Data- & Pipeline- Parallel Training

COMP6211J

Binhang Yuan

Data Parallel Paradigm

Recall the Mini-Batch SGD

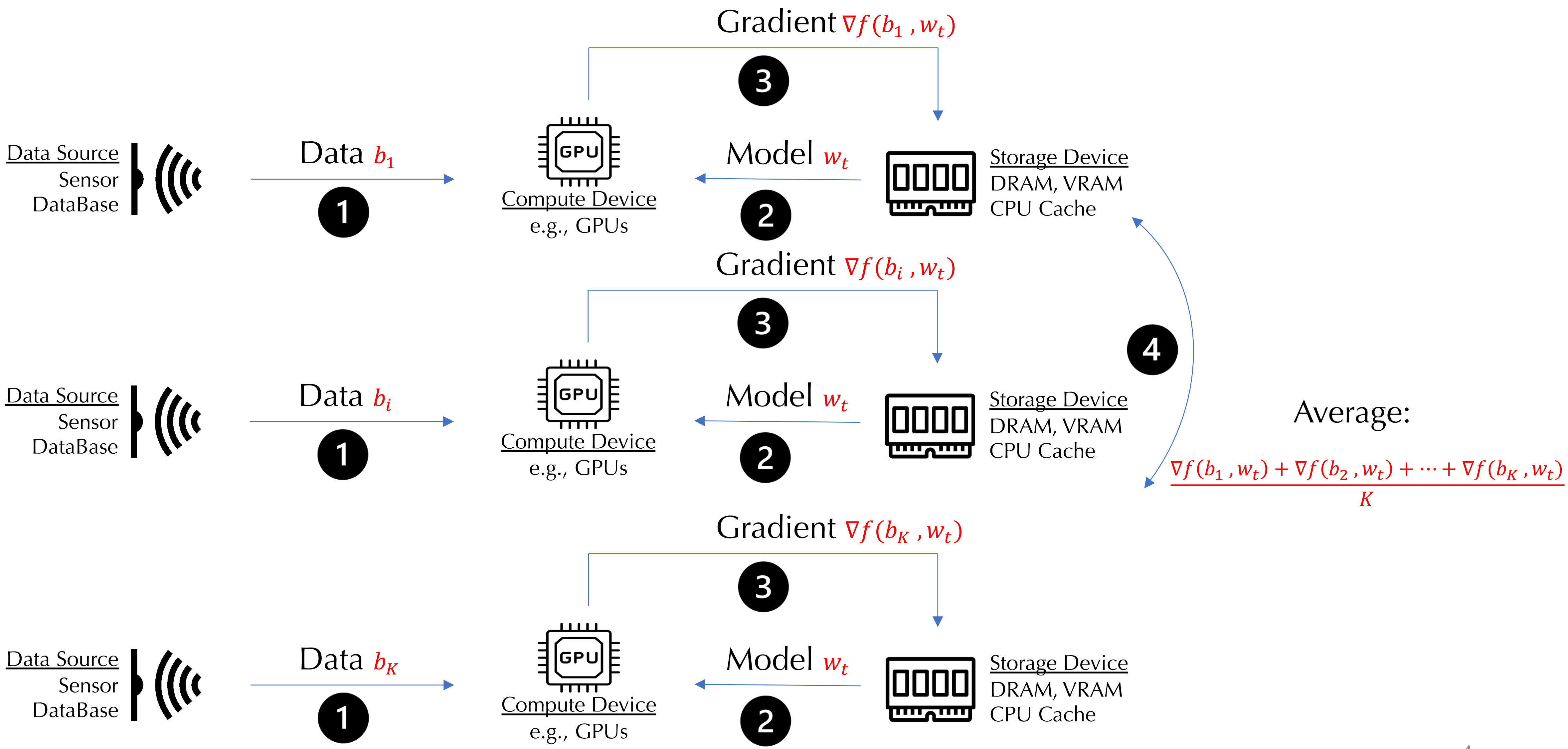
- Basic ideas:
  - To reduce the variance of stochastic gradients;
  - Split the training data into smaller batches;
  - Sampling batch (usually without replacement)
- Suppose we have:
  - B is the batch size:
  - K is the number of GPUs.
- ***Distribute the computation by the data dimension.***

$$w_{t+1} = w_t - \alpha_t \cdot \underbrace{\frac{1}{\mathrm{B}}\sum_{i=1}^{B} \nabla f(w_t;\xi_i)}_{\textbf{Local Computation}}$$

$$w_{t+1} = w_t - \alpha_t \cdot (\underbrace{\frac{1}{\mathrm{B}}\sum_{i_1=1}^{\frac{B}{K}} \nabla f\left(w_t;\xi_{i_1}\right)}_{\textbf{GPU 1}} + \underbrace{\frac{1}{\mathrm{B}}\sum_{i_2=1}^{\frac{B}{K}} \nabla f\left(w_t;\xi_{i_2}\right)}_{\textbf{GPU 2}} + \cdots + \underbrace{\frac{1}{\mathrm{B}}\sum_{i_K=1}^{\frac{B}{K}} \nabla f\left(w_t;\xi_{i_K}\right)}_{\textbf{GPU K}})$$

Aggregation

Data Parallel SGD



Gradient $\nabla f(b_1, w_t)$

3

Data Source
Sensor
DataBase

Data b_1

1

Compute Device
e.g., GPUs

Model w_t

2

Storage Device
DRAM, VRAM
CPU Cache

Gradient $\nabla f(b_i, w_t)$

3

Data Source
Sensor
DataBase

Data b_i

1

Compute Device
e.g., GPUs

Model w_t

2

Storage Device
DRAM, VRAM
CPU Cache

4

Average:

$$\frac{\nabla f(b_1, w_t) + \nabla f(b_2, w_t) + \cdots + \nabla f(b_K, w_t)}{K}$$

Gradient $\nabla f(b_K, w_t)$

3

Data Source
Sensor
DataBase

Data b_K

1

Compute Device
e.g., GPUs

Model w_t

2

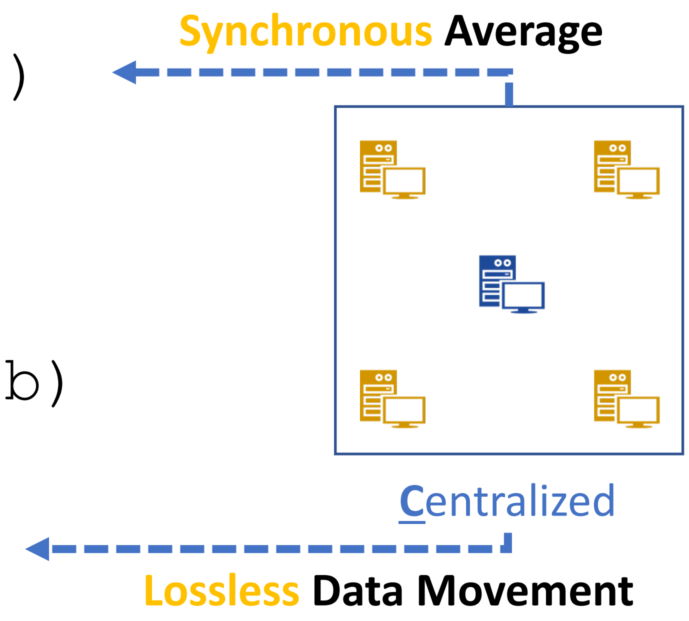
Storage Device
DRAM, VRAM
CPU Cache

Basic Implementation

Core idea of data parallelism:

- Distribute batch gradient calculation to multiple workers;
- Synchronize gradients among GPU workers:
  - A central server (or AllReduce).
  - Synchronous average;
  - Lossless data movement.

```
w = sync_model()

b = get_data()

g = get_grad(w,b)

w = update(w,g)
```



Basic Implementation

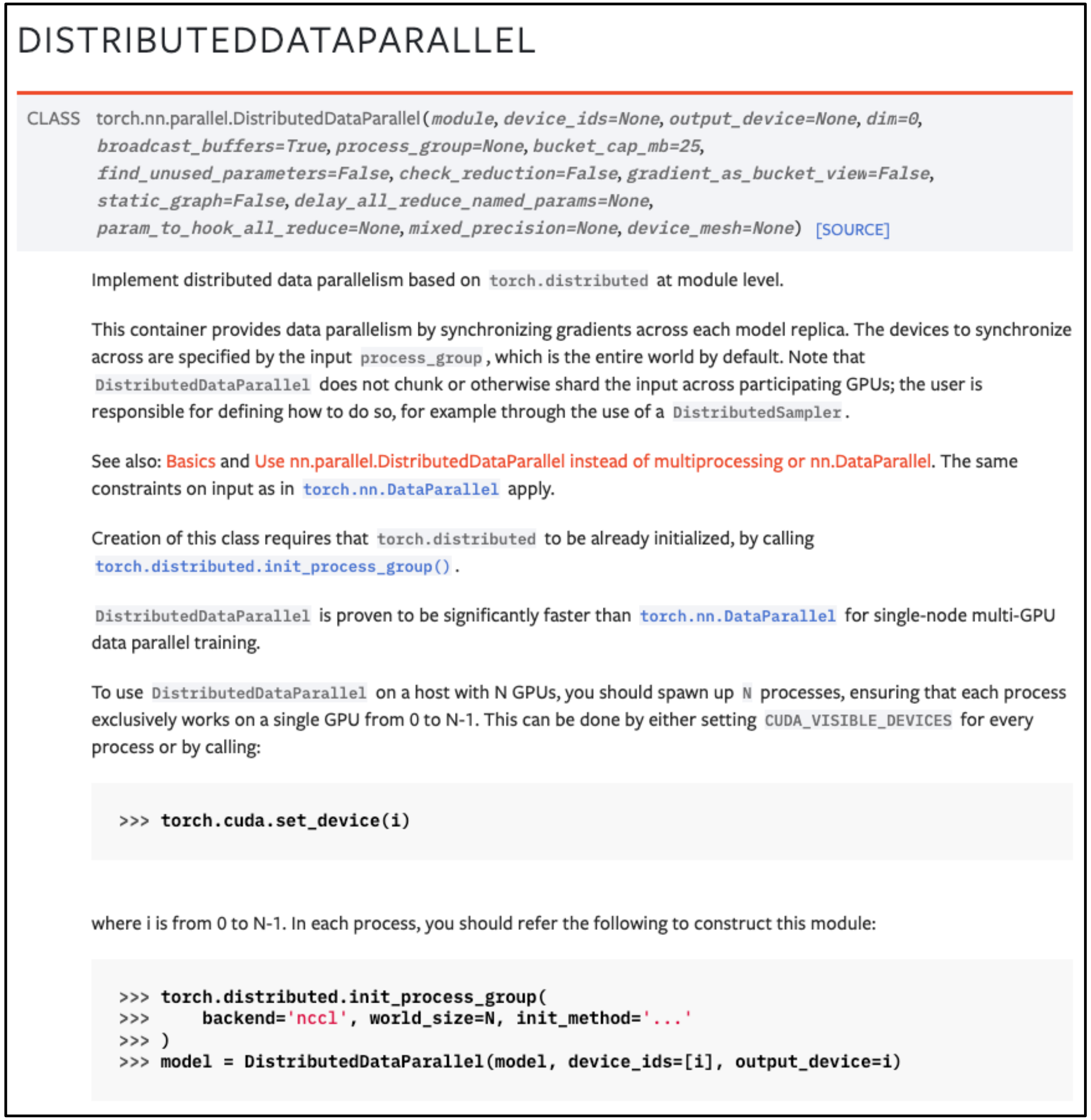
- Suppose:
  - C is the total computation in the original forward and backward computation (counted by FLOPs);
  - D is the total number of parameters of the model;
  - K is the total number of GPUs in the cluster;
- Under Data parallel training:
  - What is the total computation on each device? $\frac{C}{K}$
  - What is the communication volume? **ALLReduce** a D dimensional buffer.

PyTorch-DDP Practice

Overview

- PyTorch **DistributedDataParallel** (DDP) enables data parallel training in PyTorch.
- PyTorch **DistributedSampler** ensures each device gets a non-overlapping input batch.
- The model is replicated on all the devices.
- Each replica calculates gradients and synchronizes with the others using the **AllReduce** operation.

DDP API



DISTRIBUTEDDATAPARALLEL

CLASS torch.nn.parallel.DistributedDataParallel(*module, device_ids=None, output_device=None, dim=0, broadcast_buffers=True, process_group=None, bucket_cap_mb=25, find_unused_parameters=False, check_reduction=False, gradient_as_bucket_view=False, static_graph=False, delay_all_reduce_named_params=None, param_to_hook_all_reduce=None, mixed_precision=None, device_mesh=None*) [SOURCE]

Implement distributed data parallelism based on `torch.distributed` at module level.

This container provides data parallelism by synchronizing gradients across each model replica. The devices to synchronize across are specified by the input `process_group`, which is the entire world by default. Note that `DistributedDataParallel` does not chunk or otherwise shard the input across participating GPUs; the user is responsible for defining how to do so, for example through the use of a `DistributedSampler`.

See also: Basics and Use nn.parallel.DistributedDataParallel instead of multiprocessing or nn.DataParallel. The same constraints on input as in `torch.nn.DataParallel` apply.

Creation of this class requires that `torch.distributed` to be already initialized, by calling `torch.distributed.init_process_group()`.

`DistributedDataParallel` is proven to be significantly faster than `torch.nn.DataParallel` for single-node multi-GPU data parallel training.

To use `DistributedDataParallel` on a host with N GPUs, you should spawn up `N` processes, ensuring that each process exclusively works on a single GPU from 0 to N-1. This can be done by either setting `CUDA_VISIBLE_DEVICES` for every process or by calling:

```
>>> torch.cuda.set_device(i)
```

where i is from 0 to N-1. In each process, you should refer the following to construct this module:

```
>>> torch.distributed.init_process_group(
>>>     backend='nccl', world_size=N, init_method='...'
>>> )
>>> model = DistributedDataParallel(model, device_ids=[i], output_device=i)
```

Initialize the Process Group

Initialize the Process Group

```
import os
import sys
import tempfile
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
import torch.multiprocessing as mp

from torch.nn.parallel import DistributedDataParallel as DDP


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()
```

Use DDP API

Define the Toy Model

```
class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = nn.Linear(10, 10)
        self.relu = nn.ReLU()
        self.net2 = nn.Linear(10, 5)

    def forward(self, x):
        return self.net2(self.relu(self.net1(x)))
```

Use DDP API

Use DDP API

```
def demo_basic(rank, world_size):
    print(f"Running basic DDP example on rank {rank}.")
    setup(rank, world_size)

    # create model and move it to GPU with id rank
    model = ToyModel().to(rank)
    ddp_model = DDP(model, device_ids=[rank])

    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(rank)
    loss_fn(outputs, labels).backward()
    optimizer.step()

    cleanup()


def run_demo(demo_fn, world_size):
    mp.spawn(demo_fn,
             args=(world_size,),
             nprocs=world_size,
             join=True)
```

System Optimization

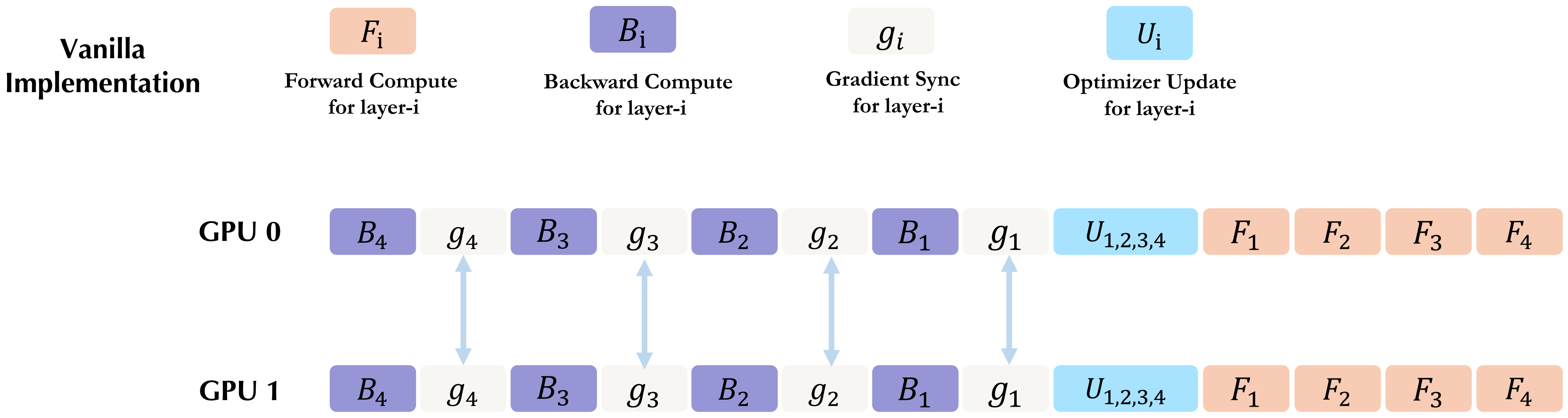
System Optimization

- Gradient Bucketing:
  - Collective communications are more efficient with large tensors.
  - Group gradient tensor to buckets on continuous memory (known as flatten.)
- Overlapping communication and computation during back-propagation：
  - The AllReduce operation on gradients can start before the local backward pass finishes.
  - With bucketing, DDP only needs to wait for all contents in the same bucket before launching communications.

System Optimization

- **PyTorch System:**

Optimize the standard data-parallel computation:



Vanilla Implementation

F_i Forward Compute for layer-i

B_i Backward Compute for layer-i

g_i Gradient Sync for layer-i

U_i Optimizer Update for layer-i

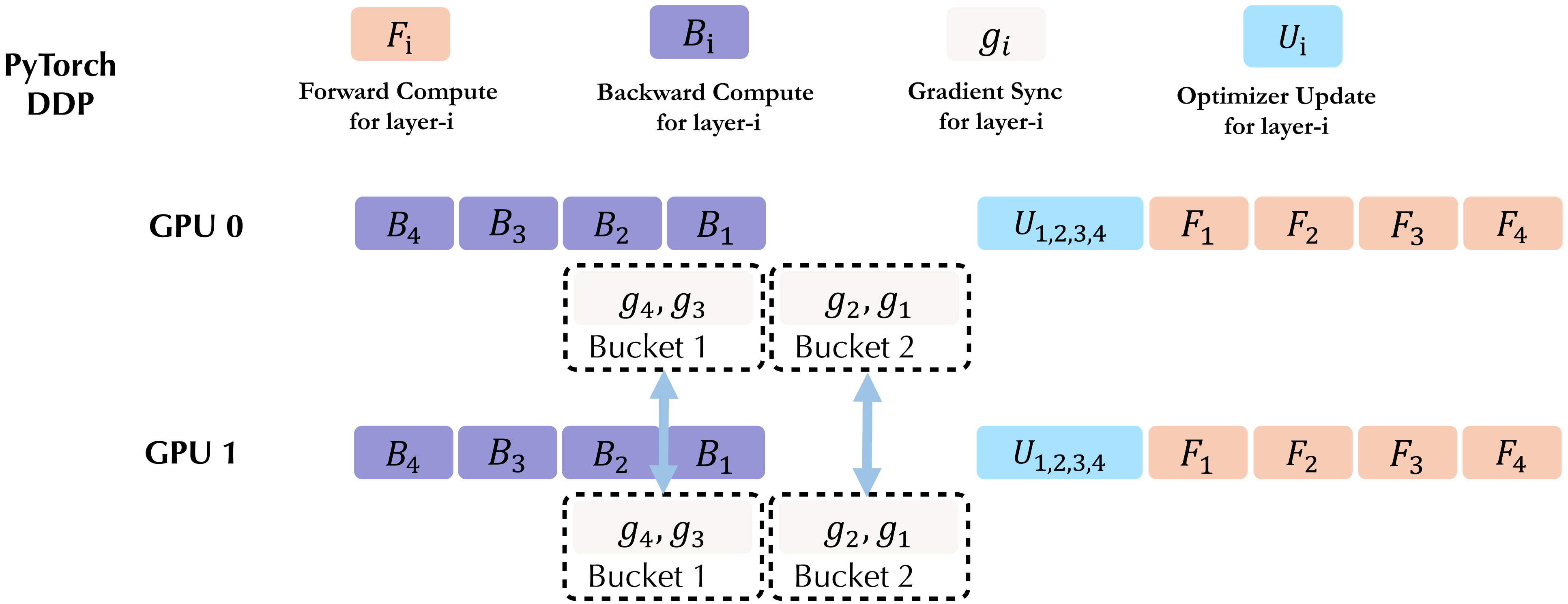
GPU 0 B_4 g_4 B_3 g_3 B_2 g_2 B_1 g_1 $U_{1,2,3,4}$ F_1 F_2 F_3 F_4

GPU 1 B_4 g_4 B_3 g_3 B_2 g_2 B_1 g_1 $U_{1,2,3,4}$ F_1 F_2 F_3 F_4

System Optimization

- **PyTorch System:**

Optimize the standard data-parallel computation:


PyTorch DDP

F_i Forward Compute for layer-i

B_i Backward Compute for layer-i

g_i Gradient Sync for layer-i

U_i Optimizer Update for layer-i

Algorithm Optimization

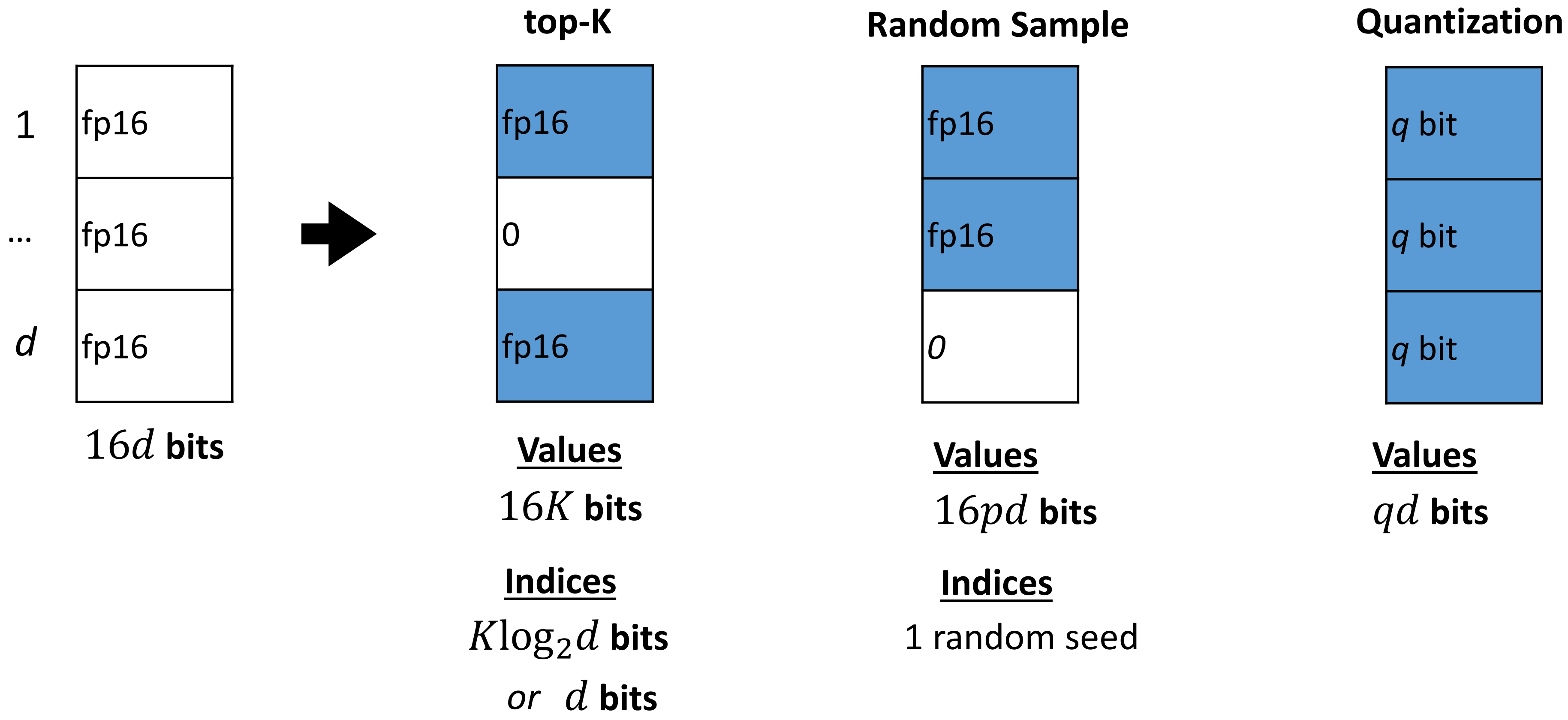
Overview

- Three main categories of algorithm design:
    - Loss communication compression;
    - Asynchronous training;
    - Decentralized communication.

Loss Communication Compression

- Instead of exchanging the gradient as full precision floating point numbers;
- The system first compresses it into a lower precision representation before communication.
- Compression methods:
    - Top-K sparsification;
    - Random sparsification;
    - Quantization.

Compression Methods



$16d$ **bits**

top-K

Values

$16K$ **bits**

Indices

$K\log_2 d$ **bits**

or d **bits**

Random Sample

Values

$16pd$ **bits**

Indices

1 random seed

Quantization

Values

qd **bits**

Expensive to compute
and to encode Indices

Might not keep top
values as in Top-K

Only provide up to 32x
compression; hard to go aggressive

Asynchronous Training

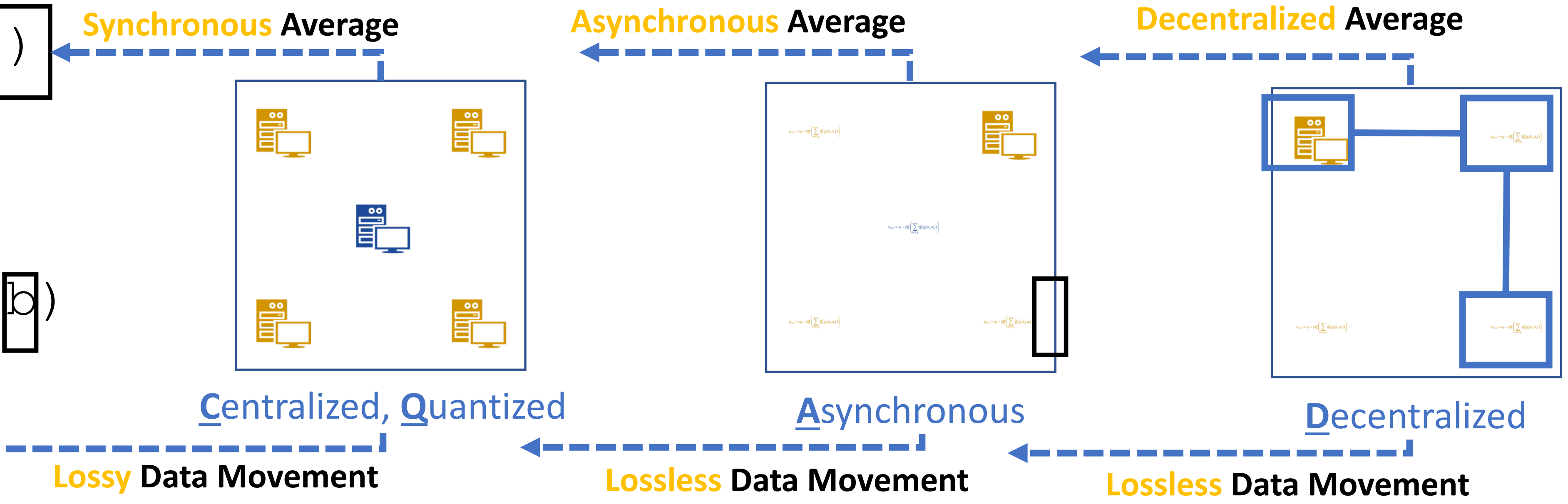
- Original paradigm: all workers conduct computation simultaneously and are all blocked until the communication among all machines is finished.
- If some machines are slower than other machines (i.e., there are stragglers), all machines need to wait for the slowest machine to finish the computation.
- Asynchronous training removes the synchronization barriers among all the machines with the consequence that each machine now has access to a staled model.

Decentralized Communication

- Lossy communication compression is designed to alleviate system bottlenecks caused by network bandwidth.
- Another type of network bottleneck is caused by latency.
- In that case, when there are K workers, the ring -based **AllReduce** implementation has an $\mathcal{O}(K)$ dependency on the network latency.
- Decentralized training:
  - Suppose a logical ring among K workers;
  - At every single iteration, a worker sends its model replica to the neighbour on its immediate left and neighbour on its immediate right;
  - Latency overhead becomes $\mathcal{O}(1)$.
  - On the downside, however, the information on a single worker only reaches its two adjacent neighbors in one round of communication.

Algorithm Illustration.

```
w = sync_model()
b = get_data()
g = get_grad(w,b)
w = update(w,g)
```



Mathematical Formulation

$$w_{t+1} = w_t - \gamma C\left(\sum_{i=1..n} C(\nabla f_i(x_t, b_i))\right)$$

$$w_{t+1} = w_t - \gamma \nabla f(x_{t-\tau_t}; b_i)$$

staleness caused by async

$$w_{t+1,i} = \frac{w_{t,i-1} + w_{t,i} + w_{t,i+1}}{3} - \gamma \nabla f(w_{t,i}; b_i)$$

Summary of Data Parallelism

- Advantages of data parallelism:
  - Easy to implement;
  - Generally applicable to most machine learning models.
- Limitation of data parallelism:
  - **Memory issue**: each device needs to maintain a complete copy of the model (parameters, gradients, and optimizer status). This is impossible for current LLMs; the most powerful GPU only has 80 GB DRAM.
  - **Statistical efficiency**: if the global batch size is too large, it may affect the convergence rate. Data parallelism cannot be used for infinite scale-out.

Pipeline Parallel Paradigm

Pipeline Parallelism Overview

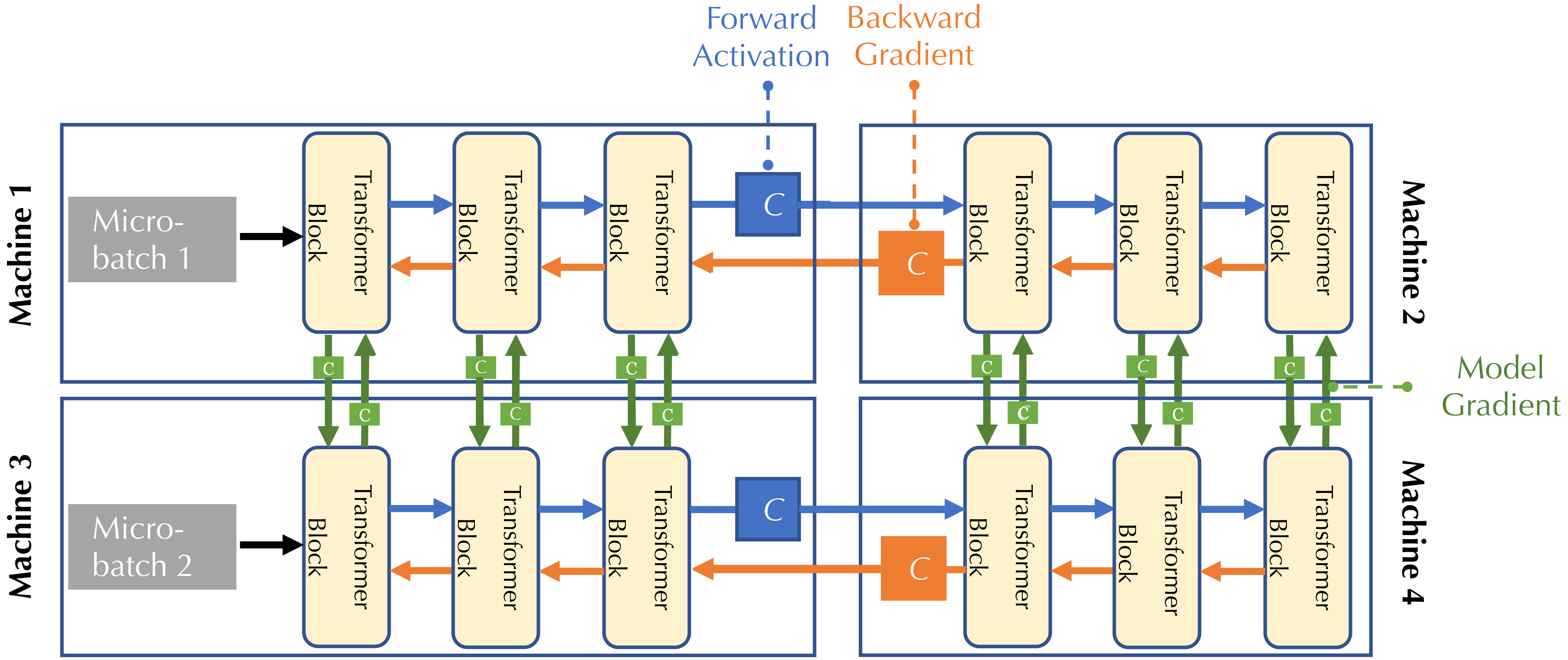
- For large models that don't fit on a single GPU, pipeline parallelism partitions the model into multiple stages.
- Different nodes are responsible for different layers of a model.
- Communication in pipeline parallel paradigm:
  - Activations in the forward propagation;
  - Gradients of the activation in the backward propagation.
- Pipeline parallelism can be combined with data parallelism:
  - The nodes handling the same stage need to perform data parallel synchronization.

Formulation

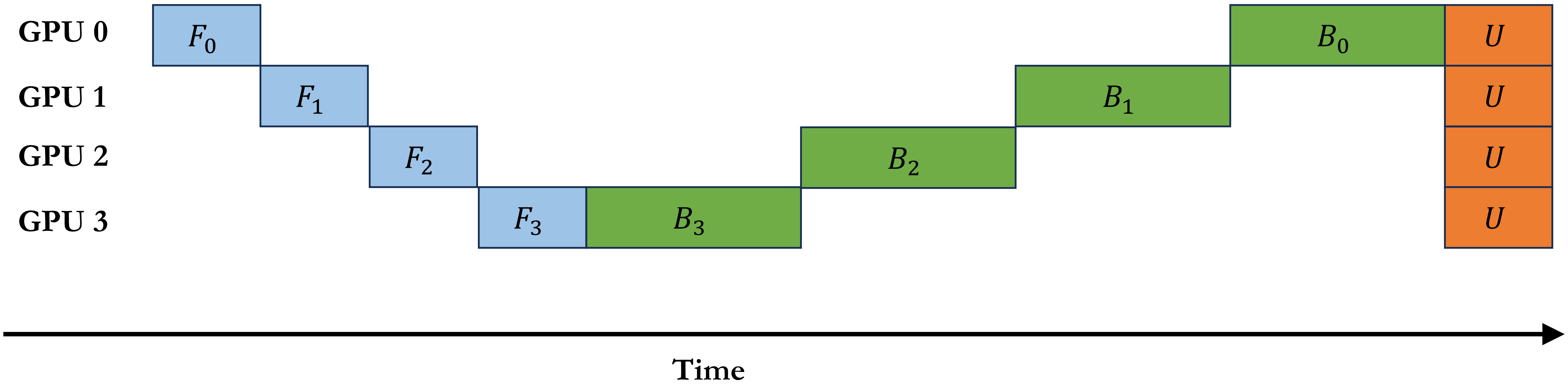
$$\min_{x} \mathbb{E}_{\xi} f(\xi, x) \Rightarrow \min_{x_f, x_g} \mathbb{E}_{\xi} \overset{\text{Stage 2}}{f}(\overset{\text{Stage 1}}{g}(\xi, x_g), x_f)$$

Forward Activation

Pipeline Parallel



A Naïve Implementation



The number on each block represents the stage index.

RELAXED
SYSTEM LAB

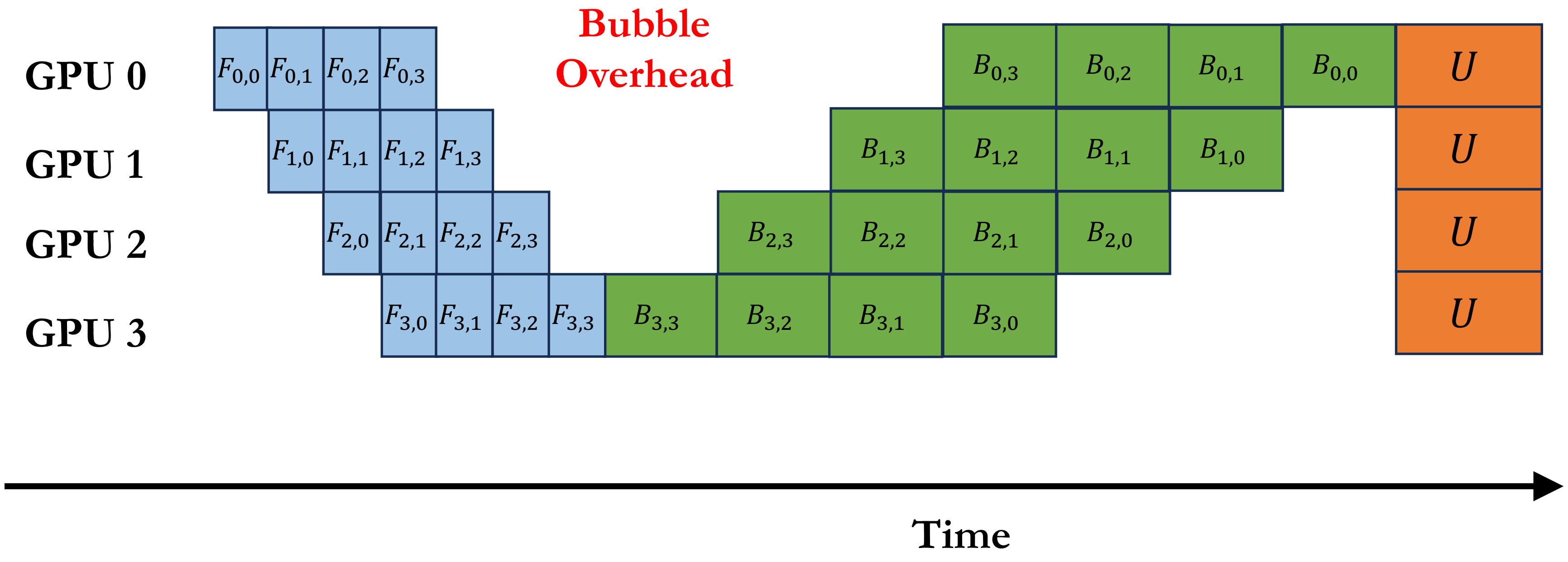
Pipeline Parallel Optimization

Pipeline Parallelism Optimization

RELAXED SYSTEM LAB

- In the naïve implementation, only one GPU is active at a time.
- This makes the MFU very low.
- Optimization: spilt each batch to **micro-batch**es so that computation can be executed as a pipeline paradigm.
- There are some system optimizations to improve the efficiency of pipeline parallelism:
  - **Gpipe**: https://arxiv.org/abs/1811.06965
  - **1F1B** from PipeDream-Flush: https://proceedings.mlr.press/v139/narayanan21a/narayanan21a.pdf

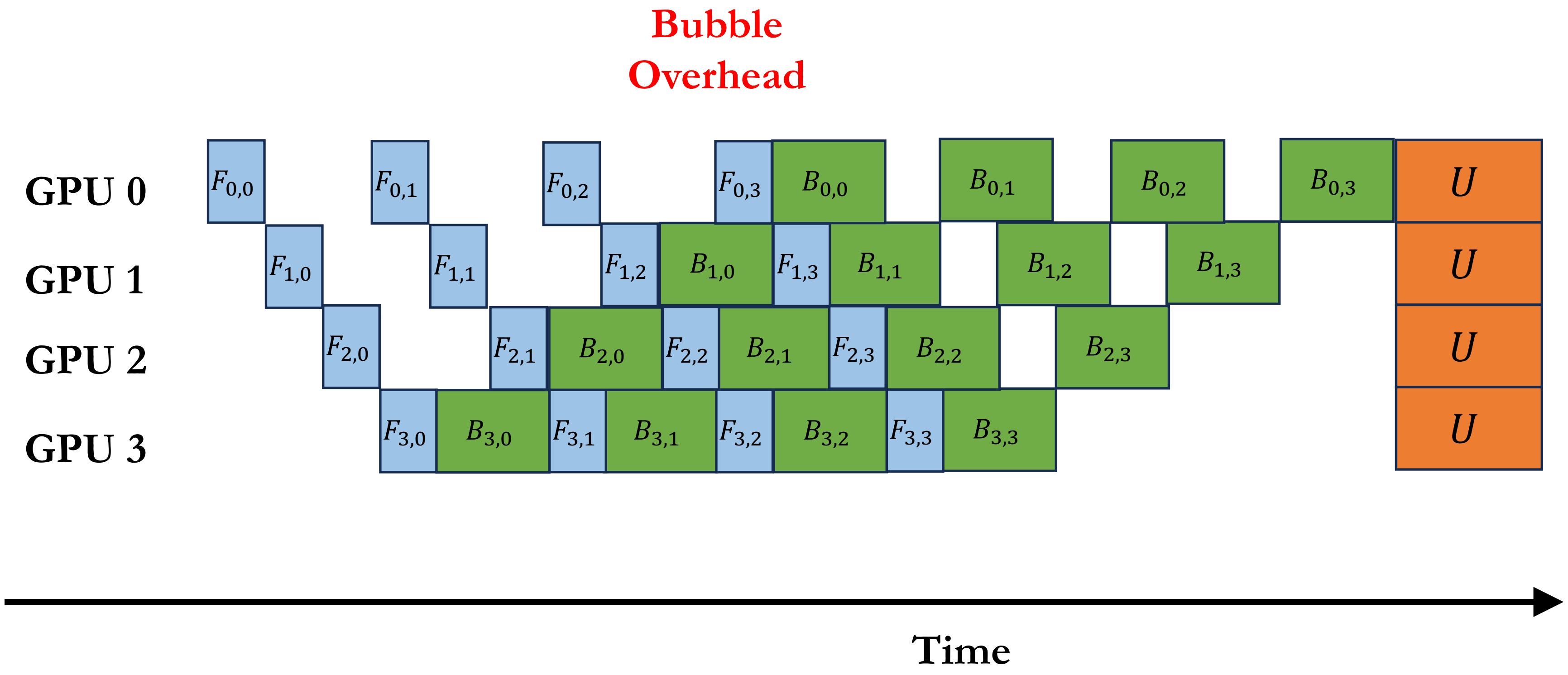
GPipe


GPU 0
GPU 1
GPU 2
GPU 3
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$
$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$
Bubble Overhead
$B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$
$B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$
$B_{2,3}$ $B_{2,2}$ $B_{2,1}$ $B_{2,0}$
$B_{3,3}$ $B_{3,2}$ $B_{3,1}$ $B_{3,0}$
U
U
U
U
Time

The number on each block represents the stage index and the micro-batch index.

- If we ignore the computation time of optimizer updates.
- Suppose:
  - K is the number of GPUs;
  - M is the number of micro-batches;
- What is the percentage of bubble overhead? $\frac{K-1}{M+K-1}$

1F1B

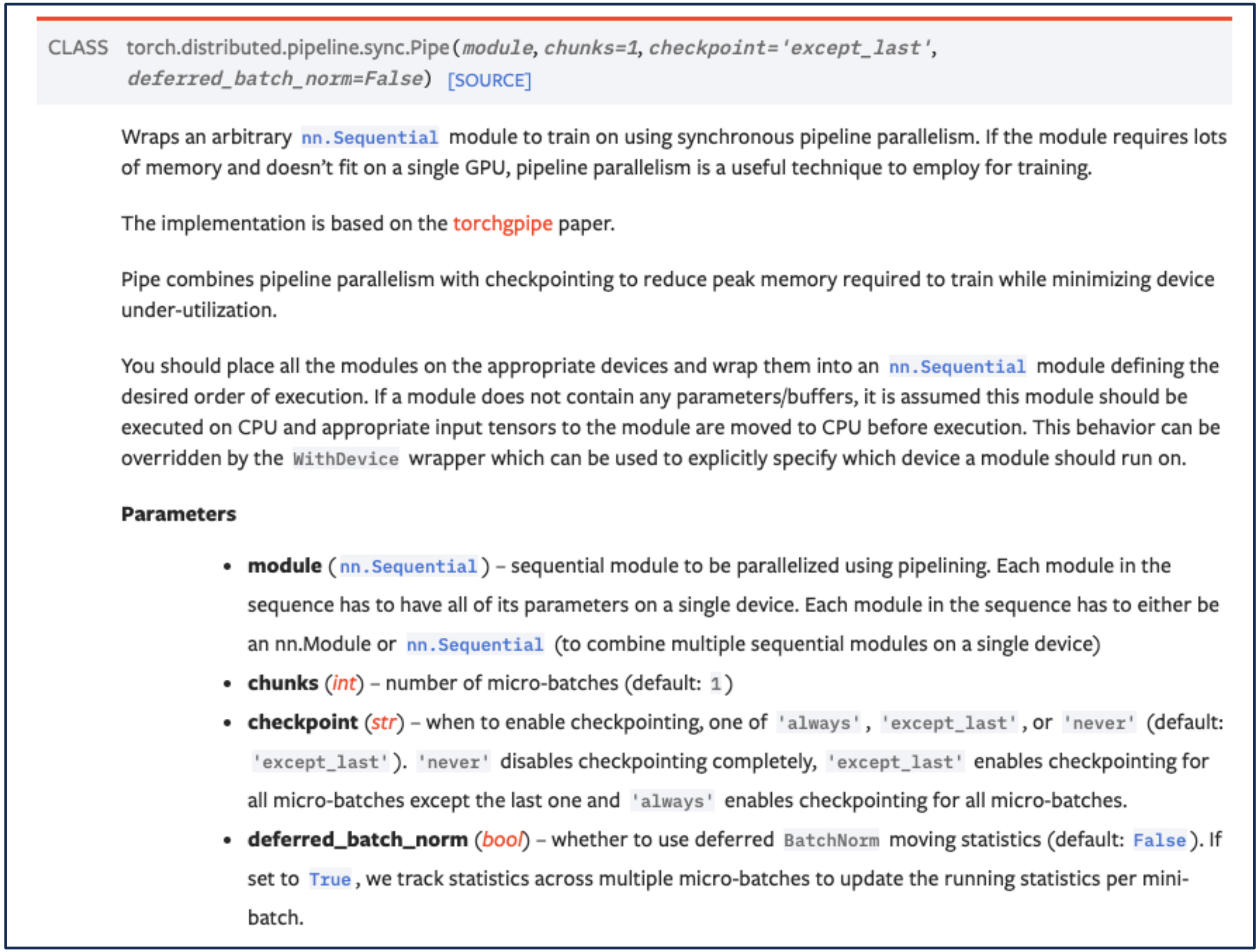


The number on each block represents the stage index and the micro-batch index.

- If we ignore the computation time of optimizer updates.
- Suppose:
  - K is the number of GPUs;
  - M is the number of micro-batches;
- What is the percentage of bubble overhead? $\frac{K-1}{M+K-1}$

Pipeline Parallelism in Practice

Pipe API in PyTorch (Gpipe Implementation)



CLASS torch.distributed.pipeline.sync.Pipe(*module*, *chunks=1*, *checkpoint='except_last'*, *deferred_batch_norm=False*) [SOURCE]

Wraps an arbitrary `nn.Sequential` module to train on using synchronous pipeline parallelism. If the module requires lots of memory and doesn't fit on a single GPU, pipeline parallelism is a useful technique to employ for training.

The implementation is based on the torchgpipe paper.

Pipe combines pipeline parallelism with checkpointing to reduce peak memory required to train while minimizing device under-utilization.

You should place all the modules on the appropriate devices and wrap them into an `nn.Sequential` module defining the desired order of execution. If a module does not contain any parameters/buffers, it is assumed this module should be executed on CPU and appropriate input tensors to the module are moved to CPU before execution. This behavior can be overridden by the `WithDevice` wrapper which can be used to explicitly specify which device a module should run on.

Parameters

- **module** (`nn.Sequential`) – sequential module to be parallelized using pipelining. Each module in the sequence has to have all of its parameters on a single device. Each module in the sequence has to either be an nn.Module or `nn.Sequential` (to combine multiple sequential modules on a single device)
- **chunks** (*int*) – number of micro-batches (default: `1`)
- **checkpoint** (*str*) – when to enable checkpointing, one of `'always'`, `'except_last'`, or `'never'` (default: `'except_last'`). `'never'` disables checkpointing completely, `'except_last'` enables checkpointing for all micro-batches except the last one and `'always'` enables checkpointing for all micro-batches.
- **deferred_batch_norm** (*bool*) – whether to use deferred `BatchNorm` moving statistics (default: `False`). If set to `True`, we track statistics across multiple micro-batches to update the running statistics per mini-batch.

Pipeline Parallelism in Deepspeed

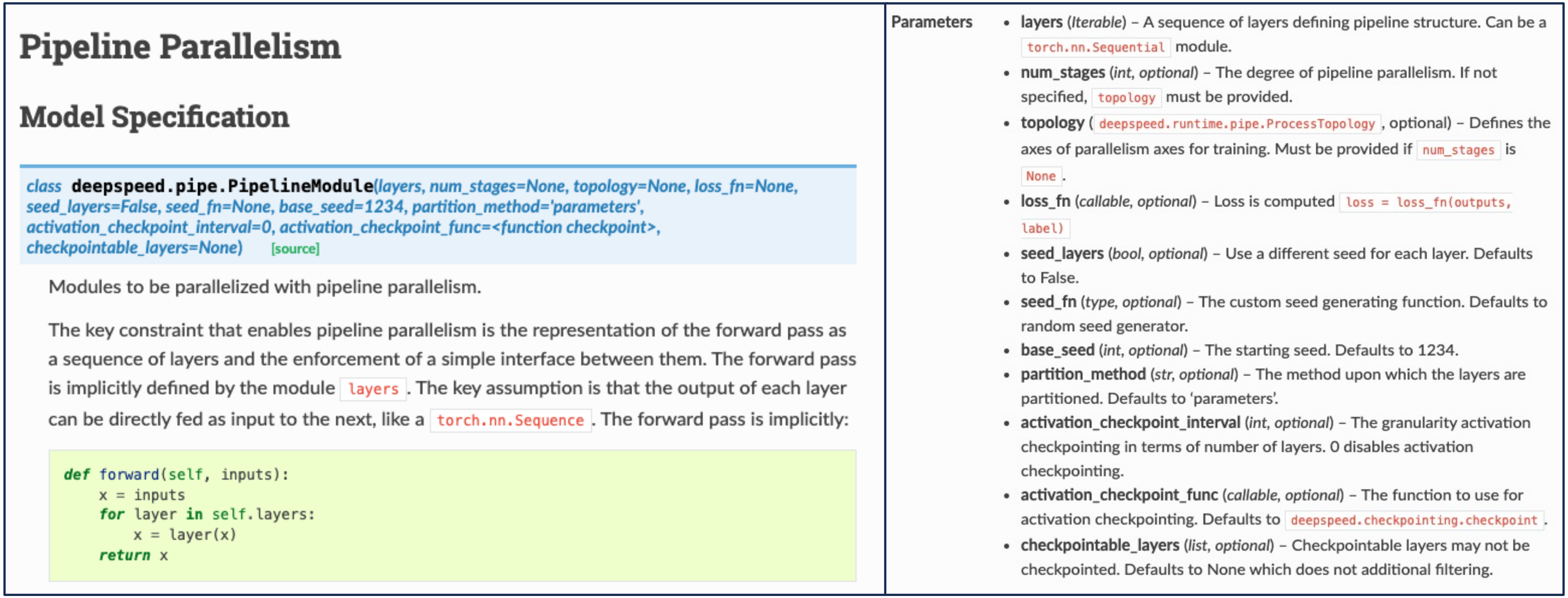
deepspeed

- *“DeepSpeed is an easy-to-use deep learning optimization software suite that enables unprecedented scale and speed for DL Training and Inference.”*

  -- Microsoft

- Support includes:
  - Large model training;
    - data parallelism, pipeline parallelism, tensor model parallelism, ZeRO-S1, S2, S3.
  - Large model inference.

Pipeline Parallelism in Deepspeed



Pipeline Parallelism

Model Specification

class **deepspeed.pipe.PipelineModule**(*layers, num_stages=None, topology=None, loss_fn=None, seed_layers=False, seed_fn=None, base_seed=1234, partition_method='parameters', activation_checkpoint_interval=0, activation_checkpoint_func=<function checkpoint>, checkpointable_layers=None*) [source]

Modules to be parallelized with pipeline parallelism.

The key constraint that enables pipeline parallelism is the representation of the forward pass as a sequence of layers and the enforcement of a simple interface between them. The forward pass is implicitly defined by the module `layers`. The key assumption is that the output of each layer can be directly fed as input to the next, like a `torch.nn.Sequence`. The forward pass is implicitly:

```
def forward(self, inputs):
    x = inputs
    for layer in self.layers:
        x = layer(x)
    return x
```

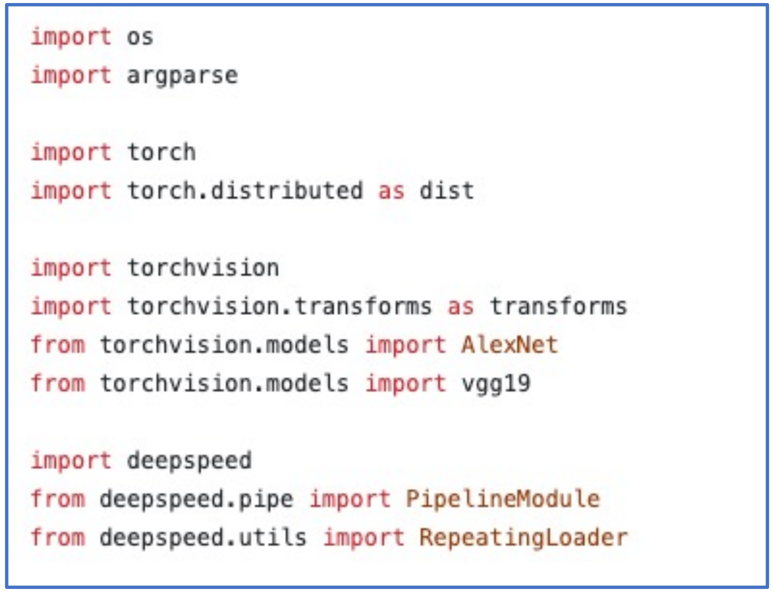
Parameters

- **layers** (*Iterable*) – A sequence of layers defining pipeline structure. Can be a `torch.nn.Sequential` module.
- **num_stages** (*int, optional*) – The degree of pipeline parallelism. If not specified, `topology` must be provided.
- **topology** (`deepspeed.runtime.pipe.ProcessTopology`, optional) – Defines the axes of parallelism axes for training. Must be provided if `num_stages` is `None`.
- **loss_fn** (*callable, optional*) – Loss is computed `loss = loss_fn(outputs, label)`
- **seed_layers** (*bool, optional*) – Use a different seed for each layer. Defaults to False.
- **seed_fn** (*type, optional*) – The custom seed generating function. Defaults to random seed generator.
- **base_seed** (*int, optional*) – The starting seed. Defaults to 1234.
- **partition_method** (*str, optional*) – The method upon which the layers are partitioned. Defaults to 'parameters'.
- **activation_checkpoint_interval** (*int, optional*) – The granularity activation checkpointing in terms of number of layers. 0 disables activation checkpointing.
- **activation_checkpoint_func** (*callable, optional*) – The function to use for activation checkpointing. Defaults to `deepspeed.checkpointing.checkpoint`.
- **checkpointable_layers** (*list, optional*) – Checkpointable layers may not be checkpointed. Defaults to None which does not additional filtering.

AlexNet Pipeline Parallel Training in Deepspeed

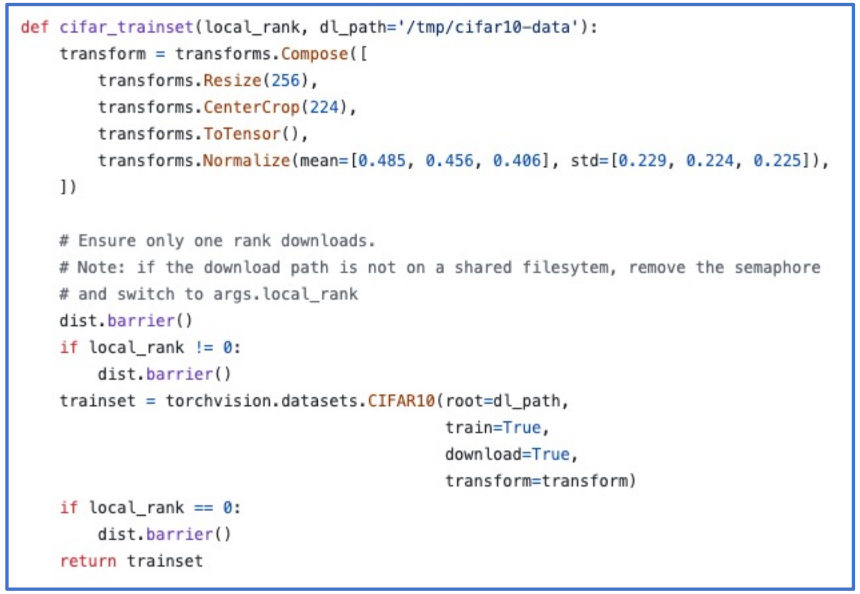
- Code: https://github.com/microsoft/DeepSpeedExamples/blob/master/training/pipeline_parallelism/train.py

Load the library



```python
import os
import argparse

import torch
import torch.distributed as dist

import torchvision
import torchvision.transforms as transforms
from torchvision.models import AlexNet
from torchvision.models import vgg19

import deepspeed
from deepspeed.pipe import PipelineModule
from deepspeed.utils import RepeatingLoader
```

Define the data loader

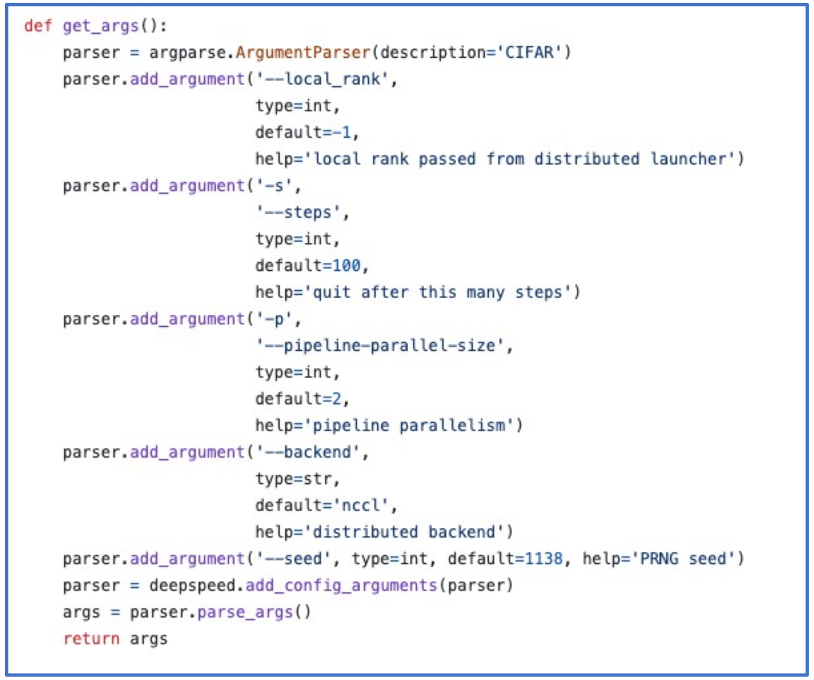


```python
def cifar_trainset(local_rank, dl_path='/tmp/cifar10-data'):
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])

    # Ensure only one rank downloads.
    # Note: if the download path is not on a shared filesytem, remove the semaphore
    # and switch to args.local_rank
    dist.barrier()
    if local_rank != 0:
        dist.barrier()
    trainset = torchvision.datasets.CIFAR10(root=dl_path,
                                            train=True,
                                            download=True,
                                            transform=transform)
    if local_rank == 0:
        dist.barrier()
    return trainset
```

AlexNet Pipeline Parallel Training in Deepspeed

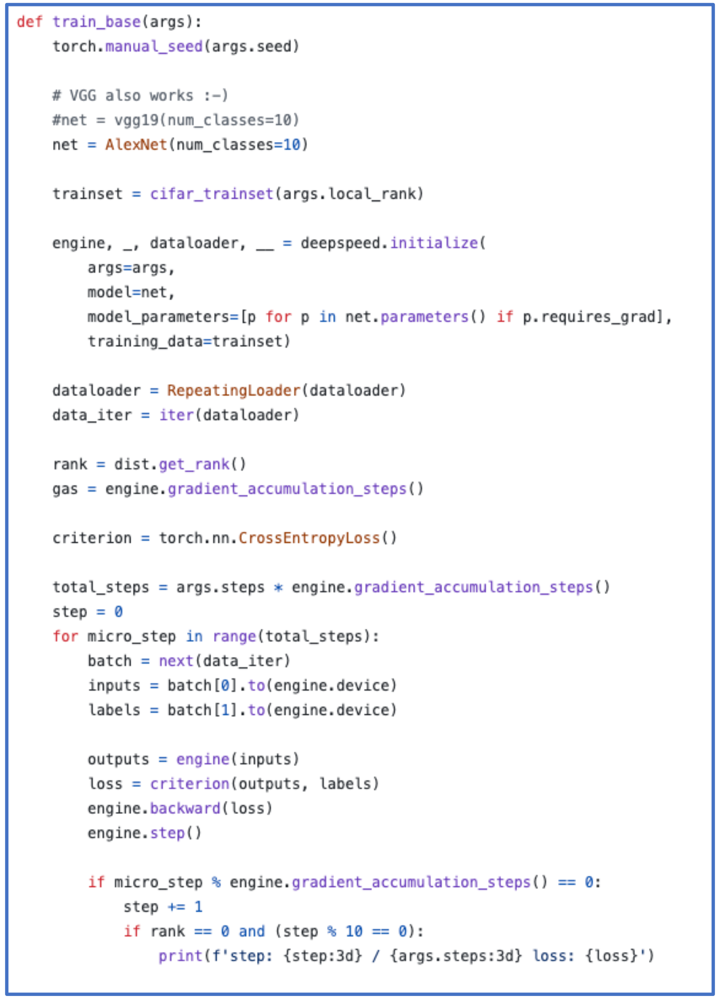
- Code: https://github.com/microsoft/DeepSpeedExamples/blob/master/training/pipeline_parallelism/train.py

Pass configurations



```
def get_args():
    parser = argparse.ArgumentParser(description='CIFAR')
    parser.add_argument('--local_rank',
                        type=int,
                        default=-1,
                        help='local rank passed from distributed launcher')
    parser.add_argument('-s',
                        '--steps',
                        type=int,
                        default=100,
                        help='quit after this many steps')
    parser.add_argument('-p',
                        '--pipeline-parallel-size',
                        type=int,
                        default=2,
                        help='pipeline parallelism')
    parser.add_argument('--backend',
                        type=str,
                        default='nccl',
                        help='distributed backend')
    parser.add_argument('--seed', type=int, default=1138, help='PRNG seed')
    parser = deepspeed.add_config_arguments(parser)
    args = parser.parse_args()
    return args
```

Non-pipeline training loop

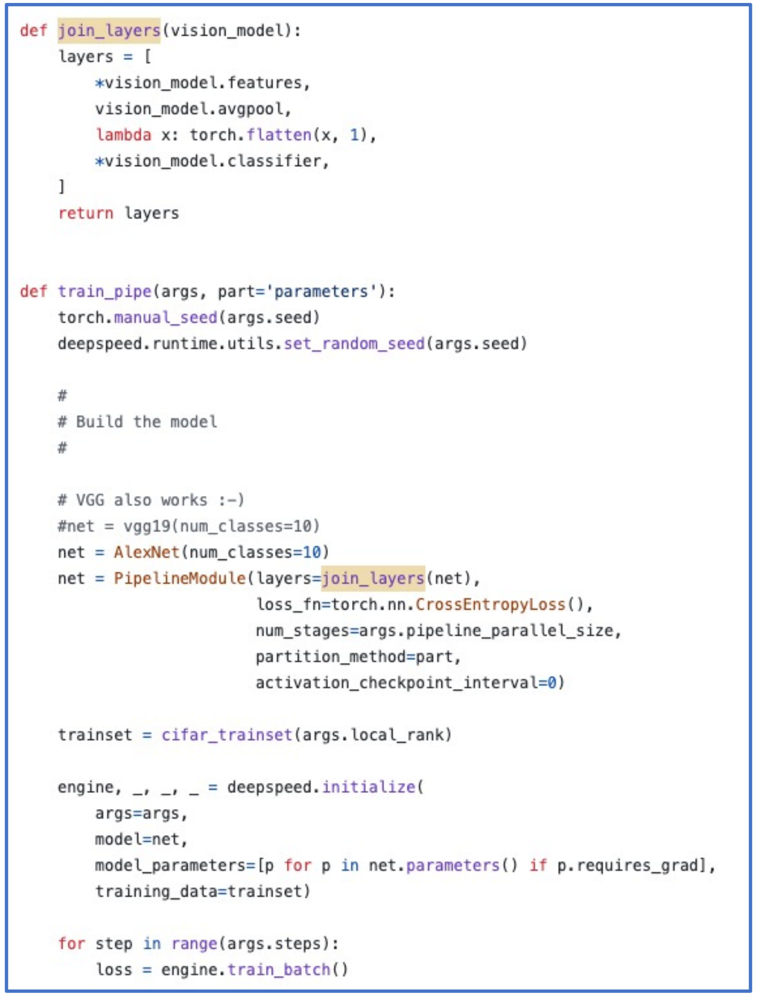


```
def train_base(args):
    torch.manual_seed(args.seed)

    # VGG also works :-)
    #net = vgg19(num_classes=10)
    net = AlexNet(num_classes=10)

    trainset = cifar_trainset(args.local_rank)

    engine, _, dataloader, __ = deepspeed.initialize(
        args=args,
        model=net,
        model_parameters=[p for p in net.parameters() if p.requires_grad],
        training_data=trainset)

    dataloader = RepeatingLoader(dataloader)
    data_iter = iter(dataloader)

    rank = dist.get_rank()
    gas = engine.gradient_accumulation_steps()

    criterion = torch.nn.CrossEntropyLoss()

    total_steps = args.steps * engine.gradient_accumulation_steps()
    step = 0
    for micro_step in range(total_steps):
        batch = next(data_iter)
        inputs = batch[0].to(engine.device)
        labels = batch[1].to(engine.device)

        outputs = engine(inputs)
        loss = criterion(outputs, labels)
        engine.backward(loss)
        engine.step()

        if micro_step % engine.gradient_accumulation_steps() == 0:
            step += 1
            if rank == 0 and (step % 10 == 0):
                print(f'step: {step:3d} / {args.steps:3d} loss: {loss}')
```

AlexNet Pipeline Parallel Training in Deepspeed

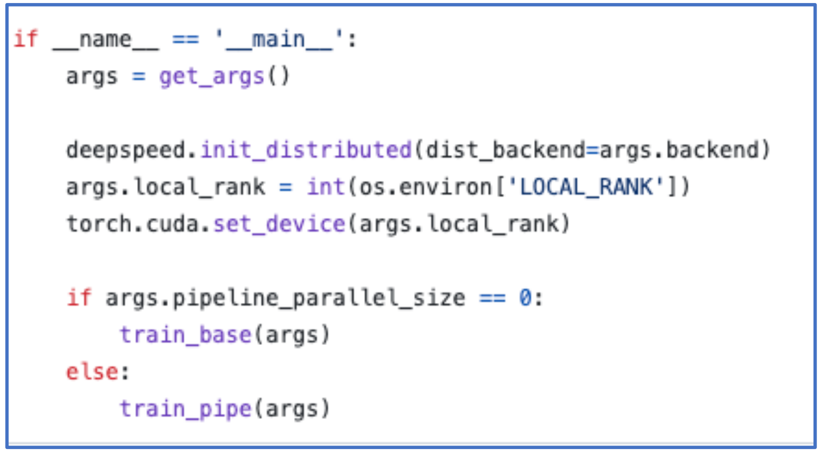
- Code: https://github.com/microsoft/DeepSpeedExamples/blob/master/training/pipeline_parallelism/train.py

Pipeline parallel training loop



```
def join_layers(vision_model):
    layers = [
        *vision_model.features,
        vision_model.avgpool,
        lambda x: torch.flatten(x, 1),
        *vision_model.classifier,
    ]
    return layers


def train_pipe(args, part='parameters'):
    torch.manual_seed(args.seed)
    deepspeed.runtime.utils.set_random_seed(args.seed)

    #
    # Build the model
    #

    # VGG also works :-)
    #net = vgg19(num_classes=10)
    net = AlexNet(num_classes=10)
    net = PipelineModule(layers=join_layers(net),
                         loss_fn=torch.nn.CrossEntropyLoss(),
                         num_stages=args.pipeline_parallel_size,
                         partition_method=part,
                         activation_checkpoint_interval=0)

    trainset = cifar_trainset(args.local_rank)

    engine, _, _, _ = deepspeed.initialize(
        args=args,
        model=net,
        model_parameters=[p for p in net.parameters() if p.requires_grad],
        training_data=trainset)

    for step in range(args.steps):
        loss = engine.train_batch()
```

Main entrance



```
if __name__ == '__main__':
    args = get_args()

    deepspeed.init_distributed(dist_backend=args.backend)
    args.local_rank = int(os.environ['LOCAL_RANK'])
    torch.cuda.set_device(args.local_rank)

    if args.pipeline_parallel_size == 0:
        train_base(args)
    else:
        train_pipe(args)
```

References

- https://dl.acm.org/doi/pdf/10.14778/3415478.3415530
- https://pytorch.org/tutorials/beginner/ddp_series_theory.html
- https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cp/ds3lab-dam/documents/ZipML.pdf
- https://pytorch.org/docs/stable/pipeline.html
- https://arxiv.org/abs/1811.06965
- https://arxiv.org/abs/1806.03377
- https://www.deepspeed.ai/tutorials/pipeline/