Lecture 9: Scheduling

CSE599G1: Spring 2017
Next Week

● Two Joint Sessions with Computer Architecture Class

● Different date, time and location, detail to be announced

● Wed: ASICs for deep learning

● Friday: FPGA in the data center
Where are we

**High level Packages**
- Programming API
- Gradient Calculation (Differentiation API)

**System Components**
- Computational Graph Optimization and Execution
- Runtime Parallel Scheduling

**Architecture**
- GPU Kernels, Optimizing Device Code
- Accelerators and Hardwares
Where are we

- Programming API
- Gradient Calculation (Differentiation API)
- Computational Graph Optimization and Execution
- Runtime Parallel Scheduling
- GPU Kernels, Optimizing Device Code
- Accelerators and Hardwares
Parallelization Problem

- Parallel execution of concurrent kernels
- Overlap compute and data transfer

Parallel over multiple streams

Serial execution
Recap: Deep Learning Training Workflow

Gradient Calculation

Interactions with Model

Parameter Update

\[ w = w - \eta \partial f(w) \]
Questions to be answered

- What are common patterns of parallelization
- How can we easily achieve these patterns
- What about dynamic style program
Model Parallel Training

- Map parts of workload to different devices
- Require special dependency patterns (wave style)
  - e.g. LSTM
Data Parallelism

● Train replicated version of model in each machine
● Synchronize the gradient
Data Parallel Training

```
data[gpu0].copyfrom(data[0:50])
fc1[_gpu0] = FullyForward(data[gpu0], fc1_weight[gpu0])
fc2[_gpu0] = FullyForward(fc1[_gpu0], fc2_weight[gpu0])
fc2ograd[_gpu0] = LossGrad(fc2[_gpu0], label[0:50])
fc1ograd[_gpu0], fc2ograd[_gpu0] = FullBackward(fc2ograd[_gpu0], fc2_weight[gpu0])
fc1ograd[_gpu0] = FullBackward(fc1ograd[_gpu0], fc1_weight[gpu0])

data = next_batch()
fc2_wgrad[cpu] = fc2_wgrad[gpu0] + fc2_wgrad[cpu]
fc2_weight[cpu] = lr * fc2_wgrad[cpu]
fc2_weight[cpu].copyto(fc2_weight[cpu0], fc2_weight[cpu1])
fc1ograd[cpu], fc2ograd[cpu] = FullBackward(fc2ograd[cpu], fc2_weight[cpu1])
fc1ograd[cpu] = FullBackward(fc1ograd[cpu], fc1_weight[cpu1])
fc1_weight[cpu] = lr * fc1_wgrad[cpu0]
fc1_weight[cpu].copyto(fc1_weight[cpu0], fc1_weight[cpu1])

data[gpu0].copyfrom(data[51:100])
fc1[cpu1] = FullyForward(data[cpu1], fc1_weight[cpu1])
fc2[cpu1] = FullyForward(fc1[cpu1], fc2_weight[cpu1])
fc2ograd[cpu1] = LossGrad(fc2[cpu1], label[51:100])
fc1ograd[cpu1], fc2ograd[cpu1] = FullBackward(fc2ograd[cpu1], fc2_weight[cpu1])
fc1ograd[cpu1] = FullBackward(fc1ograd[cpu1], fc1_weight[cpu1])
fc1_weight[cpu1] = lr * fc1_wgrad[cpu1]
fc1_weight[cpu1].copyto(fc1_weight[cpu0], fc1_weight[cpu1])
```
The Gap for Communication

Which operations can run in currently with synchronization of g2/w2?
Parallel Programs are Hard to Write

We need an automatic scheduler
Goal of Scheduler Interface

- Write Serial Program
- Possibly dynamically (not declare graph beforehand)

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

- Run in Parallel
- Respect serial execution order
Discussion: How to schedule the following ops

- Random number generator
- Memory recycling
- Cross device copy
- Send data over network channel

\[
\begin{align*}
A &= 2 \\
B &= A + 1 \\
C &= A + 2 \\
D &= B \times C
\end{align*}
\]
Data Flow Dependency

Code

\[
\begin{align*}
A &= 2 \\
B &= A + 1 \\
C &= A + 2 \\
D &= B \times C
\end{align*}
\]

Dependency

\[
\begin{align*}
A &= 2 \\
C &= A + 2 \\
B &= A + 1 \\
D &= B \times C
\end{align*}
\]
Write After Read Mutation

Code

A = 2
B = A + 1
C = A + 2
A = 3

Dependency
Memory Recycle

Code

\[
\begin{align*}
  A &= 2 \\
  B &= A + 1 \\
  C &= A + 2
\end{align*}
\]

A.__del__()
Random Number Generator

Code

```python
rnd = RandomNGenerator()
B = rnd.uniform(10, -10)
C = rnd.uniform(10, -10)
```

Dependency

```
rnd = RandomNGenerator()
rnd.uniform(10, -10)
rnd.uniform(10, -10)
```
Goal of Scheduler Interface

- Schedule any resources
  - Data
  - Random number generator
  - Network communicator

- Schedule any operation
DAG Graph based scheduler

Interface:

```
engine.push(lambda op, deps=[])  
```

- Explicit push operation and its dependencies
- Can reuse the computation graph structure
- Useful when all results are immutable
- Used in typical frameworks (e.g. TensorFlow)

- What are the drawbacks?
Pitfalls when using Scheduling Mutations

Write after Read
\[
\text{tf.assign}(A, B + 1) \\
\text{tf.assign}(T, B + 2) \\
\text{tf.assign}(B, 2)
\]

A **mutation aware** scheduler can solve these problems much easier than DAG based scheduler

Read after Write
\[
T = \text{tf.assign}(B, B + 1) \\
\text{tf.assign}(A, B + 2)
\]
MXNet Program for Data Parallel Training

```python
for dbatch in train_iter:
    % iterating on GPUs
    for i in range(ngpu):
        % pull the parameters
        for key in update_keys:
            kvstore.pull(key, execs[i].weight_array[key])
        % compute the gradient
        execs[i].forward(is_train=True)
        execs[i].backward()
        % push the gradient
        for key in update_keys:
            kvstore.push(key, execs[i].grad_array[key])
```
Mutation aware Scheduler: Tag each Resource

<table>
<thead>
<tr>
<th>Code</th>
<th>Original Resources</th>
<th>Tagged Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A.var = engine.new_variable()</code></td>
<td>A.data</td>
<td>A.data v1</td>
</tr>
<tr>
<td><code>B.var = engine.new_variable()</code></td>
<td>B.data</td>
<td>B.data v2</td>
</tr>
<tr>
<td><code>C.var = engine.new_variable()</code></td>
<td>C.data</td>
<td>C.data v3</td>
</tr>
<tr>
<td><code>rnd.var = engine.new_variable()</code></td>
<td>rnd.gen</td>
<td>rnd.gen v4</td>
</tr>
</tbody>
</table>
**Mutation aware Scheduler: Push Operation**

The Tagged Data

- **A.data**
- **B.data**

Pack Reference to Related Things into Execution Function (via Closure)

- `A.data`
- `B.data`

```
lambda: B.data = A.data + 1
```

Push the Operation to Engine

```
engine.push(
  Exec Function,  
  read = [v1],  
  mutate= [v2])
```
Example Scheduling: Data Flow

A = 2   ➡️   engine.push(lambda: A.data=2,
                      read=[], mutate= [A.var])

B = A + 1 ➡️   engine.push(lambda: B.data=A.data+1,
                      read=[A.var], mutate= [B.var])

D = A * B ➡️   engine.push(lambda: D.data=A.data * B.data,
                      read=[A.var, B.var], mutate=[D.var])
Example Scheduling: Memory Recycle

A = 2

\[ \text{engine.push(} \lambda: \text{A.data}=2, \ \
\text{read}=[], \ \text{mutate= [A.var]} \text{)} \]

B = A + 1

\[ \text{engine.push(} \lambda: \text{B.data}=\text{A.data}+1, \ \
\text{read}=[\text{A.var}], \ \text{mutate= [B.var]} \text{)} \]

A.__del__()

\[ \text{engine.push(} \lambda: \text{A.data.__del__()}, \ \
\text{read}=[], \ \text{mutate= [A.var]} \text{)} \]
Example Scheduling: Random Number Generator

\[ B = \text{rnd.uniform}(10, -10) \quad \text{engine.push}(\lambda:\quad \text{B.data} = \text{rnd.gen.uniform}(10, -10), \text{read}=[], \text{mutate}=[\text{rnd.var}]) \]

\[ C = \text{rnd.uniform}(10, -10) \quad \text{engine.push}(\lambda:\quad \text{C.data} = \text{rnd.gen.uniform}(10, -10), \text{read}=[], \text{mutate}=[\text{rnd.var}]) \]
Queue based Implementation of scheduler

- Like scheduling problem in OS
- Maintain a pending operation queue
- Schedule new operations with event update
Enqueue Demonstration

B = A + 1 (reads A, mutates B)
C = A + 2 (reads A, mutates C)
A = C * 2 (reads C, mutates A)
D = A + 3 (reads A, mutates D)

A’s queue: 
B’s queue: 
C’s queue: 
D’s queue:
Enqueue Demonstration

B = A + 1 (reads A, mutates B)

C = A + 2 (reads A, mutates C)

A = C * 2 (reads C, mutates A)

D = A + 3 (reads A, mutates D)

A’s queue:

B’s queue:

C’s queue:

D’s queue:
Enqueue Demonstration

B = A + 1 (reads A, mutates B)

C = A + 2 (reads A, mutates C)

A = C * 2 (reads C, mutates A)

D = A + 3 (reads A, mutates D)

A’s queue: 🟢-green 🟢-green 🟥-red

B’s queue: 🟥-red

C’s queue: 🟥-red 🟢-green

D’s queue:
Enqueue Demonstration

B = A + 1 (reads A, mutates B)
C = A + 2 (reads A, mutates C)
A = C * 2 (reads C, mutates A)
D = A + 3 (reads A, mutates D)

Discuss: What is the update policy of queue when an operation finishes?
Two operations are pushed. Because A and B are ready to write, we decrease the pending counter to 0. The two ops are executed directly.
Two operations are pushed. Because A and B are ready to write, we decrease the pending counter to 0. The two ops are executed directly.
Another two operations are pushed. Because A and B are not ready to read. The pushed operations will be added to the pending queues of variables they wait for.
Another two operations are pushed. Because A and B are not ready to read. The pushed operations will be added to the pending queues of variables they wait for.
A=2 finishes, as a result, the pending reads on A are activated. B=A+B still cannot run because it is still wait for B.
A.del() is a mutate operation. So it need to wait on A until all previous reads on A finishes.

\begin{align*}
B &= A + B \{1\} \\
C &= A + 2
\end{align*}
Update Policy

B=2 finishes running. B=A+B is able to run because all its dependencies are satisfied. A.del() still need to wait for B=A+B to finish for A to turn green.
Update Policy

B=2 finishes running. B=A+B is able to run because all its dependencies are satisfied. A.del() still need to wait for B=A+B to finish for A to turn green.
Take aways

- Automatic scheduling makes parallelization easier
- Mutation aware interface to handle resource contention
- Queue based scheduling algorithm