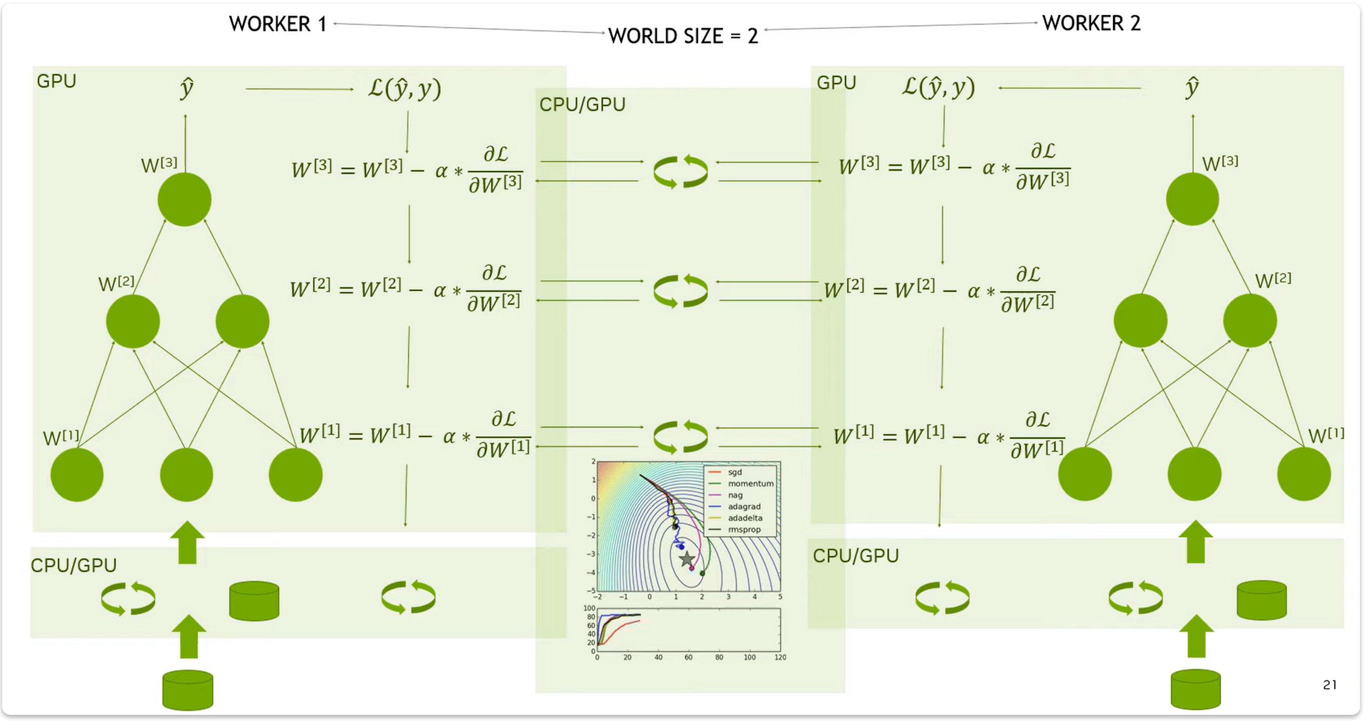
Tags: #machine-learning

_machine learning

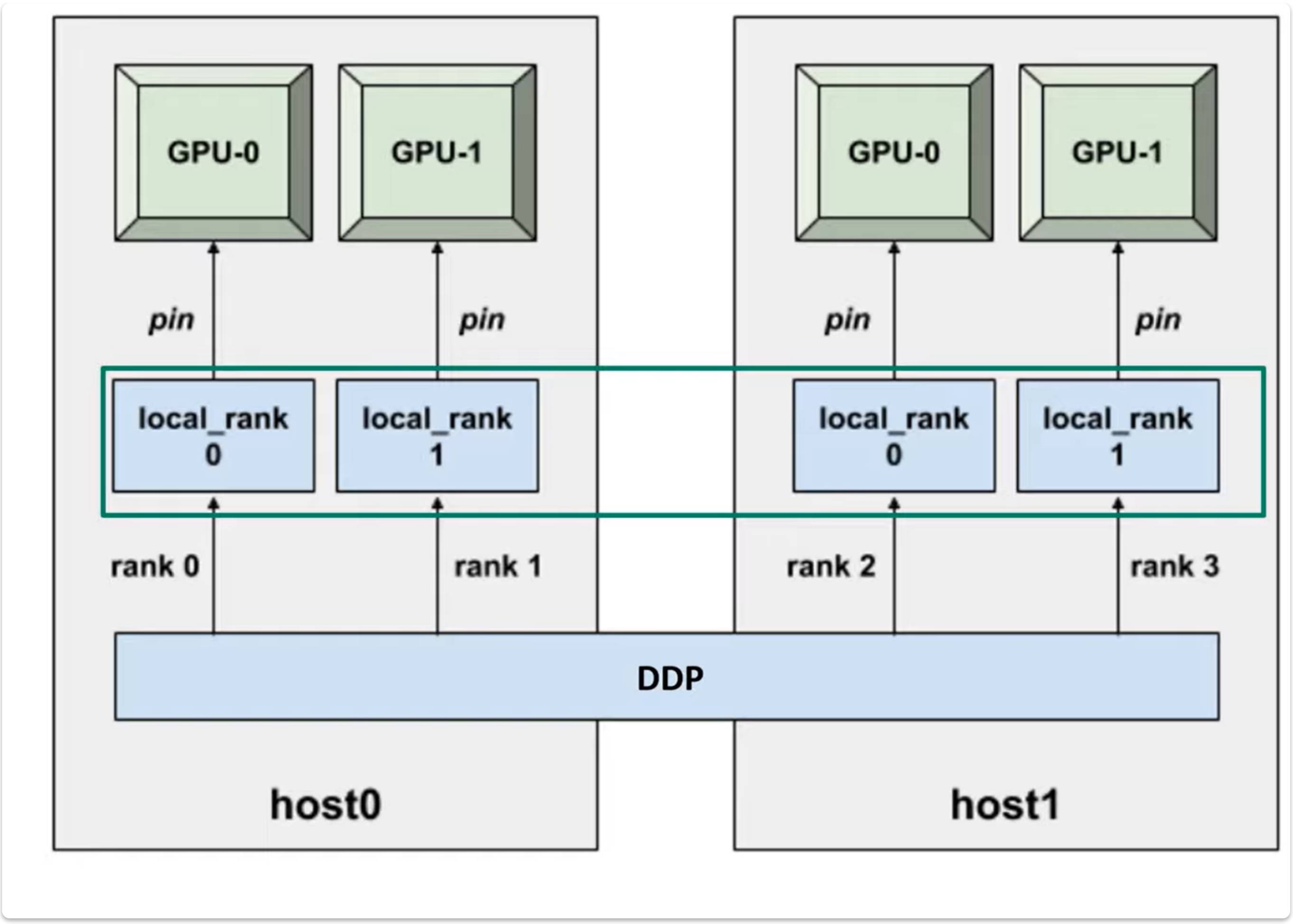
Data Parallelization in PyTorch

Terminology

Data Parallelism Using PyTorch DDP | NVAITC Webinar


WORKER 1
WORLD SIZE = 2
WORKER 2
GPU
$\hat{y}$
$\mathcal{L}(\hat{y}, y)$
CPU/GPU
$W^{[3]} = W^{[3]} - \alpha * \frac{\partial \mathcal{L}}{\partial W^{[3]}}$
$W^{[2]} = W^{[2]} - \alpha * \frac{\partial \mathcal{L}}{\partial W^{[2]}}$
$W^{[1]} = W^{[1]} - \alpha * \frac{\partial \mathcal{L}}{\partial W^{[1]}}$
W[3]
W[2]
W[1]
sgd
momentum
nag
adagrad
adadelta
rmsprop
21

Each process is known as a *worker*. In the context of multi-GPU training, each GPU is a worker. The *world size* is the number of workers that are available (which is also the number of GPUs that are available).

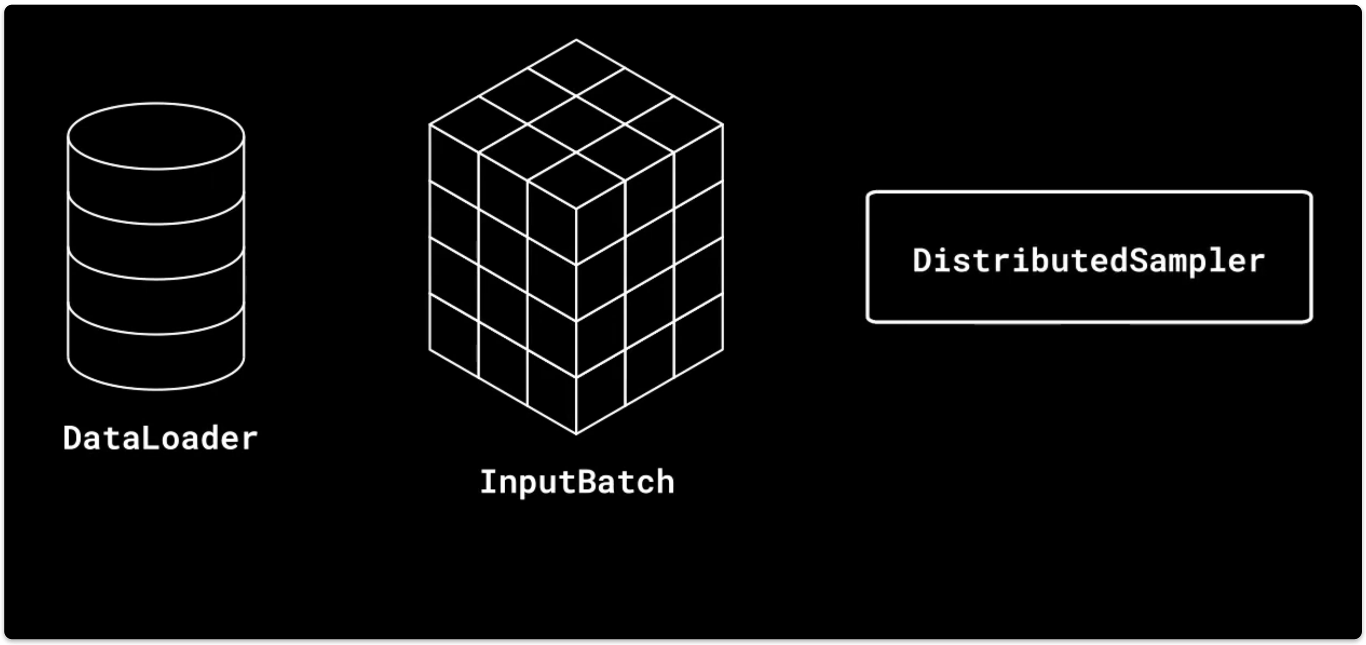


In the image above, there are two *hosts*. A host is a computer that has associated GPU instances attached. In the example above, there are 4 GPUs available across two hosts. The local rank is the unique identifier within the host. The global rank is the unique identifier pertaining to each worker across all hosts.

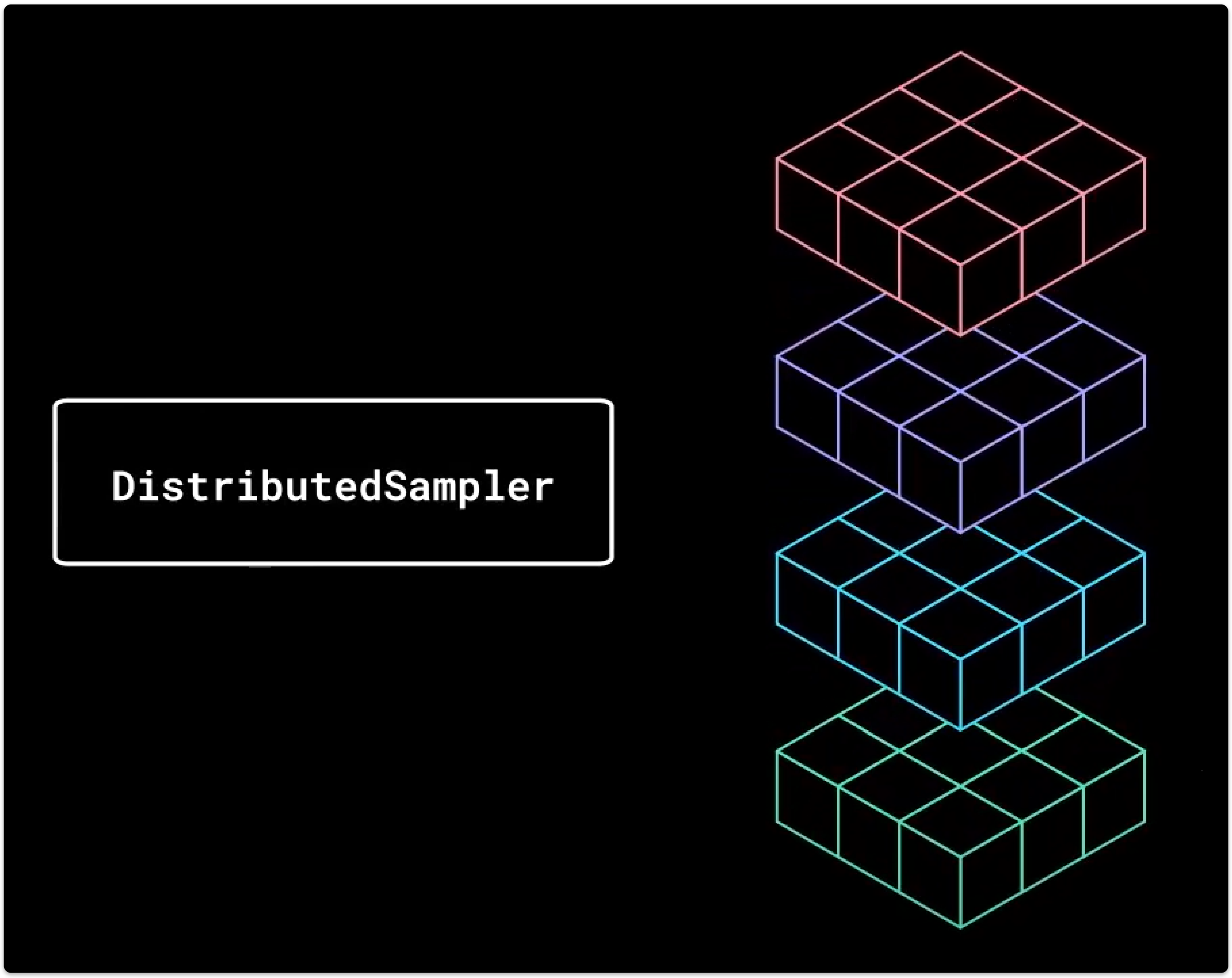
DDP Explained (What is Distributed Data Parallel (DDP)?)

According to Training on multiple GPUs and multi-node training with PyTorch DDP at 3:40, the `nn.DataParallel` module is not recommended by PyTorch. It copies the model into each GPU and then takes aggregates the gradients using some sort of average or aggregation mechanism. This is not how DDP works, and this is not nearly as efficient.

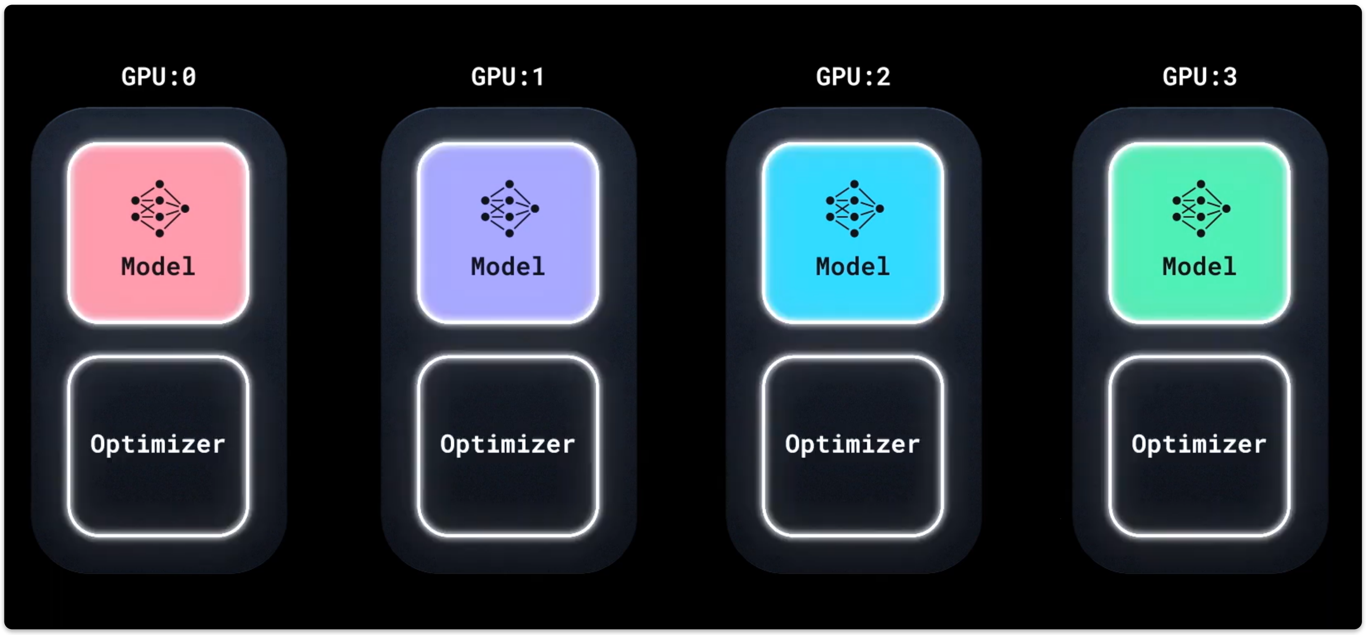
Distributed Data Parallel (DDP) is a way to efficiently parallelize computation. The way that it works is instead of the entire data batch being computed on one worker, the input batch is passed into a `DistributedSampler` object.



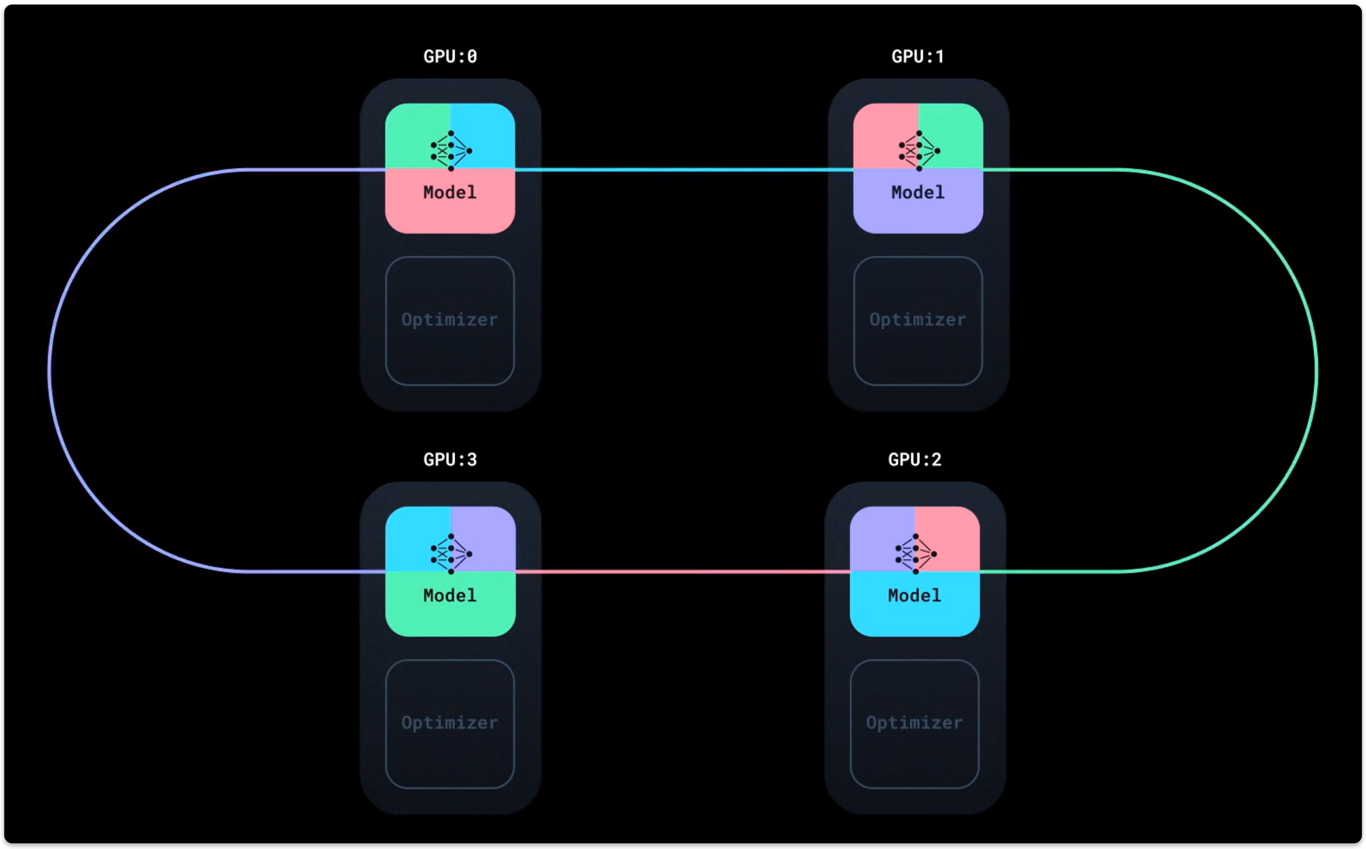
The `DistributedSampler` object partitions the data onto the multiple GPUs such that these computations can be computed in parallel.



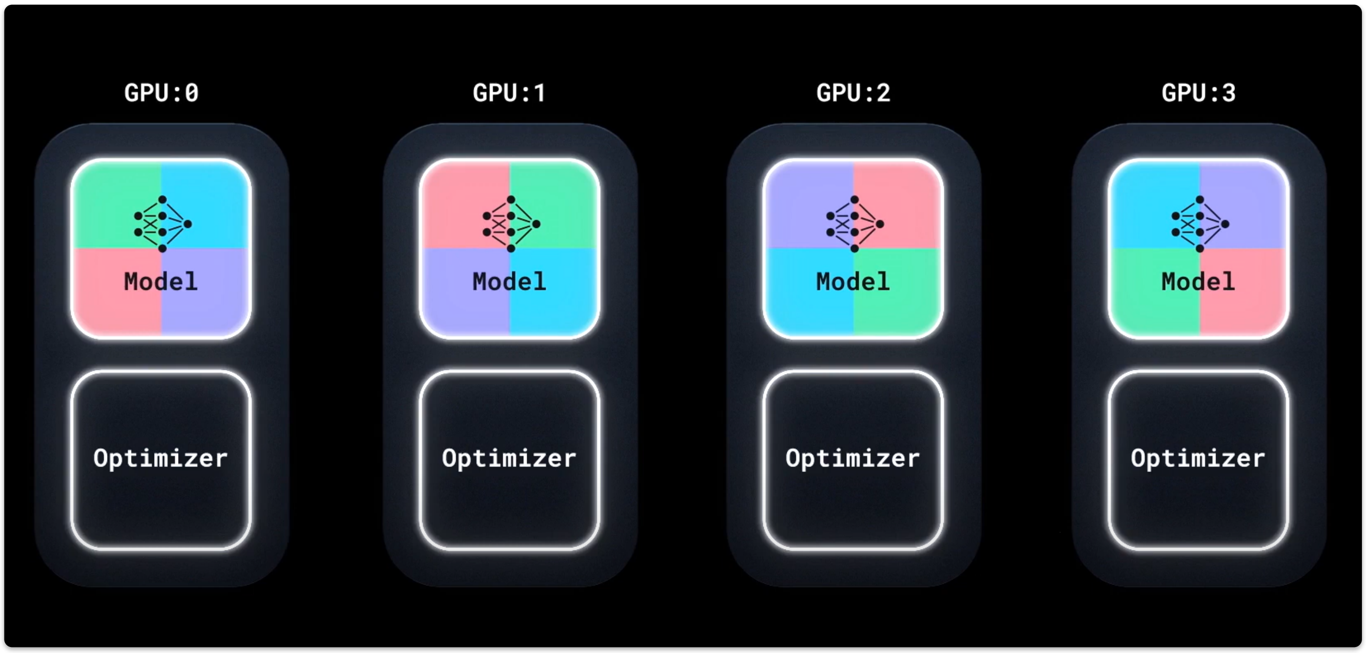
Suppose we have 4 workers. After computing the losses for each layer, updating the distributed models at this stage would result in unsynchronized models.



To ensure that the models stay synchronized, DDP initiates a synchronization step using the bucketed "all-reduce" algorithm. This algorithm overlaps gradient computation with communication. Communication along the ring does not wait for all of the models to complete the computation of the individual model gradients, which ensures that the GPUs are never idle.



Now, all of the distributed models are in sync, and so we can perform an optimization update.



DDP In-Code (Multi-GPU training with DDP)

Training Code Updates

The training code and inference code need to be separate in order for the multi-threading functionality to be enabled. Then, we can spawn multiple training threads, one for each worker (or GPU).

```
if __name__ == "__main__":
    world_size = torch.cuda.device_count()
    mp.spawn(training_thread, args=(world_size, params), nprocs=world_size)
```

What enables the multi-threading functionality at the PyTorch model level are the following lines:

```
from torch.nn.parallel import DistributedDataParallel as DDP

if gpu_id is not None:
    model = model.to(gpu_id)
    model = DDP(model, device_ids=[gpu_id])
```

In the above, the `gpu_id` is the ID of the GPU that we will be sending the model to. The model is wrapped in the `DistributedDataParallel` class . Therefore, when you want to save the model, you have to access the actual model `module`:

```
torch.save(model.module.state_dict(), model_path)
```

The `DistributedDataParallel` module will have no affect on the model unless a `DistributedSampler` is used and the multi-threading settings are enabled.

To initialize a `DataLoader` with a `DistributedSampler`, consider the following:

```
train_loader = DataLoader(
    train,
    batch_size=params["batch_size"],
    pin_memory=True,
    shuffle=False,
    sampler=DistributedSampler(train),
)
```

To initialize a training thread, ensure the following code is included:

```
def ddp_setup(rank: int, world_size: int) -> None:
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "12354"
    init_process_group(backend="nccl", rank=rank, world_size=world_size)
```

Training Code Update Summary

The code below shows the main function, the general code for the training thread, and the general code for the model training:

```
def train_model(
    ...
):
    """
    Init arguments
    """
    # Send model to GPU
    if gpu_id is not None:
        model = model.to(gpu_id)
        model = DDP(model, device_ids=[gpu_id])

    train_batcher = iter(train_loader)
    for i in range(n_updates):
        # Forward pass
        xb, yb = next(train_batcher)
        _, loss = model(xb, yb)

        if (
            checkpoint_iter is not None
            and i % checkpoint_iter == 0
            and (gpu_id is None or gpu_id == 0)
        ):
           """
           Checkpoint the training loop
           """

        # Gradient updates
        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()

def training_thread(rank: int, world_size: int, params: dict):
    ddp_setup(rank, world_size)
```

```python
    """
    Extract the data
    """

    train_loader = DataLoader(
        train,
        batch_size=params["batch_size"],
        pin_memory=True,
        shuffle=False,
        sampler=DistributedSampler(train),
    )

    """
    Initialize the model
    """

    train_model(
        model,
        train_loader,
        valid_loader,
        torch.optim.Adam(model.parameters(), lr=params["learning_rate"]),
        rank,
        n_updates=params["n_updates"],
        checkpoint_iter=params["checkpoint_iter"],
    )

    # Required for cleanup
    destroy_process_group()


def ddp_setup(rank: int, world_size: int) -> None:
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "12354"
    init_process_group(backend="nccl", rank=rank, world_size=world_size)


if __name__ == "__main__":
    world_size = torch.cuda.device_count()
    mp.spawn(training_thread, args=(world_size, params), nprocs=world_size)
```