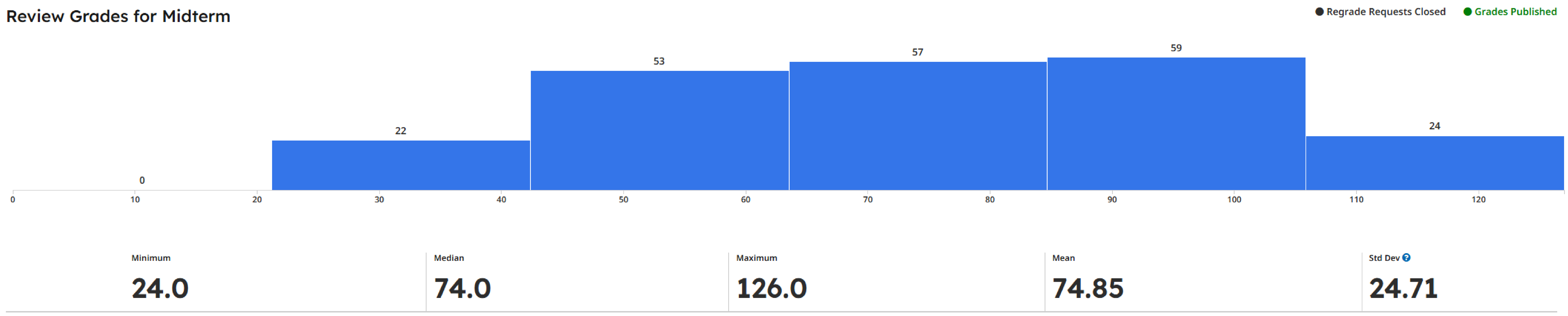
Lecture 19: Accelerating and scaling DNN training (GPU)

Data C182 (Fall 2024). Week 12. Tuesday Nov 12th, 2024

Speaker: Eric Kim

Announcements

- HW03 ("Transformers + NLP") out! Due: Fri Nov 22nd 11:59 PM PST
  - Please start early!
- Midterm finalized stats (post regrade requests)
  - (very little change from pre regrade requests)



Review Grades for Midterm

Minimum	Median	Maximum	Mean	Std Dev
24.0	74.0	126.0	74.85	24.71

Today's lecture

- (Part 1) Guest talk by William Chen (TA)!
  - Topic: Robotics + DL
- (Part 2) GPU, multi-GPU, multi-Node training
- GPU tour: CUDA, cuDNN, NCCL (all_reduce)

What is a GPU?

- GPU: "Graphics Processing Unit"
- A separate hardware device that connects to your computer. Aka "discrete graphics card"
- Reason to use them: if you can express your computation in a "GPU-friendly" way, then you can have much higher compute **throughput** than CPUs
  - Typical applications: gaming, photo/video editing, video streaming
- **CPU benefits**: flexible, low-latency, accessible (every computer has a CPU!)
- **GPU benefits**: (very) high compute throughput

Pictured: my GeForce RTX 3080 Ti!

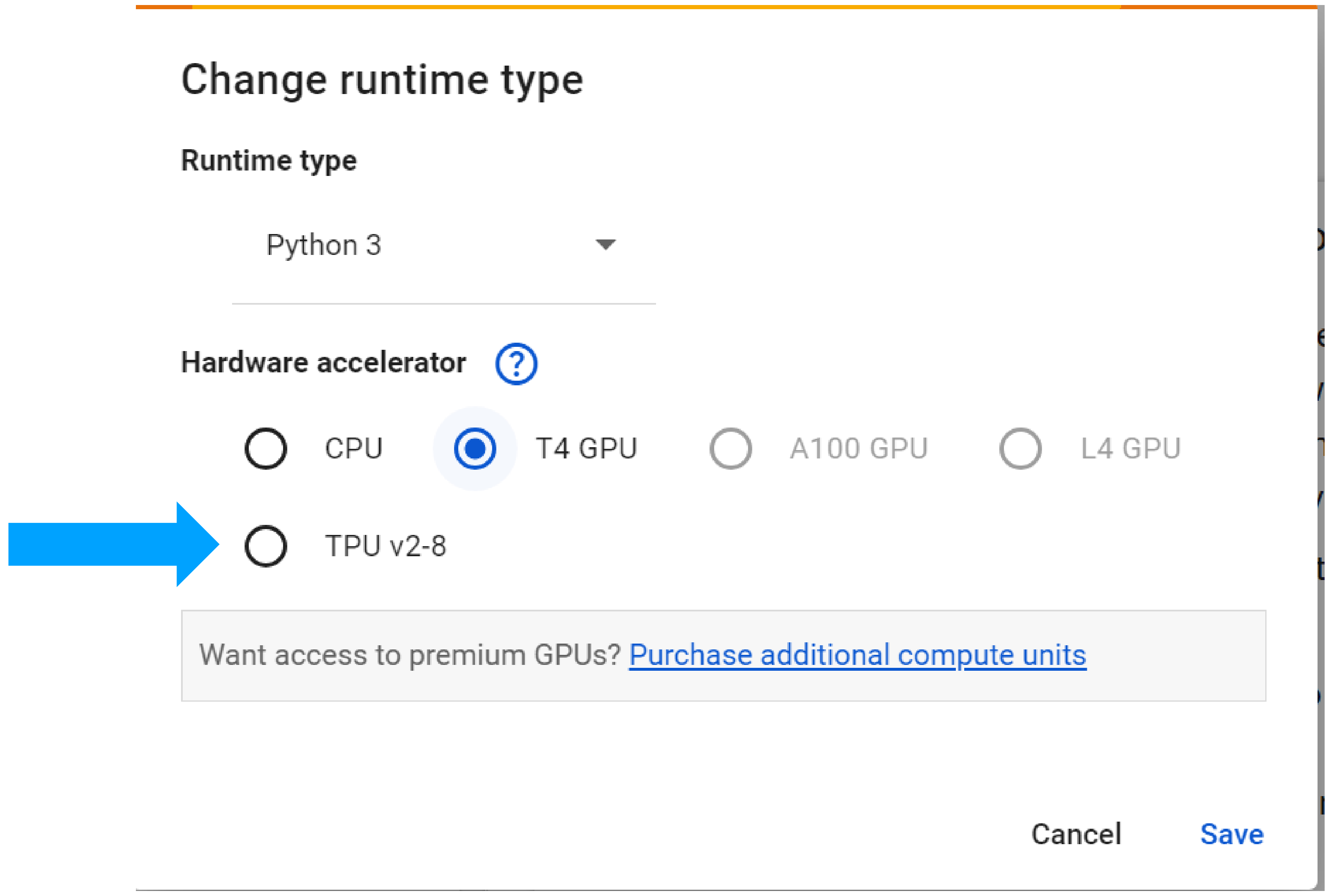
What is a GPU?

- Two classes of GPUs:
  - **Gaming/commodity**. Aka what you buy for your gaming PC.
    - Ex: Nvidia GeForce RTX 4080: >$999
  - **Data center**. Aka what big companies like Google/Meta/OpenAI train/serve their DNN models on.
    - Ex: Nvidia H100 GPU: ~$25k per card.
    - Most heavy-duty DNN train machines have 8 GPUS, so ~$200K per machine.
      - Aka Amazon Cloud + Nvidia is making the big bucks right now!

Pictured: Nvidia DGX-2 data center machine [link] at Nvidia's booth for CVPR 2018 (Salt Lake City!). Has 16 Tesla V100 GPUs, likely worth several hundreds of thousands of USD!

Major GPU types

- As of 2024: in this course (and ~99% of ML/AI): we use Nvidia GPUs (CUDA, cuDNN).
- DNN frameworks like pytorch have excellent support for Nvidia GPUs (CUDA)
- TPU ("Tensor Processing Unit"): A special type of "AI accelerator" hardware built by Google [link]
  - Ex: Tensorflow, Google papers often use TPUs instead of Nvidia GPUs
  - In Colab, you can choose either Nvidia GPU (eg T4) or a TPU.
- AMD is trying to enter the AI/ML market too (IMO, difficult to break in at the moment)



Change runtime type

Runtime type

Python 3

Hardware accelerator

○ CPU ● T4 GPU ○ A100 GPU ○ L4 GPU

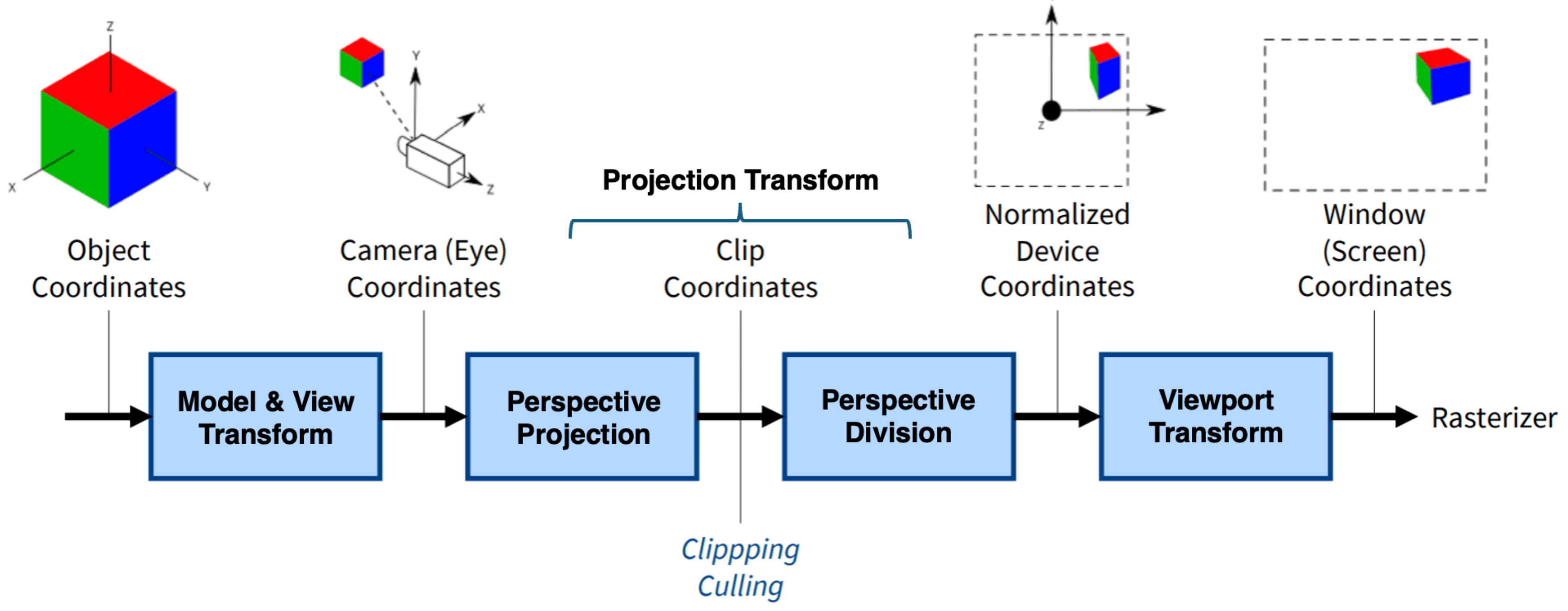
○ TPU v2-8

Want access to premium GPUs? Purchase additional compute units

Colab: T4 is Nvidia GPU, "TPU" is Google TPU

Original GPU motivation

- Original intent: accelerate graphics processing, eg for computer graphics (CGI, Pixar) and video games (real-time graphics)
    - Interested? Take CS 184! [link] Computer graphics and computer vision have a healthy relationship, lots of interesting overlap
- Motivation: graphics ultimately boils down to matrices and vectors. Need to accelerate matrix/vector computation, as the CPU wasn't enough back then (and still isn't now)
    - Ex: a 3D point is represented as a 3D* vector, rotations/translations/scales are represented as 3x3 matrices.



Object Coordinates

Camera (Eye) Coordinates

Projection Transform

Clip Coordinates

Normalized Device Coordinates

Window (Screen) Coordinates

Model & View Transform

Perspective Projection

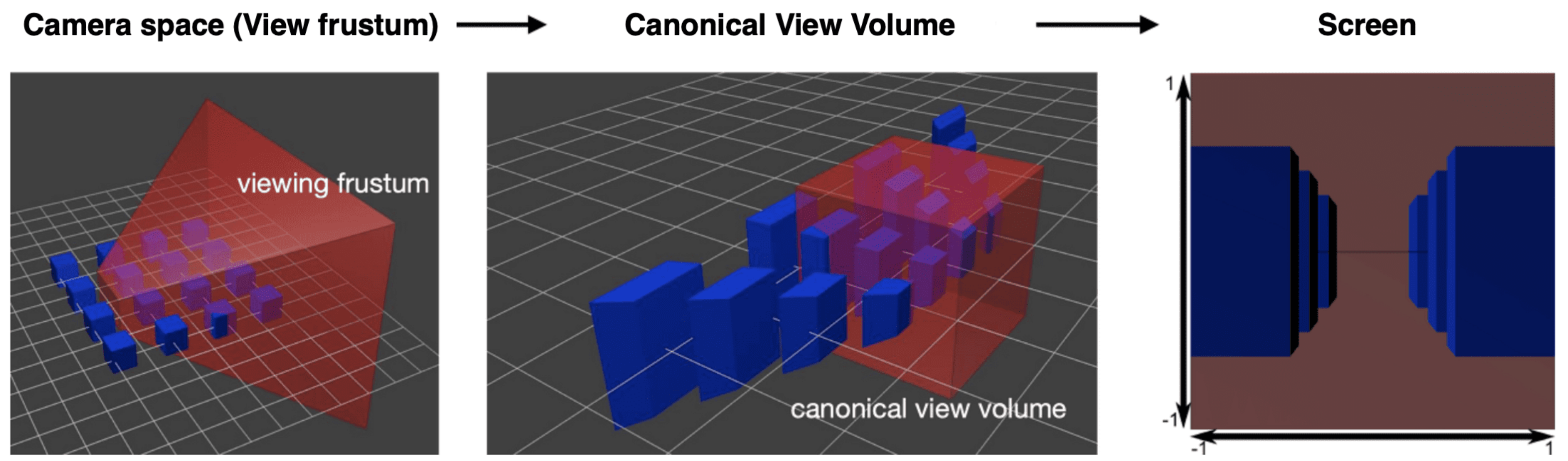
Perspective Division

Viewport Transform

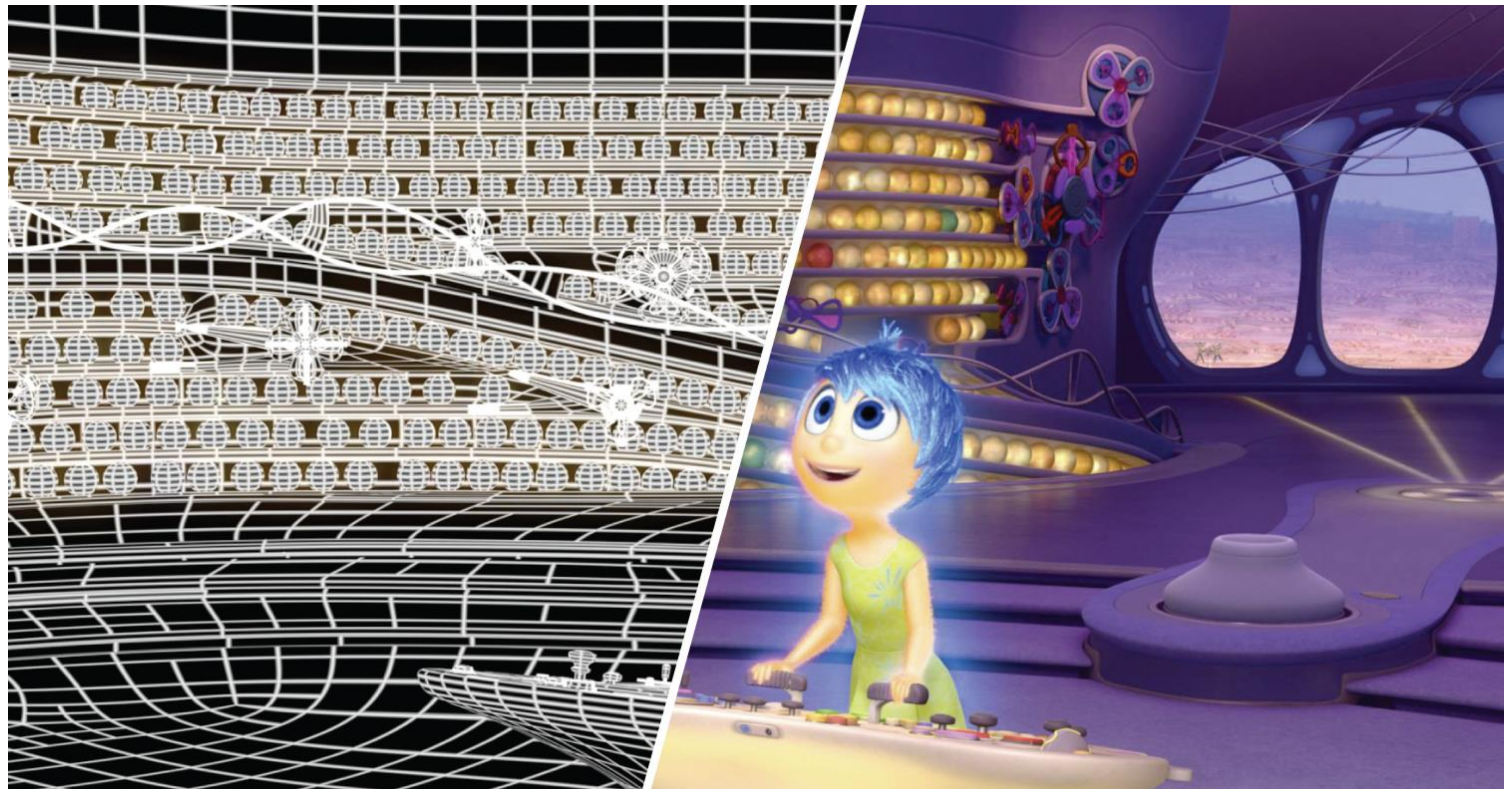
Rasterizer

Clippping Culling

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \quad \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad \mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Camera space (View frustum) → Canonical View Volume → Screen



* actually, we represent 3D points as a 4D vector [x, y, z, 1] ("homogenous" coordinates), and transformation matrices as 4x4 matrices, for good reasons (projective geometry).

https://leeyngdo.github.io/blog/computer-graphics/2024-02-29-graphics-pipeline/
https://vitaminac.github.io/Matrices-in-Computer-Graphics/
https://sciencebehindpixar.org/pipeline/rendering

What do GPUs excel at?

- GPUs can do a LOT of parallel computation, much more so than CPUs!
  - CPU: Typically has 4-16 cores (up to 8-32 active threads with hyperthreading)
  - GPU: A Tesla P100 GPU has 56 "Streaming Multiprocesesors", each with 2048 active threads (up to 114688 active threads!)
- If your computation can be easily parallelized, then GPUs are very good
- Fortunately, nearly all DNN code falls under this category!
  - Matrix/vector calculations (Linear, Conv2d, Relu, Softmax, etc.)
- (2017) "Tensor Cores" [link] are an Nvidia hardware feature that further accelerates certain DNN operations, particularly for lower-precision datatypes like float16 ("mixed precision training")

GPUs for ML

- AlexNet (2012) was an early (successful!) example of training a ConvNet on GPU hardware
- GPU acceleration caught on: DNN libraries began offering "first class" GPU support
- Caffe (2014): Developed at UC Berkeley. [link]
- Caffe2 (2017): Developed at Facebook. [link]
  - Note: basically deprecated in favor of pytorch
- Tensorflow (2015): Developed at Google [link]
- Pytorch (2016+): Developed at Facebook [link]

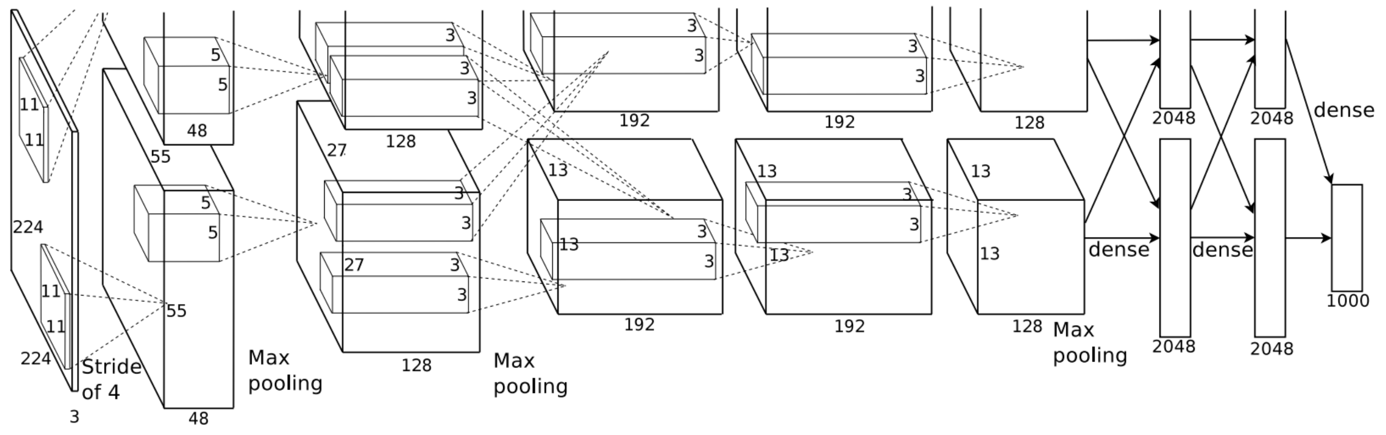


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Caffe2

TensorFlow

PyTorch

As of 2024, Pytorch and Tensorflow are the top DNN frameworks in use at both industry and academia. Personally: I prefer pytorch, but each has their strengths and weaknesses.

Nvidia: CUDA

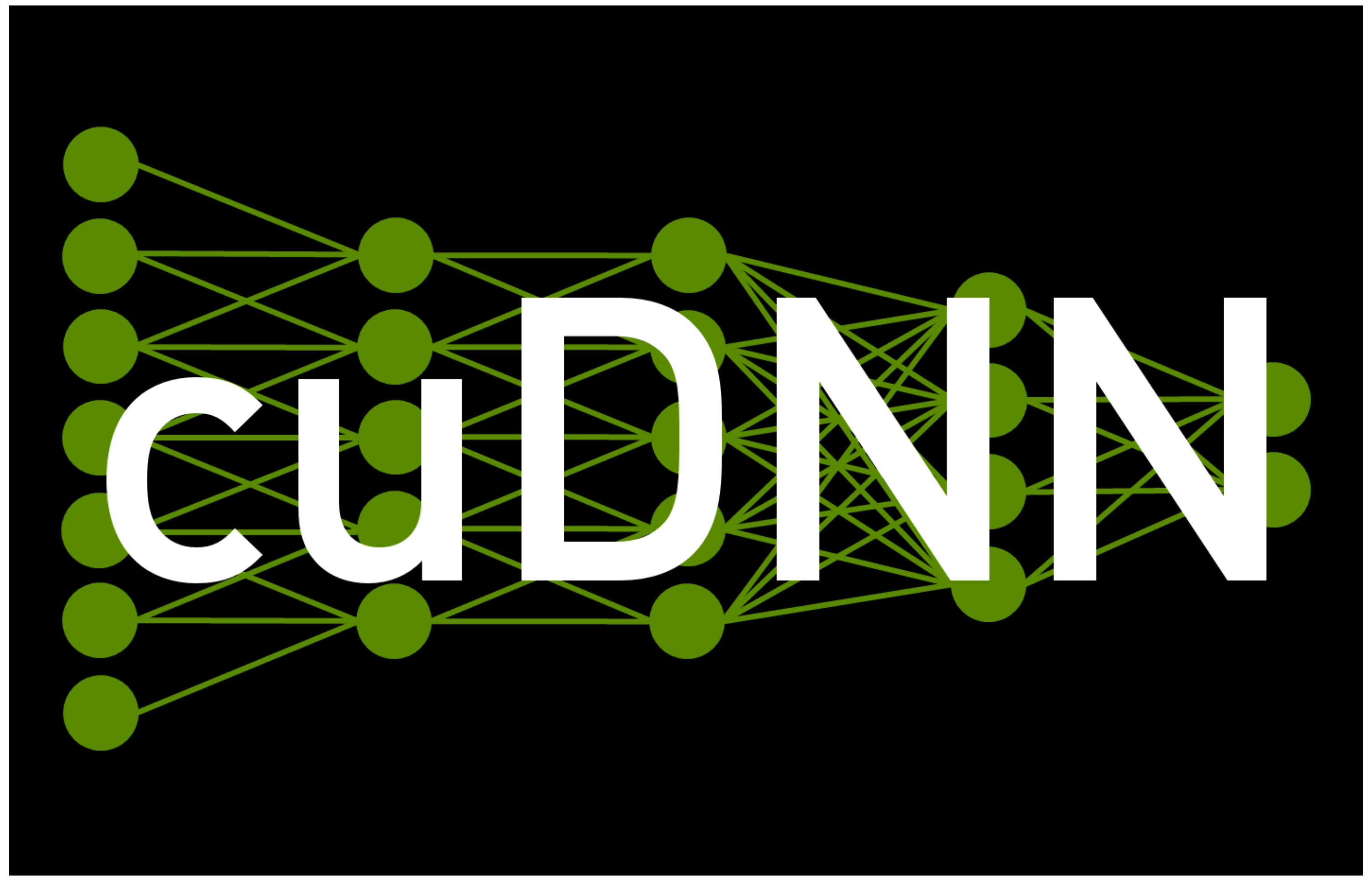
- CUDA: Compute Unified Device Architecture
- Low-level library that tells the GPU how to compute your desired code.
- CUDA code is basically C code
- Main idea: you write CUDA code that expresses your computation in a "parallelizable" way, to effectively utilize the GPU's many execution threads
  - Writing performant parallel code is an art! 99% of the time ML devs don't have to worry about this

```
__global__
void add(int n, float * x, float * y) {
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

Pictured: element-wise vector addition CUDA code. Vector-addition is easily parallelizable, so we can chunk up the input arrays ("blocks") and dispatch each block to different GPU execution threads to process in parallel!

Nvidia cuDNN

- cuDNN: an Nvidia library built on top of CUDA to provide high-performance DNN kernels (like optimized linear forward/backward, conv2d fwd/bkwd, etc)
- DNN frameworks like pytorch, tensorflow ultimately compile down to CUDA code (often via cuDNN calls) which is what actually runs on your GPU!



Pytorch + GPUs

- Pytorch makes it easy to use GPUs in your pytorch code!
- Terminology: all Tensors live on a `torch.device` [link]
  - By default: CPU device: `torch.device("cpu")`
  - GPU (cuda): torch.device("cuda")
    - (If your machine has multiple GPUs) torch.device("cuda:0"), torch.device("cuda:1"), ...
- Main principle: when doing an operation involving two tensors, both tensors must be on the same device!

Tensor(CPU) + Tensor(CPU): OK!
Tensor(GPU) + Tensor(GPU): OK!

Tensor(CPU) + Tensor(GPU): ERROR!

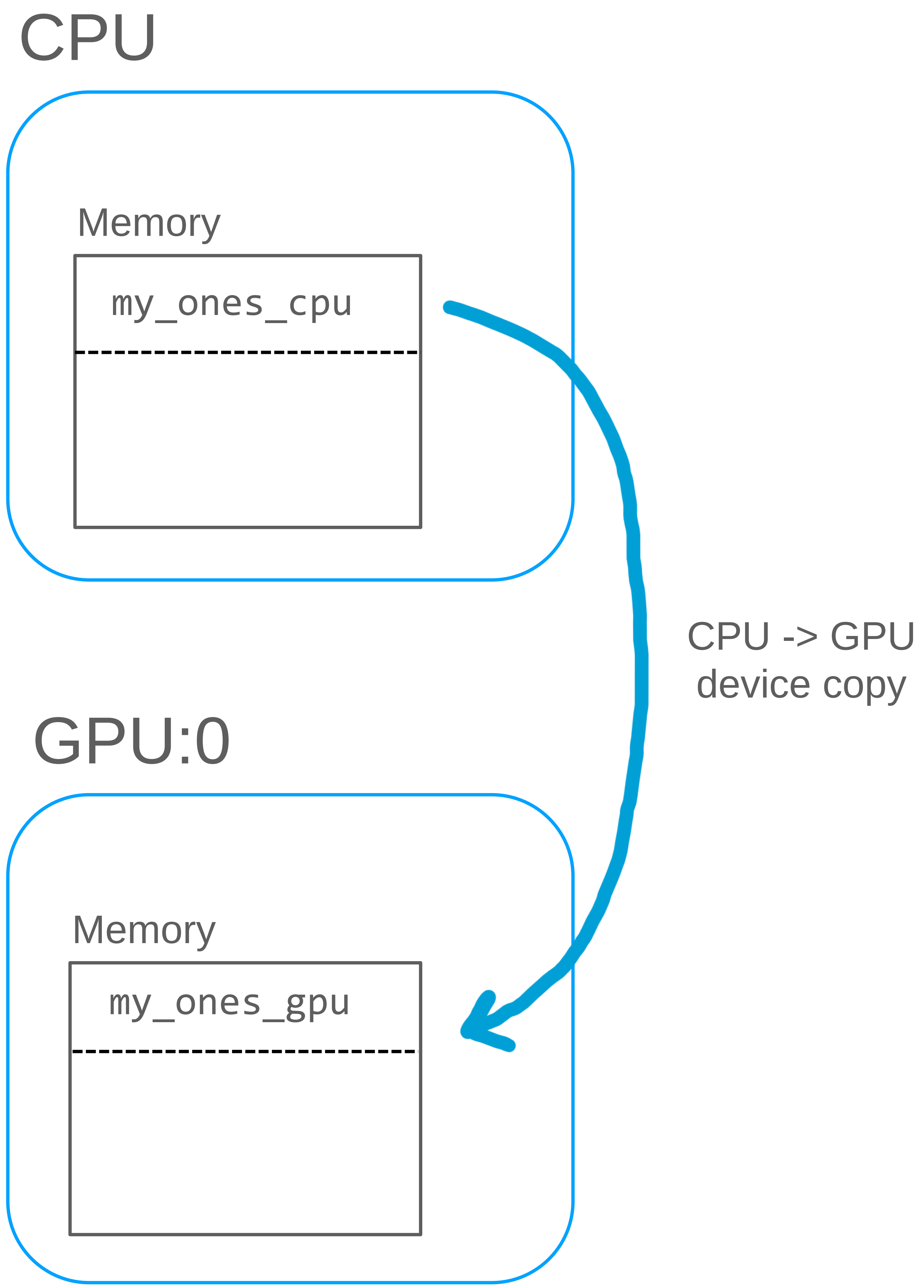
Pytorch example code: Tensor devices

```
# create tensor directly on cuda device
my_zeros_cuda = torch.zeros(size=[2, 3], device=torch.device("cuda:0"))
# create tensor on CPU, but move it to GPU (CPU -> GPU copy)
my_ones_cpu = torch.ones(size=[2, 4])
my_ones_cuda = my_ones_cpu.to(device=torch.device("cuda:0"))

# if you have multiple gpus, can send a tensor to a specific one
my_ones_cuda_device0 = my_ones_cpu.to(device=torch.device("cuda:0"))
my_ones_cuda_device1 = my_ones_cpu.to(device=torch.device("cuda:1"))

# finally, can send a tensor from GPU back to CPU
my_ones_cuda_to_cpu = my_ones_cuda.to(device=torch.device("cpu"))

# Beware: when doing ops between two tensors, they both most be on the
same device!
# RuntimeError: Expected all tensors to be on the same device, but found
at least two devices, cuda:0 and cpu!
my_ones_cpu + my_ones_cuda  # ERROR
```



CPU

Memory

my_ones_cpu

CPU -> GPU
device copy

GPU:0

Memory

my_ones_gpu

Pytorch example code: module.to(device)

```python
import torch, torch.nn as nn
class Autoencoder(nn.Module):
    def __init__(self, out_channels_first: int = 16):
        super().__init__()
        self.encoder = nn.Sequential( # like the Composition layer you built
            nn.Conv2d(1, out_channels_first, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(out_channels_first, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 7)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 7),
            nn.ReLU(),
            nn.ConvTranspose2d(32, out_channels_first, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(out_channels_first, 1, 3, stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )
    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

def print_statedict_info(module: torch.nn.Module):
    for param_key, param_val in module.state_dict().items():
        print(f"param_key={param_key}: {param_val.shape}, device={param_val.device}")
```

```python
model_a = Autoencoder()
print("CPU module")
print_statedict_info(model_a)
model_a_gpu = model_a.to(device=torch.device("cuda:0"))
print("GPU module")
print_statedict_info(model_a_gpu)
```

Takeaway: model.to(device=device) copies all of the model parameters to the target device (eg all weight/bias params)

```
(venv) PS
C:\Users\Eric\teaching\data_c182_fall2024\src\lectures\lecture19> python
.\module_gpu_demo.py
CPU module
param_key=encoder.0.weight: torch.Size([16, 1, 3, 3]), device=cpu
param_key=encoder.0.bias: torch.Size([16]), device=cpu
param_key=encoder.2.weight: torch.Size([32, 16, 3, 3]), device=cpu
param_key=encoder.2.bias: torch.Size([32]), device=cpu
param_key=encoder.4.weight: torch.Size([64, 32, 7, 7]), device=cpu
param_key=encoder.4.bias: torch.Size([64]), device=cpu
param_key=decoder.0.weight: torch.Size([64, 32, 7, 7]), device=cpu
param_key=decoder.0.bias: torch.Size([32]), device=cpu
param_key=decoder.2.weight: torch.Size([32, 16, 3, 3]), device=cpu
param_key=decoder.2.bias: torch.Size([16]), device=cpu
param_key=decoder.4.weight: torch.Size([16, 1, 3, 3]), device=cpu
param_key=decoder.4.bias: torch.Size([1]), device=cpu
GPU module
param_key=encoder.0.weight: torch.Size([16, 1, 3, 3]), device=cuda:0
param_key=encoder.0.bias: torch.Size([16]), device=cuda:0
param_key=encoder.2.weight: torch.Size([32, 16, 3, 3]), device=cuda:0
param_key=encoder.2.bias: torch.Size([32]), device=cuda:0
param_key=encoder.4.weight: torch.Size([64, 32, 7, 7]), device=cuda:0
param_key=encoder.4.bias: torch.Size([64]), device=cuda:0
param_key=decoder.0.weight: torch.Size([64, 32, 7, 7]), device=cuda:0
param_key=decoder.0.bias: torch.Size([32]), device=cuda:0
param_key=decoder.2.weight: torch.Size([32, 16, 3, 3]), device=cuda:0
param_key=decoder.2.bias: torch.Size([16]), device=cuda:0
param_key=decoder.4.weight: torch.Size([16, 1, 3, 3]), device=cuda:0
param_key=decoder.4.bias: torch.Size([1]), device=cuda:0
```

GPU memory: limited resource

- Beware: GPUs have a limited amount of memory. Exceeding GPU memory will lead to your train run being killed with a "GPU out of memory" error!
- Question: when training a DNN model, what are the main uses of GPU memory?
  - **Model weights**. Ex: Linear's weight/bias parameters.
  - **Intermediate activations**. If your model has N Conv2d's, then there will be N activation feature maps that pytorch has to keep track of that uses up GPU memory!
    - Scales linearly with your batch_size!
  - **Gradients**. Calculated during backwards()
  - **Additional optimizer state**. Ex: Adam requires `2*num_model_params` additional values (gradient moving avg, squared gradient moving avg)

Training model on GPU: Change 1/2

- Taking a CPU pytorch training code and migrating it to the GPU is (often) very easy, a one-line(s) change!
- Change 1: Move model (eg its model parameters) to the GPU device

```
# Create model (on CPU first, by default)
net = Net(hidden_num_chans=model_hidden_num_chans)

# Move model to GPU (if available to pytorch)
device_gpu_maybe = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")

print(f"(pre net.to) GPU max_memory_allocated: {torch.cuda.max_memory_allocated() / 1e6} MB")
net_gpu_maybe = net.to(device=device_gpu_maybe)
print(f"(post net.to) GPU max_memory_allocated: {torch.cuda.max_memory_allocated() / 1e6} MB")
```

```
(pre net.to) conv1.weight.device: cpu
(pre net.to) GPU max_memory_allocated: 0.0 MB
(post net.to) GPU max_memory_allocated: 0.481792 MB
(post net.to) conv1.weight.device: cuda:0
```

Note: `torch.cuda.max_memory_allocated()` tells us how much GPU memory we've used. This tells us that our model parameters takes up 0.48 MB of GPU memory. Neat!

Huzzah, our layers are on the GPU!

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#train-the-network

Training model on GPU: Change 2/2

- Change 2: move all model inputs (including targets/labels!) to the GPU before calling forward

```
def train_model(model: torch.nn.Module, optimizer, criterion, trainloader, num_epochs: int, device:
torch.device) -> torch.Tensor:
    for epoch in range(num_epochs):  # loop over the dataset multiple times
        running_loss = 0.0
        for ind_batch, data in enumerate(trainloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            # Note: dataloader outputs inputs, labels as CPU tensors
            inputs, labels = data
            inputs = inputs.to(device=device)
            labels = labels.to(device=device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels)
```

`device` is our GPU device (or CPU device if we don't have a GPU!)

Since `model` is on GPU, and `inputs, labels` are on GPU, we won't have errors like "mismatch device"

Implementation tip: this code is "device agnostic", in that it works for both CPU and GPU contexts (just pass in device=torch.device("cpu") or device=torch.device("cuda:0")). This is the way!

Demo: pytorch CPU vs GPU (single GPU)

- Demo: gpu_train_example.py

```
(venv) PS
C:\Users\Eric\teaching\data_c182_fall2024\src\lectures\lec
ture19> python .\gpu_train_example.py
Files already downloaded and verified
Files already downloaded and verified
batchsize=64, model_hidden_num_chans=128,
num_dataloader_workers=2
(CPU) Begin training (120078 model params)
(CPU) Finished Training (33.91305661201477 secs,
1474.3584033733462 imgs/sec)
(pre net.to) GPU max_memory_allocated: 0.0 MB
(post net.to) GPU max_memory_allocated: 0.481792 MB
(GPU) Begin training (120078 model params)
(post train_model) GPU max_memory_allocated: 96.306688 MB
(GPU) Finished Training (9.31104588508606 secs,
5369.966018542273 imgs/sec)
```

Device	Batchsize	Train throughput
CPU	64	1474 imgs/sec
GPU	64	5369 imgs/sec

Demo: pytorch CPU vs GPU (single GPU)

- Interestingly: the CPU doesn't always outperform the GPU! (here it does, but there are settings where it doesn't)
- A few rules of thumb:
  - Model should be big enough
  - Batchsize needs to be big enough too!
- Reason: too small model/batchsize means you spend most of your time doing CPU<->GPU communication, leading to poor GPU utilization
- **GPU's ideal computing mode**: operate on large data "all at once" (aka large batches), rather than small data one-at-a-time.

Device	Batchsize	Train throughput
CPU	64	1474 imgs/sec
GPU	64	5369 imgs/sec
CPU	4	1350 imgs/sec
GPU	4	1633 imgs/sec
CPU	2	860 imgs/sec
GPU	2	937 imgs/sec

Multi-GPU

- When working with large models + large datasets, we want to keep scaling up, ideally by throwing more compute (aka hardware, aka $) at the problem
    - Horizontal scaling: add more compute (machines, GPUs)
    - Vertical scaling: make each individual GPU faster
- Both are important! But at a given time, horizontal scaling is quicker+easier to do

Multi-GPU

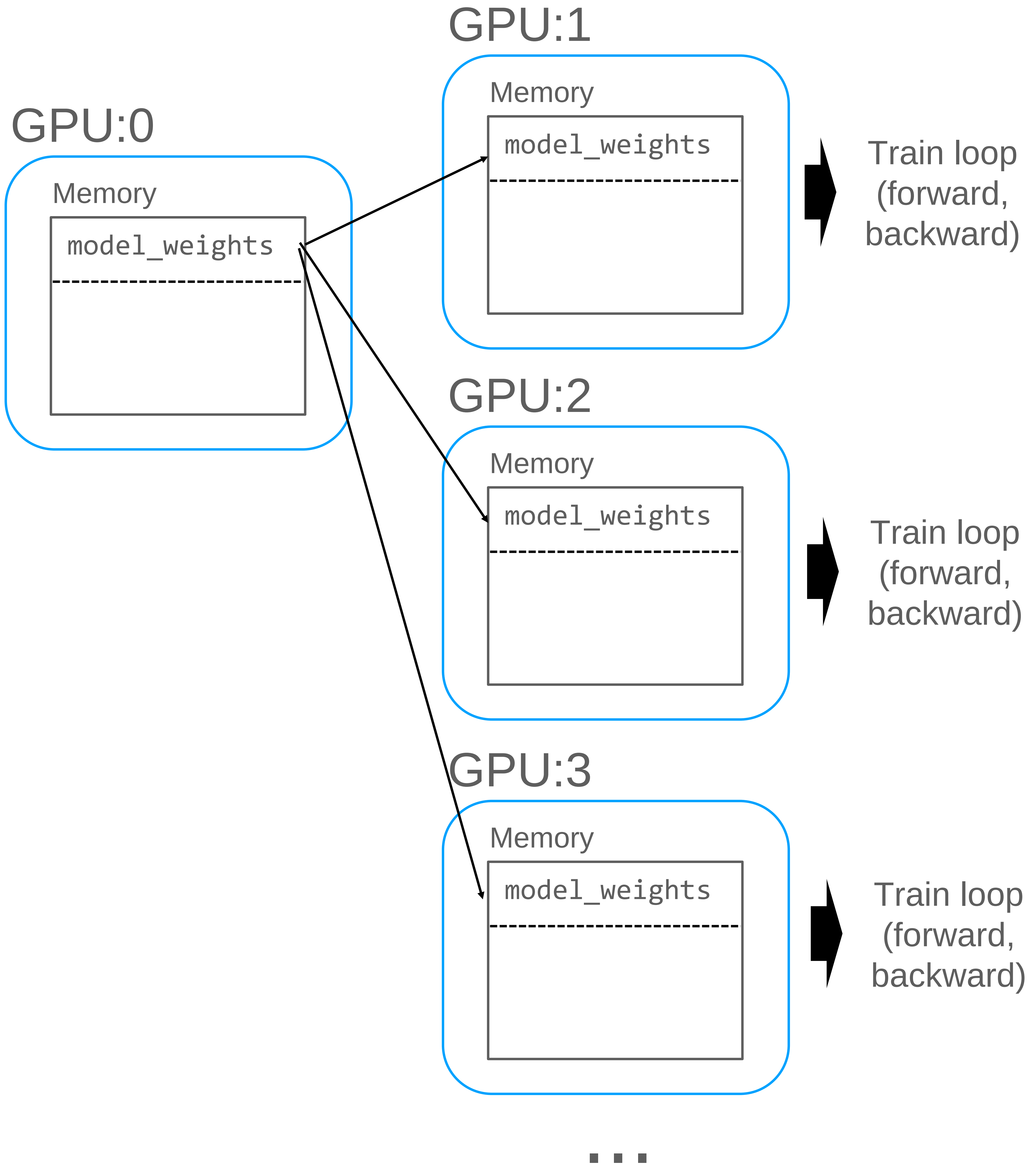
- Fortunately, we can easily* attach multiple GPUs to a single machine at a time
- Example: AWS EC2 p4d.24xlarge cloud instance type has 8 NVIDIA A100 Tensor Core GPUs, each with 40 GB GPU memory (quite large as of 2024!)
    - This is what many people in industry use to train large models (including my team). As of 2024-11, costs ~$290k per year to rent one!
- Engineering Question: how to effectively utilize multiple GPUs on a single machine for training DNN models?
- For simplicity: assume that all GPUs on a single machine are all the same type, eg same exact model (in practice this is true 99% of the time)

* Turns out that it's non-trivial to efficiently attach multiple GPUs on a single machine. See: Nvidia NVLink [link]

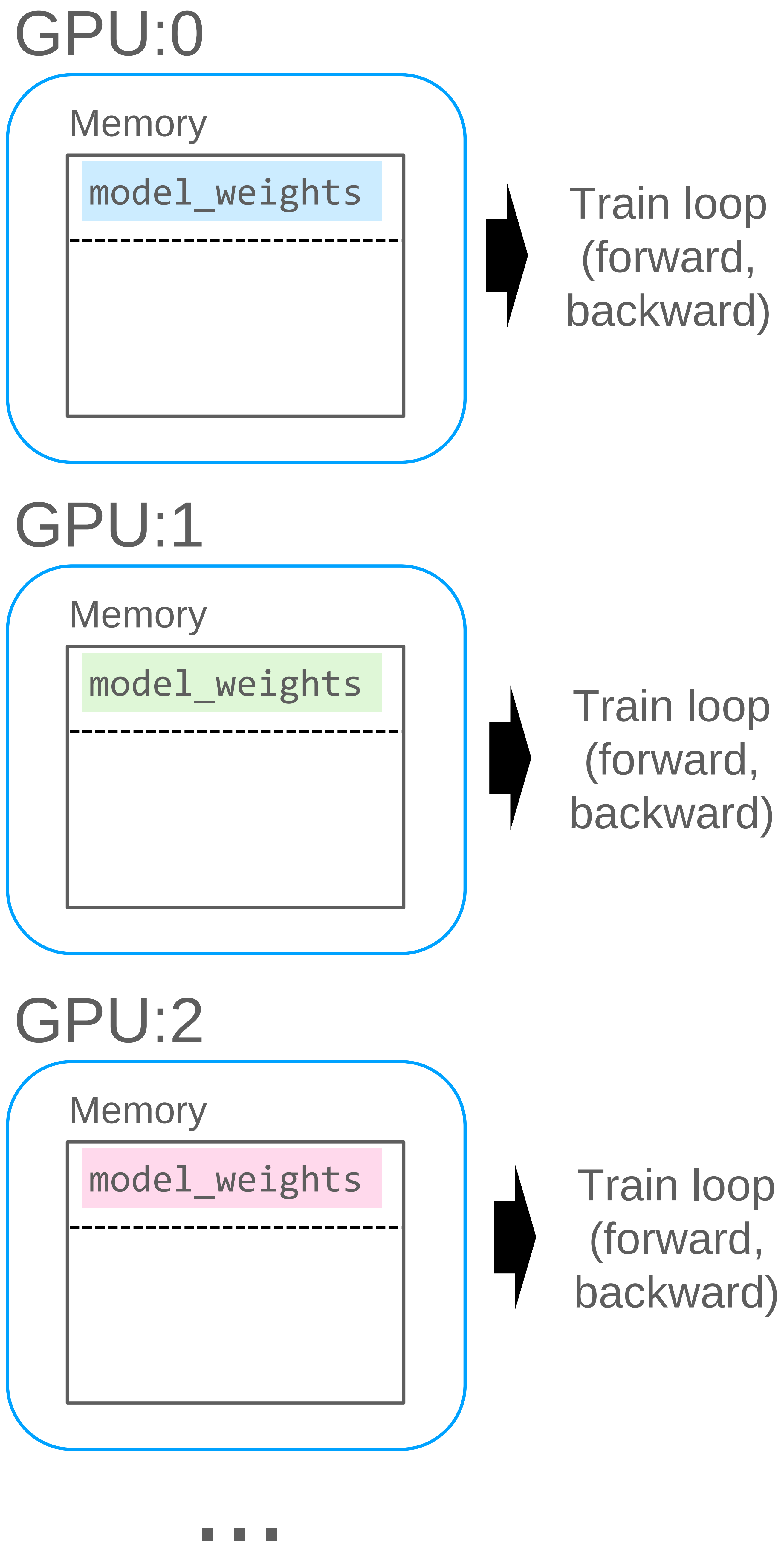
Multi-GPU: Scenario 1

- Suppose our machine has 8 GPUs, and our DNN model (and activations) can fit on a single GPU.
- Question: if we wanted to maximize training throughput, what's one way we can utilize the 8 GPUs?
- Answer: load the model onto all 8 GPUs separately, and have each GPU do their own forward/backward passes in parallel!

Implementation Note: each GPU trainer gets its own training Dataloader. Take care to ensure that each Dataloader splits up the training dataset appropriately (ex: don't want all N GPU workers to train on the same batches!)



Multi-GPU: Scenario 1


GPU:0
Memory
model_weights
Train loop (forward, backward)
GPU:1
Memory
model_weights
Train loop (forward, backward)
GPU:2
Memory
model_weights
Train loop (forward, backward)
...

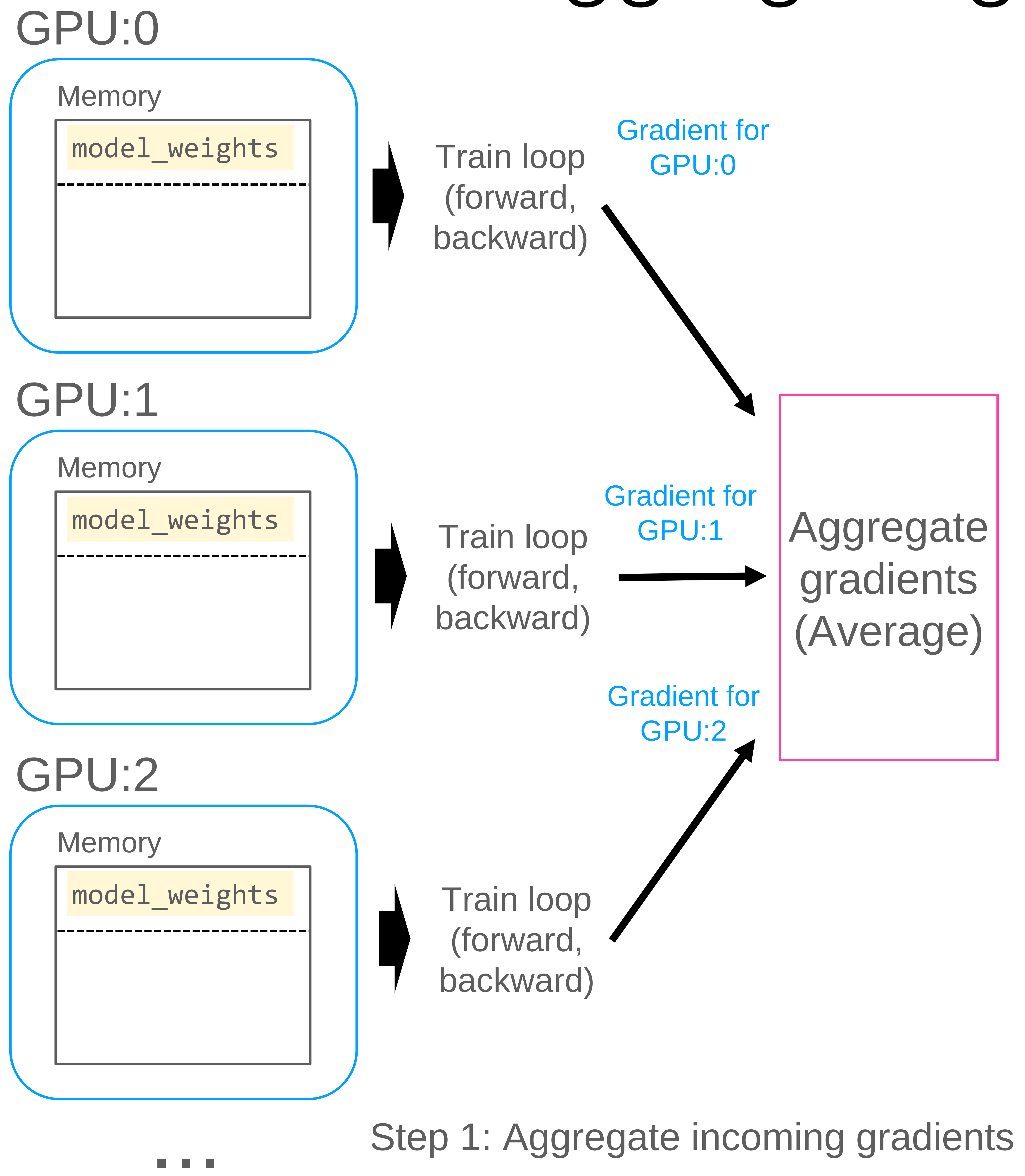
Question: suppose each GPU worker has different training dataset splits (aka each GPU worker operates on different train batches).
After N training batches, will the model parameters be the same across all GPU workers?

Answer: Nope! Each GPU worker's train loop will process different batches, thus the gradients for GPU:0 will be different than GPU:1, resulting in different model weights for each worker. This complicates things!
Question: how can we ensure that, after each training batch (for all N GPUs), the model_weights are the same ("in sync") for all N GPUs?

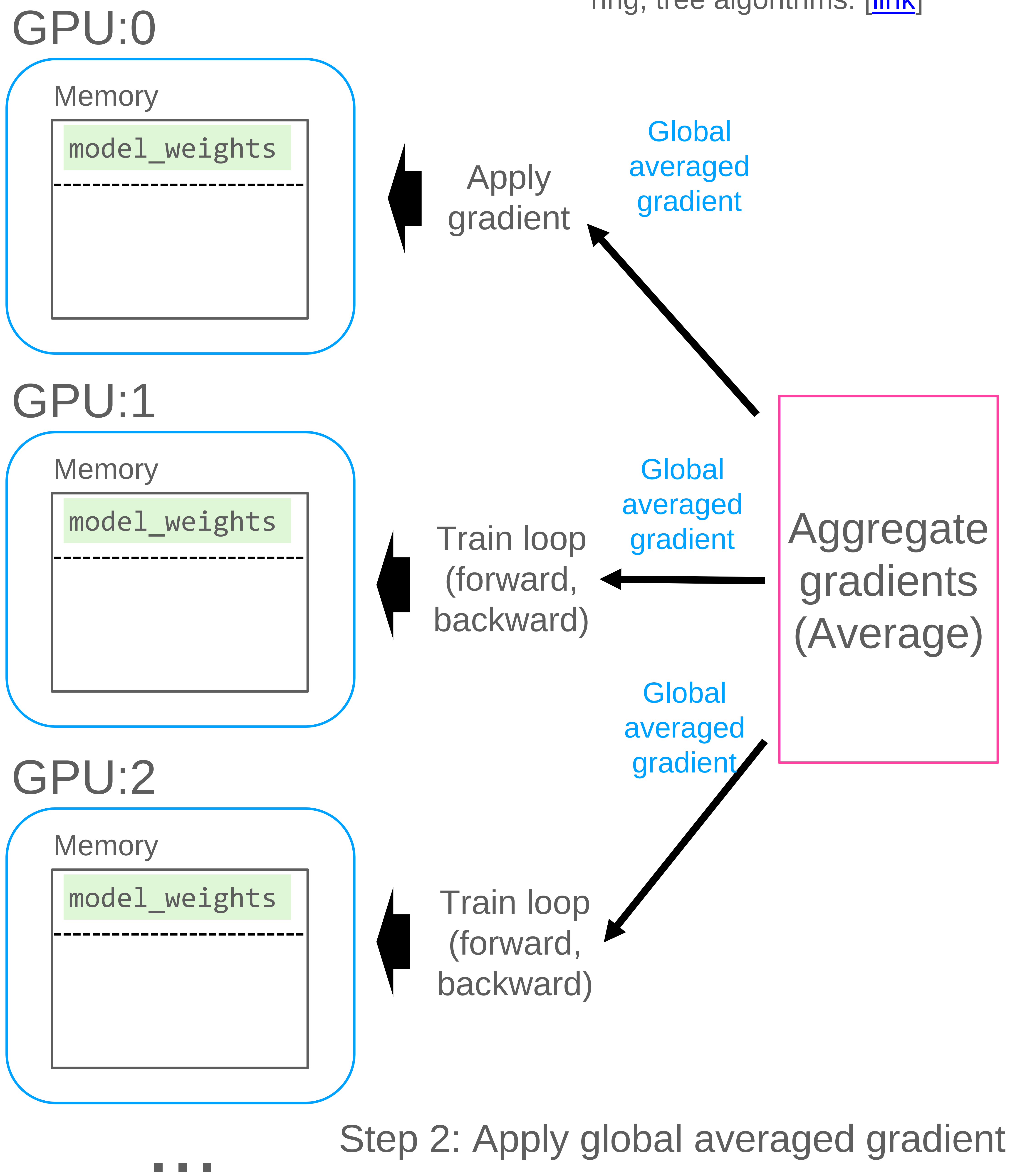
Answer: one popular way is to aggregate all of the GPU worker's gradients (average), then have each GPU worker apply the same aggregated ("global average") gradient.
Ensures that model_weights is the same on all GPU workers!

Multi-GPU: Aggregate gradients

Note: there exist other more efficient, cleverer ways of optimizing this. Ex: NCCL's ring, tree algorithms: [link]



Step 1: Aggregate incoming gradients



Step 2: Apply global averaged gradient

Pytorch: DistributedDataParallel (DDP)

- Enter: torch.nn.parallel.DistributedDataParallel (or DDP) [link]
- DDP does exactly what we proposed! Consists of two parts:
  - **Model copy**. copy the initial model weights to each GPU, via `.to(device)`
  - **Gradient synchronization**. adds a "hook" to the backwards() pass to calculate the aggregated gradients across all GPUs (via averaging)
    - This ensures that all GPUs use the same gradients for each step, which ensures that each GPU always has the same model weights
- Each GPU processes their own batches independently
- For 8 GPUs, you've effectively increased your batchsize by 8x!

Terminology: world_size, rank

- world_size: the total number of workers. Ex: total number of GPUs
- rank: an integer between [0, world_size-1] (inclusive).
- Setup: each GPU will be assigned a "rank" and the "world_size", and will use this metadata to perform its job
- Ex: Suppose world_size is 8.
- A GPU worker assigned rank=0 will use GPU0: `torch.device("cuda:0")`
  - Rank=1 will use GPU1: `torch.device("cuda:1")`
  - ...
  - Rank=7 will use GPU7: `torch.device("cuda:7")`

DDP in pytorch

```
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP

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 torch_elastic to launch script N times, one
# for each GPU worker
# --nnodes: number of nodes. For us, 1 machine
# --nproc_per_node: number of GPUs. For us: 8
torchrun --nnodes=1 --nproc_per_node=8 --rdzv_id=100 --
rdzv_backend=c10d --rdzv_endpoint=$MASTER_ADDR:29400
elastic_ddp.py
```

```
def demo_basic():
    dist.init_process_group("nccl")
    rank = dist.get_rank()
    print(f"Start running basic DDP example on rank {rank}.")
    # create model and move it to GPU with id rank
    device_id = rank % torch.cuda.device_count()
    model = ToyModel().to(device_id)
    ddp_model = DDP(model, device_ids=[device_id])
    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(device_id)
    loss_fn(outputs, labels).backward()
    optimizer.step()
    dist.destroy_process_group()
    print(f"Finished running basic DDP example on rank {rank}.")

if __name__ == "__main__":
    demo_basic()
```

Boilerplate code to initialize distributed backend (NCCL)

`device_id` is the assigned GPU device to use: f"cuda:{device_id}"

Wrap model on DDP()

Can use `ddp_model` just like regular `model`! Very convenient.

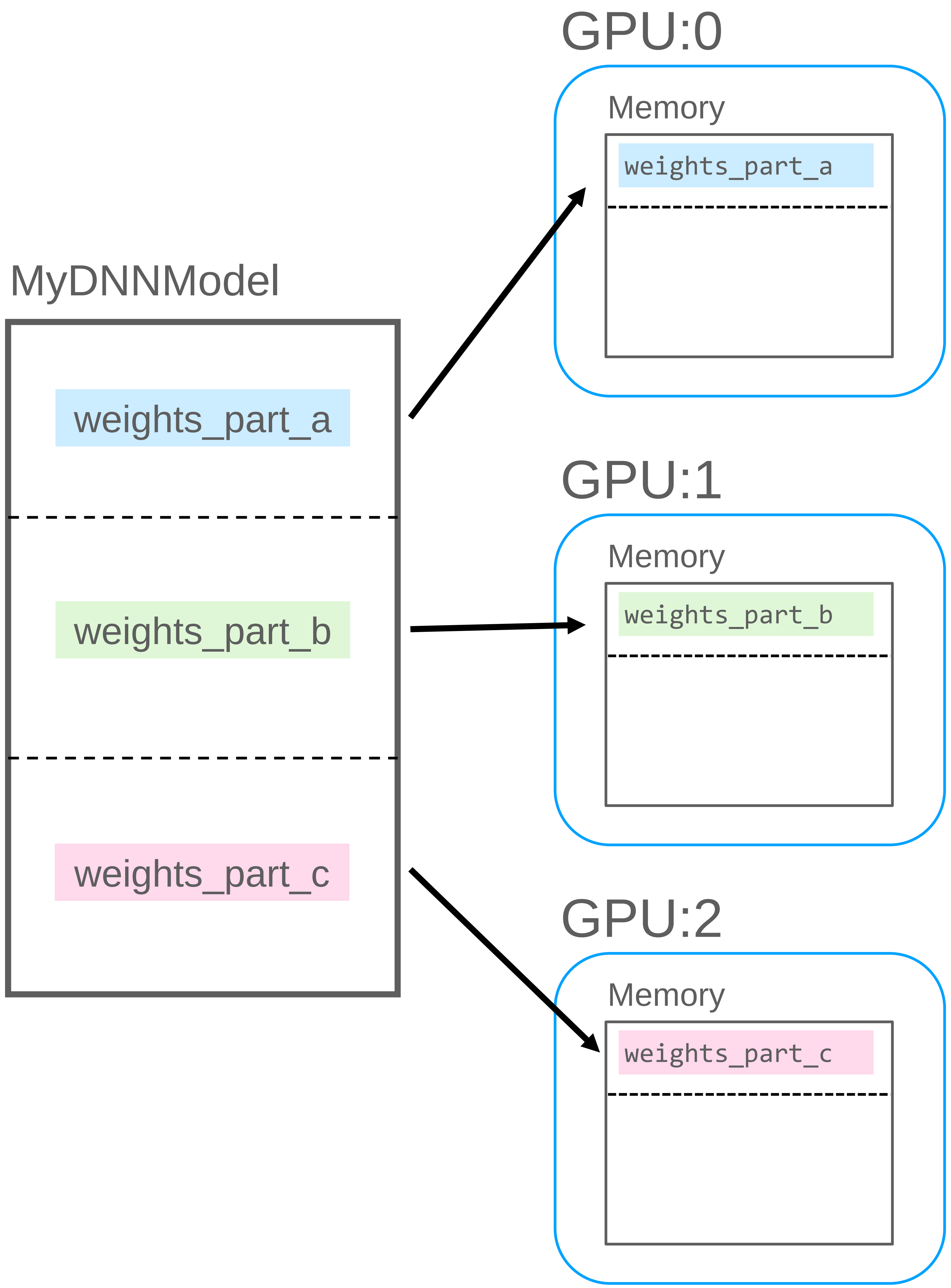
For more info, see: https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

DDP: sync and all_reduce (NCCL)

- An important part of DDP is the gradient synchronization
- For N GPUs, each GPU will independently perform their own "local" forward/backwards pass to calculate N different gradient updates
- DDP's job is to take all N gradient updates, average them, then transmit this aggregated gradient to all N GPUs so that each GPU can perform their weight updates.
- Common name for this operation: "all_reduce"
- Pytorch supports dispatching the all_reduce call to a variety of libraries [link]
  - For Nvidia GPUs: use NCCL [link]

Multi-GPU: Scenario 2

- Suppose our machine has 8 GPUs, and our DNN model (and activations) can't fit on a single GPU.
- Question: how can we utilize the 8 GPUs to train our DNN model?
- Answer: split the model up across multiple GPUs!
- Question: what downsides can you think of?
- Answer: slower training, due to additional cross-GPU communication



Pytorch: FullyShardedDataParallelism (FSDP)

- Pytorch's FSDP implements this "model sharding" idea [link]
- Implementation challenge: how to distribute model parameters (eg layers) in a way that minimizes cross-gpu device copies?
- Heuristic: FSDP works well for very large models, eg when model weights are a substantial fraction of available GPU memory
- FSDP is also handy to increase total batchsize at the expense of some efficiency

DDP vs FSDP

Name	Impact on: GPU memory	Impact on: Train throughput	When to use?
DistributedDataParallel (DDP)	For N GPUs, model weights are duplicated N times (waste). Con!	Each GPU does forward/backward independently: no cross-GPU communication required (except for all_reduce). Pro!	If your model+activations comfortably fits in GPU memory AND you are happy with your current batch_size
FullyShardedDataParallel (FSDP)	Model weights are instantiated only once (split across N GPUs). Pro!	Training throughput is worse due to cross-GPU communication. Con!	If your model can't fit on a single GPU: MUST use FSDP. Or: if you need a higher batchsize but it won't fit with DDP: try FSDP!

Multi-Node, Multi-GPU: DDP

- Suppose we have M machines, each with N GPUs. How to effectively utilize this for training DNN models?
    - Fortunately, the earlier principles generalize quite nicely!
- DDP:
    - M=1 machine, N GPUs: all_reduce across GPUs (within a single machine)
    - M>1 machines, N GPUs each: all_reduce across ALL GPUs
        - Involves cross-machine communication during gradient sync!
- FSDP
    - M=1 machine, N GPUs: split model parameters across N GPUs
    - M>1 machines, N GPUs each: split model parameters across M*N GPUs
        - Involves cross-machine comm during forward/backward

DDP in pytorch (multi-node)

```
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP

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)))

def demo_basic():
    dist.init_process_group("nccl")
    rank = dist.get_rank()
    print(f"Start running basic DDP example on rank {rank}.")
    # create model and move it to GPU with id rank
    device_id = rank % torch.cuda.device_count()
    model = ToyModel().to(device_id)
    ddp_model = DDP(model, device_ids=[device_id])
    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(device_id)
    loss_fn(outputs, labels).backward()
    optimizer.step()
    dist.destroy_process_group()
    print(f"Finished running basic DDP example on rank {rank}.")

if __name__ == "__main__":
    demo_basic()
```

```
# use torch_elastic to launch script N times, one
# for each GPU worker
# --nnodes: number of nodes. For us, 4 machines
# --nproc_per_node: number of GPUs. For us: 8
torchrun --nnodes=4 --nproc_per_node=8 --rdzv_id=100 --
rdzv_backend=c10d --rdzv_endpoint=$MASTER_ADDR:29400
elastic_ddp.py
```

Now, `rank` spans multiple machines.
Ex: if each machine has 8 GPUs, then:
Rank=[0, 1, ..., 7]: Machine 0
Rank=[8, 9, ..., 15]: Machine 1

Other than that, everything is nearly the same! Pytorch set things up nicely so that very little has to change when scaling from M=1 machines to M=4 machines. The world_size/rank abstraction makes things very convenient :)

For more info, see: https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

Example: "Train ImageNet in 1 hour"

- In "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour" (2018) [link] the authors, using a large distributed GPU cluster, trained a ResNet-50 model on ImageNet-1k "from scratch" in one hour. A neat accomplishment for that time!
  - Hardware: 256 GPUs ("Big Basin" [link] GPU cluster internal to Facebook, 16GB GPU mem per card. Nvidia Tesla P100).
- Distributed training setup: basically DDP (they used Caffe2, not pytorch, but same idea)

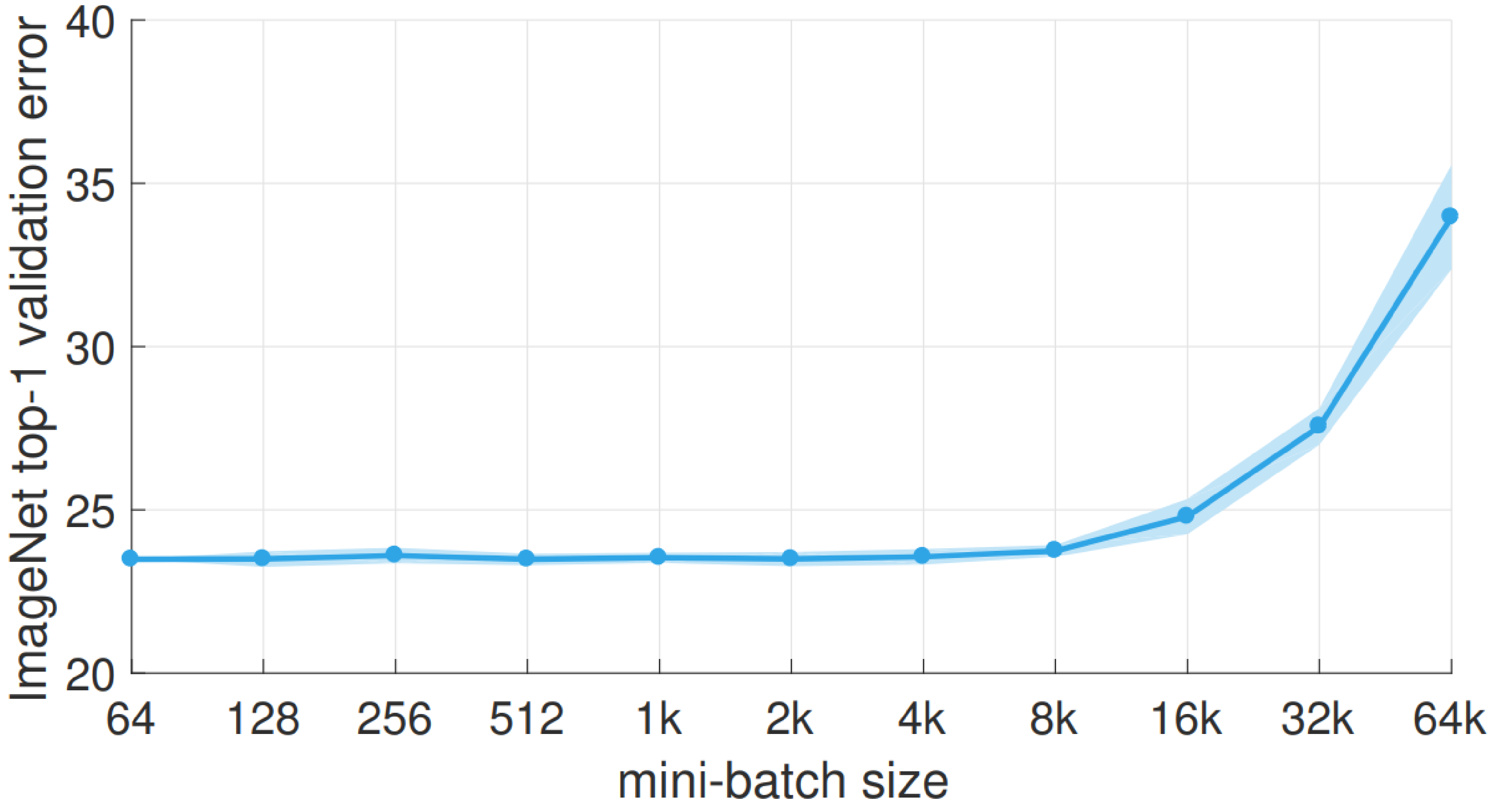


Figure 1. **ImageNet top-1 validation error *vs*. minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*. For all minibatch

Learning rate: Linear scaling rule

- Learning: when scaling up the number of GPUs (aka increasing the effective batchsize), one must adjust the learning rate accordingly ("linear scaling rule").
  - Rule: double the batchsize -> double the learning rate.

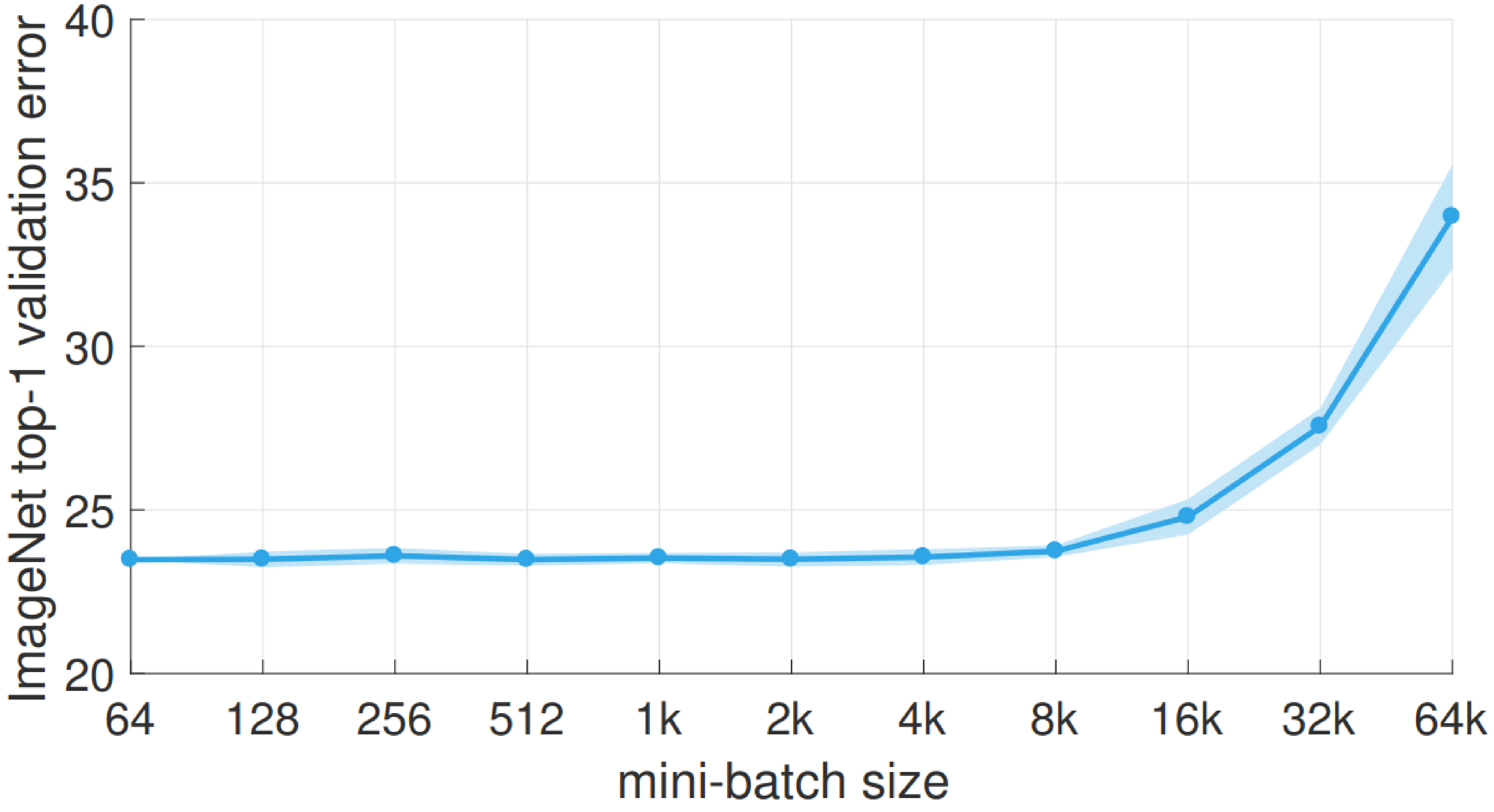


Figure 1. **ImageNet top-1 validation error *vs*. minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*. For all minibatch

Gradient quality vs num steps?

- Observation: if you keep the number training epochs fixed, then increasing the batchsize leads to fewer model updates.
  - Higher batchsize -> higher quality gradient updates, but fewer parameter updates
  - Lower batchsize -> noisier gradient updates, but more parameter updates.
- What is best? Paper's answer: higher batchsize AND higher learning rate.
  - ...to a point. Beyond batchsize=8k, classification error **starts increasing**.

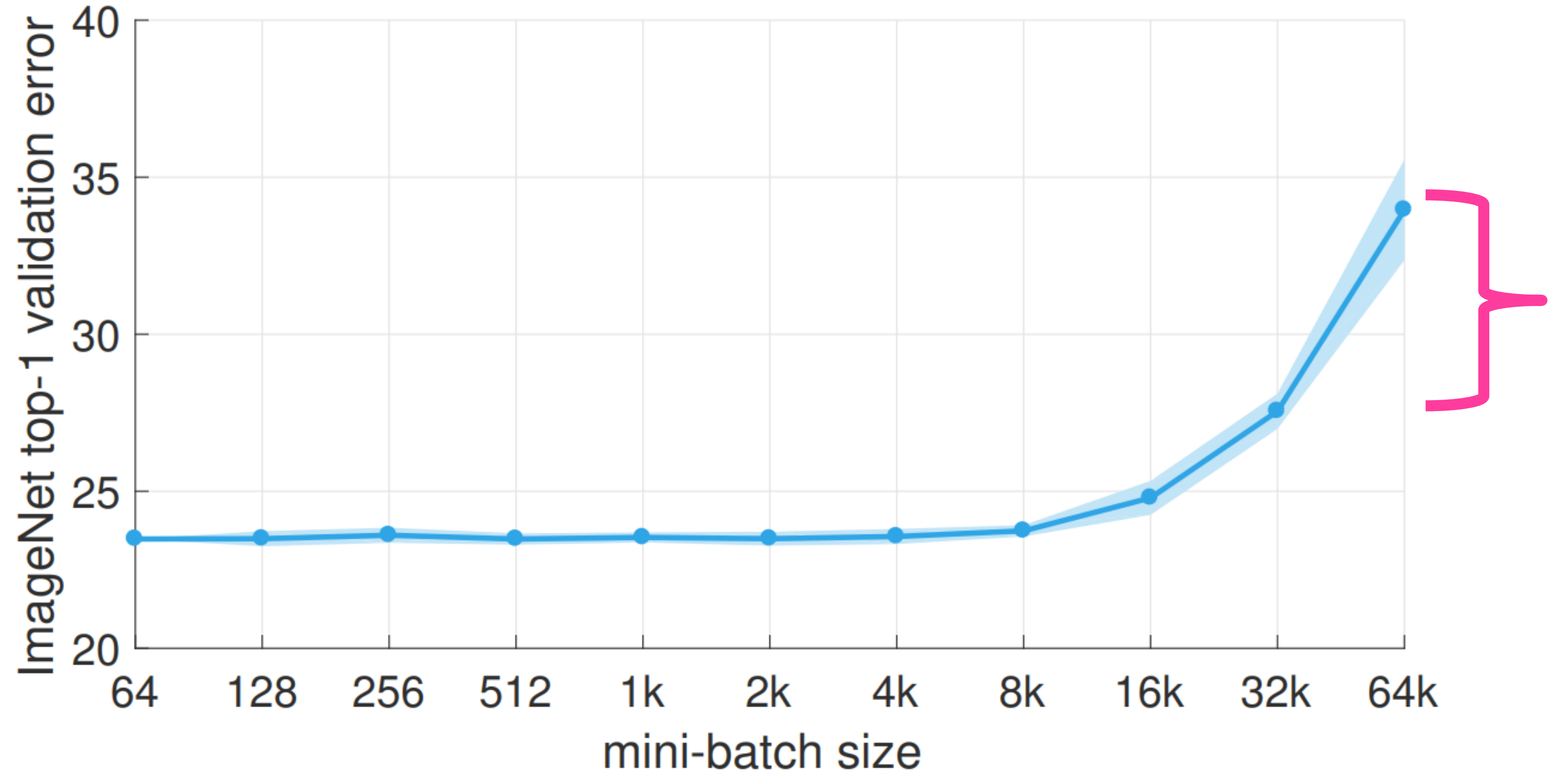


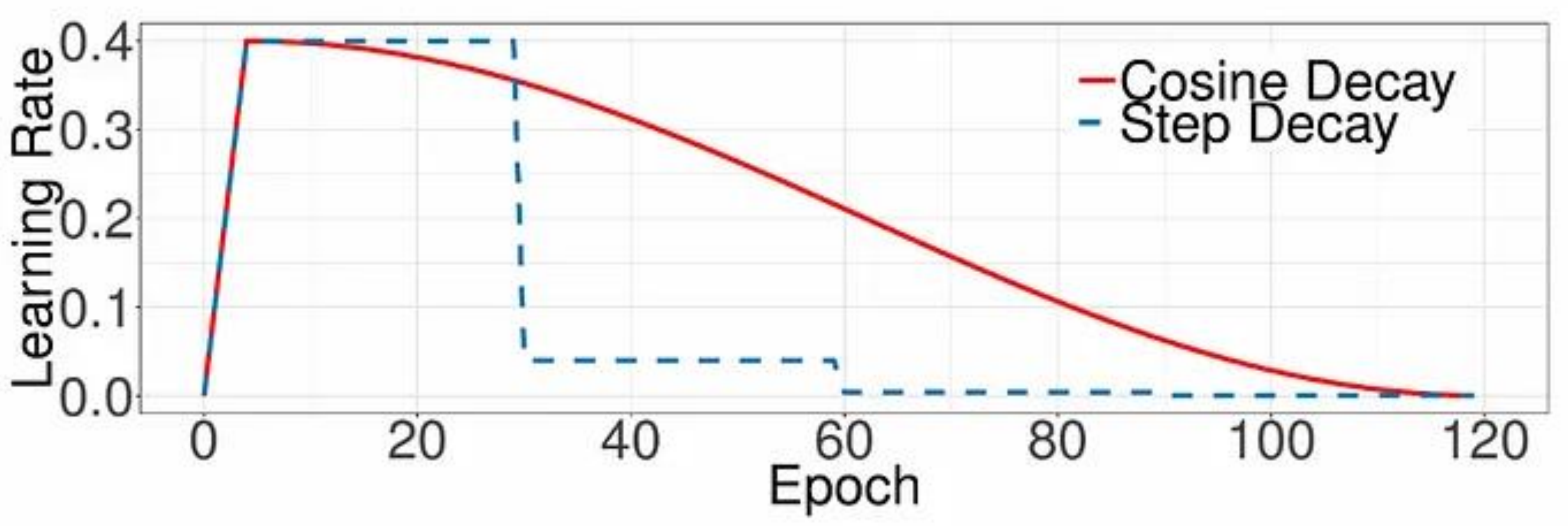
Figure 1. **ImageNet top-1 validation error *vs.* minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*. For all minibatch

"Train ImageNet in 1 hour": Hardware advances

- A sign that GPU hardware (and DNN libraries + distributed training frameworks) is advancing quickly
- **...And, a hint that ImageNet-1k is starting to feel small!**
- (later in Aug 2018, someone showed we can train ImageNet in 18 mins for $40 using AWS cloud! [link])
- In 2024, I bet things are even faster + cheaper! Technology marches on...

Tangent: Learning rate schedules

- So far in this class, we've used a single learning rate. In practice, it's better to use learning rate schedules
- Start learning rate small, then gradually ramp it up to a larger value (eg the first ~100 iterations)
  - Intuition: starting learning rate too high often leads to training divergence (eg NaN losses). Thus, we start it low to get the model weights in a "healthy" region, then slowly increase the learning rate
- Over the course of training, decay the learning rate
  - Intuition: during early parts of training, model needs to make big steps (high LR). But, near the end of training, model is focusing on finer-grained details (small LR).



https://pub.towardsai.net/are-you-sure-that-you-can-implement-image-classification-networks-d5f0bffb242d

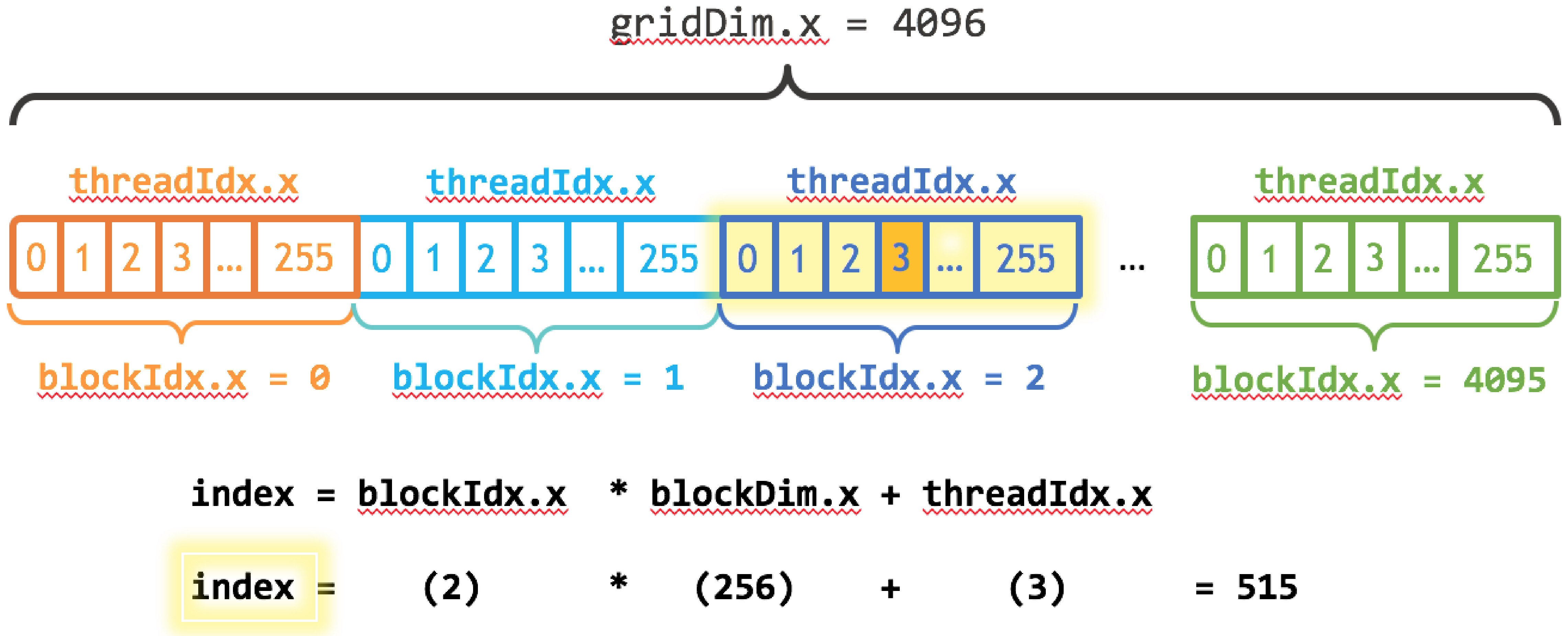
(unused) CUDA example: element-wise vector addition

Consider: elementwise-vector addition. Given two vectors x, y (with shape=[N]), output (x+y).
This is an "embarrassingly parallel" problem: chunk up the input vectors into K chunks, and process each chunk independently in parallel!

```
__global__
void add(int n, float * x, float * y) {
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

Ex: for vector element-wise addition, one way to do it ("embarassingly parallel"):
- Break the input vector into numBlocks chunks
- within each block, have a separate thread perform a single `y[i] = a[i] + b[i]`

Here, we divide the input array into blocks. Within each block, we assign threads to each block element.



To learn more, see: https://developer.nvidia.com/blog/even-easier-introduction-cuda/