Lecture 10: Parallel Scheduling

CSE599W: Spring 2018
NOTE

● Office hour CSE 220 2:30pm - 3:30pm

● No class on next Thursday (OSDI)
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
The Gap for Communication

Which operations can run in currently with synchronization of g2/w2?
Parallel Program 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

```
A = 2
C = A + 2
B = A + 1
D = B * C
```
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

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

Code

\[
\begin{align*}
A &= 2 \\
B &= A + 1 \\
C &= A + 2 \\
A &= 3
\end{align*}
\]

Dependency

\[
\begin{align*}
A &= 2 \\
C &= A + 2 \\
B &= A + 1 \\
A &= 3
\end{align*}
\]
Memory Recycle

Code

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

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)`
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**
\[
\text{tf.assign}(A, B + 1) \\
\text{tf.assign}(T, B + 2) \\
\text{tf.assign}(B, 2)
\]

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

A **mutation aware** scheduler can solve these problems much easier than DAG based scheduler.
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>A.var = engine.new_variable()</td>
<td>A.data</td>
<td>A.data v1</td>
</tr>
<tr>
<td>B.var = engine.new_variable()</td>
<td>B.data</td>
<td>B.data v2</td>
</tr>
<tr>
<td>C.var = engine.new_variable()</td>
<td>C.data</td>
<td>C.data v3</td>
</tr>
<tr>
<td>rnd.var = engine.new_variable()</td>
<td>rnd.gen</td>
<td>rnd.gen v4</td>
</tr>
</tbody>
</table>
Mutation aware Scheduler: Push Operation

The Tagged Data

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

lambda: B.data = A.data + 1

Push the Operation to Engine

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

A = 2

\[ A = 2 \]

\[ B = A + 1 \]

\[ B = 2 + 1 = 3 \]

\[ D = A \times B \]

\[ D = 2 \times 3 = 6 \]

\[ D = 6 \]

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

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

\[ engine.push(lambda: D.data=A.data \times B.data, \]
\[ \quad \text{read}=[A.var, B.var] \quad \text{mutate}=[D.var]) \]
Example Scheduling: Memory Recycle

A = 2

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

B = A + 1

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

A.__del__()

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

\[ B = \text{rnd.uniform}(10, -10) \]  
\[ C = \text{rnd.uniform}(10, -10) \]

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

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

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

\[ B = A + 1 \text{ (reads } A, \text{ mutates } B) \]

\[ C = A + 2 \text{ (reads } A, \text{ mutates } C) \]

\[ A = C \times 2 \text{ (reads } C, \text{ mutates } A) \]

\[ D = A + 3 \text{ (reads } A, \text{ mutates } D) \]

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: 🟠 🟠 🟥

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)

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.

```
operation {wait counter} operation and the number of pending dependencies it need to wait for
var ready to read and mutate
var ready to read, but still have uncompleted reads. Cannot mutate
var still have uncompleted mutations. Cannot read/write
```
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.
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