Fundamentals of Accelerated Neural Network Training with Multi-GPUs on HiPerGator-AI

Yunchao Yang, PhD

AI support team

UF Research Computing

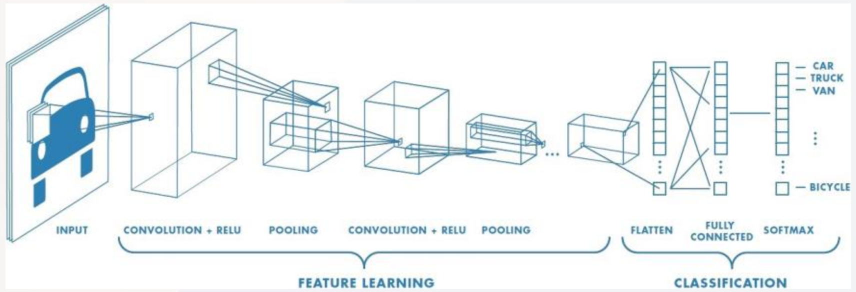
yunchyang@ufl.edu

The slides and demo code will be hosted on
https://github.com/YunchaoYang/BoF-MultiGPUTutorial

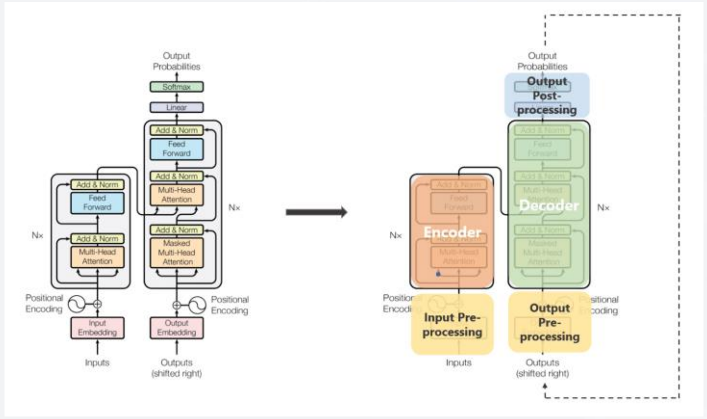
UF | Information Technology
UNIVERSITY of FLORIDA

Deep Neural Networks is transforming everywhere

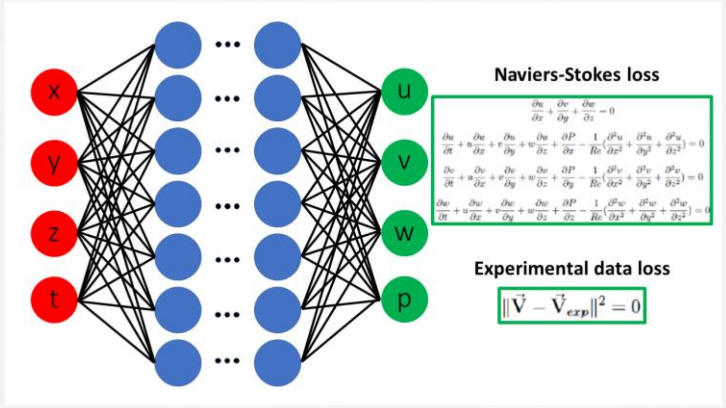
Convolutional Neural Networks In Computer Vision



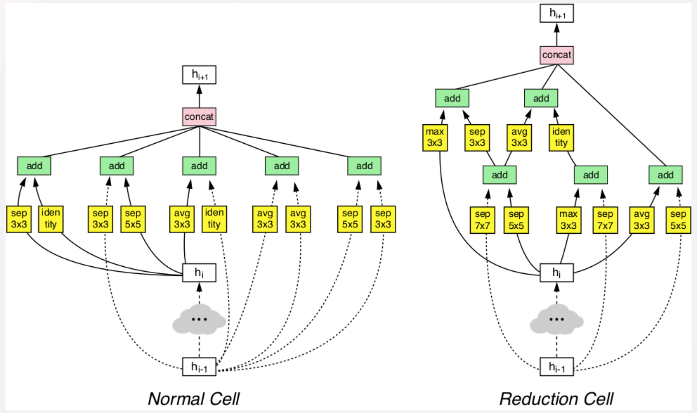
Transformers in NLP



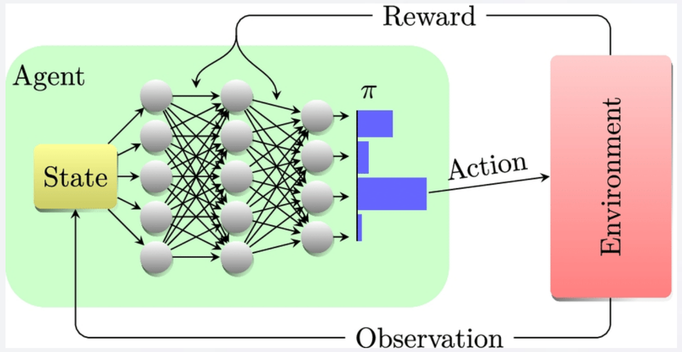
Physics-informed Machine Learning



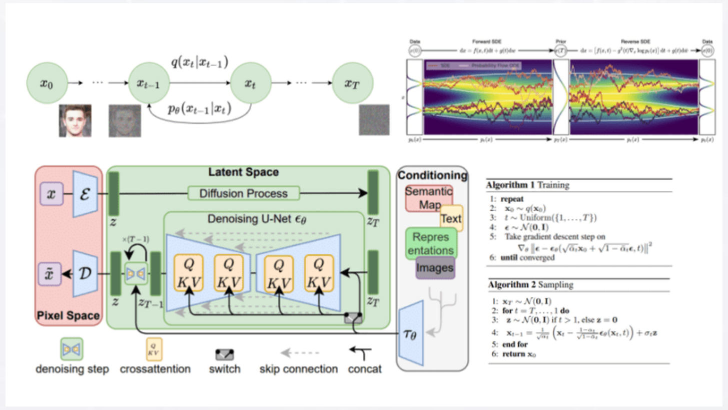
Neural Architecture Search



Reinforcement Learning



Diffusion models



Neural Architecture Search

Challenges for accelerating neural networks training

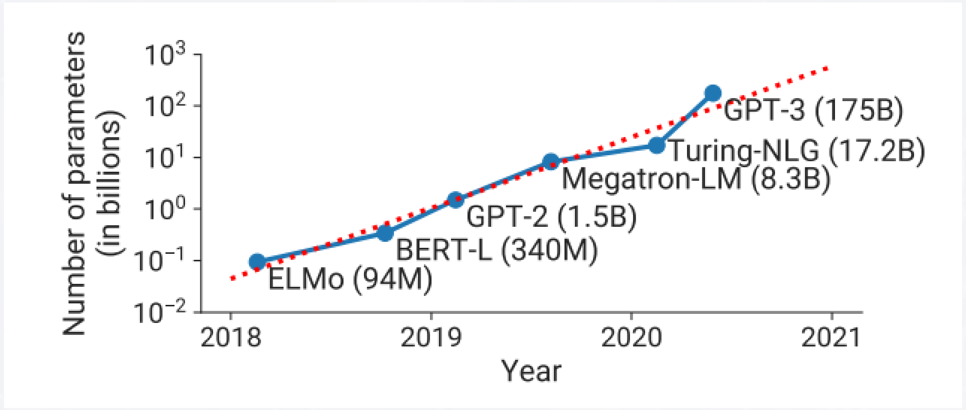
Neural networks are getting bigger and bigger to trillions of parameters.

GPT-3 has 175,000,000,000 parameters, at least 800 GB memory to load the model

- Memory requirement
  - Huge parameter space
  - large batch size
- Computationally intensive
  - High FLOPs on training
    - Massive amount of training dataset (millions of images)
    - Large number of parameters – (GPT-3, 175 billion trainable weights)
  - Inference speed

Large Neural Networks

- Deep learning models are getting bigger and bigger:
  - **GPT-3: 175 billion parameters**
  - **Megatron-Turing: 530 billion parameters**

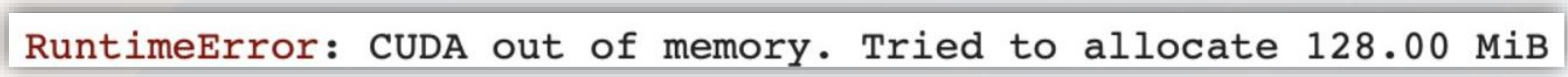


Model	**# parameters**	Memory requirement for parameters (FP32)	**type**
BERT-L	345 m	~ 2 GB	NLP, transformer
GPT-3	175 b	~ 800 GB	NLP, transformer
Megatron-Turing	530 b	~ 2 TB	Natural Language Generation
SEER(SElf-SupERvised)	10 b	~ 40 GB	computer vision

can only scale using model parallel training

single GPU memory on HPG-AI: an A100 has 80 GB vRAM

CUDA Out Of Memory (OOM) error



```
RuntimeError: CUDA out of memory. Tried to allocate 128.00 MiB
```

- OOM (Out Of Memory) errors can occur when building and training a neural network model on the GPU. The size of the model is limited by the available memory on the GPU.

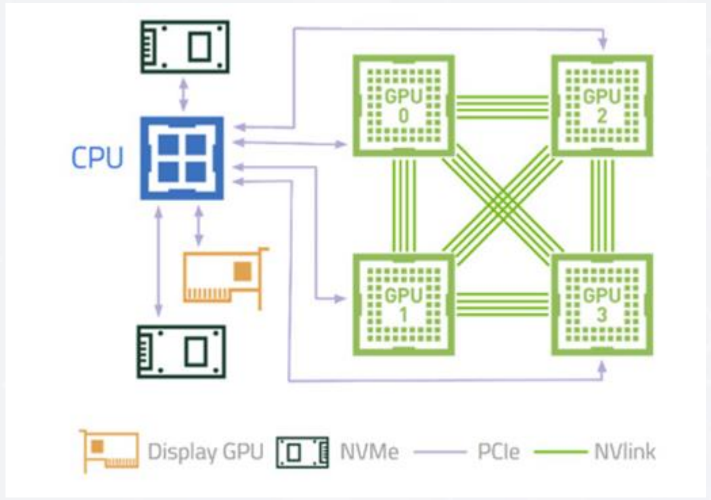
	vRAM	CUDA® Cores
A100 Tensor Core GPU	80 GB	7936

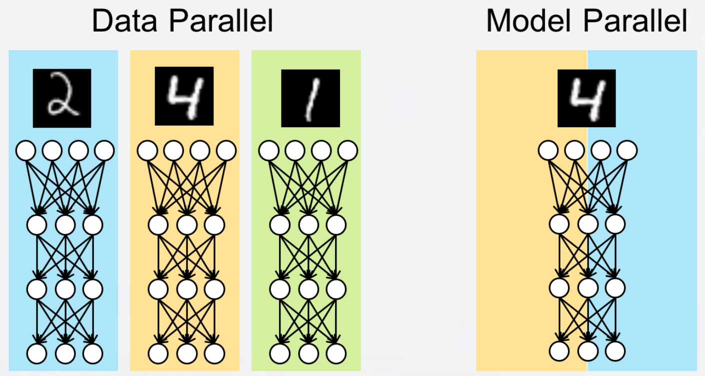
Model	**# parameters**	Memory requirement for parameters (FP32)	**type**
GPT-3	175 b	~ 800 GB	NLP, transformer

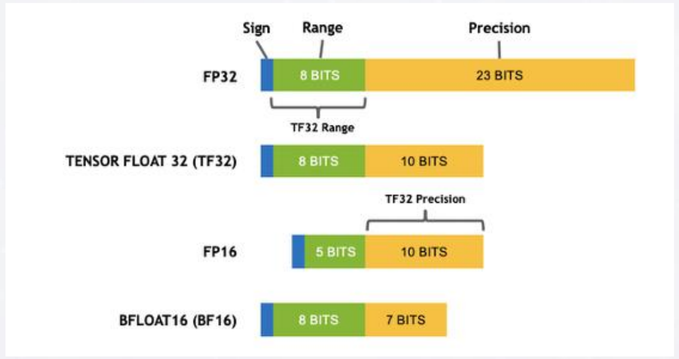
How to train faster

- Hardware:
  - fast connectivity between GPUs
    - intra-node: NVLink
    - inter-node: Infiniband
- Software:
  - Distributed and Parallel Training
    - Data parallel
    - Model parallel
  - Mixed precision
    - fp16 (amp)

- GPU-to-GPU: NvLink: A100-to-A100 peer bandwidth is 300 GB/s bi-directional
- CPU to GPU:connected by PCIe Gen4 buses each provide 31.5 Gb/s for a total of 252 Gb/s







Hardware: GPUs available in HiPerGator

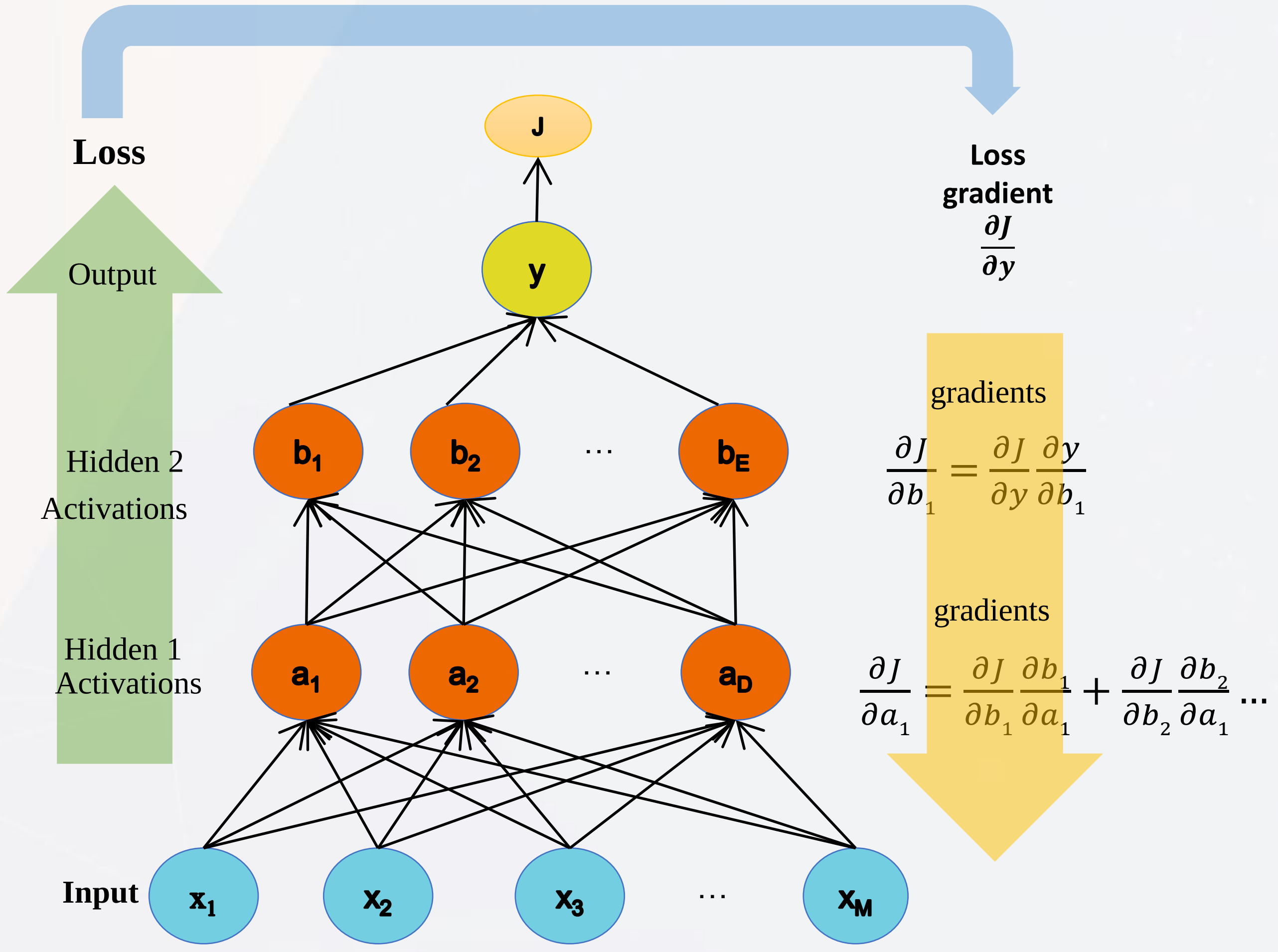
HiPerGator AI

- NVIDIA A100
  - 1,120 GPUs
  - 140 nodes
  - 8 GPUs/node
  - 80GB GPU memory per GPU

GPU	Host Quantity	Host Architecture	Host Memory	Host Interconnect	CPUs per Host	CPUS per Socket	GPUs per Host	CPUs per GPU	Memory per GPU	SLURM Feature	GRES GPU type
GeForce 1080Ti	1	Intel Haswell	128 GB	FDR IB	28	14	2	14	11GB	n/a	geforce
GeForce 2080Ti	32	Intel Skylake	187 GB	EDR IB	32	16	8	4	11GB	2080ti	geforce
GeForce 2080Ti	38	Intel Cascade Lake	187 GB	EDR IB	32	16	8	4	11GB	2080ti	geforce
Quadro RTX 6000	6	Intel Cascade Lake	187 GB	EDR IB	32	16	8	4	23GB	rtx6000	quadro
NVIDIA A100	140	AMD EPYC ROME	2 TB	HDR IB	128	16	8	16	80GB	a100	a100

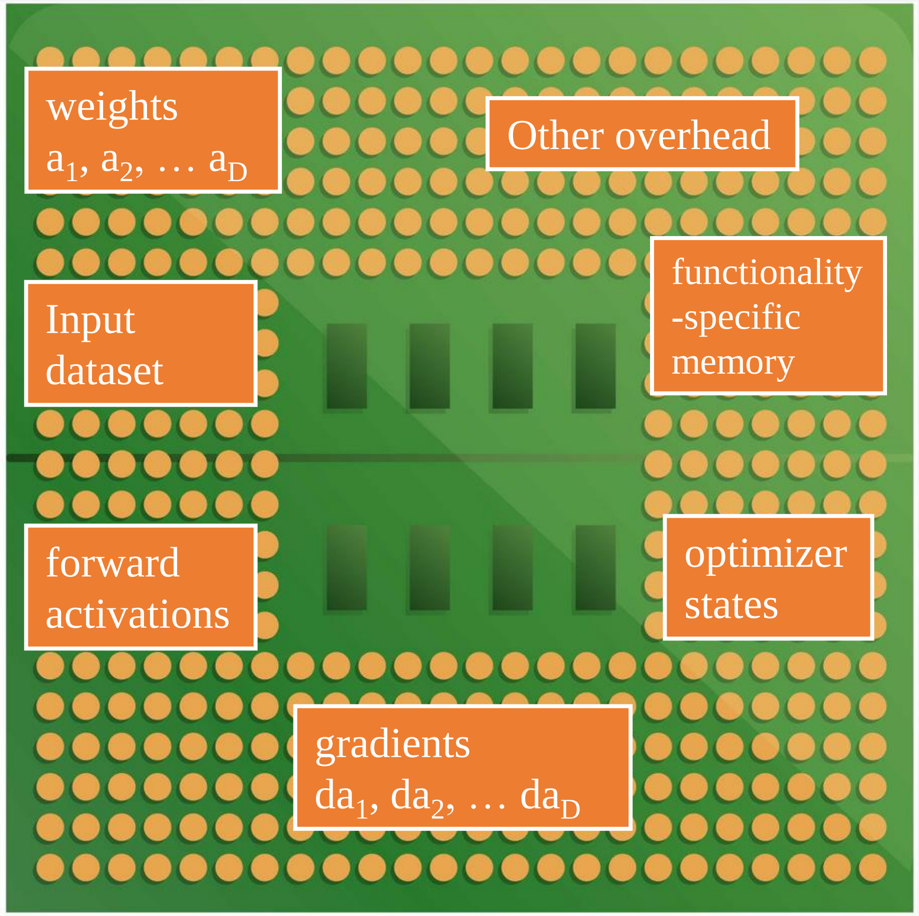
Recognize host and GPU memory are different

Anatomy of GPU memory allocation



Loss
Output
Hidden 2 Activations
Hidden 1 Activations
Input
J
y
b_1 b_2 … b_E
a_1 a_2 … a_D
x_1 x_2 x_3 … x_M
Loss gradient $\frac{\partial J}{\partial y}$
gradients
$\frac{\partial J}{\partial b_1} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b_1}$
gradients
$\frac{\partial J}{\partial a_1} = \frac{\partial J}{\partial b_1}\frac{\partial b_1}{\partial a_1} + \frac{\partial J}{\partial b_2}\frac{\partial b_2}{\partial a_1} \dots$

Neural Network Model



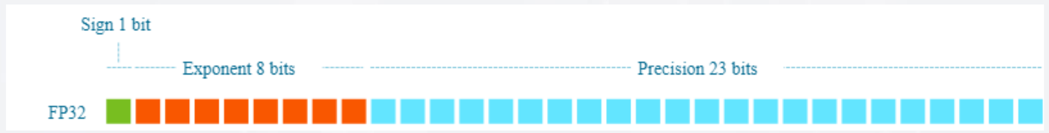
weights $a_1, a_2, \dots a_D$
Other overhead
Input dataset
functionality -specific memory
forward activations
optimizer states
gradients $da_1, da_2, \dots da_D$

GPU memory

Memory requirement estimation

Estimation of total VRAM memory consumption

- Consider only model parameters , gradients and optimizer states
- Ignore input, forward activations, and memory overhead
- Use FP32 data representation (4 bytes per floating number)
- Adam optimizer (storing 16 bytes per parameter)



FP32 data representation

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t.$$

Consider a model with <u>one billion</u> parameters

$$\mathbf{10^9 * (4B + 4B + 16B) = 10^9 * 24\ Byte = 24\ GB}$$

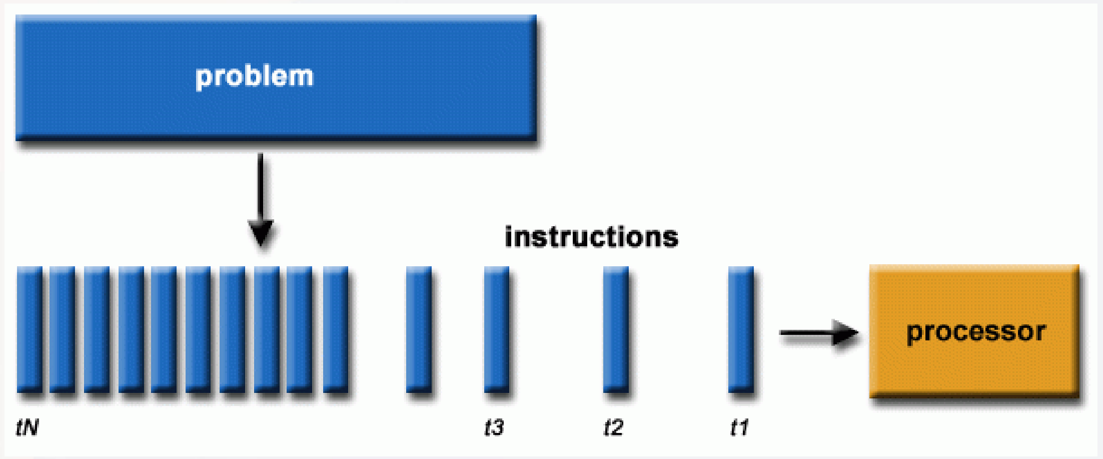
4 bytes per parameter

4 bytes per gradient

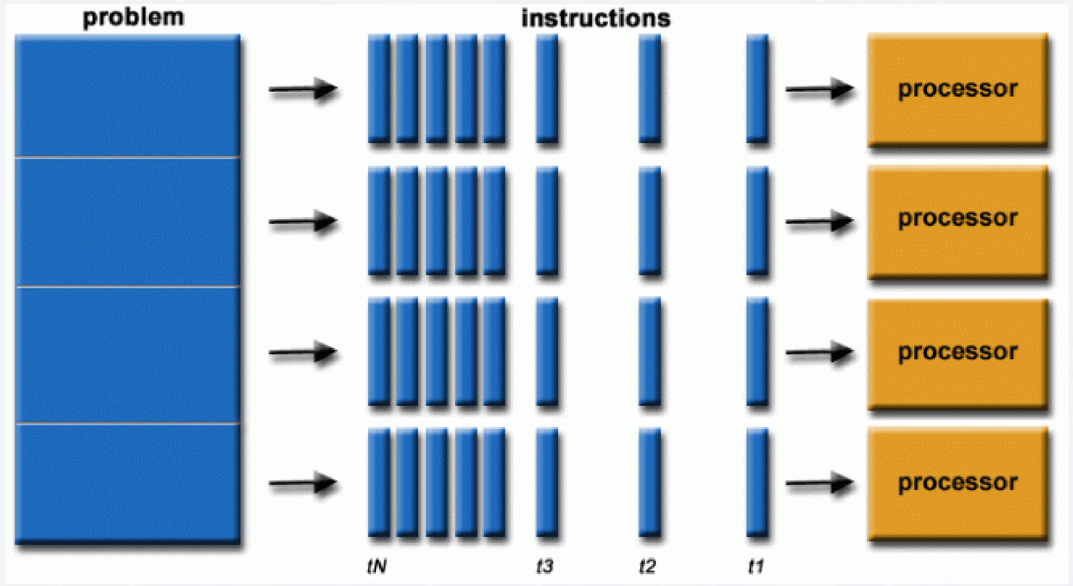
16 bytes for Adam optimizer states

Fundamentals of parallel computing

Serial Computing



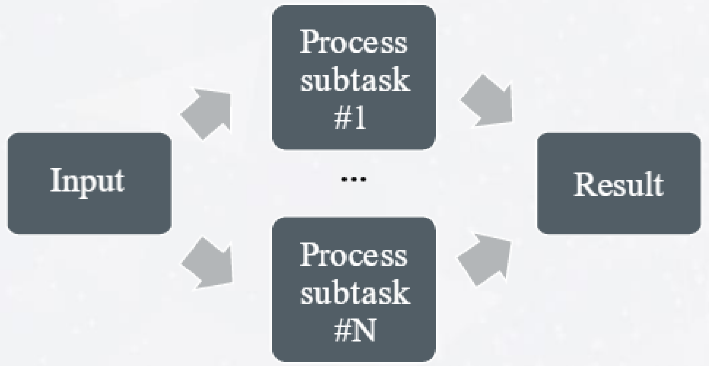
Parallel Computing

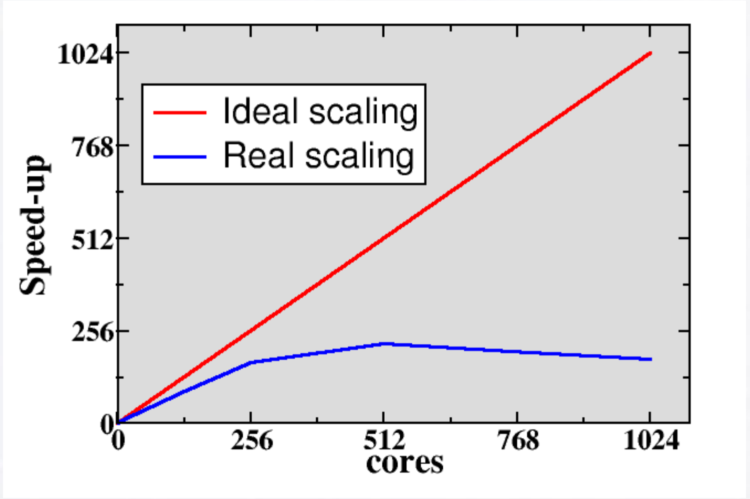


- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed

Basic concepts and metrics of parallel computing

- **Computational Speed of the process**
  - **Scalability**
    - Efficiency of a system for a growing workload
  - **Speedup**
    - the ratio of solution time for the sequential vesus its parallel counterpart
  - **Throughput**
    - number of computing tasks per time unit
      - the number of images per unit time that can be processed



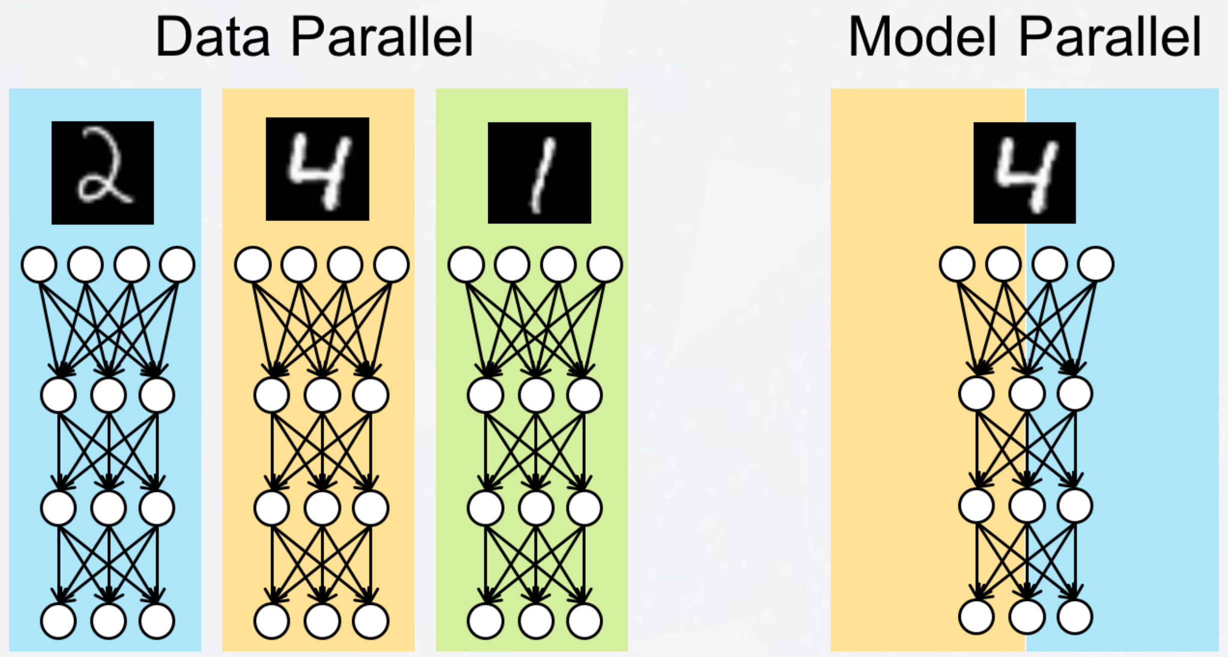


Types of training parallelism

- **Types of training parallelism**
  - Data Parallel
    - distributed training data for parallel optimization
    - perform allReduce on gradient
  - Model Parallelism

    deep learning model is partitioned across multiple devices (GPU)

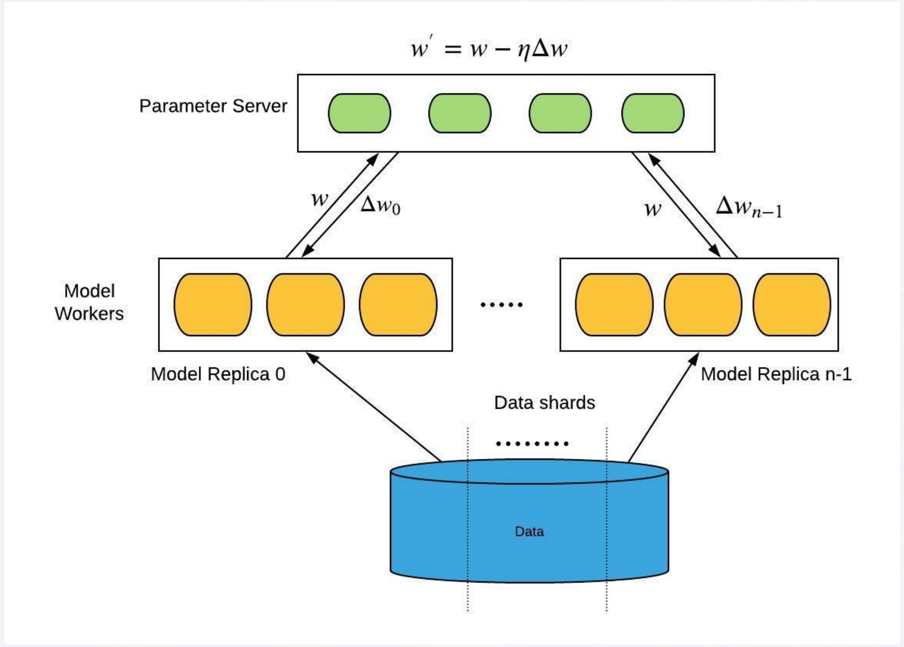
    - Pipeline Parallel
    - Tensor Parallel
    - Optimizer-Level Parallel
      - Zero Redundancy Optimizer (ZeRO)
    - Other techniques
      - Activation checkpointing



Data parallelism

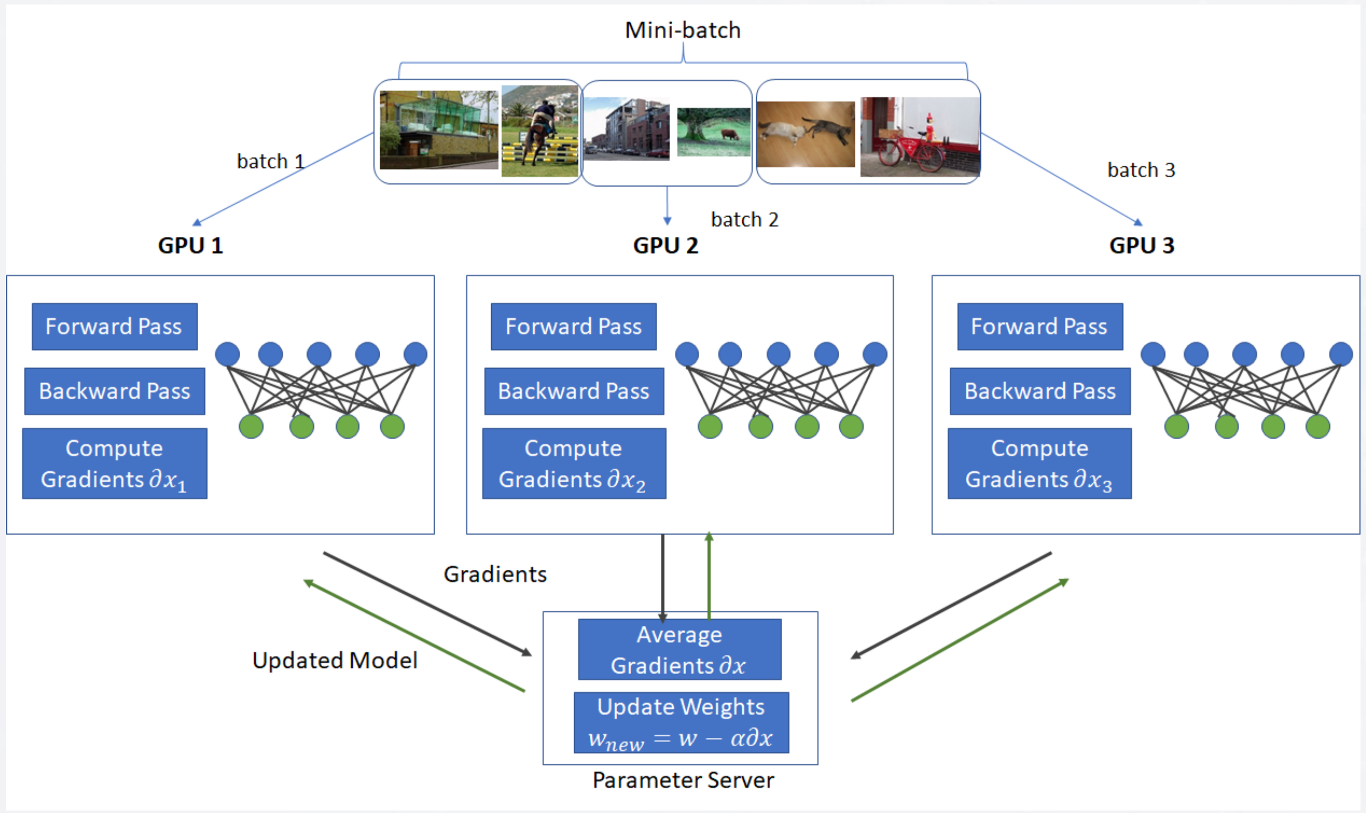
- Data parallelism can be defined as the splitting of the data into N partitions where each of the partitions can be used for training into different machines or devices like CPU cores, GPUs, or even machines.

- Accumulate gradients, there are two strategies to update:
  - Synchronous Distributed
  - Asynchronous Distributed



Synchronous Distributed and gradient accumulation

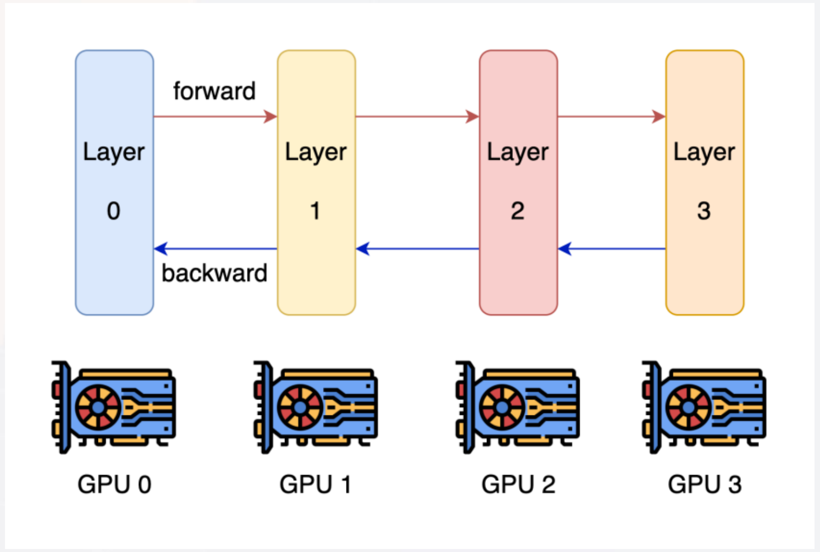
Divide mini-batch in micro-batches, assign on each GPU , accumulate gradient at the end of each iteration


Mini-batch
batch 1
batch 2
batch 3
GPU 1
GPU 2
GPU 3
Forward Pass
Backward Pass
Compute Gradients ∂x_1
Compute Gradients ∂x_2
Compute Gradients ∂x_3
Gradients
Updated Model
Average Gradients ∂x
Update Weights $w_{new} = w - \alpha \partial x$
Parameter Server

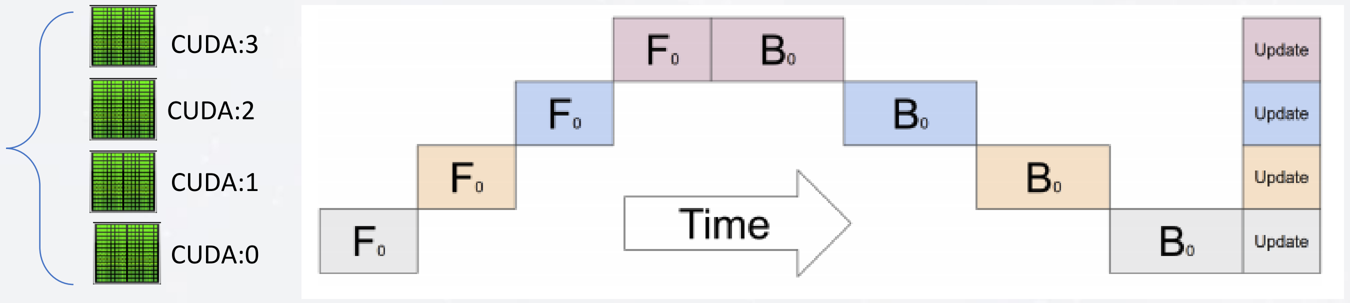
Picture from https://www.telesens.co/2017/12/25/understanding-data-parallelism-in-machine-learning/

Pipeline parallelism

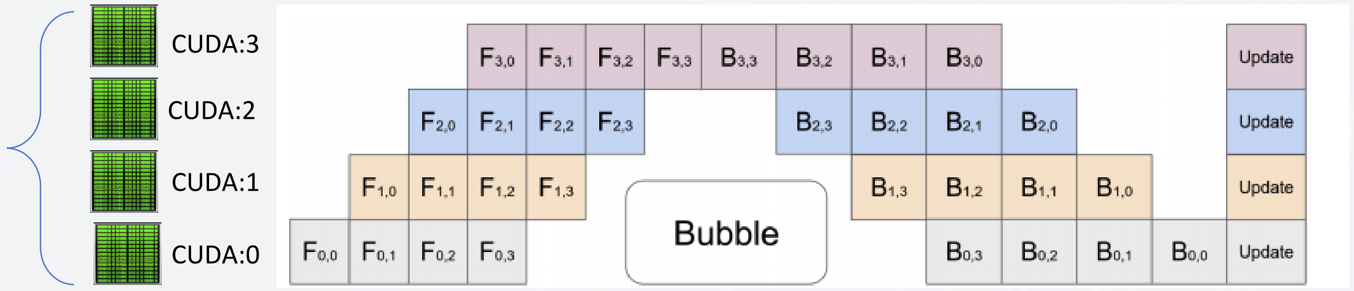
- Partition DNN according to depth, assign layers to specific processor



Pipeline parallel illustration



Naïve model parallelism execution: the training process suffer from GPU under utilization since only one GPU is activate.



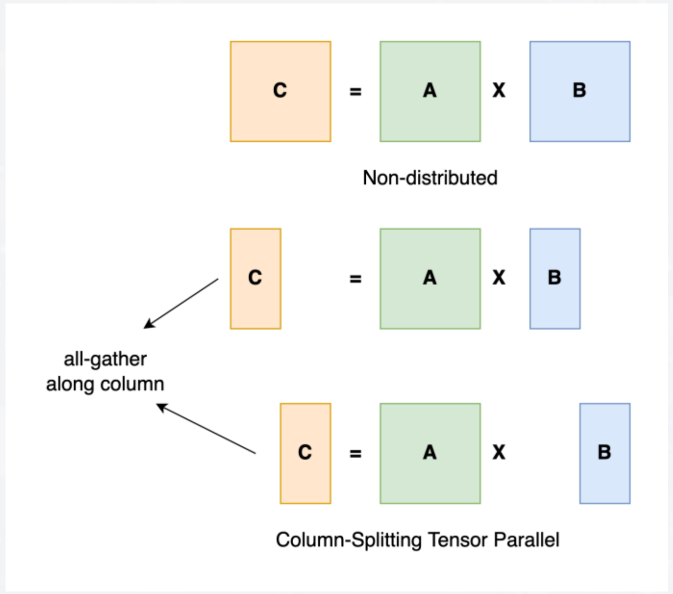
CUDA:3
CUDA:2
CUDA:1
CUDA:0
$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$ $B_{3,3}$ $B_{3,2}$ $B_{3,1}$ $B_{3,0}$
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$ $B_{2,3}$ $B_{2,2}$ $B_{2,1}$ $B_{2,0}$
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$ $B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$ $B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$
Bubble
Update

Pipelined model parallelism Execution: splits the input minibatch into multiple microbatches and pipelines the execution of these microbatches across multiple GP

Tensor parallelism

- Each tensor in one individual layer is split into multiple devices.
- each shard of the tensor resides on its designated gpu.

- Tensor parallelism is required in cases in which a single layer consumes most of the GPU memory
- Tensor parallelism is useful for extremely large models in which a pure pipelining is simply not enough. For example, in Megatron-LM library.



Tensor parallel illustration

PyTorch - Data Parallel Utilities

- PyTorch mostly provides two functions namely nn.DataParallel and nn.DistributedDataParallel to use multiple GPUs in a single node and multiple nodes during the training.

- DataParallel (DP)
  - Simple to quick start
- DistribtuedDataParallel (DDP)
  - robust
  - recommended

PyTorch - Data Parallel (DP)

- Implements data parallelism at the module level.

torch.nn.DataParallel(*module*, *device_ids=None*, *output_device=None*, *dim=0*)

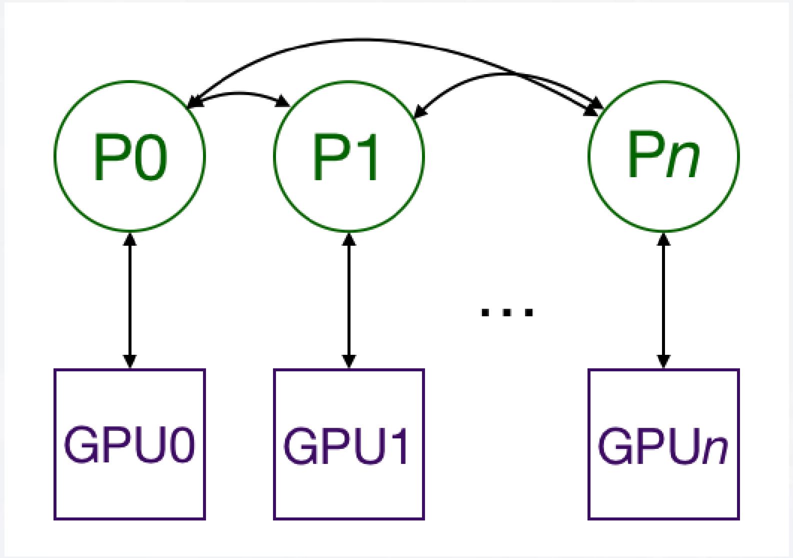
```
>>> net = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
>>> output = net(input_var)  # input_var can be on any device, including CPU
```

- Simple to use, just wrap your model by nn.DataParallel, Pytorch will do everything else for you:
    - 1) replicates model to each participating GPU
    - 2) split minibatch to GPUs
    - 3) sync the gradients.
- Single process, multi-thread, subject to GIL mutex
- Only applicable to GPUs on the same node (machine).
- Does not work with model parallel
- DistributedDataParallel (DDP) is preferred instead of DataParallel (DP) even if on a single node.

Try to use PyTorch DDP instead of DP !

PyTorch - Distributed Data Parallel (DDP)

- Multiprocessing with DistributedDataParallel duplicates the model across multiple GPUs, each of which is controlled by one process.
- The GPUs can all be on the same node or spread across multiple nodes.
- Every process does identical task, and each process communicates with all the others.
- Only gradients are passed between the processes/GPUs so that network communication is less of a bottleneck.



PyTorch - Distributed Data Parallel (DDP)

- DistributedDataParallel (DDP) API implements data parallelism at the module level which can run across multiple machines.
- Collective communications in the torch.distributed package;

- Parameters:
  - **Number of GPUs**:
    - Spawn up N processes for N GPUs.
    - Each process exclusively works on a single GPU from 0 to N-1.
  - **Batch size**:
    - The batch size should be larger than the number of GPUs;
  - **Learning step**
    - Multiple learning step by number of GPUs
  - **Collective Communication Backend**
    - nccl backend is currently the fastest and highly recommended backend when using GPUs.

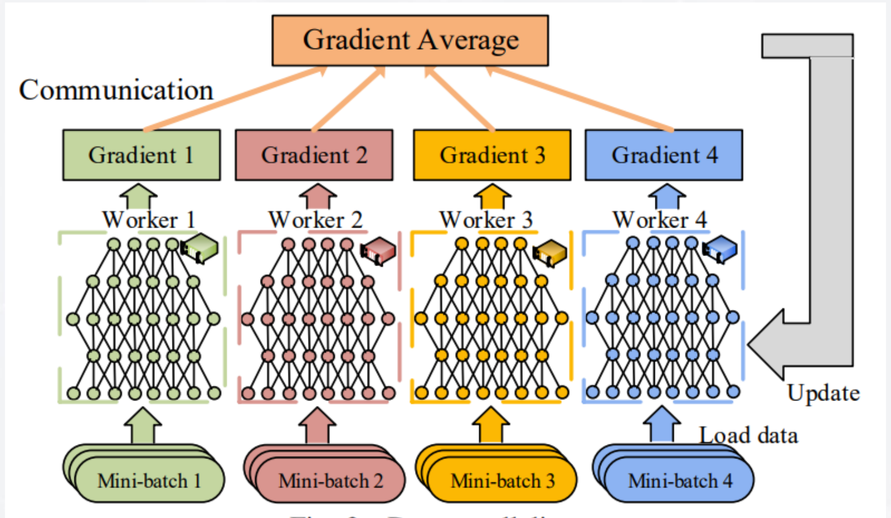
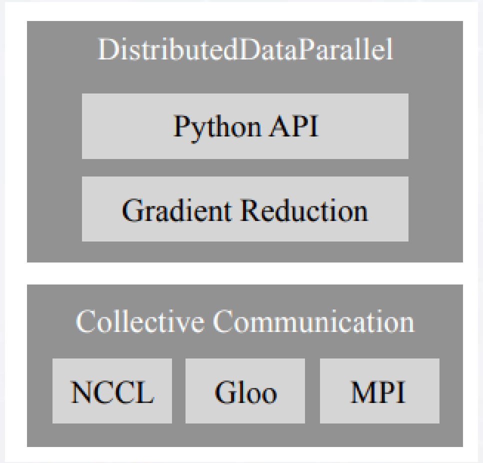


Fig. 2. Data parallelism.

Tang et al. 2020



Li et al 2019

PyTorch DDP code walkthrough

- 1. Constructing the process group
- 2. Wrap the model with DDP
- 3. Distribute dataloader (DistributedSampler wrapper)
- 4. Launch with process
  - using torchrun utility
  - torch.multiprocessing.spawn function
- 5. Clean up the process group

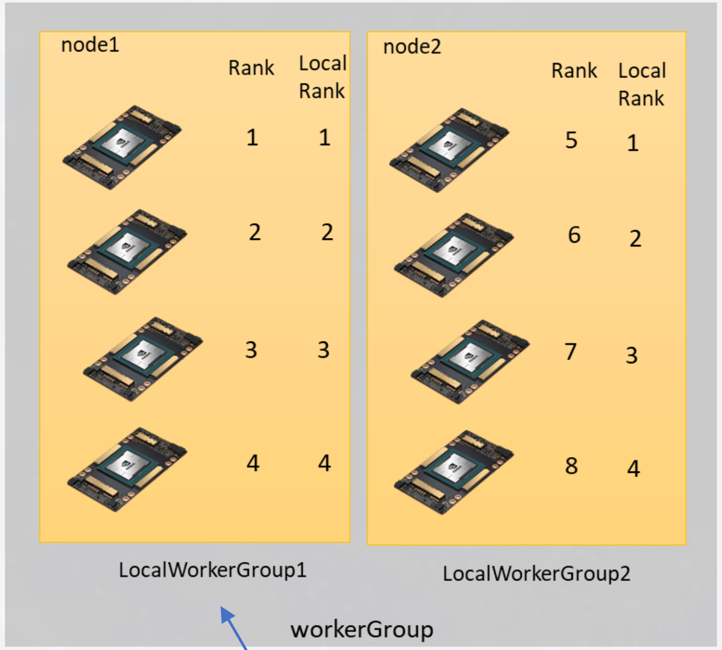
Pytorch DDP code walkthrough

- 1. import required libraries

```
+ 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
```

- 2. Constructing the process group

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



For single node only, for multi-node, use slurm to set "MASTER_ADDR"
localhost can be used for single node

Pytorch DDP code walkthrough

- 3. wrap the DDP model

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

- 4. Distributing input dataloader

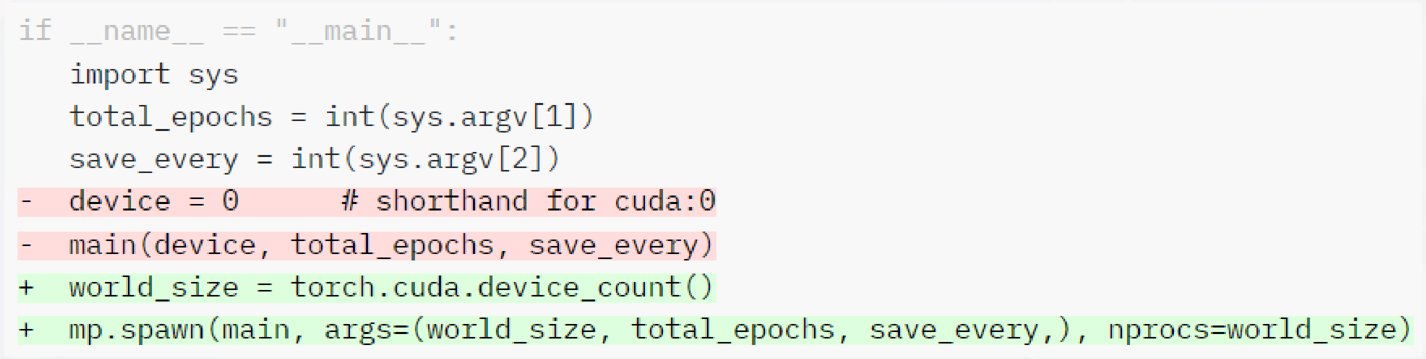
```
train_data = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=32,
-   shuffle=True,
+   shuffle=False,
+   sampler=DistributedSampler(train_dataset),
)
```

- 5. Calling the *set_epoch()* method on the *DistributedSampler* at the beginning of each epoch

```
def _run_epoch(self, epoch):
    b_sz = len(next(iter(self.train_data))[0])
+   self.train_data.sampler.set_epoch(epoch)
    for source, targets in self.train_data:
      ...
      self._run_batch(source, targets)
```

Pytorch DDP code walkthrough

- 7. launch the training process without using torchrun utility
  - 7.1 use mp.spawn() to spin up one process for each GPU



```
if __name__ == "__main__":
    import sys
    total_epochs = int(sys.argv[1])
    save_every = int(sys.argv[2])
-   device = 0        # shorthand for cuda:0
-   main(device, total_epochs, save_every)
+   world_size = torch.cuda.device_count()
+   mp.spawn(main, args=(world_size, total_epochs, save_every,), nprocs=world_size)
```

  - How to launch in terminal?
    - Exactly same as single GPU training:

      python train_script.py --parameters

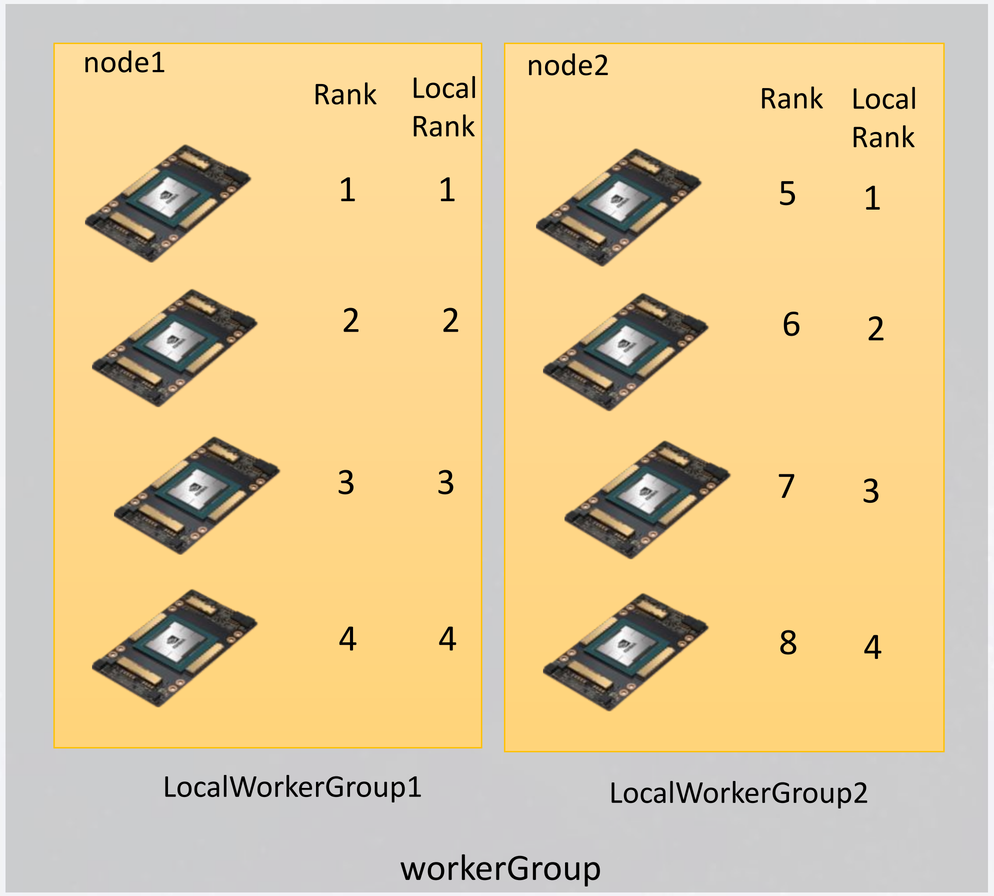
Torchrun utility

- torchrun is a python console script, equals to

    python –m torch.distributed.run

- Definitions for torchrun
    - WORLD_SIZE
    - RANK
    - LOCAL_RANK
        - gpu_id
    - LOCAL_WORLD_SIZE
    - MASTER_ADDR
    - MASTER_PORT

RANK and WORLD_SIZE are assigned automatically by torchrun



Launch MultiGPU process - torchrun

- 7. launch the training process using torchrun
  - Single-node multi-GPU
  - Multi-node multi-GPU
    - Use torchrun to launch on each node

Changes to your training script:

```
import os
local_rank =
int(os.environ["LOCAL_RANK"])
```

```
import torch.distributed as dist
dist.init_process_group(backend="gloo|nccl")
```

launch in terminal for single Node

```
torchrun
    --standalone
    --nnodes=1
    --nproc_per_node=$NUM_TRAINERS
    YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

launch in SLURM script for single or multi node:

```
srun --export=ALL torchrun \
--nnodes 4 \
--nproc_per_node 1 \
--rdzv_id $RANDOM \
--rdzv_backend c10d \
--rdzv_endpoint $head_node_ip:29500 \
multigpu_torchrun.py 50 10
```

Demo time

- OpenOnDemand
    - 1. Launch a OpenOnDemand terminal with 1 Node and 2 GPUs
    - 2. training with single GPU
    - 3. training with 2 GPUs in 1 node
- SLURM training job
    - 4. submit a SLURM training job on multi-Node and multi-GPUs

Questions?