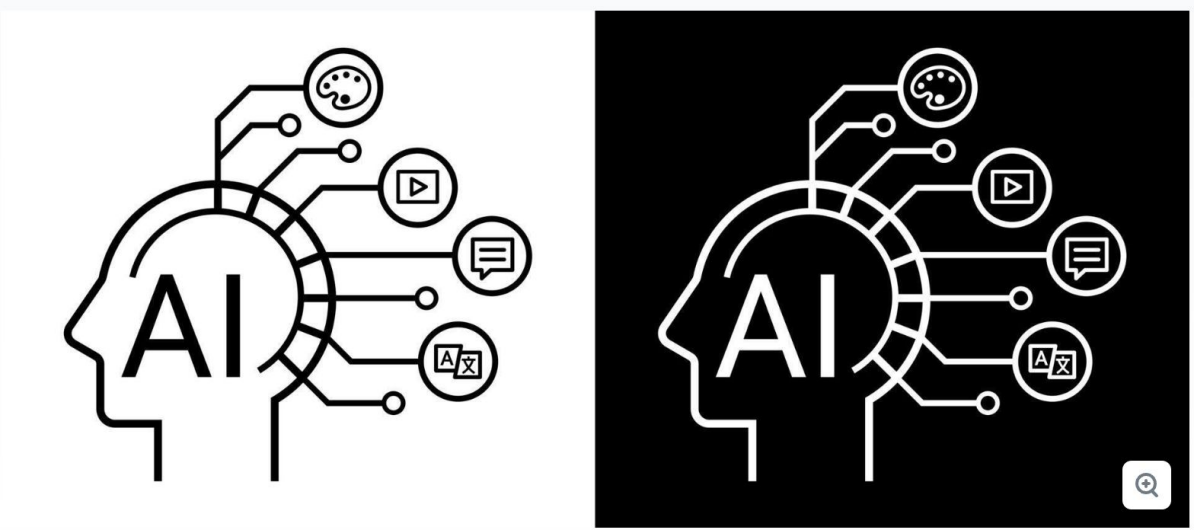
AIMS 5740
Generative Artificial Intelligence

(2026 Term 2)



Computer Science & Engineering
The Chinese University of Hong Kong

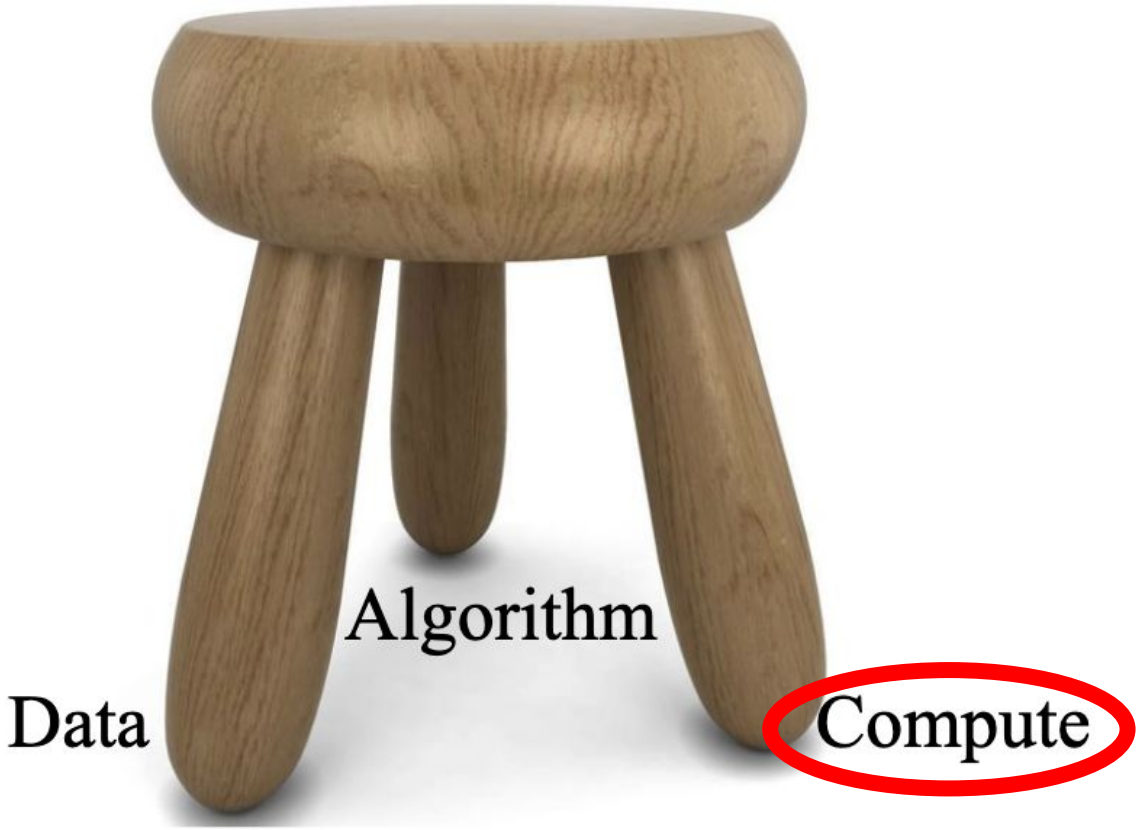
Announcement

- We will have a tutorial next Wed and the first assignment following that.

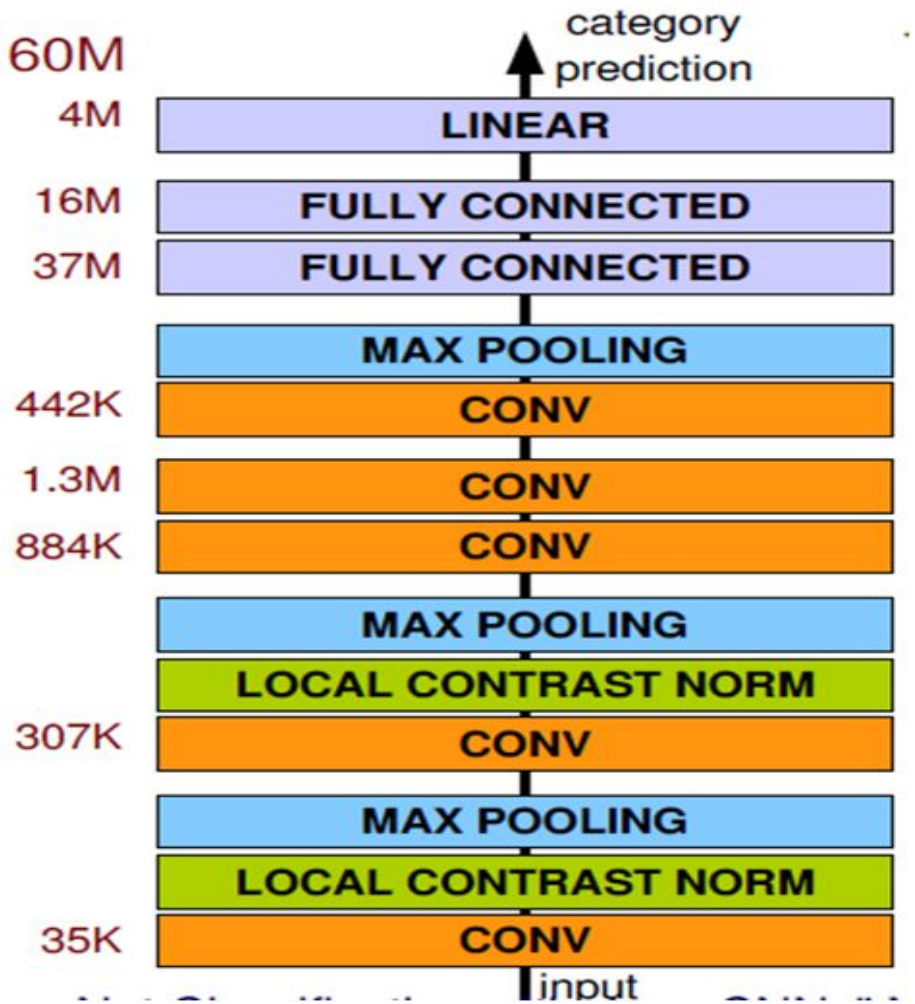
- Our course slides will be released on the website one hour before the class.

Generative AI

- Compute is another key components for GenAI.



Computational Resource



Architecture of Alexnet

Q: what kind of framework/platform do we run DNN on?

7 hidden layers, 650,000 neurons 60,000,000 parameters

CPU and GPU

CPU (central processing unit)

GPU (graphical processing unit)

CPU vs. GPU

CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

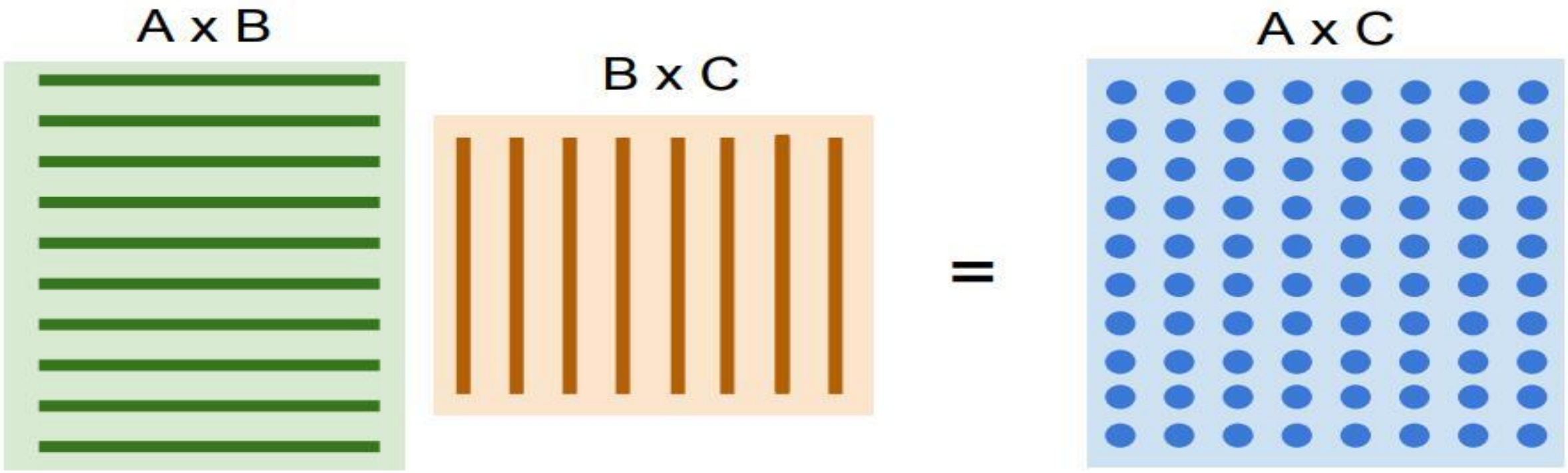
GPU: More cores, but each core is much slower and “dumber”; great for parallel tasks

	# Cores	Clock Speed	Memory	Price
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.4 GHz	Shared with system	$339
CPU (Intel Core i7-6950X)	10 (20 threads with hyperthreading)	3.5 GHz	Shared with system	$1723
GPU (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	$1200
GPU (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	$399

Nvidia GPUs

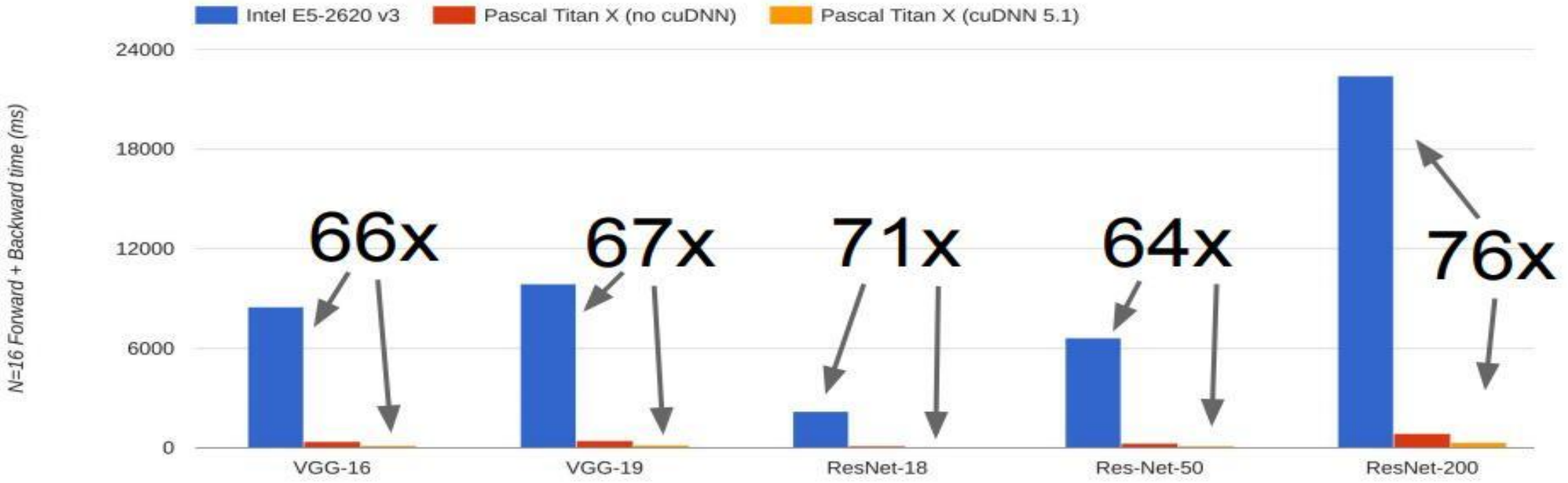
	A100s	H100s	H200s
Architecture Name	Ampere	Hopper	Hopper
Memory (GB)	40/80 GB	80 GB	141 GB
Bandwidth	2.0 Tbps	3.4 Tbps	4.8 Tbps
TDP	250-400 Watts	700 Watts	700 Watts
Cooling	Air	Air/Liquid	Liquid
AI-Optimized Design	Indirectly	Yes	Yes

Need GPU?



Example: Matrix Multiplication

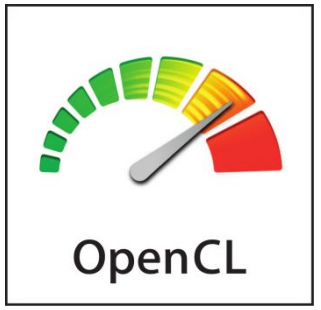
CPU and GPU in Practice



Programming GPUs

- CUDA:
  - C-library which runs directly on NVIDAI GPUs
  - High-level APIs: cuDNN, cuFFT etc.

- OpenCL:
  - Can run on any GPUs
  - C based code, usually slower than CUDA



Deep learning platform

Caffe
(UC Berkeley) → Caffe2
(Facebook)

Torch
(NYU / Facebook) → PyTorch
(Facebook)

Theano
(U Montreal) → TensorFlow
(Google)

Paddle
(Baidu)

CNTK
(Microsoft)

MXNet
(Amazon)

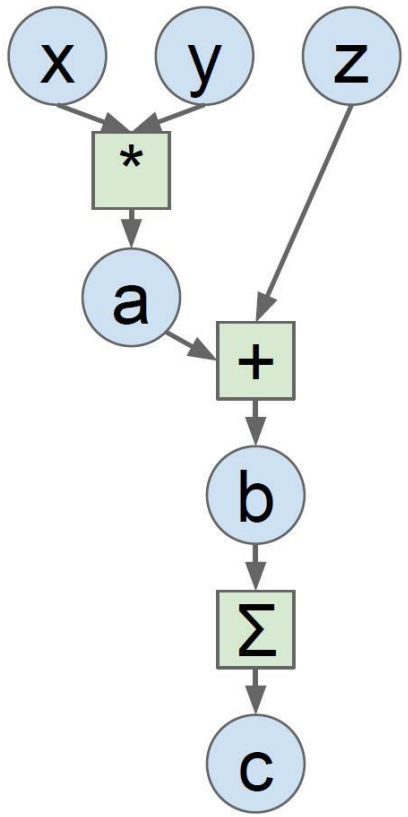
Points of Deep Learning Platform

- Easily build complicated network
- Easily compute gradients in computational graphs
- Run it all efficiently on GPU (wrap cuDNN, cuBLAS, etc)

Numpy

Numpy

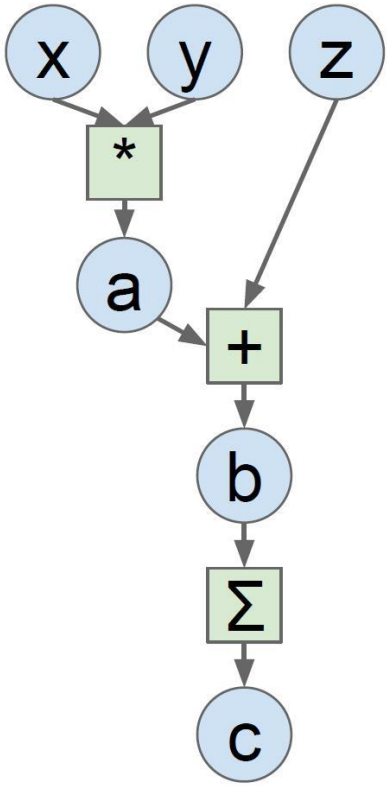
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Numpy

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

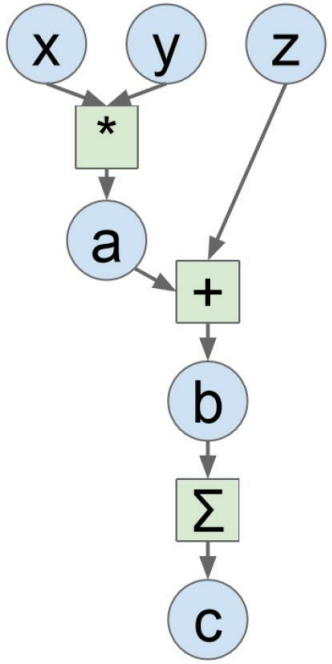


Problems:

- Can't run on GPU
- Have to compute our own gradients

TensorFlow

Computational Graphs



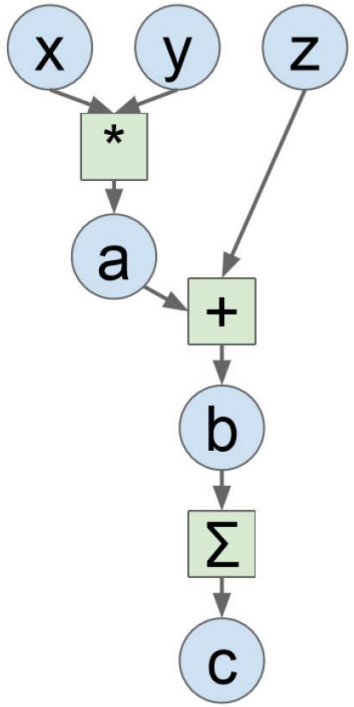
Create forward computational graph

TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

TensorFlow

Computational Graphs



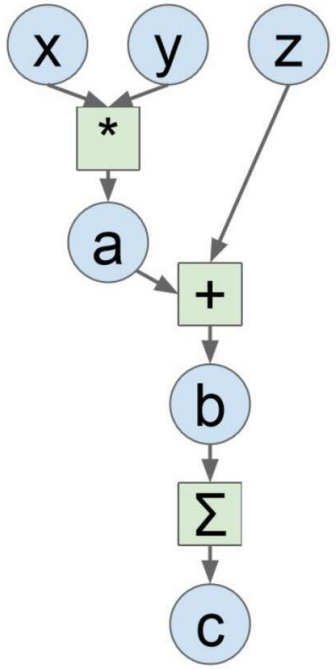
Ask TensorFlow to compute gradients

TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

TensorFlow

Computational Graphs



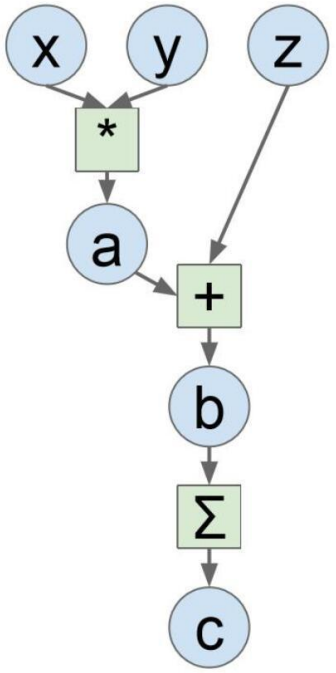
Tell TensorFlow to run on **CPU**

TensorFlow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3000, 4000

with tf.device('/cpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

TensorFlow

Computational Graphs



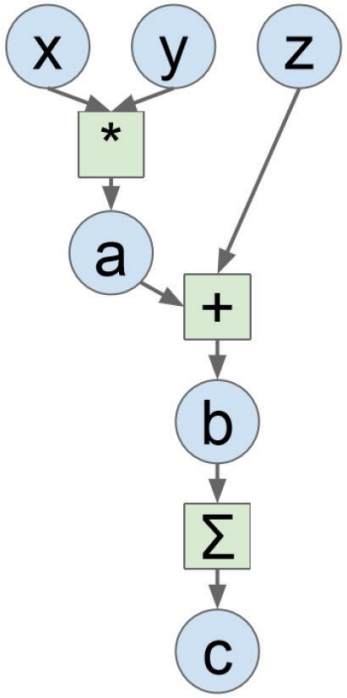
Tell TensorFlow to run on **GPU**

TensorFlow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3000, 4000

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

Computational Graphs



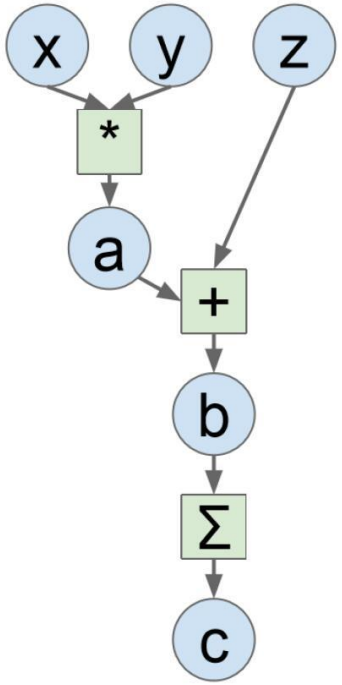
Define **Variables** to start building a computational graph

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

PyTorch

Computational Graphs



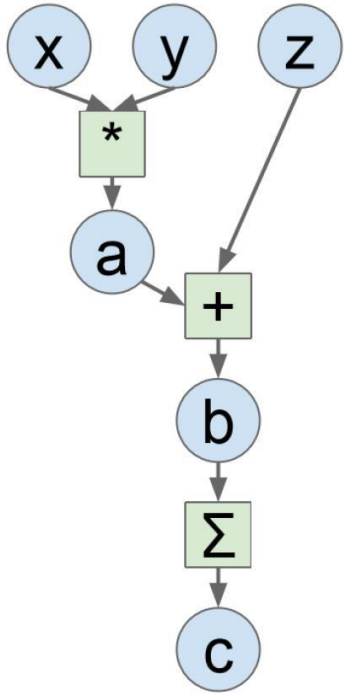
Forward pass looks just like numpy

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

PyTorch

Computational Graphs



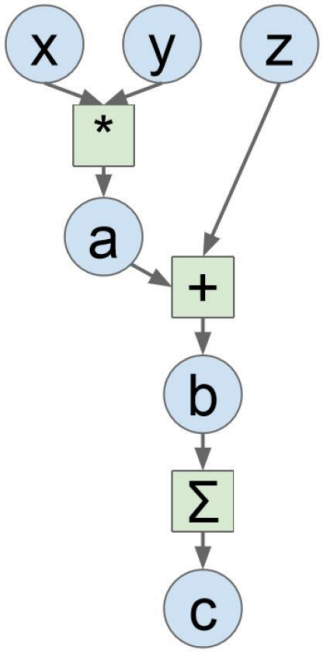
Calling c.backward() computes all gradients

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

PyTorch

Computational Graphs



Run on GPU by casting to .cuda()

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

PyTorch

A **DataLoader** wraps a **Dataset** and provides minibatching, shuffling, multithreading, for you

When you need to load custom data, just write your own Dataset class


```
import torch
from torch.autograd import Variable
from torch.utils.data import TensorDataset, DataLoader

N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

loader = DataLoader(TensorDataset(x, y), batch_size=8)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for epoch in range(10):
    for x_batch, y_batch in loader:
        x_var, y_var = Variable(x), Variable(y)
        y_pred = model(x_var)
        loss = criterion(y_pred, y_var)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

PyTorch

Iterate over loader to form minibatches

Loader gives Tensors so you need to wrap in Variables

```
import torch
from torch.autograd import Variable
from torch.utils.data import TensorDataset, DataLoader

N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

loader = DataLoader(TensorDataset(x, y), batch_size=8)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for epoch in range(10):
    for x_batch, y_batch in loader:
        x_var, y_var = Variable(x), Variable(y)
        y_pred = model(x_var)
        loss = criterion(y_pred, y_var)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

PyTorch

Super easy to use pretrained models with torchvision
https://github.com/pytorch/vision

```
import torch
import torchvision

alexnet = torchvision.models.alexnet(pretrained=True)
vgg16 = torchvision.models.vgg16(pretrained=True)
resnet101 = torchvision.models.resnet101(pretrained=True)
```

Comparison

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

TensorFlow

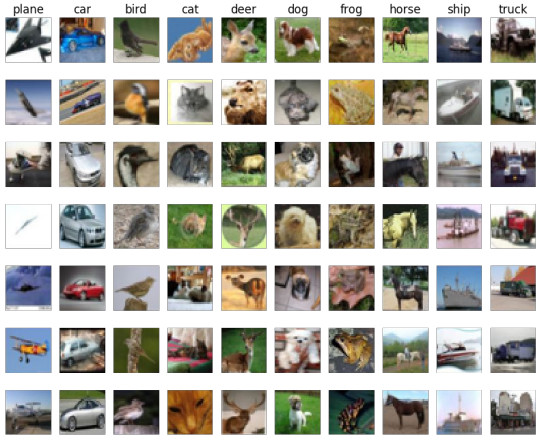
```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

Pytorch: Image Classification

- CIFAR-10: https://www.cs.toronto.edu/~kriz/cifar.html



- Data download and import libararies

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='/home/CIFAR-10 Classifier Using CNN in PyTorch/da
                                        train=True,
                                        download=True,
                                        transform=transform)
trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=4,
                                          shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data',
                                       train=False,
                                       download=True,
                                       transform=transform)
testloader = torch.utils.data.DataLoader(testset,
                                         batch_size=4,
                                         shuffle=False)

classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')
```

```
import torch
import torchvision
import torchvision.transforms as transforms
```

Pytorch: Image Classification

- Network architecture:

`Input > Conv (ReLU) > MaxPool > Conv (ReLU) > MaxPool > FC (ReLU) > FC (ReLU) > FC (SoftMax) > 10 outputs`

where:

`Conv` is a convolutional layer, `ReLU` is the activation function, `MaxPool` is a pooling layer, `FC` is a fully connected layer and `SoftMax` is the activation function of the output layer.

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).        ()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()
```

Pytorch: Image Classification

- Loss function and optimizer:

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

- Train the network:

```
import os

model_directory_path = '/home/CIFAR-10 Classifier Using CNN in PyTorch/model/'
model_path = model_directory_path + 'cifar-10-cnn-model.pt'

if not os.path.exists(model_directory_path):
    os.makedirs(model_directory_path)

if os.path.isfile(model_path):
    # load trained model parameters from disk
    net.load_state_dict(torch.load(model_path))
    print('Loaded model parameters from disk.')
else:
    for epoch in range(2):  # loop over the dataset multiple times

        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # get the inputs
            inputs, labels = data

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            if i % 2000 == 1999:    # print every 2000 mini-batches
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 2000))
                running_loss = 0.0
    print('Finished Training.')
    torch.save(net.state_dict(), model_path)
    print('Saved model parameters to disk.')
```

Pytorch: Image Classification

- Predict and test:

```
total_correct = 0
total_images = 0
confusion_matrix = np.zeros([10,10], int)
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total_images += labels.size(0)
        total_correct += (predicted == labels).sum().item()
        for i, l in enumerate(labels):
            confusion_matrix[l.item(), predicted[i].item()] += 1

model_accuracy = total_correct / total_images * 100
print('Model accuracy on {0} test images: {1:.2f}%'.format(total_images, model_accuracy))
```

TF: Image Classification

- Import TensorFlow:

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

- Prepare data:

```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

TF: Image Classification

- Add Conv layer:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

- Add dense layer:

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

TF: Image Classification

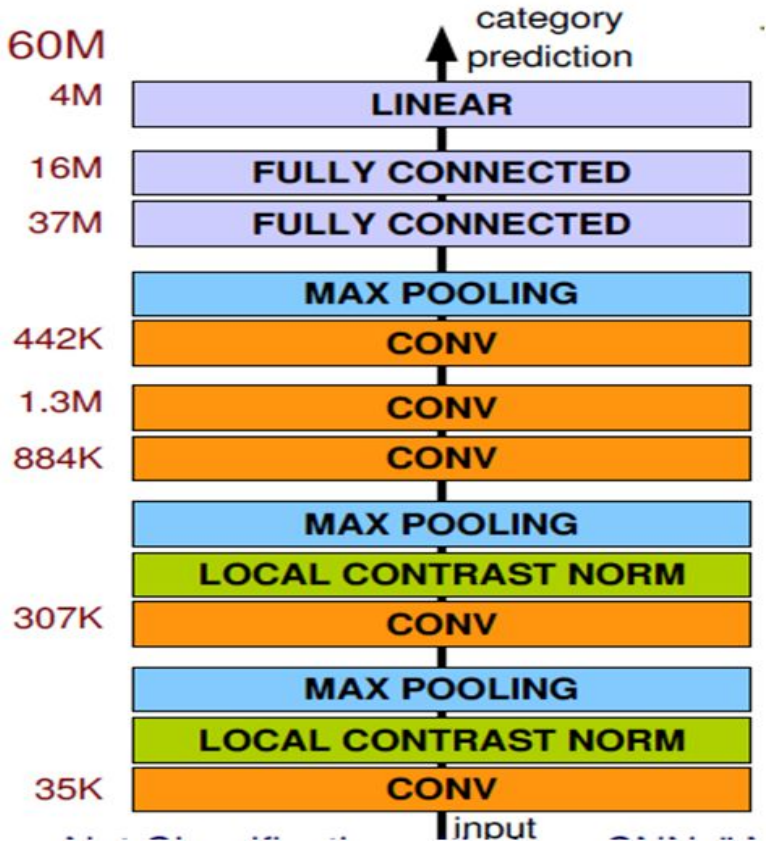
- Train:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

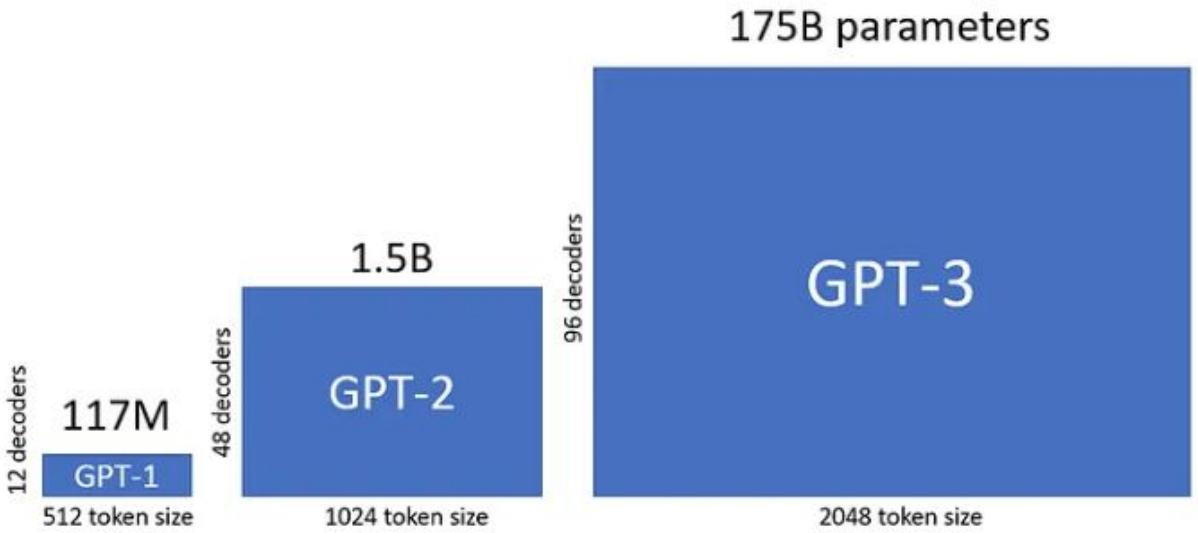
- Eval:

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

Pytorch or TF for LLMs?



VS.



Pytorch or TF for LLMs?

VS.

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Challenging Issues

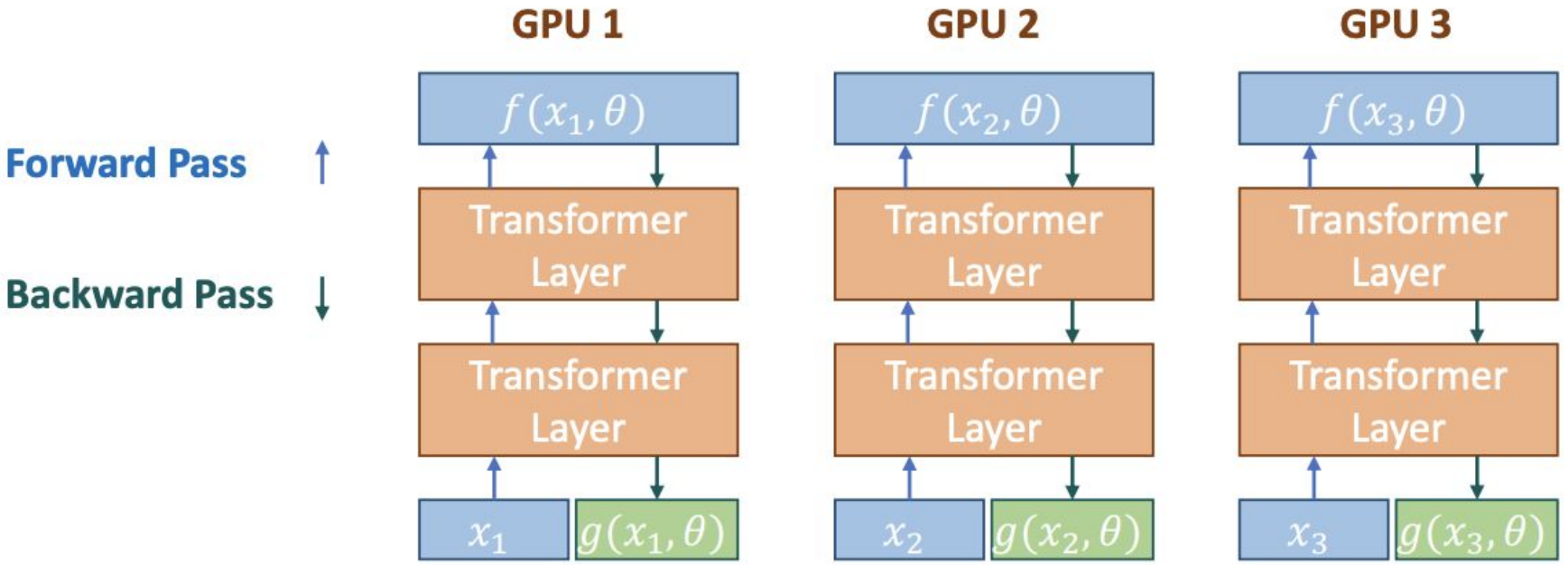
- As scale grows, training with one GPU is not enough.
  - Data parallelism

- When model training cannot fit into single-GPU.
  - Model parallelism: pipeline or tensor

- There are many other ways to improve efficiency on single-GPU training
  - Moving part of the operations to CPU memory
  - Quantizing different part of the optimization

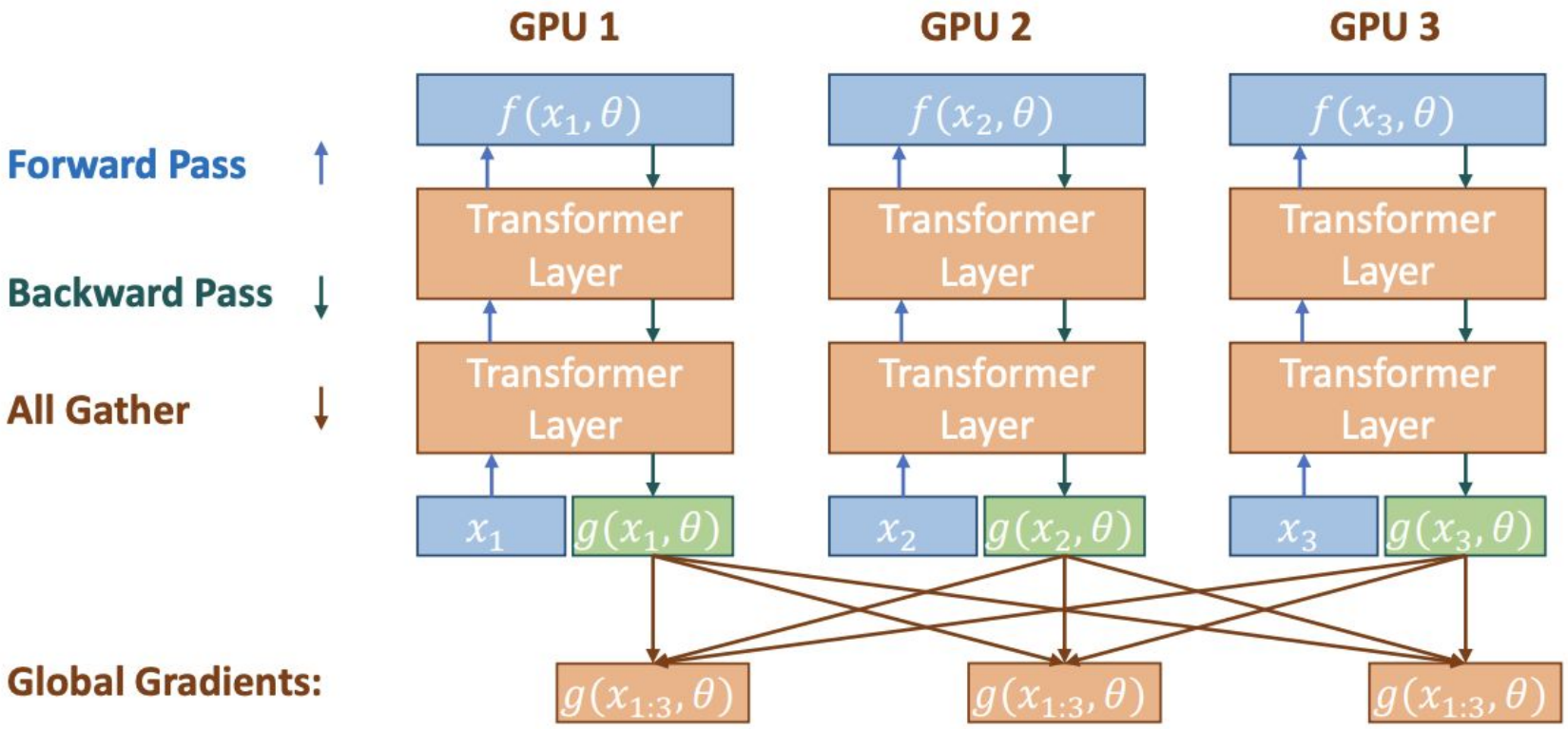
Data Parallelism (1)

- Split training data batch into different GPUs
  - Each GPU maintains its own copy of model and optimizer
  - Each GPU gets a different local data batch, calculates its gradients
  - Gather local gradients together to each GPU for global updates



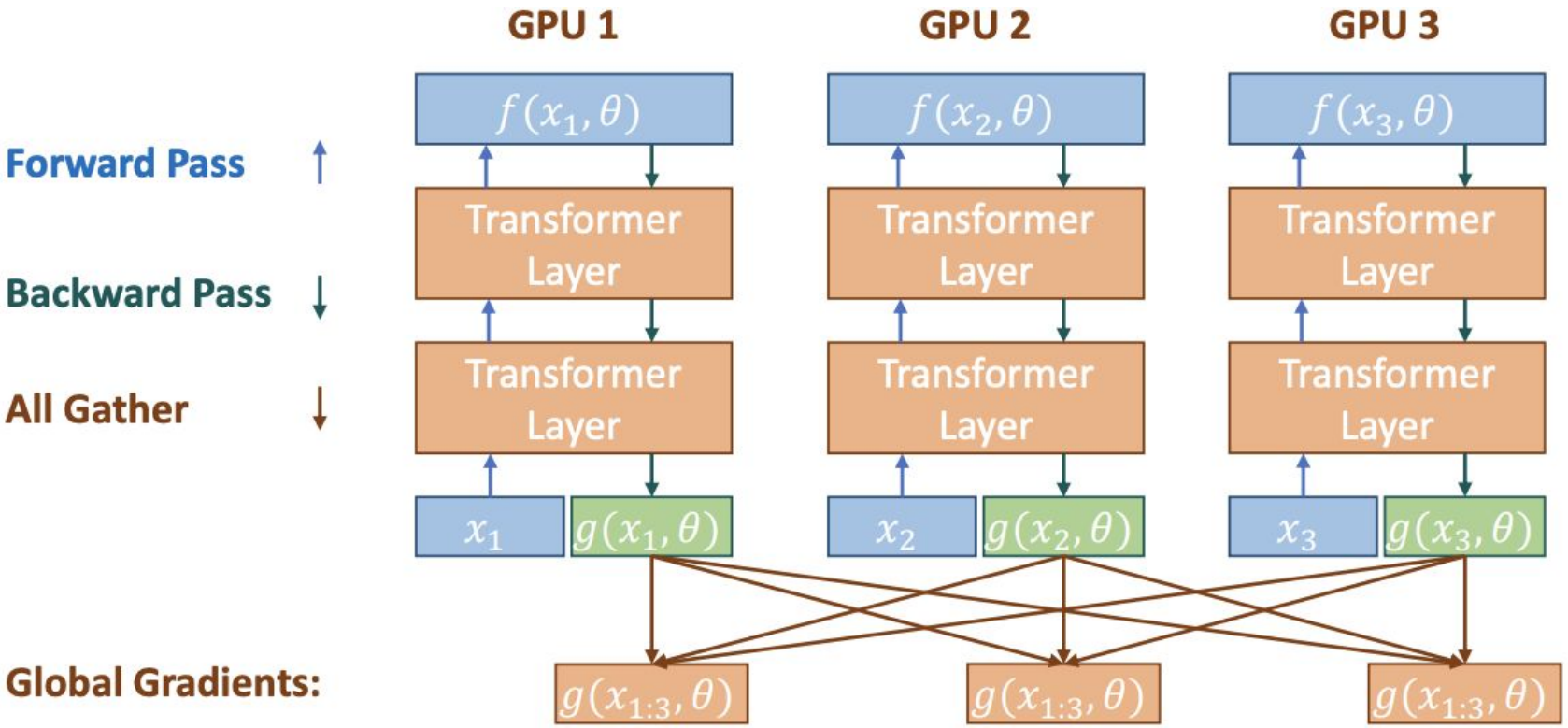
Data Parallelism (2)

- Split training data batch into different GPUs
  - Each GPU maintains its own copy of model and optimizer
  - Each GPU gets a different local data batch, calculates its gradients
  - Gather local gradients together to each GPU for global updates



Data Parallelism (3)

- Split training data batch into different GPUs
  - Each GPU maintains its own copy of model and optimizer
  - Each GPU gets a different local data batch, calculates its gradients
  - Gather local gradients together to each GPU for global updates



Communication:

- The full gradient tensor between every pair of GPUs, at each training batch.
- Not an issue between GPUs in the same machine or machines with infinity band
- Will need work around without fast cross-GPU connection

Model Parallelism (1)

- LLM size grew quickly and passed the limit of single GPU memory

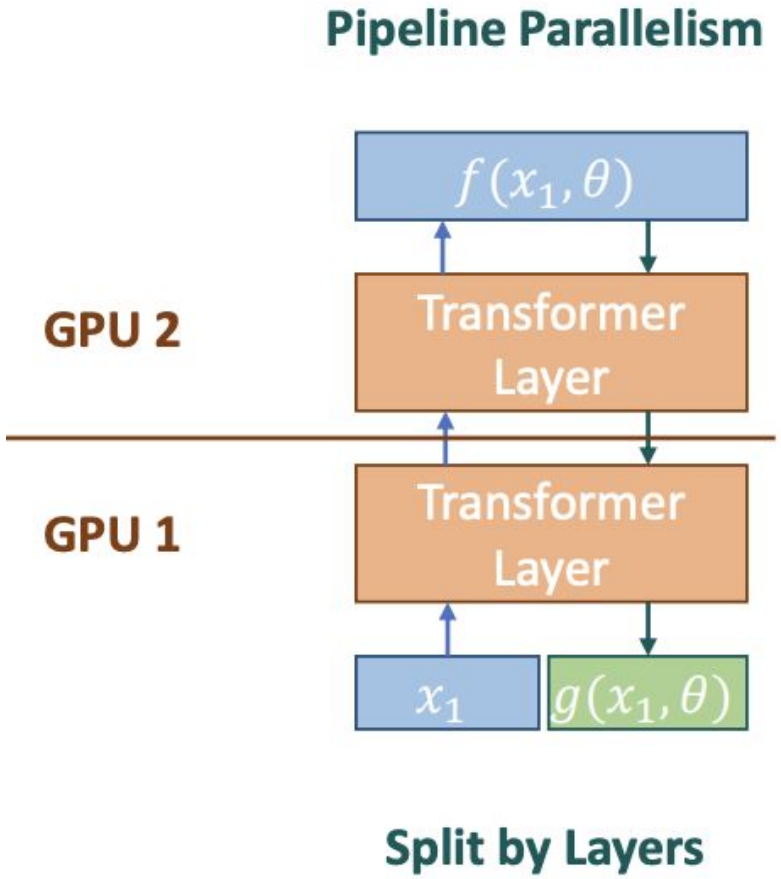
	Cost of 10B Model	Function to parameter count (Ψ)
Parameter Bytes	20GB	2Ψ
Gradient Bytes	20GB	2Ψ
Optimizer State: 1st Order Momentum	20GB	2Ψ
Optimizer State: 2nd Order Momentum	20GB	2Ψ
Total Per Model Instance	**80GB**	8Ψ

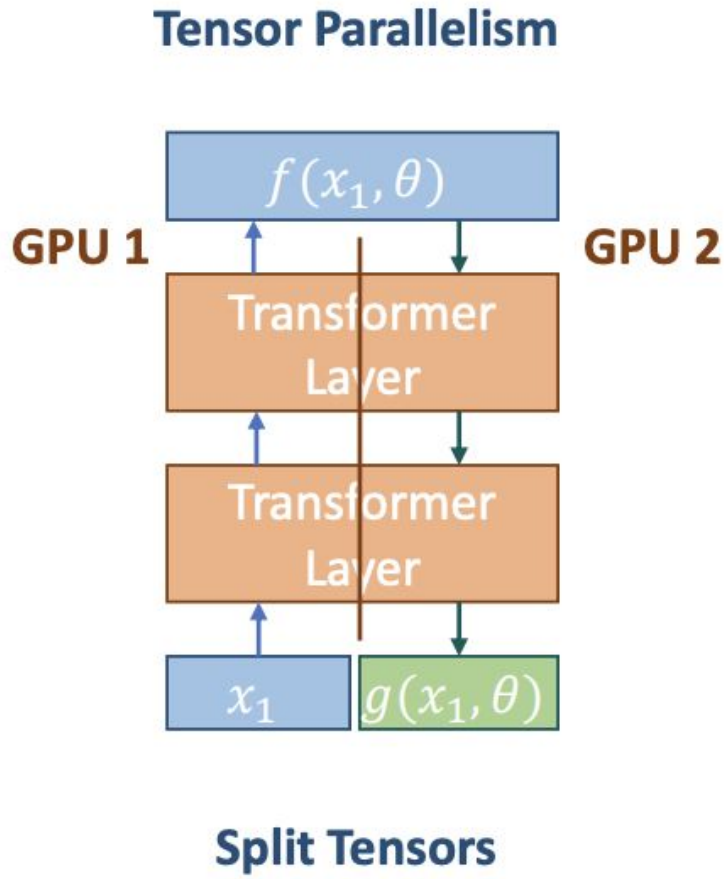
Table 1: Memory Consumption of Training Solely with BF16 (Ideal case) of a model sized Ψ

Solution: Split network parameters (thus their gradients and corresponding optimizer states) to different GPUs

Model Parallelism (2)

- Two ways of splitting network parameters





Pipeline Parallelism (1)

- Split network by layers, aligning devices by layer order to a pipeline, and pass data through devices

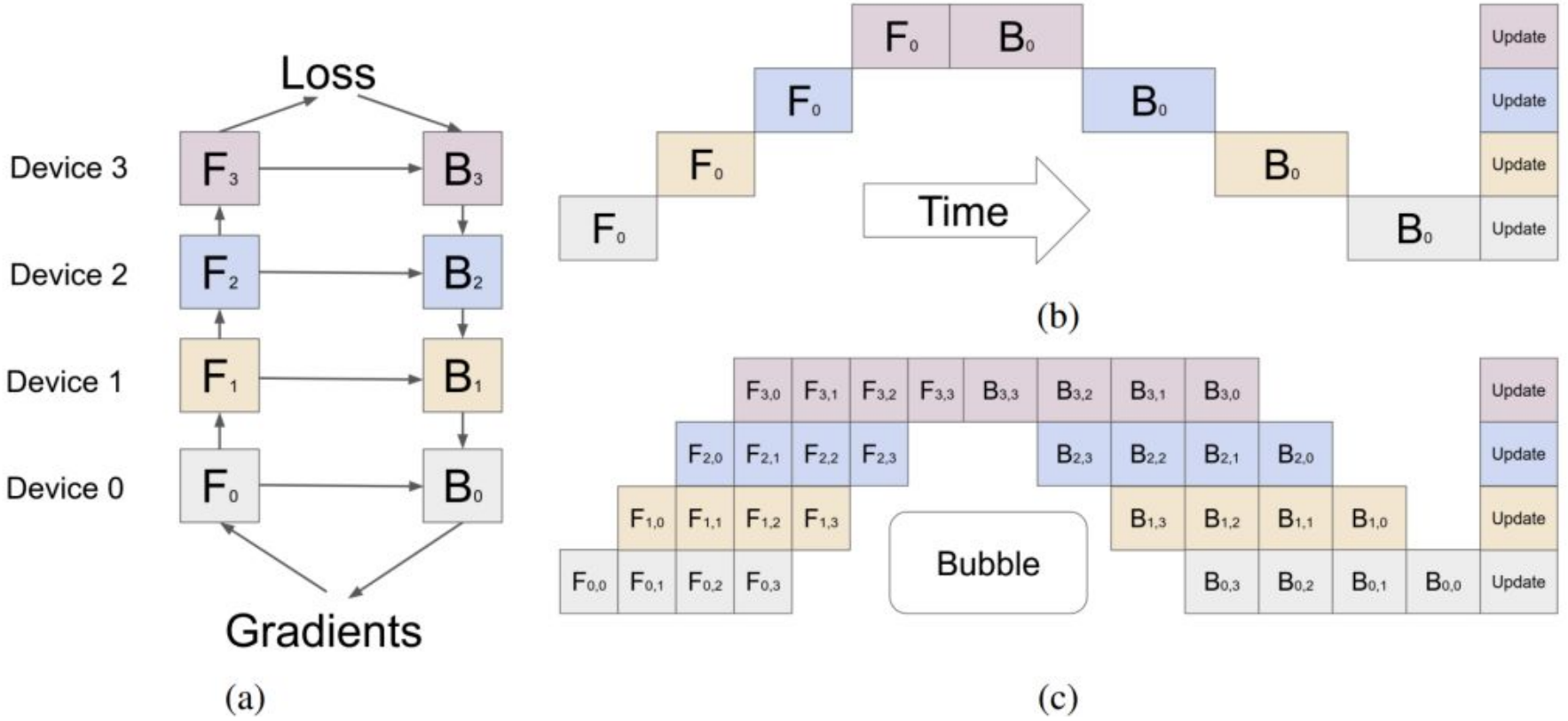
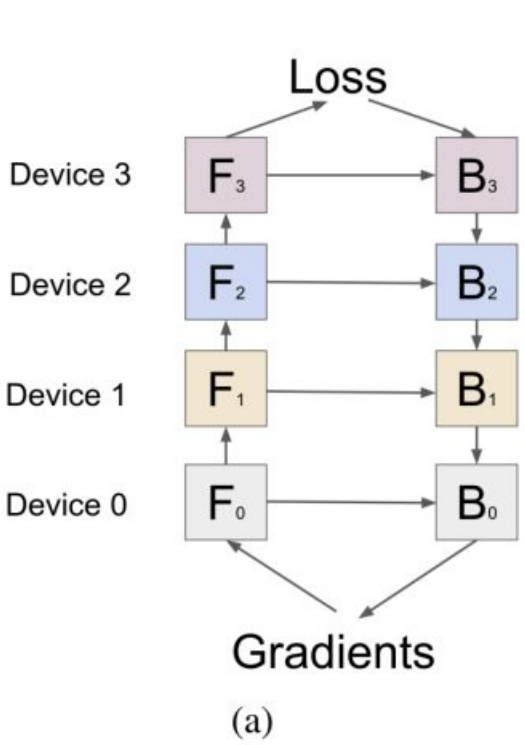
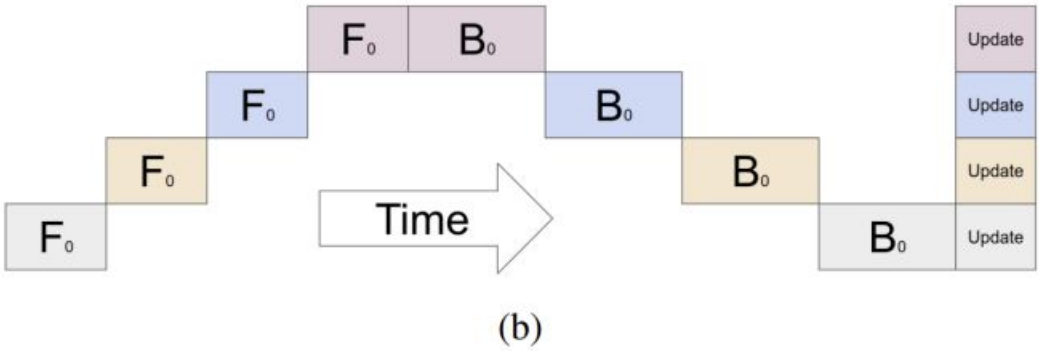


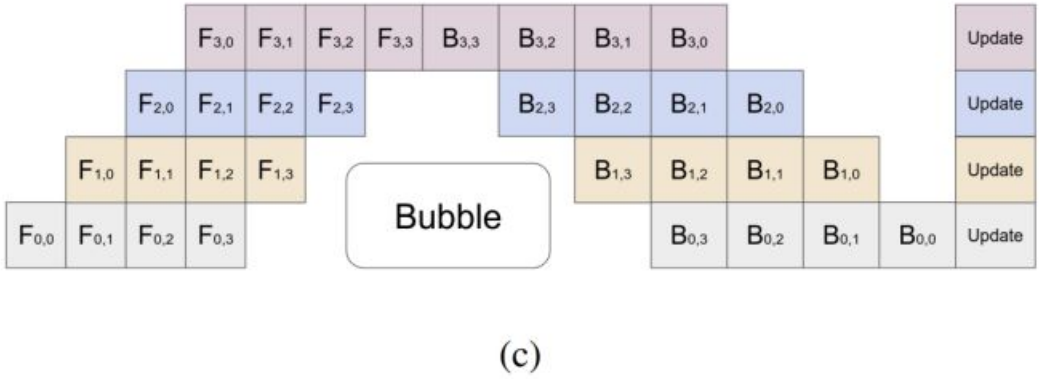
Illustration of Pipeline Parallelism

Pipeline Parallelism (2)

- Split network by layers, aligning devices by layer order to a pipeline, and pass data through devices



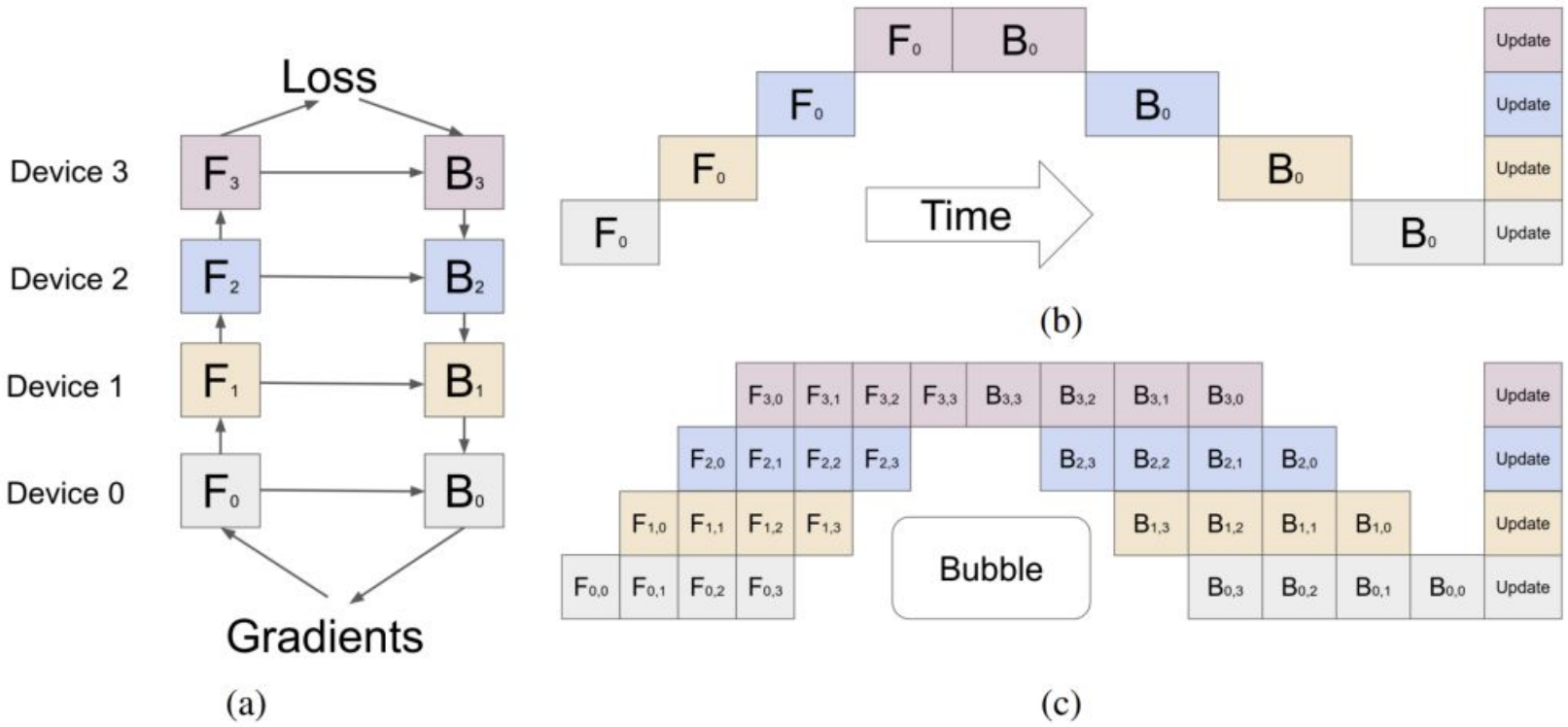




Communication:

- Activations between nearby devices in forward pass
- Partial gradients between nearby devices in backward
- Full gradients from Device 0 to All others

Pipeline Parallelism (3)



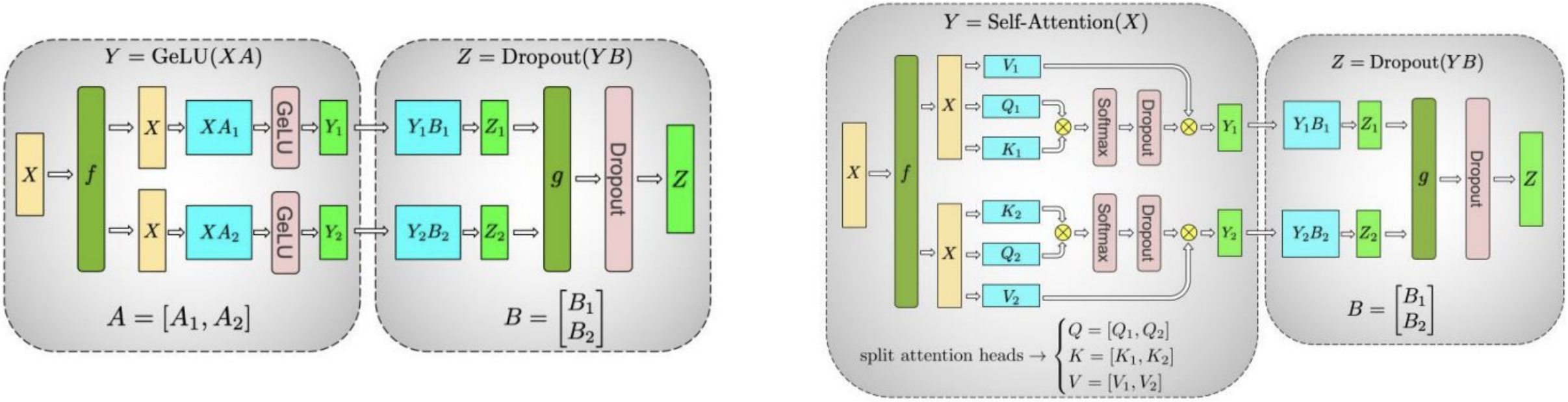
Communication:

- Activations between nearby devices in forward pass
- Partial gradients between nearby devices in backward
- Full gradients from Device 0 to All others

Pros: Conceptually simple and not coupled with network architectures. All networks have multiple layers.
Cons: Waste of compute in the Bubble. Bubble gets bigger with more devices and bigger batches.

Tensor Parallelism (1)

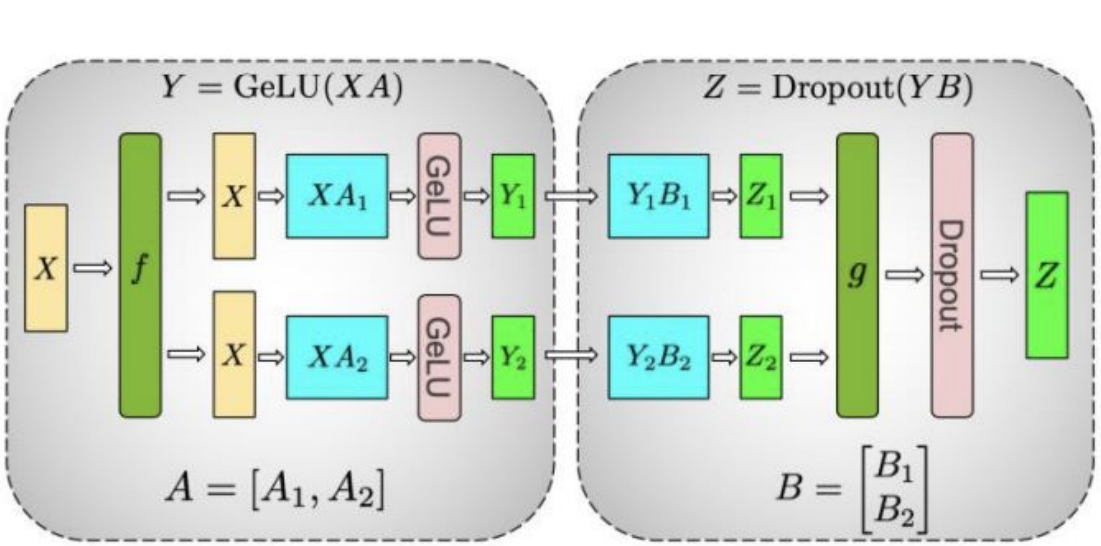
Split the parameter tensors of target layers into different devices

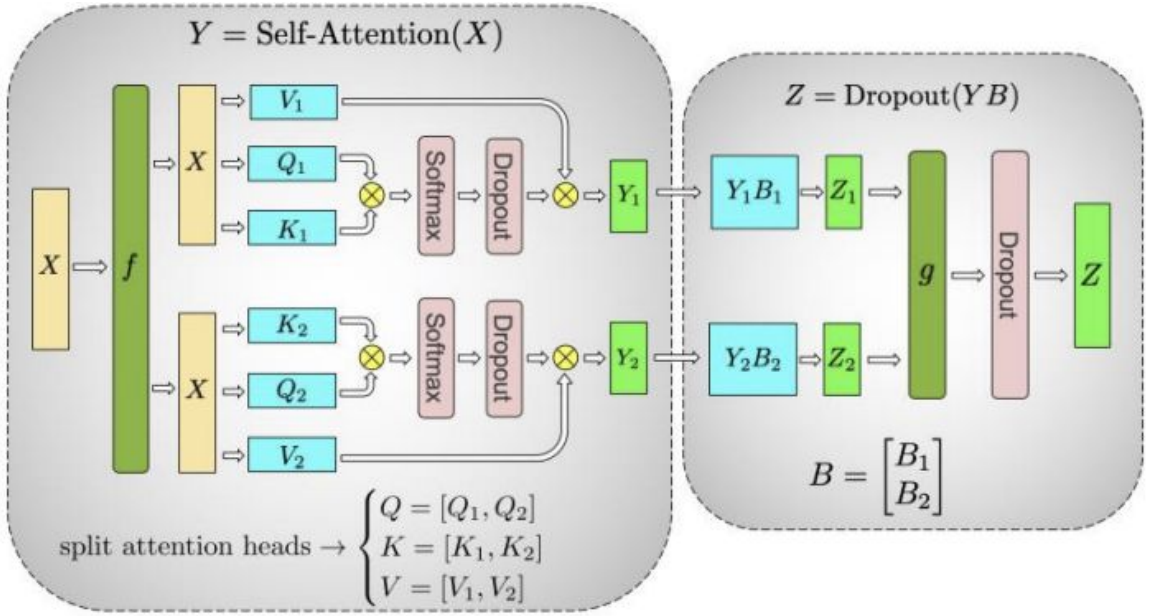


Tensor Parallelism of MLP blocks and Self-attention Blocks

Tensor Parallelism (2)

Split the parameter tensors of target layers into different devices





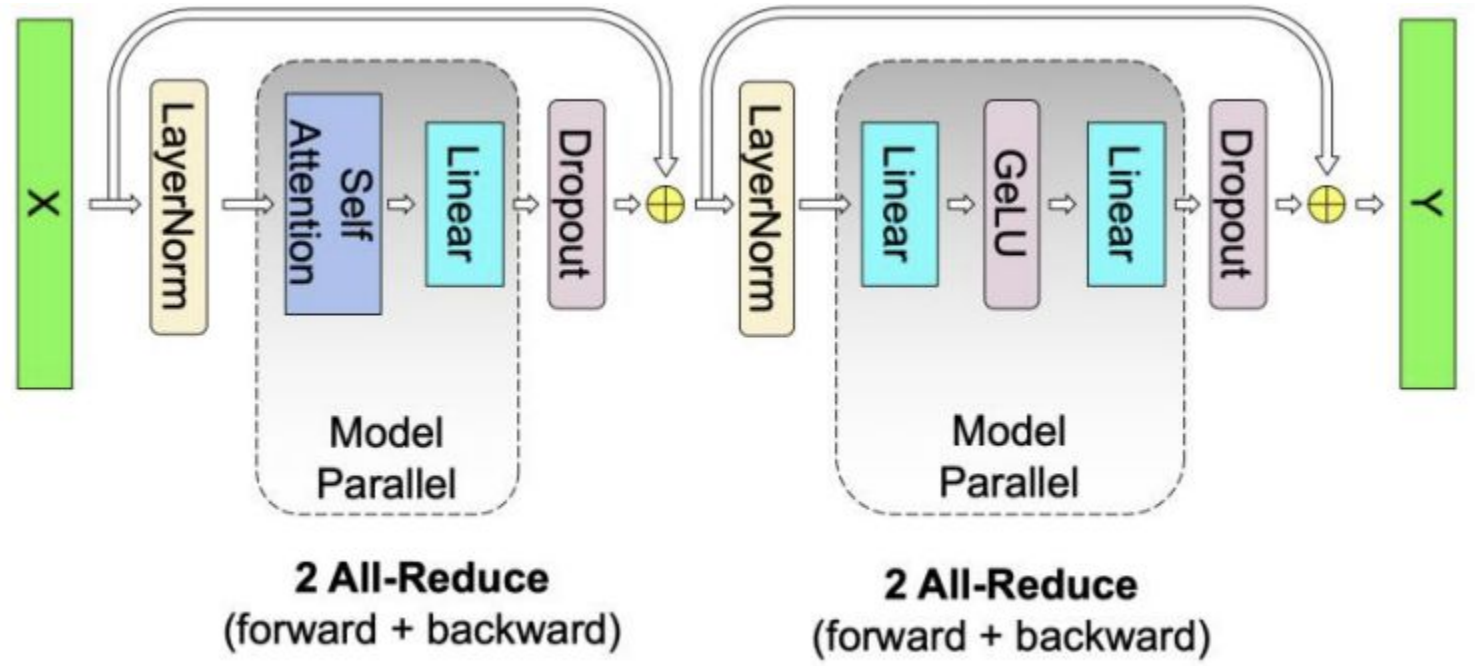
Pros: No bubble

Cons: Different blocks are better split differently, lots of customizations

Tensor Parallelism (3)

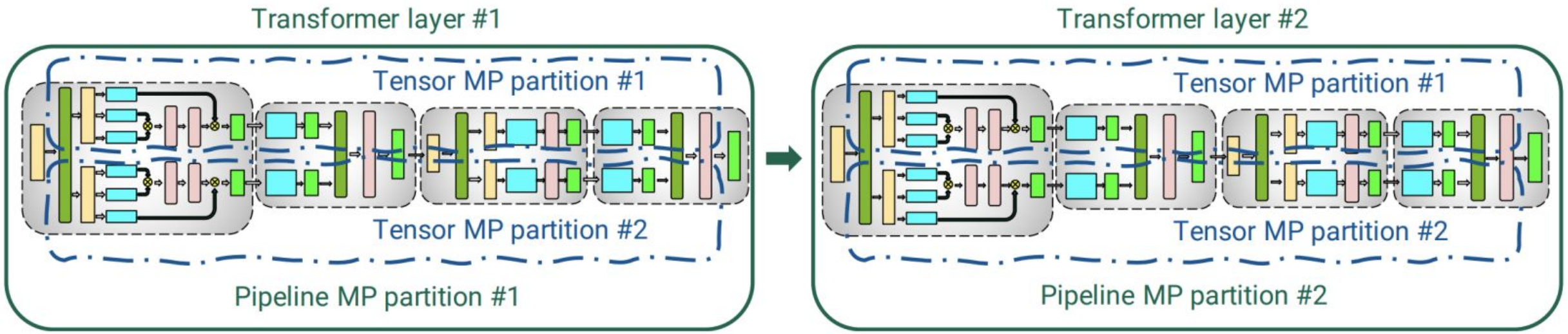
Communication:

• All-gather of partial activations and gradients for each split tensor



Combining Different Parallelism

- Often data parallelism and model parallelism are used together.
- Pipeline Parallelism and Tensor Parallelism can also be used together.

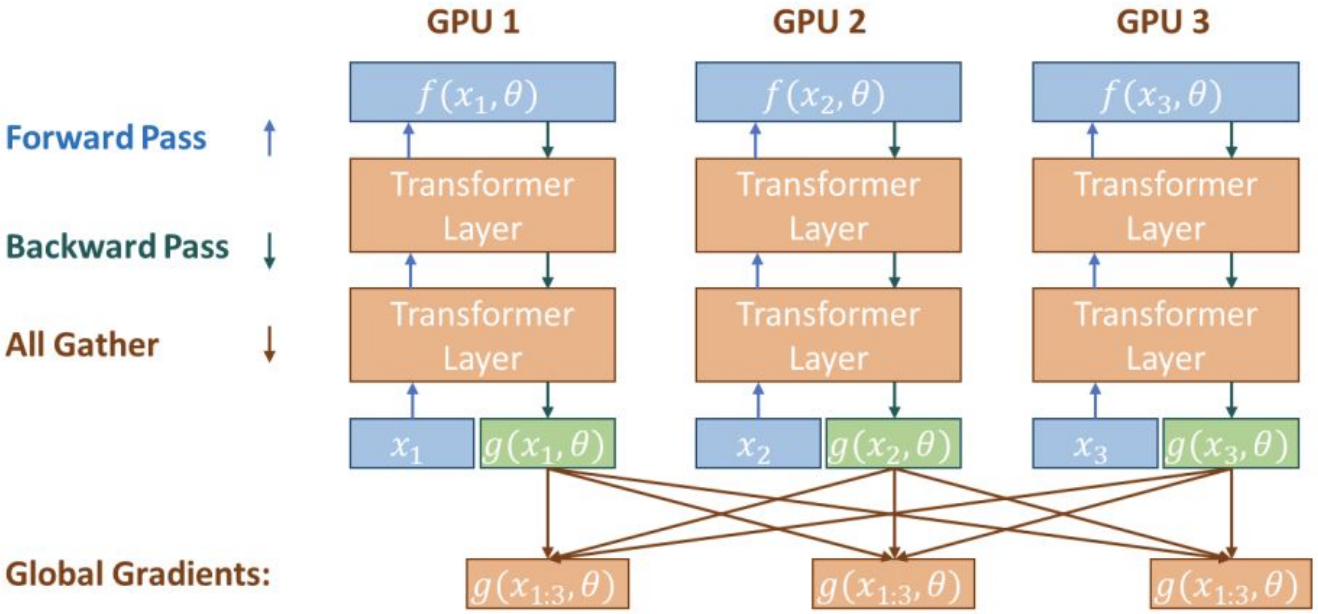


ZeRO: Redundancy in Data Parallelism (1)

- Majority of GPU memory consumption is on the optimization side: gradients and optimizer momentums

	Cost of 10B Model	Function to parameter count (Ψ)
Parameter Bytes	20GB	2Ψ
Gradient Bytes	20GB	2Ψ
Optimizer State: 1st Order Momentum	20GB	2Ψ
Optimizer State: 2nd Order Momentum	20GB	2Ψ
Total Per Model Instance	**80GB**	8Ψ

Table 1: Memory Consumption of Training Solely with BF16 (Ideal case) of a model sized Ψ

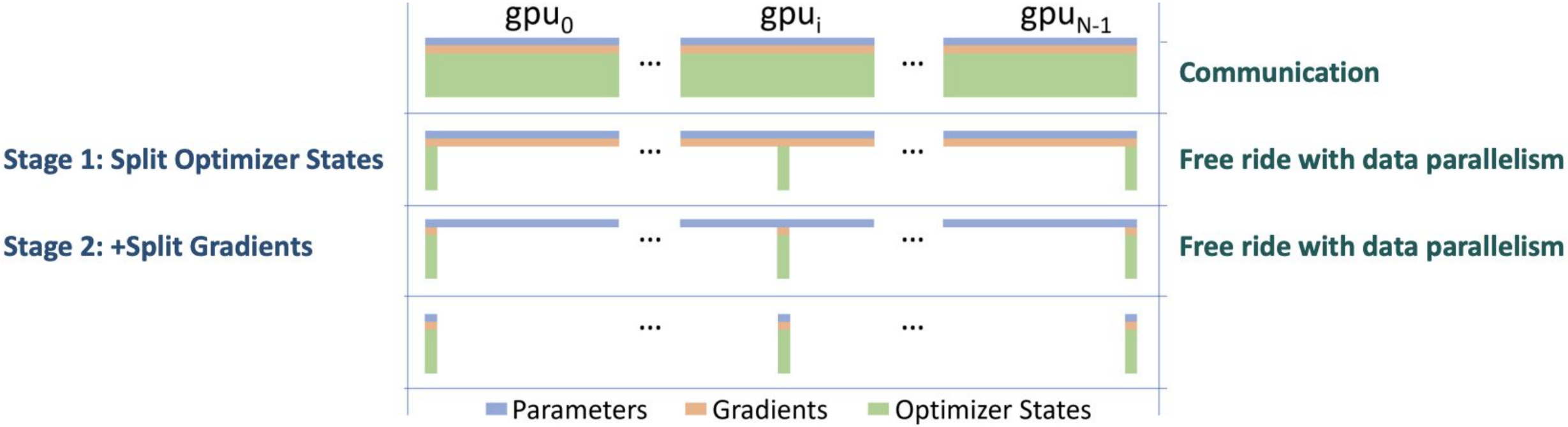


Observation:

- In data parallelism, each device only has access to local gradient
- All gather operation required on all gradients anyway

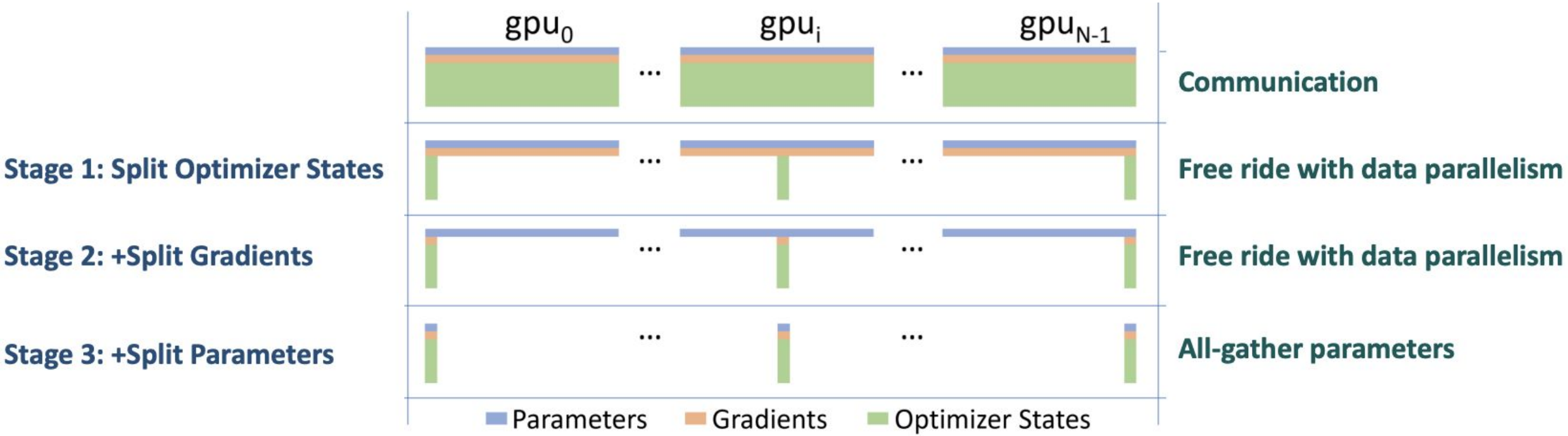
ZeRO: Redundancy in Data Parallelism (2)

- ZeRO Optimizer (Zero Redundancy Optimizer): Split GPU memory consumption into multiple GPUs during data parallelism



ZeRO: Redundancy in Data Parallelism (3)

- ZeRO Optimizer: Split GPU memory consumption into multiple GPUs during data parallelism



ZeRO: Redundancy in Data Parallelism (4)

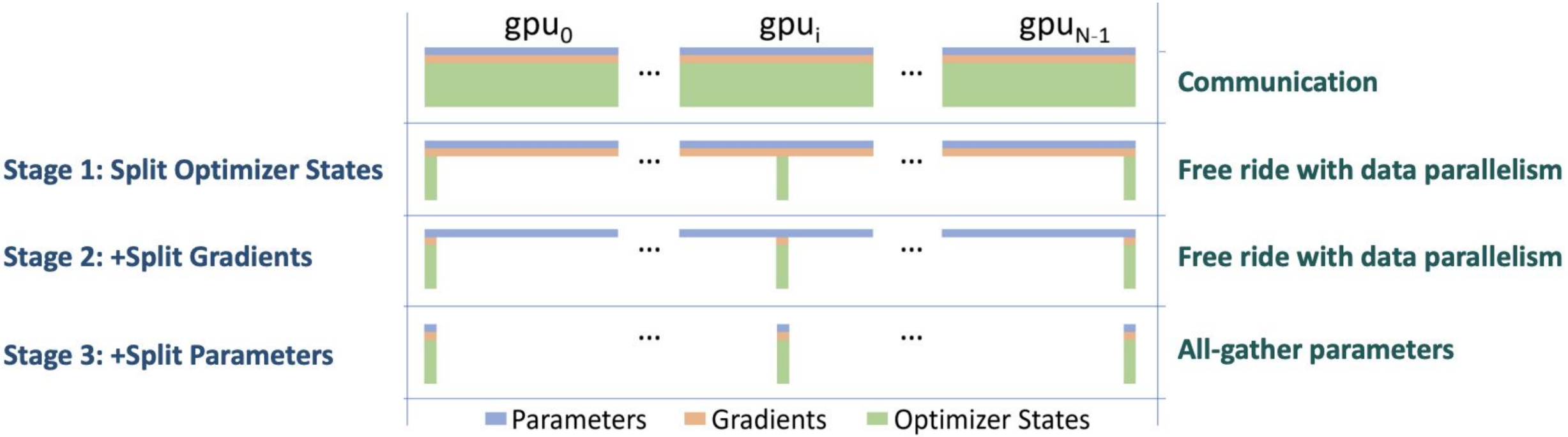


Figure 11: ZeRO Optimizer Stages [10]

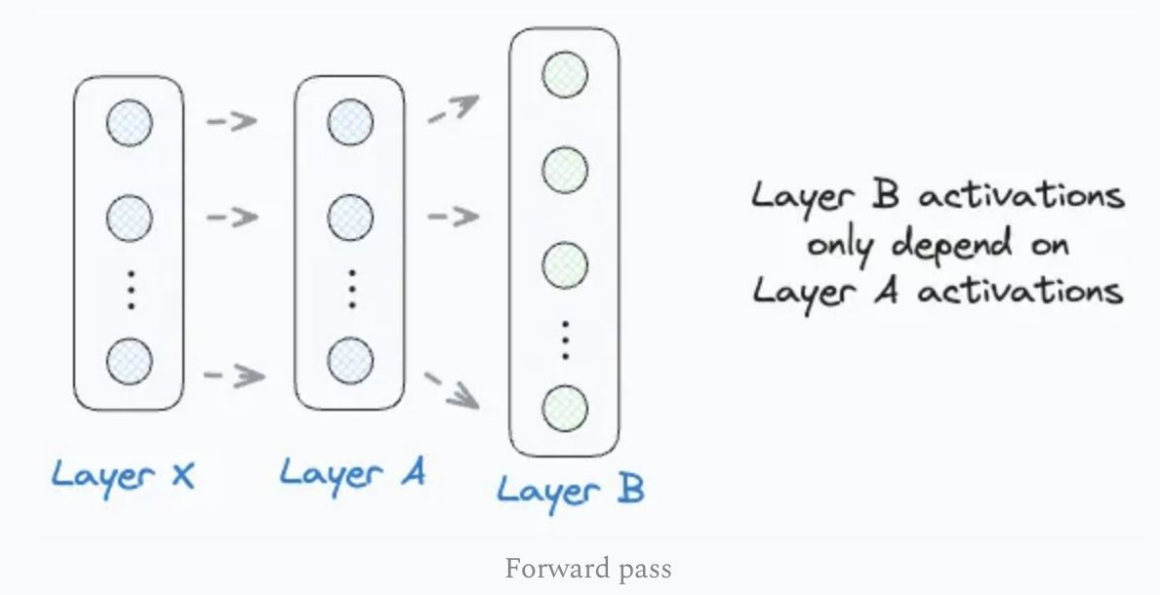
Pros: Stage 1 and 2 free ride with data parallelism with huge GPU memory savings

Notes: Stage 3 is a variant of tensor parallelism, but passing parameters instead of activations and gradients

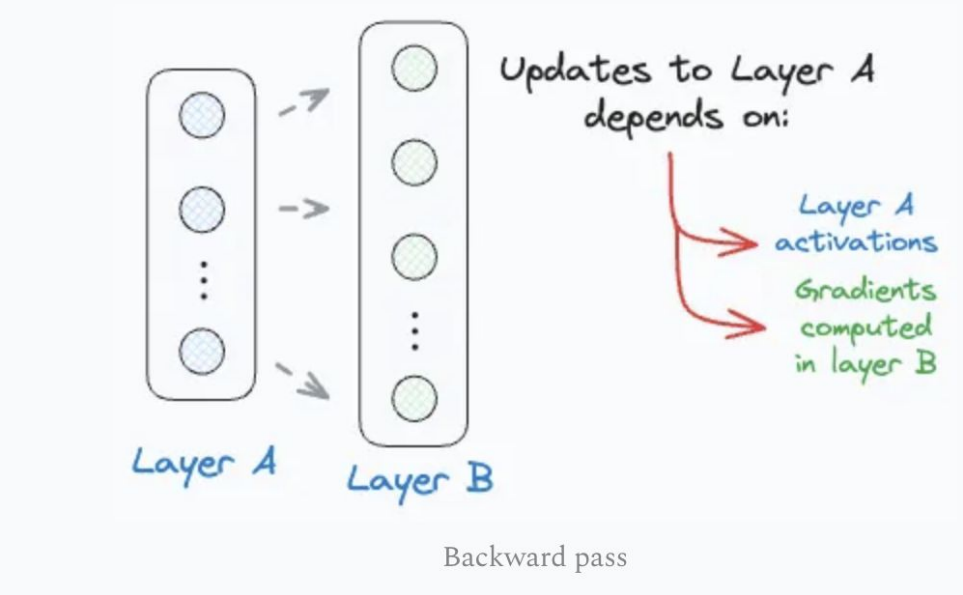
Cons: Open-source support not as good

Gradient Checkpointing (1)

- Storing model weights
- During training:
  - Forward pass to compute and store activations of all layers
  - Backward pass to compute gradients at each layer

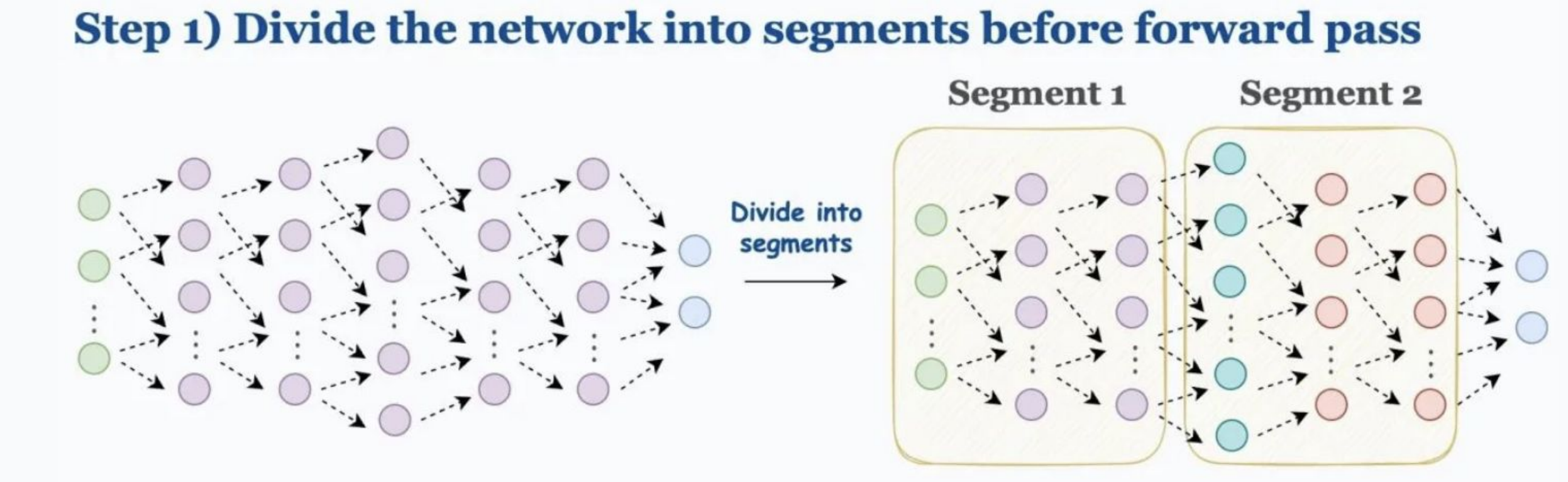


Forward pass

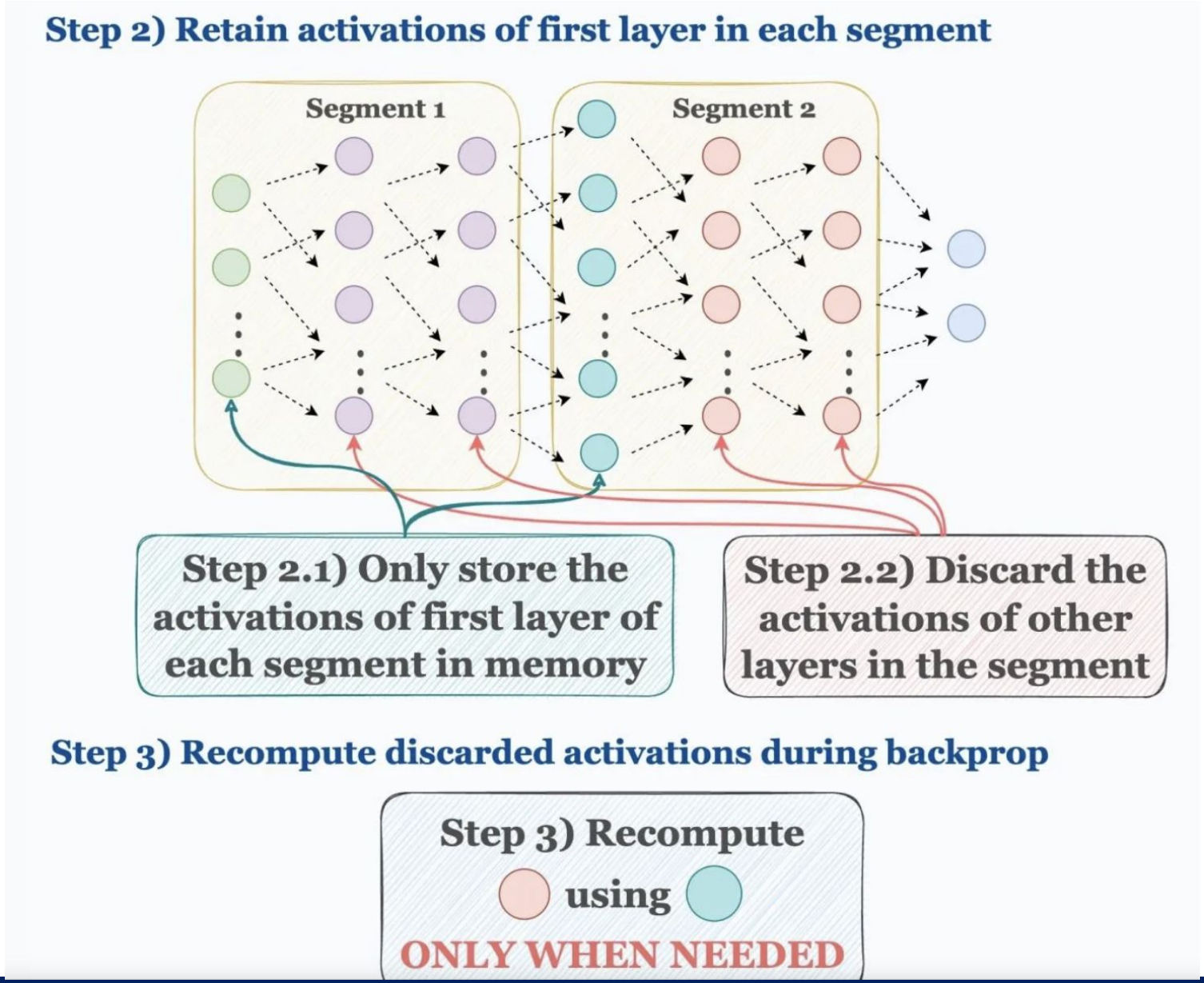


Backward pass

Gradient Checkpointing (2)

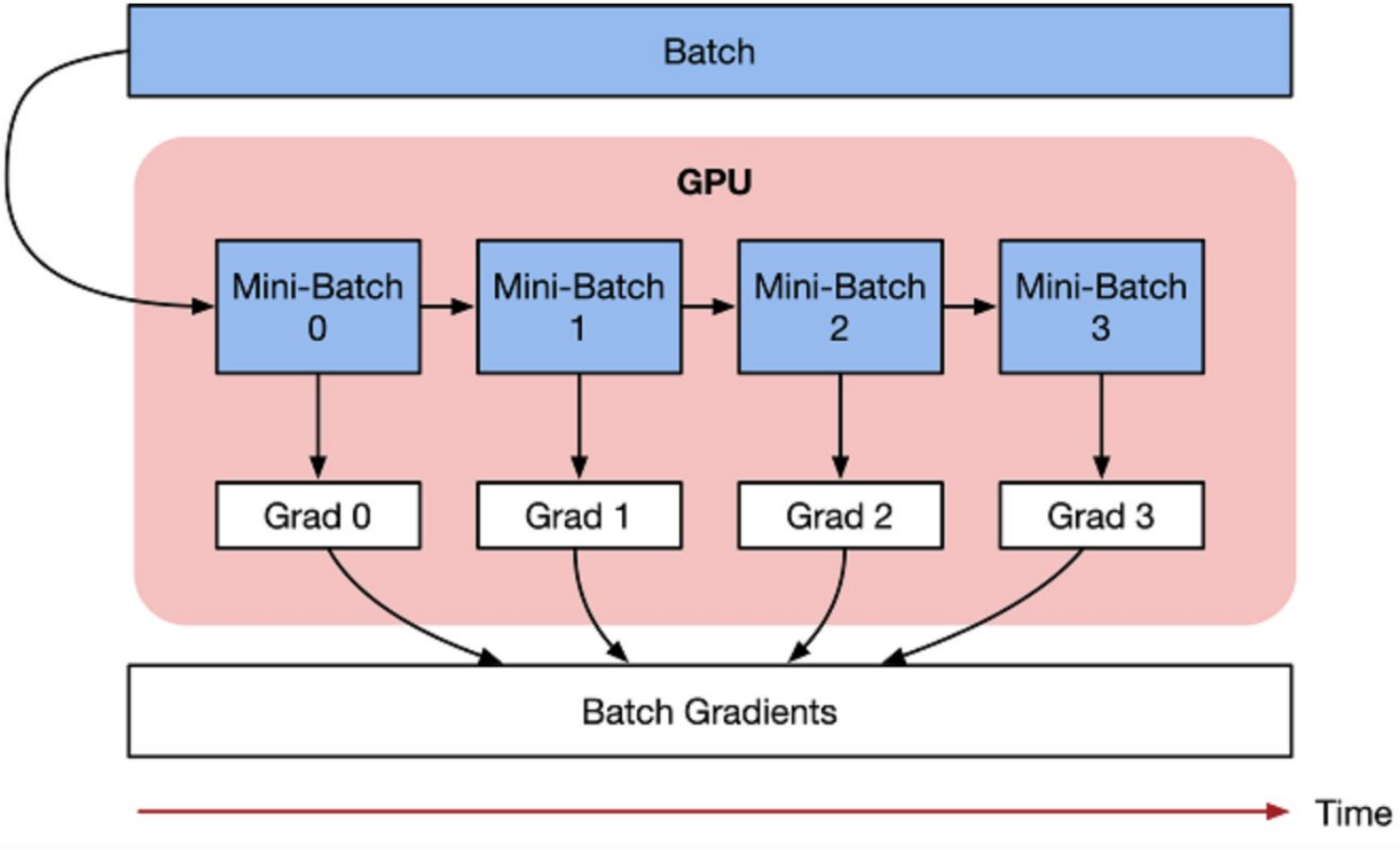


Gradient Checkpointing (3)



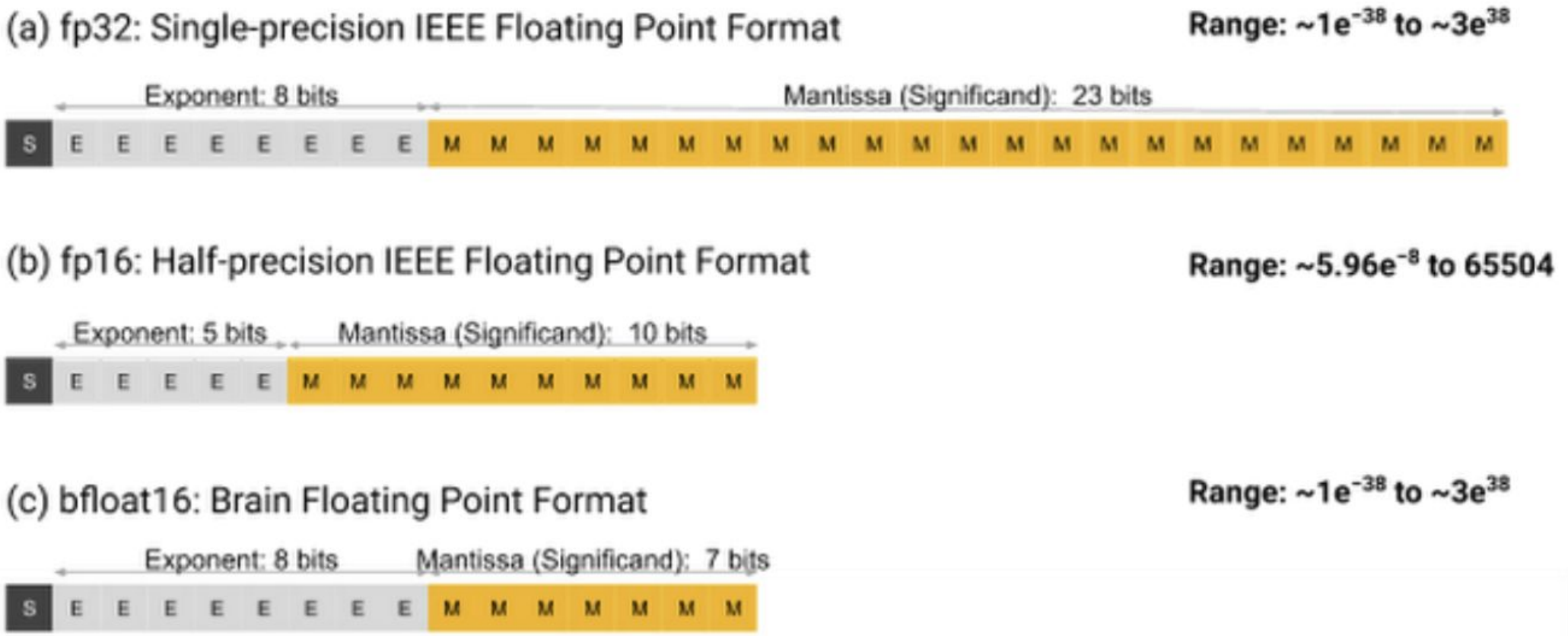
Gradient Accumulation

- Eventually accumulating the gradient in the previous step yields the sum of the gradients of the same size as using the global Batch size.



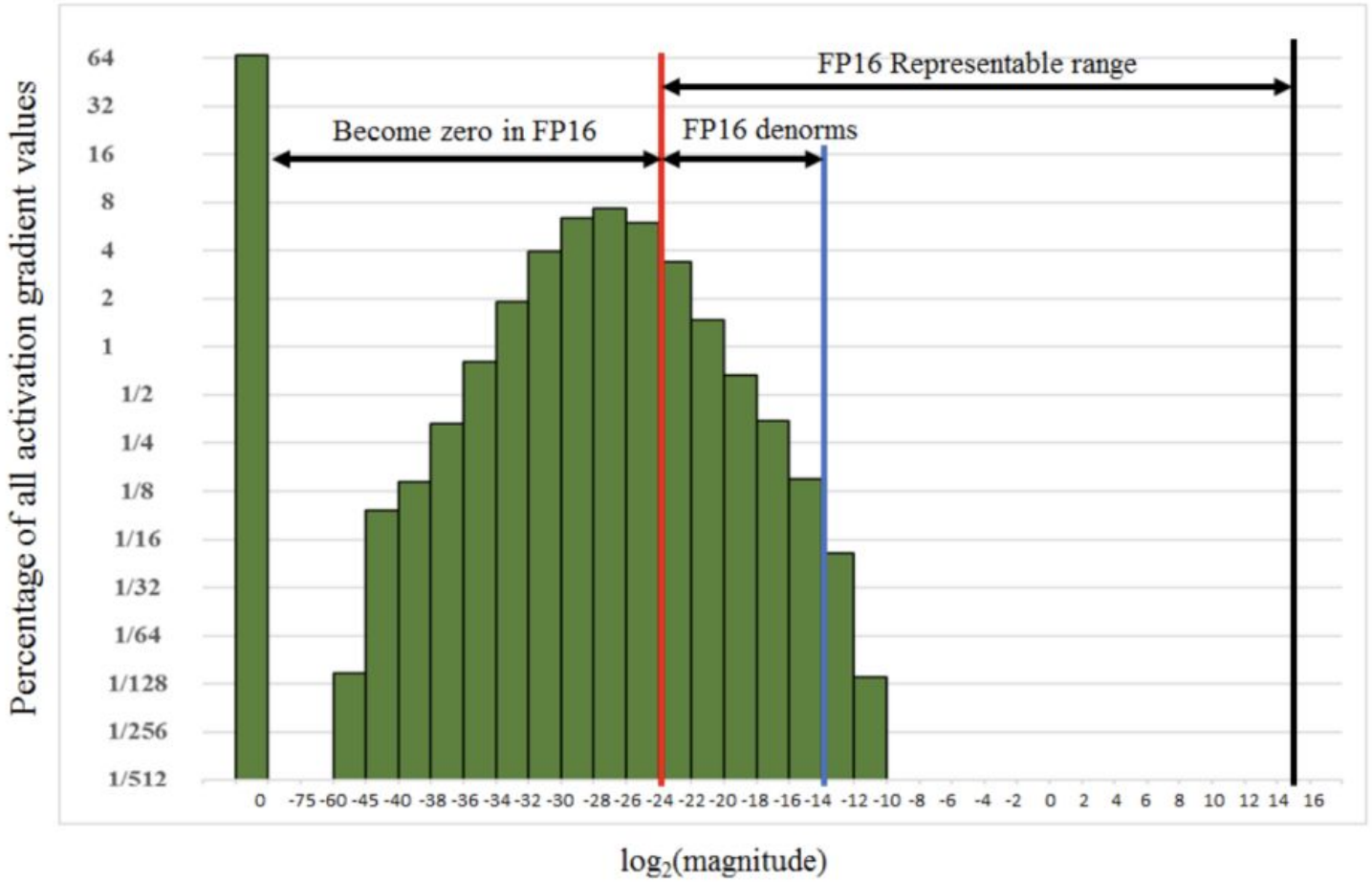
Mixed Precision Training (1)

- fp32, fp16 and bf16 are all supported in GPUs and TPUs.



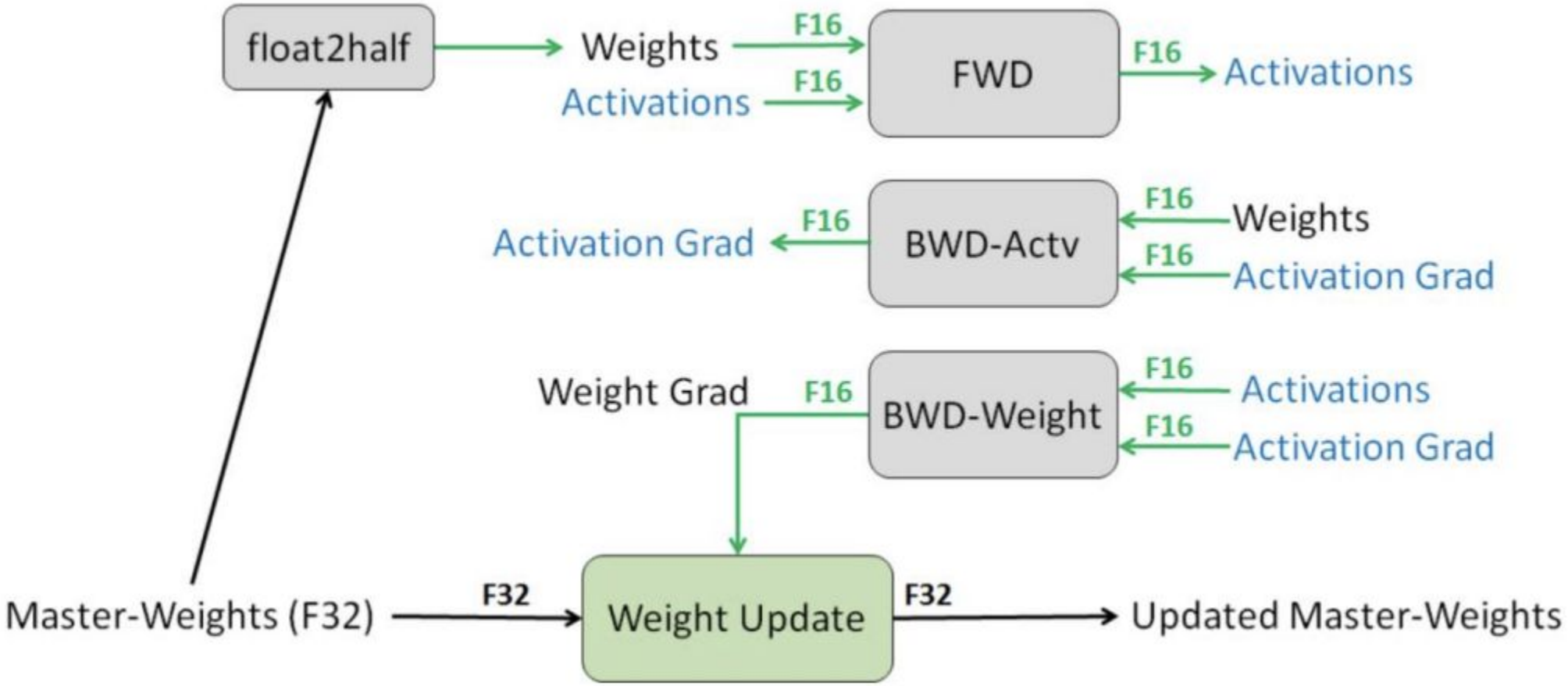
Mixed Precision Training (2)

- Many computation needs bigger range than FP16
  - Parameters, activations, and gradients often use FP16
  - Optimizer states often needs FP32



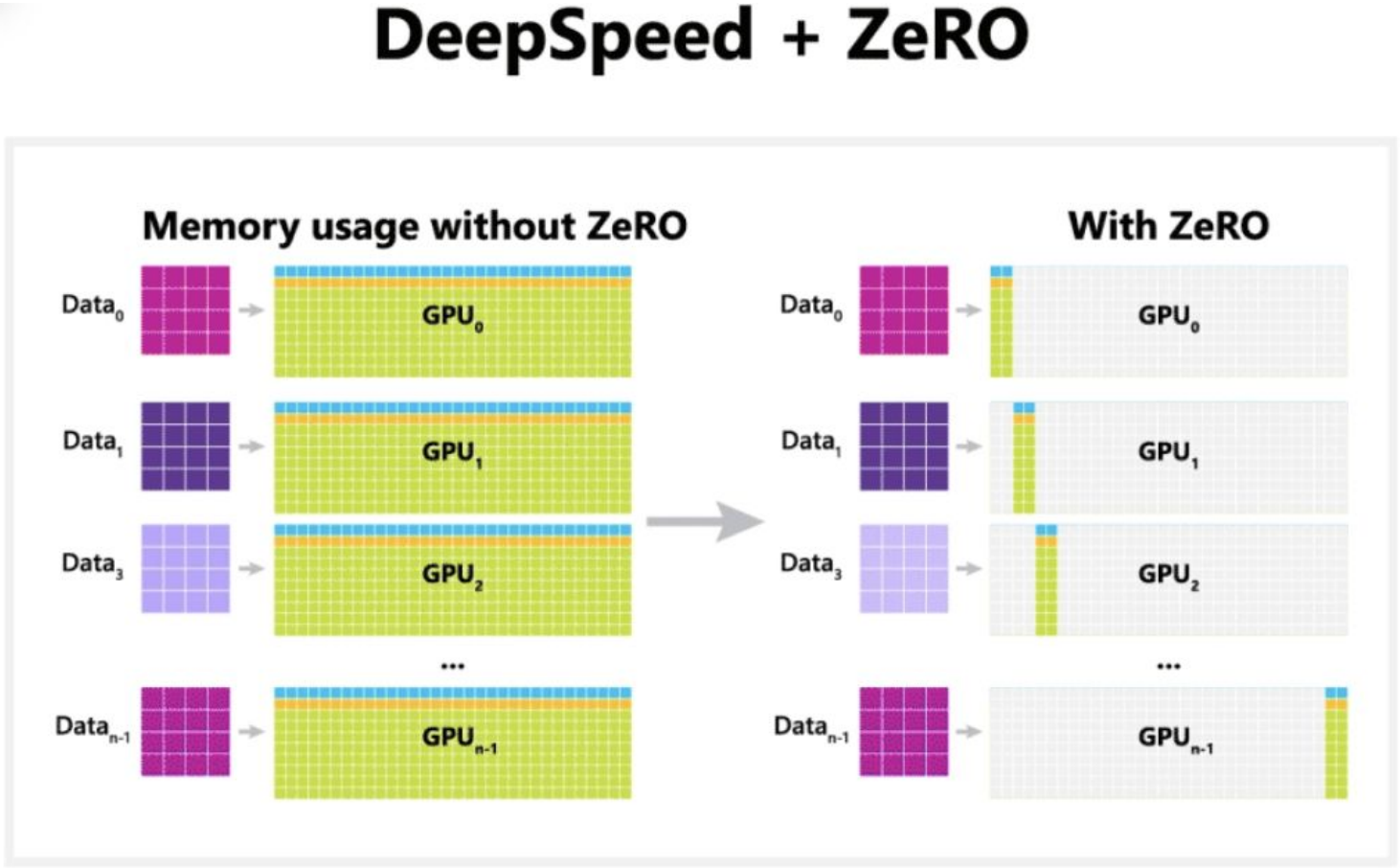
Mixed Precision Training (3)

- Maintaining main copies of FP32 for calculations, Dynamically scaling up loss to fit gradients etc. in FP16 range



Pre-training Platform: DeepSpeed

- Proposed by Microsoft in 2020: ZeRO, DP/MP
- Cons: does not support PP/TP training well



DeepSpeed

Pre-training Platform: Megatron-LM

- Proposed by Nvidia 2019: ZeRO, DP/MP/PP/TP, many good models
- Cons: not friendly for new users

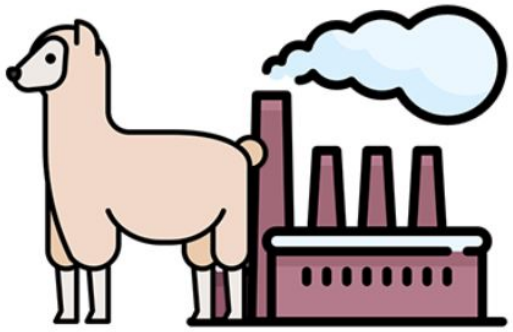
Megatron-LM & Megatron-Core

GPU optimized techniques for training transformer models at-scale

docs latest | release 0.5.0 | license OpenBSD

Pre-training Platform: others

- Not specific for pretraining, most of them support continue pretraining
- vLLM, LLaMA Factory



LLaMA-Factory

Easy and Efficient LLM Fine-Tuning

vLLM

Easy, fast, and cheap LLM serving for everyone

Thanks

Q&A

Prof. Yu Cheng
chengyu@cse.cuhk.edu.hk