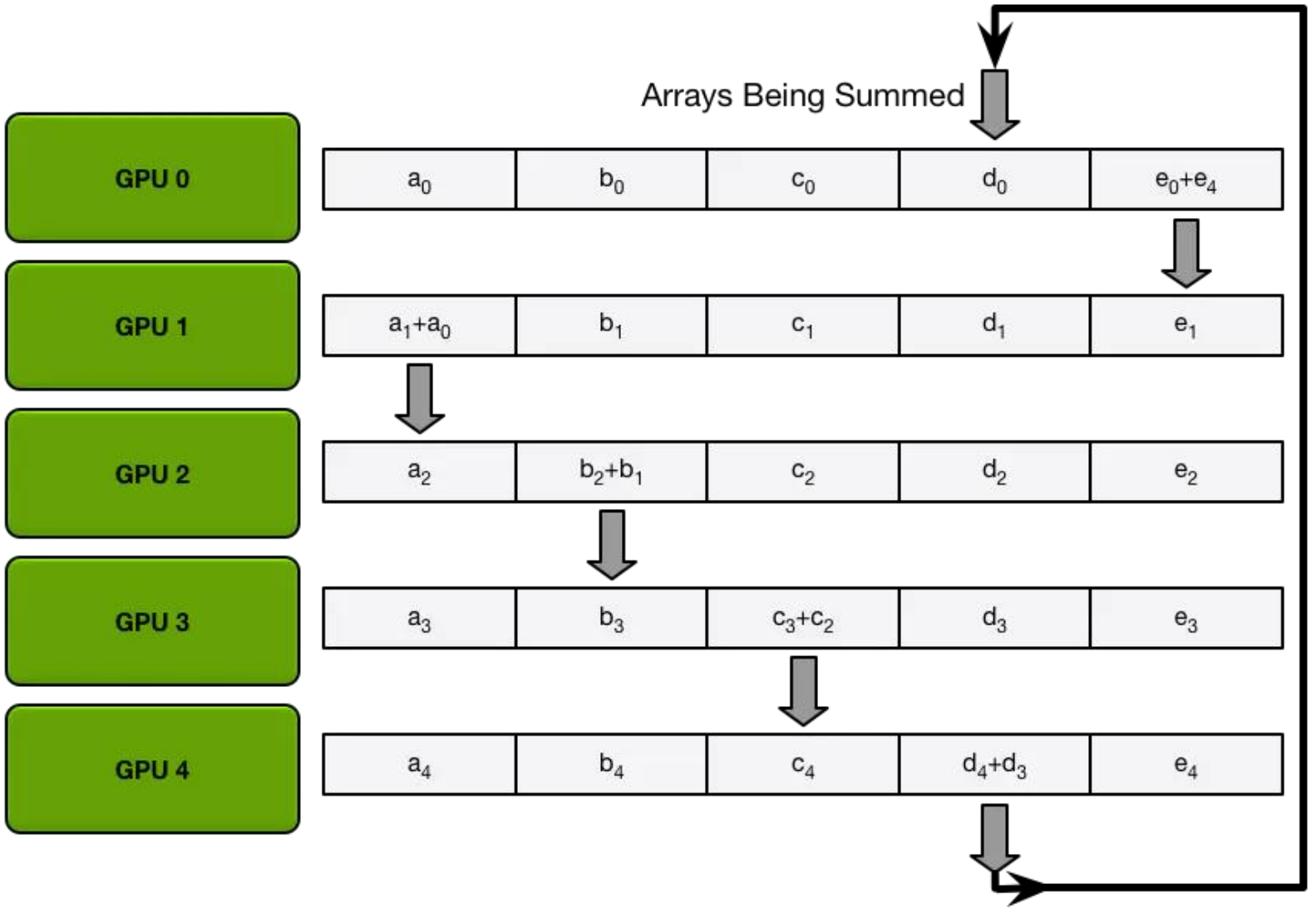


Scatter-Reduce 的核心思想是将数据分块，并在多个节点之间逐步传递和聚合。具体步骤如下：
通过这种方式，Scatter-Reduce 将全局聚合任务分解为多个局部聚合任务，减少了单次通信的数据量，从而提高了效率。

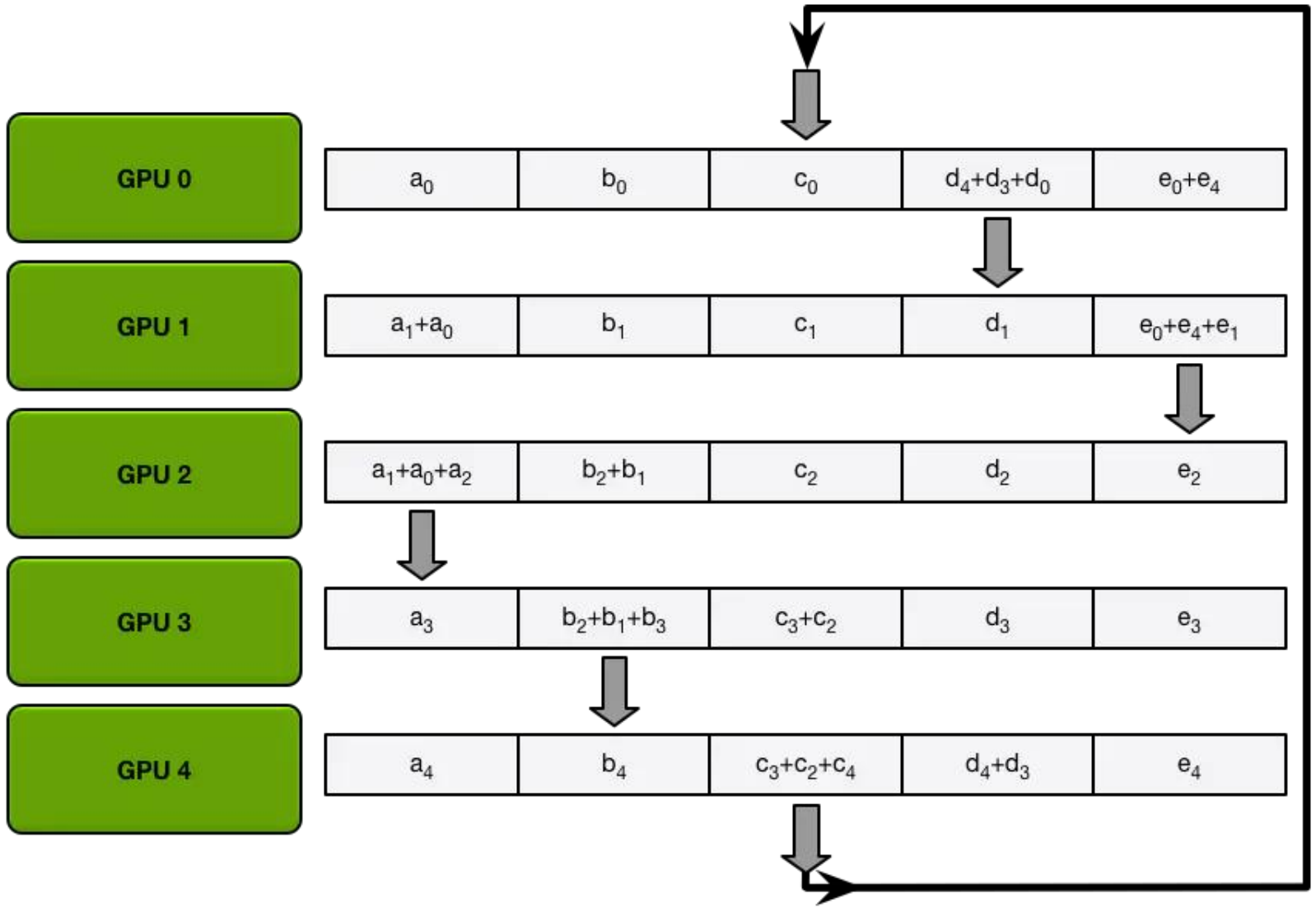
Scatter-Reduce



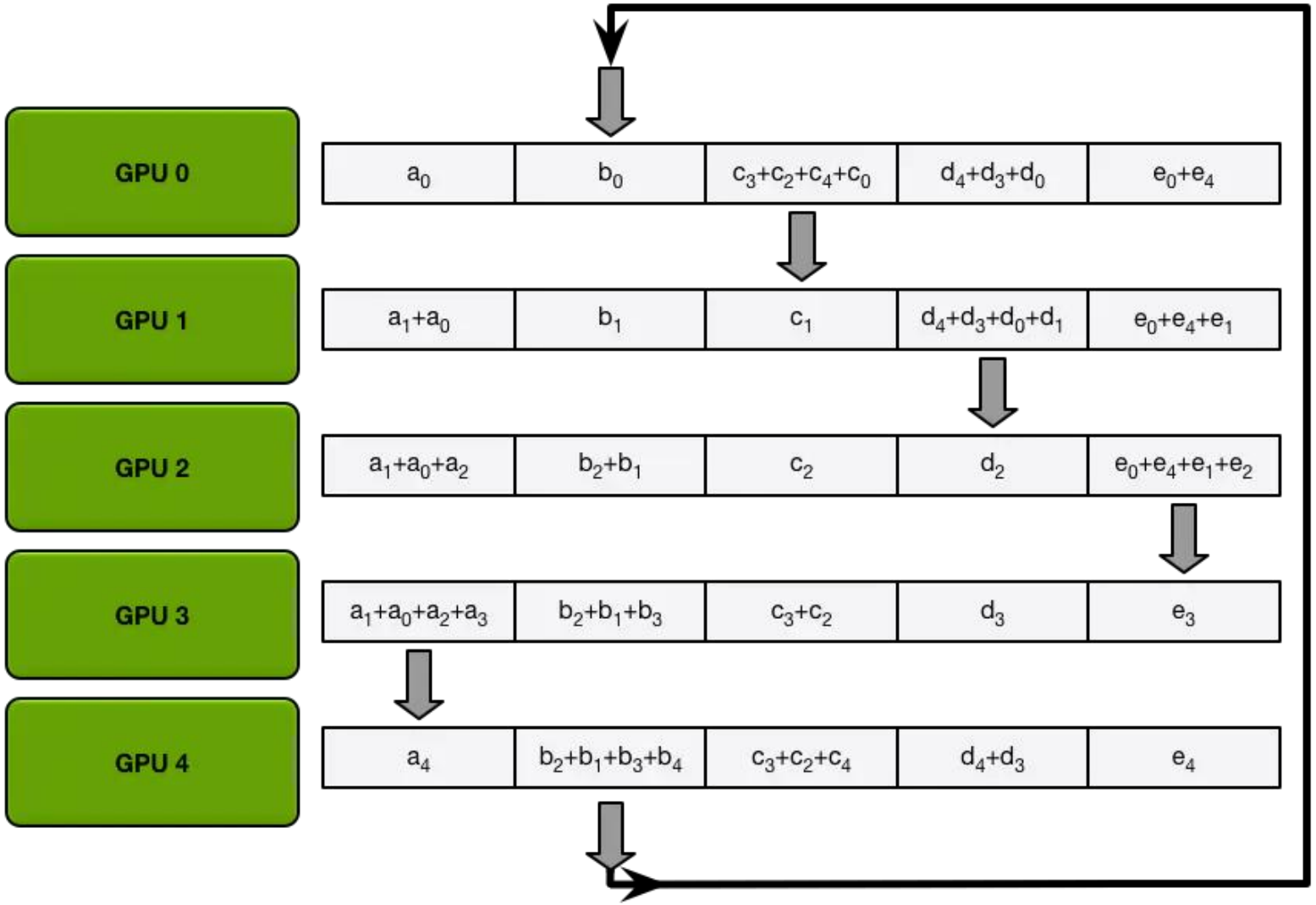
Scatter-Reduce



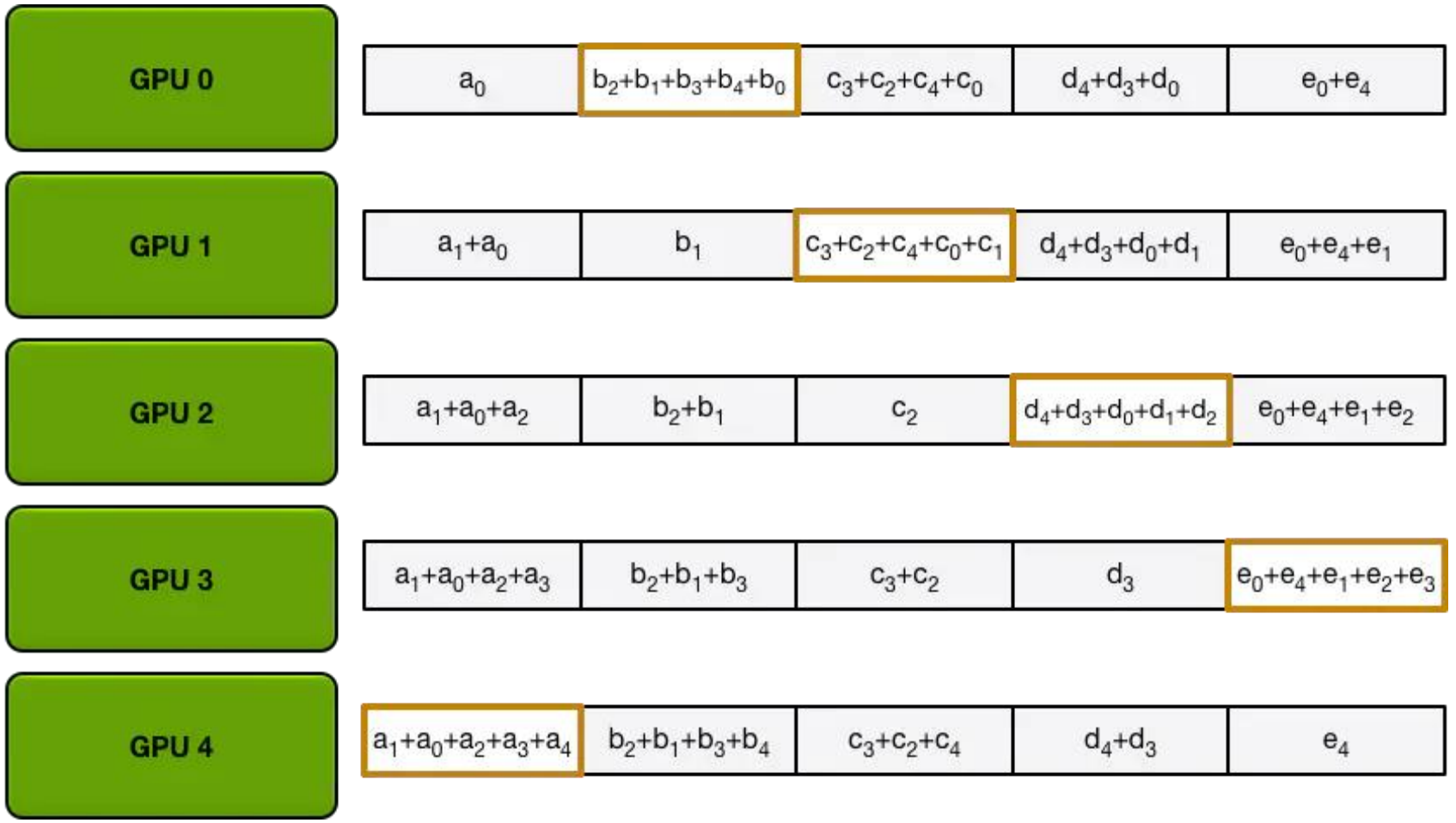
Scatter-Reduce



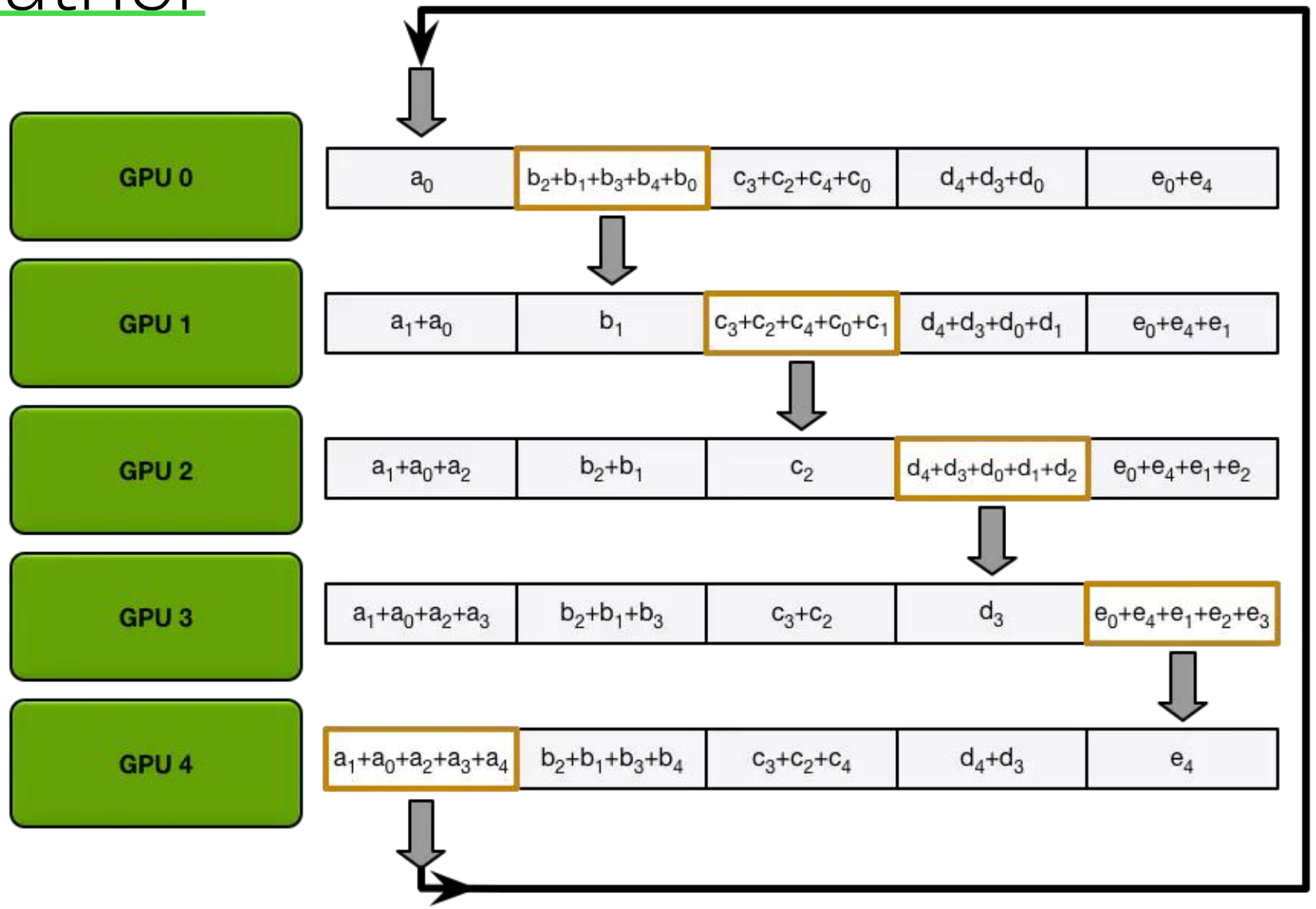
Scatter-Reduce



Scatter-Reduce

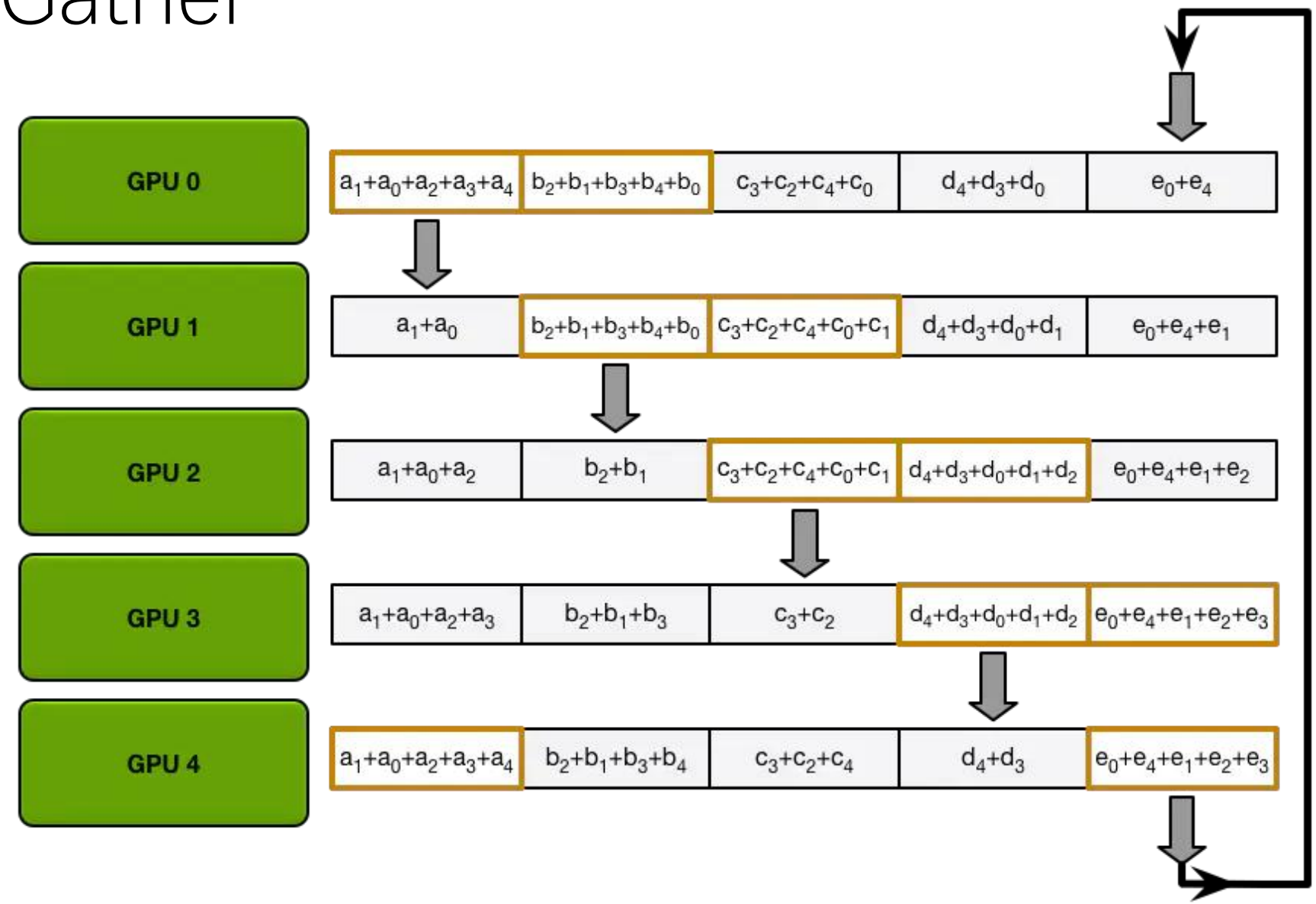


All-Gather

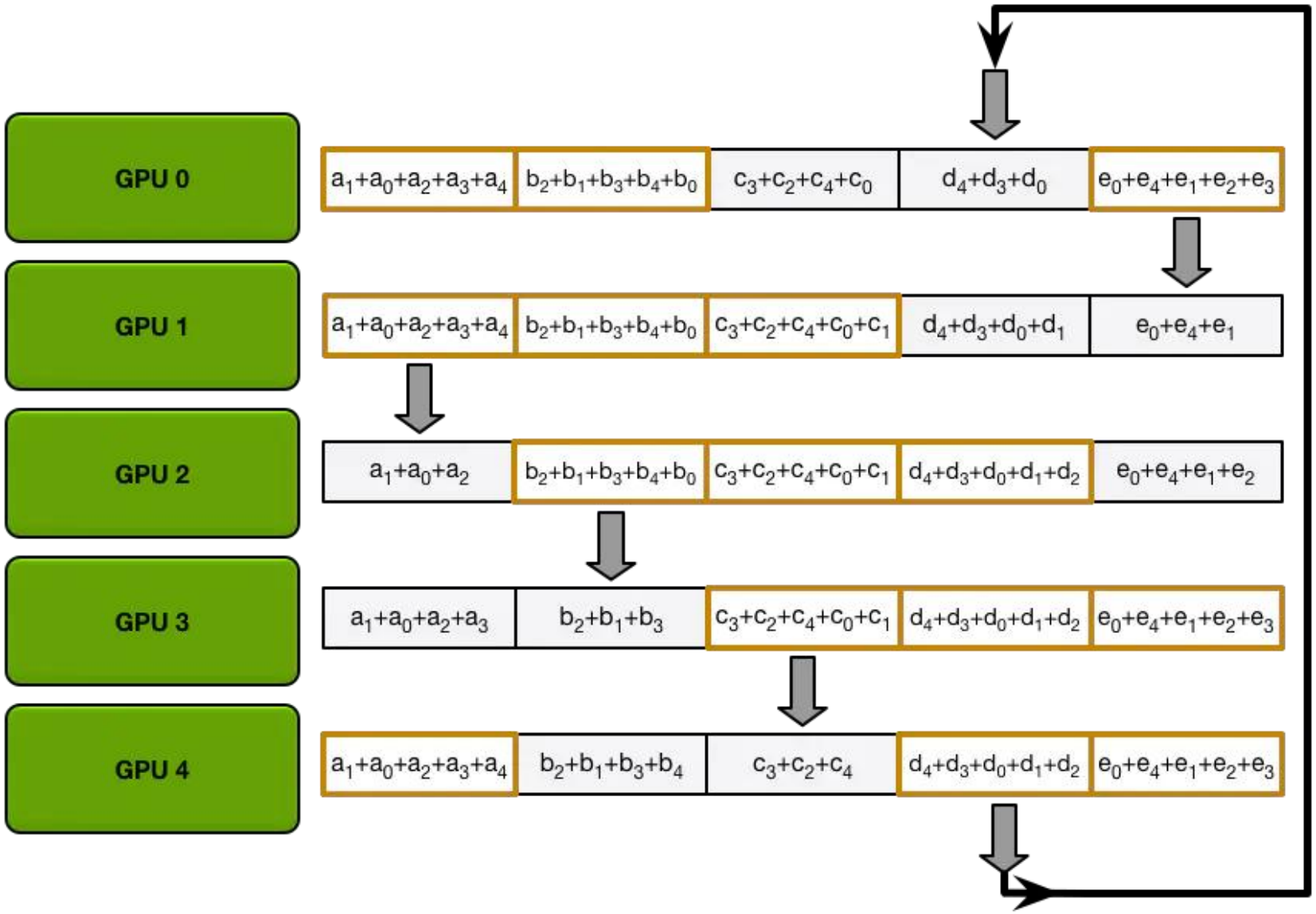


All-Gather 的核心思想是将数据分块，并在多个节点之间逐步传递和收集。具体步骤如下：通过这种方式，All-Gather 将全局数据广播任务分解为多个局部传递任务，减少了单次通信的数据量，从而提高了效率。

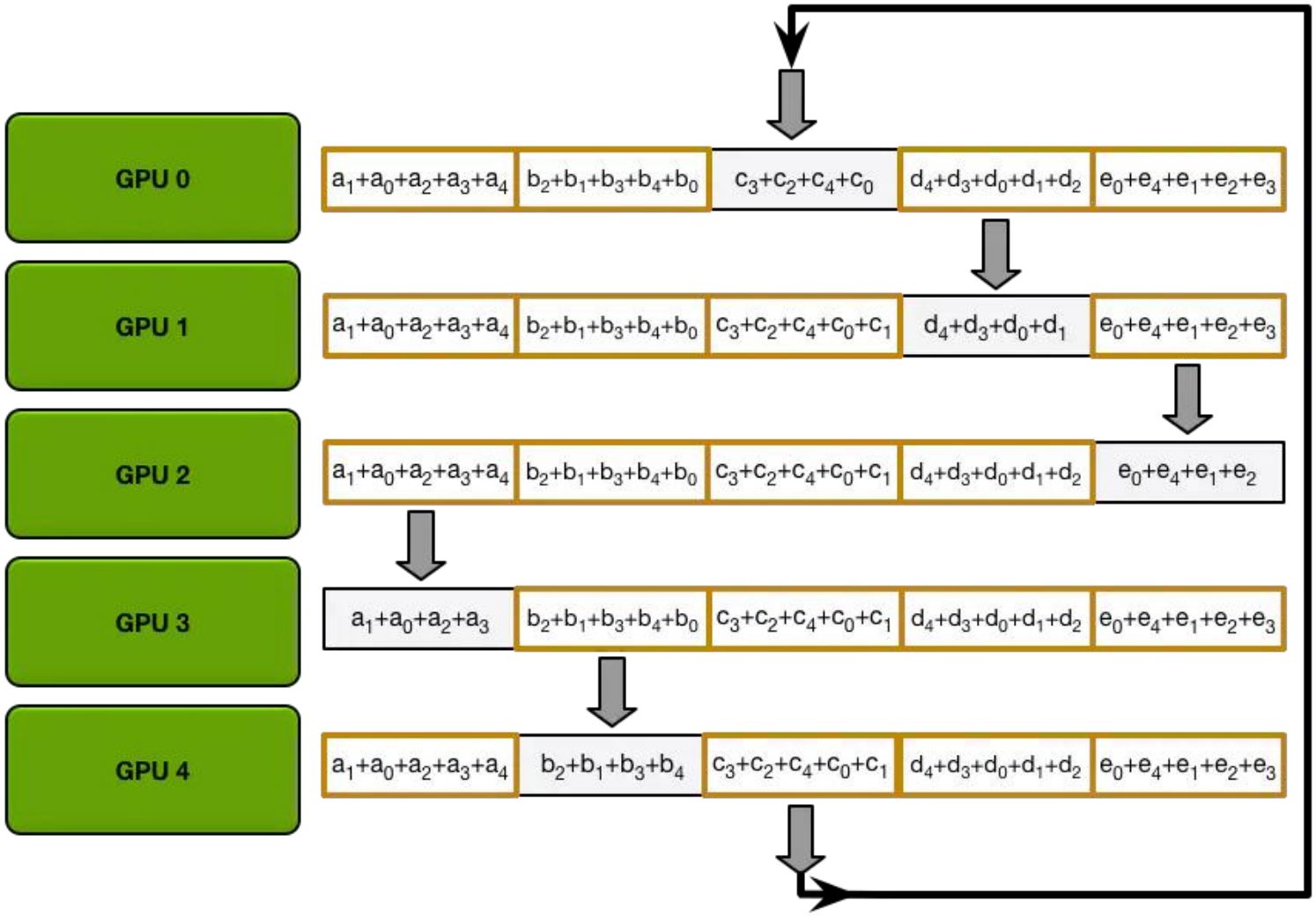
All-Gather


GPU 0
GPU 1
GPU 2
GPU 3
GPU 4
$a_1+a_0+a_2+a_3+a_4$ $b_2+b_1+b_3+b_4+b_0$ $c_3+c_2+c_4+c_0$ $d_4+d_3+d_0$ e_0+e_4
a_1+a_0 $b_2+b_1+b_3+b_4+b_0$ $c_3+c_2+c_4+c_0+c_1$ $d_4+d_3+d_0+d_1$ $e_0+e_4+e_1$
$a_1+a_0+a_2$ b_2+b_1 $c_3+c_2+c_4+c_0+c_1$ $d_4+d_3+d_0+d_1+d_2$ $e_0+e_4+e_1+e_2$
$a_1+a_0+a_2+a_3$ $b_2+b_1+b_3$ c_3+c_2 $d_4+d_3+d_0+d_1+d_2$ $e_0+e_4+e_1+e_2+e_3$
$a_1+a_0+a_2+a_3+a_4$ $b_2+b_1+b_3+b_4$ $c_3+c_2+c_4$ d_4+d_3 $e_0+e_4+e_1+e_2+e_3$

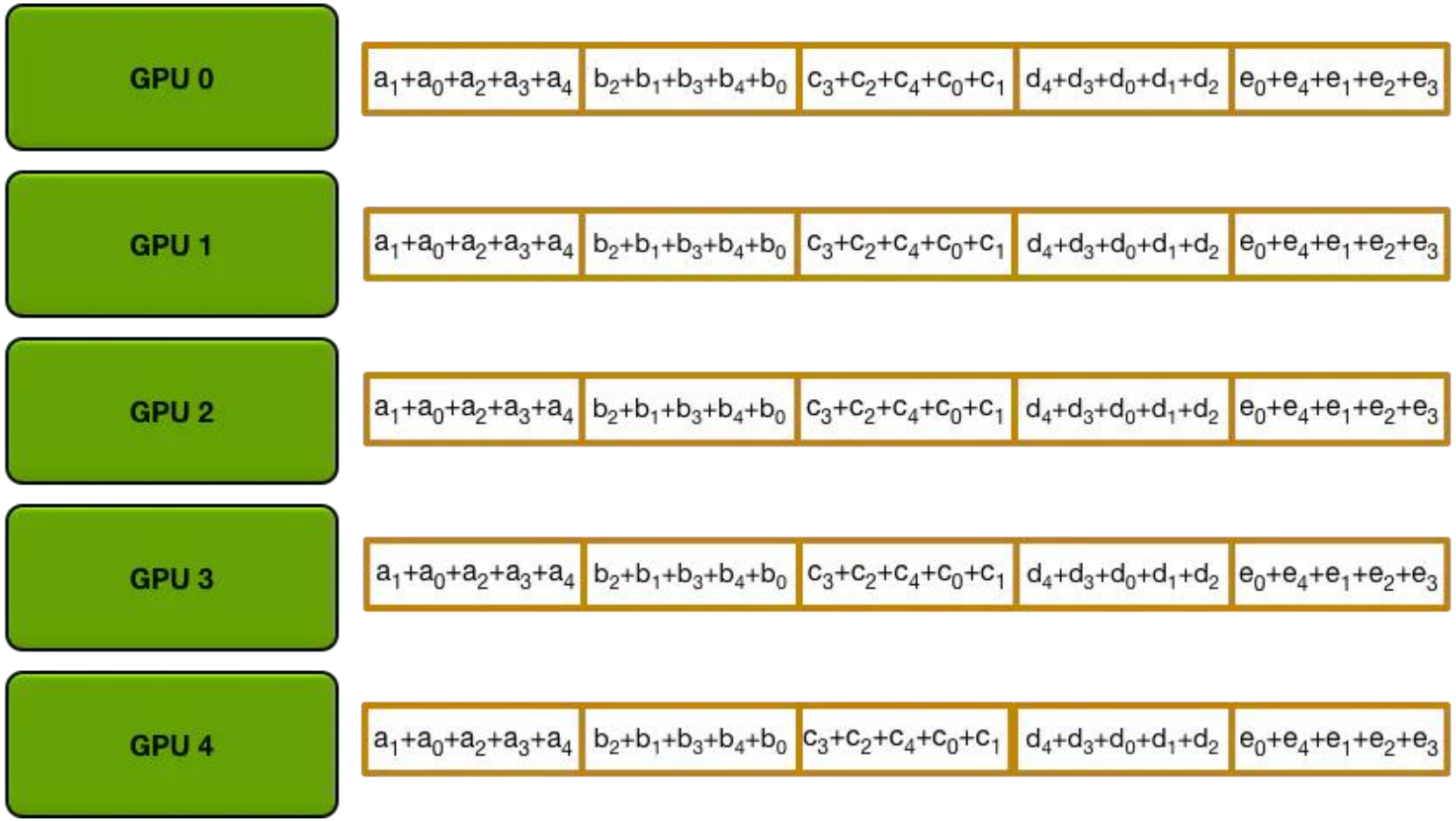
All-Gather


GPU 0
GPU 1
GPU 2
GPU 3
GPU 4
$a_1+a_0+a_2+a_3+a_4$ $b_2+b_1+b_3+b_4+b_0$ $c_3+c_2+c_4+c_0$ $d_4+d_3+d_0$ $e_0+e_4+e_1+e_2+e_3$
$a_1+a_0+a_2+a_3+a_4$ $b_2+b_1+b_3+b_4+b_0$ $c_3+c_2+c_4+c_0+c_1$ $d_4+d_3+d_0+d_1$ $e_0+e_4+e_1$
$a_1+a_0+a_2$ $b_2+b_1+b_3+b_4+b_0$ $c_3+c_2+c_4+c_0+c_1$ $d_4+d_3+d_0+d_1+d_2$ $e_0+e_4+e_1+e_2$
$a_1+a_0+a_2+a_3$ $b_2+b_1+b_3$ $c_3+c_2+c_4+c_0+c_1$ $d_4+d_3+d_0+d_1+d_2$ $e_0+e_4+e_1+e_2+e_3$
$a_1+a_0+a_2+a_3+a_4$ $b_2+b_1+b_3+b_4$ $c_3+c_2+c_4$ $d_4+d_3+d_0+d_1+d_2$ $e_0+e_4+e_1+e_2+e_3$

All-Gather


GPU 0
GPU 1
GPU 2
GPU 3
GPU 4
$a_1+a_0+a_2+a_3+a_4$
$b_2+b_1+b_3+b_4+b_0$
$c_3+c_2+c_4+c_0$
$d_4+d_3+d_0+d_1+d_2$
$e_0+e_4+e_1+e_2+e_3$
$a_1+a_0+a_2+a_3+a_4$
$b_2+b_1+b_3+b_4+b_0$
$c_3+c_2+c_4+c_0+c_1$
$d_4+d_3+d_0+d_1$
$e_0+e_4+e_1+e_2+e_3$
$a_1+a_0+a_2+a_3+a_4$
$b_2+b_1+b_3+b_4+b_0$
$c_3+c_2+c_4+c_0+c_1$
$d_4+d_3+d_0+d_1+d_2$
$e_0+e_4+e_1+e_2$
$a_1+a_0+a_2+a_3$
$b_2+b_1+b_3+b_4+b_0$
$c_3+c_2+c_4+c_0+c_1$
$d_4+d_3+d_0+d_1+d_2$
$e_0+e_4+e_1+e_2+e_3$
$a_1+a_0+a_2+a_3+a_4$
$b_2+b_1+b_3+b_4$
$c_3+c_2+c_4+c_0+c_1$
$d_4+d_3+d_0+d_1+d_2$
$e_0+e_4+e_1+e_2+e_3$

All-Gather


GPU 0
GPU 1
GPU 2
GPU 3
GPU 4
$a_1+a_0+a_2+a_3+a_4$ $b_2+b_1+b_3+b_4+b_0$ $c_3+c_2+c_4+c_0+c_1$ $d_4+d_3+d_0+d_1+d_2$ $e_0+e_4+e_1+e_2+e_3$

Ring Allreduce Cost

- N-1次scatter-reduce
- N-1次allgather。
- 每个GPU一次通信量：K / N，其中K是总数据大小
- 每个GPU通信次数：2(N-1)

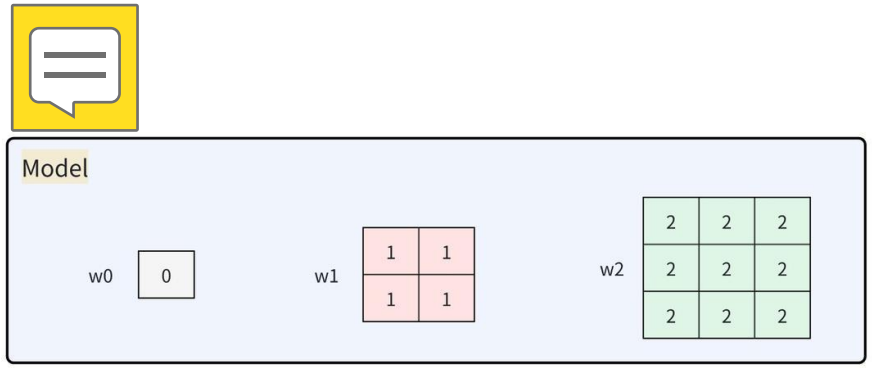
因此，总通信量为：

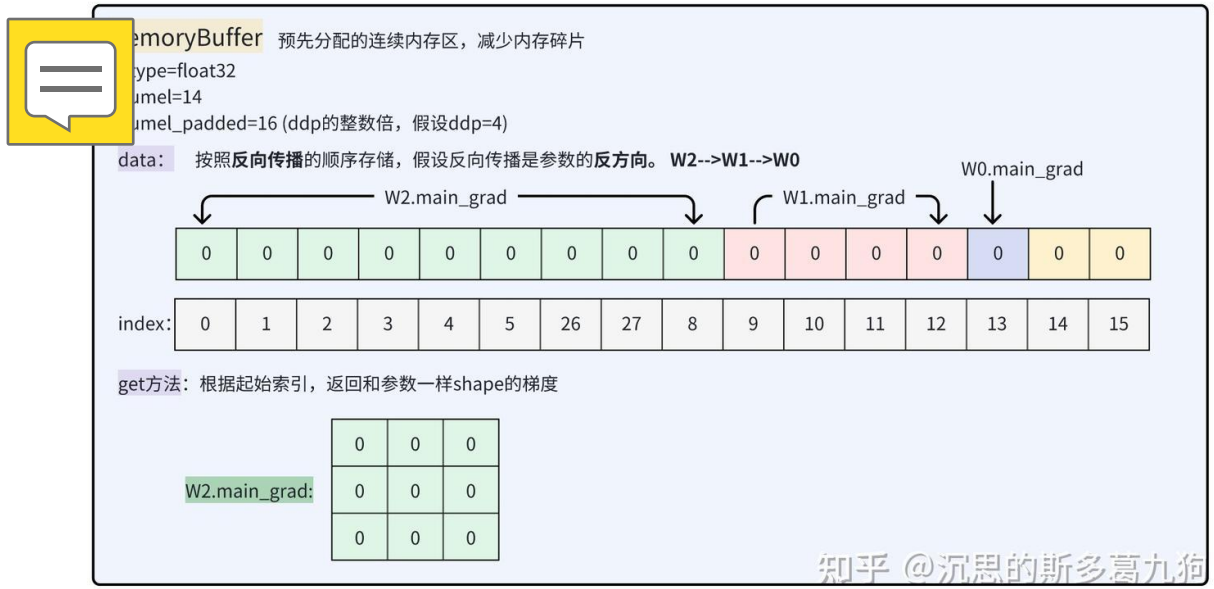
$$\text{Data Transferred} = 2(N-1)\frac{K}{N}$$

Gradient Bucketing

Motivation：集合通信在大张量上更有效。

Idea：在短时间内等待并将多个梯度存储到一个数据桶，然后进行AllReduce操作。而不是对每个参数梯度立刻启动AllReduce操作。





MindSpore数据并行

```python
def forward_fn(data, target):
    """forward propagation"""
    logits = net(data)
    loss = loss_fn(logits, target)
    return loss, logits

grad_fn = ms.value_and_grad(forward_fn, None, net.trainable_params(), has_aux=True)
grad_reducer = nn.DistributedGradReducer(optimizer.parameters)

for epoch in range(10):
    i = 0
    for image, label in data_set:
        (loss_value, _), grads = grad_fn(image, label)
        grads = grad_reducer(grads)
        optimizer(grads)
        if i % 10 == 0:
            print("epoch: %s, step: %s, loss is %s" % (epoch, i, loss_value))
        i += 1
```

数据并行:

数据并行是指将数据分割成多个小批量，并在多个 GPU 上并行处理这些小批量。在每次迭代中，每个 GPU 都会处理一部分数据，并独立地计算梯度。

最后，通过 DistributedGradReducer 将所有 GPU 上的梯度进行归约，确保模型参数的一致性。

梯度归约:

DistributedGradReducer 是一个关键组件，它负责将多个 GPU 上的梯度进行归约。归约操作通常采用环形全reduce（Ring AllReduce）算法，以减少通信延迟。

高效性:

通过数据并行，可以充分利用多 GPU 环境中的计算资源，显著提高训练速度。value_and_grad 函数同时计算损失值和梯度，减少了不必要的计算开销。

灵活性:

MindSpore 提供了灵活的 API，使得用户可以轻松地实现数据并行。可以根据具体需求调整训练过程中的参数和逻辑

扩展性:

这种策略可以很好地扩展到更多的 GPU，只需增加设备数量即可。MindSpore 支持自动分片和分布式训练，使得扩展性非常好。

https://gitee.com/mindspore/docs/tree/master/docs/sample_code/distributed_data_parallel

MindNLP数据并行

```python
def update_gradient_by_distributed_type(self, model: nn.Module) -> None:
    """update gradient by distributed_type"""
    if accelerate_distributed_type == DistributedType.NO:
        return
    if accelerate_distributed_type == DistributedType.MULTI_NPU:
        from mindspore.communication import get_group_size
        from mindspore.communication.comm_func import all_reduce
        rank_size = get_group_size()
        for parameter in model.parameters():
            new_grads_mean = all_reduce(parameter.grad) / rank_size
            parameter.grad = new_grads_mean
```

MindNLP数据并行

```
    def training_step(self, model: nn.Module, inputs: Dict[str, Union[mindspore.Tensor, Any]]) -> Tuple[List[mindspore.Tensor],
mindspore.Tensor]:
        """
        Perform a training step on a batch of inputs.

@@ -1382,7 +1422,7 @@ def forward(inputs):

            self.grad_fn = value_and_grad(forward, weights, attach_grads=True)


        loss = self.grad_fn(inputs)
-
+        self.update_gradient_by_distributed_type(model)
        return loss / self.args.gradient_accumulation_steps
```

详细解释
函数定义:
定义了一个名为 training_step 的方法，该方法接受一个 nn.Module 类型的模型和一个包含输入数据的字典。
前向传播函数:
使用 value_and_grad 函数来同时计算损失值和梯度。
forward: 前向传播函数。
weights: 模型的权重参数。
attach_grads=True: 表示需要计算梯度。
计算损失:
调用 self.grad_fn 来计算当前批次的损失值。
更新梯度:
调用 self.update_gradient_by_distributed_type(model) 方法来更新梯度。
这个方法负责执行分布式梯度归约操作，确保所有设备上的梯度一致。
返回损失:
返回计算得到的损失值，并除以 gradient_accumulation_steps，这是为了在多个小批量上累积梯度后再进行一次优化器更新。

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

MindNLP数据并行实操

```
EXEC_PATH=$(pwd)
if [ ! -d "${EXEC_PATH}/data" ]; then
    if [ ! -f "${EXEC_PATH}/emotion_detection.tar.gz" ]; then
        wget wget https://baidu-nlp.bj.bcebos.com/emotion_detection-dataset-1.0.0.tar.gz -O emotion_detection.tar.gz
    fi
    tar xvf emotion_detection.tar.gz
fi
export DATA_PATH=${EXEC_PATH}/data/

rm -rf bert_imdb_finetune_cpu_mindnlp_trainer_npus_same
mkdir bert_imdb_finetune_cpu_mindnlp_trainer_npus_same
echo "start training"

export MULTI_NPU="true"
export ASCEND_SLOG_PRINT_TO_STDOUT=1

msrun --worker_num=2 --local_worker_num=2 --master_port=8121 \
--log_dir=bert_imdb_finetune_cpu_mindnlp_trainer_npus_same --join=True \
--cluster_time_out=10 bert_imdb_finetune_cpu_mindnlp_trainer_npus_same.py
```

https://github.com/mindspore-lab/mindnlp/tree/master/examples/parallel/bert_imdb_finetune

作业

- 基于Roberta模型
- 选择一个新的分类数据集
- 尝试仿照Bert数据并行代码进行多卡训练(2卡即可)

Q&A