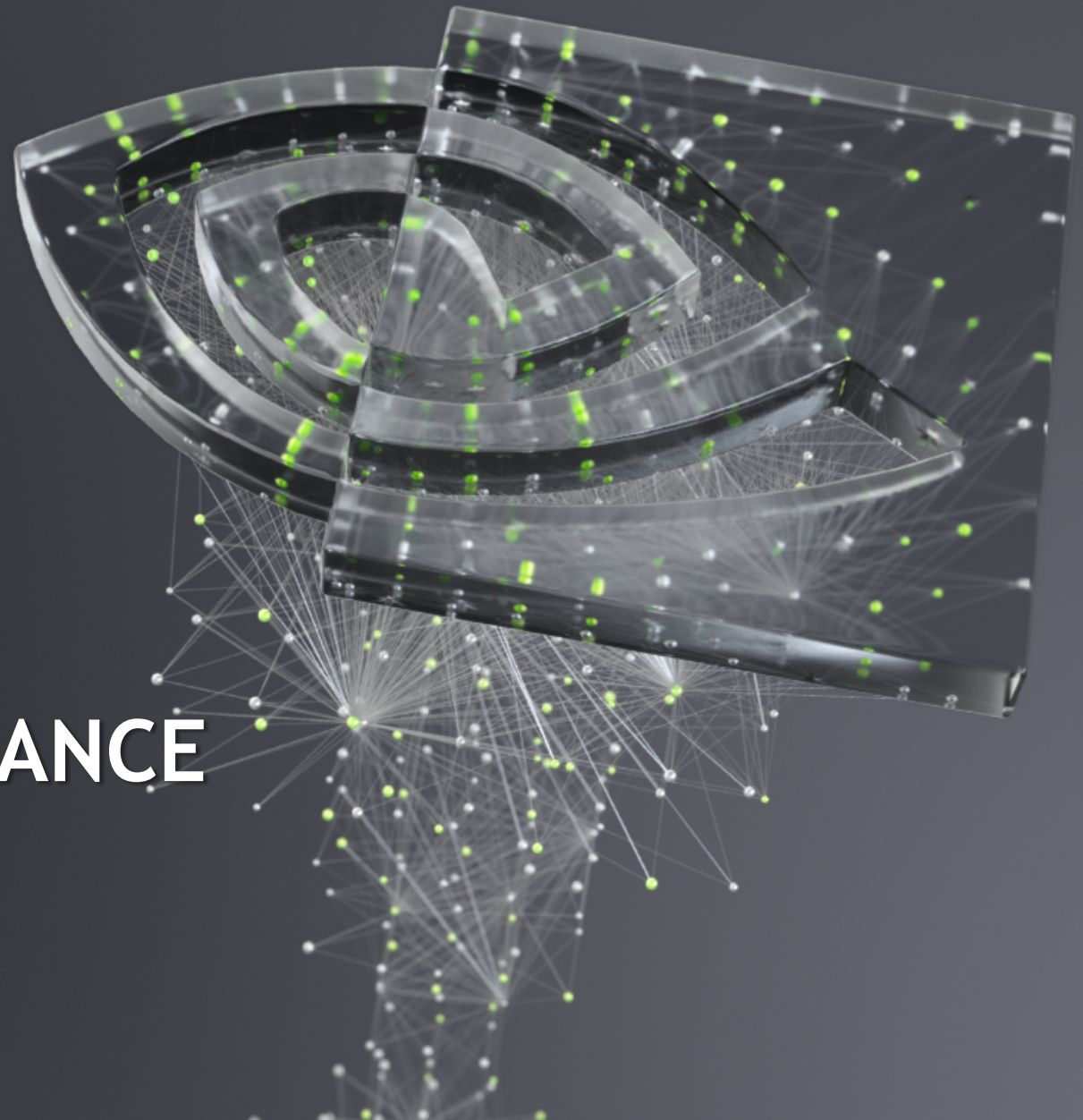


PYTORCH PERFORMANCE TUNING GUIDE

Szymon Migacz, 08/23/2020

CONTENT

PyTorch Performance Tuning Guide

Simple techniques to improve training performance

Implement by changing a few lines of code

Work well in conjunction with AMP

ENABLE ASYNC DATA LOADING & AUGMENTATION

- PyTorch DataLoader supports asynchronous data loading / augmentation
  - Default settings: `num_workers=0`, `pin_memory=False`
- Use `num_workers > 0` to enable asynchronous data processing
- It's almost always better to use `pin_memory=True`

Example: PyTorch MNIST example: DataLoader with `{'num_workers': 1, 'pin_memory': True}`.

Setting for the training DataLoader	Time for one training epoch
`{'num_workers': 0, 'pin_memory': False}`	8.2 s
`{'num_workers': 1, 'pin_memory': False}`	6.75 s
`{'num_workers': 1, 'pin_memory': True}`	6.7 s
`{'num_workers': 2, 'pin_memory': True}`	4.2 s
`{'num_workers': 4, 'pin_memory': False}`	4.5 s
`{'num_workers': 4, 'pin_memory': True}`	4.1 s
`{'num_workers': 8, 'pin_memory': True}`	4.5 s

PyTorch 1.6, NVIDIA Quadro RTX 8000

ENABLE cuDNN AUTOTUNER

For convolutional neural networks, enable cuDNN autotuner by setting:

```
torch.backends.cudnn.benchmark = True
```

- cuDNN supports many algorithms to compute convolution
- autotuner runs a short benchmark and selects the algorithm with the best performance

Example:
nn.Conv2d with 64 3x3 filters applied to an input with batch size = 32, channels = width = height = 64.

Setting	`cudnn.benchmark = False` (the default)	`cudnn.benchmark = True`	Speedup
Forward propagation (FP32) [us]	1430	840	1.70
Forward + backward propagation (FP32) [us]	2870	2260	1.27

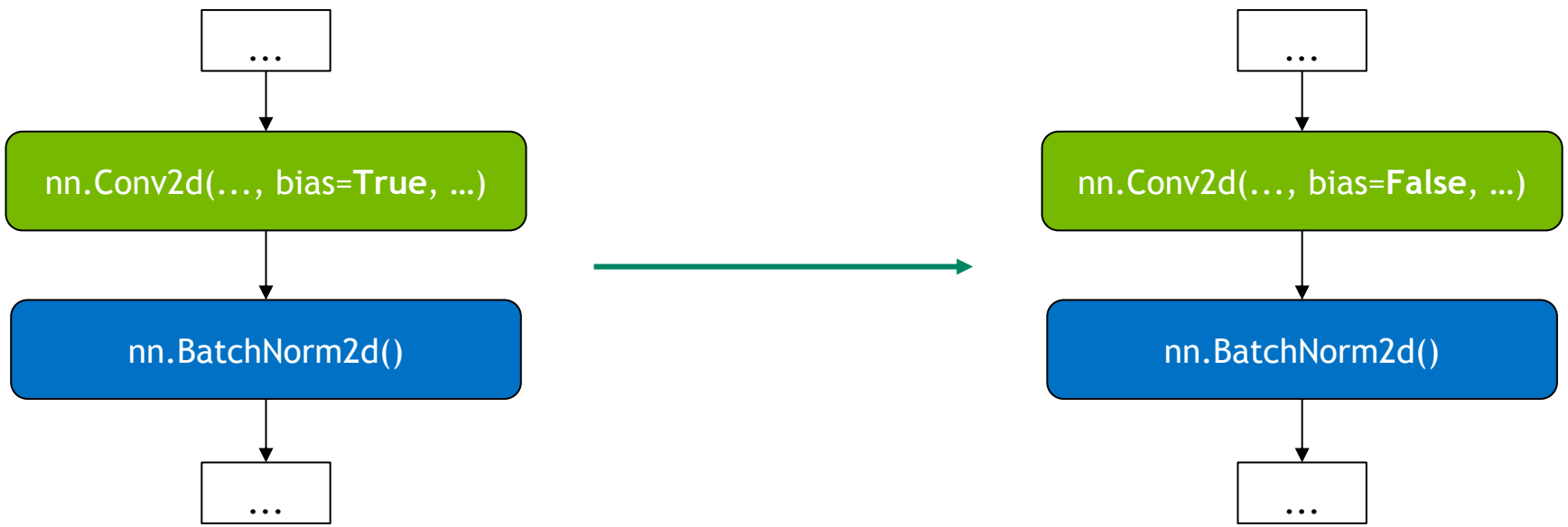
PyTorch 1.6, NVIDIA Quadro RTX 8000

INCREASE BATCH SIZE

- Increase the batch size to max out GPU memory
  - often AMP reduces mem requirements → increase batch size even more

- When increasing batch size:
  - tune learning rate, add learning rate warmup and learning rate decay, tune weight decay
  - or switch to optimizer designed for large-batch training
    - LARS,
    - LAMB,
    - NVLAMB,
    - NovoGrad

DISABLE BIAS FOR CONVOLUTIONS DIRECTLY FOLLOWED BY A BATCH NORM



Also applicable to Conv1d, Conv3d as long as BatchNorm normalizes on the same dimension as convolution's bias.

Use parameter.grad = None instead of model.zero_grad()

```
model.zero_grad()
```

```
for param in model.parameters():
    param.grad = None
```

- executes memset for every parameter in the model
- backward pass updates gradients with "+=" operator (read + write)

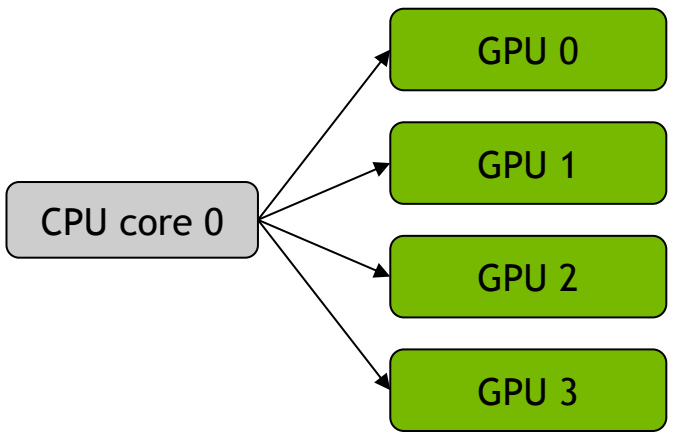
- doesn't execute memset for every parameter
- memory is zeroed-out by the allocator in a more efficient way
- backward pass updates gradients with "=" operator (write)

DISABLE DEBUG APIs FOR FINAL TRAINING

- anomaly detection:
  - torch.autograd.detect_anomaly
  - torch.autograd.set_detect_anomaly(True)
- autograd profiler:
  - torch.autograd.profiler.profile
- automatic NVTX ranges:
  - torch.autograd.profiler.emit_nvtx
- autograd gradcheck:
  - torch.autograd.gradcheck
  - torch.autograd.gradgradcheck

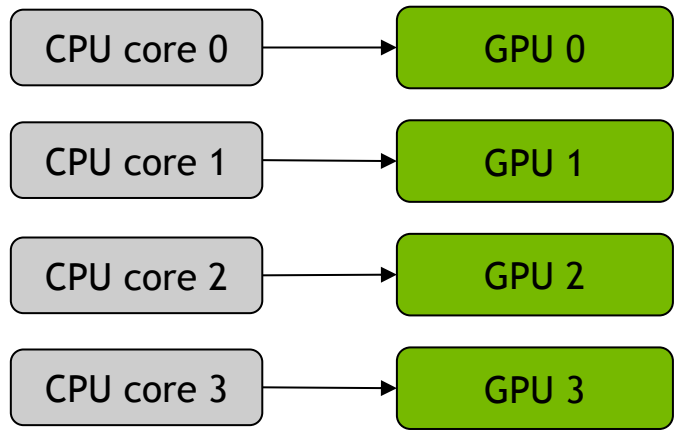
USE EFFICIENT MULTI-GPU BACKEND

DataParallel



- 1 CPU core drives multiple GPUs
- 1 python process drives multiple GPUs (GIL)
- only up to a single node

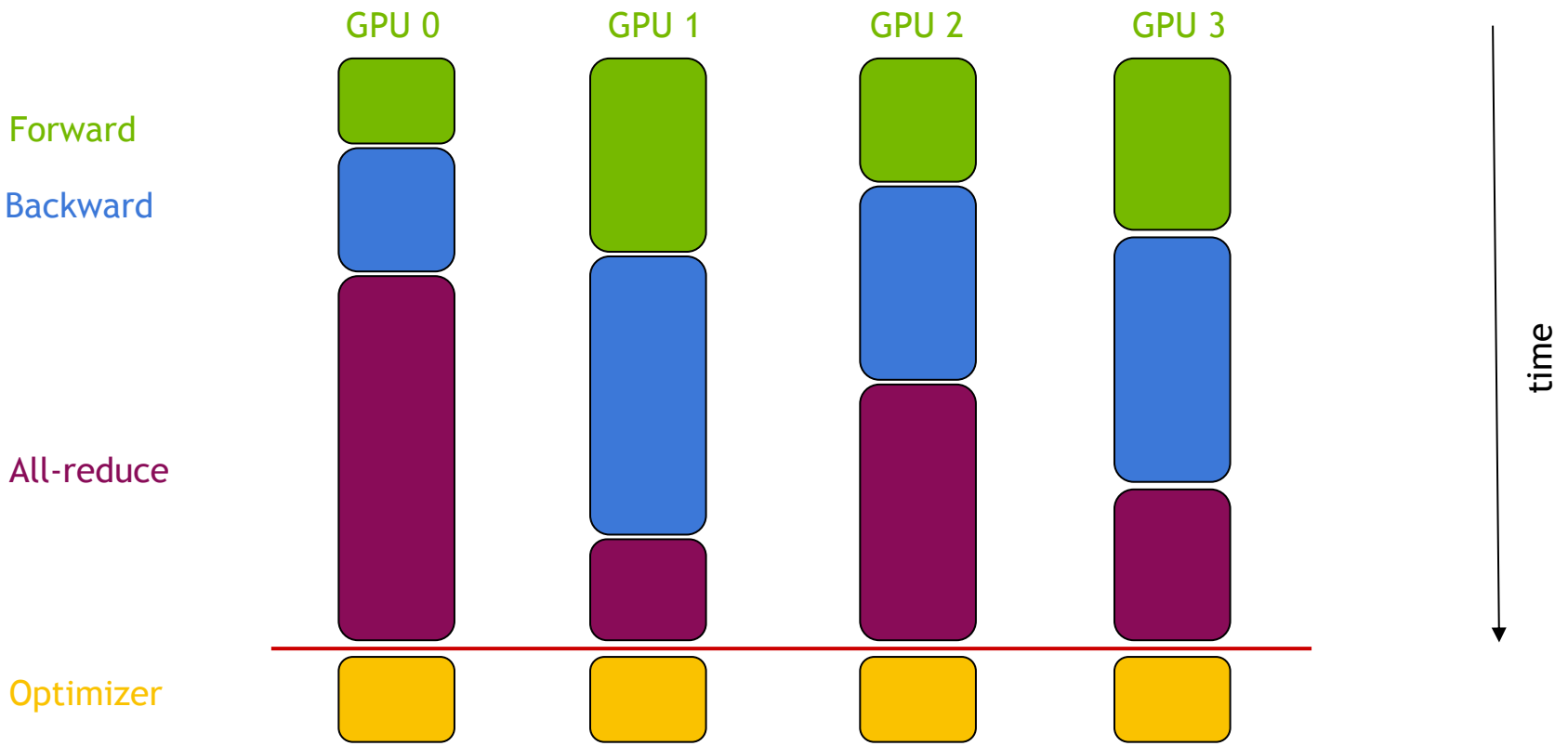
DistributedDataParallel



- 1 CPU core for each GPU
- 1 python process for each GPU
- single-node and multi-node (same API)
- efficient implementation:
  - automatic bucketing for grad all-reduce
  - all-reduce overlapped with backward pass
- multi-process programing

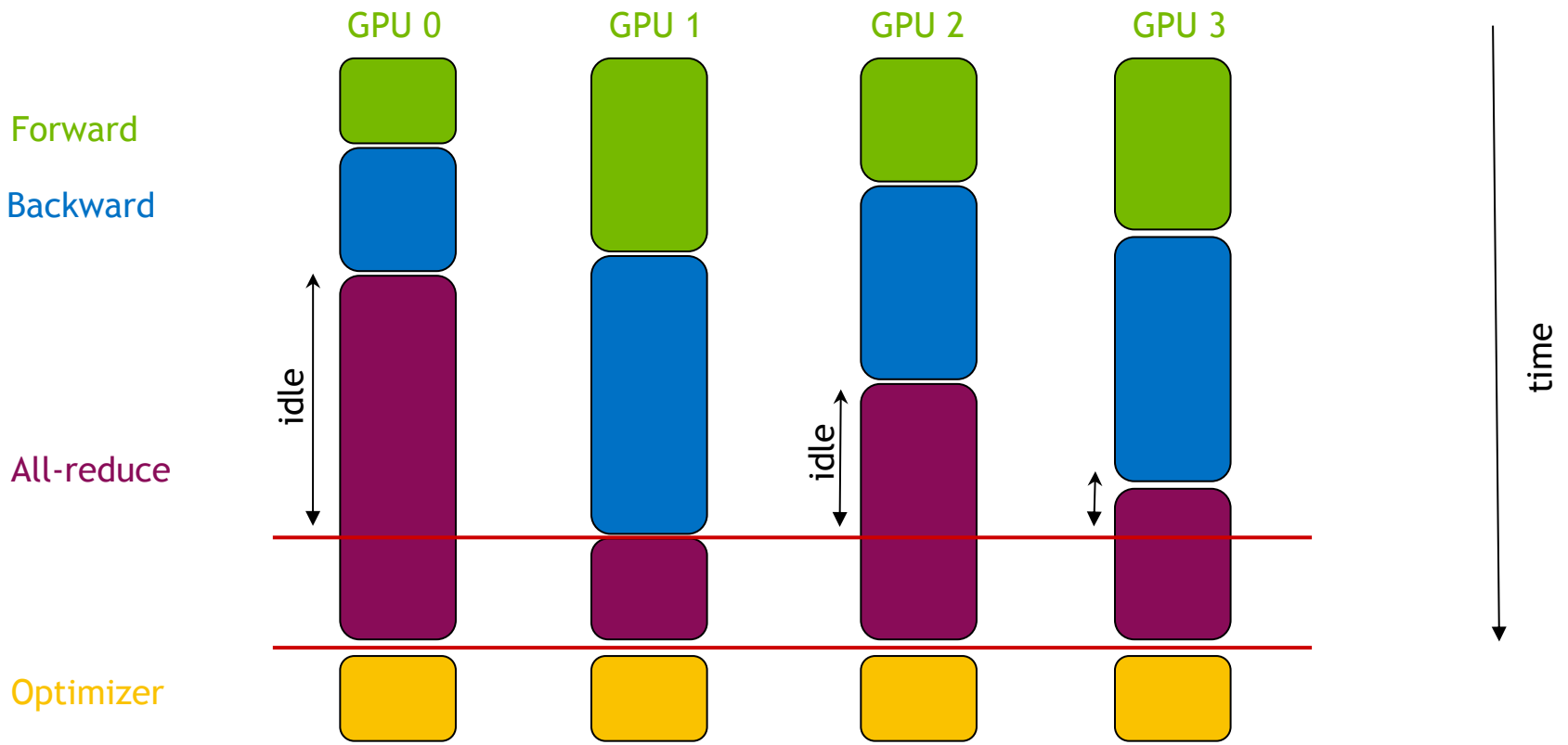
LOAD-BALANCE WORKLOAD ON MULTIPLE GPUs

Gradient all-reduce after backward pass is a synchronization point in a multi-GPU setting



LOAD-BALANCE WORKLOAD ON MULTIPLE GPUs

Gradient all-reduce after backward pass is a synchronization point in a multi-GPU setting



USE FUSED BUILDING BLOCKS FROM APEX

- NVIDIA APEX (repository, documentation) offers optimized, reusable building blocks.
- APEX can be pip-installed from github: https://github.com/NVIDIA/apex#quick-start
- APEX is pre-installed in PyTorch NGC docker containers

Components:

- distributed training:
  - apex.parallel.SyncBatchNorm
- fused optimizers:
  - apex.optimizers.FusedAdam
  - apex.optimizers.FusedLAMB
  - apex.optimizers.FusedNovoGrad
  - apex.optimizers.FusedSGD
- apex.normalization.FusedLayerNorm

CHECKPOINT TO RECOMPUTER INTERMEDIATES

Regular training:

- forward: store outputs for all operations (more memory)
- backward: no re-computation
- training with smaller batch size

Activation checkpointing:

- Forward; stores outputs only for some operations (less memory)
- backward: remaining intermediate values are re-computed (additional compute)
- enables larger batch size → better GPU and TensorCores utilization

- native PyTorch API: torch.utils.checkpoint
- select operations for checkpointing:
  - small re-computation cost and large memory footprint
    - activation (ReLU, Sigmoid, …),
    - up/down sampling,
    - matrix-vector ops with small accumulation depth

FUSE POINTWISE OPERATIONS

- PyTorch JIT can fuse pointwise operations into a single CUDA kernel.
- Unfused pointwise operations are memory-bound, for each unfused op PyTorch has to:
  - launch a separate CUDA kernel,
  - load data from global memory,
  - perform computation,
  - store results back into global memory,

Example:

```
def gelu(x):
    return x * 0.5 * (1.0 + torch.erf(x / 1.41421))
```

```
@torch.jit.script
def fused_gelu(x):
    return x * 0.5 * (1.0 + torch.erf(x / 1.41421))
```

Function name	Number of CUDA kernels launched	Execution time [us] (input vector with 1M elements)
`gelu(x)`	5	65
`fused_gelu(x)`	1	16

PyTorch 1.6, NVIDIA Quadro RTX 8000

CONCLUSION

- use async data loading / augmentation
- enable cuDNN autotuner
- increase the batch size, disable debug APIs and remove unnecessary computation
- use building blocks from NVIDIA APEX
- apply PyTorch JIT to fuse pointwise operations
- efficiently zero-out gradients
- use DistributedDataParallel instead of DataParallel
- load balance work on multiple GPUs