THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

RELAXED SYSTEM LAB

Pipeline Parallel Training

COMP4901Y

Binhang Yuan

Recall 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 <u>a complete copy of the model (parameters, gradients, and optimizer status)</u>. 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

RELAXED SYSTEM LAB

- 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

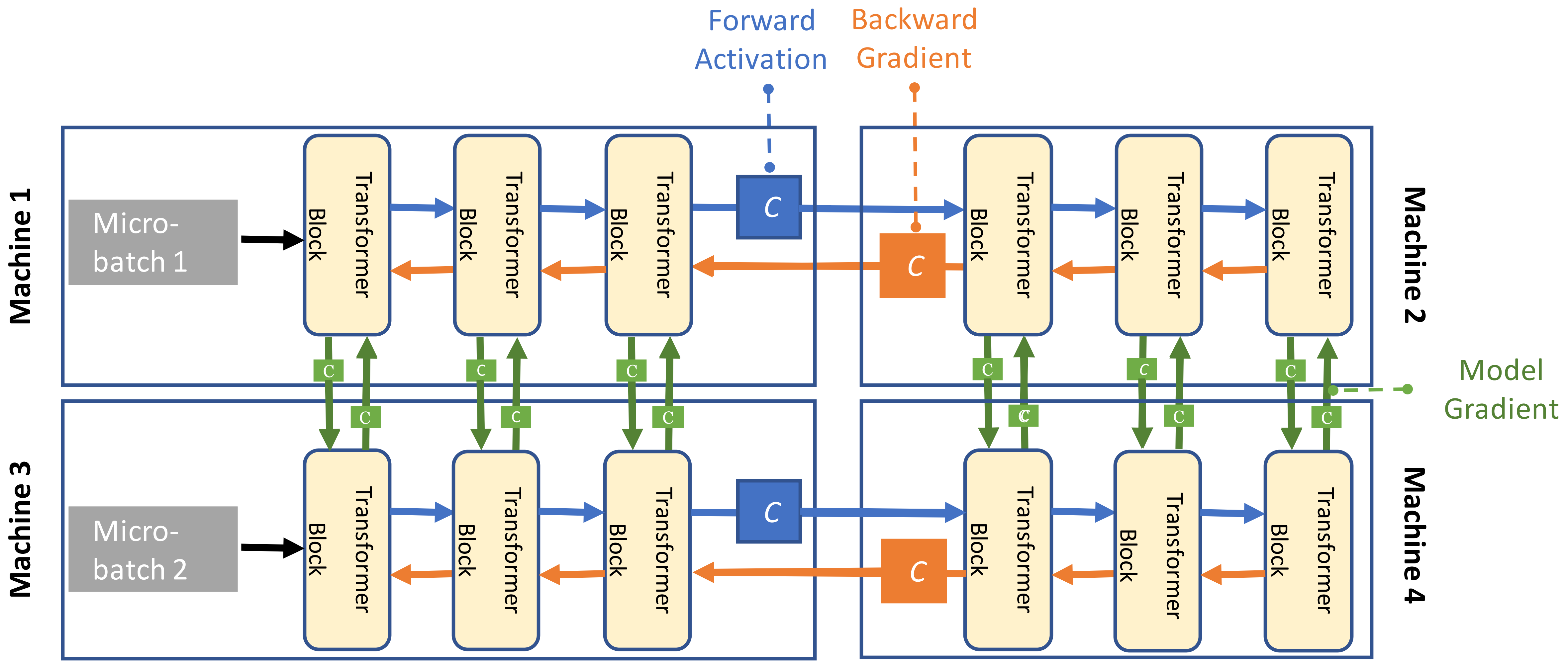
RELAXED SYSTEM LAB

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

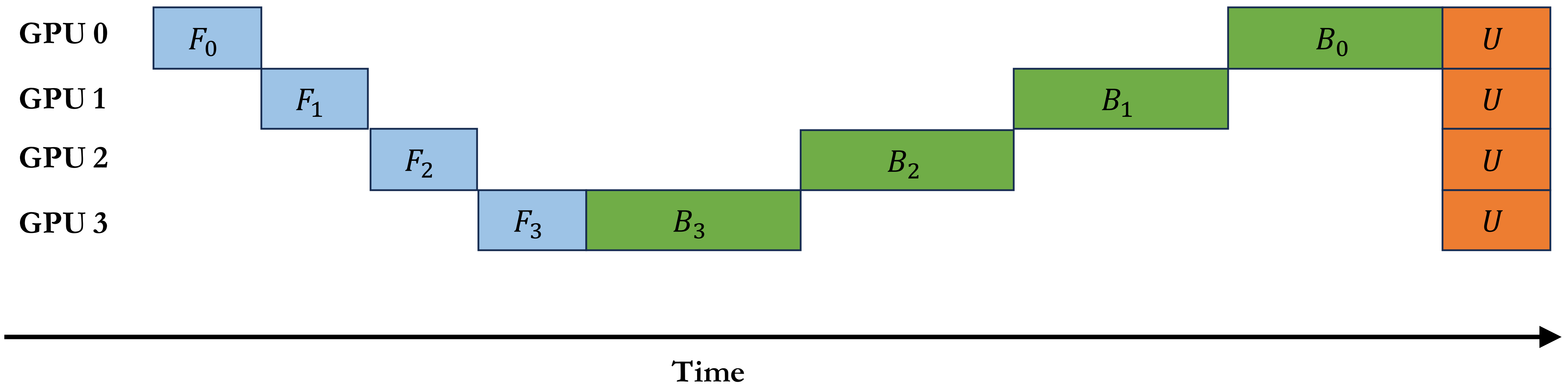
Forward Activation

Pipeline Parallel

RELAXED SYSTEM LAB



A Naïve Implementation



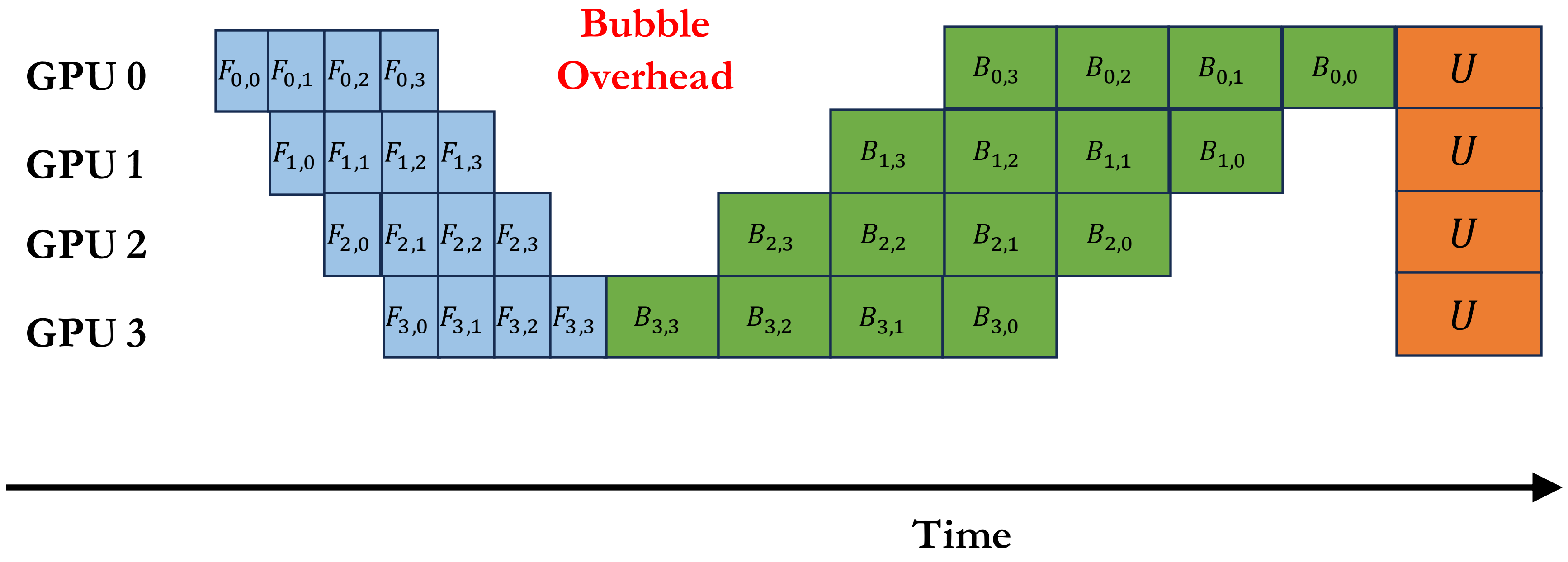
The number on each block represents the stage index.

Pipeline Parallel Optimization

Pipeline Parallelism Optimization

- 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

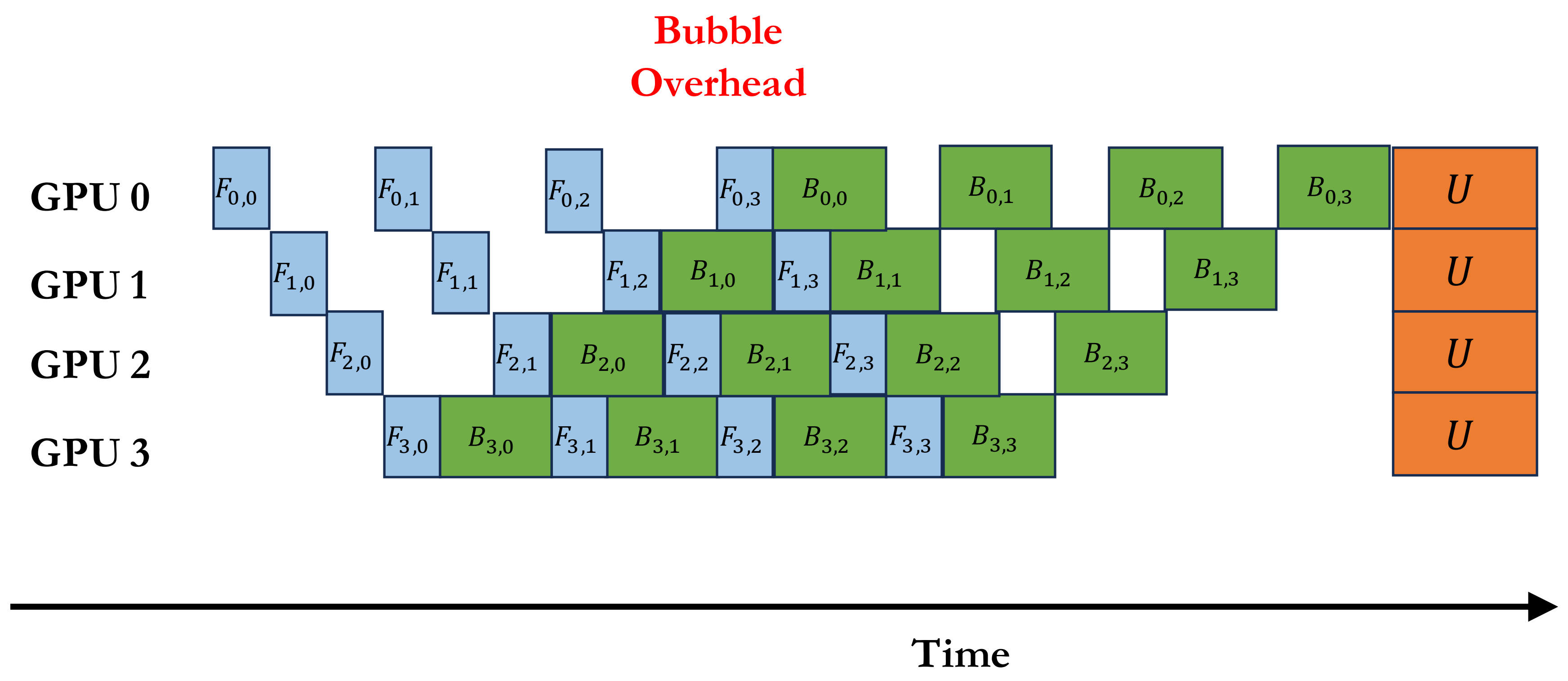


Bubble Overhead
GPU 0
GPU 1
GPU 2
GPU 3
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$
$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$
$B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$
$B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$
$B_{2,3}$ $B_{2,2}$ $B_{2,1}$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

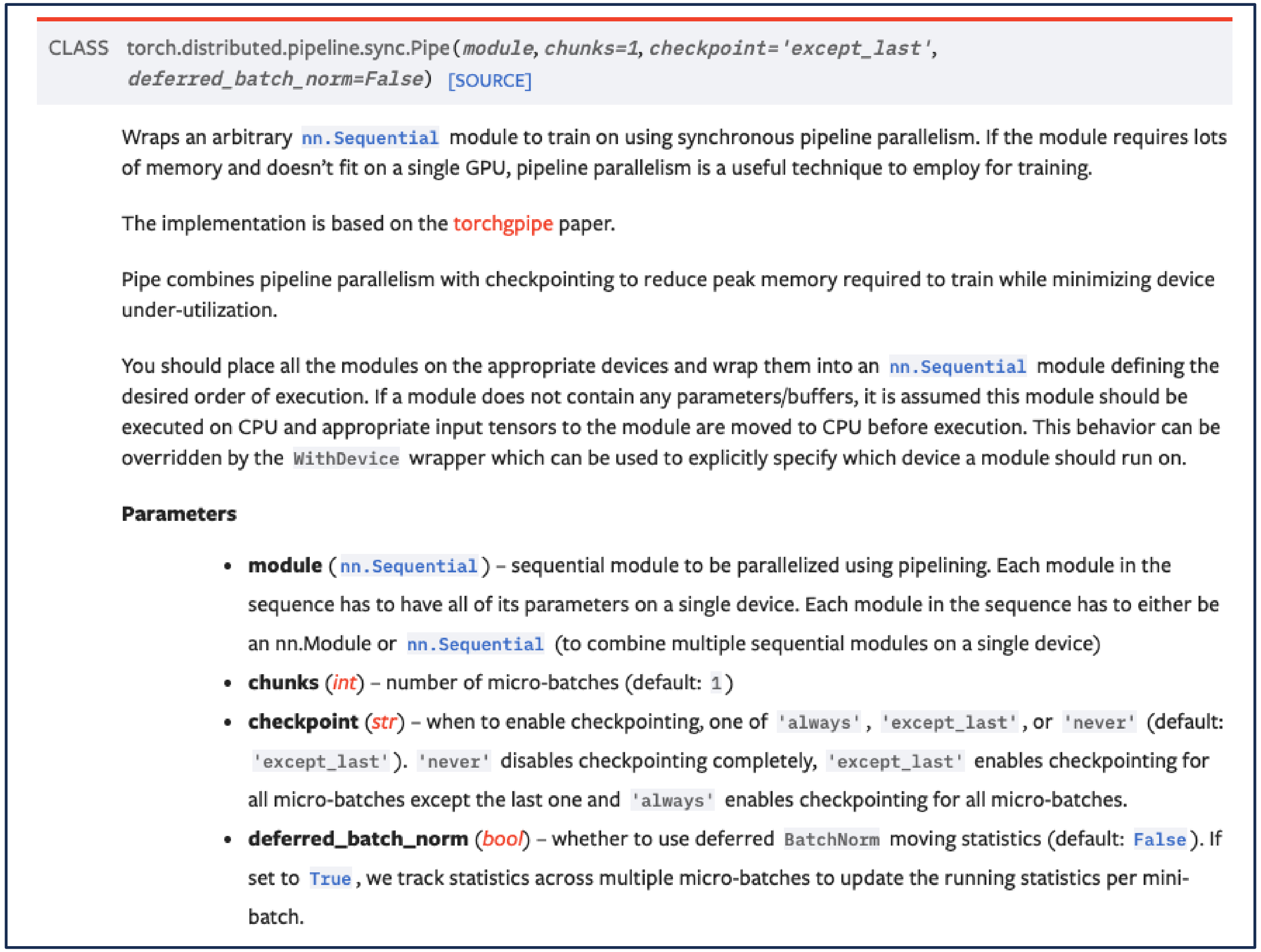

Bubble Overhead
GPU 0
GPU 1
GPU 2
GPU 3
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$ $B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ U
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $B_{1,0}$ $F_{1,3}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ U
$F_{2,0}$ $F_{2,1}$ $B_{2,0}$ $F_{2,2}$ $B_{2,1}$ $F_{2,3}$ $B_{2,2}$ $B_{2,3}$ U
$F_{3,0}$ $B_{3,0}$ $F_{3,1}$ $B_{3,1}$ $F_{3,2}$ $B_{3,2}$ $F_{3,3}$ $B_{3,3}$ 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? $\frac{K-1}{M+K-1}$

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.

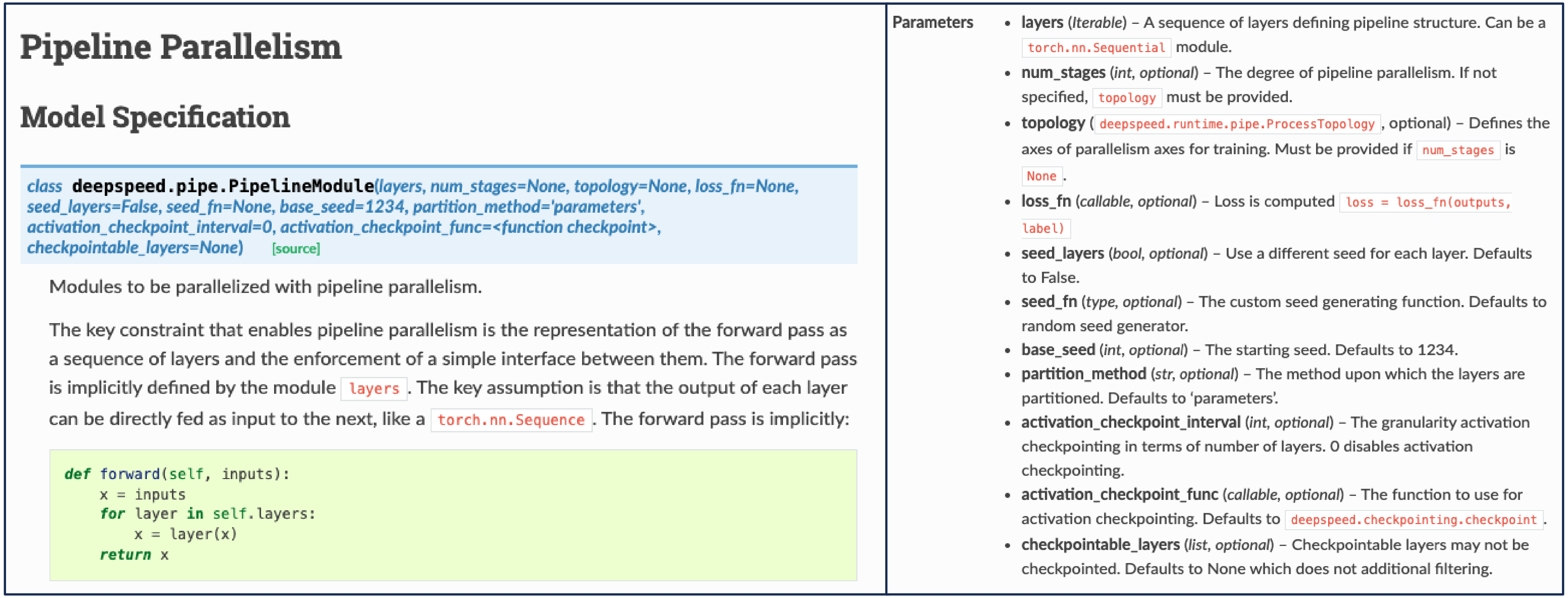
Parameters

- **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 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

RELAXED SYSTEM LAB

- 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

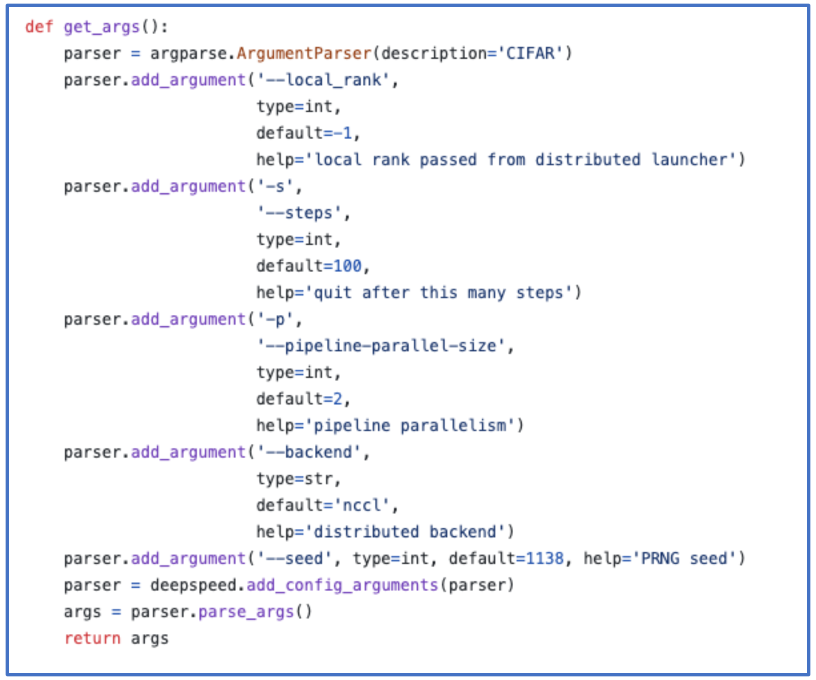
```python
def cifar_trainset(local_rank, dl_path='/tmp/cifar10-data'):
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])

    # 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

RELAXED SYSTEM LAB

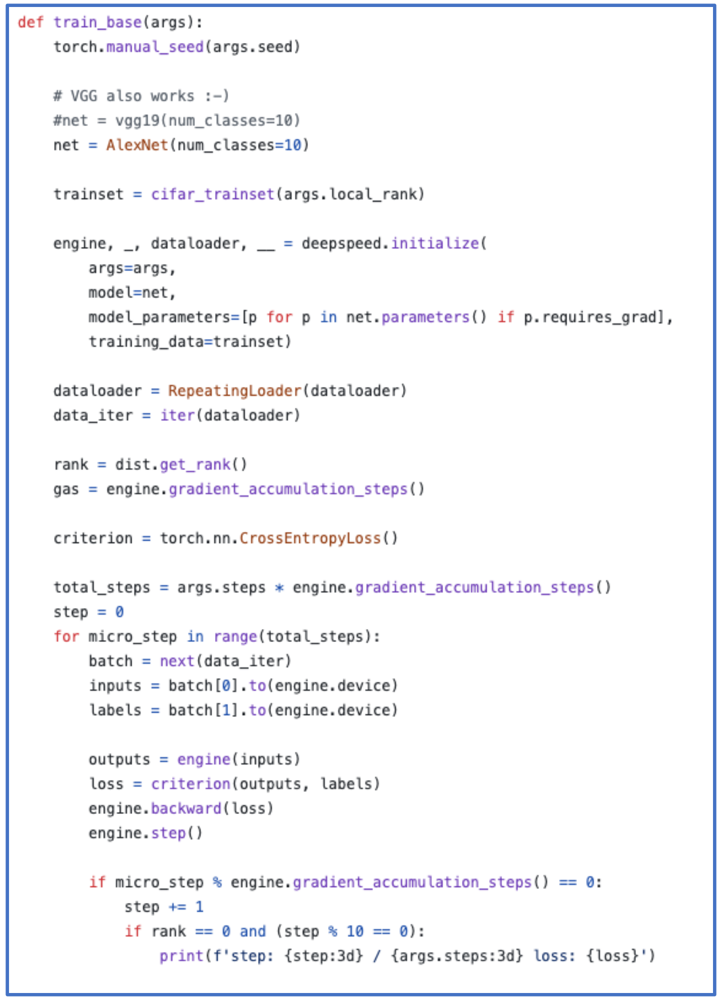
- Code: https://github.com/microsoft/DeepSpeedExamples/blob/master/training/pipeline_parallelism/train.py

Pass configurations



```python
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

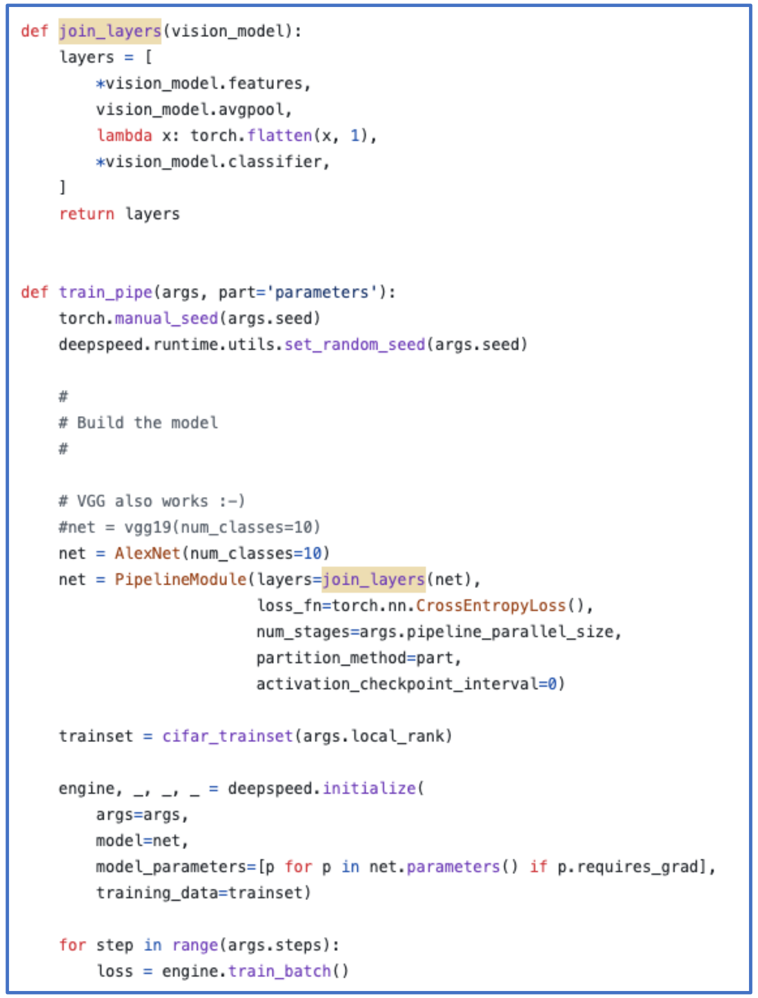


```python
def train_base(args):
    torch.manual_seed(args.seed)

    # 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

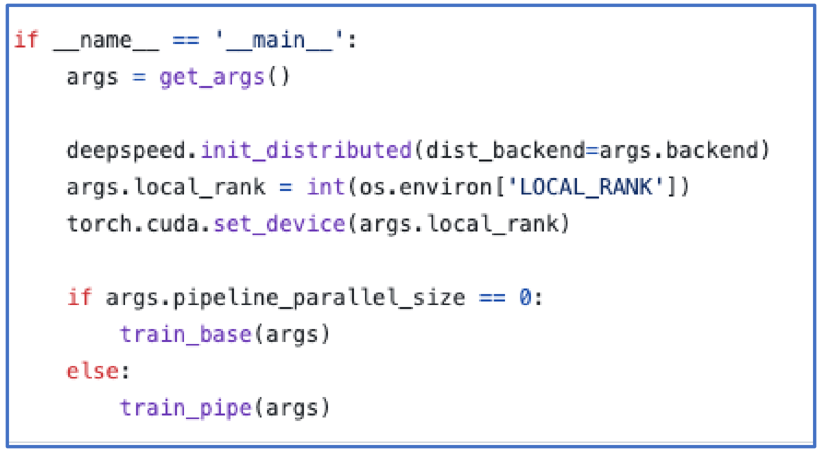
- Code: https://github.com/microsoft/DeepSpeedExamples/blob/master/training/pipeline_parallelism/train.py

Pipeline parallel training loop



```python
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



```python
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://pytorch.org/docs/stable/pipeline.html
- https://arxiv.org/abs/1811.06965
- https://arxiv.org/abs/1806.03377
- https://www.deepspeed.ai/tutorials/pipeline/