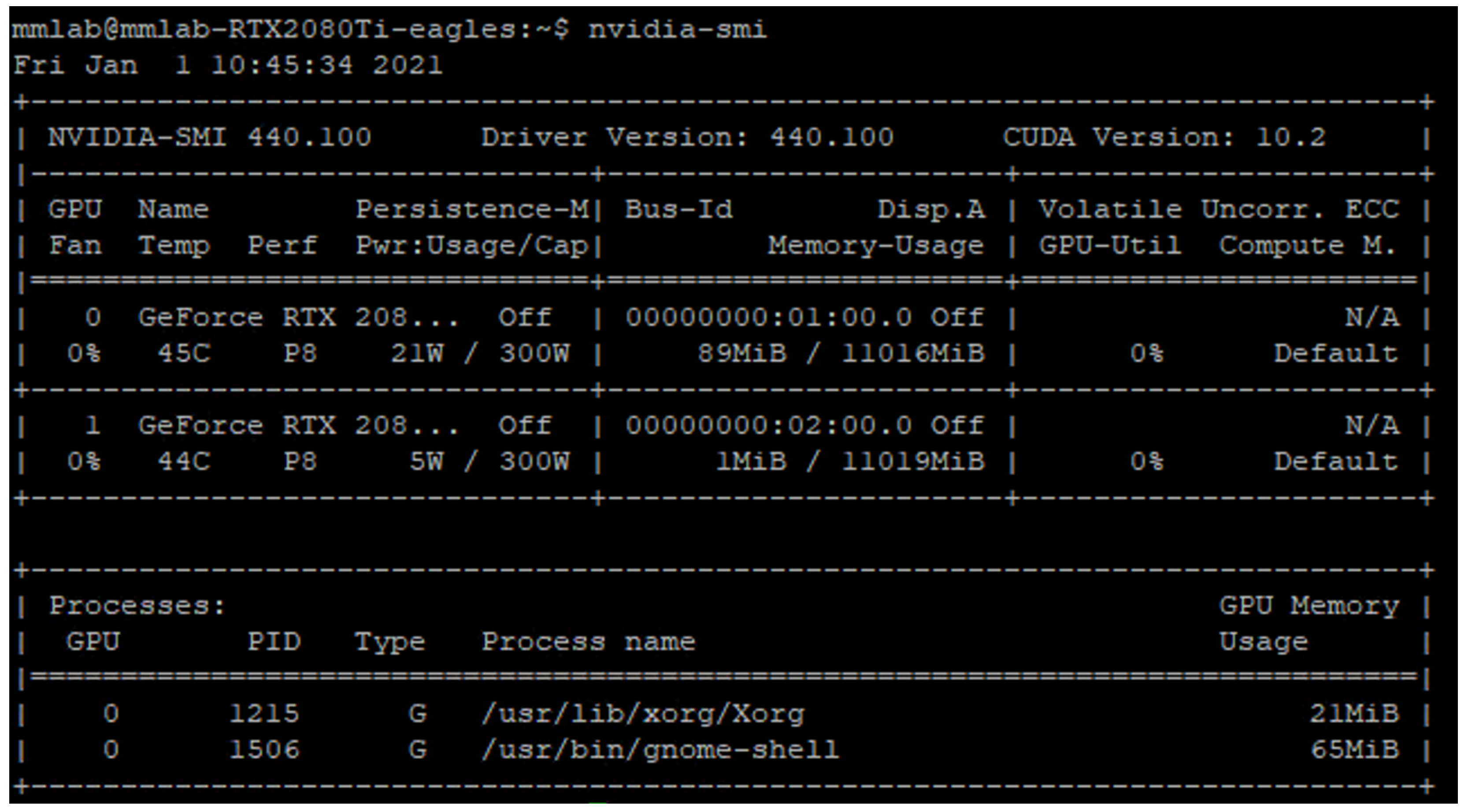
Dive into Deep Learning

Chapter 12. Computational Performance

Wayne L, Jan, 1, 2021

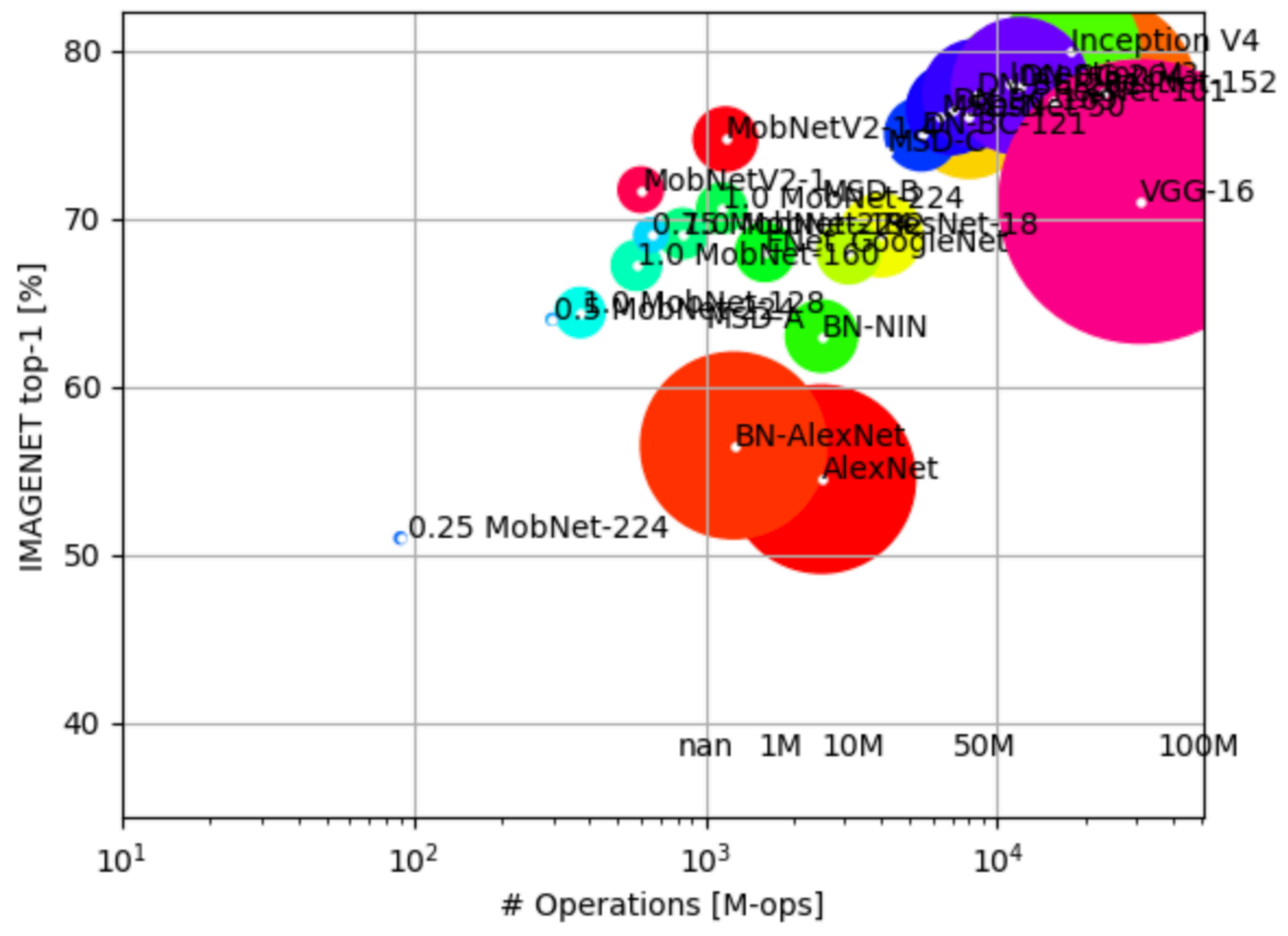
Training on Multiple GPUs

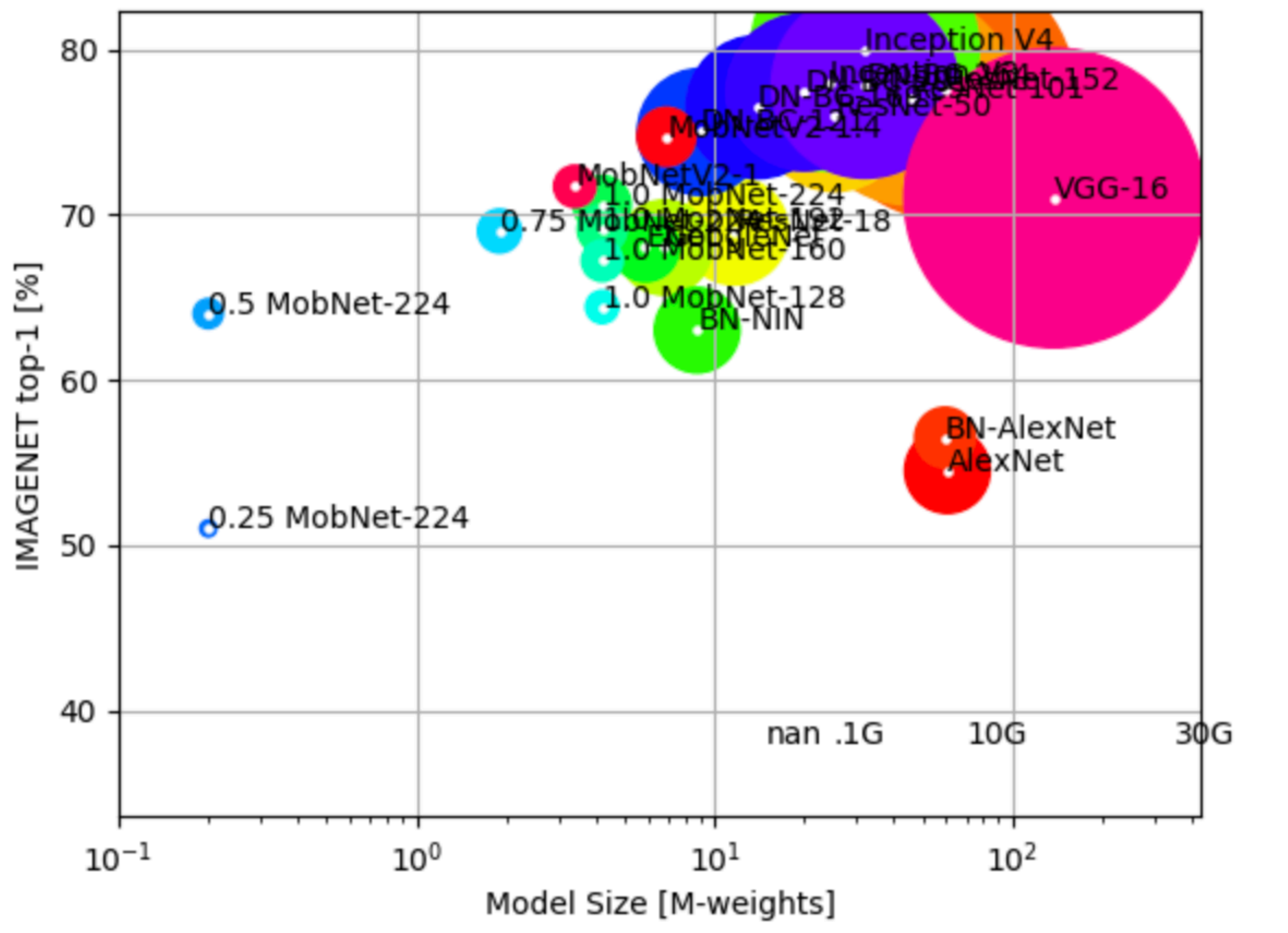
How to actually parallelize deep learning training



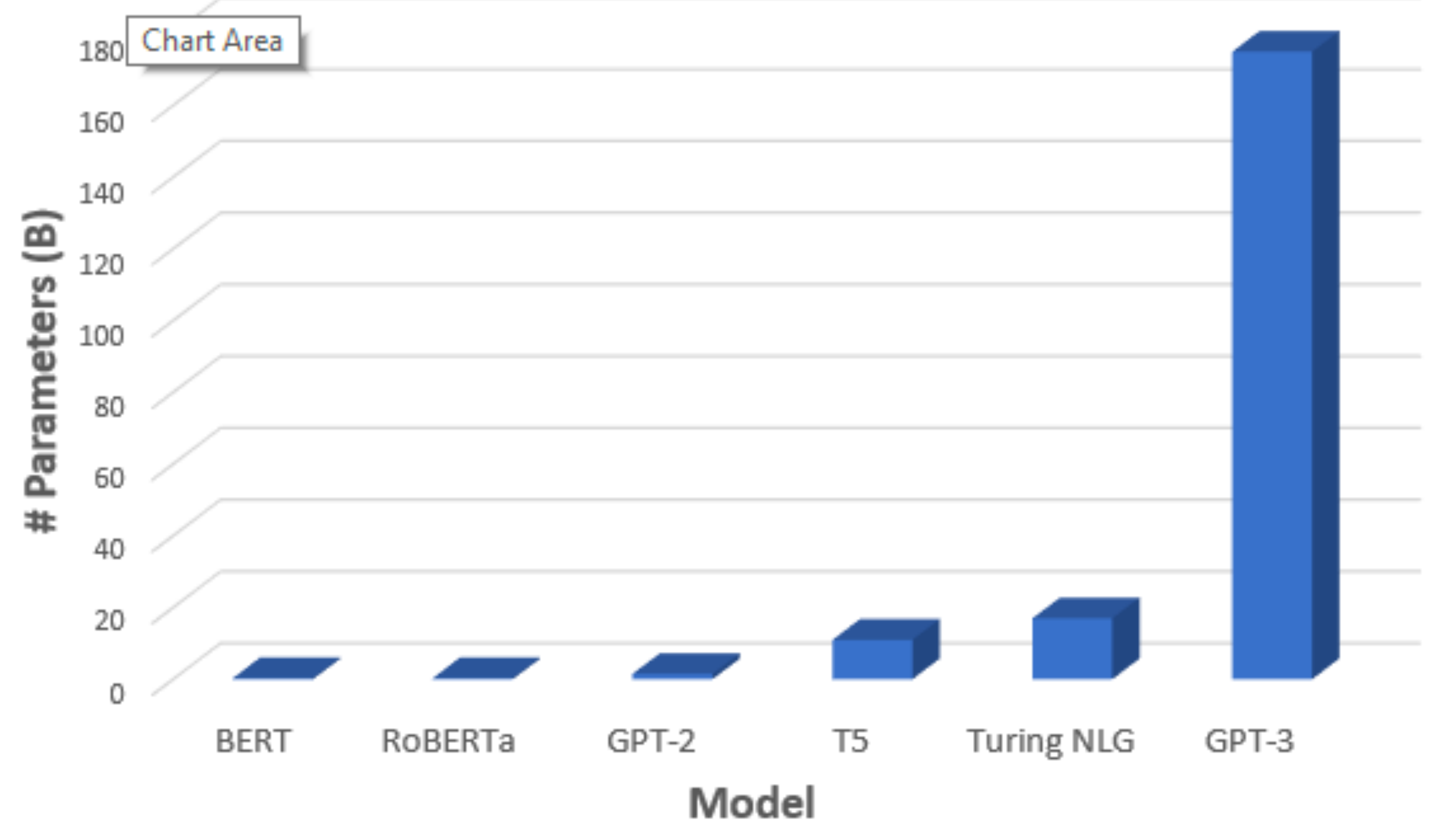
```
mmlab@mmlab-RTX2080Ti-eagles:~$ nvidia-smi
Fri Jan  1 10:45:34 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.100      Driver Version: 440.100      CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce RTX 208...  Off  | 00000000:01:00.0 Off |                  N/A |
| 0%   45C    P8    21W / 300W |     89MiB / 11016MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce RTX 208...  Off  | 00000000:02:00.0 Off |                  N/A |
| 0%   44C    P8     5W / 300W |      1MiB / 11019MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1215      G   /usr/lib/xorg/Xorg                            21MiB |
|    0      1506      G   /usr/bin/gnome-shell                          65MiB |
+-----------------------------------------------------------------------------+
```

Multiple GPUs and increased memory size

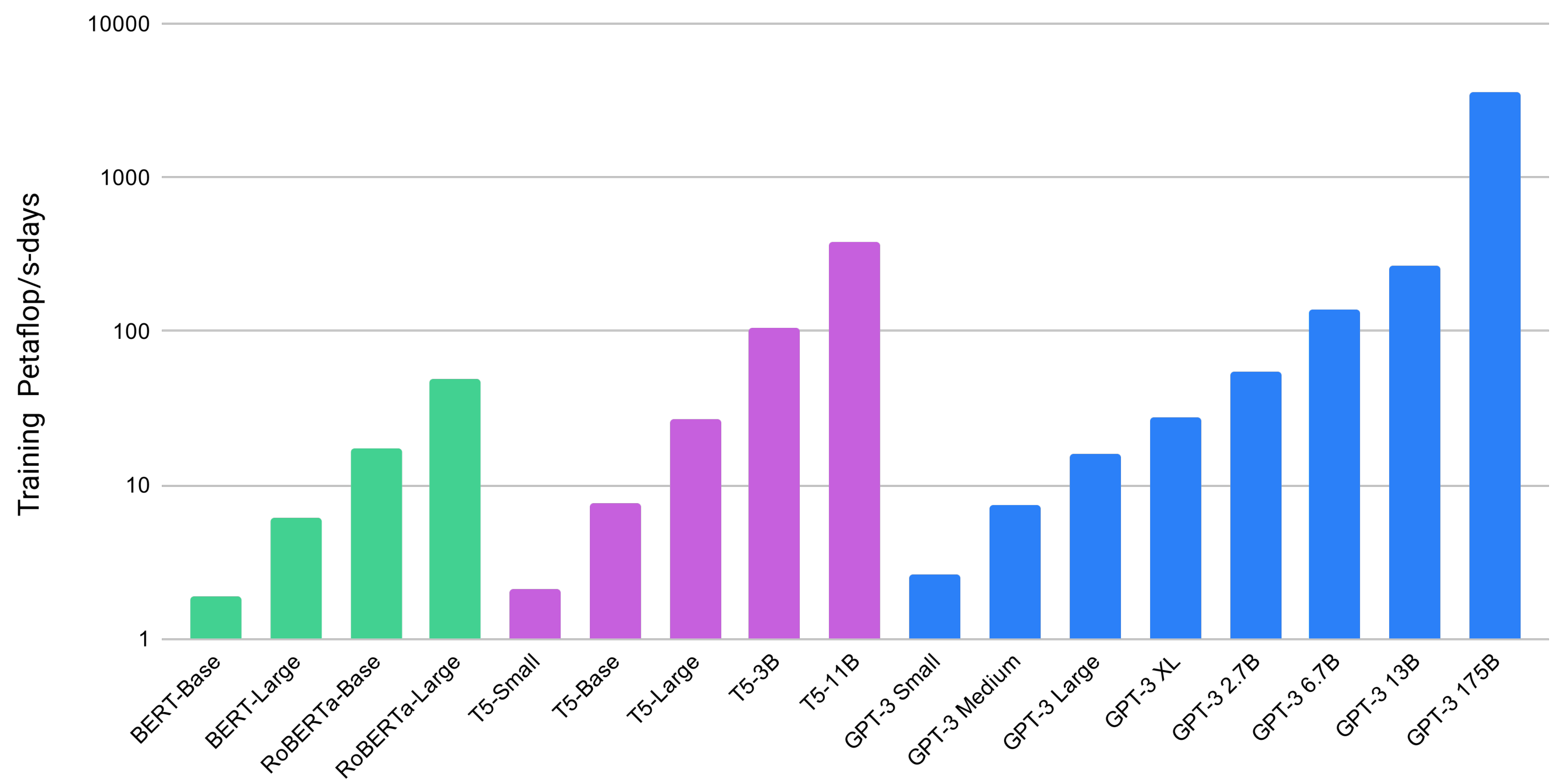




Computer vision



Total Compute Used During Training



NLP

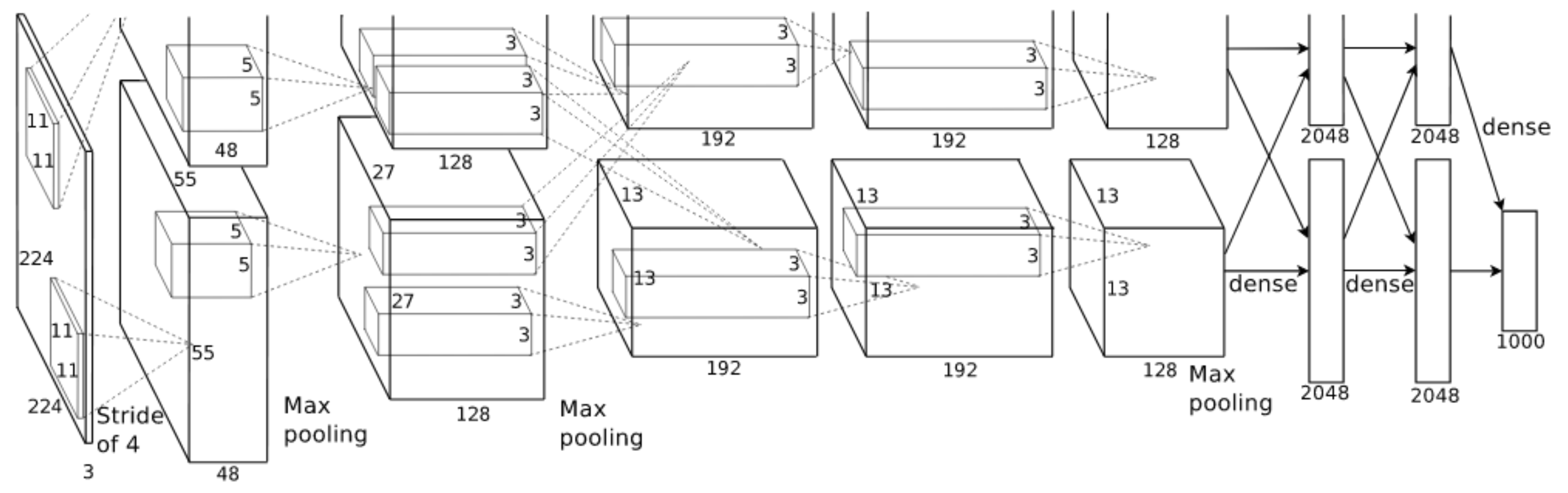
Training on Multiple GPUs

Partition the network layers across multiple GPUs

- Each GPU takes as input the data flowing into a particular layer, processes data across a number of subsequent layers and then **sends** the data to the next GPU.
- This allows us to **process** data with larger networks when compared to what a single GPU could handle.
- Memory footprint per GPU can be well controlled (it is a fraction of the total network footprint)
- The interface between layers (and thus GPUs) **requires** tight synchronization. This can be tricky, in particular if the **computational workloads** are not properly matched between layers. The problem is exacerbated for large numbers of GPUs.
- The interface between layers **requires** large amounts of data transfer (activations, gradients). This may overwhelm the **bandwidth** of the GPU buses.
- We do not recommend it unless there is **excellent framework / OS support** for chaining together multiple GPUs.

Training on Multiple GPUs

Split the work required by individual layers



very small memory footprint (<=2GB)

- Rather than computing 64 channels on a single GPU we could **split up the problem** across 4 GPUs, each of which generate data for 16 channels.
- This allows for good scaling in terms of computation, provided that the number of channels (or neurons) is not too small.
- Multiple GPUs can process **increasingly larger networks** since the memory available scales linearly.
- We need a *very large* number of **synchronization / barrier operations** since each layer depends on the results from all other layers.
- The amount of data that needs to be transferred is potentially even larger than when distributing layers across GPUs. We do not recommend this approach due to its bandwidth cost and complexity.

Training on Multiple GPUs

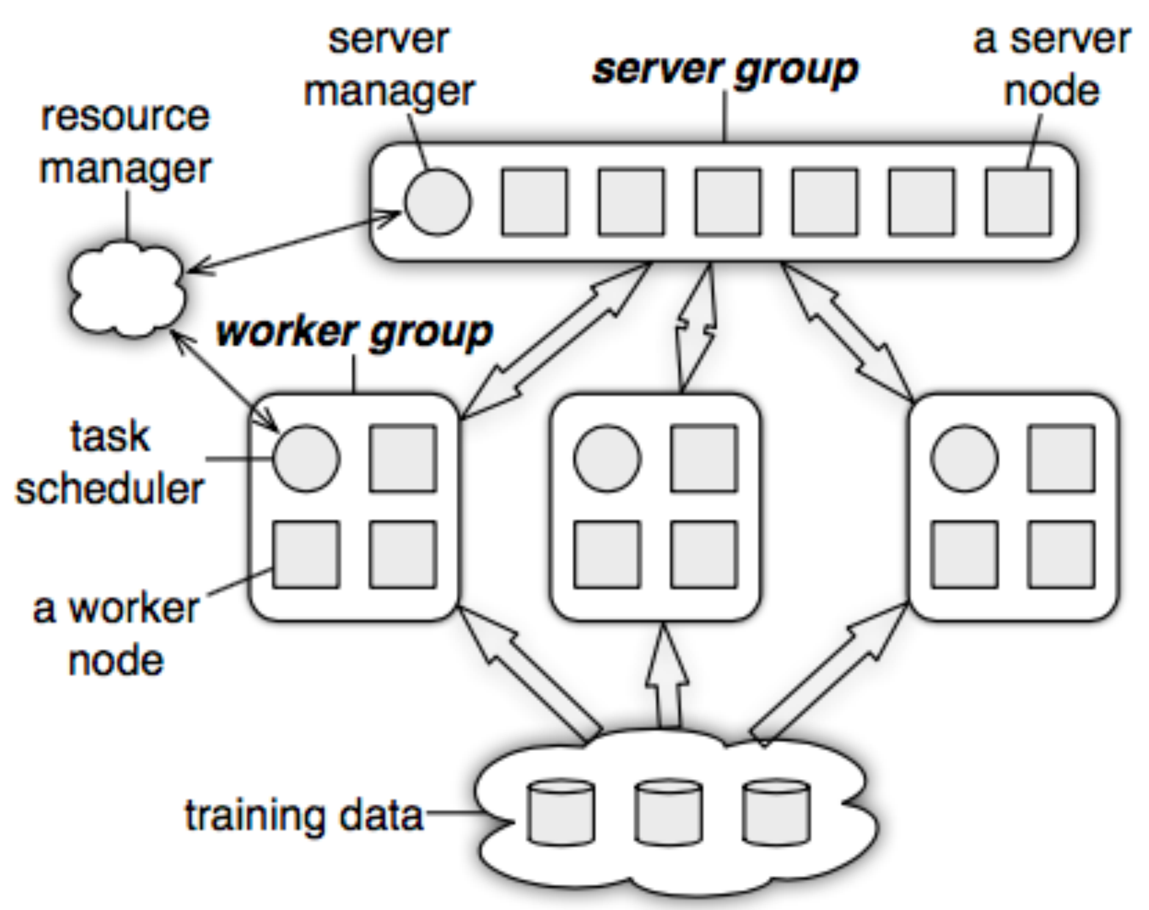
Partition data across multiple GPUs

- This way all GPUs perform the same type of work and gradients are **aggregated** between GPUs after each mini-batch.
- This is the simplest approach and it can be applied in **any situation**.
- Adding more GPUs does not allow us to train larger models.
- We only need to **synchronize** after each mini-batch. That said, it is highly desirable to start exchanging gradients parameters already while others are still being computed.
- Large numbers of GPUs lead to very large mini-batch sizes, thus reducing training efficiency.

GPU memory used to be a problem in the early days

Parameter server

- Scaling distributed machine learning with the parameter server.
  Link: https://www.cs.cmu.edu/~muli/file/parameter_server_osdi14.pdf
- Proposed a parameter server framework for distributed machine learning problems.
- Manages asynchronous data communication between nodes, and supports flexible consistency models, elastic scalability, and continuous fault tolerance



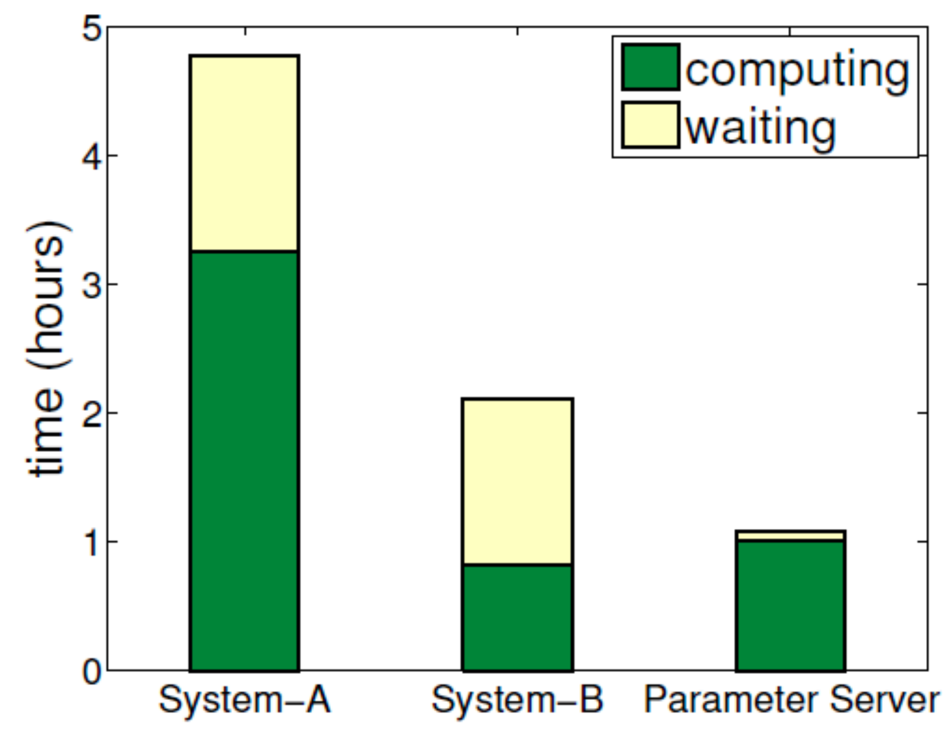


Figure 10: Time per worker spent on computation and waiting during sparse logistic regression.

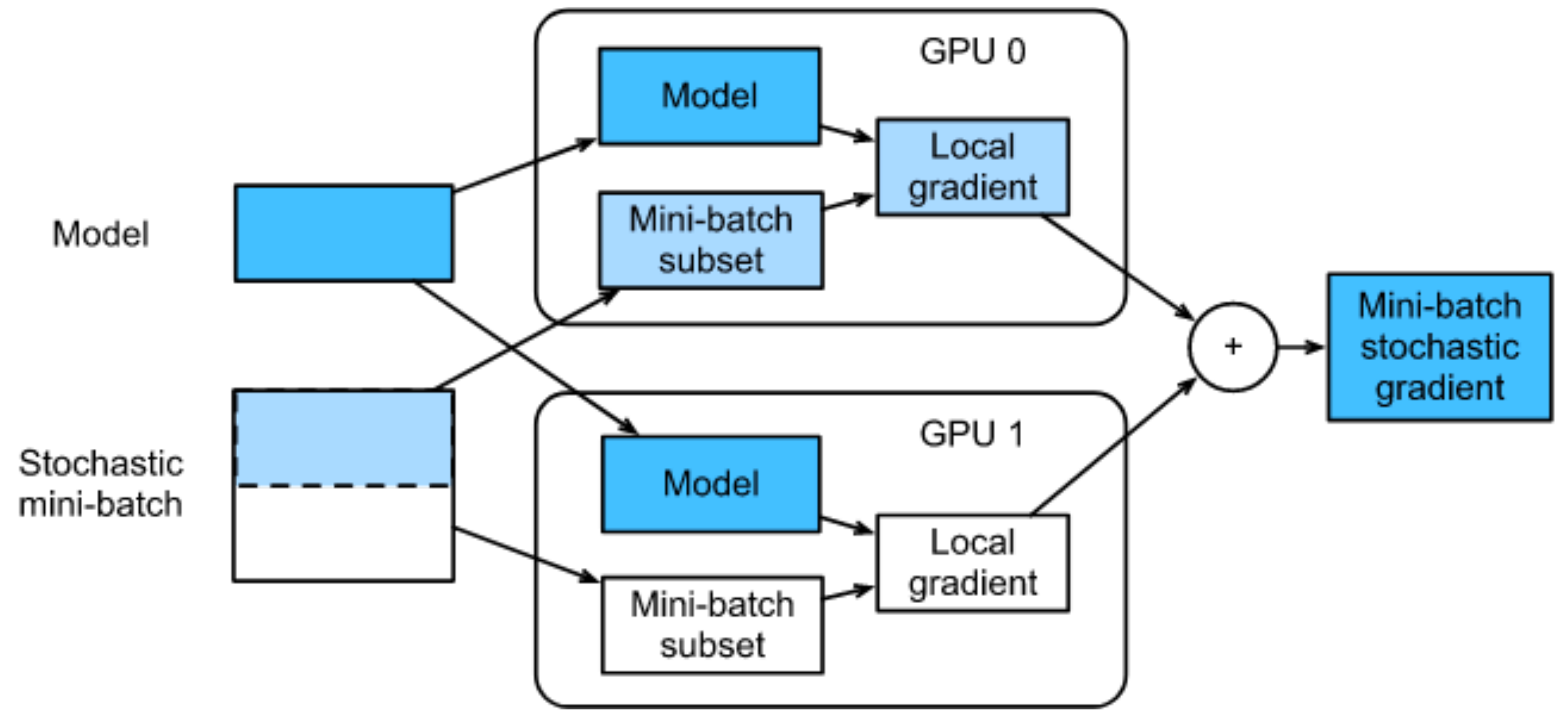
Data Parallelism

Data flow (1)

- In any iteration of training, given a random mini-batch, we split the examples in the batch into K portions and distribute them evenly across the GPUs.
- Each GPU calculates loss and gradient of the model parameters based on the mini-batch subset it was assigned and the model parameters it maintains.
- The local gradients of each of the K GPUs are aggregated to obtain the current mini-batch stochastic gradient.
- The aggregate gradient is re-distributed to each GPU.
- Each GPU uses this mini-batch stochastic gradient to update the complete set of model parameters that it maintains.

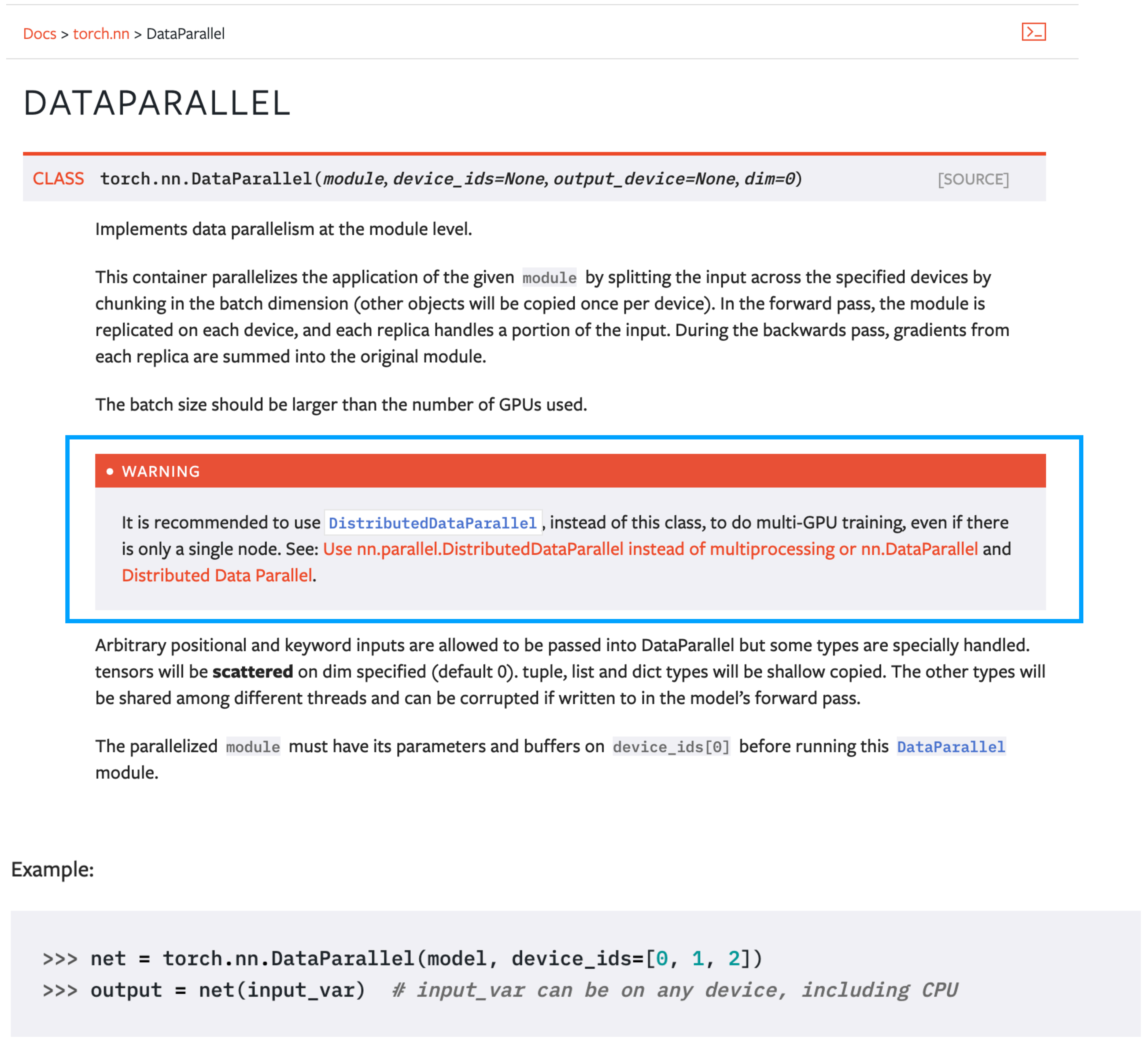
Data Parallelism

Data flow (2)



- Scatter
- Replicate
- Apply (Parallel)
- Gather

https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html



Docs > torch.nn > DataParallel

DATAPARALLEL

CLASS `torch.nn.DataParallel(module, device_ids=None, output_device=None, dim=0)` [SOURCE]

Implements data parallelism at the module level.

This container parallelizes the application of the given `module` by splitting the input across the specified devices by chunking in the batch dimension (other objects will be copied once per device). In the forward pass, the module is replicated on each device, and each replica handles a portion of the input. During the backwards pass, gradients from each replica are summed into the original module.

The batch size should be larger than the number of GPUs used.

• WARNING

It is recommended to use `DistributedDataParallel`, instead of this class, to do multi-GPU training, even if there is only a single node. See: Use nn.parallel.DistributedDataParallel instead of multiprocessing or nn.DataParallel and Distributed Data Parallel.

Arbitrary positional and keyword inputs are allowed to be passed into DataParallel but some types are specially handled. tensors will be **scattered** on dim specified (default 0). tuple, list and dict types will be shallow copied. The other types will be shared among different threads and can be corrupted if written to in the model's forward pass.

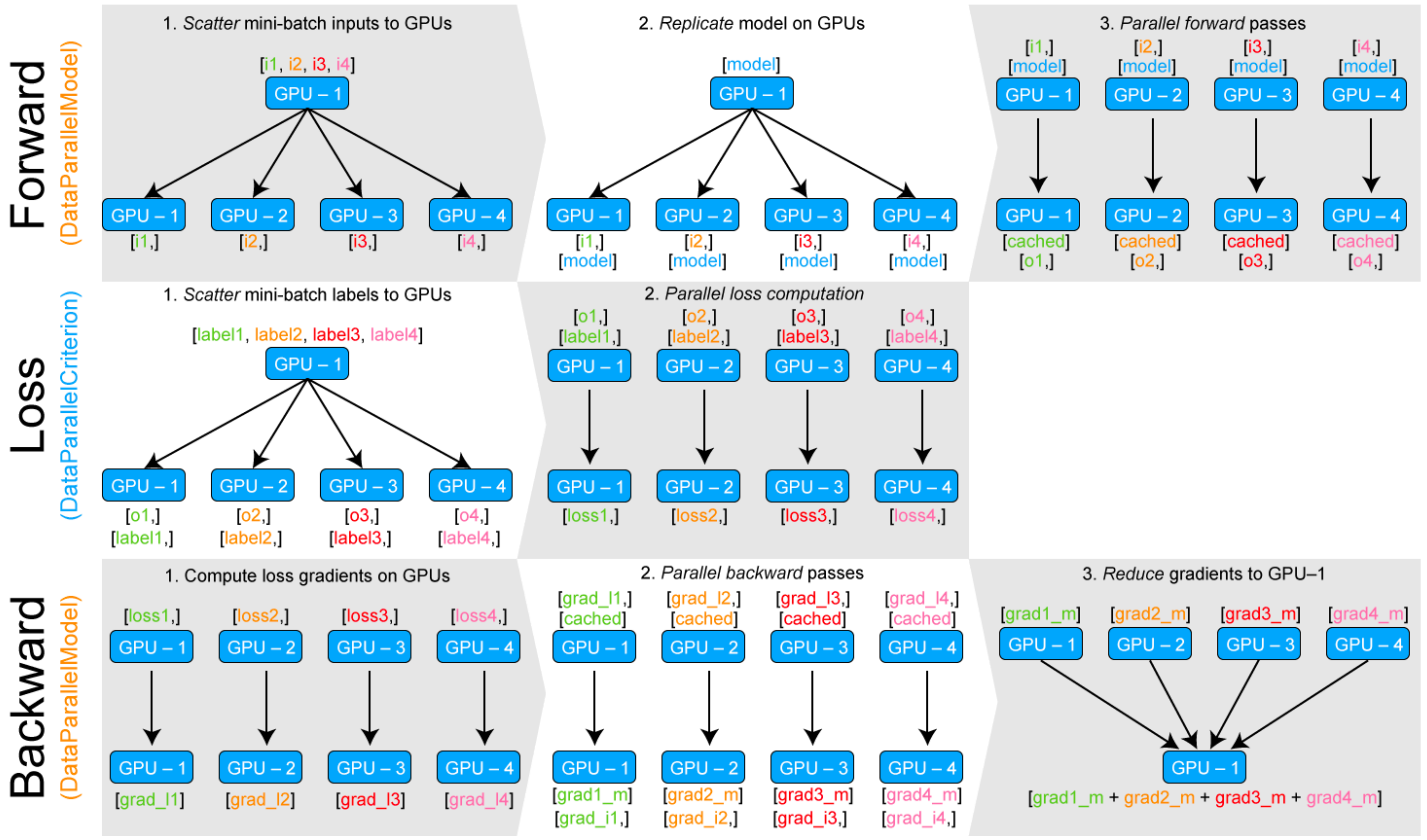
The parallelized `module` must have its parameters and buffers on `device_ids[0]` before running this `DataParallel` module.

Example:

```
>>> net = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
>>> output = net(input_var)  # input_var can be on any device, including CPU
```

Data Parallelism

Distributed data parallelism (1)



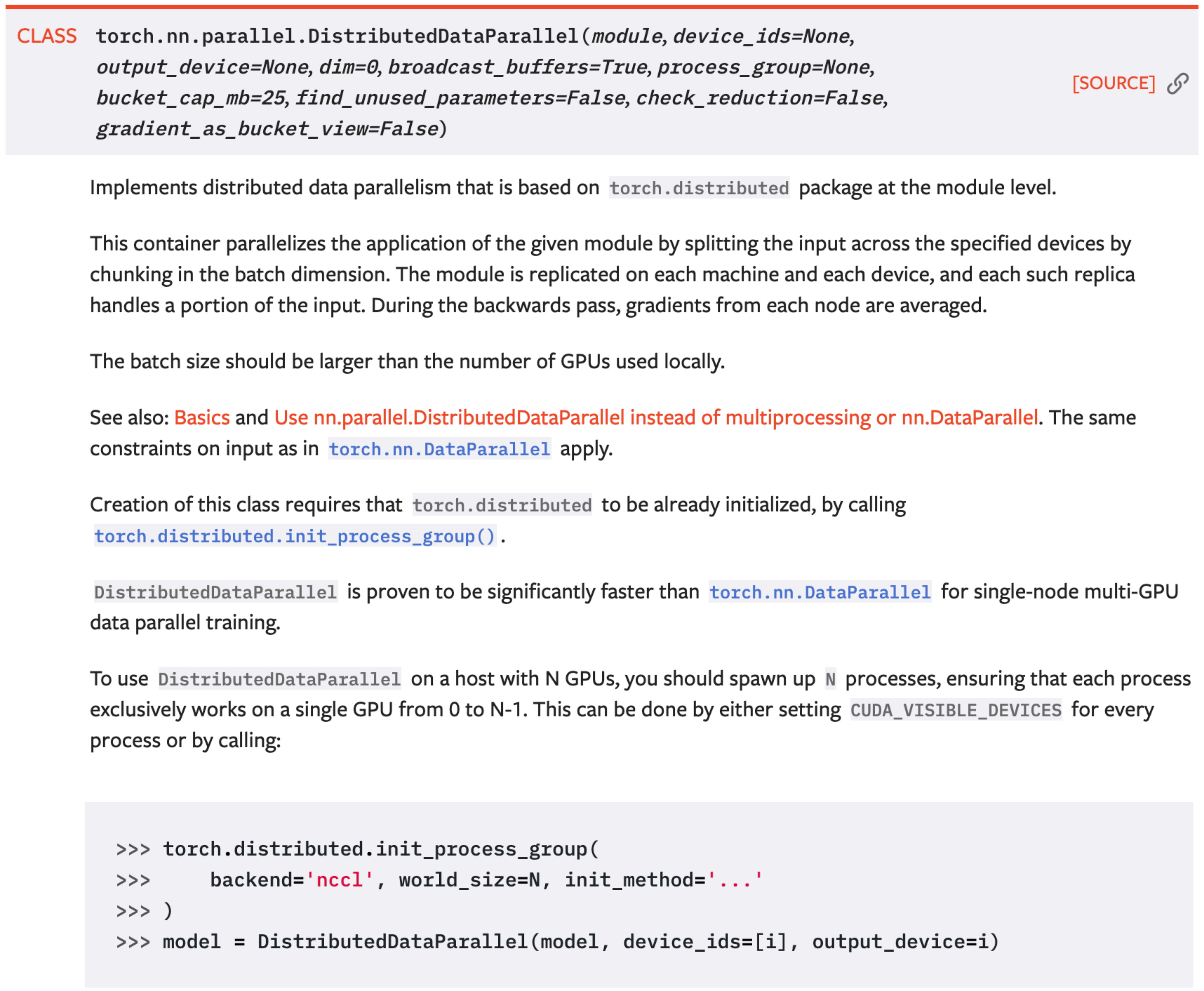
Forward
(DataParallelModel)
1. Scatter mini-batch inputs to GPUs
[i1, i2, i3, i4]
GPU – 1
GPU – 1 GPU – 2 GPU – 3 GPU – 4
[i1,] [i2,] [i3,] [i4,]
2. Replicate model on GPUs
[model]
GPU – 1
GPU – 1 GPU – 2 GPU – 3 GPU – 4
[i1,] [i2,] [i3,] [i4,]
[model] [model] [model] [model]
3. Parallel forward passes
[i1,] [i2,] [i3,] [i4,]
[model] [model] [model] [model]
GPU – 1 GPU – 2 GPU – 3 GPU – 4
[cached] [cached] [cached] [cached]
[o1,] [o2,] [o3,] [o4,]
Loss
(DataParallelCriterion)
1. Scatter mini-batch labels to GPUs
[label1, label2, label3, label4]
GPU – 1
GPU – 1 GPU – 2 GPU – 3 GPU – 4
[o1,] [o2,] [o3,] [o4,]
[label1,] [label2,] [label3,] [label4,]
2. Parallel loss computation
[o1,] [o2,] [o3,] [o4,]
[label1,] [label2,] [label3,] [label4,]
GPU – 1 GPU – 2 GPU – 3 GPU – 4
[loss1,] [loss2,] [loss3,] [loss4,]
Backward
(DataParallelModel)
1. Compute loss gradients on GPUs
[loss1,] [loss2,] [loss3,] [loss4,]
GPU – 1 GPU – 2 GPU – 3 GPU – 4
[grad_l1] [grad_l2] [grad_l3] [grad_l4]
2. Parallel backward passes
[grad_l1,] [grad_l2,] [grad_l3,] [grad_l4,]
[cached] [cached] [cached] [cached]
GPU – 1 GPU – 2 GPU – 3 GPU – 4
[grad1_m] [grad2_m] [grad3_m] [grad4_m]
[grad_i1,] [grad_i2,] [grad_i3,] [grad_i4,]
3. Reduce gradients to GPU–1
[grad1_m] [grad2_m] [grad3_m] [grad4_m]
GPU – 1 GPU – 2 GPU – 3 GPU – 4
GPU – 1
[grad1_m + grad2_m + grad3_m + grad4_m]

https://medium.com/huggingface/training-larger-batches-practical-tips-on-1-gpu-multi-gpu-distributed-setups-ec88c3e51255

Data Parallelism

Distributed data parallelism (2)

https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html#torch.nn.parallel.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)` [SOURCE]

Implements distributed data parallelism that is based on `torch.distributed` package at the module level.

This container parallelizes the application of the given module by splitting the input across the specified devices by chunking in the batch dimension. The module is replicated on each machine and each device, and each such replica handles a portion of the input. During the backwards pass, gradients from each node are averaged.

The batch size should be larger than the number of GPUs used locally.

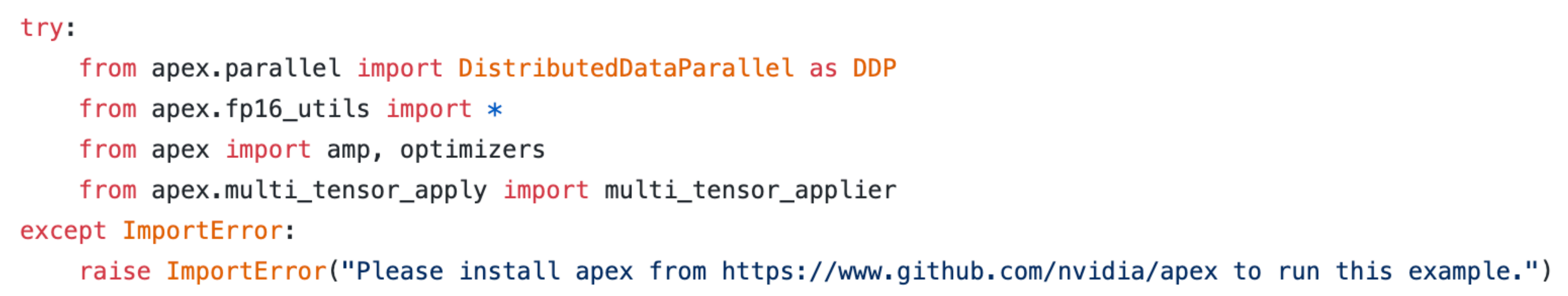
See also: Basics and Use nn.parallel.DistributedDataParallel instead of multiprocessing or nn.DataParallel. The same constraints on input as in `torch.nn.DataParallel` apply.

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

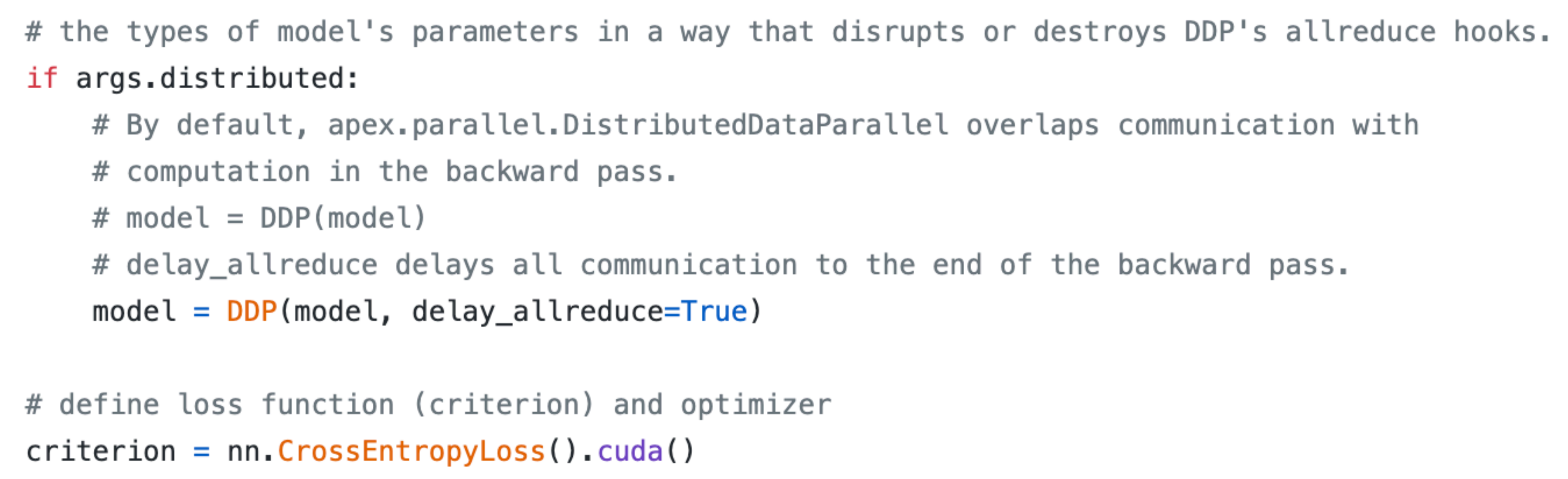
`DistributedDataParallel` is proven to be significantly faster than `torch.nn.DataParallel` for single-node multi-GPU data parallel training.

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



```
try:
    from apex.parallel import DistributedDataParallel as DDP
    from apex.fp16_utils import *
    from apex import amp, optimizers
    from apex.multi_tensor_apply import multi_tensor_applier
except ImportError:
    raise ImportError("Please install apex from https://www.github.com/nvidia/apex to run this example.")
```



```
    # the types of model's parameters in a way that disrupts or destroys DDP's allreduce hooks.
    if args.distributed:
        # By default, apex.parallel.DistributedDataParallel overlaps communication with
        # computation in the backward pass.
        # model = DDP(model)
        # delay_allreduce delays all communication to the end of the backward pass.
        model = DDP(model, delay_allreduce=True)

    # define loss function (criterion) and optimizer
    criterion = nn.CrossEntropyLoss().cuda()
```

https://github.com/NVIDIA/apex/blob/master/examples/imagenet/main_amp.py