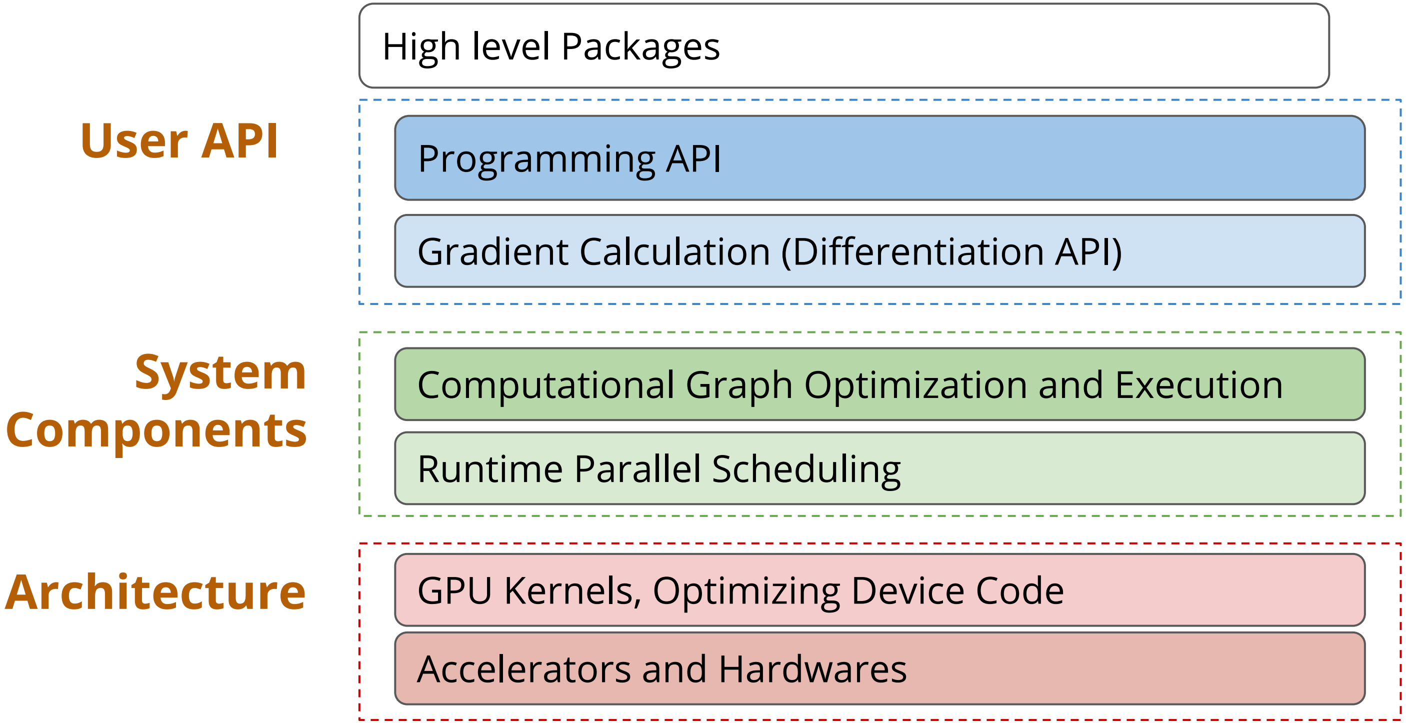
Lecture 9: Scheduling

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

User API

Programming API

Gradient Calculation (Differentiation API)

System Components

Computational Graph Optimization and Execution

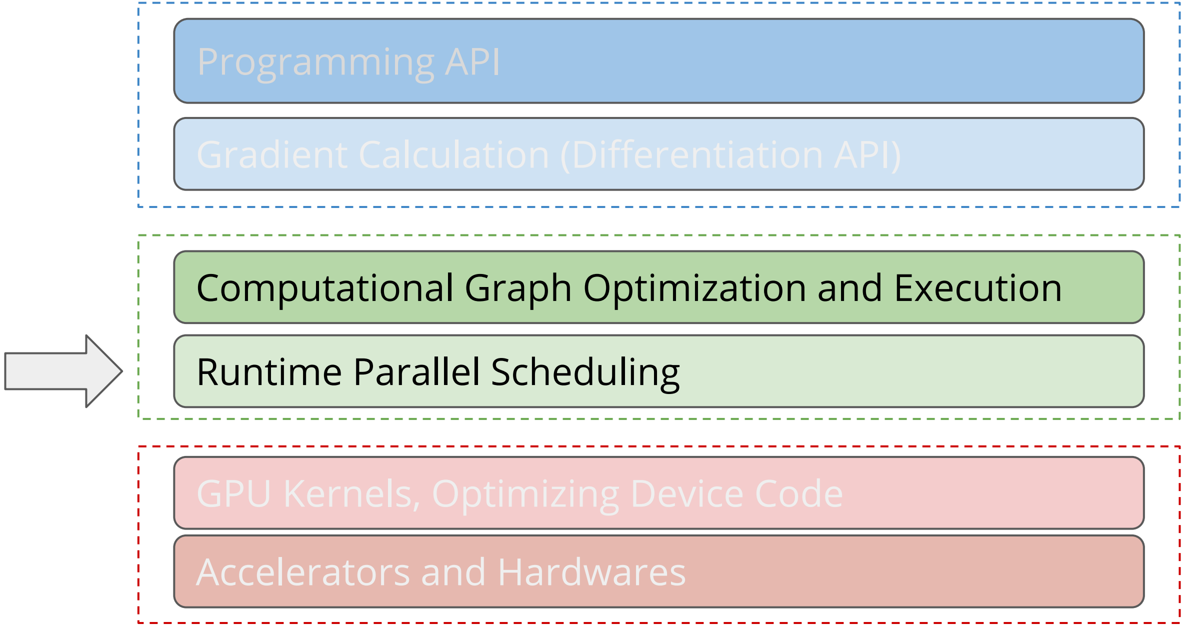
Runtime Parallel Scheduling

Architecture

GPU Kernels, Optimizing Device Code

Accelerators and Hardwares

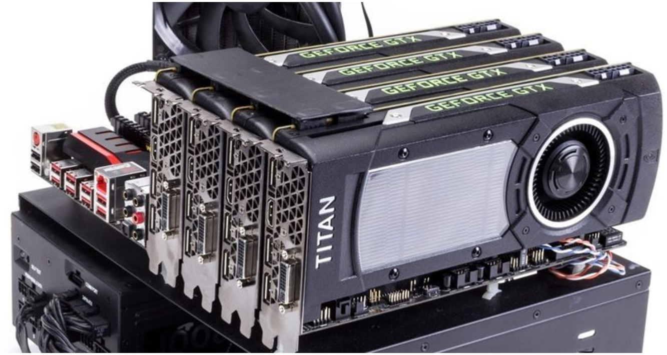
Where are we

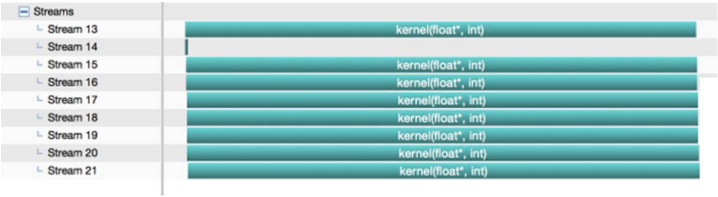

Programming API
Gradient Calculation (Differentiation API)
Computational Graph Optimization and Execution
Runtime Parallel Scheduling
GPU Kernels, Optimizing Device Code
Accelerators and Hardwares

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

Parallelization Problem

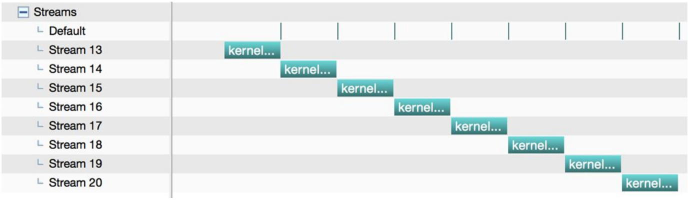
- Parallel execution of concurrent kernels
- Overlap compute and data transfer





```
Streams
  Stream 13   kernel(float*, int)
  Stream 14
  Stream 15   kernel(float*, int)
  Stream 16   kernel(float*, int)
  Stream 17   kernel(float*, int)
  Stream 18   kernel(float*, int)
  Stream 19   kernel(float*, int)
  Stream 20   kernel(float*, int)
  Stream 21   kernel(float*, int)
```

👍 Parallel over multiple streams

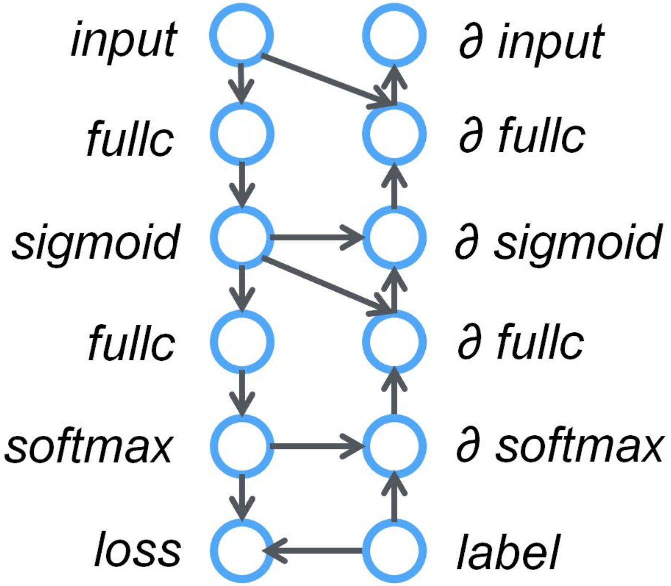


```
Streams
  Default
  Stream 13   kernel...
  Stream 14   kernel...
  Stream 15   kernel...
  Stream 16   kernel...
  Stream 17   kernel...
  Stream 18   kernel...
  Stream 19   kernel...
  Stream 20   kernel...
```

👎 Serial execution

Recap: Deep Learning Training Workflow

Gradient Calculation



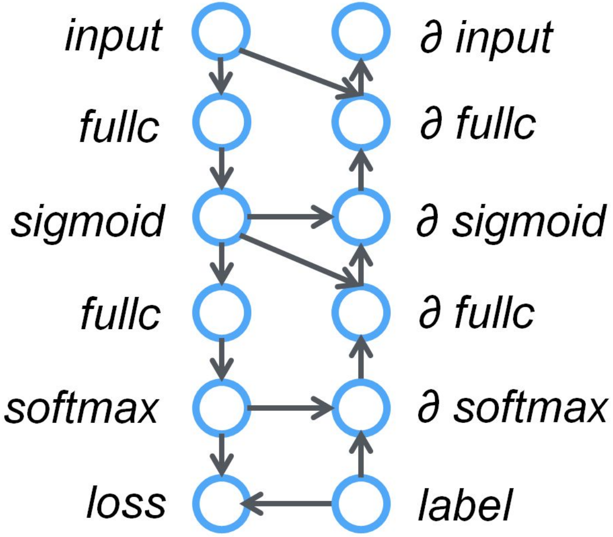
Interactions with Model

Parameter Update

$$w = w - \eta \, \partial f(w)$$

Questions to be answered

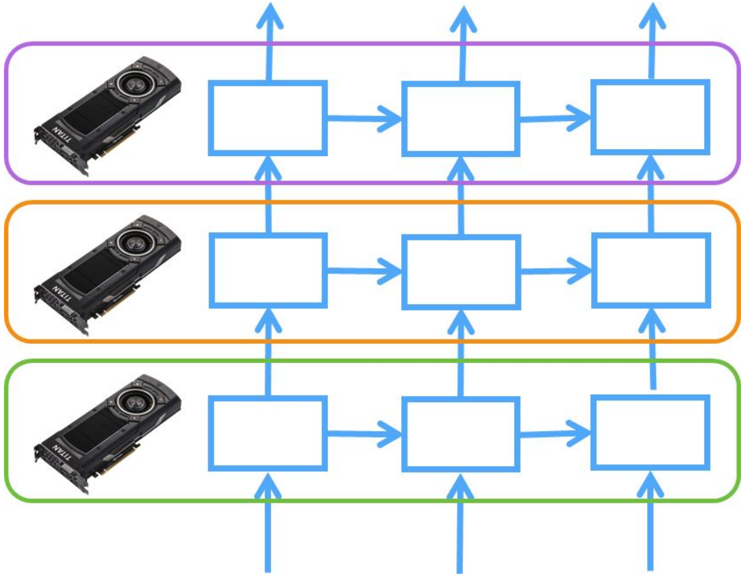
- What are common patterns of parallelization
- How can we easily achieve these patterns
- What about dynamic style program



Model Parallel Training

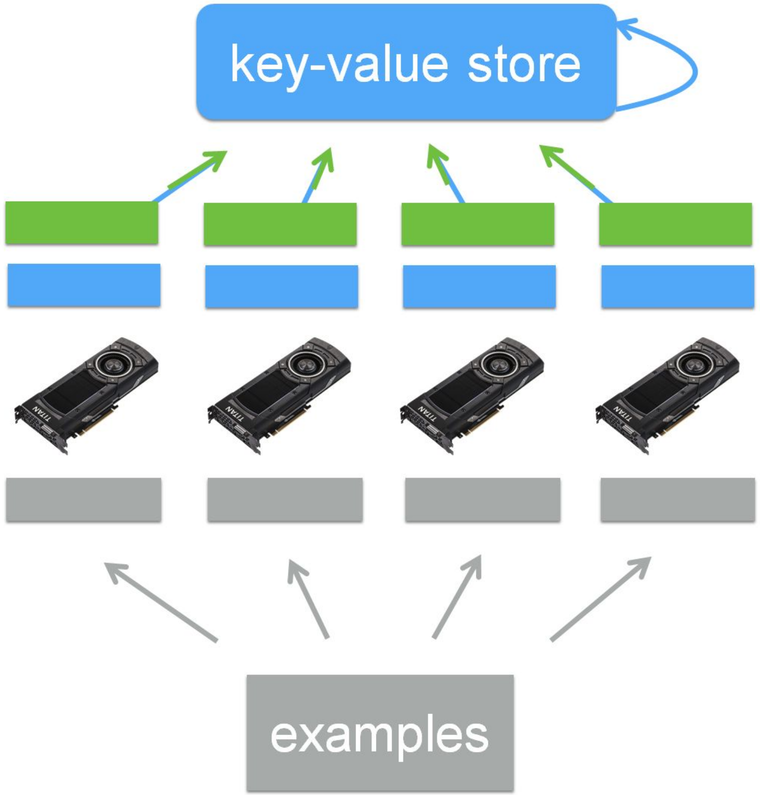
- Map parts of workload to different devices

- Require special dependency patterns (wave style)
  - e.g. LSTM

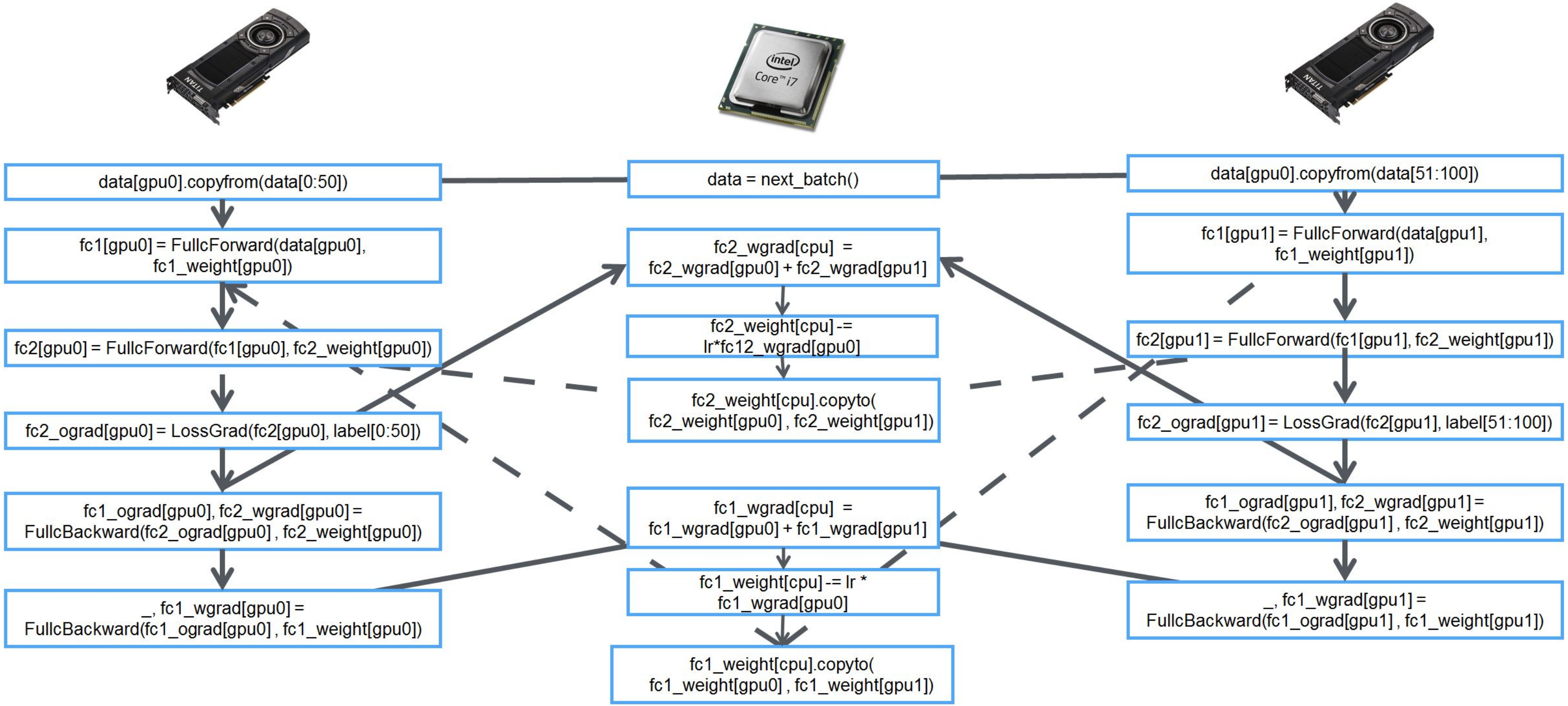


Data Parallelism

- Train replicated version of model in each machine
- Synchronize the gradient



Data Parallel Training

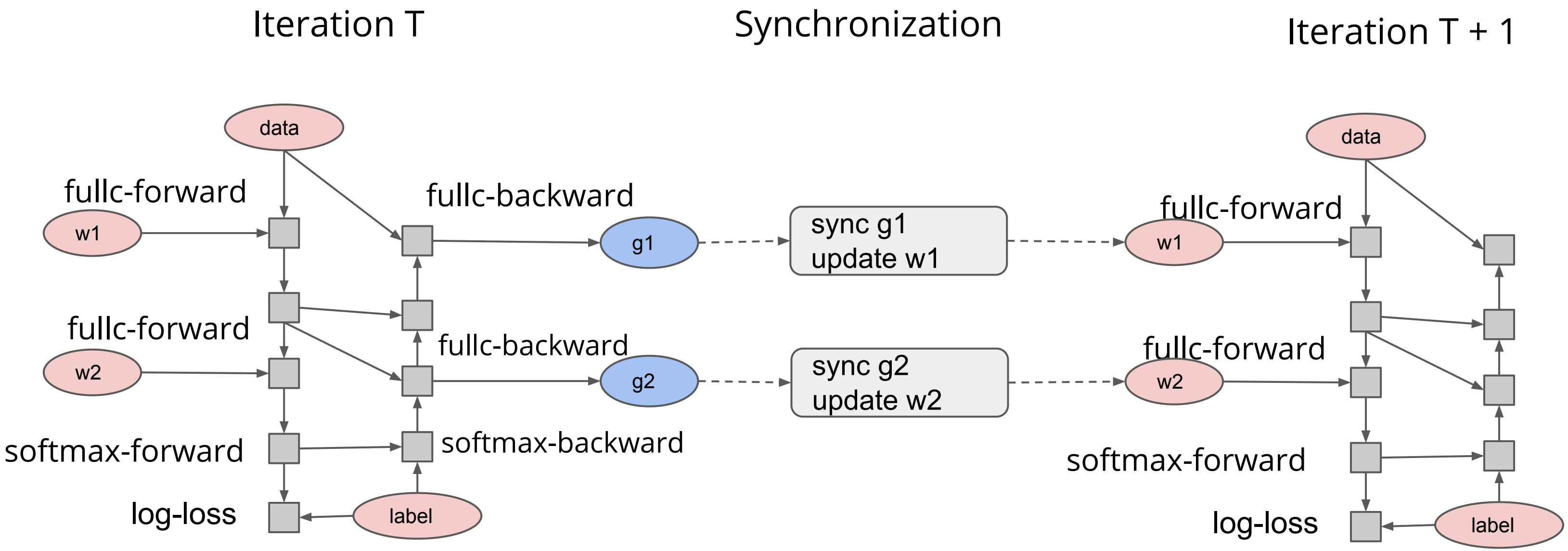


```
data[gpu0].copyfrom(data[0:50])

fc1[gpu0] = FullcForward(data[gpu0],
fc1_weight[gpu0])

fc2[gpu0] = FullcForward(fc1[gpu0], fc2_weight[gpu0])

fc2_ograd[gpu0] = LossGrad(fc2[gpu0], label[0:50])

fc1_ograd[gpu0], fc2_wgrad[gpu0] =
FullcBackward(fc2_ograd[gpu0] , fc2_weight[gpu0])

_, fc1_wgrad[gpu0] =
FullcBackward(fc1_ograd[gpu0] , fc1_weight[gpu0])
```

```
data = next_batch()

fc2_wgrad[cpu] =
fc2_wgrad[gpu0] + fc2_wgrad[gpu1]

fc2_weight[cpu] -=
lr*fc12_wgrad[gpu0]

fc2_weight[cpu].copyto(
fc2_weight[gpu0] , fc2_weight[gpu1])

fc1_wgrad[cpu] =
fc1_wgrad[gpu0] + fc1_wgrad[gpu1]

fc1_weight[cpu] -= lr *
fc1_wgrad[gpu0]

fc1_weight[cpu].copyto(
fc1_weight[gpu0] , fc1_weight[gpu1])
```

```
data[gpu0].copyfrom(data[51:100])

fc1[gpu1] = FullcForward(data[gpu1],
fc1_weight[gpu1])

fc2[gpu1] = FullcForward(fc1[gpu1], fc2_weight[gpu1])

fc2_ograd[gpu1] = LossGrad(fc2[gpu1], label[51:100])

fc1_ograd[gpu1], fc2_wgrad[gpu1] =
FullcBackward(fc2_ograd[gpu1] , fc2_weight[gpu1])

_, fc1_wgrad[gpu1] =
FullcBackward(fc1_ograd[gpu1] , fc1_weight[gpu1])
```

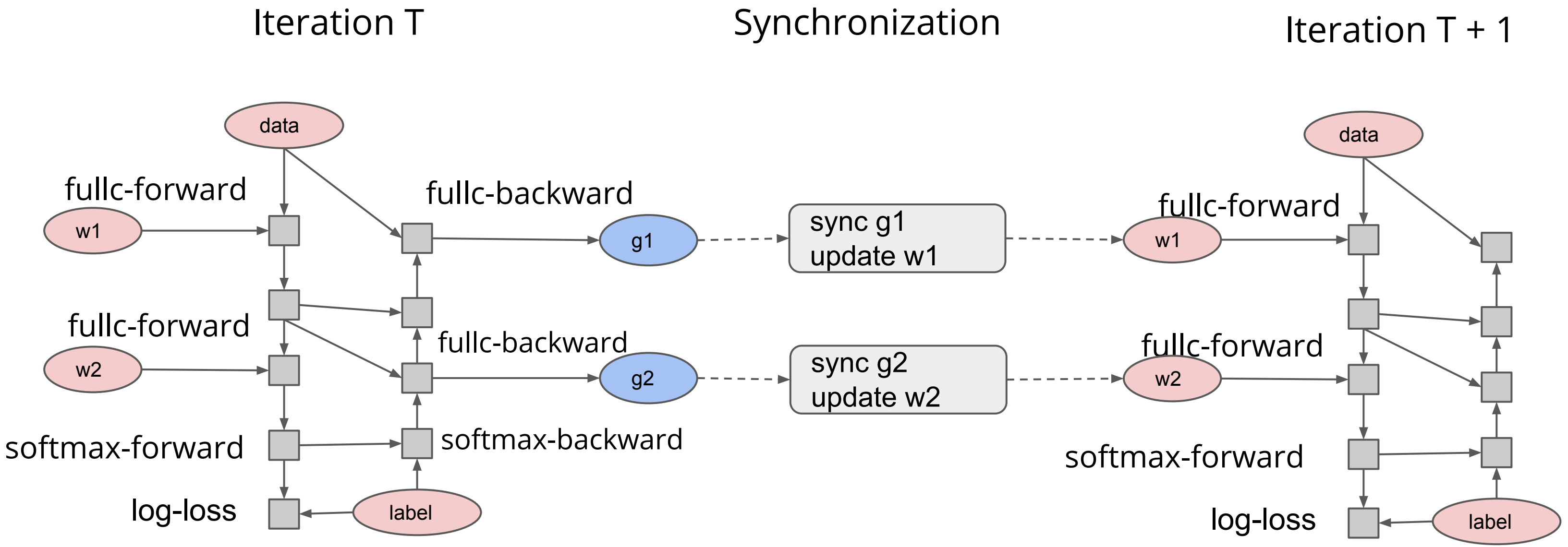
The Gap for Communication



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

Parallel Program are Hard to Write

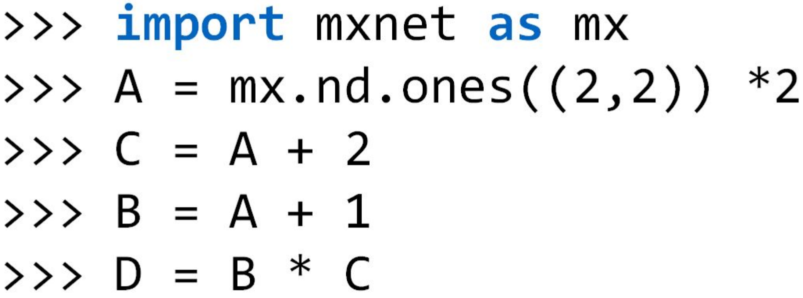
We need a automatic scheduler



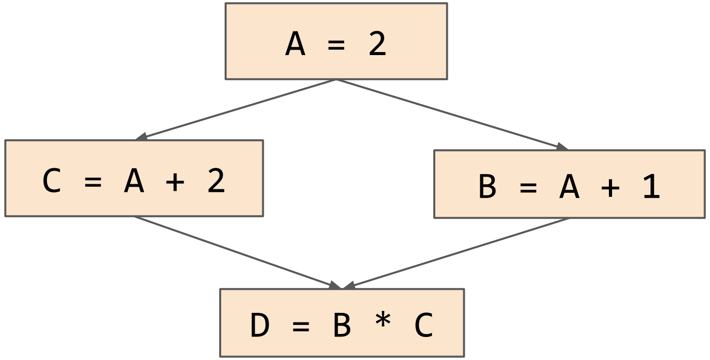
Goal of Scheduler Interface

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

- Run in Parallel
- Respect serial execution order

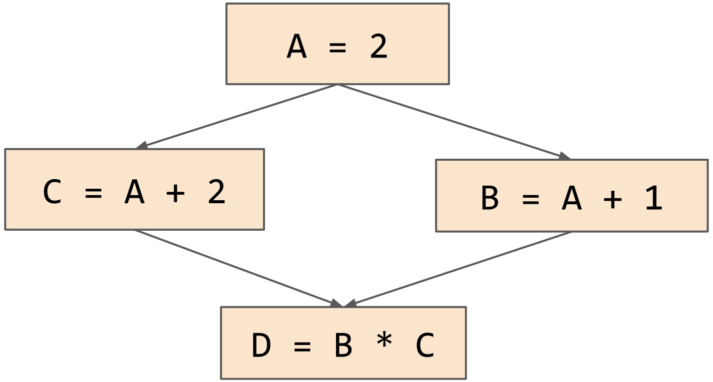


```
>>> import mxnet as mx
>>> A = mx.nd.ones((2,2)) *2
>>> C = A + 2
>>> B = A + 1
>>> D = B * C
```



Discussion: How to schedule the following ops

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

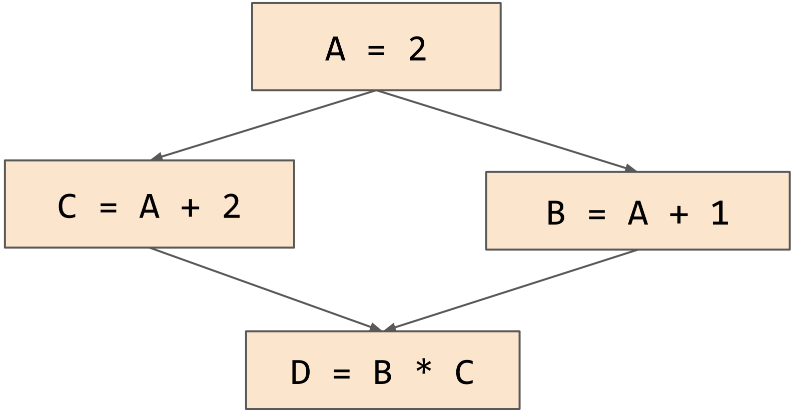


Data Flow Dependency

Code

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

Dependency

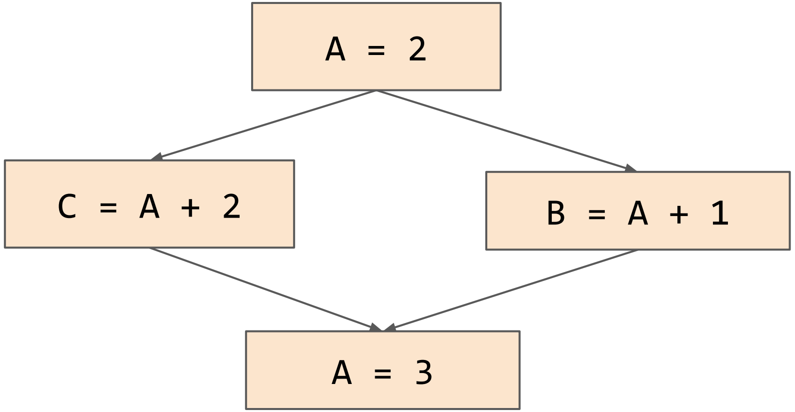


Write After Read Mutation

Code

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

Dependency



A = 2

C = A + 2

B = A + 1

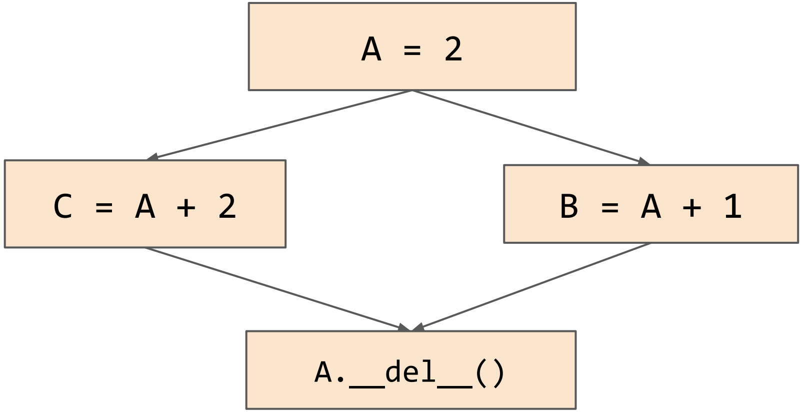
A = 3

Memory Recycle

Code

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

A.__del__()
```

Dependency

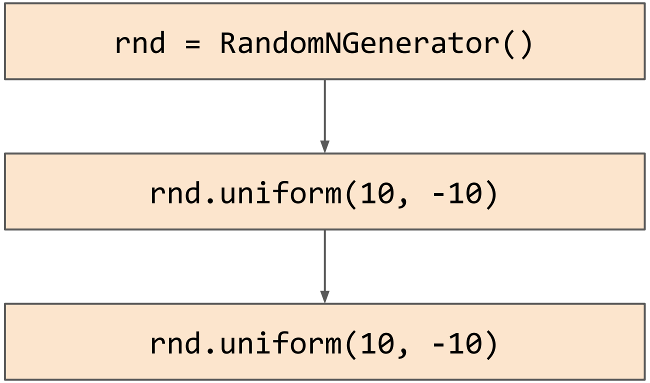


Random Number Generator

Code

```
rnd = RandomNGenerator()

B = rnd.uniform(10, -10)

C = rnd.uniform(10, -10)
```

Dependency



Goal of Scheduler Interface

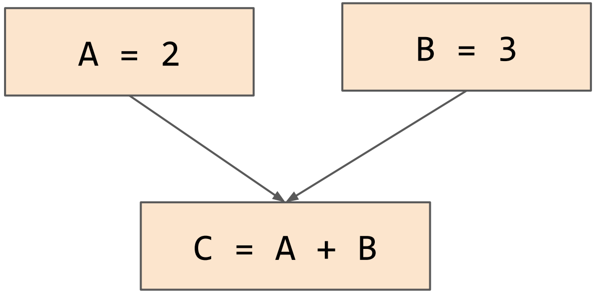
- Schedule any resources
  - Data
  - Random number generator
  - Network communicator

- 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

```
tf.assign(A, B + 1)
tf.assign(T, B + 2)
tf.assign(B, 2)
```

Read after Write

```
T = tf.assign(B, B + 1)
tf.assign(A, B + 2)
```

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

MXNet Program for Data Parallel Training

```
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])
```

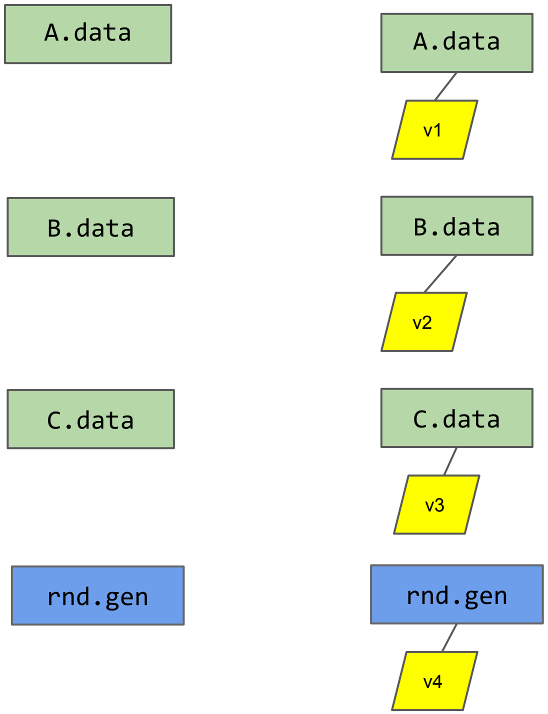
Mutation aware Scheduler: Tag each Resource

Code

```
A.var = engine.new_variable()

B.var = engine.new_variable()

C.var = engine.new_variable()

rnd.var = engine.new_variable()
```

Original Resources

Tagged Resources

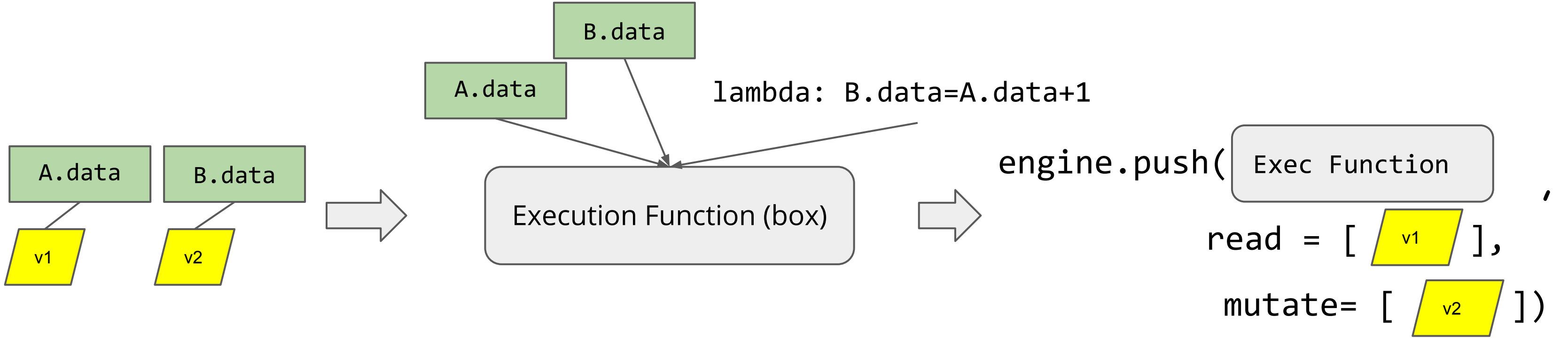


Mutation aware Scheduler: Push Operation

The Tagged Data

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

Push the Operation to Engine



A.data

B.data

v1

v2

B.data

A.data

lambda: B.data=A.data+1

Execution Function (box)

```
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          ⇨   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])
```

```
A.__del__()    ⇨   engine.push(lambda: A.data._del__(),
                               read=[], mutate= [A.var])
```

Example Scheduling: Random Number Generator

```
B = rnd.uniform(10, -10)
```

⇨

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

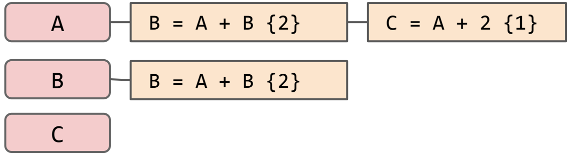
```
C = rnd.uniform(10, -10)
```

⇨

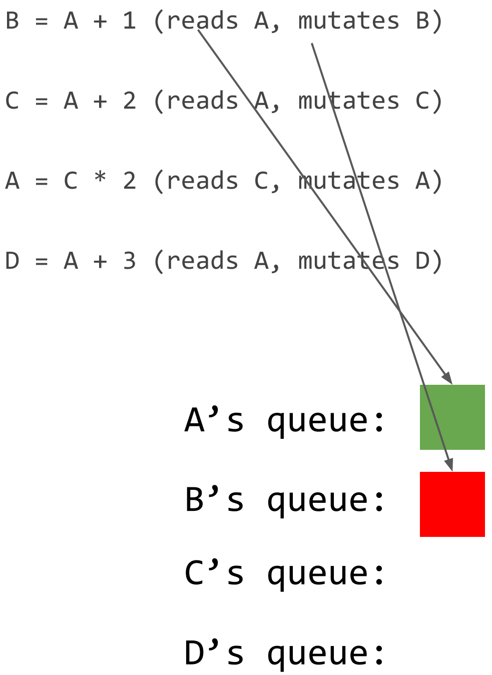
```
engine.push(lambda:
               C.data = rnd.gen.uniform(10,-10),
            read=[], mutate= [rnd.var])
```

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 (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)
```

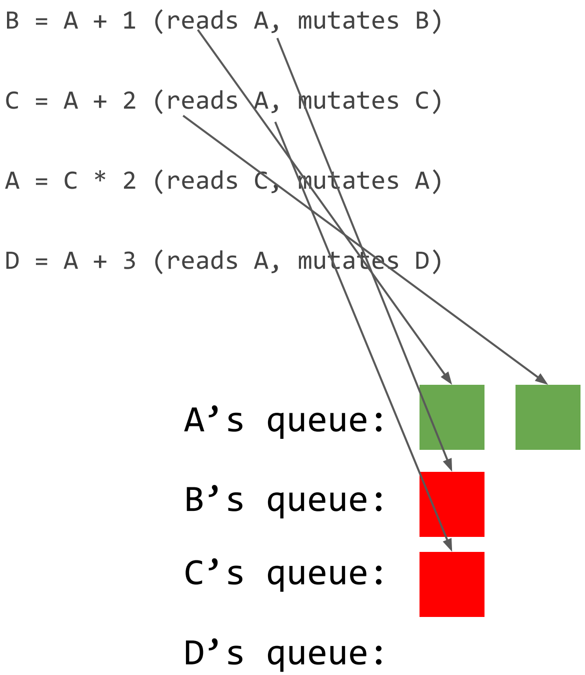
A’s queue: 🟩

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: 🟩 🟩

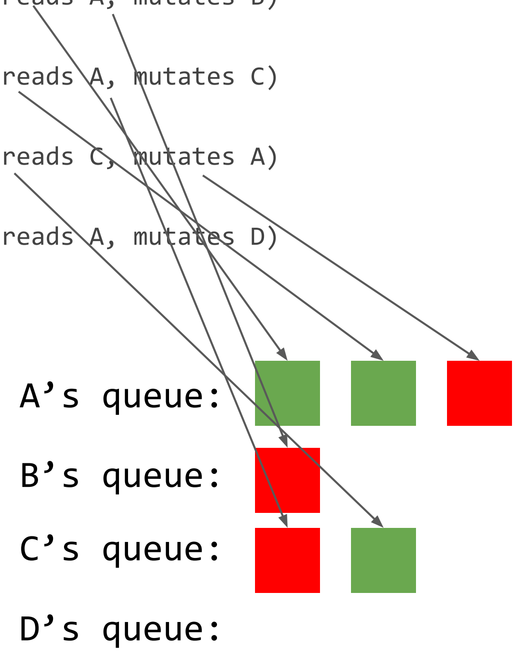
B’s queue: 🟥

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: 🟩 🟩 🟥

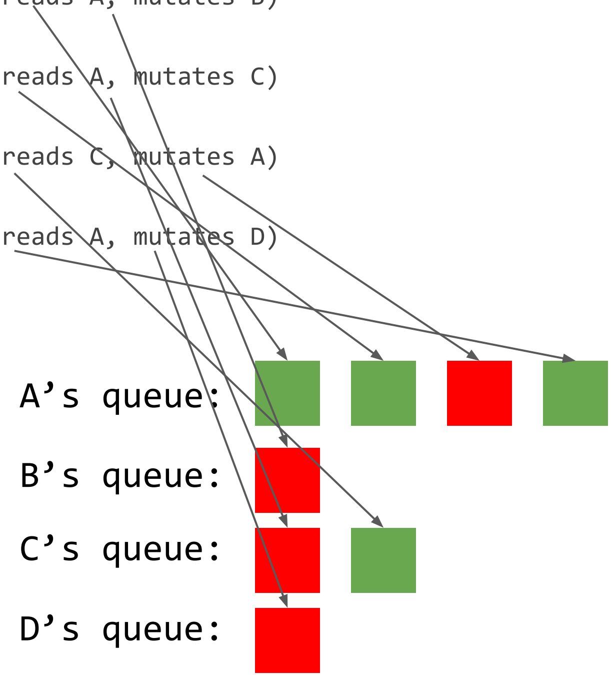
B’s queue: 🟥

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: 🟩 🟩 🟥 🟩

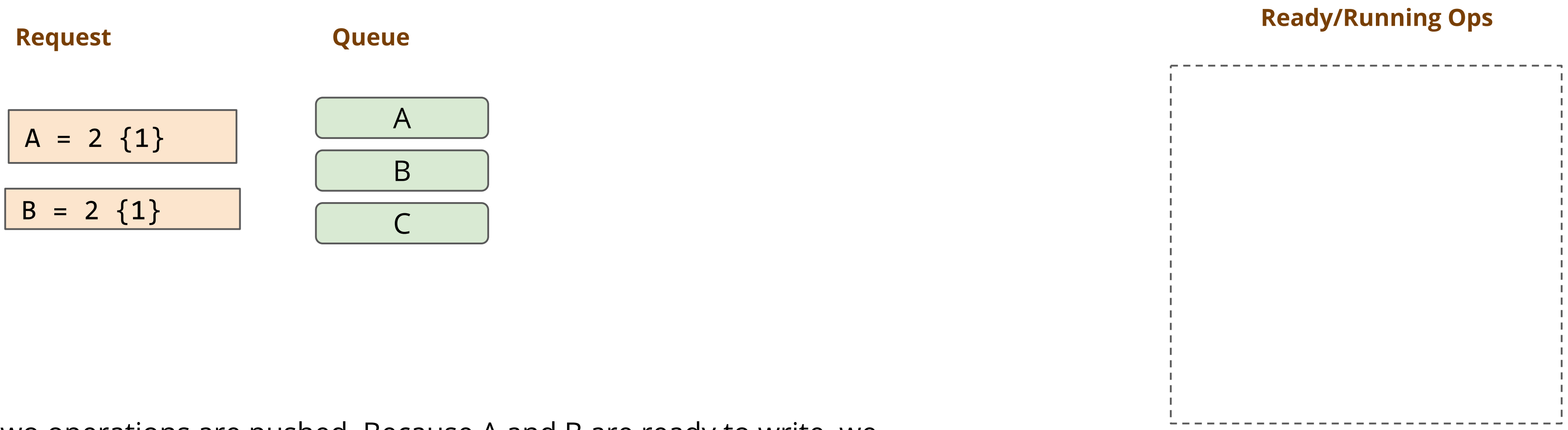
B’s queue: 🟥

C’s queue: 🟥 🟩

D’s queue: 🟥

Discuss: What is the update policy of queue when an operation finishes?

Update Policy



Request

A = 2 {1}

B = 2 {1}

Queue

A

B

C

Ready/Running Ops

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.

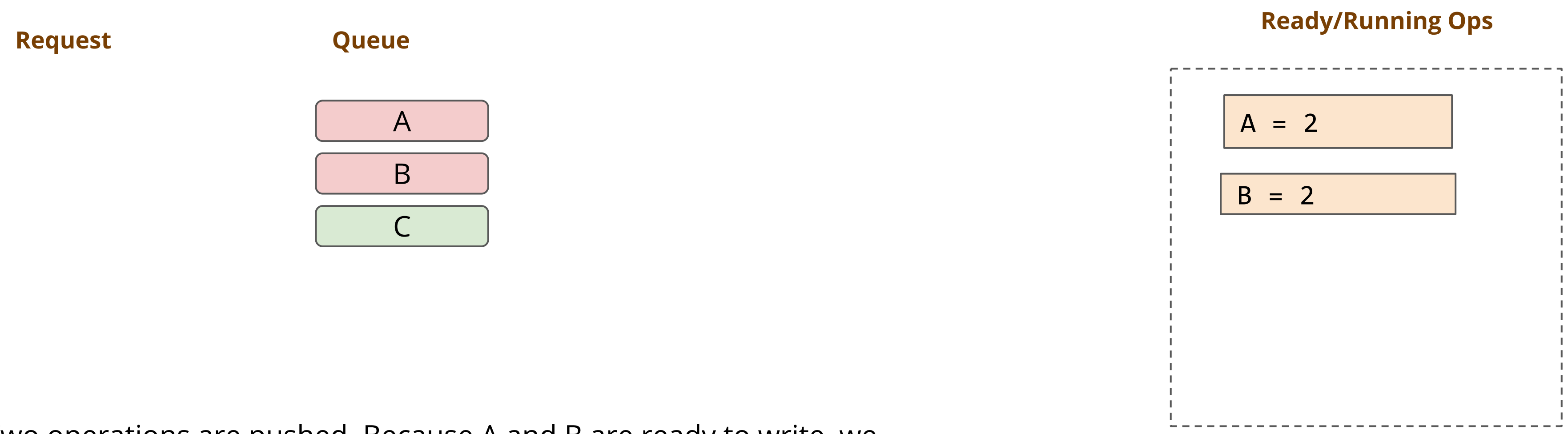
`operation {wait counter}` operation and the number of pending dependencies it need to wait for

🟩 `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

Update Policy



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.

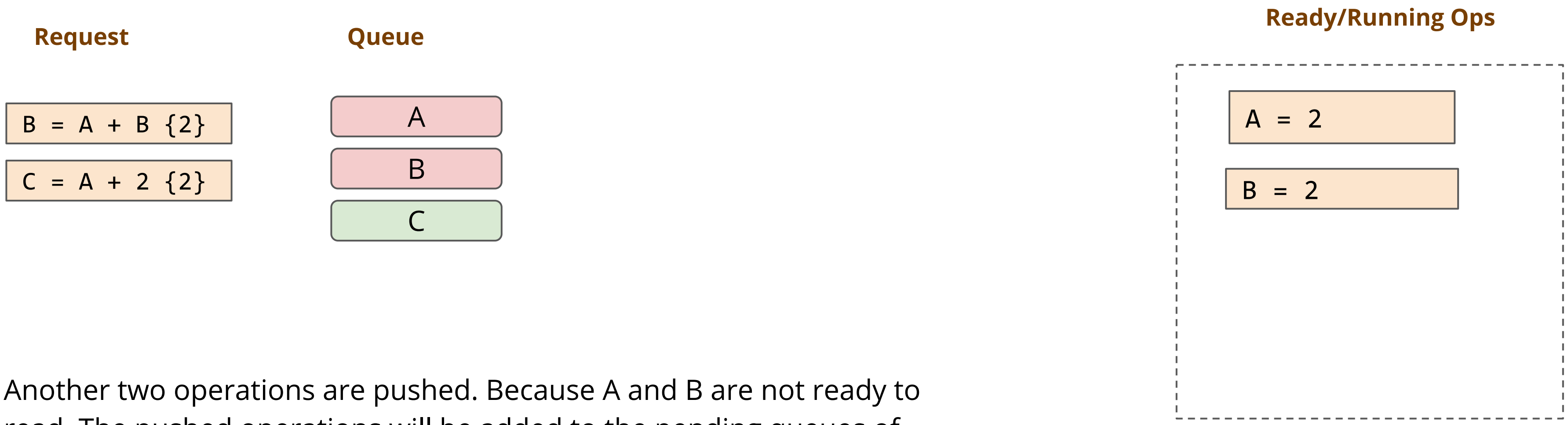
`operation {wait counter}`
operation and the number of pending dependencies it need to wait for

🟩 `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

Update Policy



Request

B = A + B {2}

C = A + 2 {2}

Queue

A

B

C

Ready/Running Ops

A = 2

B = 2

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.

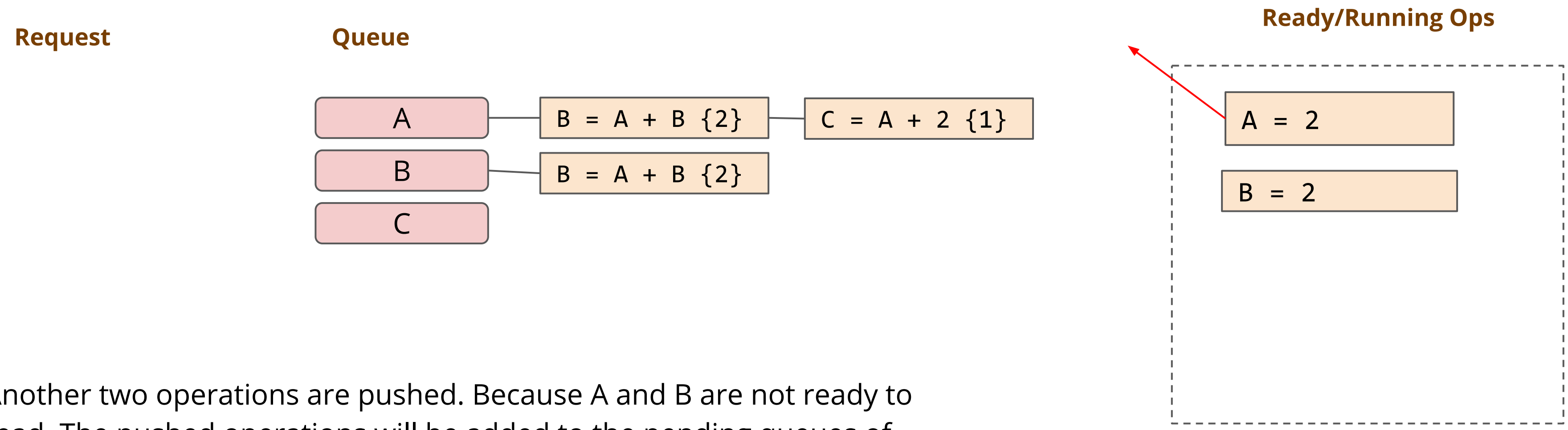
operation {wait counter}
operation and the number of pending dependencies it need to wait for

🟩 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

Update Policy



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.

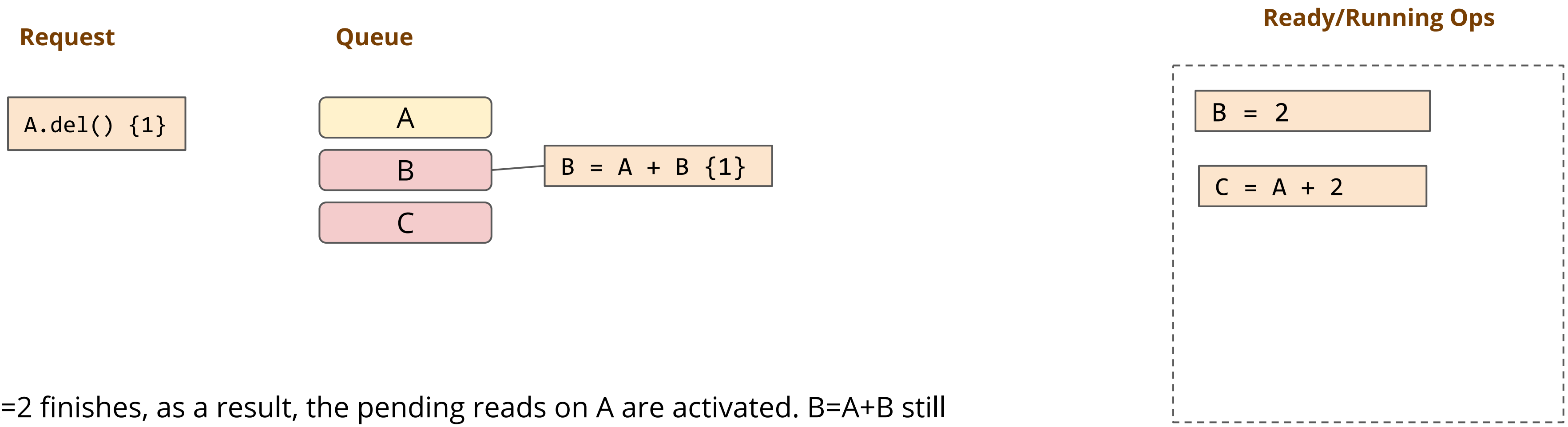
operation {wait counter}: operation and the number of pending dependencies it need to wait for

🟩 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

Update Policy



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.

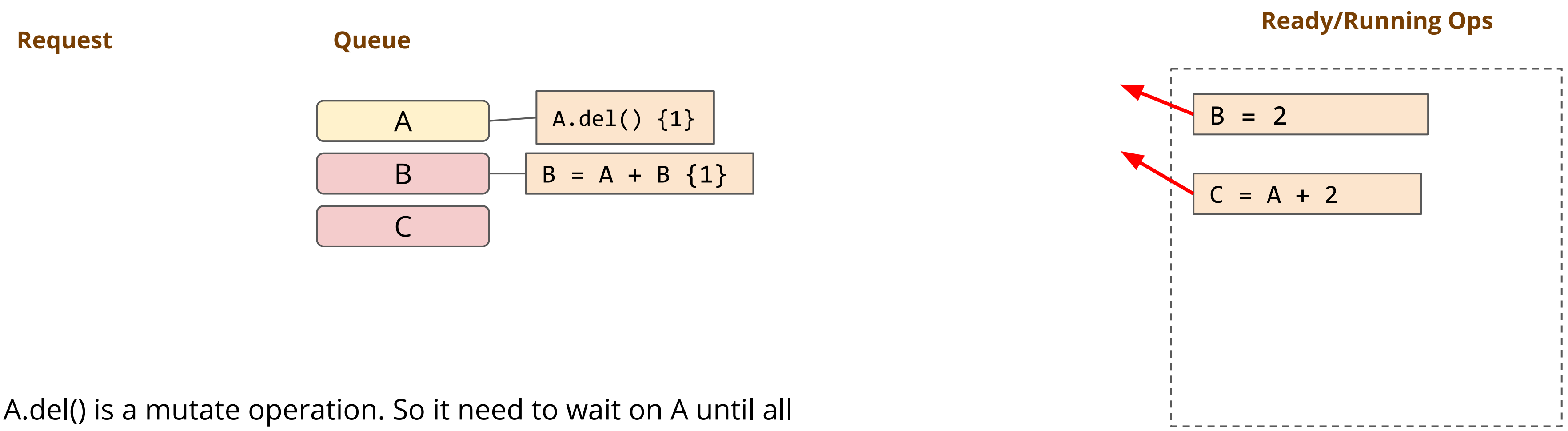
`operation {wait counter}` operation and the number of pending dependencies it need to wait for

🟩 `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

Update Policy



A.del() is a mutate operation. So it need to wait on A until all previous reads on A finishes.

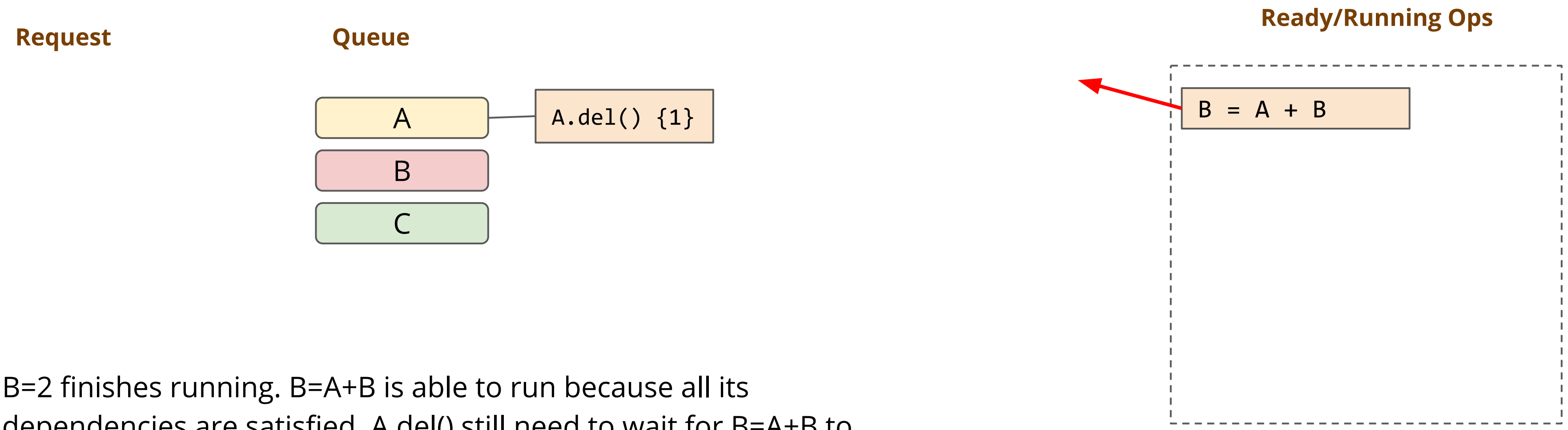
`operation {wait counter}` operation and the number of pending dependencies it need to wait for

🟩 `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

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

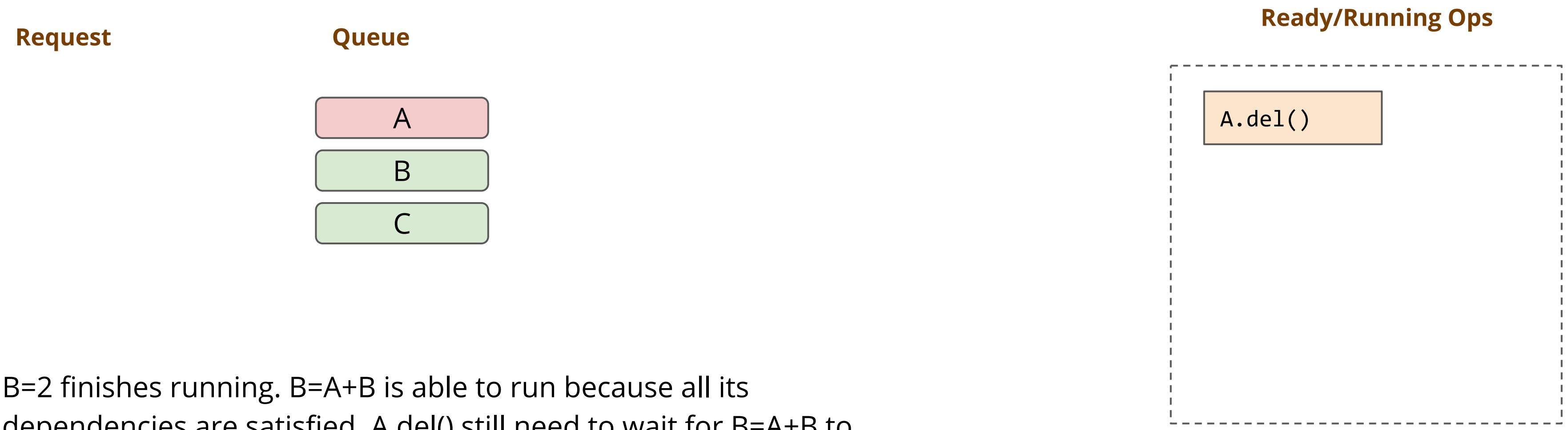
`operation {wait counter}` operation and the number of pending dependencies it need to wait for

🟩 `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

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

`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

Take aways

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