Using ‘PyTorch Distributed’

Fabian Berressem

AI DevTech Engineer

December 10, 2024

Agenda

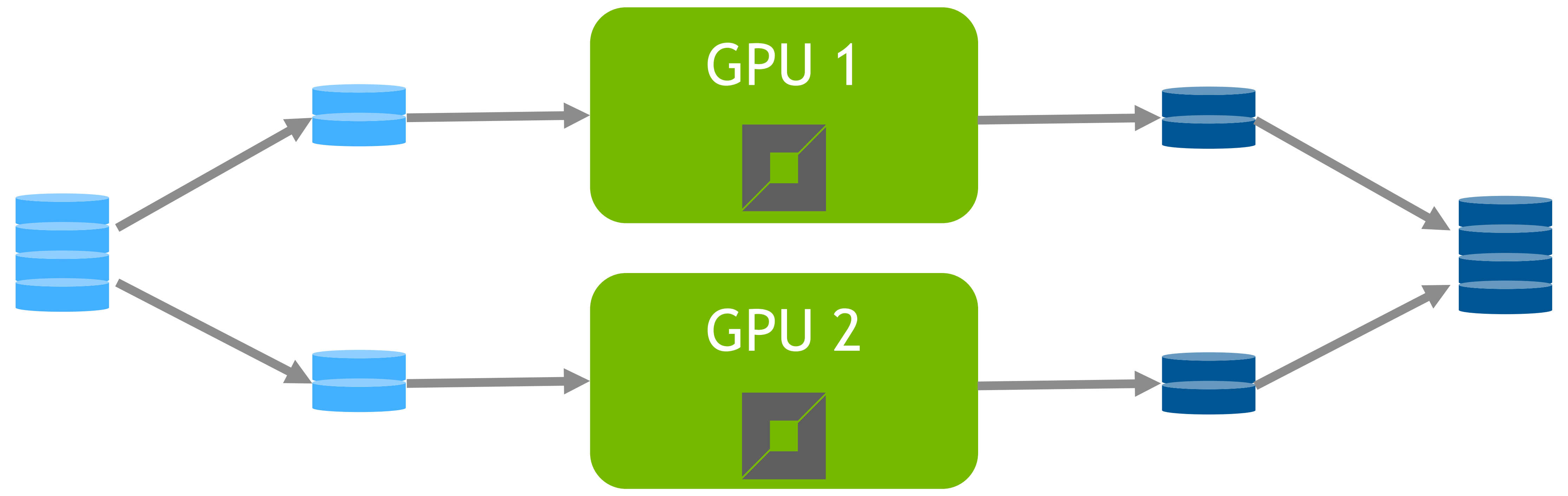
- Overview of Parallel Training
- Data Parallelism with PyTorch Distributed
- Data Parallelism with torchrun
- Model Parallelism with PyTorch Distributed
- [FullyShardedDataParallel with PyTorch Distributed]

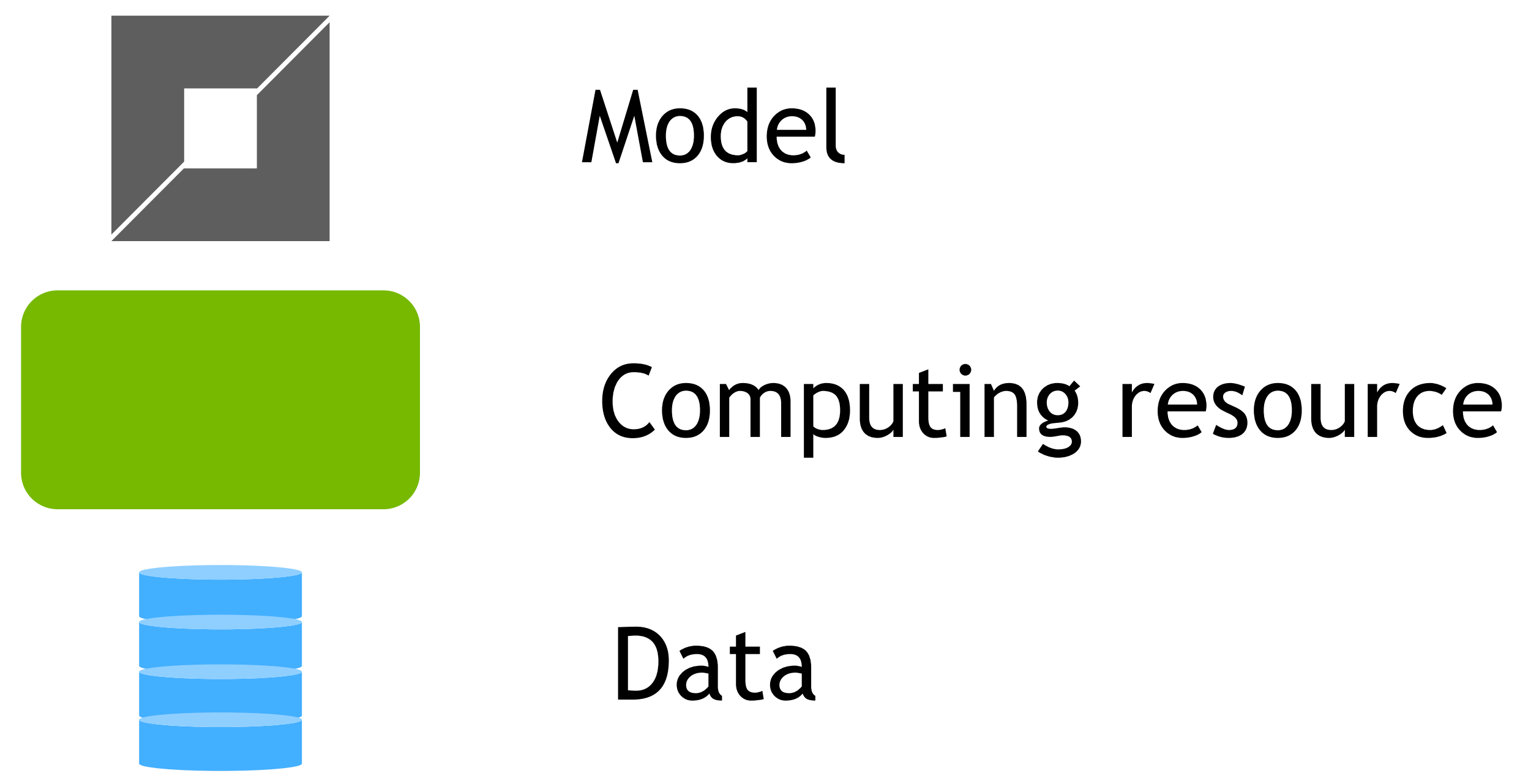
Overview of Parallel Training

Overview of Parallel Training

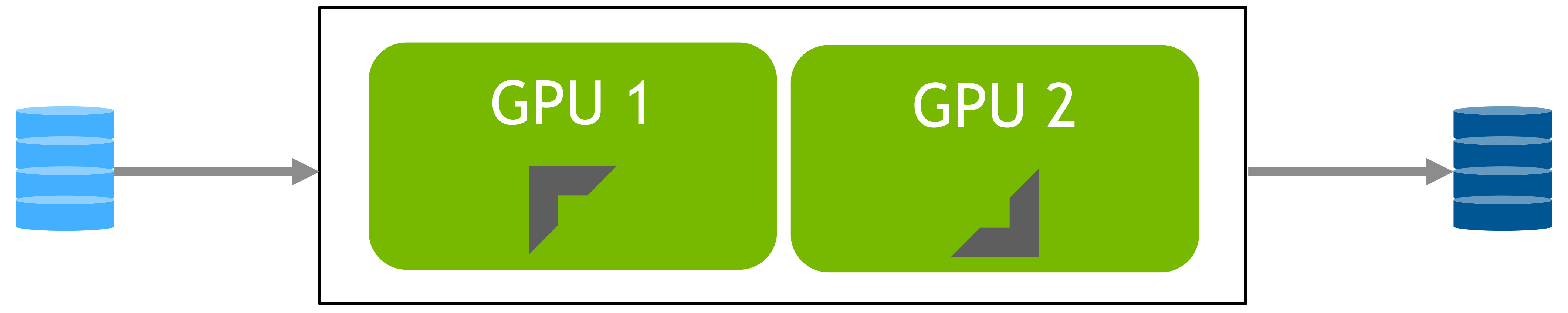
When do we need it? What types are there?

- Training is too slow
  - Use data parallelism, i.e. replicated model on GPUs applied to different data
  - SPMD (Single Program Multiple Data)
  - As model is replicated, has to fit on single GPU





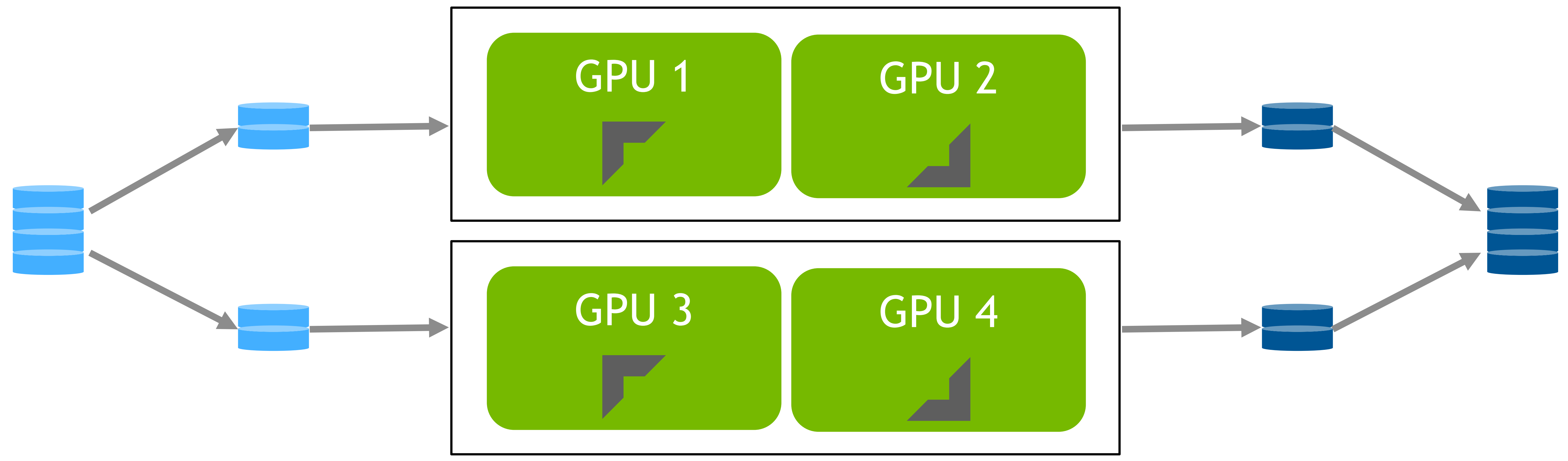
- Model is too big for single GPU
  - Use model (/sharded data) parallelism, i.e. model split between GPUs

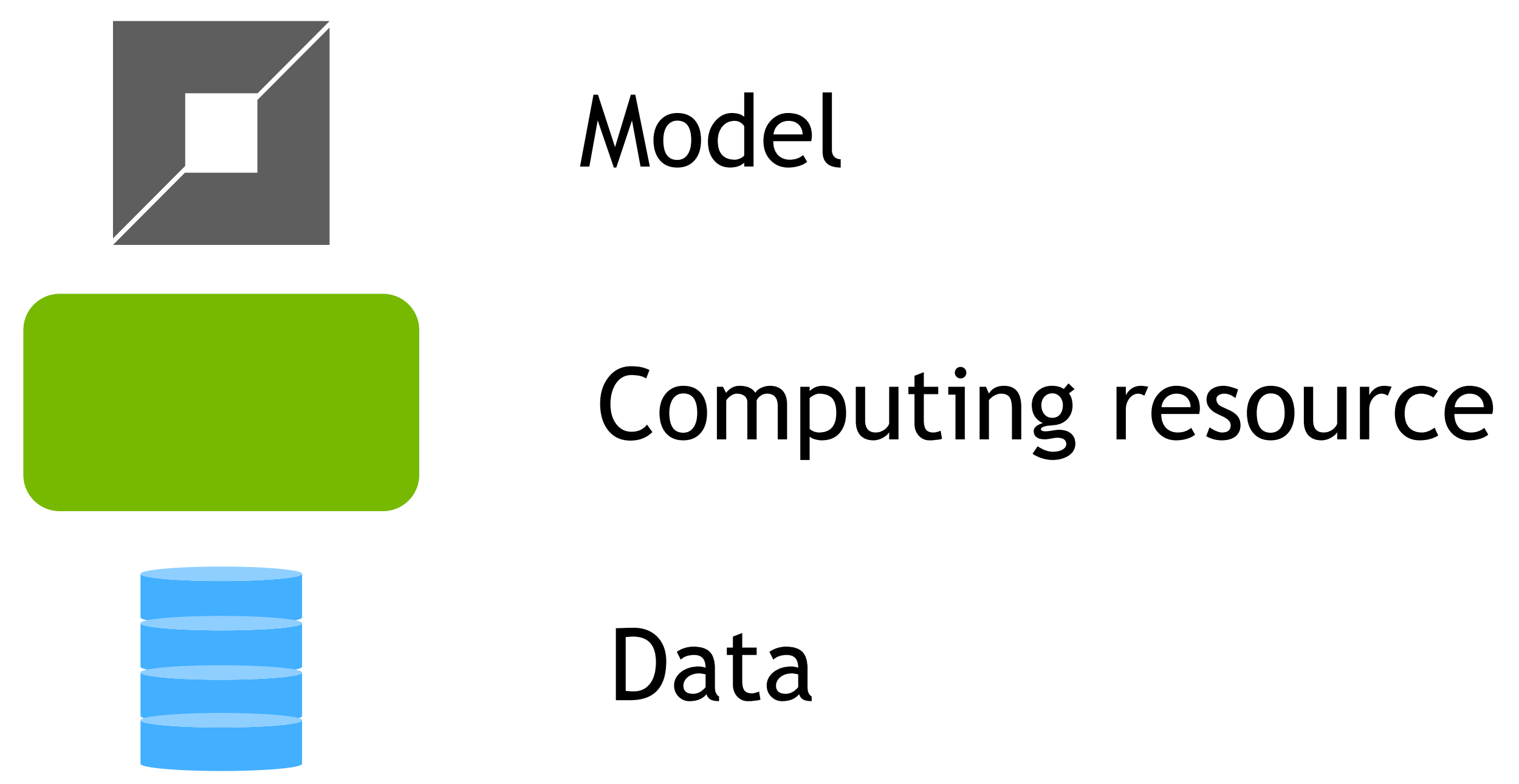


Overview of Parallel Training

When do we need it? What types are there?

- Training is too slow and model is too big for single GPU
  - Use data and task parallelism, i.e. split models between GPUs and apply to different data
  - **Beware**, very intensive task, especially **communications**
  - Needs very good interconnect, in particular when using multiple Nodes, e.g. NVLink, NVSwitch, Infiniband

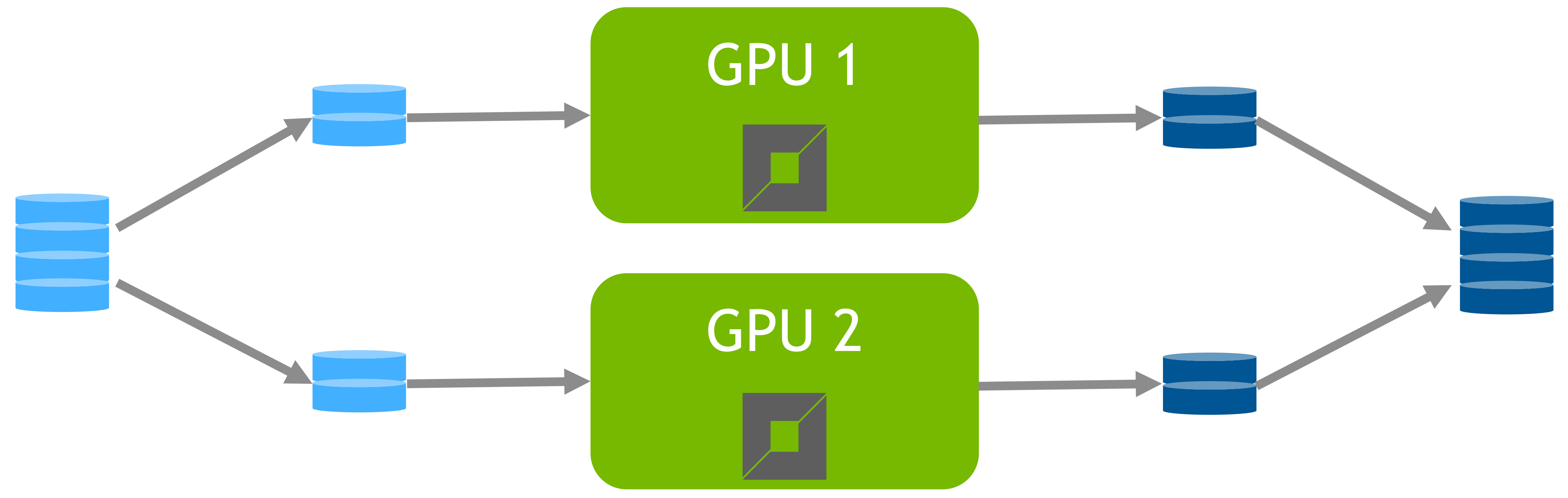




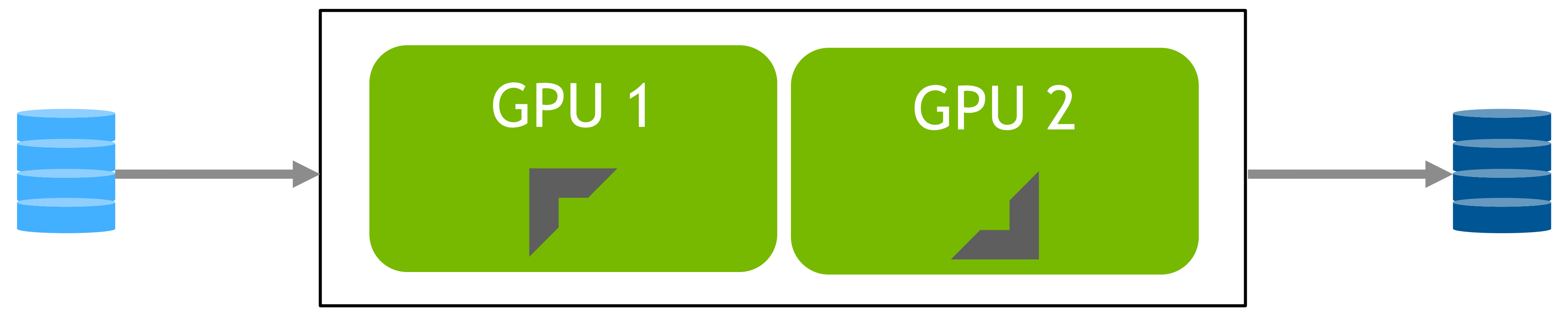
Overview of Parallel Training

PyTorch Distributed

- PyTorch Distributed offers different parallelization schemes:
  - Data parallelism via:
    - DistributedDataParallel (ddp) for single-node runs
    - torchrun launcher script for multi-node runs

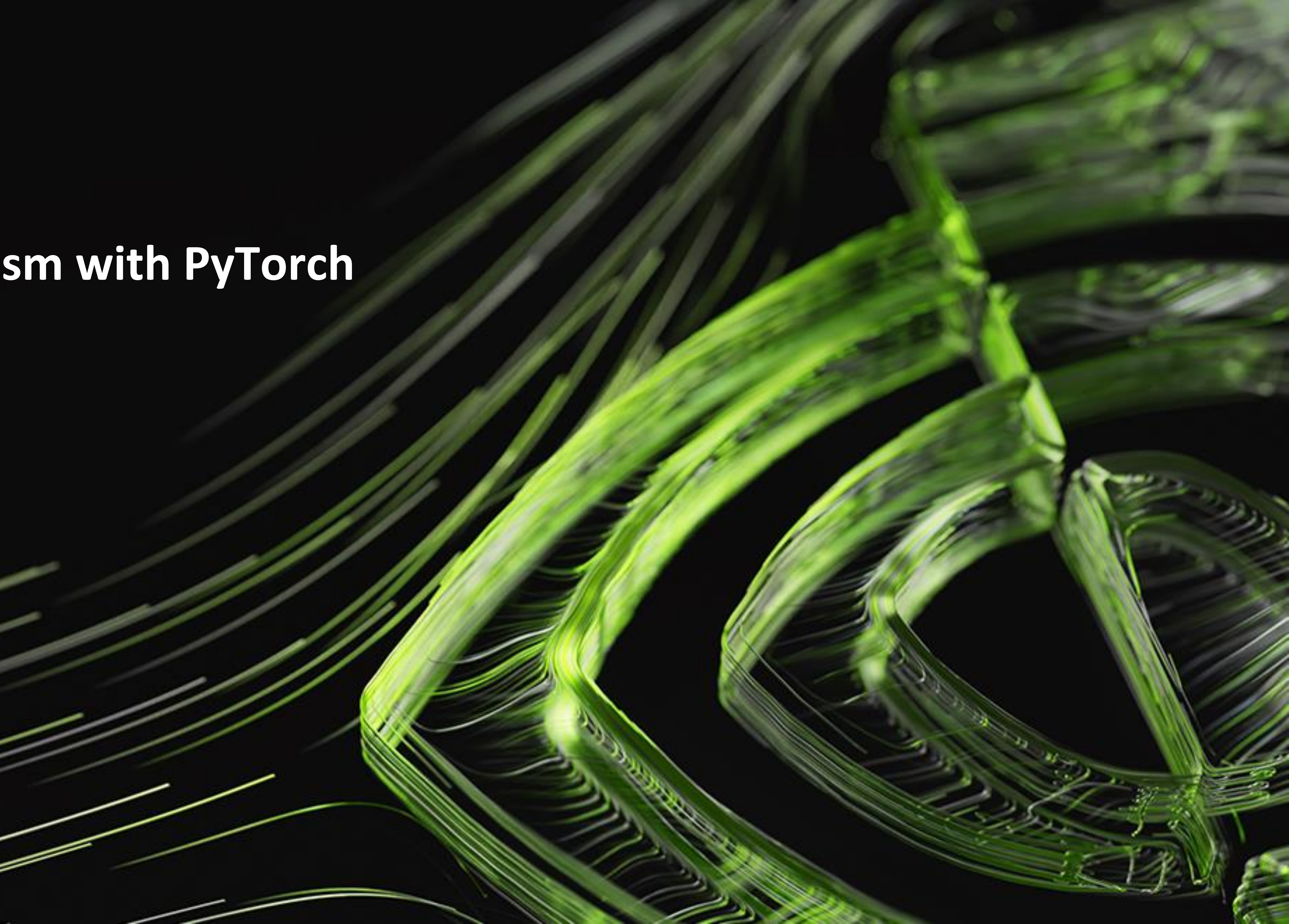


  - Model parallelism via:
    - FullyShardedDataParallel (FSDP)
    - In case of scaling limitations with FSDP:
      - Tensor Parallel (TP)
      - Pipeline Parallel (PP)



- Further communications APIs and sharding primitives available

Data Parallelism with PyTorch Distributed



Data Parallelism with PyTorch Distributed

Single GPU Base Script

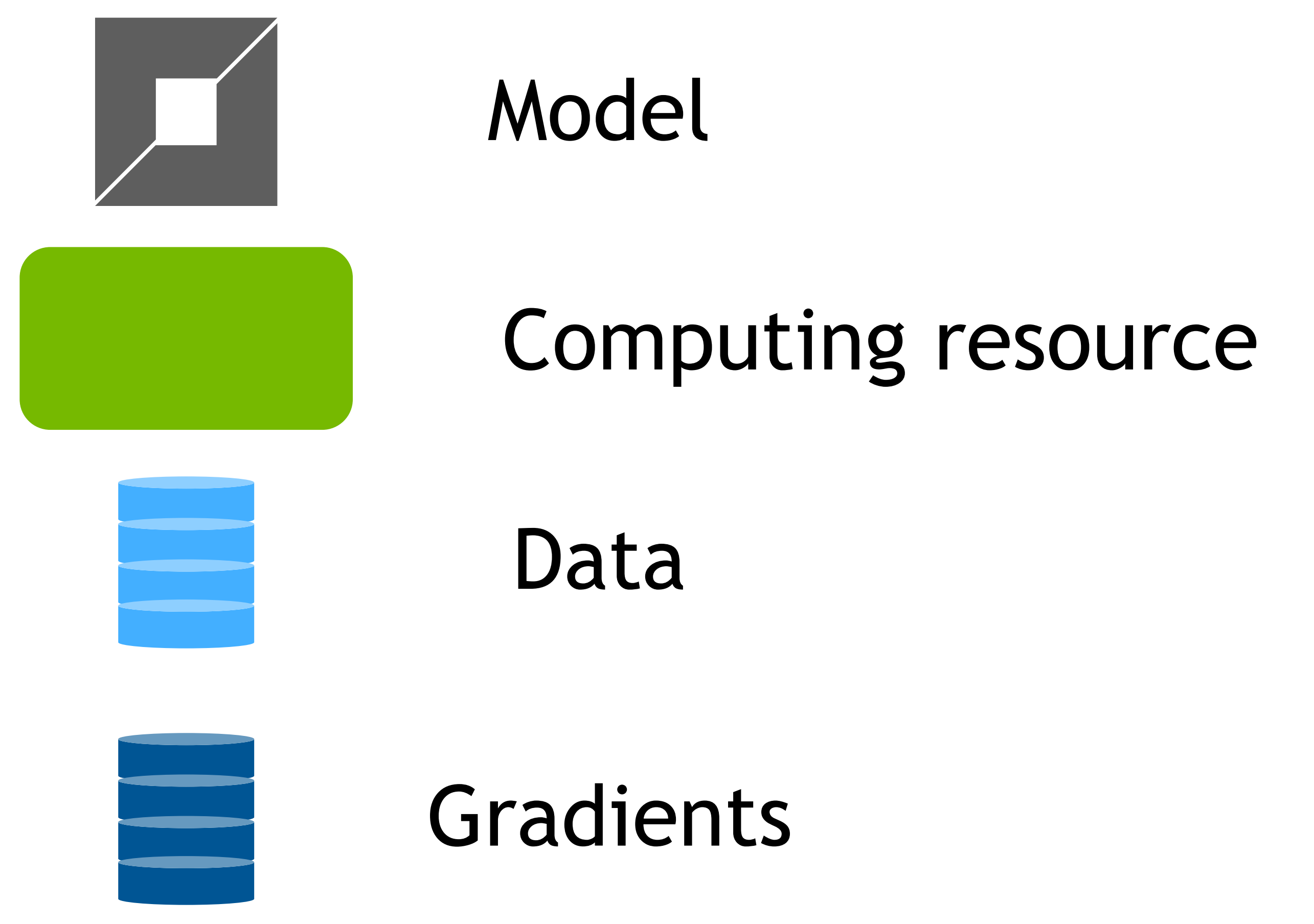
```python
import torch
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from datautils import MyTrainDataset


class Trainer:
    def __init__(
        self,
        model: torch.nn.Module,
        train_data: DataLoader,
        optimizer: torch.optim.Optimizer,
        gpu_id: int,
        save_every: int,
    ) -> None:
        self.gpu_id = gpu_id
        self.model = model.to(gpu_id)
        self.train_data = train_data
        self.optimizer = optimizer
        self.save_every = save_every

    def _run_batch(self, source, targets):
        self.optimizer.zero_grad()
        output = self.model(source)
        loss = F.cross_entropy(output, targets)
        loss.backward()
        self.optimizer.step()

    def _run_epoch(self, epoch):
        b_sz = len(next(iter(self.train_data))[0])
        print(f"[GPU{self.gpu_id}] Epoch {epoch} | Batchsize: {b_sz} | Steps: {len(self.train_data)}")
        for source, targets in self.train_data:
            source = source.to(self.gpu_id)
            targets = targets.to(self.gpu_id)
            self._run_batch(source, targets)

    def _save_checkpoint(self, epoch):
        ckp = self.model.state_dict()
        PATH = "checkpoint.pt"
        torch.save(ckp, PATH)
        print(f"Epoch {epoch} | Training checkpoint saved at {PATH}")

    def train(self, max_epochs: int):
        for epoch in range(max_epochs):
            self._run_epoch(epoch)
            if epoch % self.save_every == 0:
                self._save_checkpoint(epoch)
```

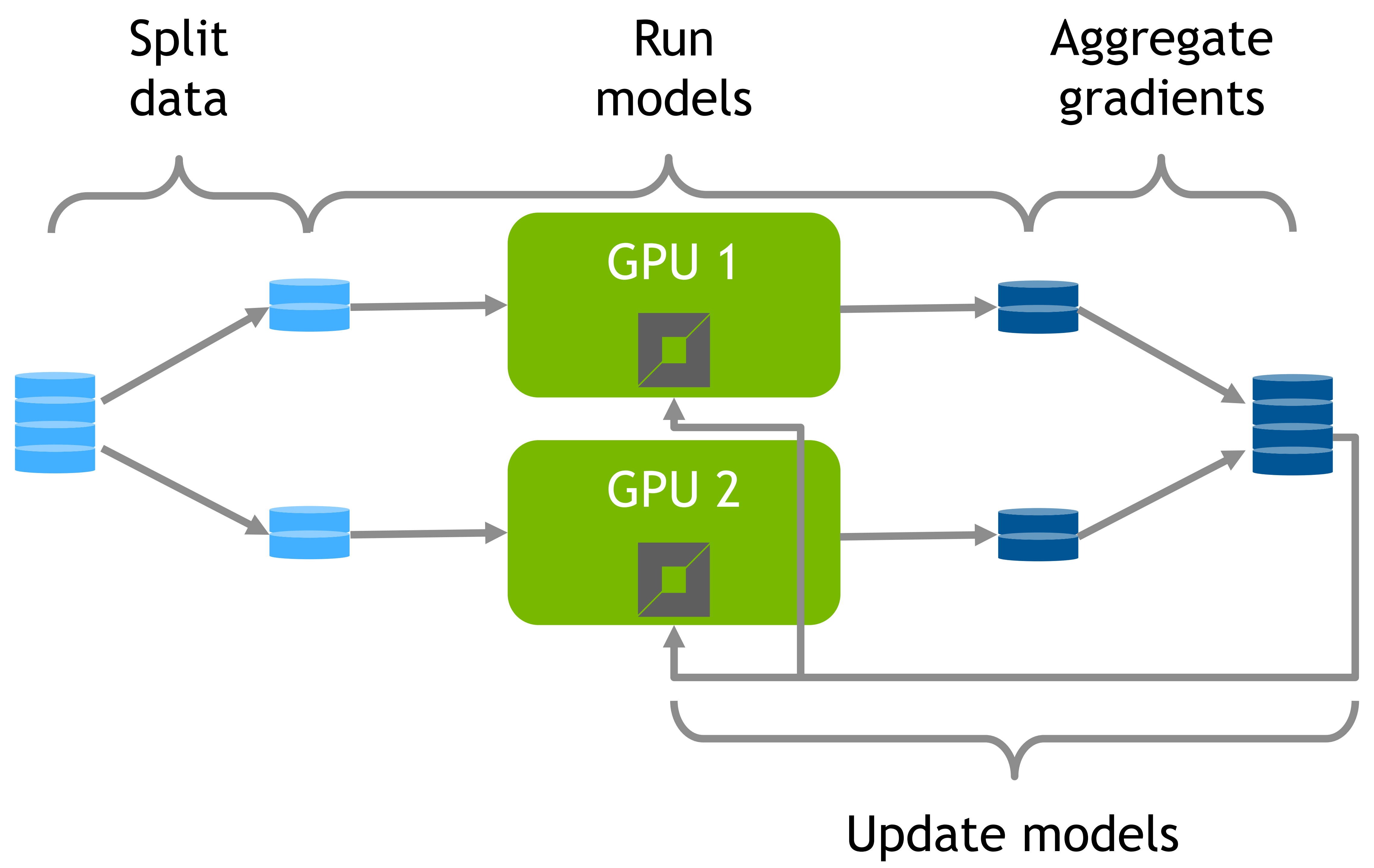
```python
def load_train_objs():
    train_set = MyTrainDataset(2048)  # load your dataset
    model = torch.nn.Linear(20, 1)  # load your model
    optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
    return train_set, model, optimizer


def prepare_dataloader(dataset: Dataset, batch_size: int):
    return DataLoader(
        dataset,
        batch_size=batch_size,
        pin_memory=True,
        shuffle=True
    )


def main(device, total_epochs, save_every, batch_size):
    dataset, model, optimizer = load_train_objs()
    train_data = prepare_dataloader(dataset, batch_size)
    trainer = Trainer(model, train_data, optimizer, device, save_every)
    trainer.train(total_epochs)


if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description='simple distributed training job')
    parser.add_argument('total_epochs', type=int, help='Total epochs to train the model')
    parser.add_argument('save_every', type=int, help='How often to save a snapshot')
    parser.add_argument('--batch_size', default=32, type=int, help='Input batch size on each device (default: 32)')
    args = parser.parse_args()

    device = 0  # shorthand for cuda:0
    main(device, args.total_epochs, args.save_every, args.batch_size)
```

Data Parallelism with PyTorch Distributed

Schematics of Process

- Training process in data parallelism:
  1. Data is split and passed to different GPUs
  2. Data is run through identical models on different GPUs
  3. GPUs calculate gradients based on their inputs (with exact optimizers)
  4. Gradients are aggregated asynchronously to keep GPUs busy
  5. Models are updated using gradients
  6. Repeat
- Needs communication during splitting of data and accumulation of gradients

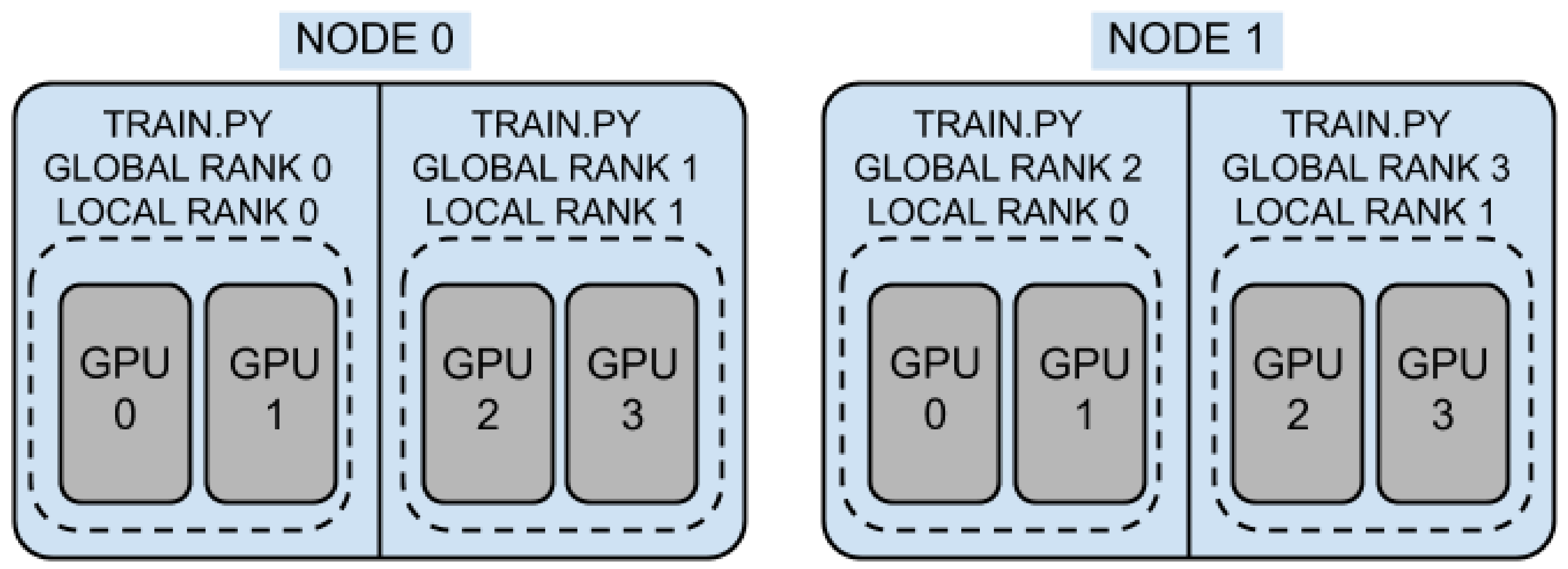




Data Parallelism with PyTorch Distributed

world_size and rank

- There's two terms needed to distribute the load to the GPUs for PyTorch:
- world_size:
  - (Local) World size = Number of processes (running on single node)
  - Is provided by user on call of mp.spawn
  - Necessary to know the number of running processes, e.g. in setup, destruction, etc.
- rank:
  - Local/Global rank = application process ID on single node / for all of the application processes
  - Will be created by mp.spawn
  - Necessary to know own id, e.g. in setup, checkpointing, data movements



Data Parallelism with PyTorch Distributed

Multi GPU Script

```python
import torch
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from datautils import MyTrainDataset

import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process_group
import os

class Trainer:
    def __init__(
        self,
        model: torch.nn.Module,
        train_data: DataLoader,
        optimizer: torch.optim.Optimizer,
        gpu_id: int,
        save_every: int,
    ) -> None:
        self.gpu_id = gpu_id
            self.model = model.to(gpu)
            self.train_data = train_data
        self.optimizer = optimizer
        self.save_every = save_every
            self.model = DDP(self.model, device_ids=[gpu_id])

    def _run_batch(self, source, targets):
        self.optimizer.zero_grad()
        output = self.model(source)
        loss = F.cross_entropy(output, targets)
        loss.backward()
        self.optimizer.step()

    def _run_epoch(self, epoch):
        b_sz = len(next(iter(self.train_data))[0])
        print(f"[GPU{self.gpu_id}] Epoch {epoch} | Batchsize: {b_sz} | Steps: {len(self.train_data)}")
        self.train_data.sampler.set_epoch(epoch)
        for source, targets in self.train_data:
            source = source.to(self.gpu_id)
            targets = targets.to(self.gpu_id)
            self._run_batch(source, targets)

    def _save_checkpoint(self, epoch):
        ckp = self.model.module.state_dict()
        PATH = "checkpoint.pt"
        torch.save(ckp, PATH)
        print(f"Epoch {epoch} | Training checkpoint saved at {PATH}")

    def train(self, max_epochs: int):
        for epoch in range(max_epochs):
            self._run_epoch(epoch)
            if self.gpu_id == 0 and epoch % self.save_every == 0:
                self._save_checkpoint(epoch)
```

```python
def ddp_setup(rank, world_size):
    """
    Args:
        rank: Unique identifier of each process
        world_size: Total number of processes
    """
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "12355"
    torch.cuda.set_device(rank)
    init_process_group(backend="nccl", rank=rank, world_size=world_size)


def load_train_objs():
    train_set = MyTrainDataset(2048)  # load your dataset
    model = torch.nn.Linear(20, 1)  # load your model
    optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
    return train_set, model, optimizer



def prepare_dataloader(dataset: Dataset, batch_size: int):
    return DataLoader(
        dataset,
        batch_size=batch_size,
        pin_memory=True,
        shuffle=False,
        sampler=DistributedSampler(dataset)
    )

def main(rank: int, world_size: int, save_every: int, total_epochs: int, batch_size: int):
    ddp_setup(rank, world_size)
    dataset, model, optimizer = load_train_objs()
    train_data = prepare_dataloader(dataset, batch_size)
    trainer = Trainer(model, train_data, optimizer, rank, save_every)
    trainer.train(total_epochs)
    destroy_process_group()


if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description='simple distributed training job')
    parser.add_argument('total_epochs', type=int, help='Total epochs to train the model')
    parser.add_argument('save_every', type=int, help='How often to save a snapshot')
    parser.add_argument('--batch_size', default=32, type=int, help='Input batch size on each device (default: 32)')
    args = parser.parse_args()

    world_size = torch.cuda.device_count()
    mp.spawn(main, args=(world_size, args.save_every, args.total_epochs, args.batch_size), nprocs=world_size)
```

Data Parallelism with PyTorch Distributed

Some Notes

- Choose NCCL (NVIDIA Collective Communications Library) as backend for GPUs (instead of gloo), especially for NVIDIA GPUs
- Different process mappings possible but typically one process per GPU is good balance between I/O and computation
- Always use main guard due to multi-threaded spawn
- Parameters specified in DataLoader are per process
- Remember to weigh different methods of data parallelism:
  - Optimize (model) latency: True data parallelism (ddp)
  - Optimize (model) throughput: Multi-GPU hyperparameter search
  - Mixtures

Data Parallelism with torchrun

Data Parallelism with torchrun

Multi GPU Script with torchrun

```
import torch
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from datautils import MyTrainDataset
import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process_group
import os

class Trainer:
    def __init__(
        self,
        model: torch.nn.Module,
        train_data: DataLoader,
        optimizer: torch.optim.Optimizer,
        save_every: int,
        snapshot_path: str,
    ) -> None:
        self.gpu_id = int(os.environ["LOCAL_RANK"])
            self.global_rank = int(os.environ["RANK"])
        self.model = model.to(self.gpu_id)
        self.train_data = train_data
        self.optimizer = optimizer
        self.save_every = save_every
        self.epochs_run = 0
        self.snapshot_path = snapshot_path
        if os.path.exists(snapshot_path):
            print("Loading snapshot")
            self._load_snapshot(snapshot_path)

        self.model = DDP(self.model, device_ids=[self.gpu_id])

    def _load_snapshot(self, snapshot_path):
        loc = f"cuda:{self.gpu_id}"
        snapshot = torch.load(snapshot_path, map_location=loc)
        self.model.load_state_dict(snapshot["MODEL_STATE"])
        self.epochs_run = snapshot["EPOCHS_RUN"]
        print(f"Resuming training from snapshot at Epoch {self.epochs_run}")

    def _run_batch(self, source, targets):
        self.optimizer.zero_grad()
        output = self.model(source)
        loss = F.cross_entropy(output, targets)
        loss.backward()
        self.optimizer.step()

    def _run_epoch(self, epoch):
        b_sz = len(next(iter(self.train_data))[0])
        print(f"[GPU{self.global_rank}] Epoch {epoch} | Batchsize: {b_sz} | Steps: {len(self.train_data)}")
        self.train_data.sampler.set_epoch(epoch)
        for source, targets in self.train_data:
            source = source.to(self.gpu_id)
            targets = targets.to(self.gpu_id)
            self._run_batch(source, targets)
```

```
# Still in Trainer class
            def _save_snapshot(self, epoch):
        snapshot = {
            "MODEL_STATE": self.model.module.state_dict(),
            "EPOCHS_RUN": epoch,
        }
        torch.save(snapshot, self.snapshot_path)
        print(f"Epoch {epoch} | Training snapshot saved at {self.snapshot_path}")

            def train(self, max_epochs: int):
        for epoch in range(self.epochs_run, max_epochs):
            self._run_epoch(epoch)
            if self.gpu_id == 0 and epoch % self.save_every == 0:
                self._save_snapshot(epoch)

def ddp_setup():
    torch.cuda.set_device(int(os.environ["LOCAL_RANK"]))
    init_process_group(backend="nccl")

def load_train_objs():
    train_set = MyTrainDataset(2048)  # load your dataset
    model = torch.nn.Linear(20, 1)  # load your model
    optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
    return train_set, model, optimizer

def prepare_dataloader(dataset: Dataset, batch_size: int):
    return DataLoader(
        dataset,
        batch_size=batch_size,
        pin_memory=True,
        shuffle=False,
        sampler=DistributedSampler(dataset)
    )

def main(save_every: int, total_epochs: int, batch_size: int, snapshot_path: str = "snapshot.pt"):
    ddp_setup()
    dataset, model, optimizer = load_train_objs()
    train_data = prepare_dataloader(dataset, batch_size)
    trainer = Trainer(model, train_data, optimizer, save_every, snapshot_path)
    trainer.train(total_epochs)
    destroy_process_group()


if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description='simple distributed training job')
    parser.add_argument('total_epochs', type=int, help='Total epochs to train the model')
    parser.add_argument('save_every', type=int, help='How often to save a snapshot')
    parser.add_argument('--batch_size', default=32, type=int, help='Input batch size on each device (default: 32)')
    args = parser.parse_args()

    main(args.save_every, args.total_epochs, args.batch_size)
```

Data Parallelism with torchrun

Single and Multi Node Call to torchrun

- Single node call:

```
$ torchrun --standalone \                  <- ”Run on single node”
           --nproc_per_node=gpu \          <- Number of procs per node (int or “gpu” for all)
           train.py …                      <- Script with parameters
```

- Multi node call:
  - Run this call with a unique node_rank on every participating node
  - Nodes can be heterogeneous

```
$ torchrun --nnodes=2 \                    <- Number of nodes
           --nproc_per_node=4 \            <- Number of procs per node (int or “gpu” for all)
           --node_rank=0 \                 <- Rank of the current node
           --rdzv_id=42 \                  <- Some shared unique ID for calls (for comm.)
           --rdzv_backend=c10d             <- PyTorch’s comm. backend
           --rdzv_endpoint=10.63.130.75:123 <- IP and port, where data is gathered
           train.py …                      <- Script with parameters
```

Data Parallelism with torchrun

torchrun and Slurm

```bash
#!/bin/bash

#SBATCH --job-name=multinode-training
#SBATCH --nodes=4
#SBATCH --ntasks=4
#SBATCH --gpus-per-task=1
#SBATCH --cpus-per-task=4

nodes=( $( scontrol show hostnames $SLURM_JOB_NODELIST ) )
nodes_array=($nodes)
head_node=${nodes_array[0]}
head_node_ip=$(srun --nodes=1 --ntasks=1 -w "$head_node" hostname --ip-address)

echo Node IP: $head_node_ip
export LOGLEVEL=INFO

srun torchrun \
        --nnodes 4 \
        --nproc_per_node 1 \
        --rdzv_id $RANDOM \
        --rdzv_backend c10d \
        --rdzv_endpoint $head_node_ip:29500 \
        train.py ...
```

Model Parallelism with PyTorch Distributed

Model Parallelism with PyTorch Distributed

Very Simple Model Parallelism

- Simple form of model parallelism can be implemented manually:
  - Not efficient
  - AFAIK, not compatible with DDP

```python
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from datautils import MyTrainDataset

class ModelParallelNN(nn.Module):
    def __init__(self, device1, device2):
        super(ModelParallelNN, self).__init__()
        self.device1 = device1
        self.device2 = device2

        # Define layers on specific GPUs
        self.linear1 = nn.Linear(20, 20).to(device1)  # First layer on GPU `device1`
        self.linear2 = nn.Linear(20, 1).to(device2)  # Second layer on GPU `device2`

    def forward(self, x):
        # Move data to device1 for linear1
        x = self.linear1(x.to(self.device1))

        # Move data to device2 for linear2
        x = self.linear2(x.to(self.device2))
        return x
```

```python
def main():
    device1 = torch.device('cuda:0')  # First GPU
    device2 = torch.device('cuda:1')  # Second GPU

    model = ModelParallelNN(device1, device2)
    dataloader = DataLoader(
        MyTrainDataset(2048),
        batch_size=32,
        pin_memory=True,
    )

    criterion = nn.MSELoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
    for epoch in range(10):
        model.train()
        for inputs, targets in dataloader:
            # Move inputs and targets to the first GPU for this process
            inputs, targets = inputs.to(device1), targets.to(device1)

            # Forward pass through the model (handling model parallelism)
            outputs = model(inputs)

            # Ensure that both outputs and targets are on the same device
            outputs = outputs.to(device1)
            loss = criterion(outputs, targets)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
    print("Training completed.")

if __name__ == "__main__":
    main()
```

FullyShardedDataParallel with PyTorch Distributed

Model Parallelism with PyTorch Distributed

FullyShardedDataParallel (FSDP) in PyTorch Distributed

- FullyShardedDataParallel doesn’t save separate layers to separate GPUs (as we did before)
- Instead: Models are sharded, i.e. every GPU gets part of the whole model
  - In FSDP, each GPU holds only a shard of the model's parameters, not the entire model
  - If a model has P parameters and there are N GPUs, each GPU stores approximately P/N parameters
- When unsharding happens, each GPU only processes computations involving the parameters it owns
- Sharding keeps memory usage low by not storing full tensors on any single GPU during inactive periods
- No Redundant Computations:
  - GPUs work collaboratively, with each GPU contributing to the computations needed for the full model's forward and backward passes
- FSDP enables training models that would otherwise not fit in GPU memory

Model Parallelism with PyTorch Distributed

Steps of FSDP in PyTorch Distributed

- Model sharding:
  - Split model parameters, gradients, and optimizer states across multiple GPUs
  - Each GPU holds only a shard of the model's parameters, not the full model
- Forward pass:
  - For each layer, each GPU unshards (temporarily assembles) the necessary parameters it needs for computation
  - GPUs compute their portion of the forward pass using their shard and unsharded parameters from other GPUs
- Backward pass:
  - Each GPU computes the gradients for its shard of the model parameters
  - GPUs unshard gradients and optimizer states for their relevant parts to compute and apply updates
- Optimizer step:
  - Gradients from all GPUs are aggregated and synchronized across GPUs using communication (e.g., all-reduce) to ensure consistent updates
  - After gradients are synchronized, the optimizer states (like momentum or Adam’s second moment) are updated in a sharded manner
- Resharding**:**
  - After each computation (forward and backward), model parameters, gradients, and optimizer states are resharded across GPUs to reduce memory footprint

NVIDIA®