# Machine Learning Systems

## Lecture 10: Machine Learning System Stack

Pooyan Jamshidi

UofSC

# Machine Learning Systems Ecosystem

TensorFlow

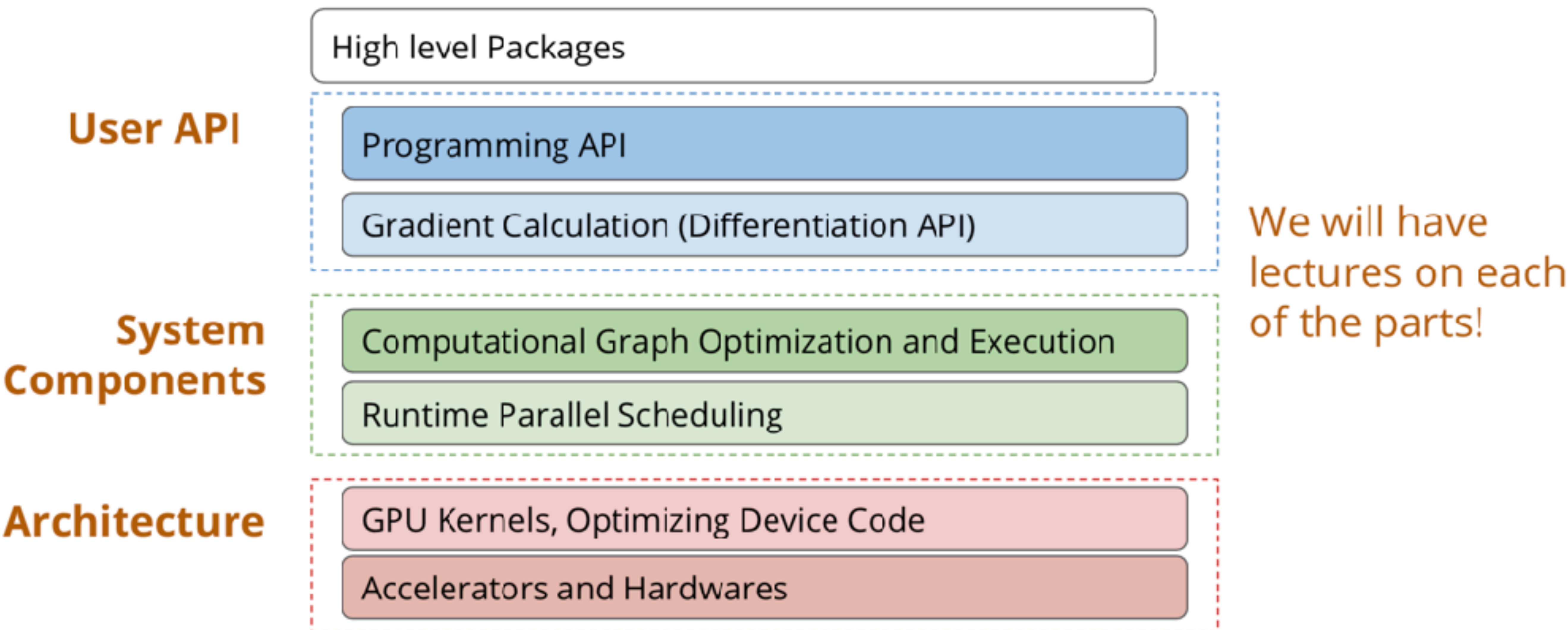
Microsoft CNTK

Chainer

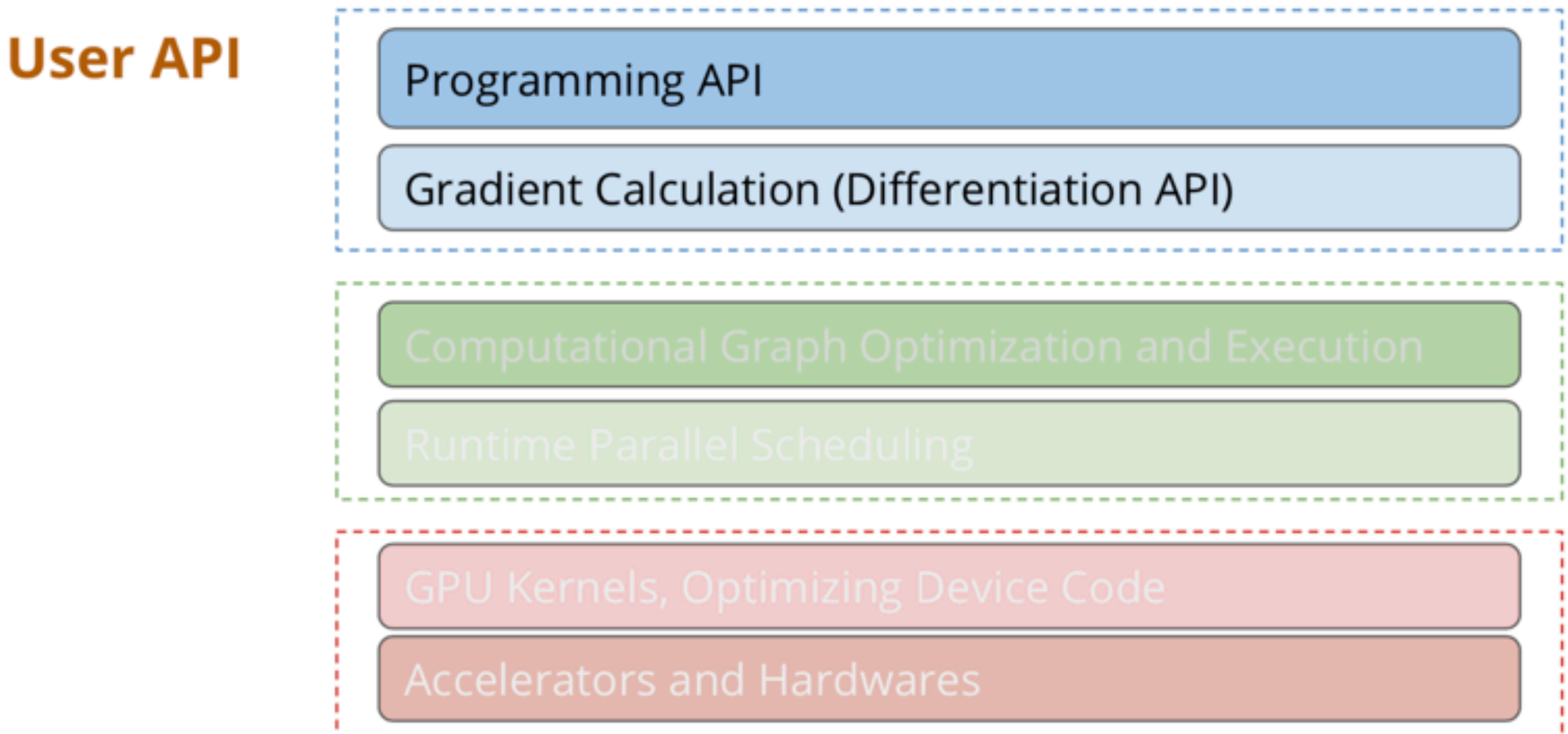
theano

We won't focus on a specific one, but will discuss the common and useful elements of these systems

# Typical Machine Learning System Stack



# Typical Machine Learning System Stack



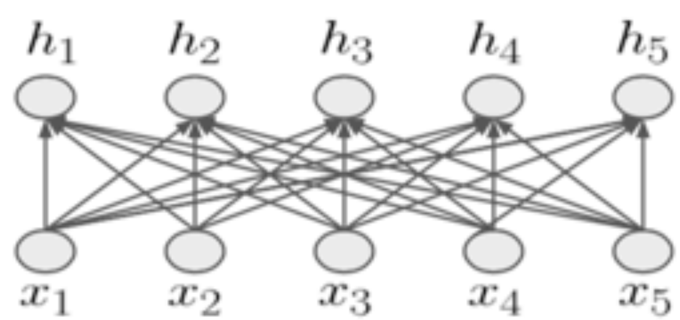
# Example: Logistic Regression

**Data**

$$x_i = \begin{bmatrix} \text{pixel}_1 \\ \text{pixel}_2 \\ \cdots \\ \text{pixel}_m \end{bmatrix}$$

**Fully Connected Layer**

$$h_k = w_k^T x_i$$




**Softmax**

$$P(y_i = k | x_i) = \frac{\exp(h_k)}{\sum_{j=1}^{10} \exp(h_i)}$$

# Logistic Regression in Numpy

```python
import numpy as np
from tinyflow.datasets import get_mnist
def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    x = np.exp(x)
    x = x / np.sum(x, axis=1, keepdims=True)
    return x
# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    # forward
    y = softmax(np.dot(batch_xs, W))
    # backward
    y_grad = y - batch_ys
    W_grad = np.dot(batch_xs.T, y_grad)
    # update
    W = W - learning_rate * W_grad
```

Forward computation:
Compute probability of each class y given input

- Matrix multiplication
  - `np.dot(batch_xs, W)`
- Softmax transform the result
  - `softmax(np.dot(batch_xs, W))`

# Logistic Regression in Numpy

```python
import numpy as np
from tinyflow.datasets import get_mnist
def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    x = np.exp(x)
    x = x / np.sum(x, axis=1, keepdims=True)
    return x
# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    # forward
    y = softmax(np.dot(batch_xs, W))
    # backward
    y_grad = y - batch_ys
    W_grad = np.dot(batch_xs.T, y_grad)
    # update
    W = W - learning_rate * W_grad
```

Weight Update via SGD

$$w \leftarrow w - \eta \nabla_w L(w)$$

# Logistic Regression in TinyFlow (TensorFlow like API)

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Forward Computation **Declaration**

# Logistic Regression in TinyFlow

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Loss function **Declaration**

$$P(\text{label} = k) = y_k$$

$$L(y) = \sum_{k=1}^{10} I(\text{label} = k) \log(y_i)$$

# Logistic Regression in TinyFlow

```python
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Automatic Differentiation: Details in next lecture!

# Logistic Regression in TinyFlow

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

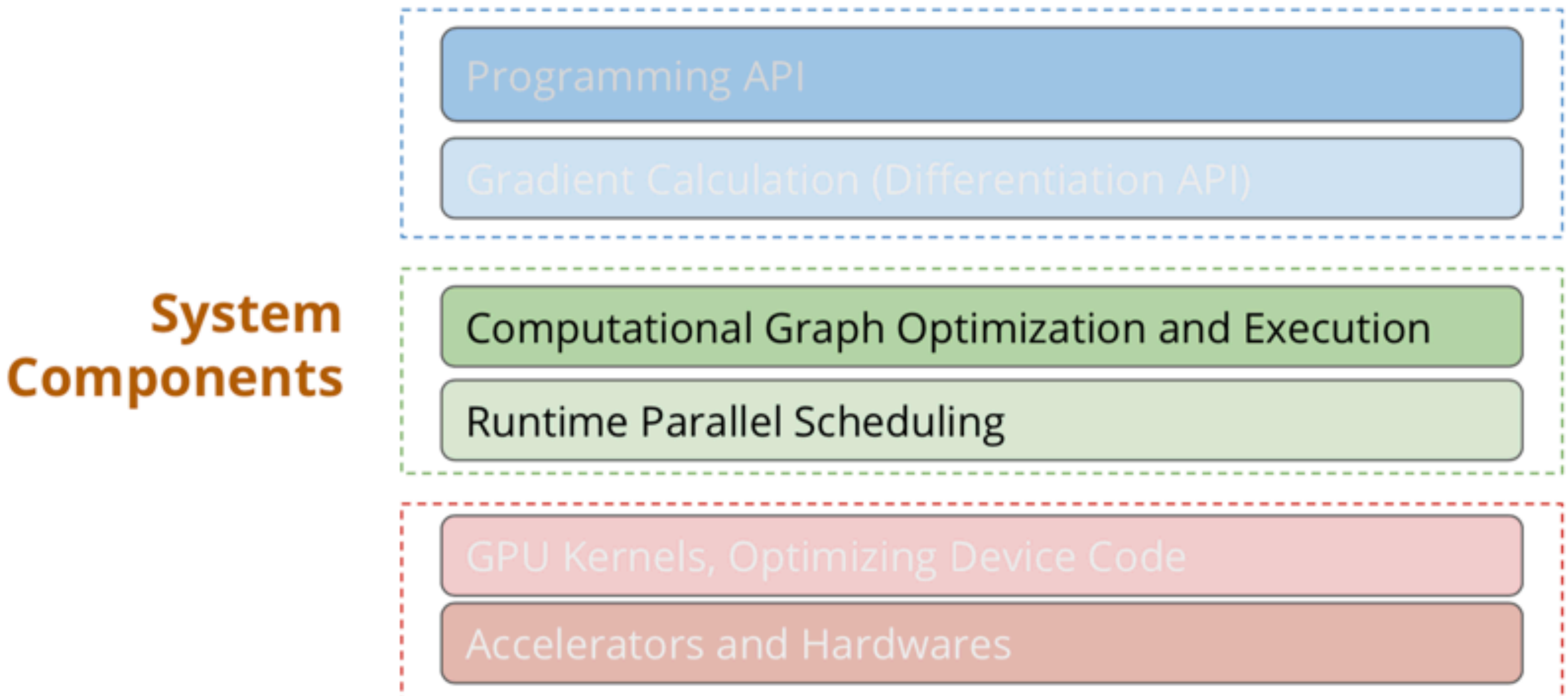
SGD update rule

# Logistic Regression in TinyFlow

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Real execution happens here!

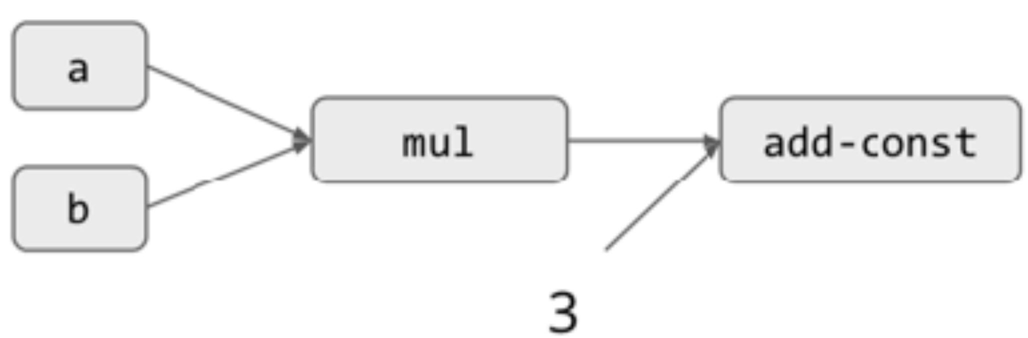
# Typical Deep Learning System Stack



# The Declarative Language: Computation Graph

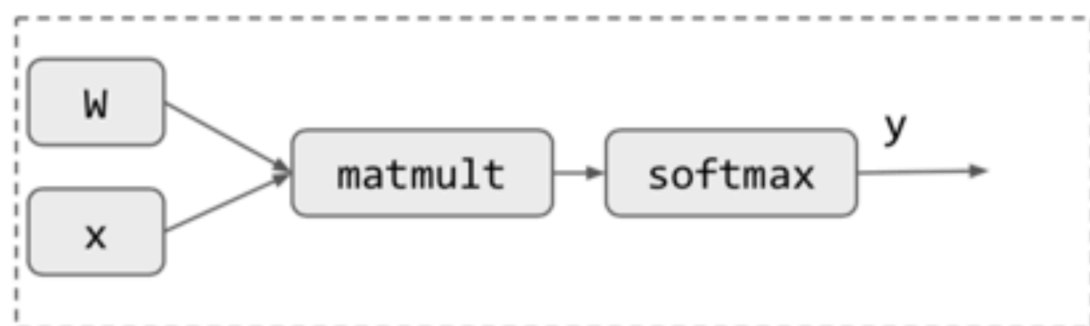
- Nodes represents the computation (operation)
- Edge represents the data dependency between operations

Computational Graph for `a * b +3`



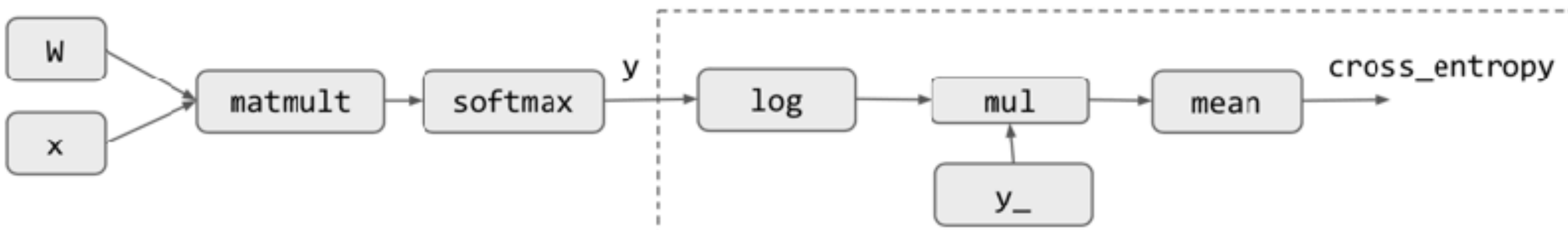
# Computational Graph Construction by Step

```
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
```



# Computational Graph Construction by Step

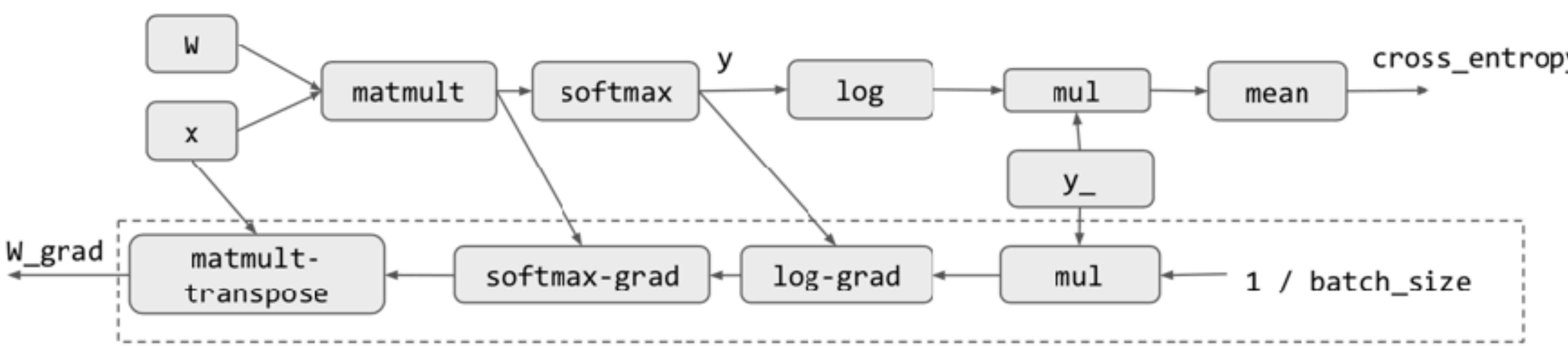
```
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```



# Computational Graph Construction by Step

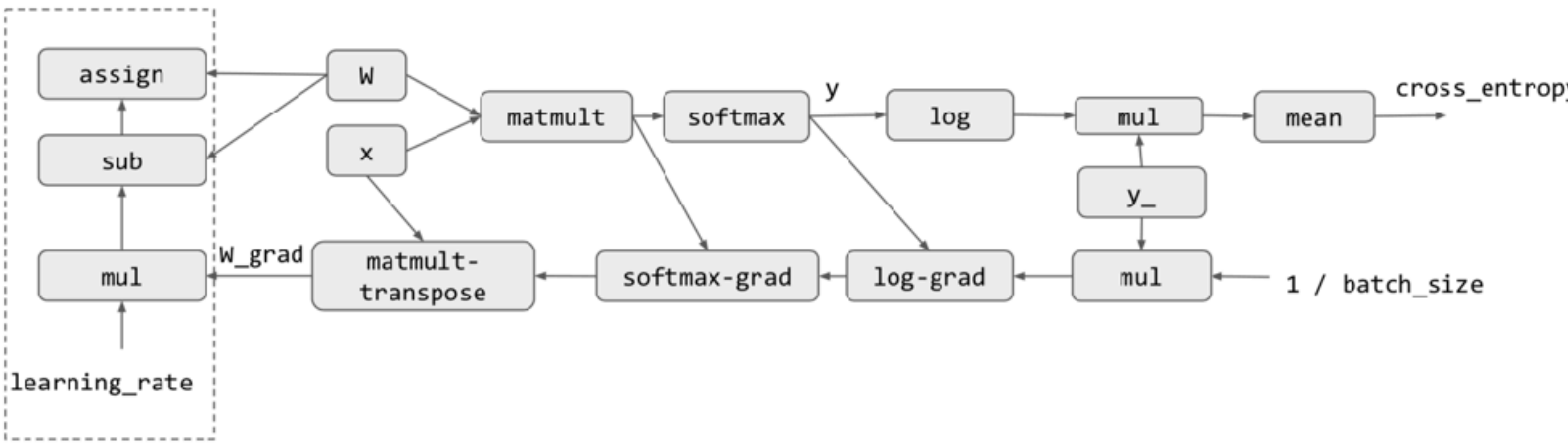
```
W_grad = tf.gradients(cross_entropy, [W])[0]
```

Automatic Differentiation, detail in next lecture!



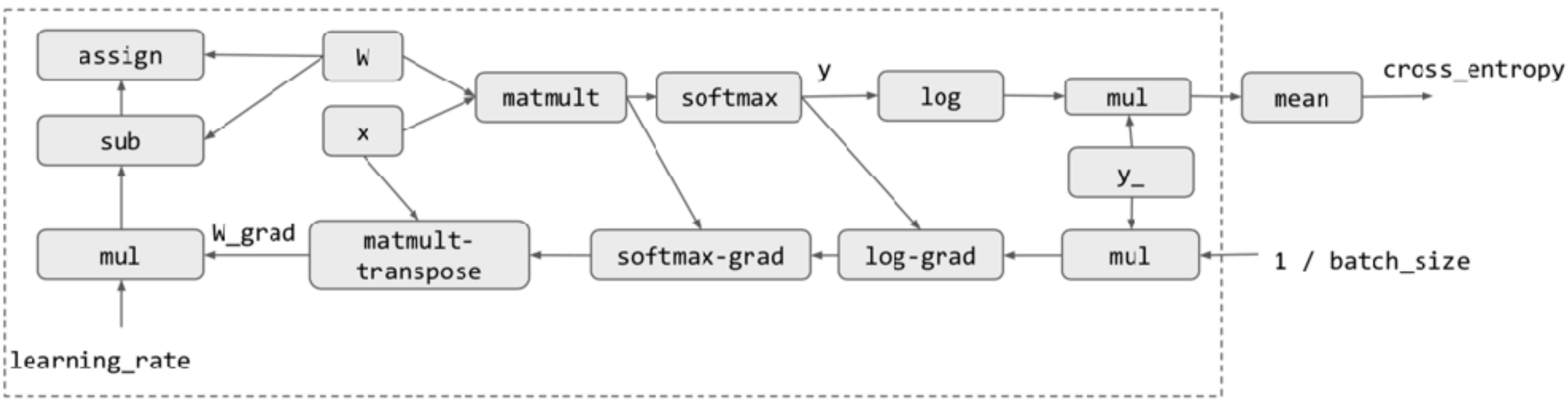
# Computational Graph Construction by Step

```
train_step = tf.assign(W, W - learning_rate * W_grad)
```



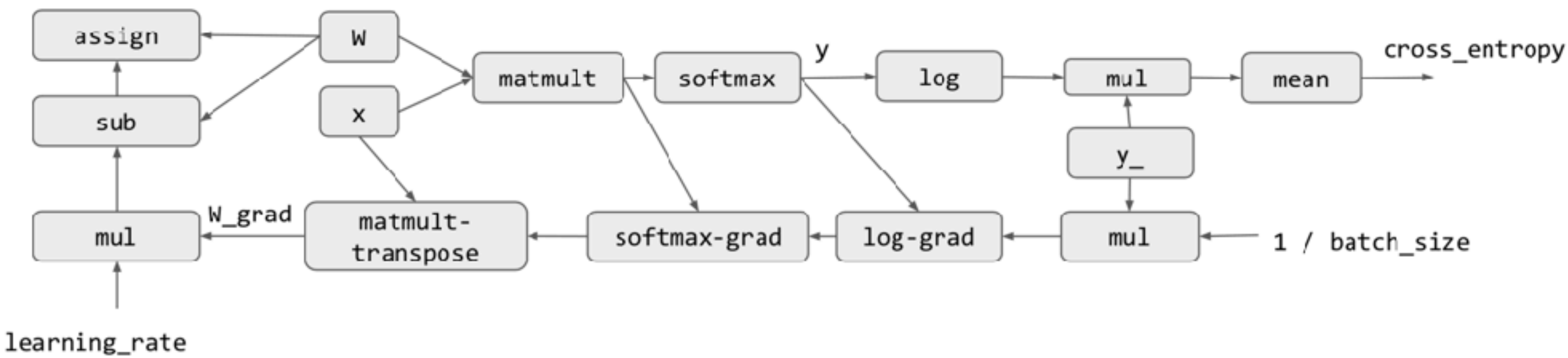
# Execution only Touches the Needed Subgraph

```
sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```



# Discussion: Computational Graph

- What is the benefit of computational graph?
- How can we deploy the model to mobile devices?



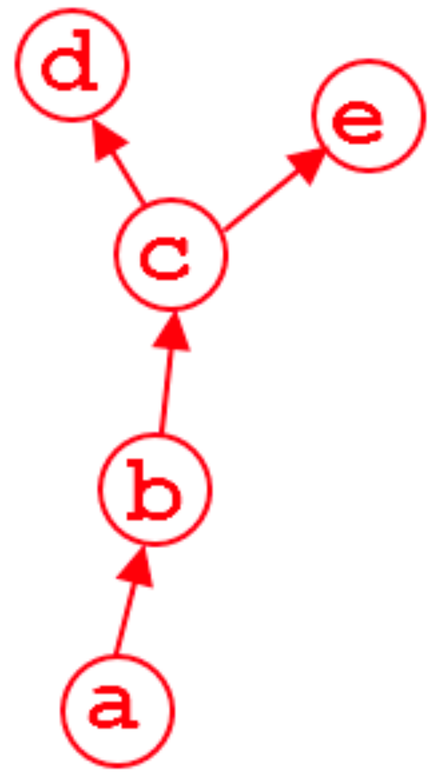
# Discussion: Numpy vs TF Program

What is the benefit/drawback of the TF model vs Numpy Model

```python
import numpy as np
from tinyflow.datasets import get_mnist
def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    x = np.exp(x)
    x = x / np.sum(x, axis=1, keepdims=True)
    return x
# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    # forward
    y = softmax(np.dot(batch_xs, W))
    # backward
    y_grad = y - batch_ys
    W_grad = np.dot(batch_xs.T, y_grad)
    # update
    W = W - learning_rate * W_grad
```

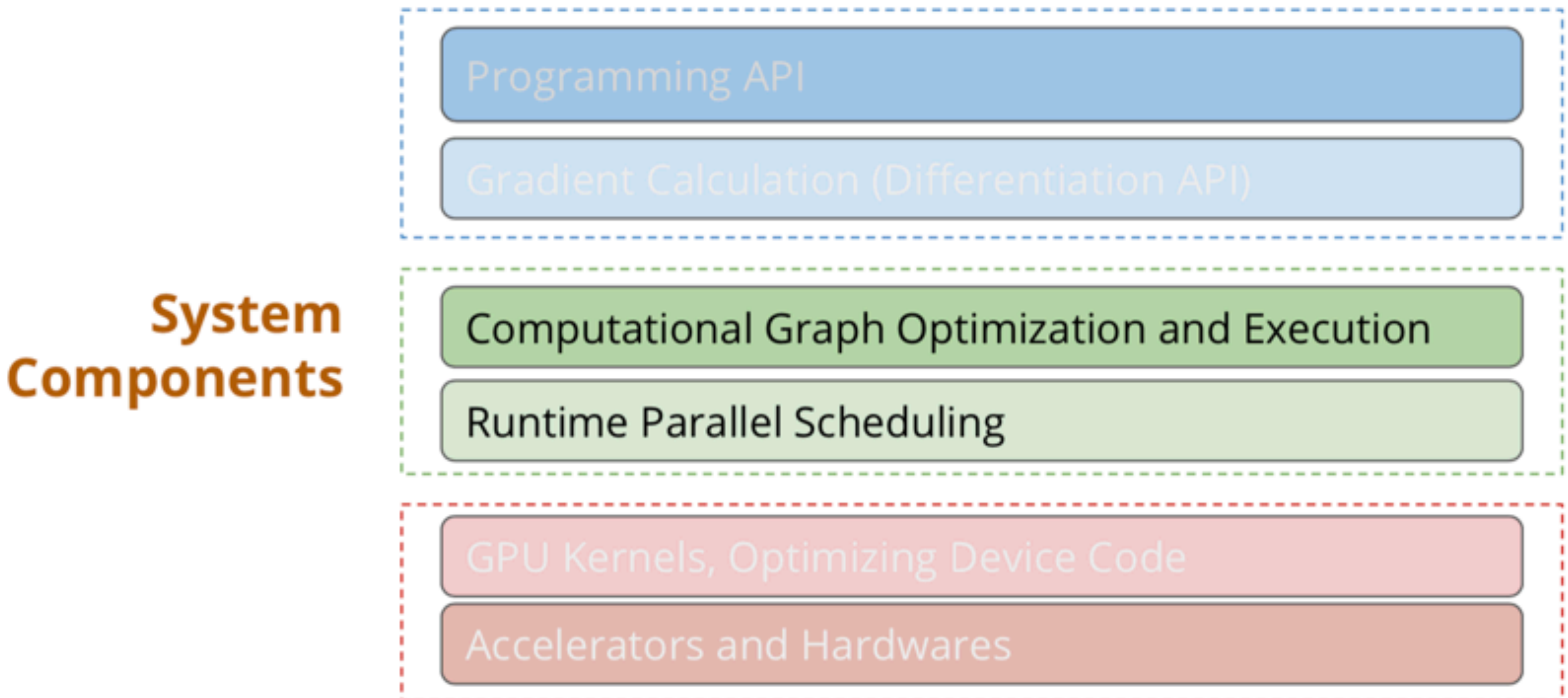
```python
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

# Computational graphs in other frameworks: PyTorch



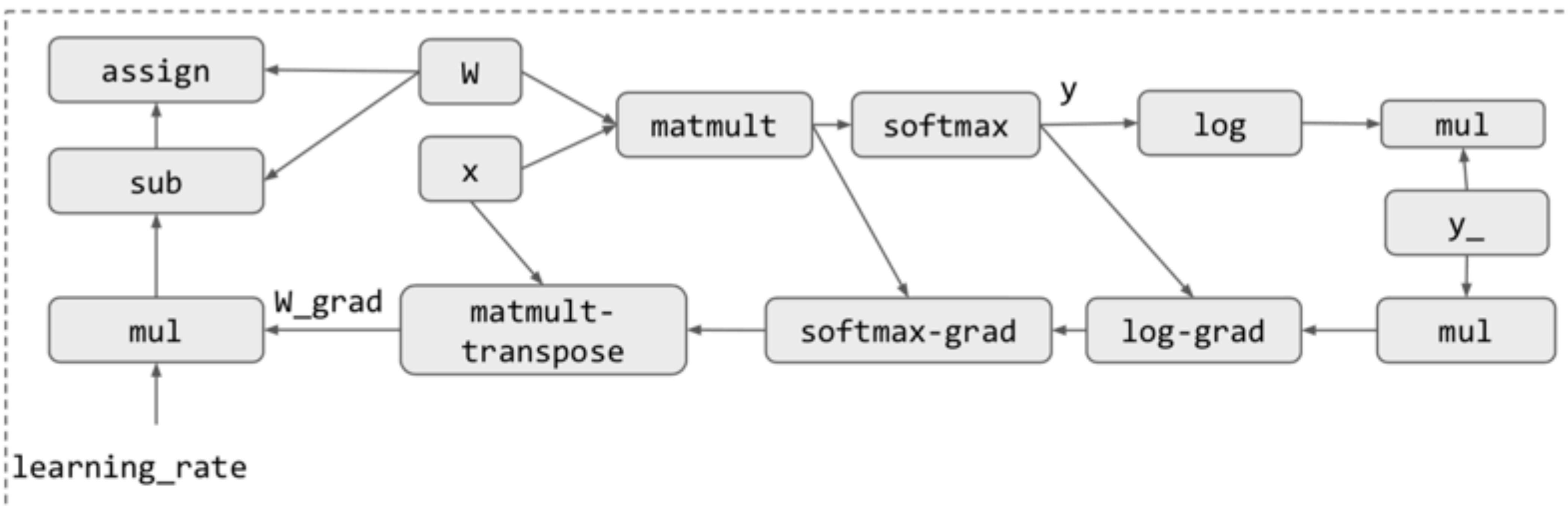

```
import torch
from torch.autograd import Variable
a = Variable(torch.rand(1, 4), requires_grad=True)
b = a**2
c = b*2
d = c.mean()
e = c.sum()
```

# Typical Deep Learning System Stack



# Computation Graph Optimization

- E.g. Deadcode elimination
- Memory planning and optimization
- What other possible optimization can we do given a computational graph?

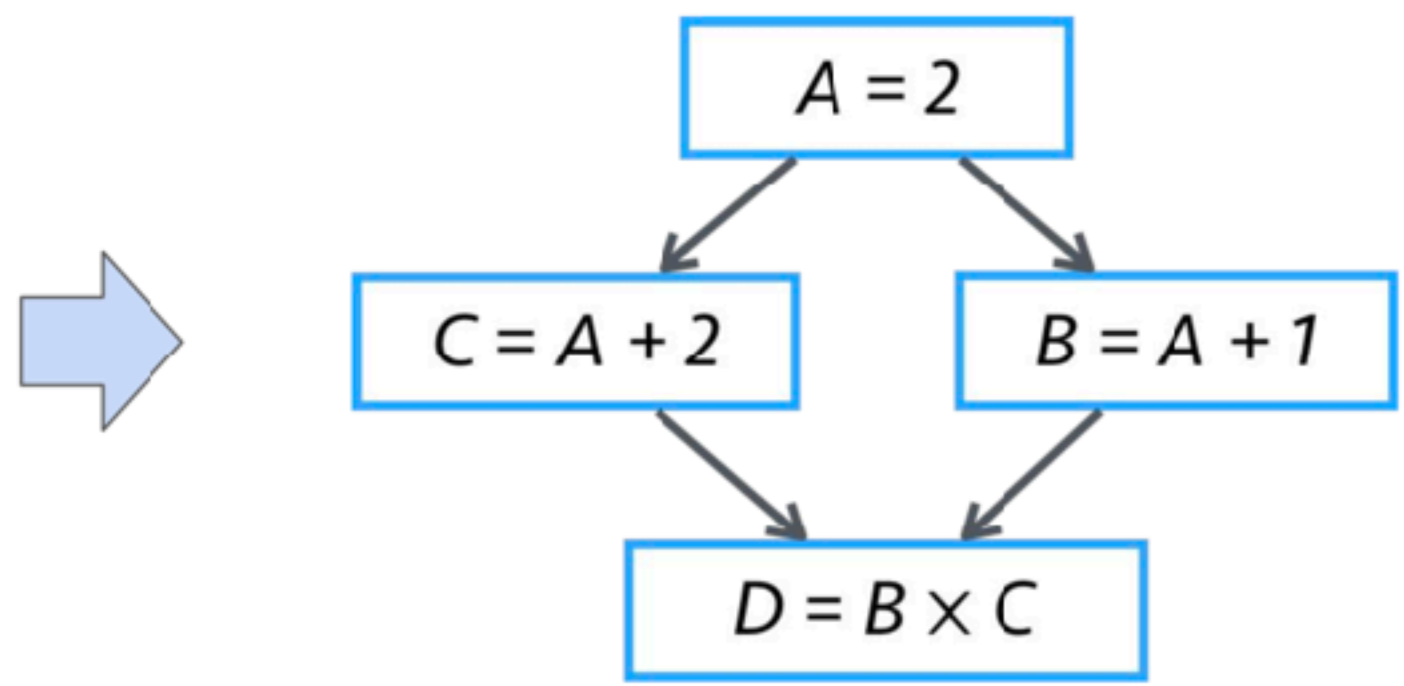


# Parallel Scheduling

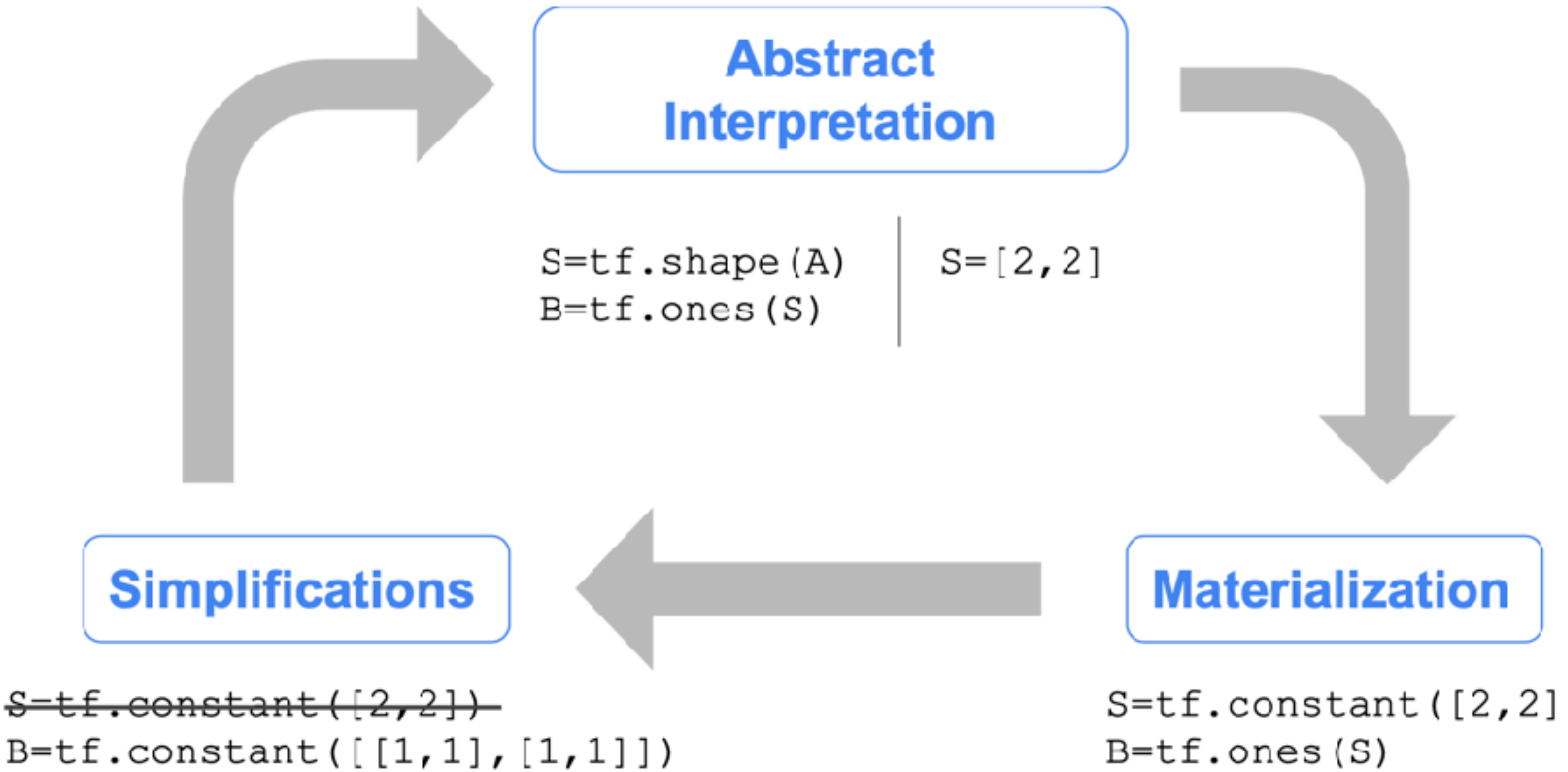
- Code need to run parallel on multiple devices and worker threads
- Detect and schedule parallelizable patterns
- Detail lecture on later

MXNet Example

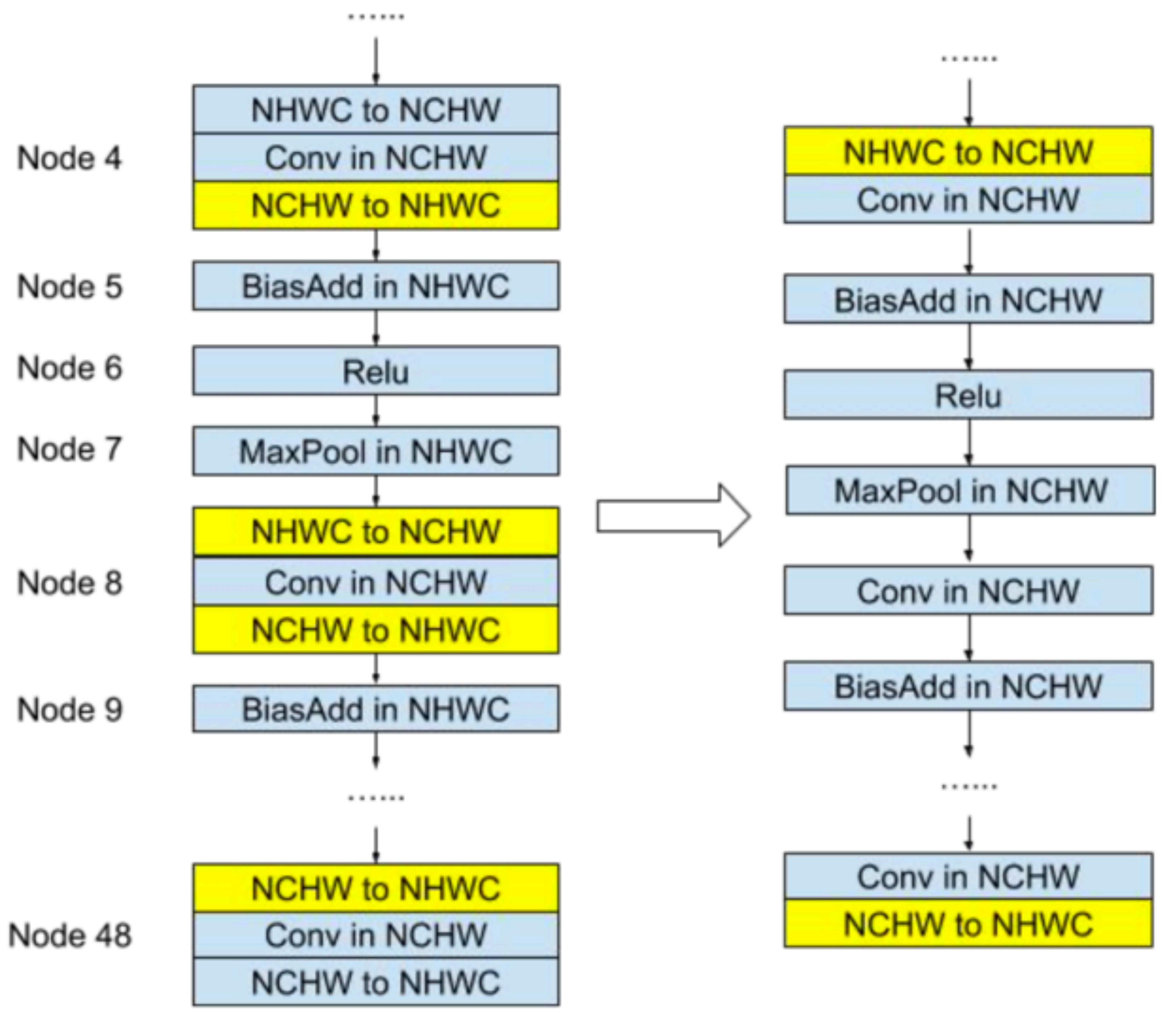
```
>>> import mxnet as mx
>>> A = mx.nd.ones((2,2)) *2
>>> C = A + 2
>>> B = A + 1
>>> D = B * C
```



# Graph Simplifications

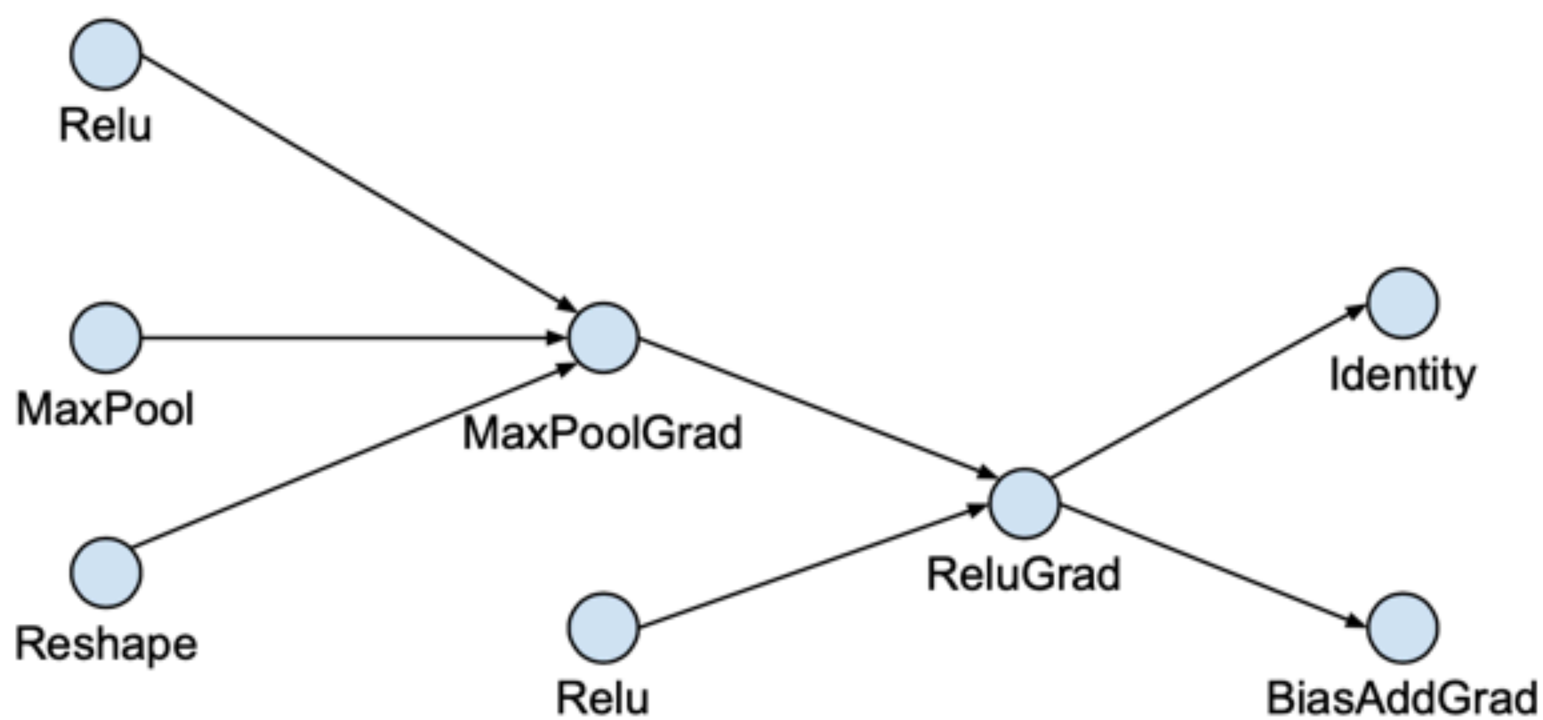


# Layout Optimization



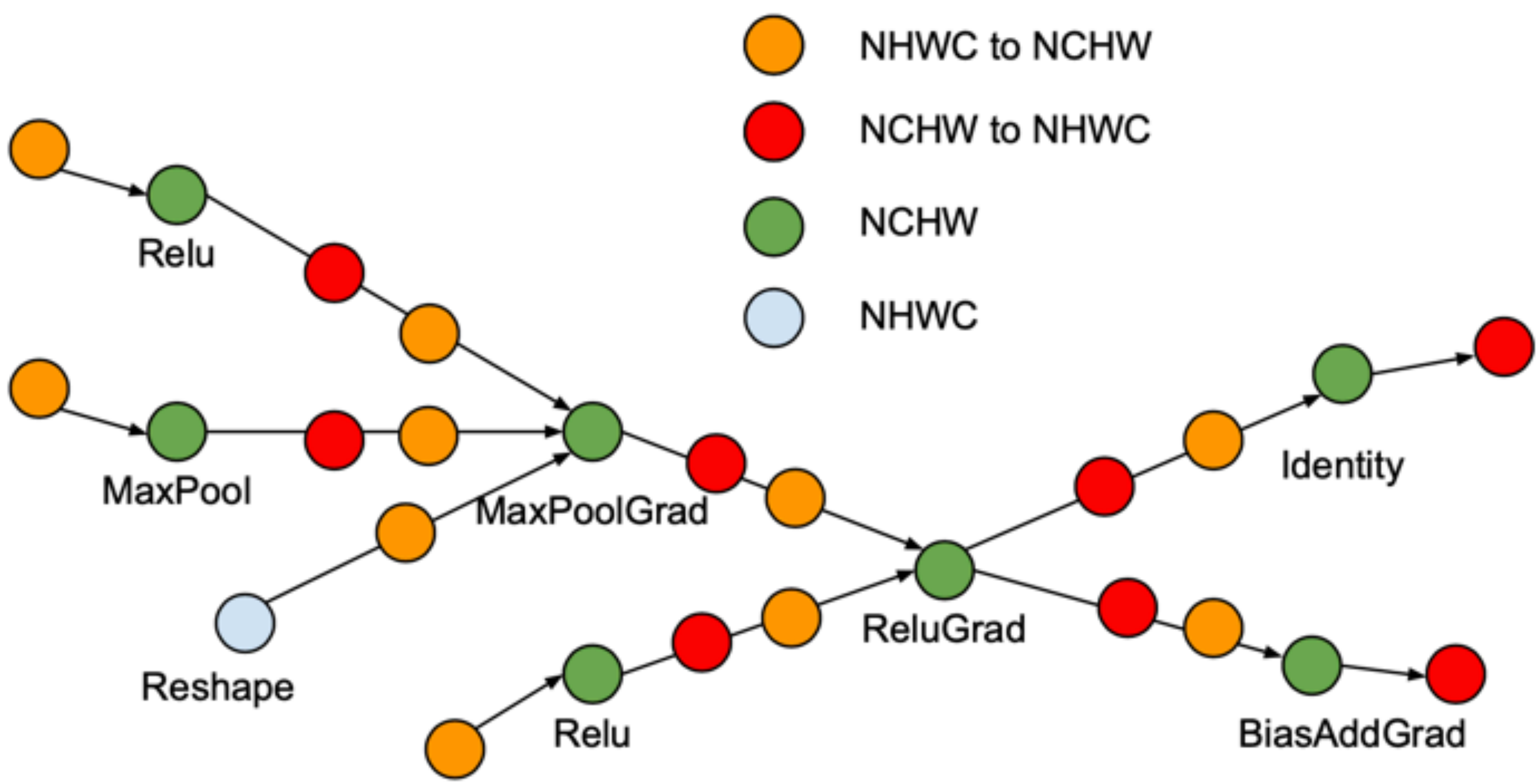
# Layout Optimization

Example: Original graph with all ops in NHWC format

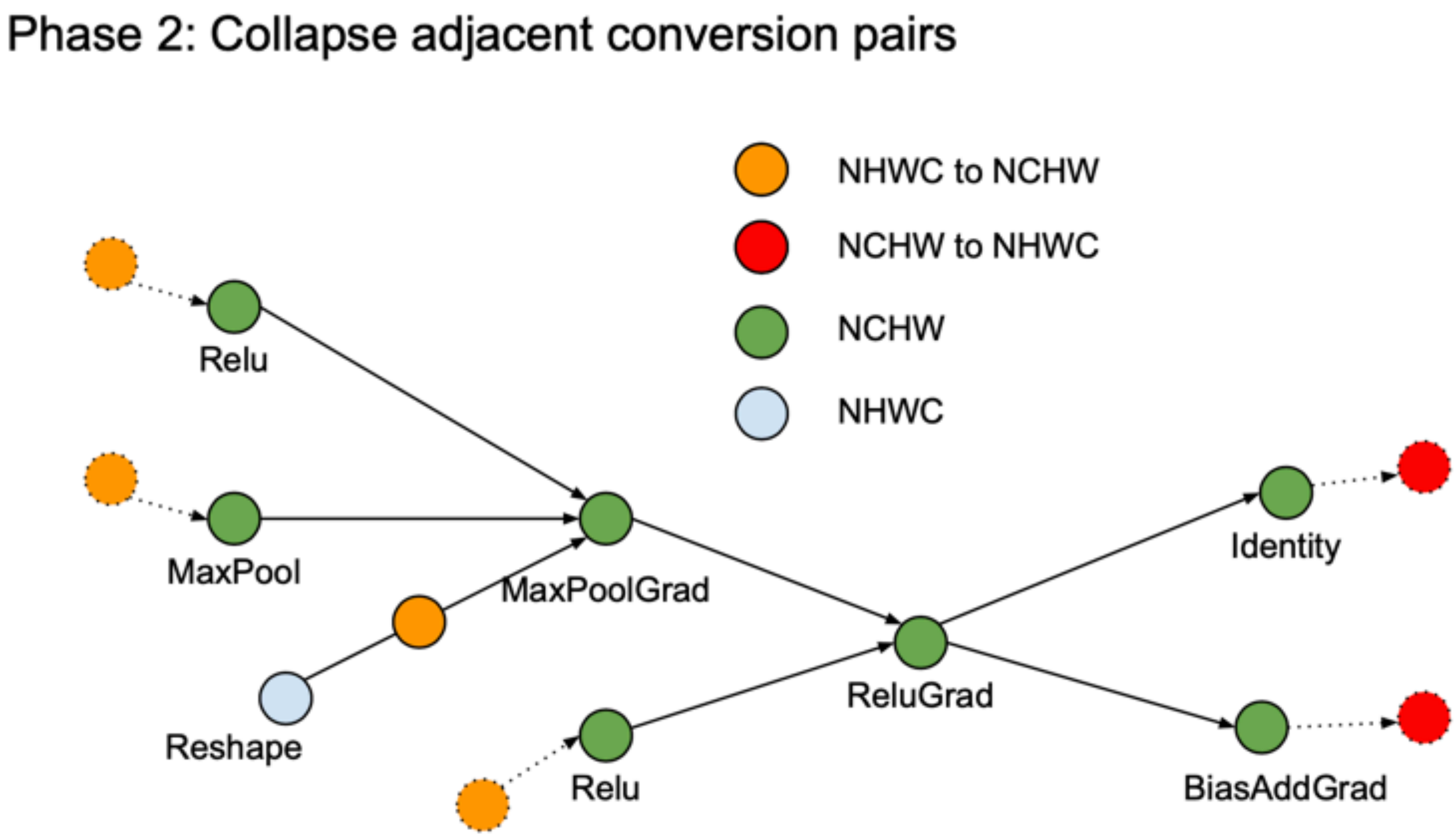


# Layout Optimization

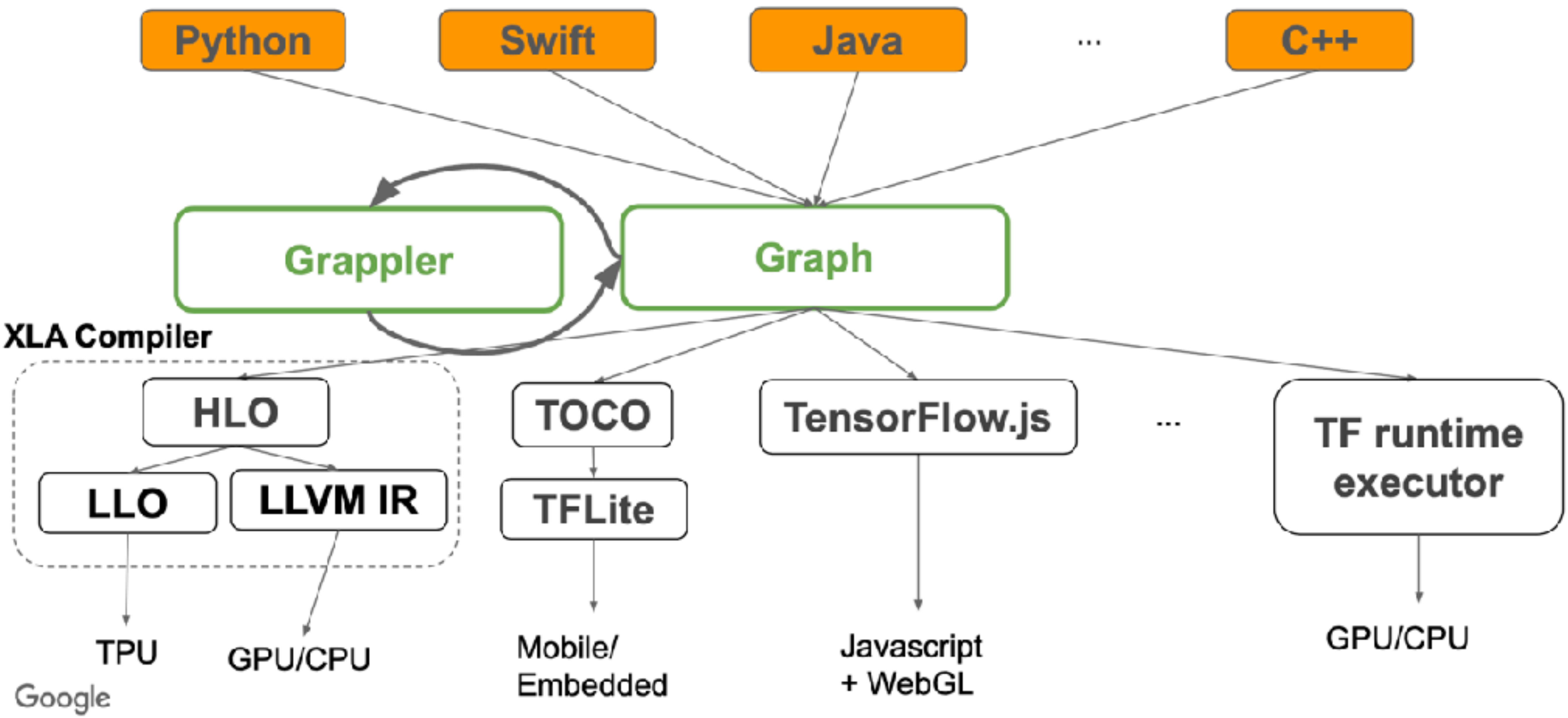
Phase 1: Expand by inserting conversion pairs



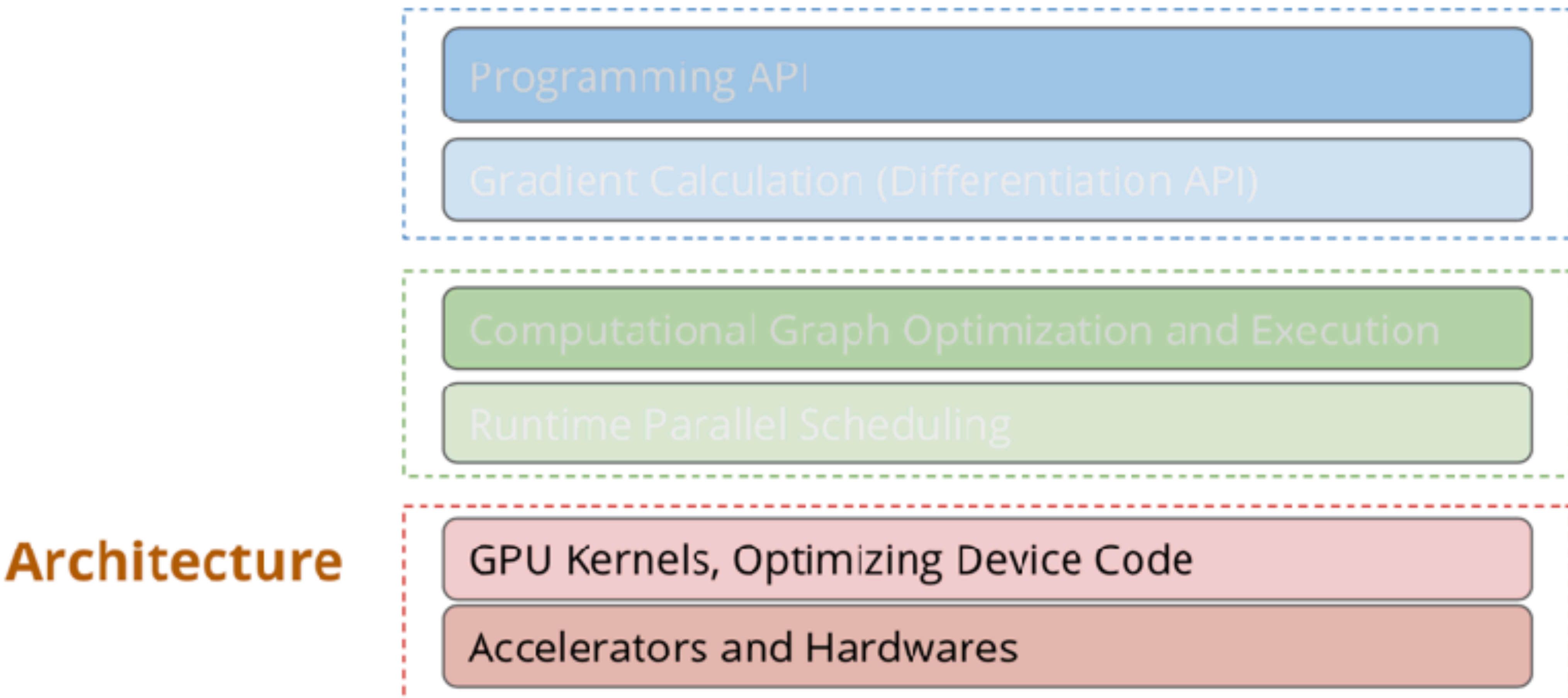
# Layout Optimization



# Computation Graph Optimization



# Typical Deep Learning System Stack

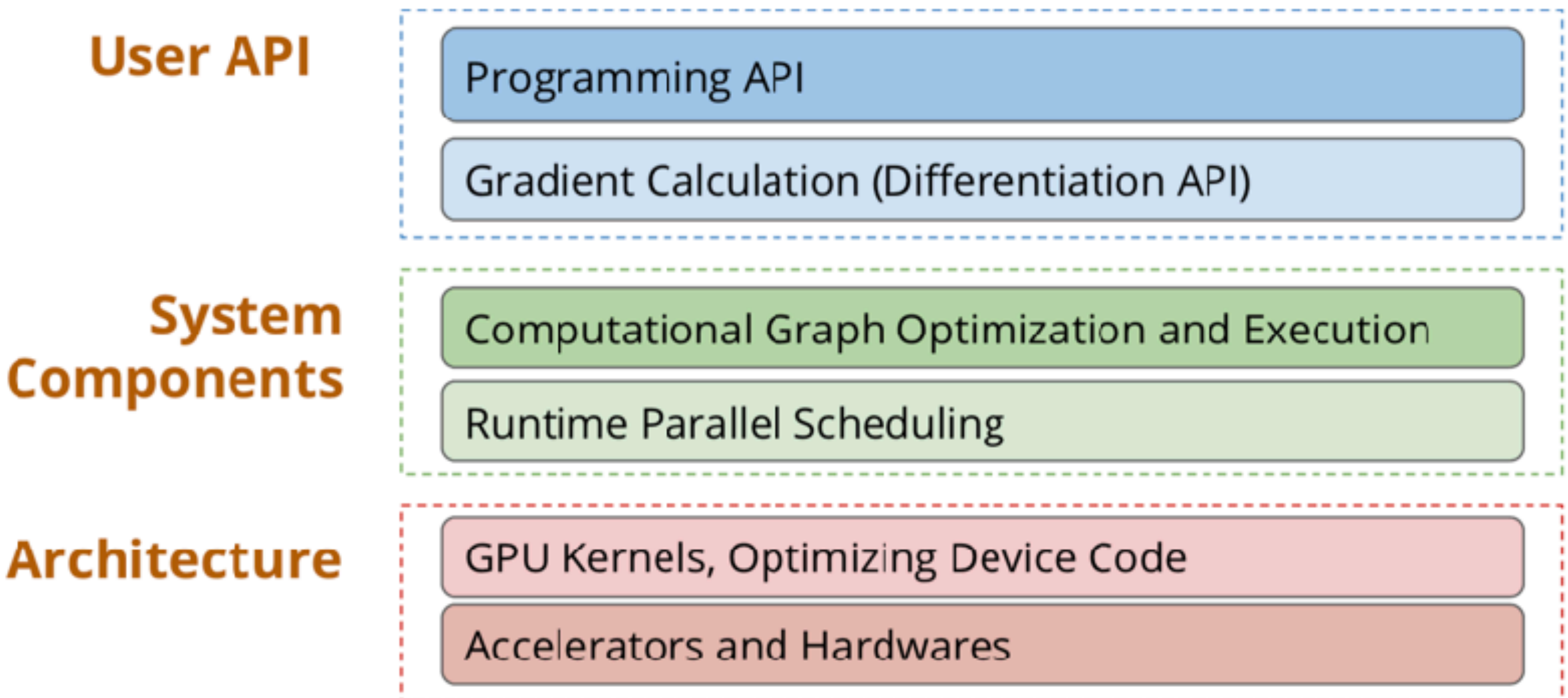


# GPU Acceleration

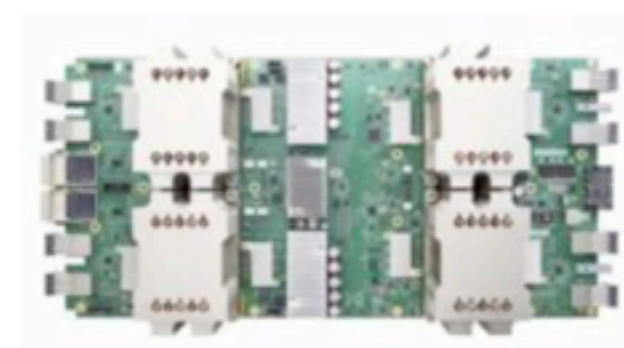
- Most existing deep learning programs runs on GPUs
- Modern GPU have Teraflops of computing power

# Typical Deep Learning System Stack

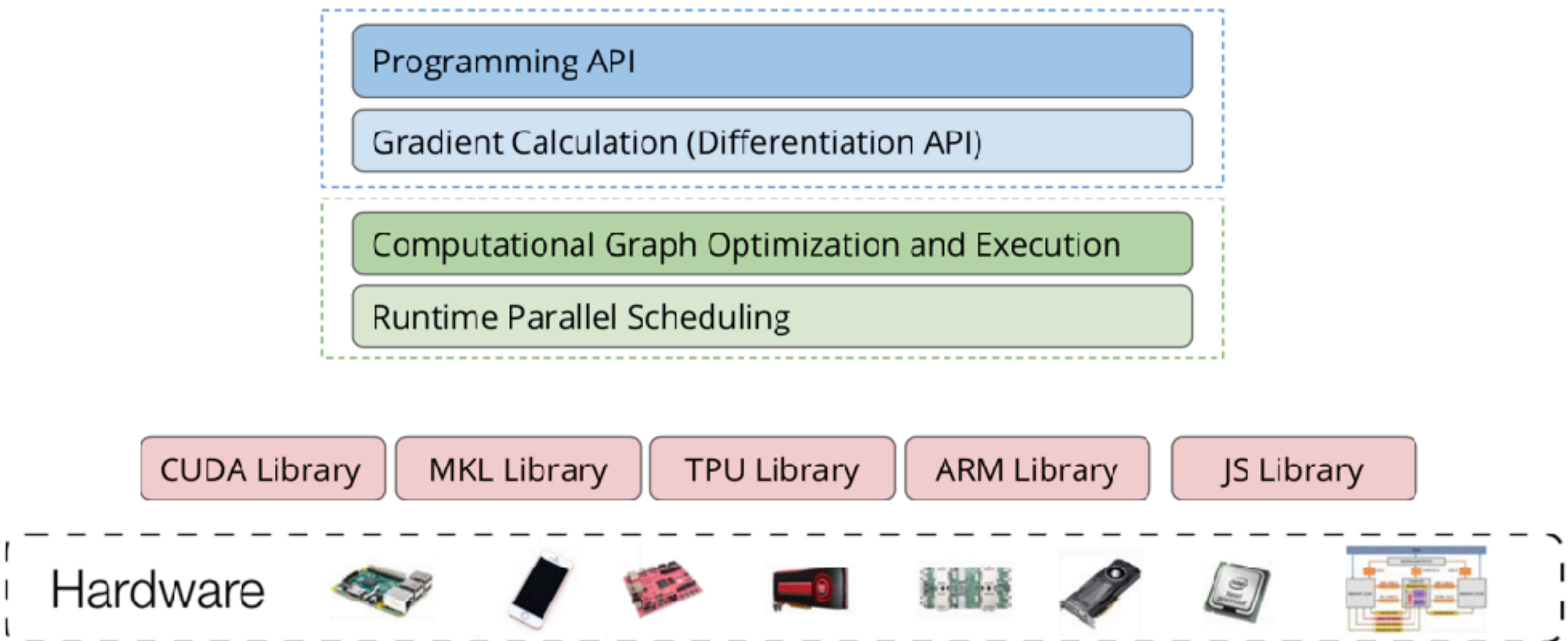
Not a comprehensive list of elements
The systems are still rapidly evolving :)



# Supporting More Hardware backends

# Each Hardware backend requires a software stack



# New Trend: Compiler based Approach