DSCI 565: COMPUTATIONAL PERFORMANCE

Ke-Thia Yao

Lecture 18: 2025 November 3

Deep Learning Framework Implementation

Two main approaches to implement a deep learning framework:

Imperative Programming

- Program consists of statements
- Each statement changes the state of the program
- Statement: `a = a + 1`
  - Changes the state of a by incrementing a by 1

Symbolic Programming

- Program defines the computation to be performed
- This symbolic specification is then compiled and executed
- Statement: `a = a + 1`
  - Specifies variable a is incremented by 1

Imperative Programming Example

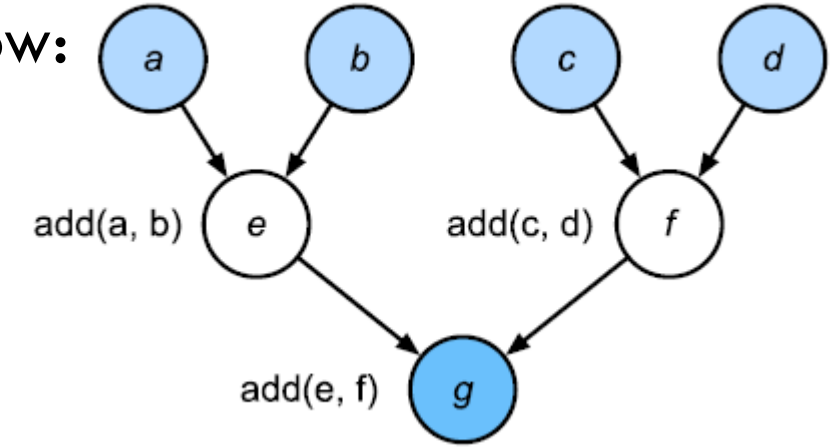
Program:

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))
```

Data Flow:



- Call fancy_func to add 4 numbers
- Call the add function 3 times
- More difficult to optimize, should suppress the generation intermediate variables e and f?

Symbolic Programming Example

- Specifies the fancy_func function
- Compiles the specification
- Opportunity to optimizes function during compilation, such as transforming its abstract semantic graph
- Executes the compiled specification

```
def add_():
    return '''
def add(a, b):
    return a + b
'''

def fancy_func_():
    return '''
def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
'''

def evoke_():
    return add_() + fancy_func_() \
    + 'print(fancy_func(1, 2, 3, 4))'

prog = evoke_()
print(prog)
y = compile(prog, '', 'exec')
exec(y)
```

Differences

Imperative Programming

PyTorch

- Majority of the code is easier to write
- Easier to debug, just use Python debugging environment on the actual written code
- Easier to implement new layers, just write Python code
- Arbitrary control flow as Python code in the forward() function

Symbolic Programming

TensorFlow

- More efficient and easier to port
- Optimize code during compilation
- Compile to non-Python backend
- Easier to inspect model, i.e., model summary, weights, shape
- Provides type checking and debugging in model definition
- Easier to save and store model

Hybrid Programming

- PyTorch implements the imperative approach,
  but it has added TorchScript Just-In-Time compiler
- TensorFlow implements the symbolic approach,
  but it has added Imperative API for version 2

TensorFlow Hybrid Programming

Imperative Programming

```
class CNN_Encoder(tf.keras.Model):
  def __init__(self, embedding_dim):
    super(CNN_Encoder, self).__init__()
    self.fc = tf.keras.layers.Dense(embedding_dim)

  def call(self, x):
    x = self.fc(x)
    x = tf.nn.relu(x)
    return x
```

Your symbolically defined model

Symbolic Programming

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

PyTorch Hybrid Programming


```
import torch
from torch import nn
from d2l import torch as d2l


# Factory for networks
def get_net():
    net = nn.Sequential(nn.Linear(512, 256),
                        nn.ReLU(),
                        nn.Linear(256, 128),
                        nn.ReLU(),
                        nn.Linear(128, 2))
    return net

x = torch.randn(size=(1, 512))
net = get_net()
net(x)
```

```
net = torch.jit.script(net)
net(x)
```

```
net = get_net()
with Benchmark('Without torchscript'):
    for i in range(1000): net(x)

net = torch.jit.script(net)
with Benchmark('With torchscript'):
    for i in range(1000): net(x)
```

```
Without torchscript: 0.1469 sec
With torchscript: 0.0835 sec
```

Asynchronous Computation

- Asynchronous computation allows your program to start a computation task, say on the GPU, and still be able to perform other tasks
- TensorFlow and MXNet adopts an asynchronous programming model to improve performance
- PyTorch use Python's own scheduler
  - GPU tasks are queued and executed asynchronously
  - The official default implementation of Python is single-threaded
    - But see PEP 703 Making the Global Interpreter Lock Optional in CPython

Asynchrony via Backend

- Using GPU backend

```
# Warmup for GPU computation
device = d2l.try_gpu()
a = torch.randn(size=(1000, 1000), device=device)
b = torch.mm(a, a)

with d2l.Benchmark('numpy'):
    for _ in range(10):
        a = numpy.random.normal(size=(1000, 1000))
        b = numpy.dot(a, a)

with d2l.Benchmark('torch'):
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
```

```
numpy: 1.4693 sec
torch: 0.0022 sec
```

```
with d2l.Benchmark():
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
    torch.cuda.synchronize(device)
```

```
Done: 0.0058 sec
```

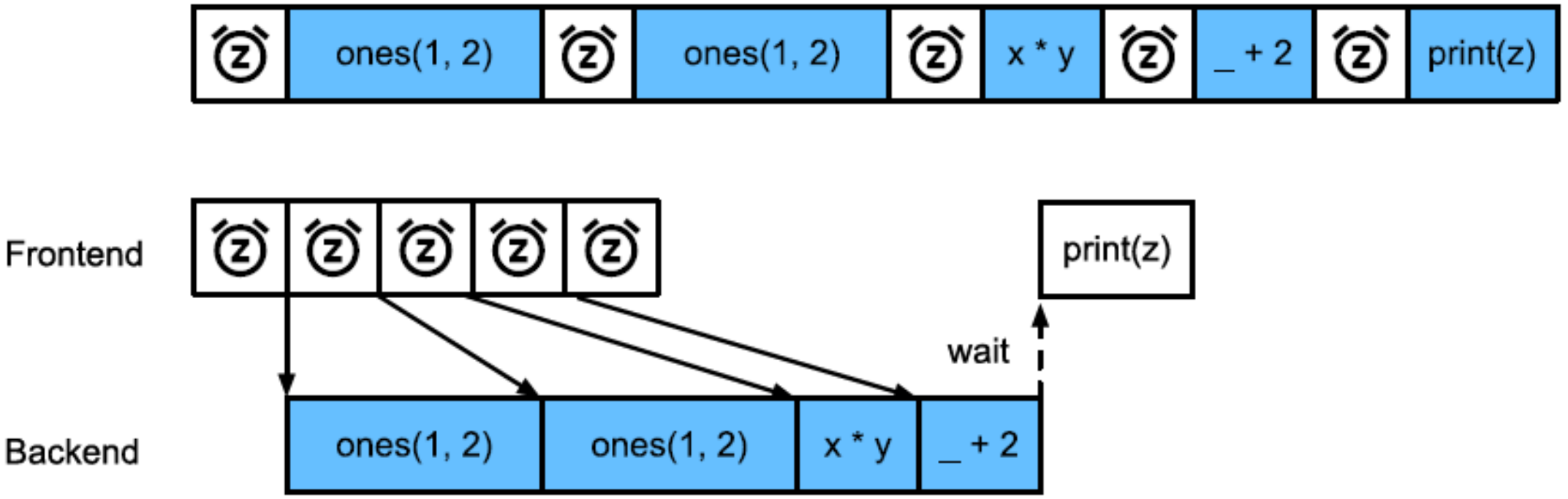
Waits until device finishes

asynchrony

If try to access b right away, PyTorch will automatically add synchronize

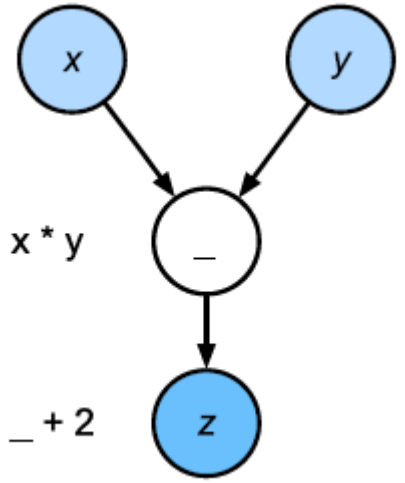
Returns right away

Barriers


ones(1, 2)
ones(1, 2)
x * y
_ + 2
print(z)
Frontend
Backend
print(z)
wait
ones(1, 2)
ones(1, 2)
x * y
_ + 2

Automatic Parallelization

- Deep learning frameworks automatically create computational graphs
- Computational graphs explicitly defines data dependencies, which can be used to parallelize computation
- Example: Computation of x and y can be performed in parallel



Parallel Computation on GPUs


```
devices = d2l.try_all_gpus()
def run(x):
    return [x.mm(x) for _ in range(50)]

x_gpu1 = torch.rand(size=(4000, 4000), device=devices[0])
x_gpu2 = torch.rand(size=(4000, 4000), device=devices[1])
```

```
run(x_gpu1)
run(x_gpu2)  # Warm-up all devices
torch.cuda.synchronize(devices[0])
torch.cuda.synchronize(devices[1])

with d2l.Benchmark('GPU1 time'):
    run(x_gpu1)
    torch.cuda.synchronize(devices[0])

with d2l.Benchmark('GPU2 time'):
    run(x_gpu2)
    torch.cuda.synchronize(devices[1])
```

```
with d2l.Benchmark('GPU1 & GPU2'):
    run(x_gpu1)
    run(x_gpu2)
    torch.cuda.synchronize()
```

```
GPU1 & GPU2: 0.4659 sec
```

```
GPU1 time: 0.4660 sec
GPU2 time: 0.4510 sec
```

Parallel Computation and Communication

- Often, we need copy data between various devices, e.g., CPU to GPU, GPU to CPU, GPU to GPU

Moving data between CPU and GPU is costly

```
def copy_to_cpu(x, non_blocking=False):
    return [y.to('cpu', non_blocking=non_blocking) for y in x]

with d2l.Benchmark('Run on GPU1'):
    y = run(x_gpu1)
    torch.cuda.synchronize()

with d2l.Benchmark('Copy to CPU'):
    y_cpu = copy_to_cpu(y)
    torch.cuda.synchronize()
```

```
Run on GPU1: 0.4656 sec
Copy to CPU: 2.3125 sec
```

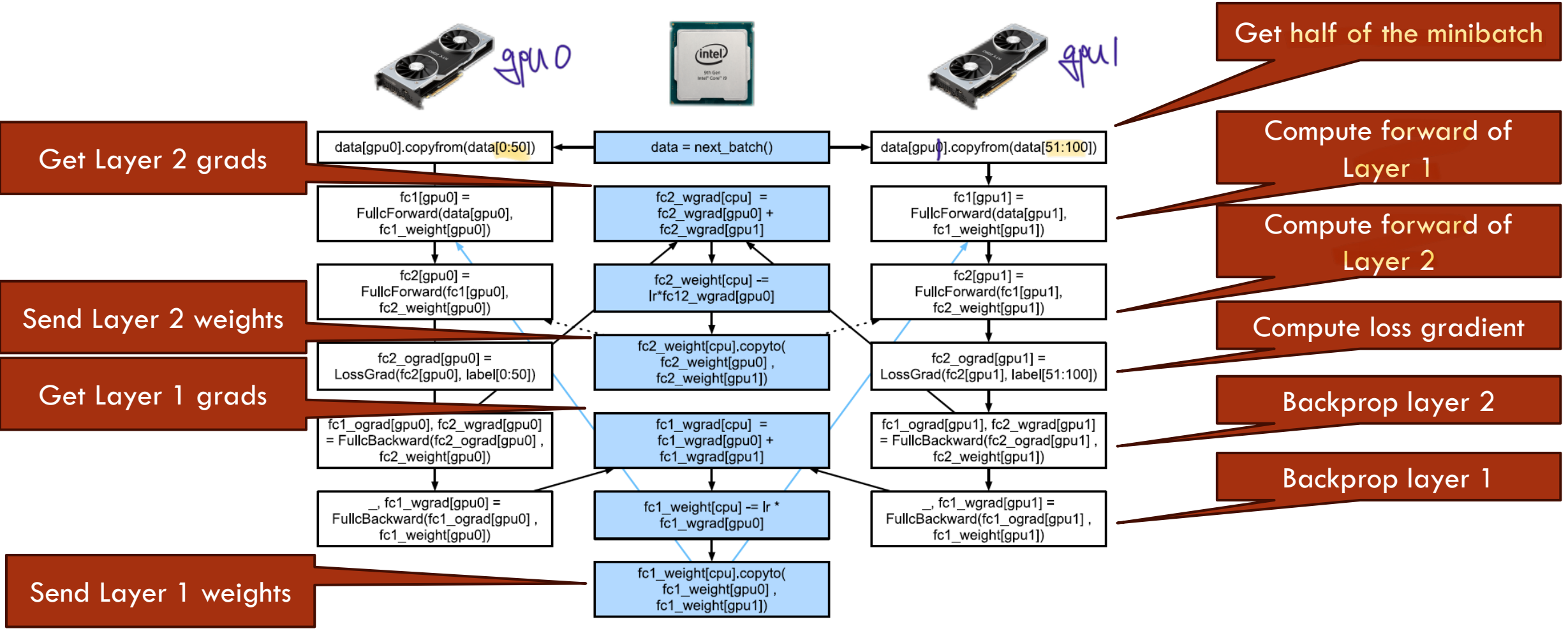
Asynchronous processing can improve performance

```
with d2l.Benchmark('Run on GPU1 and copy to CPU'):
    y = run(x_gpu1)
    y_cpu = copy_to_cpu(y, True)
    torch.cuda.synchronize()
```

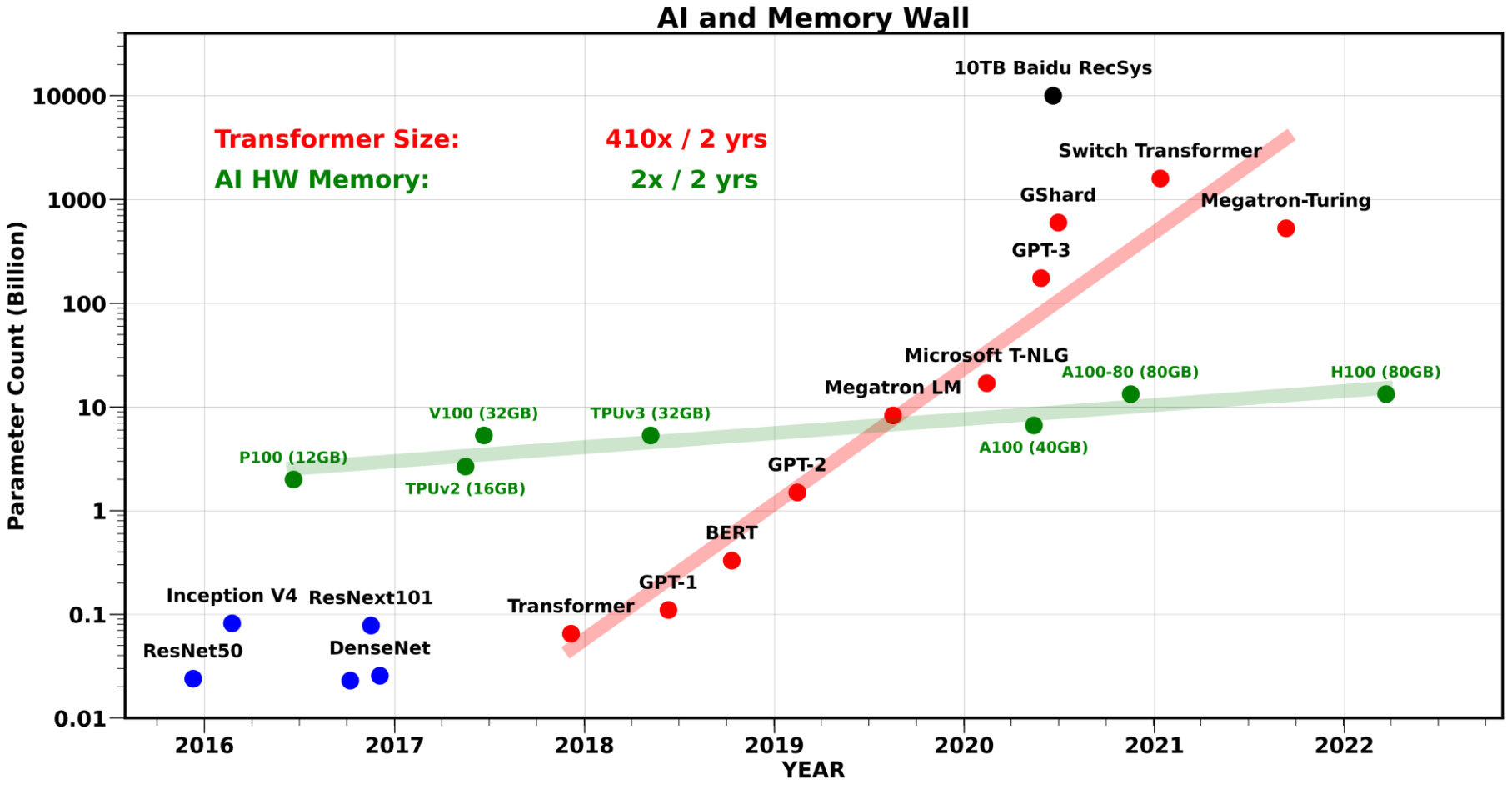
```
Run on GPU1 and copy to CPU: 1.6907 sec
```

Computation Graph Example

- Two-layer MLP on a CPU and two GPUs



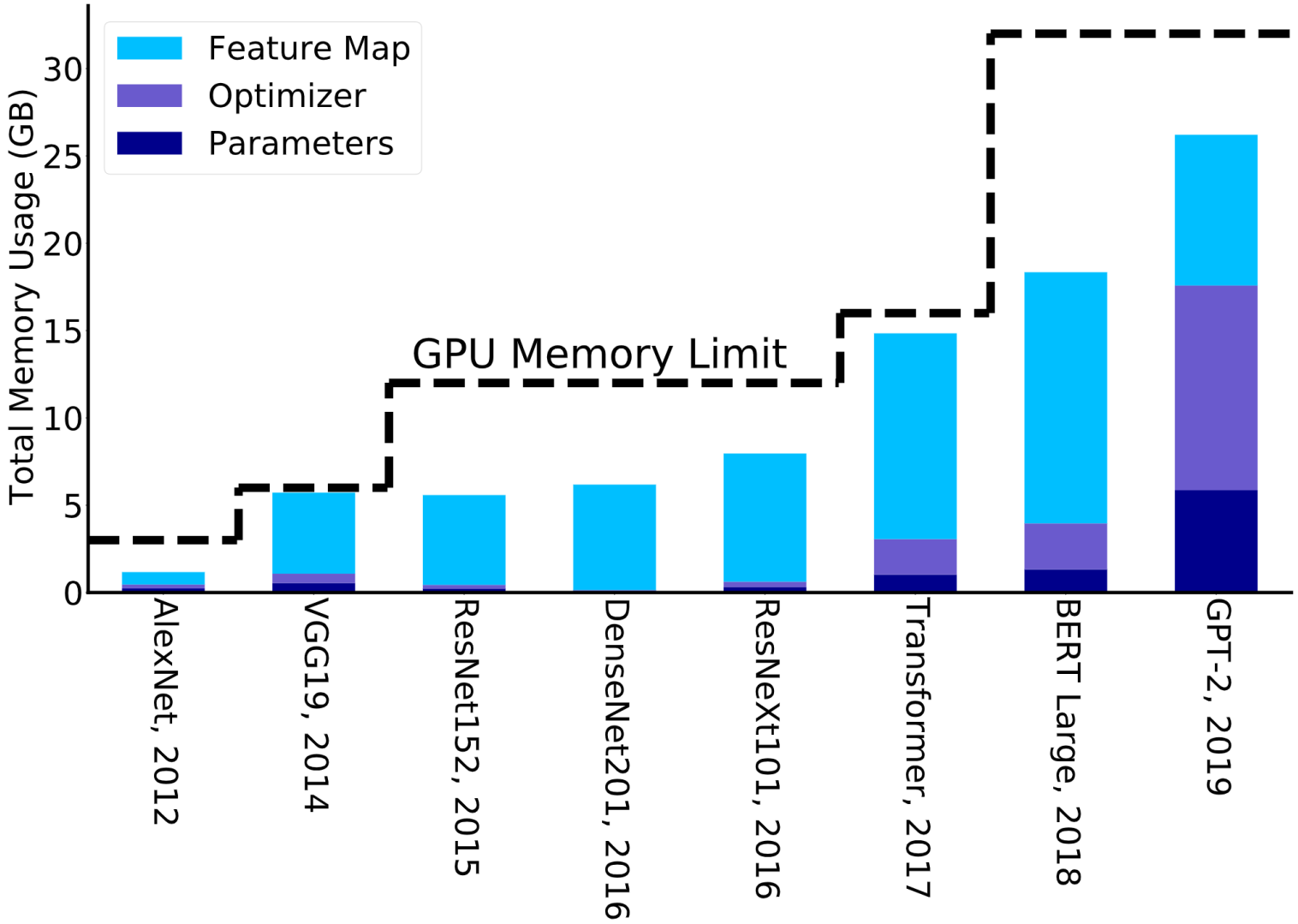
AI and Memory Wall



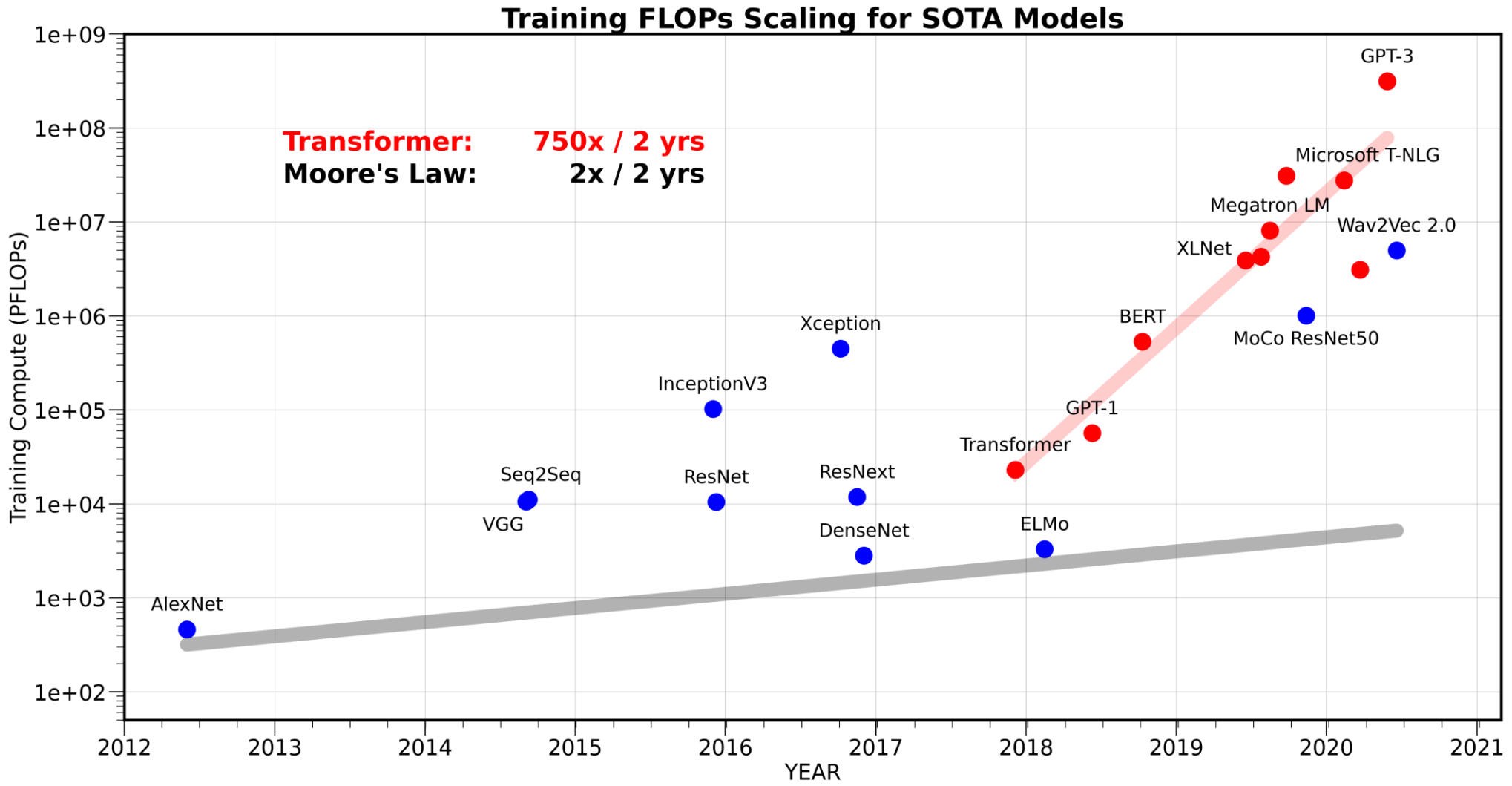
https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8

https://github.com/amirgholami/ai_and_memory_wall

GPU Memory Wall



Compute Needed to Train Transformer Models



Compute and Memory Bandwidth Scaling