PyTorch Distributed and Parallel Training - 2

20230412
박현우

Table of Contents

1. What is FSDP

1. Remind DDP
2. What is FSDP
3. Superiority of FSDP over other parallelisms

1.1 Remind DDP

In DistributedDataParallel, (DDP) training,
each process/ worker

1. owns a replica of the (complete) model
2. and processes own batch of data,
3. finally it uses all-reduce to sum up gradients over different workers.

In DDP, these are replicated across all workers:

1. the model weights
2. and optimizer states

1.2 What is FSDP

Fully Sharded Data Parallel FSDP is Data and Model parallel training algorithm.

FSDP shards across ranks:

1. model parameters
2. optimizer states
3. and gradients

This is called Full Parameter Sharding.

Thus, FSDP can be regarded as Inter-layer Parallelism or Pipeline Parallelism.

1.2 What is FSDP

Suppose a model : $F(x) = l_3(l_2(l_1(x)))$

Optimizer State : Gradients through backward pass of complete model.
i.e. Backward pass through all l_1, l_2, l_3 (= F(x)).

Gradient : Gradient through both forward and backward pass of specific subset of the model (here, layers).
e.g.

1. Forward pass through only l_1.
2. Backward pass through l_2 (given Gradient of l_3)

i.e. Optimizer State = all backward Gradients

1.2 What is FSDP

Another core aspect of FSDP is that
although the parameters are sharded to different devices (GPUs),

FSDP makes the computation for each microbatch of data still local to each GPU worker.

= Minimize amount of communications while having Full Parameter Sharding across the cluster.

1.2 What is FSDP

Although minimized, FSDP has still larger communication volume compared to only data-parallelisms,
caused by Gradient Synchronization.

Such increased communication overhead is reduced by internal optimizations of FSDP;

i.e.

1. Decomposing Communication and Computation
2. and Overlapping them during training.

1.3 Superiority of FSDP over other parallelism

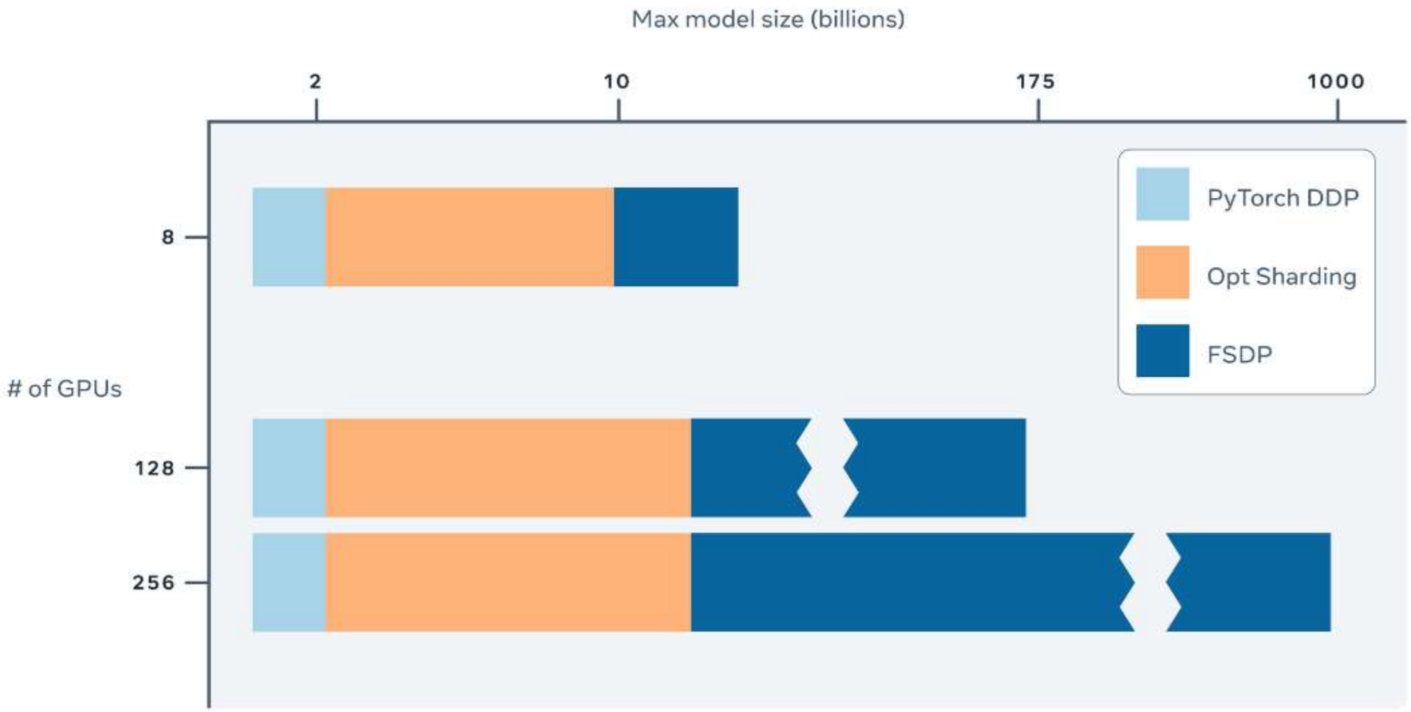
FSDP Device Local memory footprint would be smaller than that of DDP across all workers;

as FSDP workers only contains shards of the model, not the whole replica.

Gains of FSDP:

1. This makes the training feasible of some very large models
2. and helps to fit larger batch sizes for our training job.

1.3 Superiority of FSDP over other parallelism

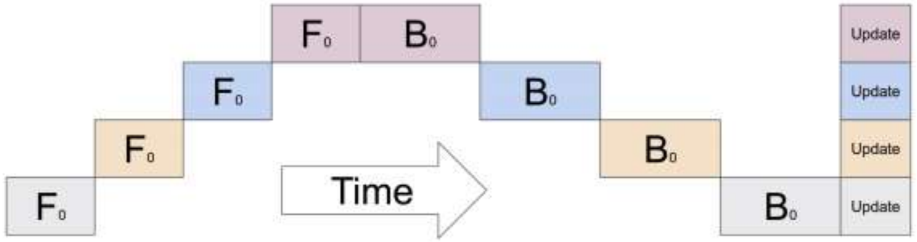


1.3 Superiority of FSDP over other parallelism

Even with Full Parameter Sharding,
FSDP makes computation local on each device.

This conceptual simplicity makes FSDP

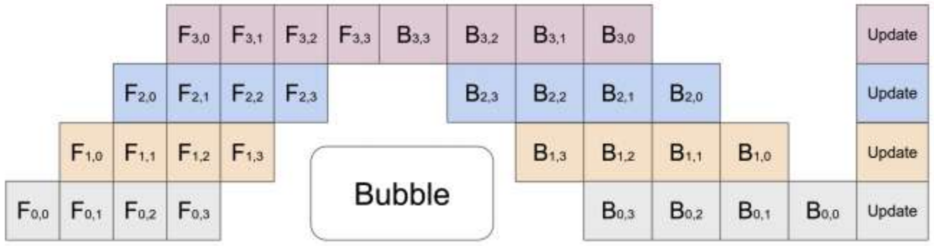
1. easier to understand
2. and more applicable to a wide range of usage scenarios.



(compared to

1. intra-layer parallelism
2. and pipeline parallelism

which are typically model-specific).



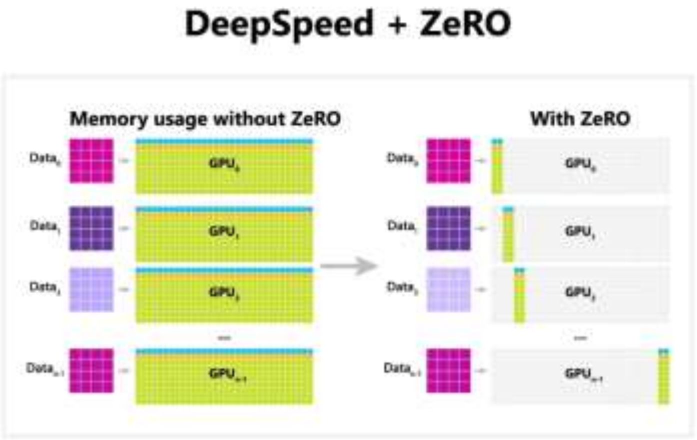
1.3 Superiority of FSDP over other parallelism

Such Full Parameter Sharding of FSDP is compatible with many other strategies.

Thus, FSDP can be easily integrated with other algorithms.

e.g. ZeRO-3 (ZeRO-Infinity) of DeepSpeed.

i.e. Parallelized optimizer
for model and data parallelism



1.3 Superiority of FSDP over other parallelism

As well as main FSDP parallelism,
FSDP also supports CPU offload for parameters and gradient.

e.g. Zero-Offload series of DeepSpeed

This makes possible to train even larger model.

(However, due to frequent copying of tensors from host to device

this feature may slow down the training considerably.)

2. How FSDP operates

1. Decomposition of All-Reduce
2. Structure of FSDP
3. Procedure of FSDP
4. Comparison to DDP
5. Full Architecture of FSDP

2.1. Decomposition of All-Reduce

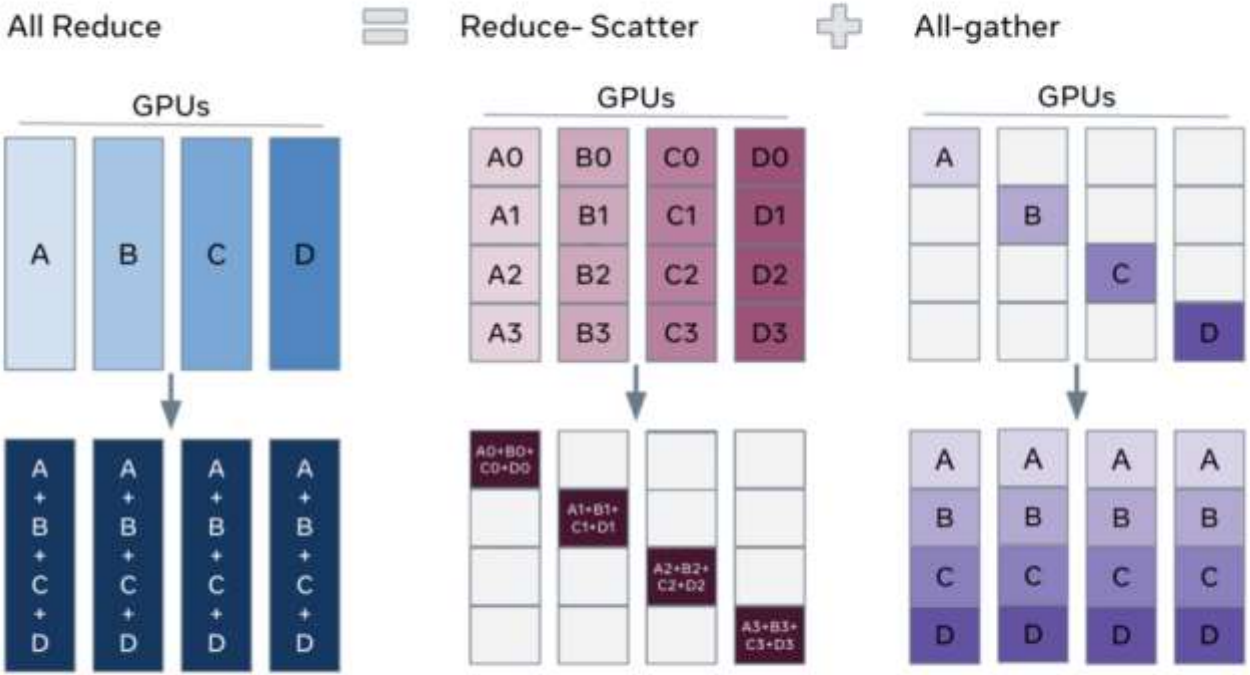
In standard DDP training,

1. every worker processes own separate batch
2. and the gradients are summed (sync-ed) across workers using an all-reduce operation.

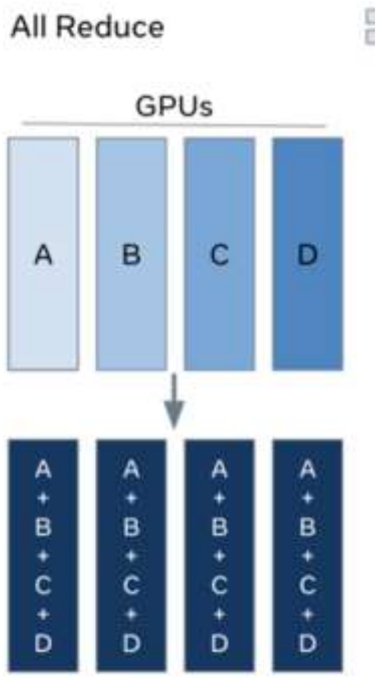
The key insight to unlock full parameter sharding is that
we can decompose the all-reduce operations in DDP into separate operations:

1. reduce-scatter
2. and all-gather

2.1. Decomposition of All-Reduce



2.1. Decomposition of All-Reduce

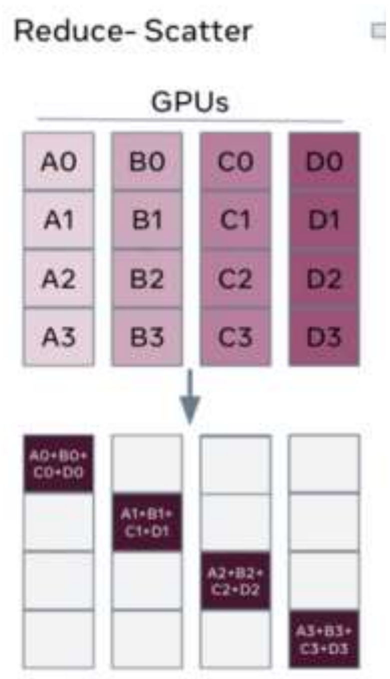


In standard DDP,

Optimizer states with respect to each batches are sync-ed across all devices via All-Reduce.

In FSDP,
such Optimizer states are decomposed into multiple Gradients.

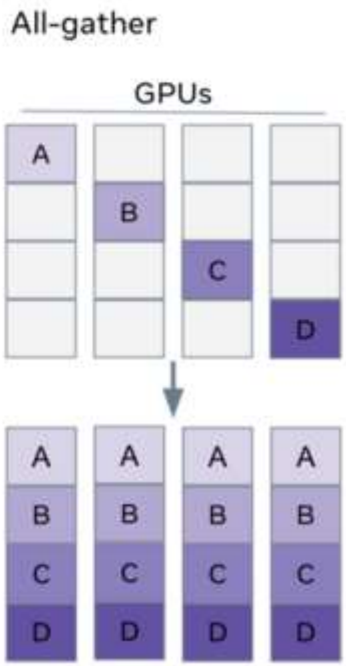
2.1. Decomposition of All-Reduce



During the reduce-scatter phase,

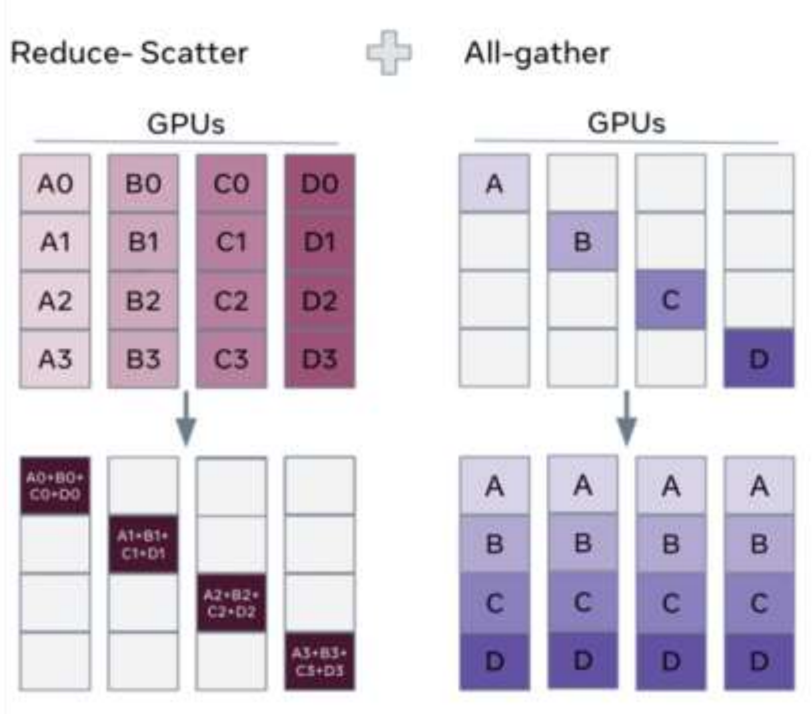
the gradients are aggregated in equal blocks among ranks based on their rank index.

2.1. Decomposition of All-Reduce



During the all-gather phase,
the sharded portion of aggregated gradients on each GPU are propagated to all GPUs.

2.1. Decomposition of All-Reduce



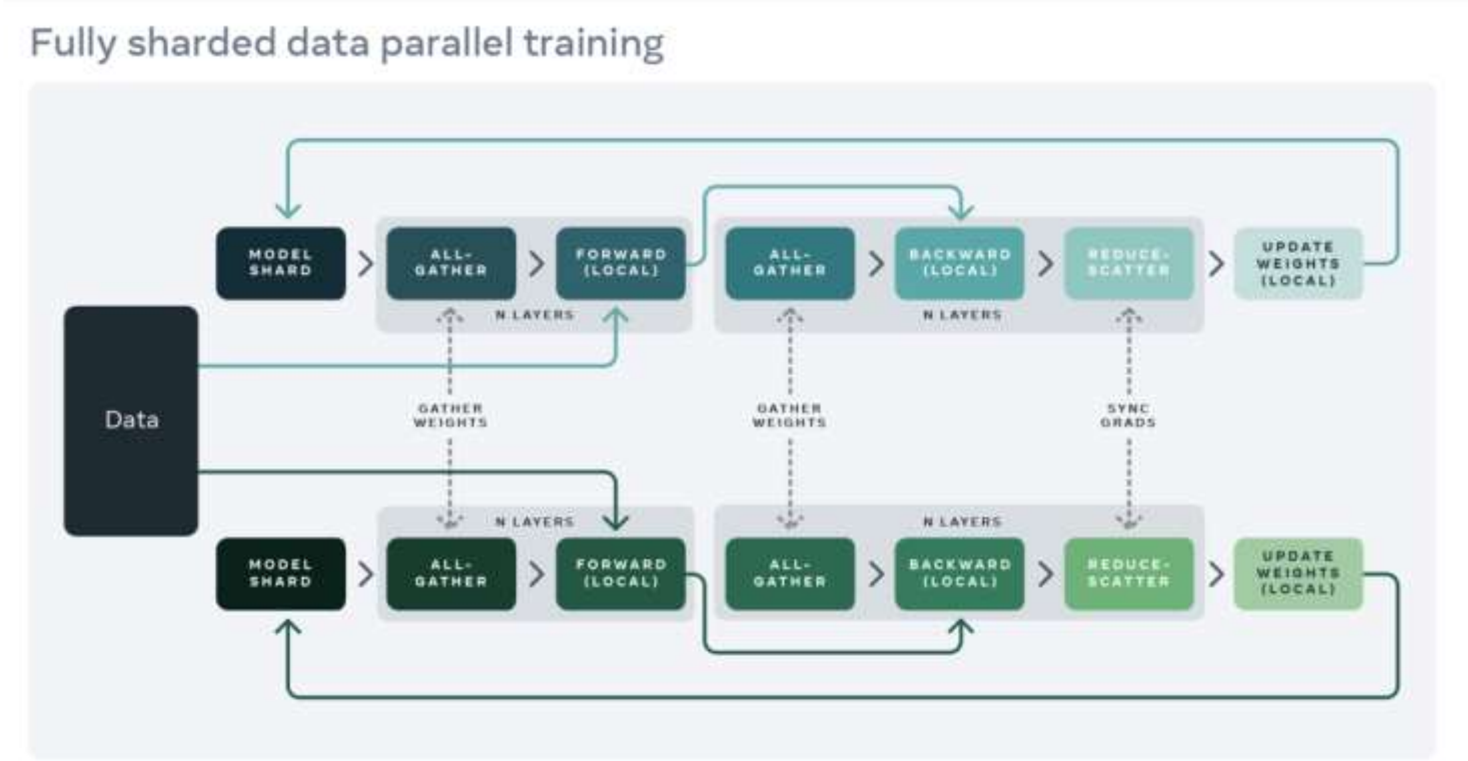
Once All-Reduce decomposed into multiple Reduce-Scatter + All-Gather,

we can then rearrange them

so that each worker stores only a single shard of

1. parameters
2. and optimizer states (= Gradients).

2.2. Structure of FSDP



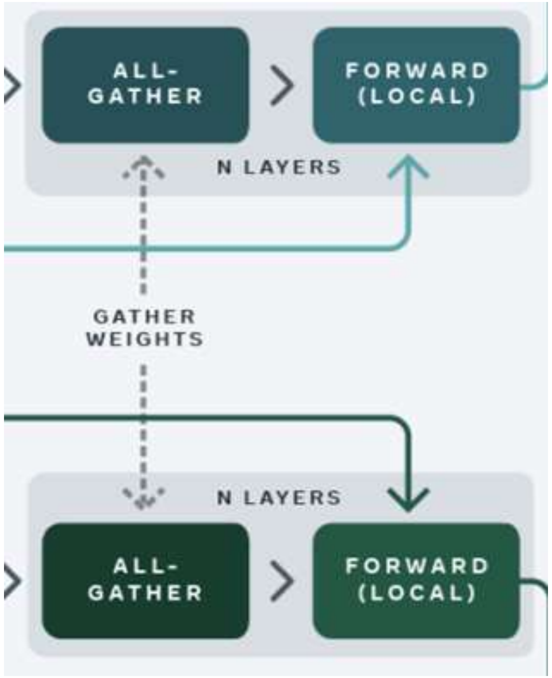
2.2. Structure of FSDP

In FSDP, only a shard of the model is present on a GPU.
i.e. In this figure, N layers of the model.

c.f. Full model is replicated across all GPU in DDP.

Then, in order to compute full forward/backward pass,
all weights are gathered from the other GPUs by means of an all-gather step.

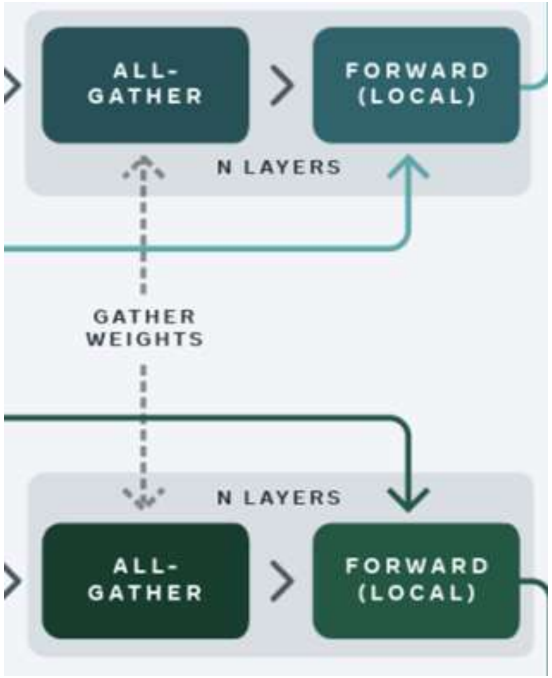
2.3. Steps of FSDP



Assume :

1. There are 2 Machines with 1 GPU each
2. First N Layers are in Rank 0, remain N layers are in Rank 1.
   i.e. FSDP instance 1:N layers
3. Dataset is sharded for each machine.
   (Dataset 0 and Dataset 1)

2.3.1. Forward Pass

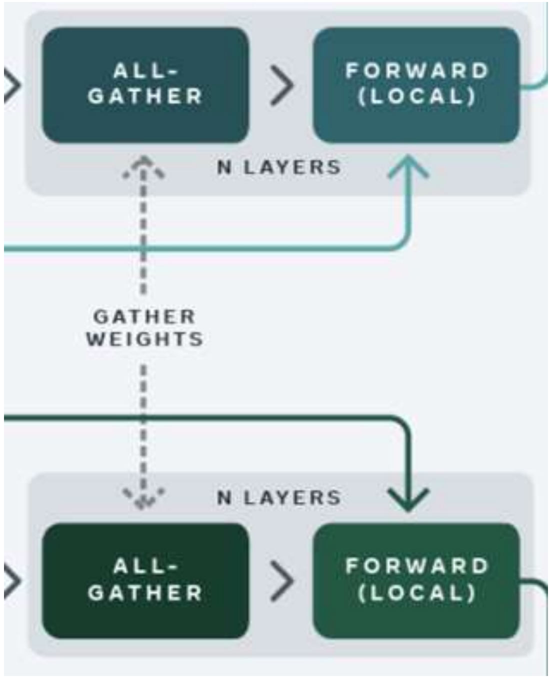


For Layer_1, (in Rank 0)

1. Rank 1 calls All-Gather for Layer_1,
   as Rank 1 doesn't have it.
   (Rank 0 can skip this, as Rank 0 holds it on itself.)
2. Both Rank 0 and 1 computes for Layer_1 on their own dataset.

(... for all layers)

2.3.1. Forward Pass

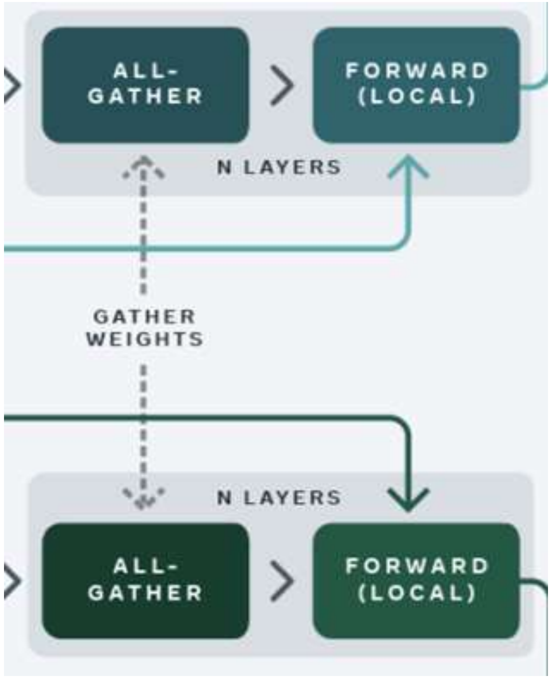



```
FSDP forward pass:
    for layer_i in layers:
        all-gather full weights for layer_i
        forward pass for layer_i
        discard full weights for layer_i
```

To maximize memory efficiency,

discard full weights for each layer after that layer's forward pass.

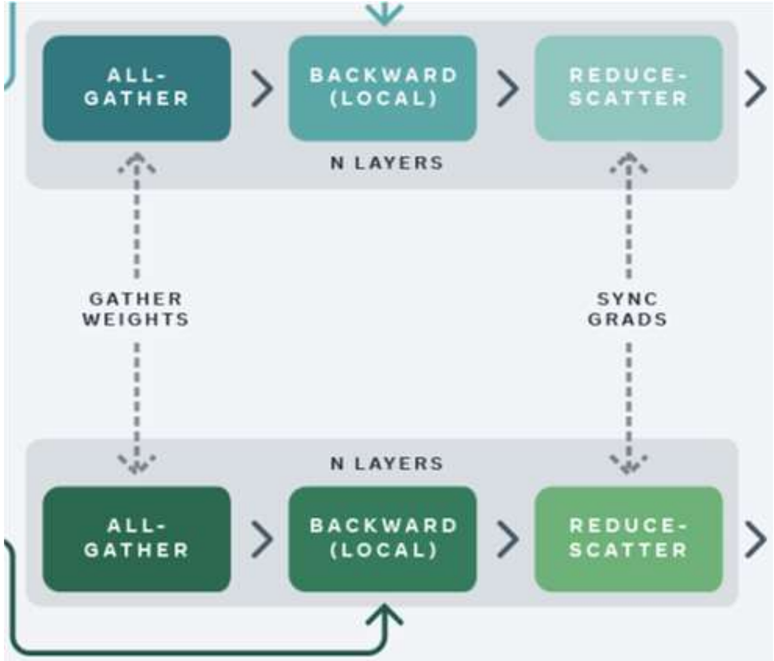
2.3.1. Forward Pass



At the end of forward pass,

each device contains outputs of full model with respect to their own dataset.

2.3.2. Backward Pass



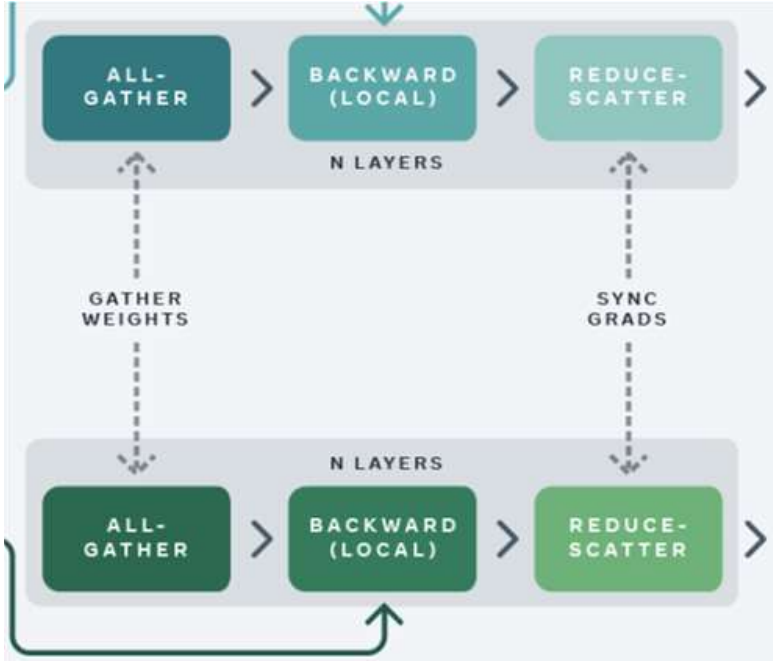
Computation of backward pass is the same as forward pass.

Only difference is that after each backward pass
(i.e. for each layer),

the local gradients are sync-ed (= averaged) across the GPUs

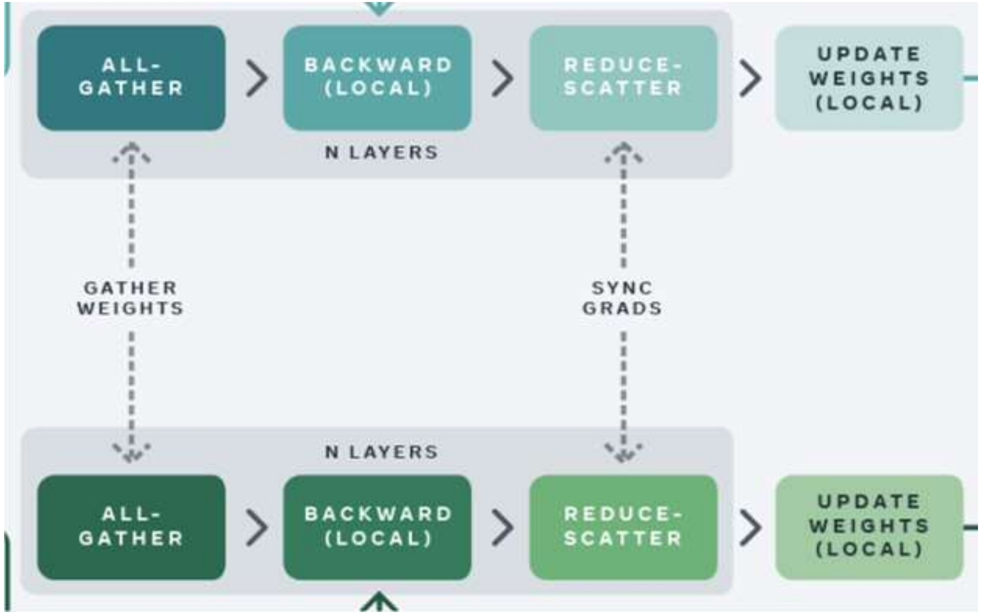
by means of a reduce-scatter step

2.3.2. Backward Pass




```
FSDP backward pass:
    for layer_i in layers:
        all-gather full weights for layer_i
        backward pass for layer_i
        discard full weights for layer_i
        reduce-scatter gradients for layer_i
```

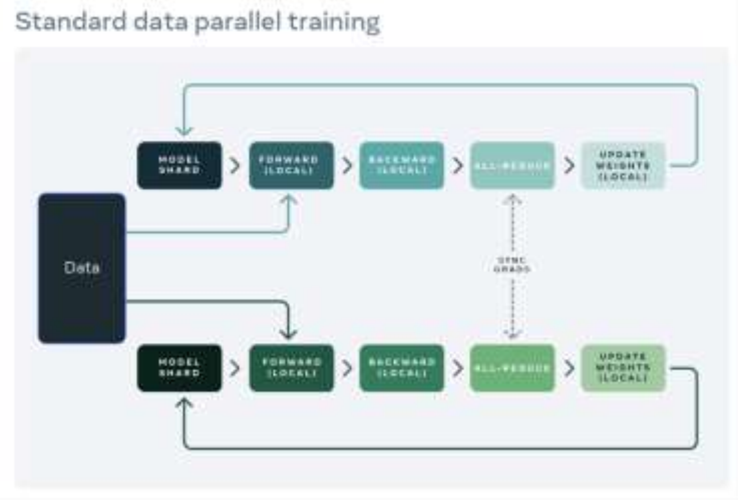
2.3.3. Update

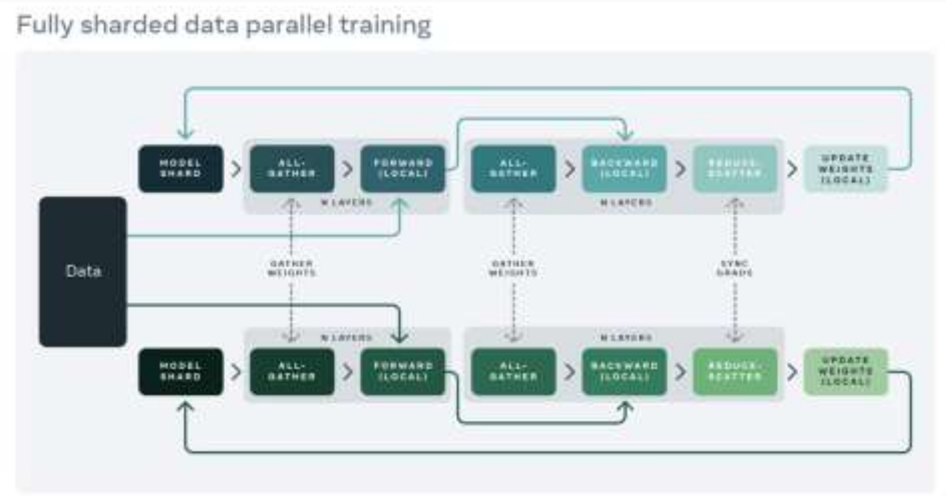


Given reduced gradients from other devices,

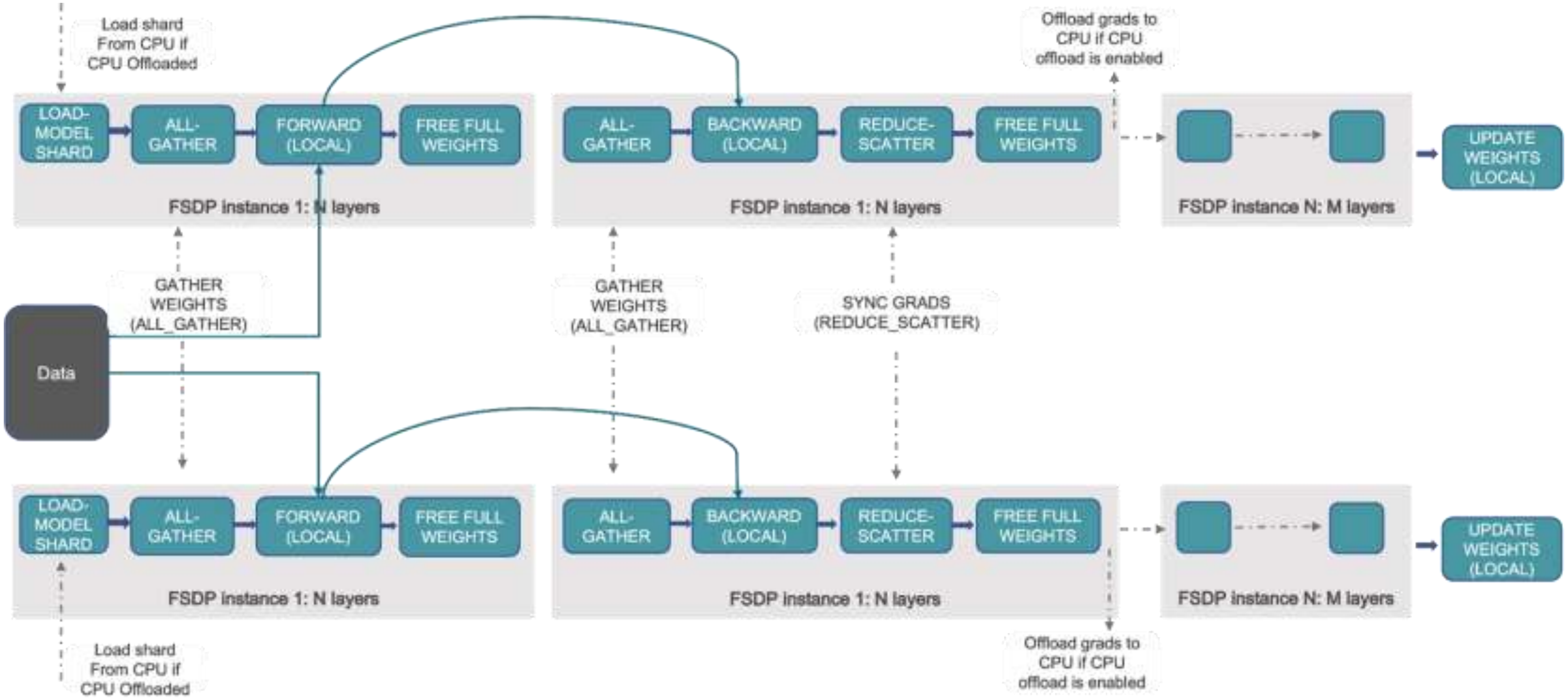
each device updates its local weight.

2.4. Comparison to DDP

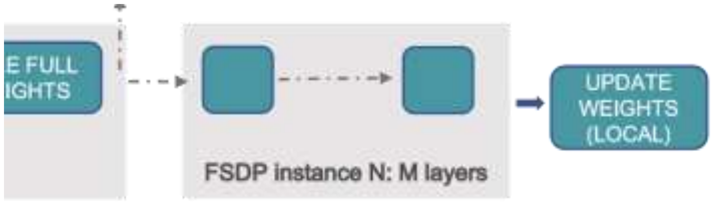


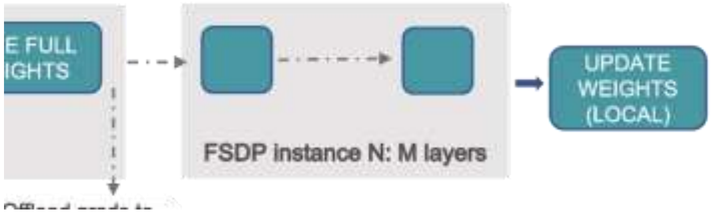


2.4. Full Architecture of FSDP



2.4. Full Architecture of FSDP



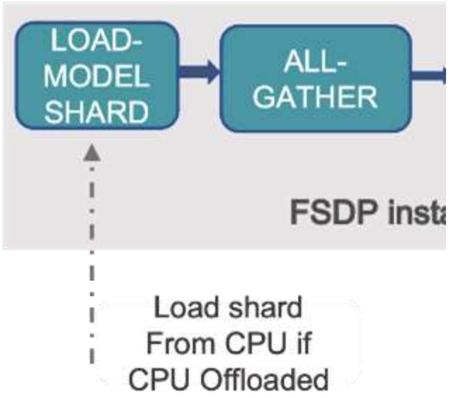


Although previous example has 1:N layers per FSDP instance,

FSDP can also map N:M layers.

In this case, some instances contain redundant layers, (= waste of memory) but are cooperating in communication (= share load across instances).

2.4. Full Architecture of FSDP



FSDP also supports CPU-offload (Zero-Offload)

for cases where even sharded parameters are too large to fit in each devices.

3. How to use FSDP

1. FSDP produces identical results as standard distributed data parallel (DDP) training

2. and is available in an easy-to-use interface
   that’s a drop-in replacement for PyTorch’s `DistributedDataParallel` module.

3. How to use FSDP

```python
def setup(rank, world_size):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'

    # initialize the process group
    dist.init_process_group("nccl", rank=rank, world_size=world_size)

def cleanup():
    dist.destroy_process_group()
```

Helper functions

1. to initialize the processes for distributed training
2. and clean up.

3. How to use FSDP

```
def train(args, model, rank, world_size, train_loader, optimizer, epoch, sampler=None):
    model.train()
    ddp_loss = torch.zeros(2).to(rank)
    if sampler:
        sampler.set_epoch(epoch)
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(rank), target.to(rank)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target, reduction='sum')
        loss.backward()
        optimizer.step()
        ddp_loss[0] += loss.item()
        ddp_loss[1] += len(data)

    dist.all_reduce(ddp_loss, op=dist.ReduceOp.SUM)
    if rank == 0:
        print('Train Epoch: {} \tLoss: {:.6f}'.format(epoch, ddp_loss[0] / ddp_loss[1]))
```

Function for training is almost the same as normal PyTorch,

except:
loss is packaged and distributed via All-Reduce.

(FSDP internally decomposes and re-schedules All-Reduce)

3. How to use FSDP

```
def fsdp_main(rank, world_size, args):
    setup(rank, world_size)

    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])

    dataset1 = datasets.MNIST('../data', train=True, download=True,
                        transform=transform)
    dataset2 = datasets.MNIST('../data', train=False,
                        transform=transform)

    sampler1 = DistributedSampler(dataset1, rank=rank, num_replicas=world_size, shuf
fle=True)
    sampler2 = DistributedSampler(dataset2, rank=rank, num_replicas=world_size)
```

transformers is TorchVision Transformer model.

DistributedSapmler uniformly distributes data across all processes.

3. How to use FSDP

```
    model = Net().to(rank)

    model = FSDP(model)

    optimizer = optim.Adadelta(model.parameters(), lr=args.lr)

    scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
    init_start_event.record()
    for epoch in range(1, args.epochs + 1):
        train(args, model, rank, world_size, train_loader, optimizer, epoch, sampler
=sampler1)
        test(model, rank, world_size, test_loader)
        scheduler.step()

    init_end_event.record()
```

Once wrapped the model with FSDP,

now model can start training simply.