Forward Progress on GPU Concurrency

Alastair Donaldson, Imperial College London

Slides and accompanying code prepared jointly with Tyler Sorensen

Accompanying paper written jointly with Jeroen Ketema, Tyler Sorensen and John Wickerson

Agenda

- Overview of key concepts in OpenCL – a major GPU programming model
- Implementing a *reduction* in OpenCL, showcasing pitfalls
- Demo of GPUVerify – static data race detection for OpenCL
- Discussion of how to achieve global synchronization in OpenCL
- Problems with global synchronization due to unfair scheduling
- Discovery protocol to enable portable global synchronization

Threads and blocks

An OpenCL kernel is executed by a set of *threads*

The threads are sub-divided into *workgroups* of equal size

Example: 3 blocks, 4 threads per block

global id:	0	1	2	3	4	5	6	7	8	9	10	11
local id:	0	1	2	3	0	1	2	3	0	1	2	3
group id:	0	0	0	0	1	1	1	1	2	2	2	2

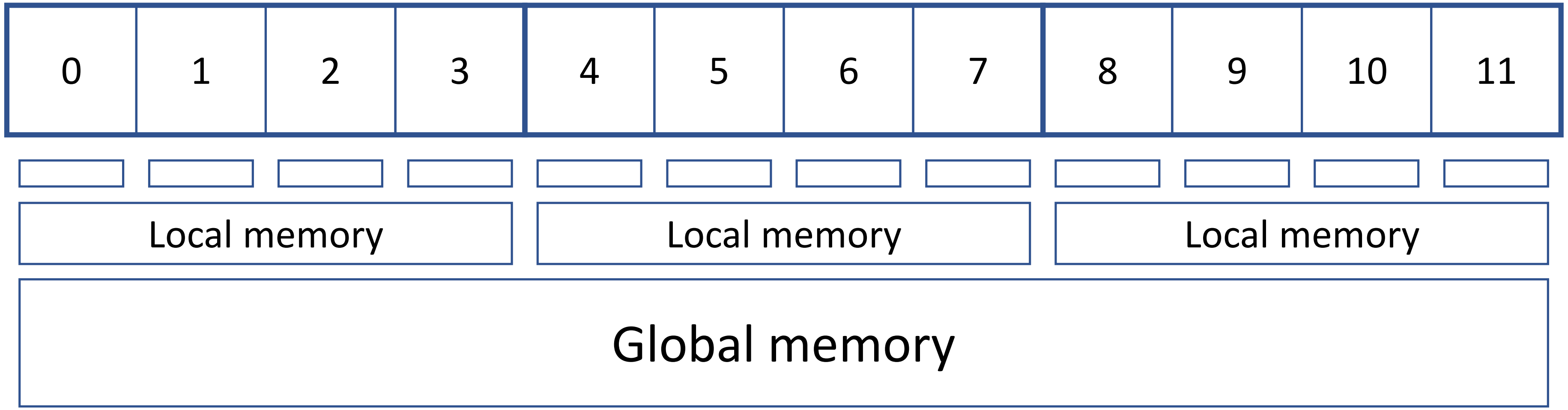
global id = group size x **group id + local id**

Memory

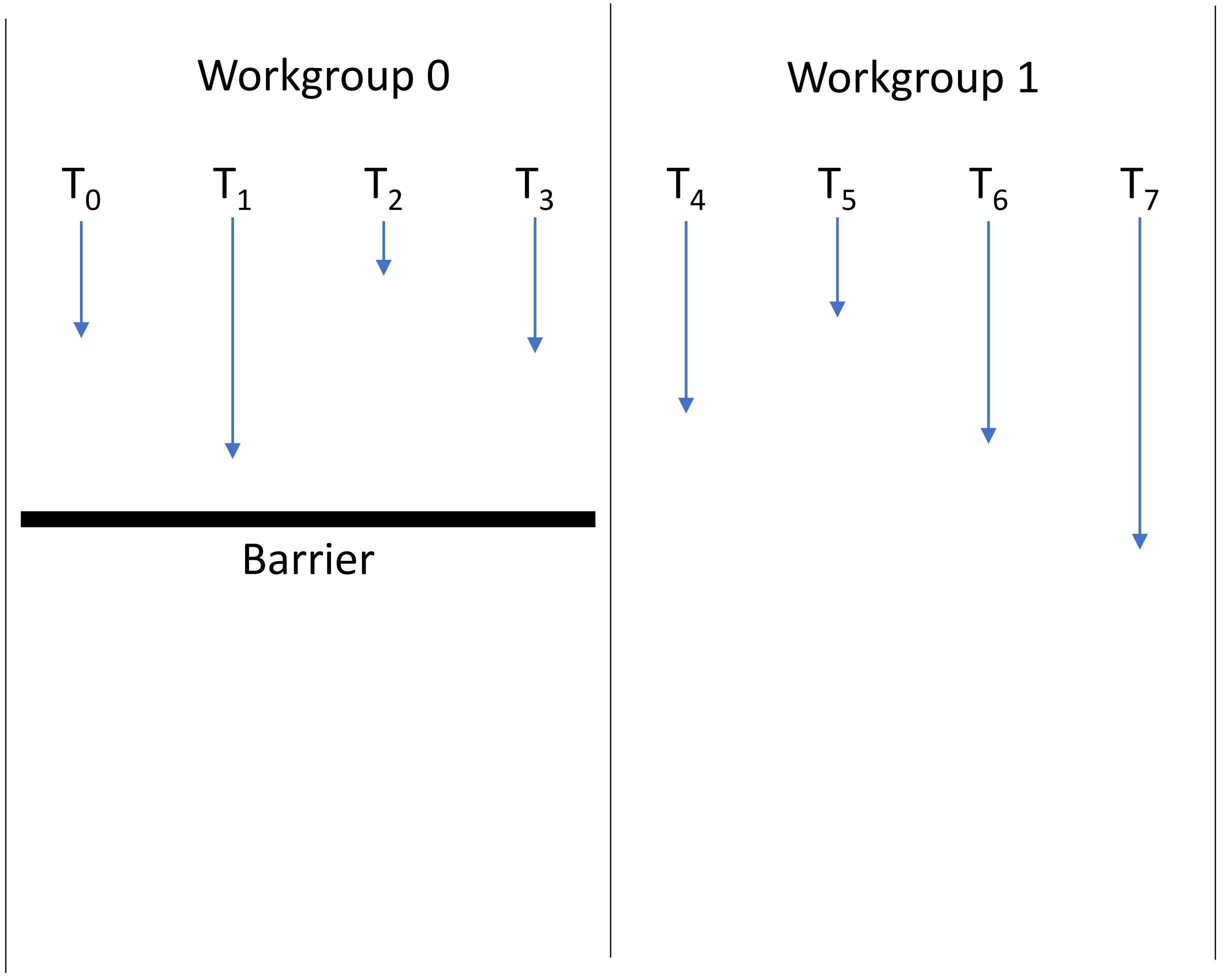
Each thread has access to its own *private* memory

Threads in a workgroup share *local* memory

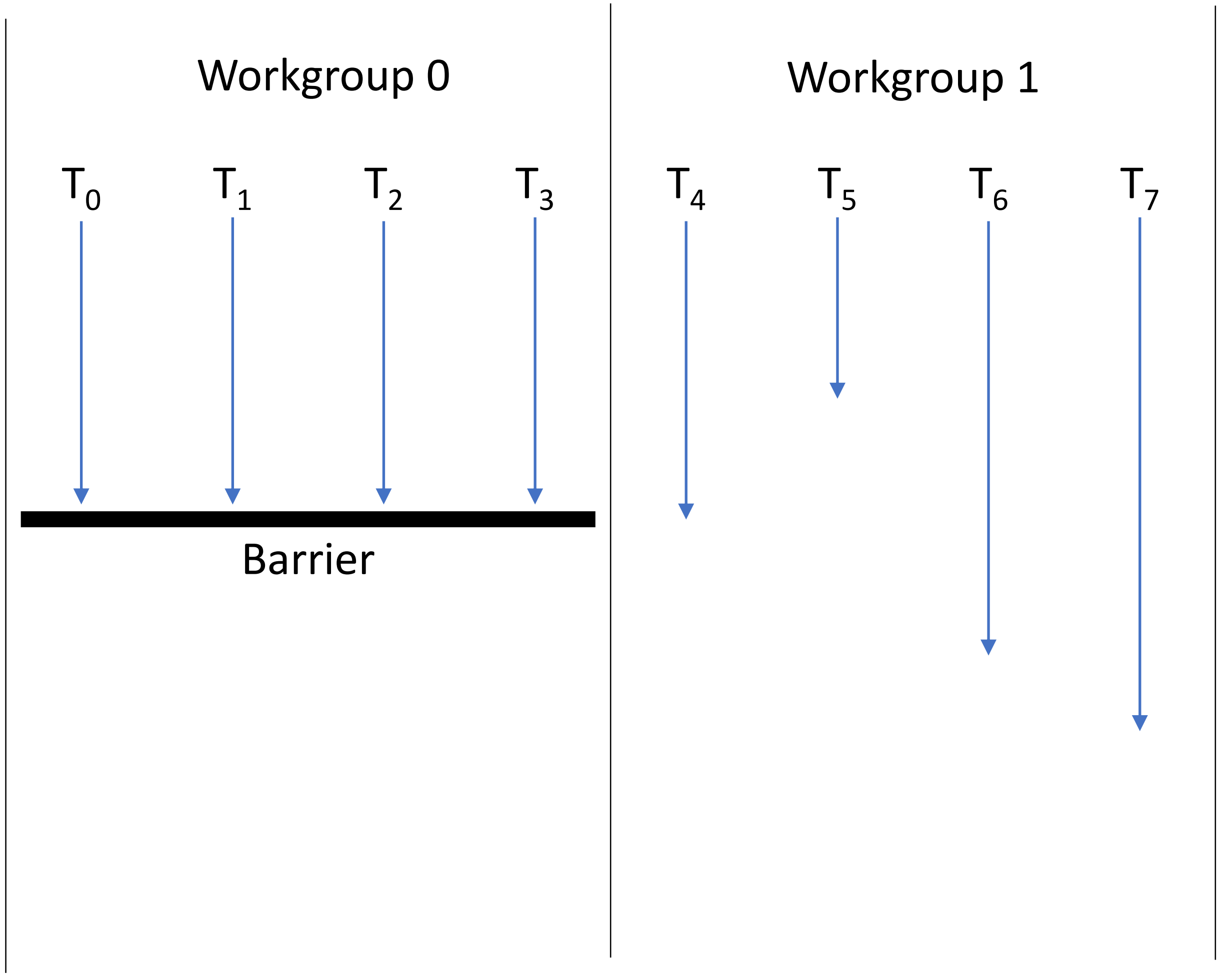
All threads share *global* memory



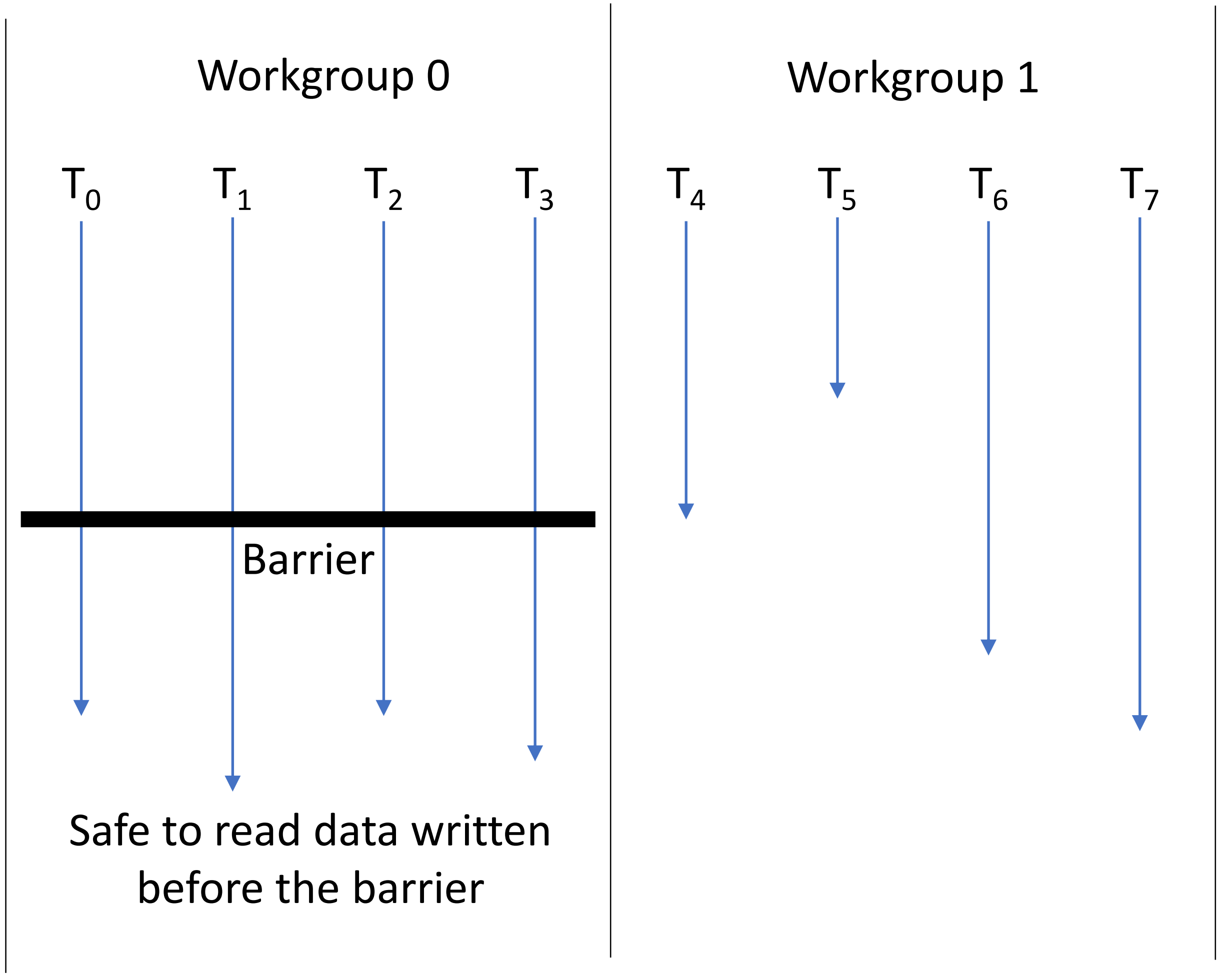
Barrier synchronization



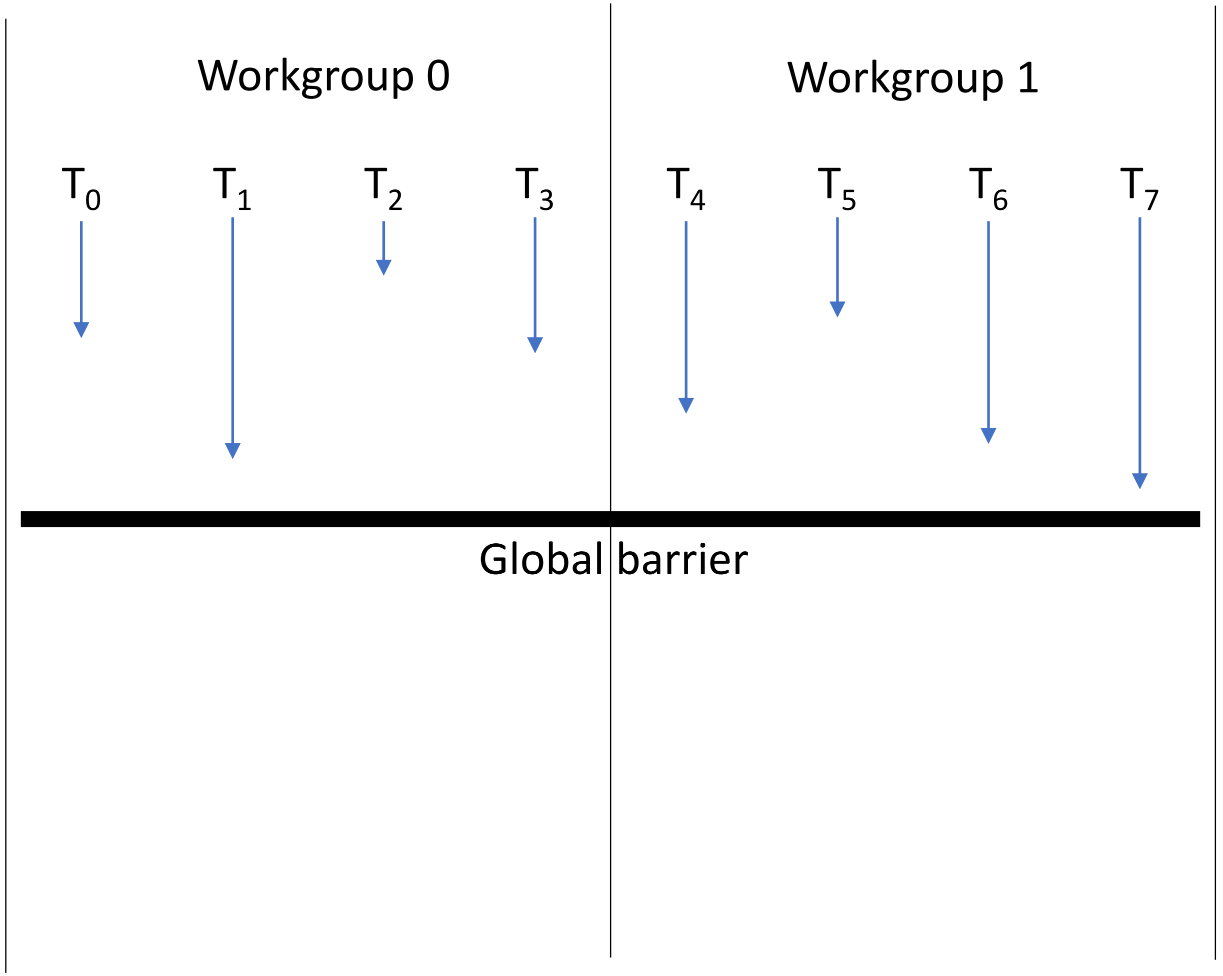
Barrier synchronization



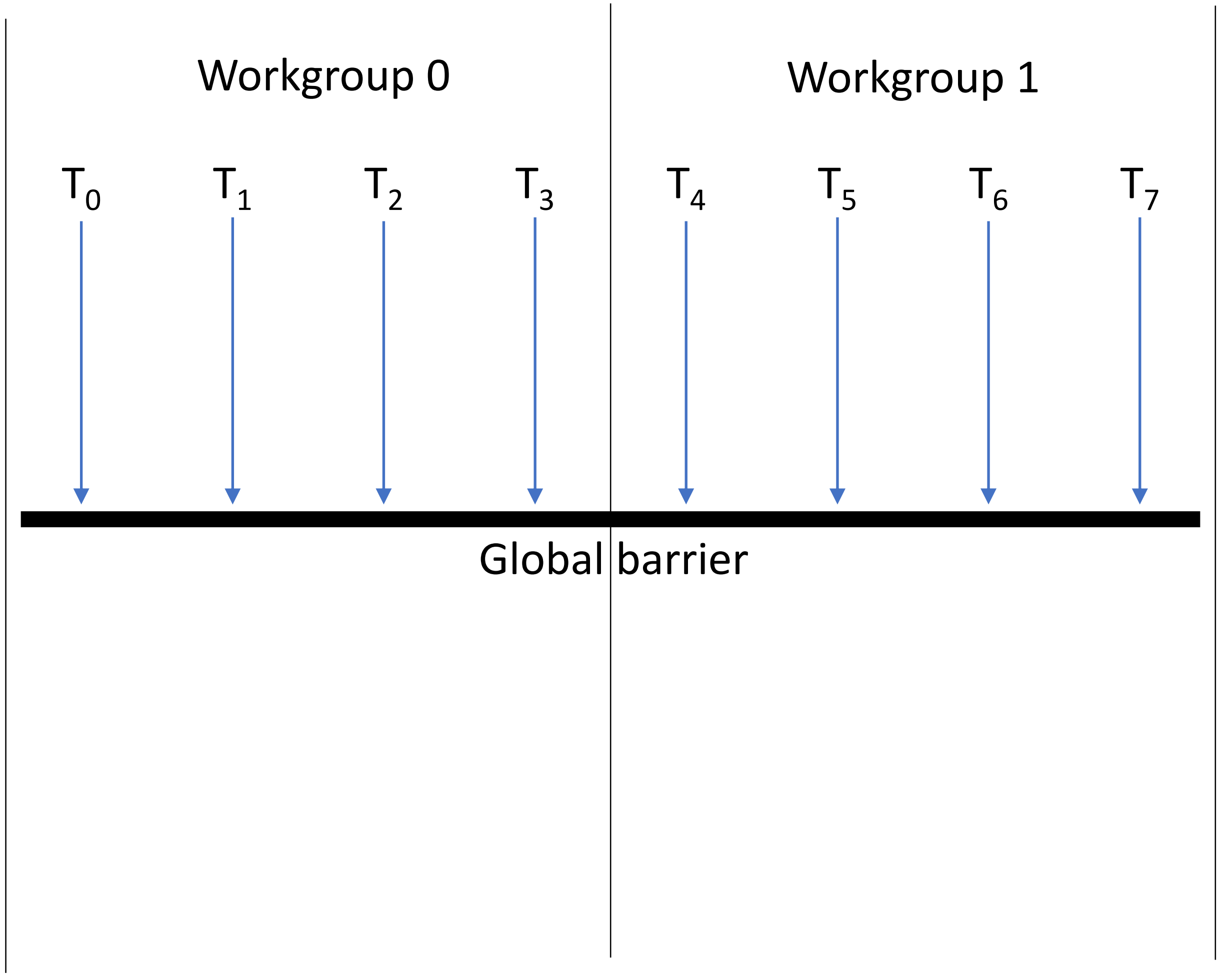
Barrier synchronization



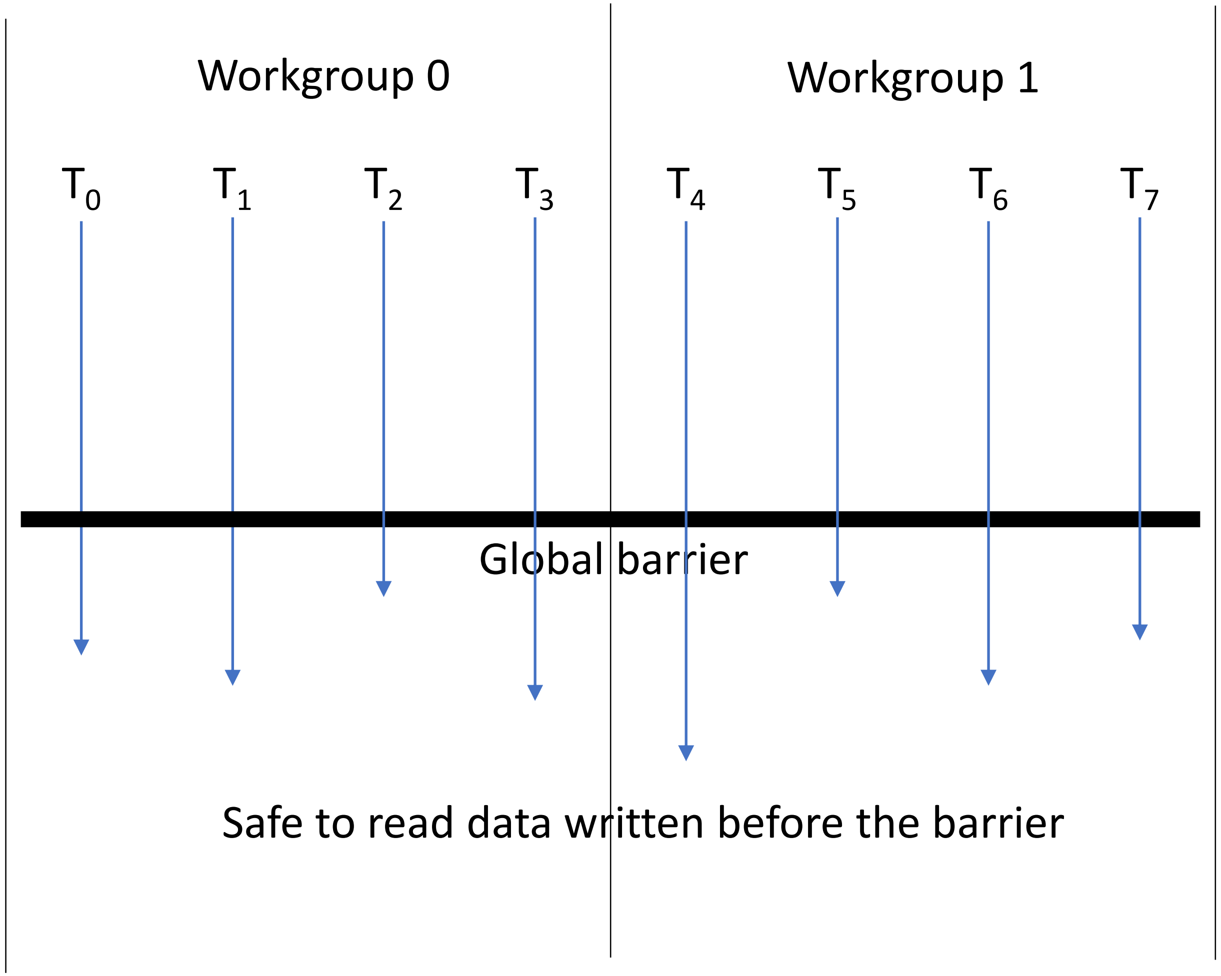
Global barrier?



Global barrier?



Global barrier?



Useful construct, but not provided as a primitive in OpenCL – more later!

Atomics

OpenCL 2.0 provides atomic operations for fine-grained communication between threads

The OpenCL memory model is *relaxed*: non sequentially consistent behaviours are allowed

Data race

Two memory accesses *race* if and only if:

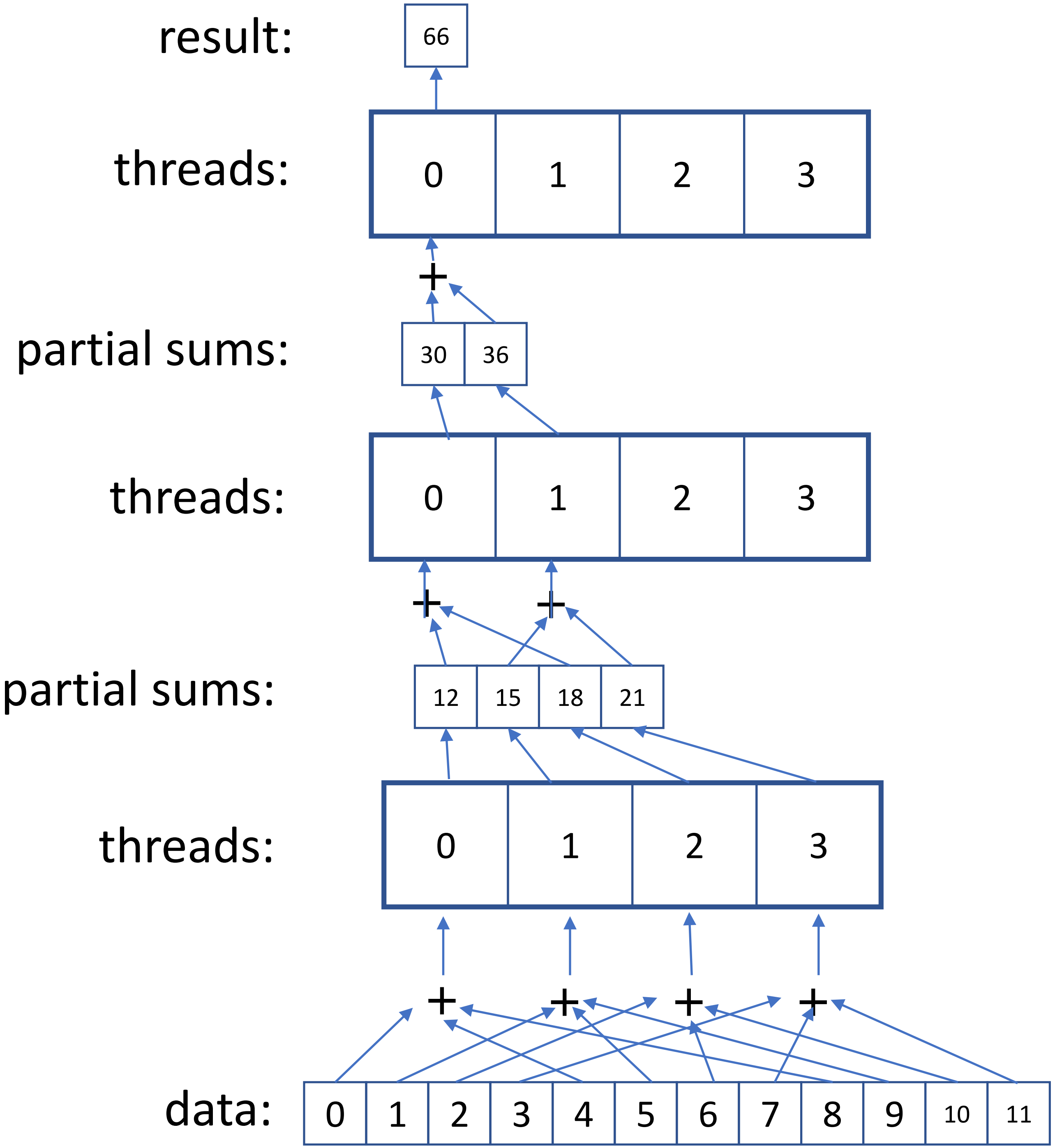
- They are issued by different threads
- They access a common memory location
- At least one access is non-atomic
- At least one access modifies the memory location
- No synchronization operation separates the accesses

Synchronization can be via a *barrier*, or through *synchronizing atomics*

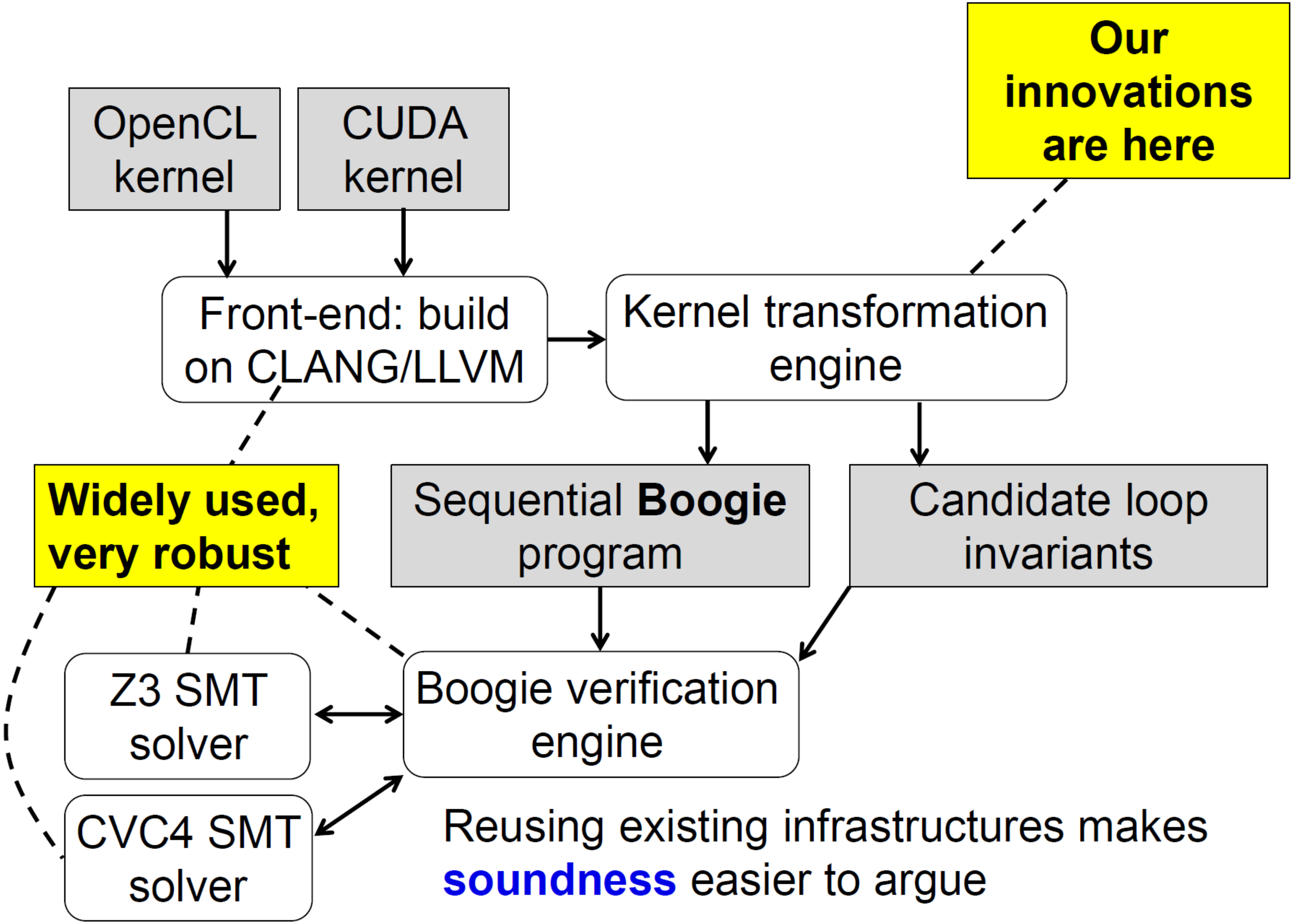
Let’s program a reduction in OpenCL

$$\text{reduce}([x_1, x_2, \ldots, x_n]) = x_1 + x_2 + \ldots + x_n$$

Reduction example with a single workgroup



GPUVerify: static data race analysis for GPU kernels



Reusing existing infrastructures makes **soundness** easier to argue

To learn more about GPUVerify:

The original paper:

- Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, Paul Thomson: *GPUVerify: a verifier for GPU kernels*. OOPSLA 2012: 113-132

Extended journal version:

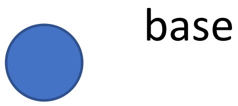
- Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, John Wickerson: *The Design and Implementation of a Verification Technique for GPU Kernels*. ACM Trans. Program. Lang. Syst. 37(3): 10:1-10:49 (2015)

Tool paper on engineering details, including optimizations:

- Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, Shaz Qadeer: *Engineering a Static Verification Tool for GPU Kernels*. CAV 2014: 226-242

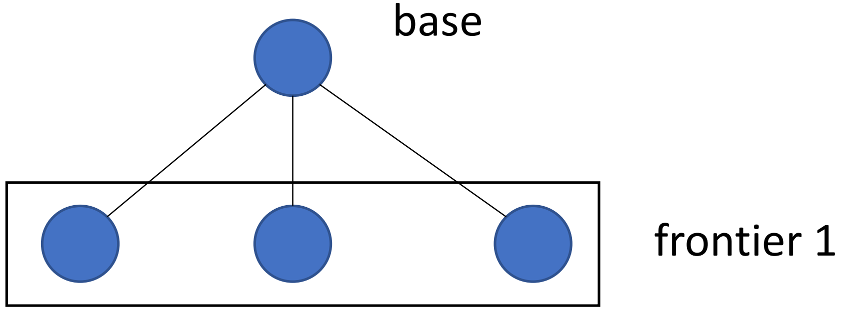
Global barrier motivation: graph traversal

- Frontier based graph traversal framework



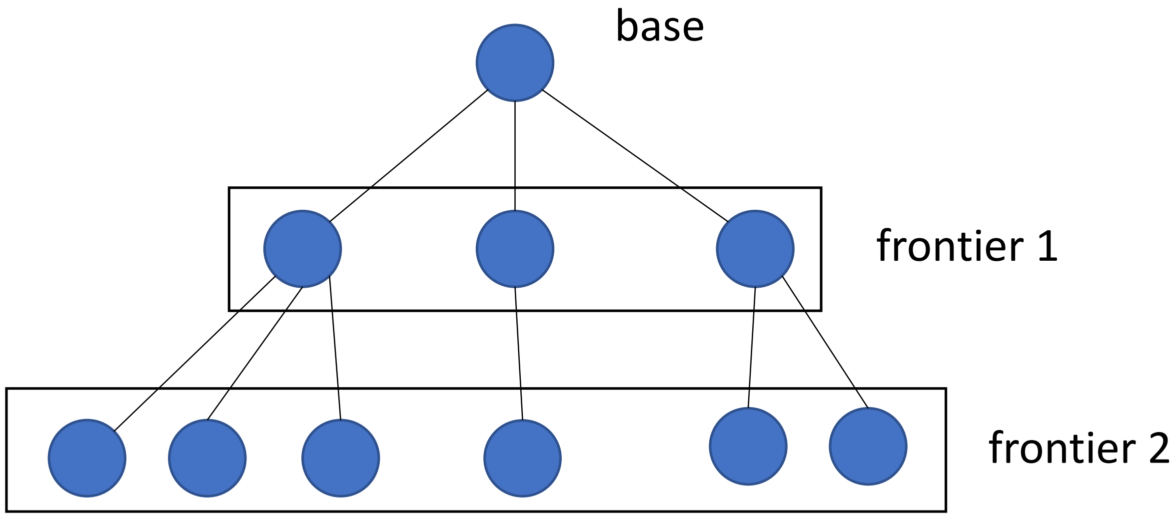
Global barrier motivation: graph traversal

- Frontier based graph traversal framework



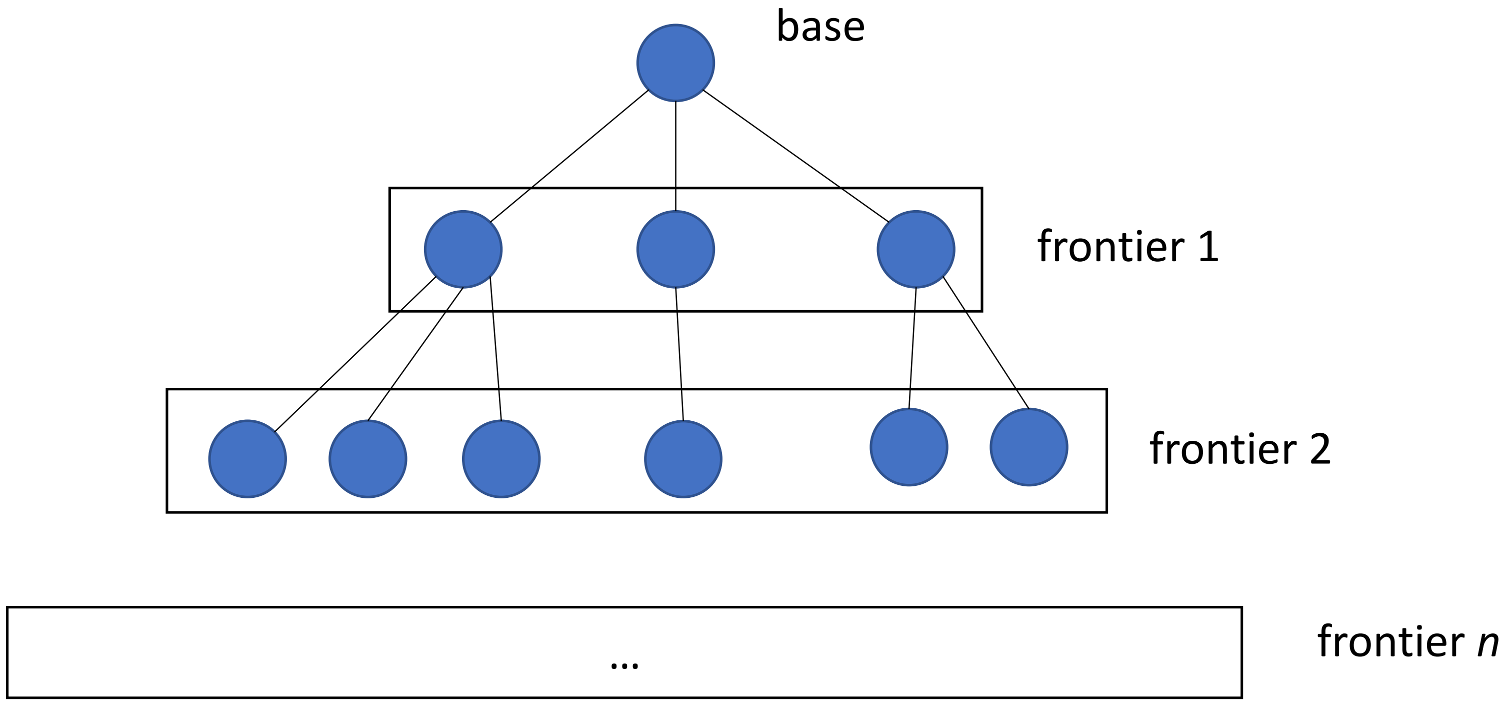
Global barrier motivation: graph traversal

- Frontier based graph traversal framework



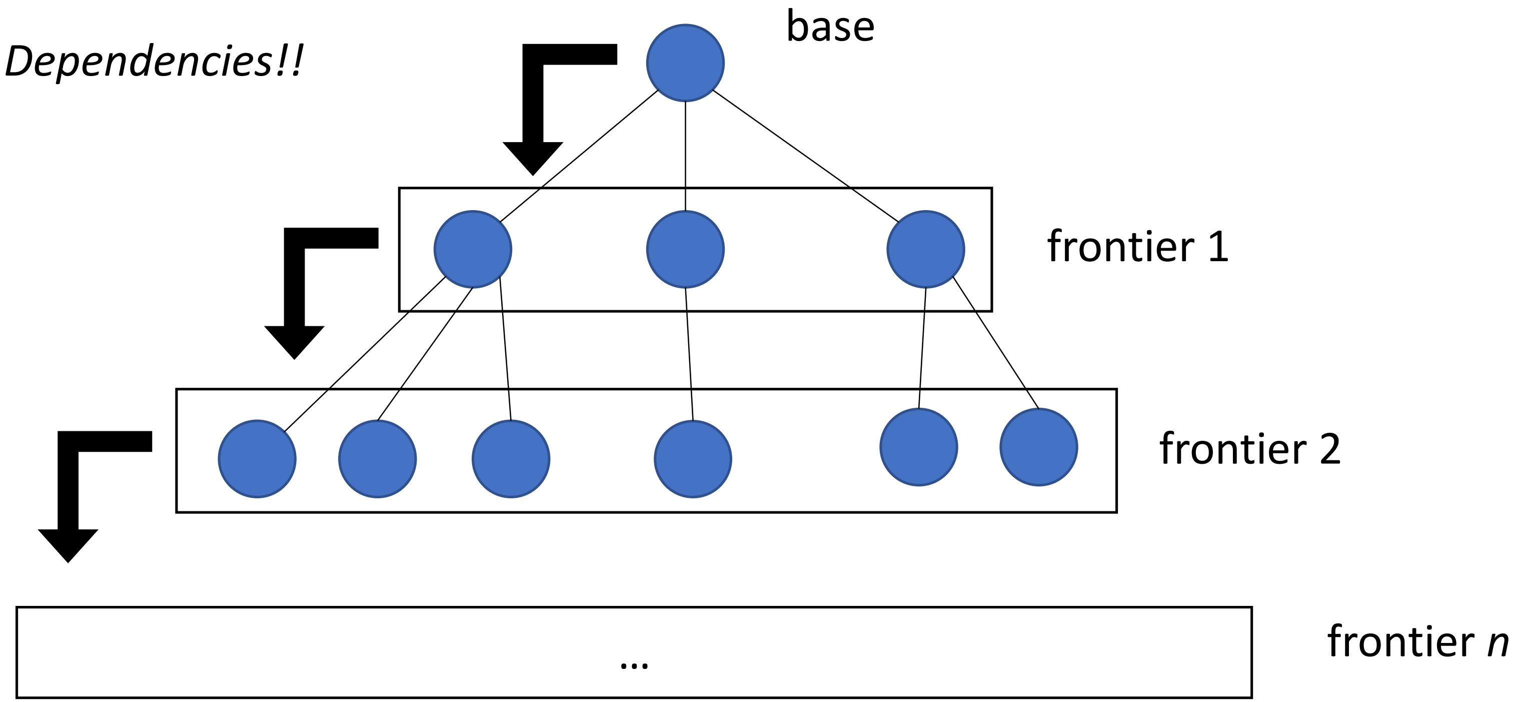
Global barrier motivation: graph traversal

- Frontier based graph traversal framework



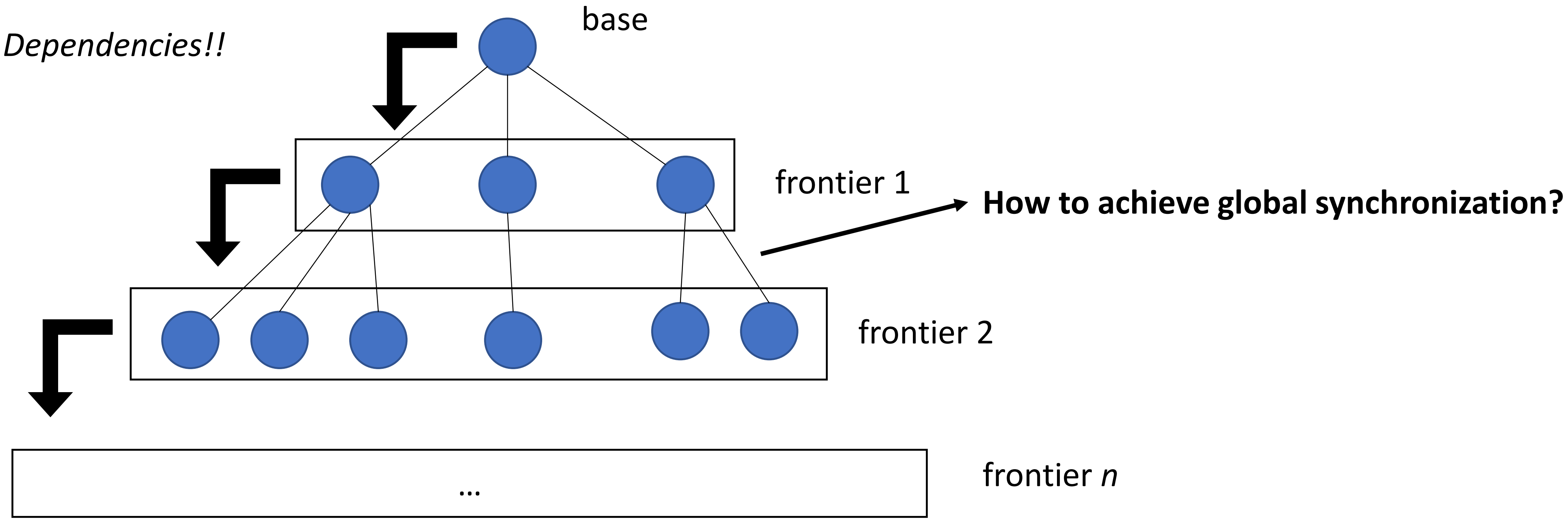
Global barrier motivation: graph traversal

- Frontier based graph traversal framework



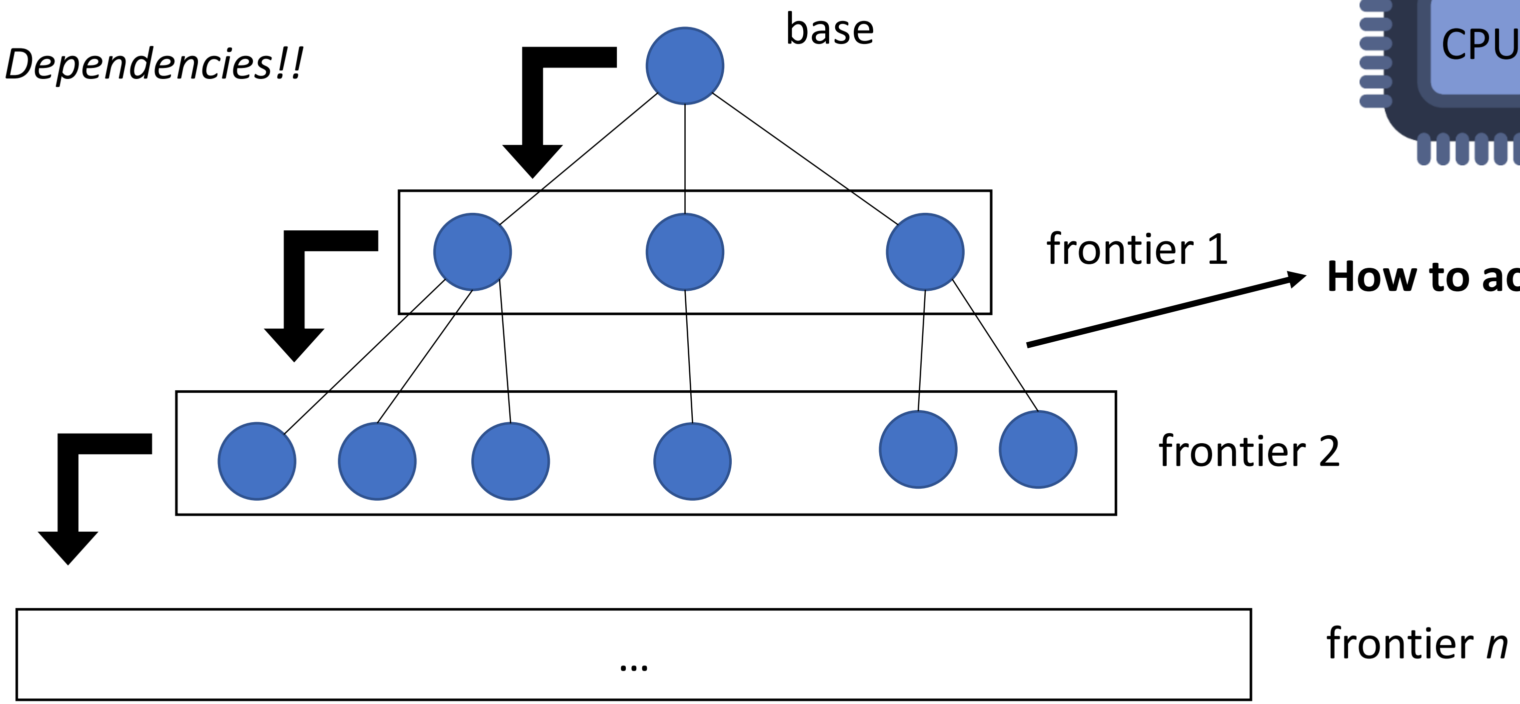
Global barrier motivation: graph traversal

- Frontier based graph traversal framework

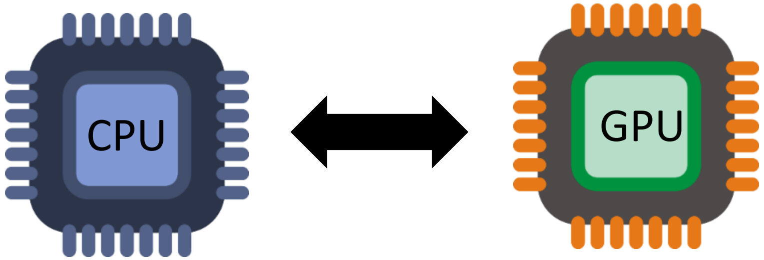


Global barrier motivation: graph traversal

- Frontier based graph traversal framework



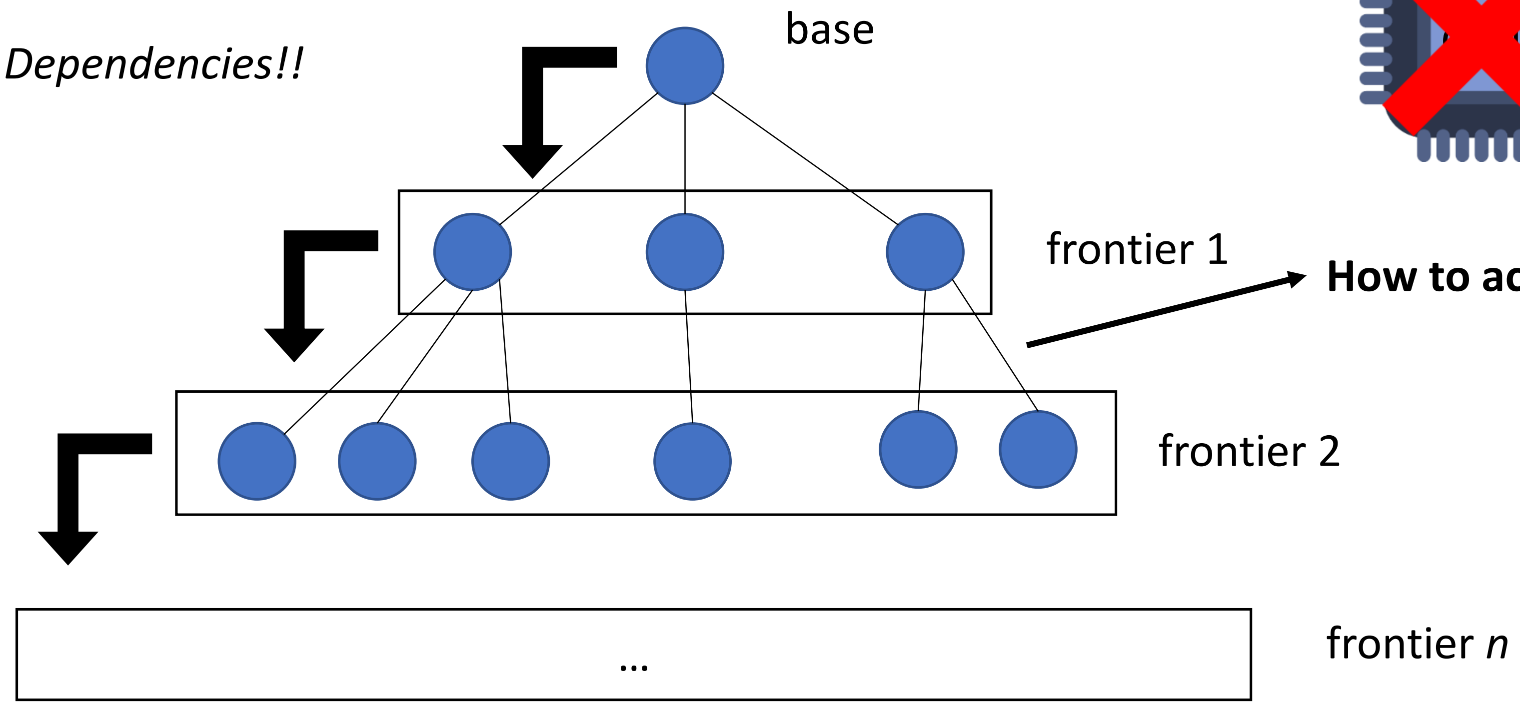
without global barrier, need to call GPU n times



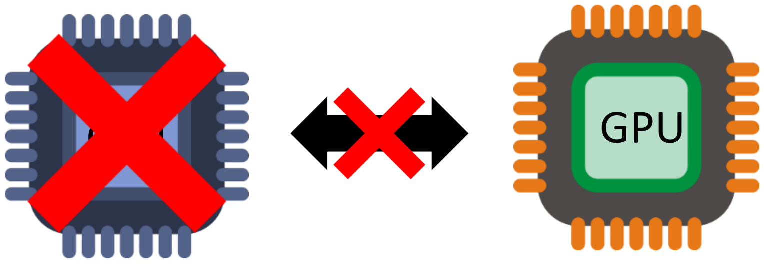
How to achieve global synchronization?

Global barrier motivation: graph traversal

- Frontier based graph traversal framework



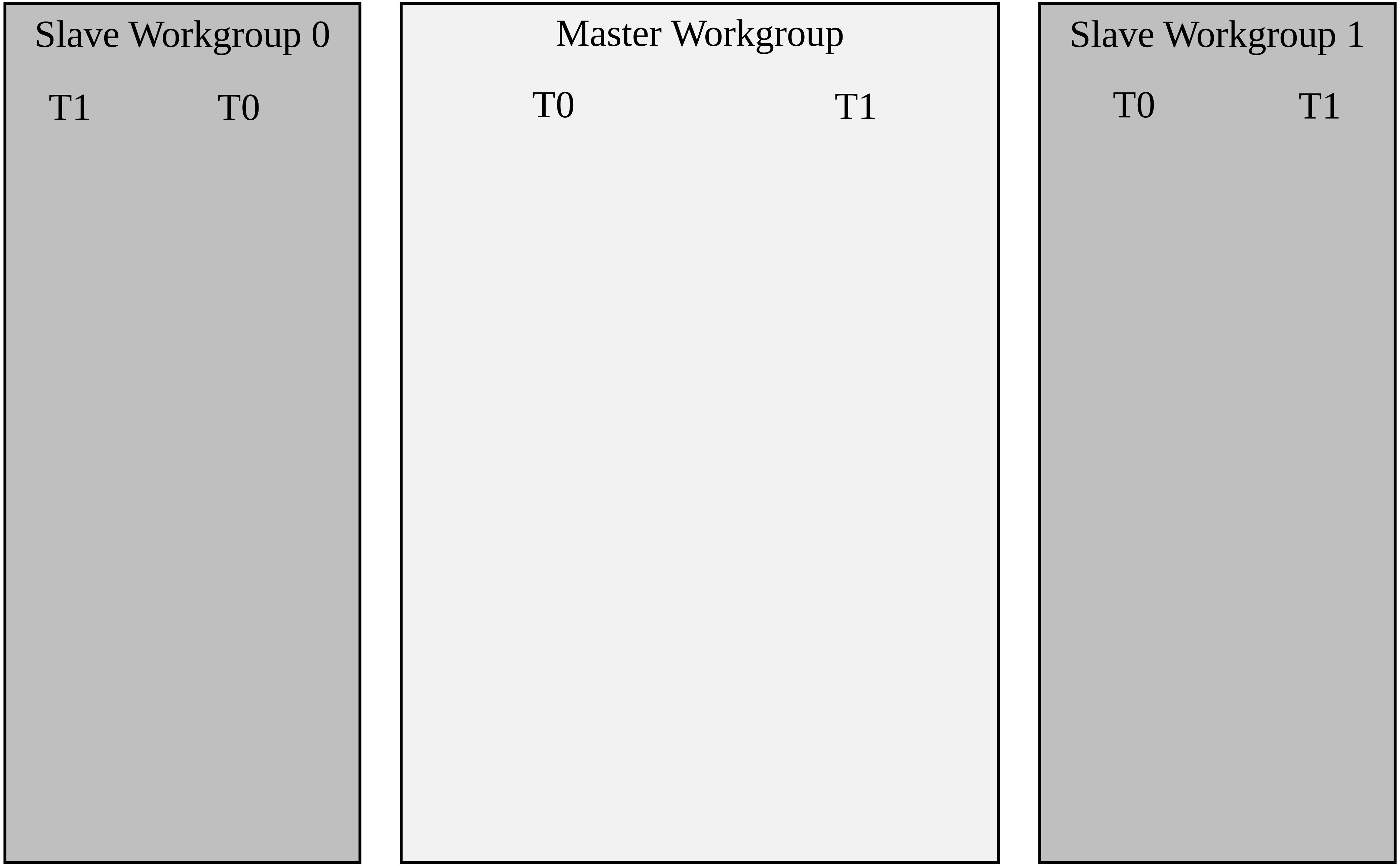
with global barrier, can use blocking synchronization

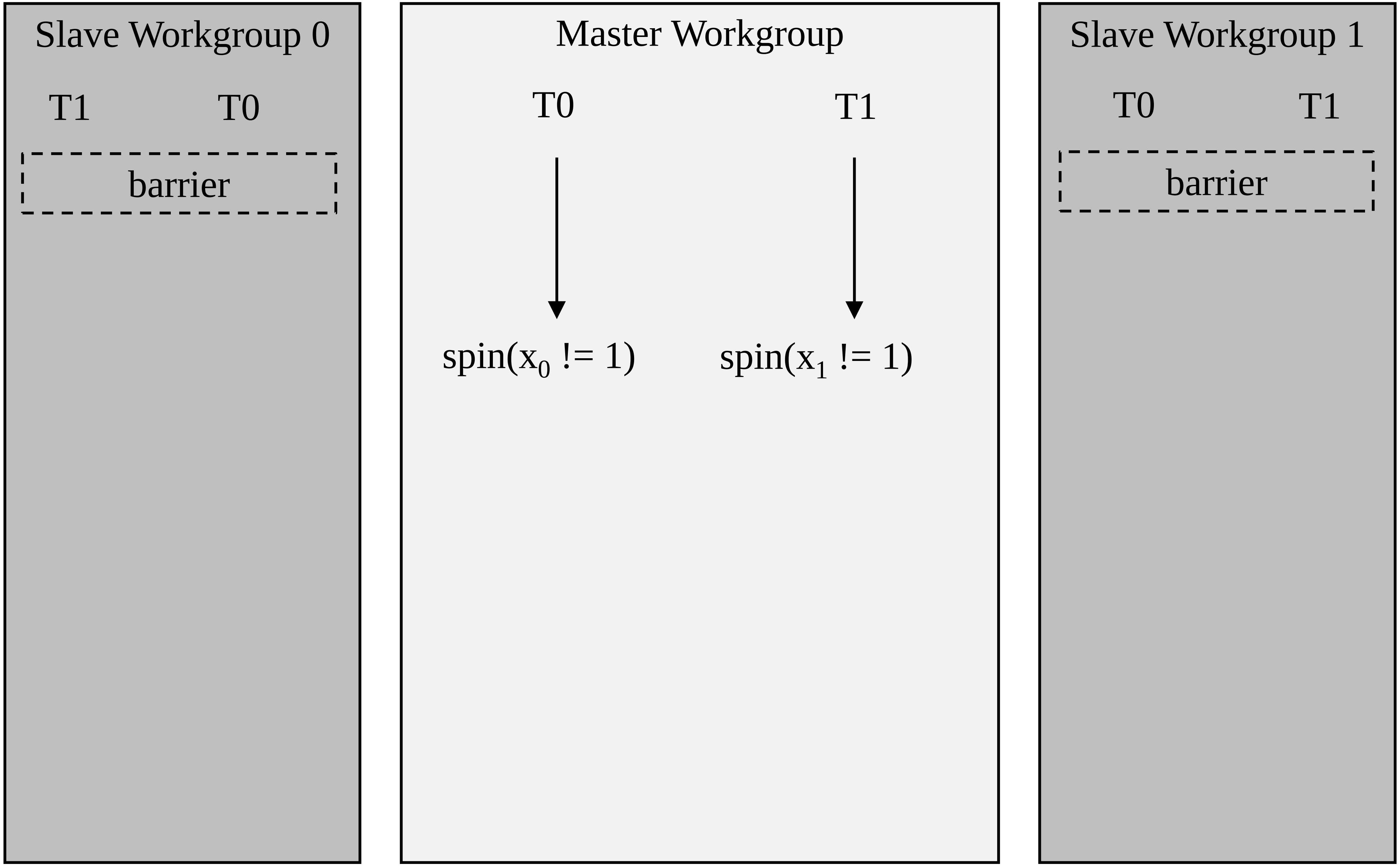


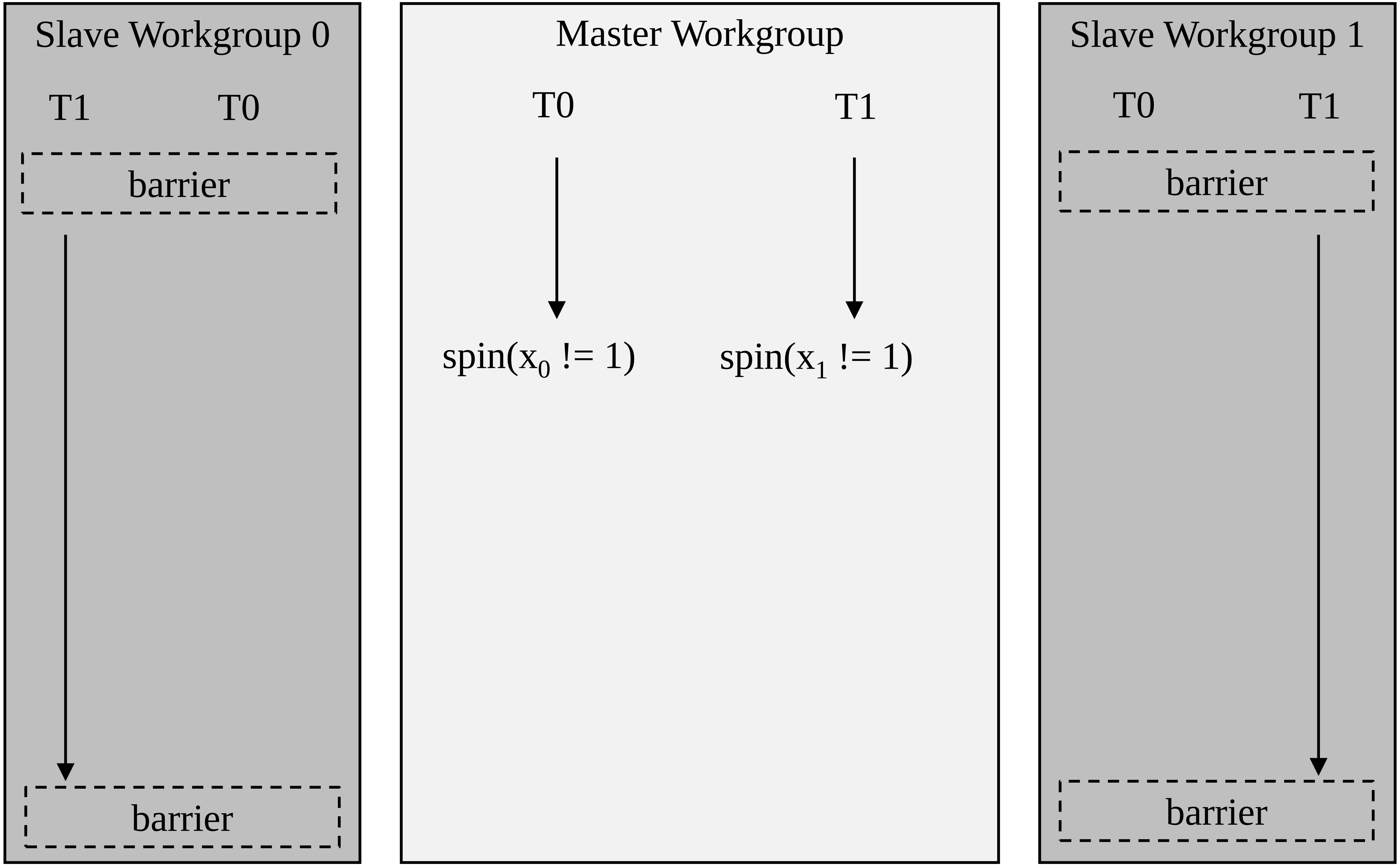
How to achieve global synchronization?

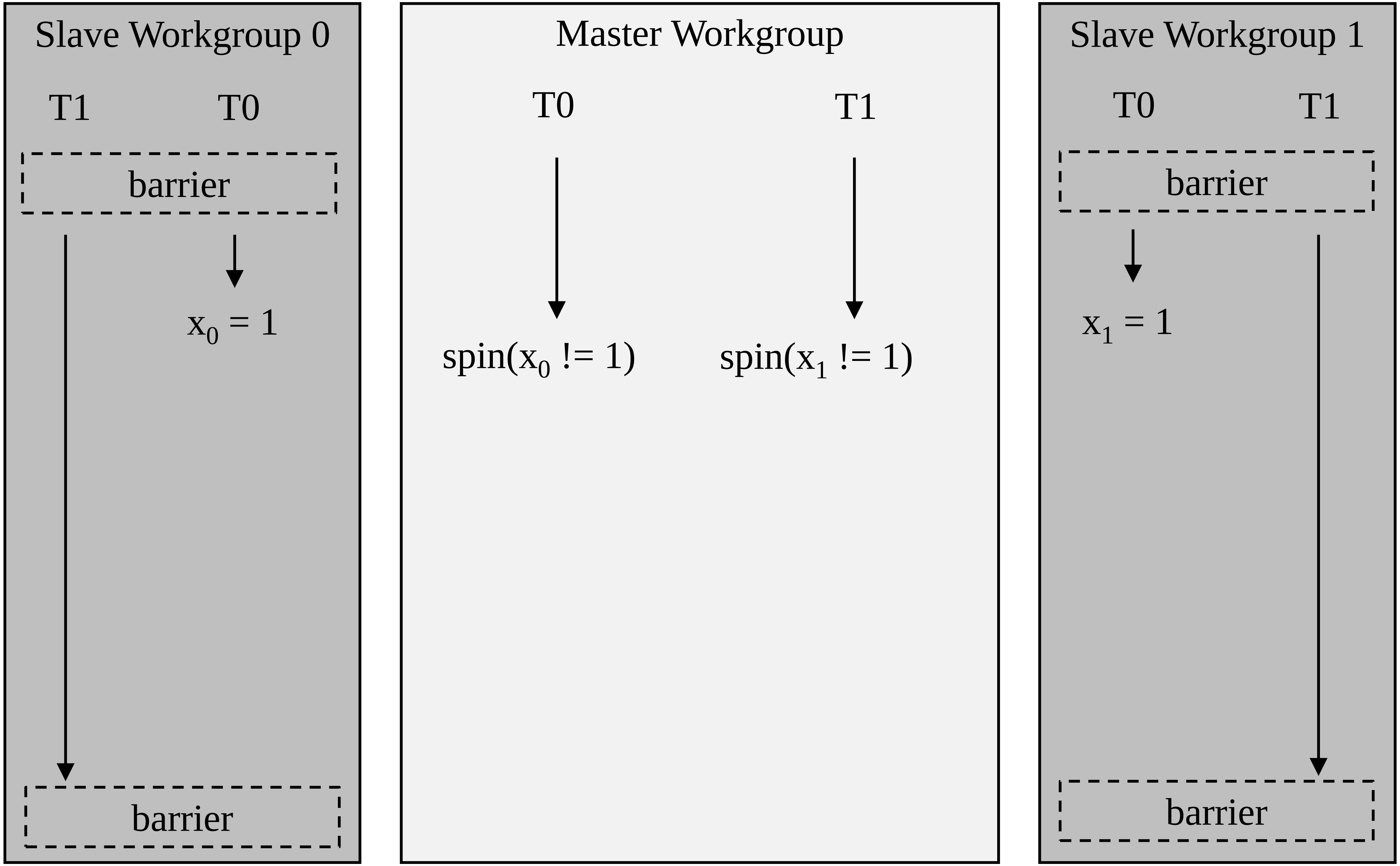
Barrier implementation

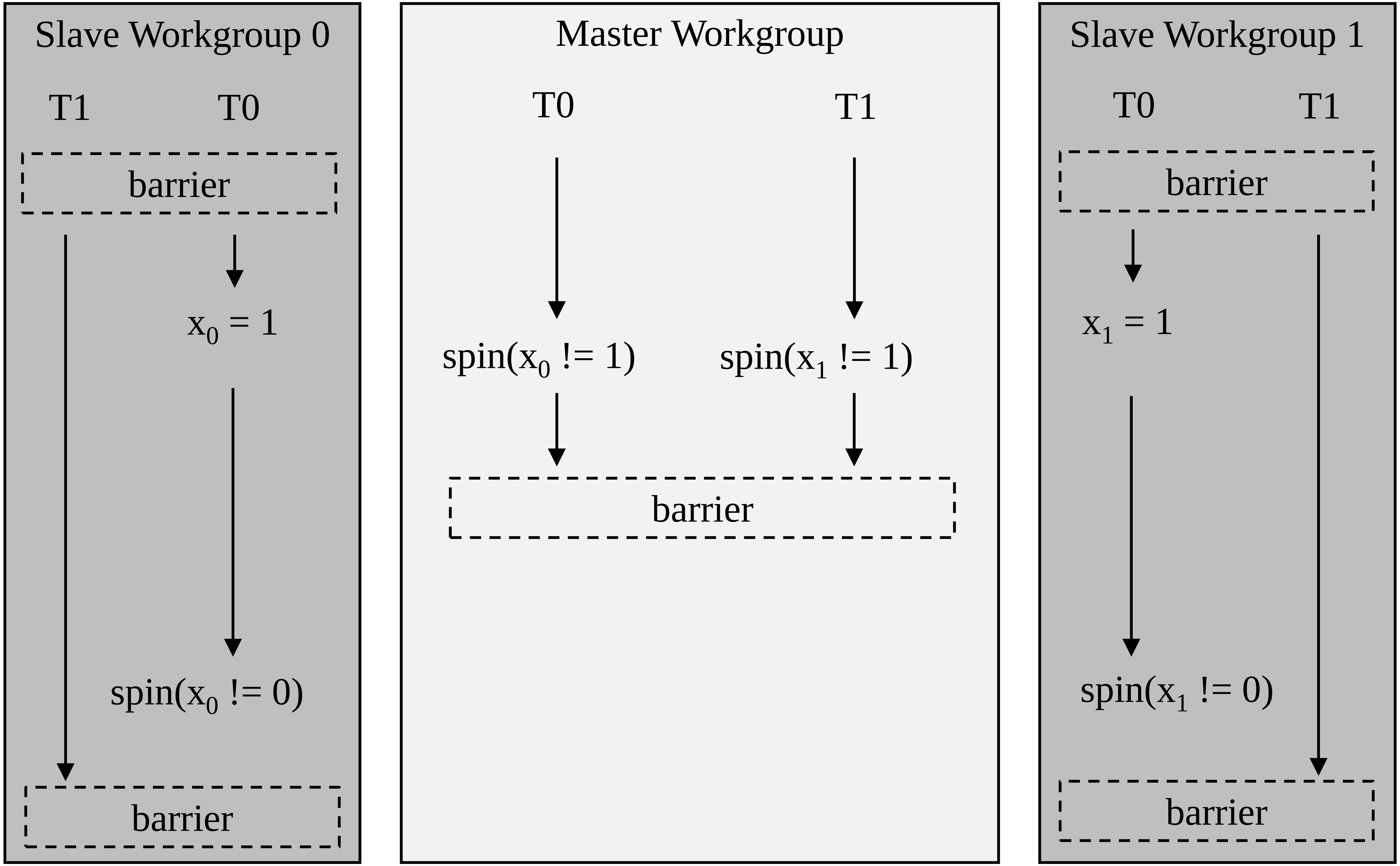
- Start with an implementation from Xiao and Feng (2010)
  - Written in CUDA (ported to OpenCL)
  - No formal memory consistency properties

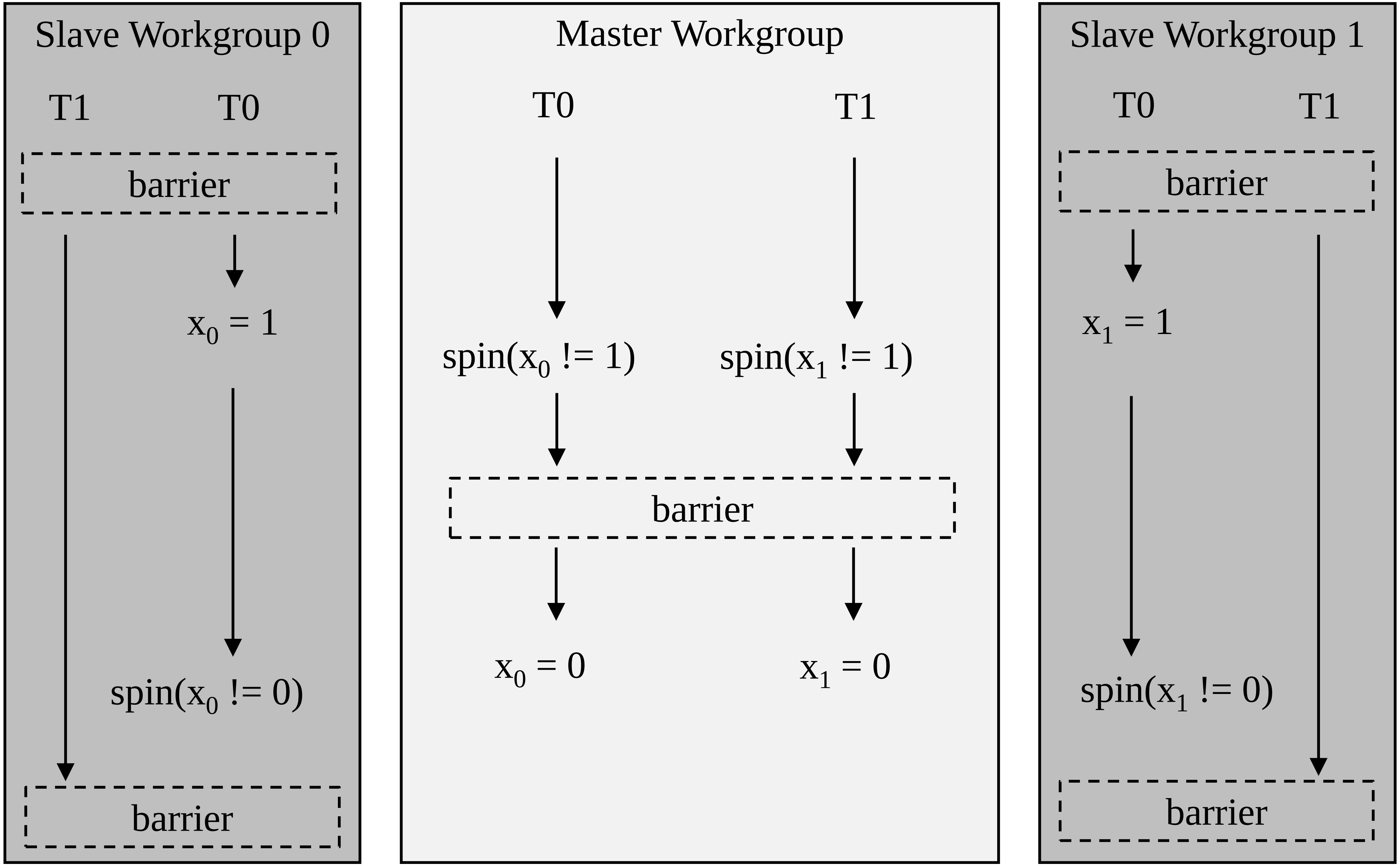

Slave Workgroup 0
T1
T0
Master Workgroup
T0
T1
Slave Workgroup 1
T0
T1


Slave Workgroup 0
T1
T0
barrier
Master Workgroup
T0
T1
spin(x_0 != 1)
spin(x_1 != 1)
Slave Workgroup 1
T0
T1
barrier

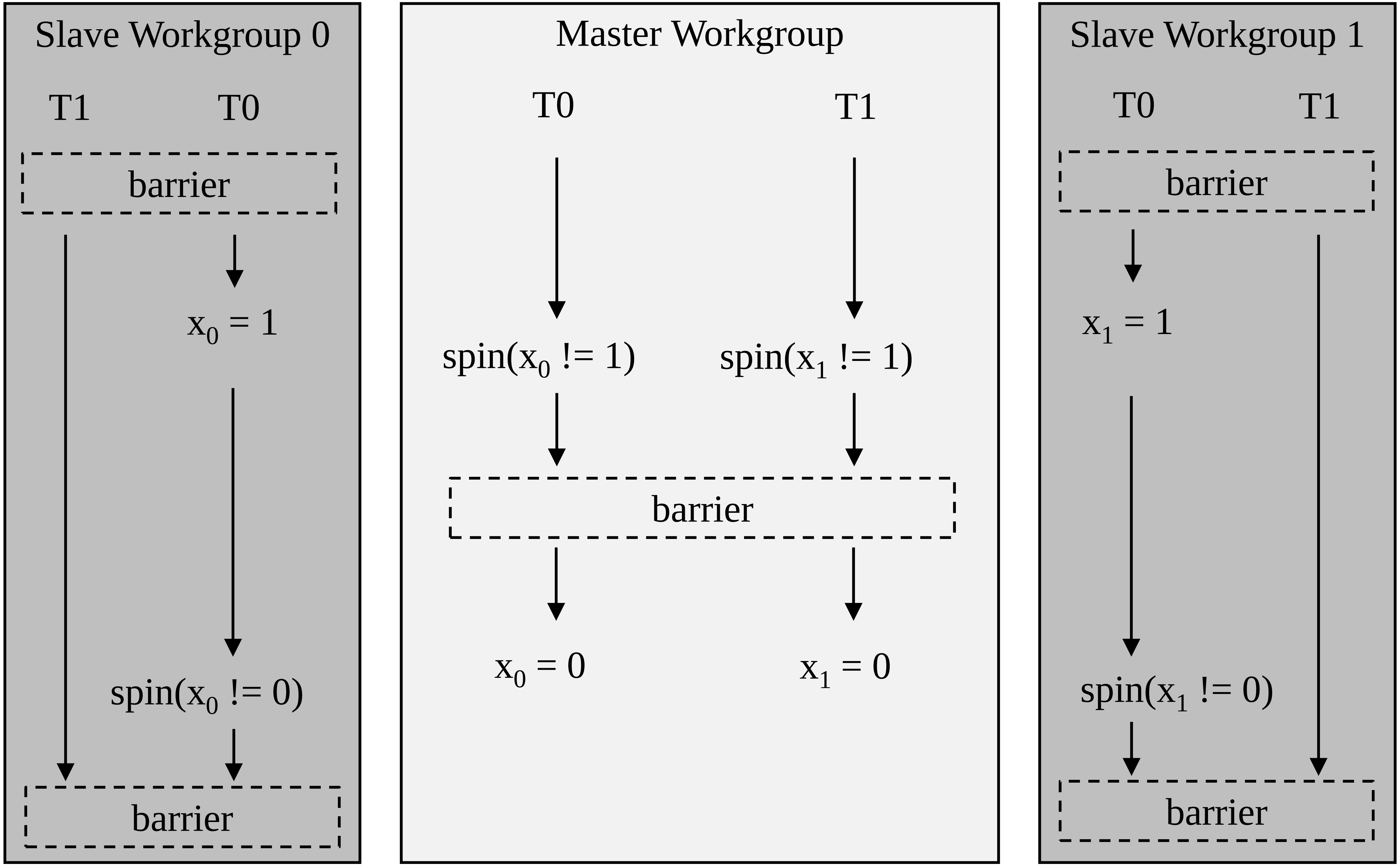

Slave Workgroup 0
T1
T0
barrier
barrier
Master Workgroup
T0
T1
spin(x_0 != 1)
spin(x_1 != 1)
Slave Workgroup 1
T0
T1
barrier
barrier

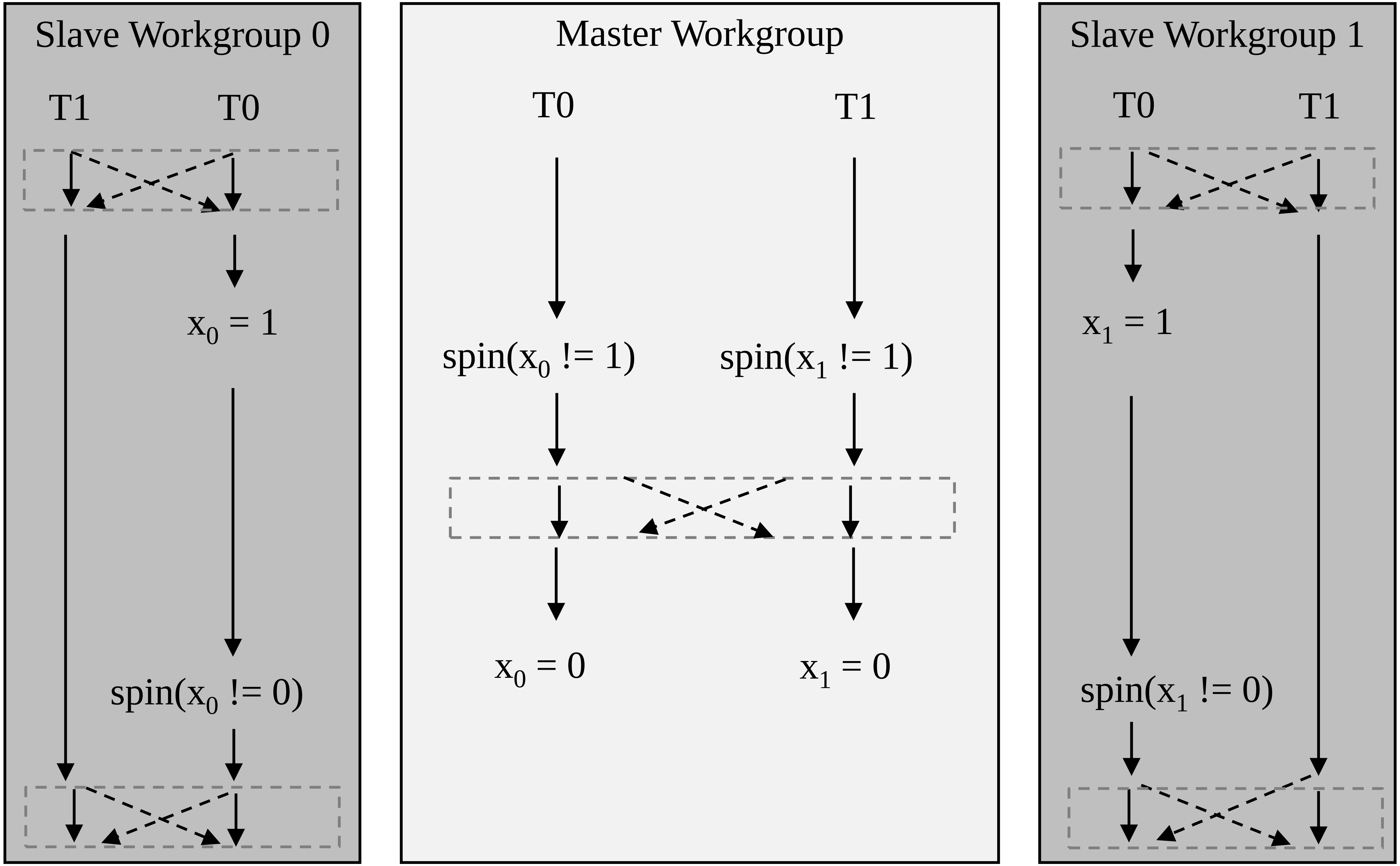

Slave Workgroup 0
T1
T0
barrier
$x_0 = 1$
barrier
Master Workgroup
T0
T1
spin(x_0 != 1)
spin(x_1 != 1)
Slave Workgroup 1
T0
T1
barrier
$x_1 = 1$
barrier


Slave Workgroup 0
T1
T0
barrier
$x_0 = 1$
spin(x_0 != 0)
barrier
Master Workgroup
T0
T1
spin(x_0 != 1)
spin(x_1 != 1)
barrier
Slave Workgroup 1
T0
T1
barrier
$x_1 = 1$
spin(x_1 != 0)
barrier



Slave Workgroup 0
T1
T0
barrier
$x_0 = 1$
spin(x_0 != 0)
barrier
Master Workgroup
T0
T1
spin(x_0 != 1)
spin(x_1 != 1)
barrier
$x_0 = 0$
$x_1 = 0$
Slave Workgroup 1
T0
T1
barrier
$x_1 = 1$
spin(x_1 != 0)
barrier


Slave Workgroup 0
T1
T0
barrier
$x_0 = 1$
spin(x_0 != 0)
barrier
Master Workgroup
T0
T1
spin(x_0 != 1)
spin(x_1 != 1)
barrier
$x_0 = 0$
$x_1 = 0$
Slave Workgroup 1
T0
T1
barrier
$x_1 = 1$
spin(x_1 != 0)
barrier



Slave Workgroup 0
T1
T0
$x_0 = 1$
spin(x_0 != 0)
Master Workgroup
T0
T1
spin(x_0 != 1)
spin(x_1 != 1)
$x_0 = 0$
$x_1 = 0$
Slave Workgroup 1
T0
T1
$x_1 = 1$
spin(x_1 != 0)

Let’s implement this

Barrier memory consistency

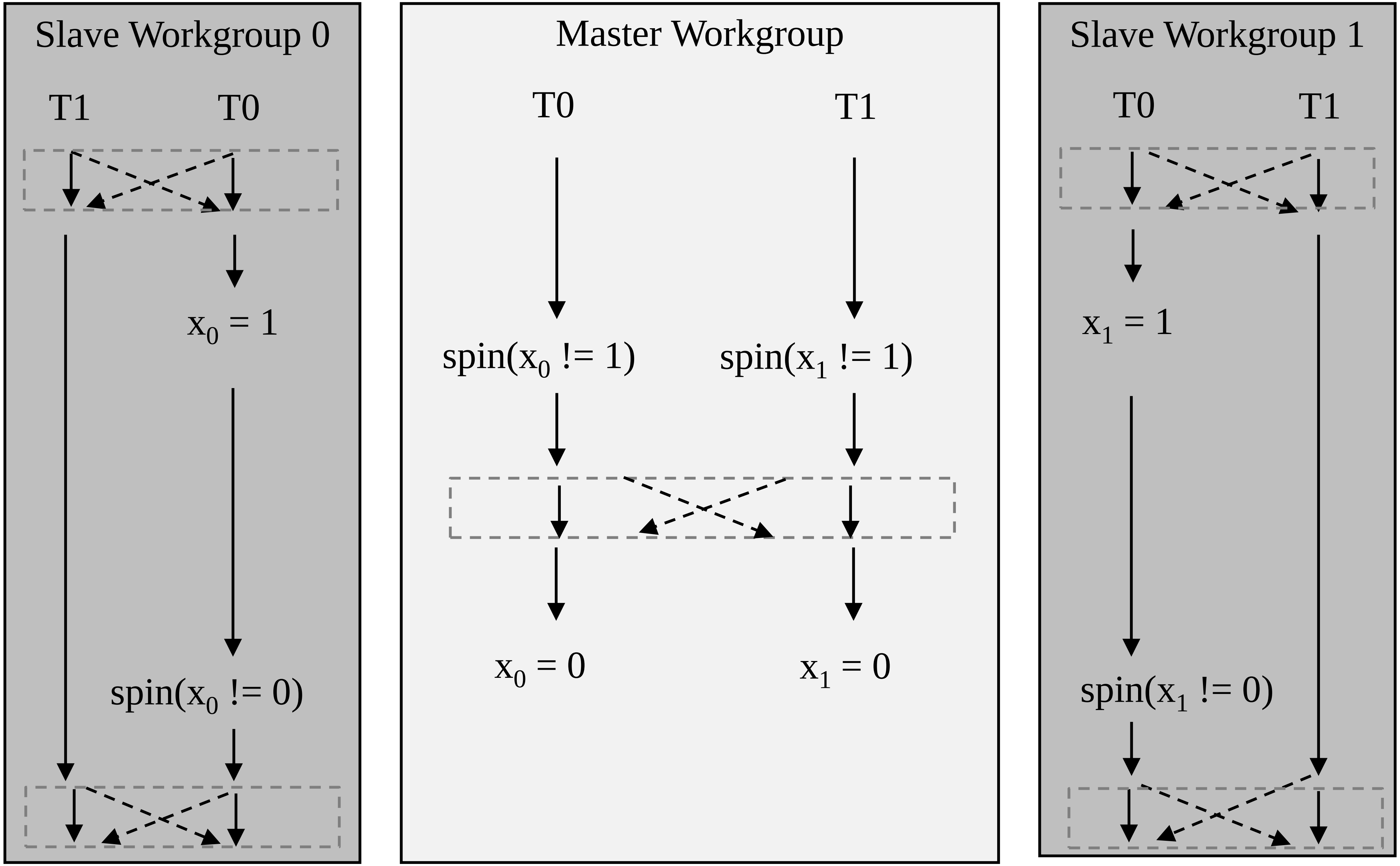
- Device release-acquire rule

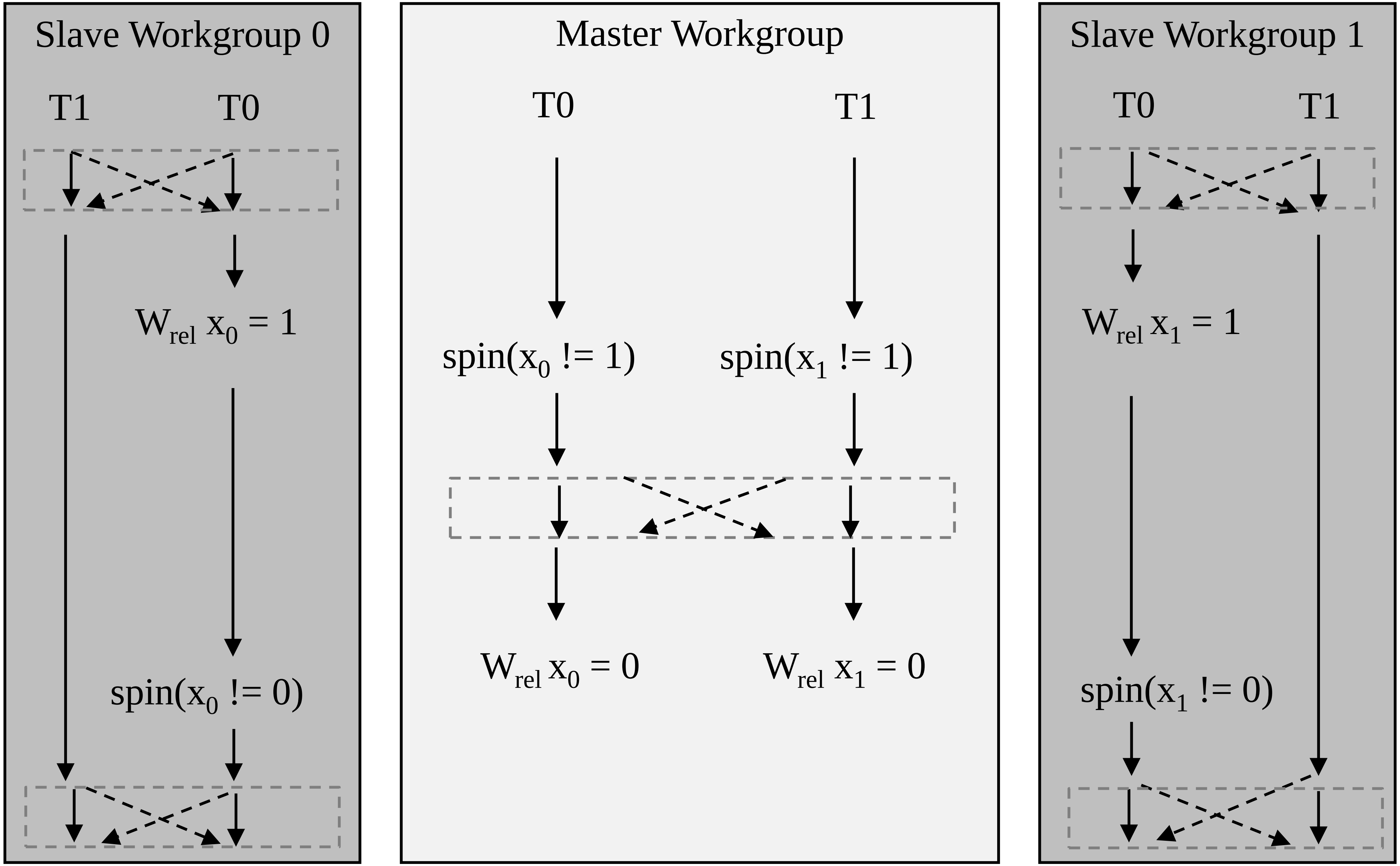
T0 and T1 in different workgroups

T0

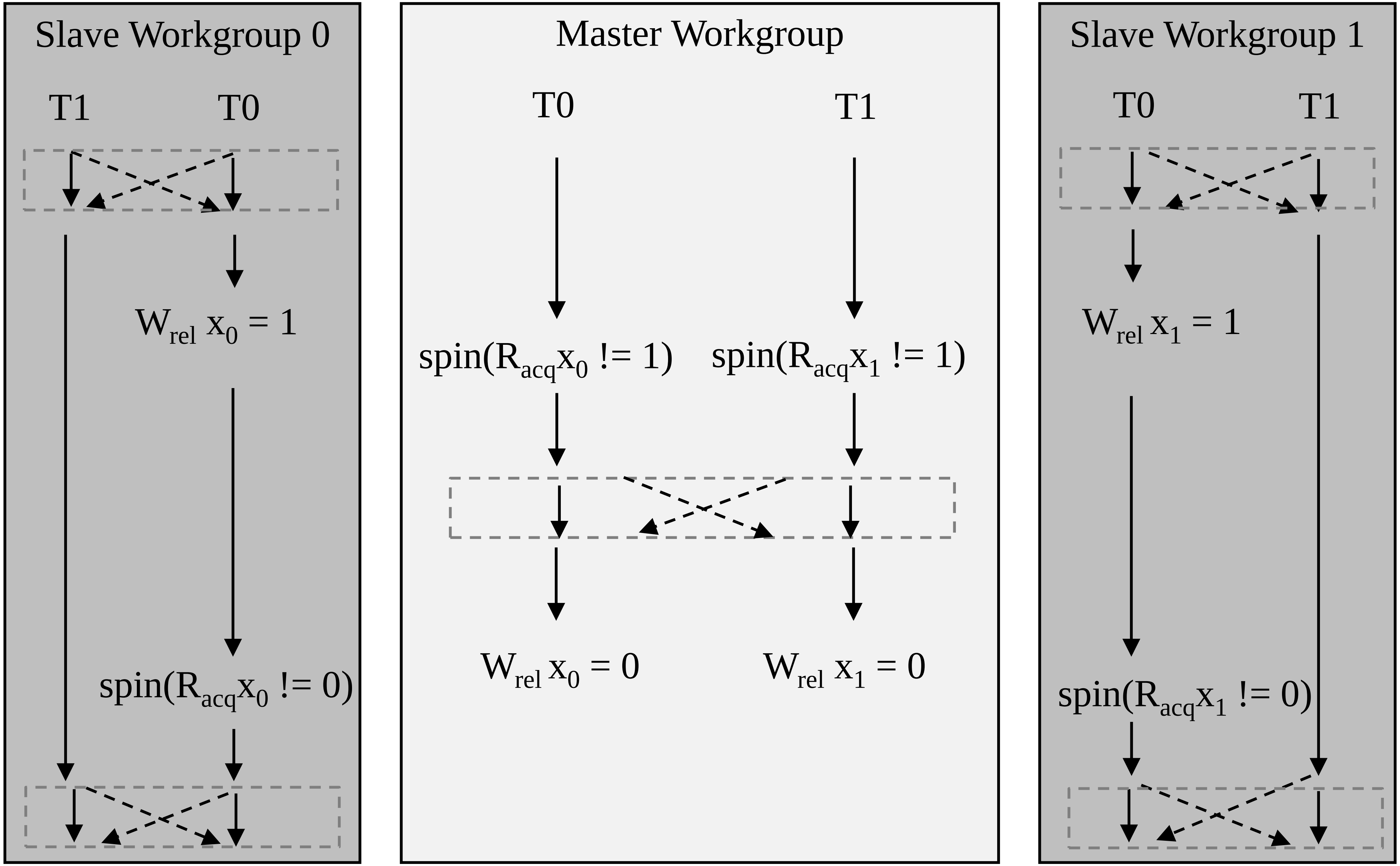
T1

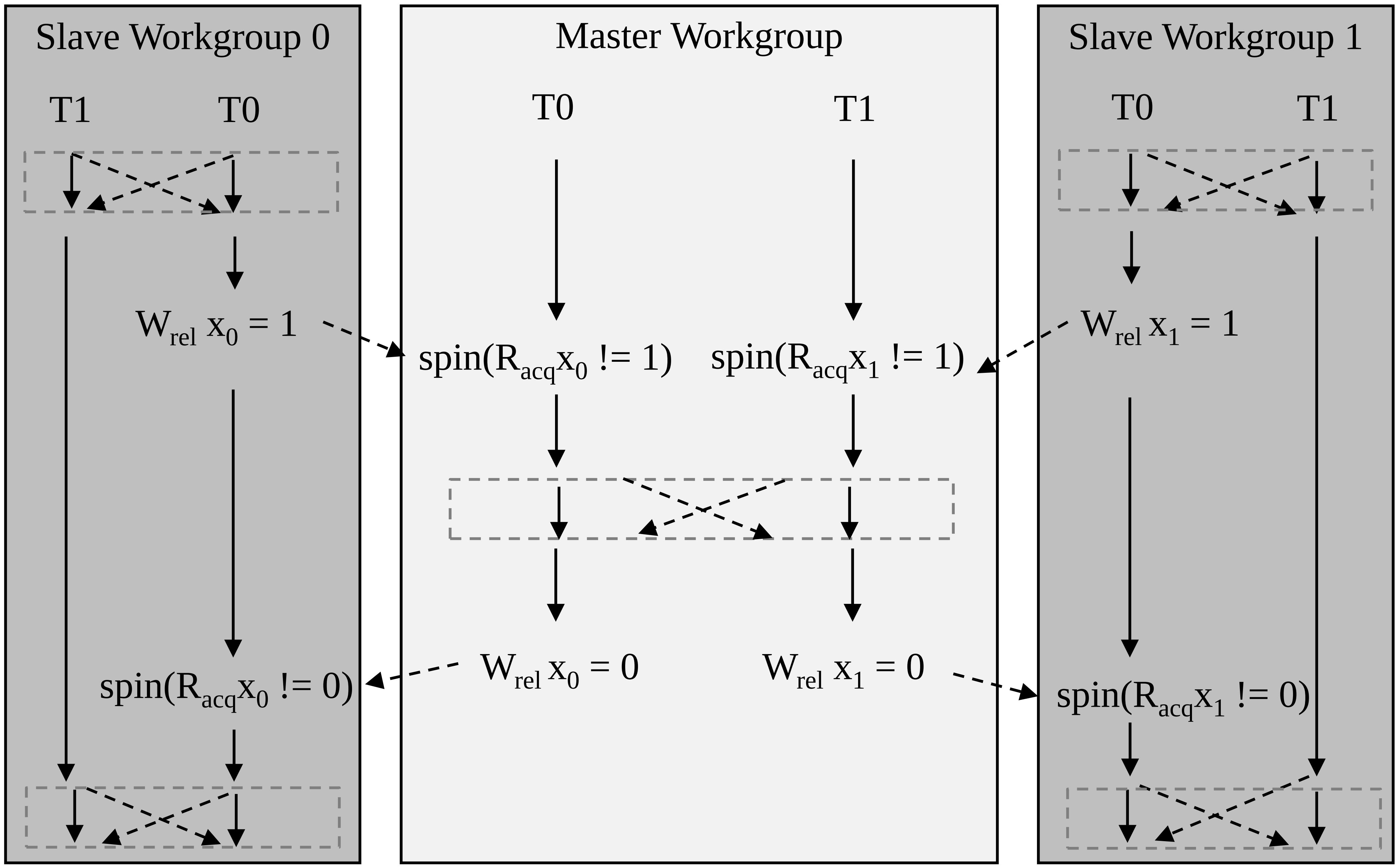
$W_{rel}\ y = 1$ - - - - - - - - - - - - - - - - - - → $R_{acq}\ y = 1$


Slave Workgroup 0
T1
T0
$x_0 = 1$
spin(x_0 != 0)
Master Workgroup
T0
T1
spin(x_0 != 1)
spin(x_1 != 1)
$x_0 = 0$
$x_1 = 0$
Slave Workgroup 1
T0
T1
$x_1 = 1$
spin(x_1 != 0)



Slave Workgroup 0
T1
T0
$W_{rel}\ x_0 = 1$
spin(x_0 != 0)
Master Workgroup
T0
T1
spin(x_0 != 1)
spin(x_1 != 1)
$W_{rel}\ x_0 = 0$
$W_{rel}\ x_1 = 0$
Slave Workgroup 1
T0
T1
$W_{rel}\ x_1 = 1$
spin(x_1 != 0)


Slave Workgroup 0
T1
T0
$W_{rel}\ x_0 = 1$
$spin(R_{acq}x_0\ != 0)$
Master Workgroup
T0
T1
$spin(R_{acq}x_0\ != 1)$
$spin(R_{acq}x_1\ != 1)$
$W_{rel}\ x_0 = 0$
$W_{rel}\ x_1 = 0$
Slave Workgroup 1
T0
T1
$W_{rel}\ x_1 = 1$
$spin(R_{acq}x_1\ != 0)$


Slave Workgroup 0
Master Workgroup
Slave Workgroup 1
T1
T0
T0
T1
T0
T1
W_rel x_0 = 1
spin(R_acq x_0 != 1)
spin(R_acq x_1 != 1)
W_rel x_1 = 1
spin(R_acq x_0 != 0)
W_rel x_0 = 0
W_rel x_1 = 0
spin(R_acq x_1 != 0)

Let’s implement this

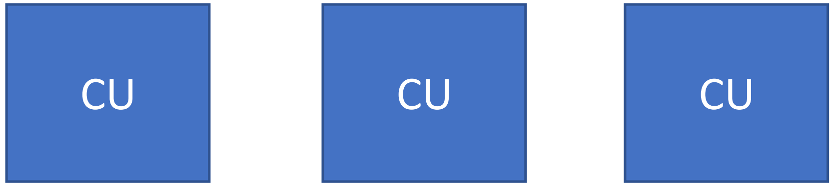
Problem with global barrier

- Global synchronization leads to **deadlock** if too many workgroups participate in barrier

Occupancy bound execution

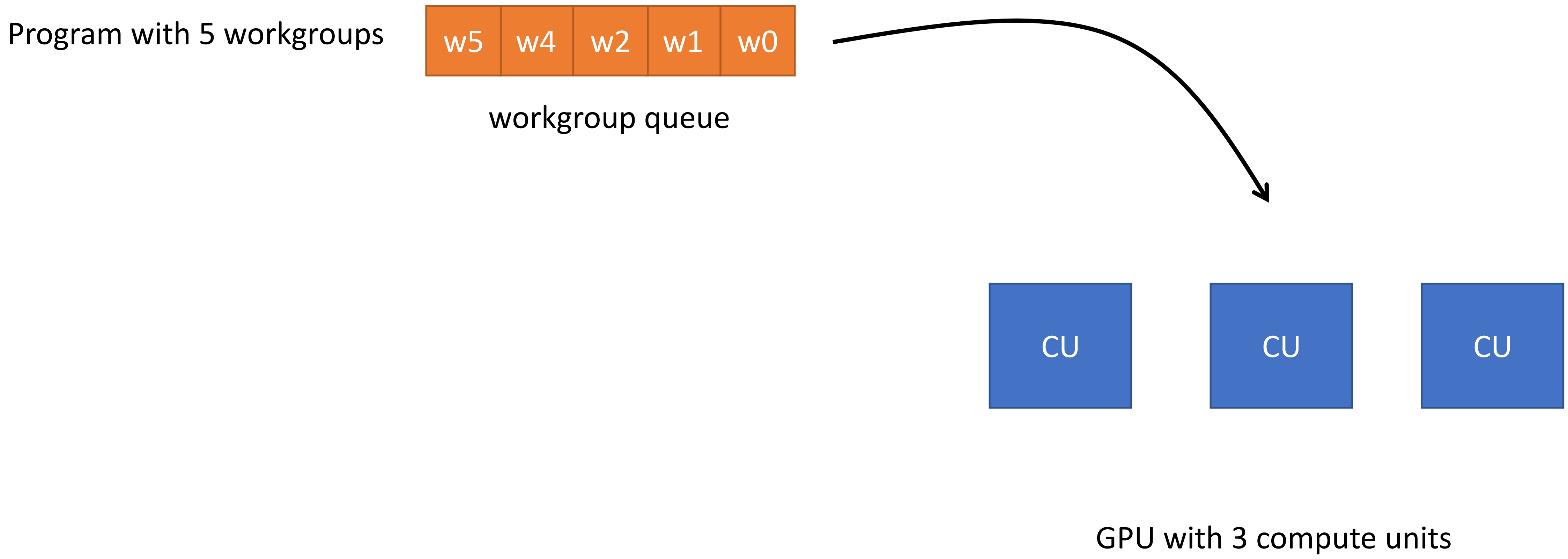
Program with 5 workgroups

workgroup queue

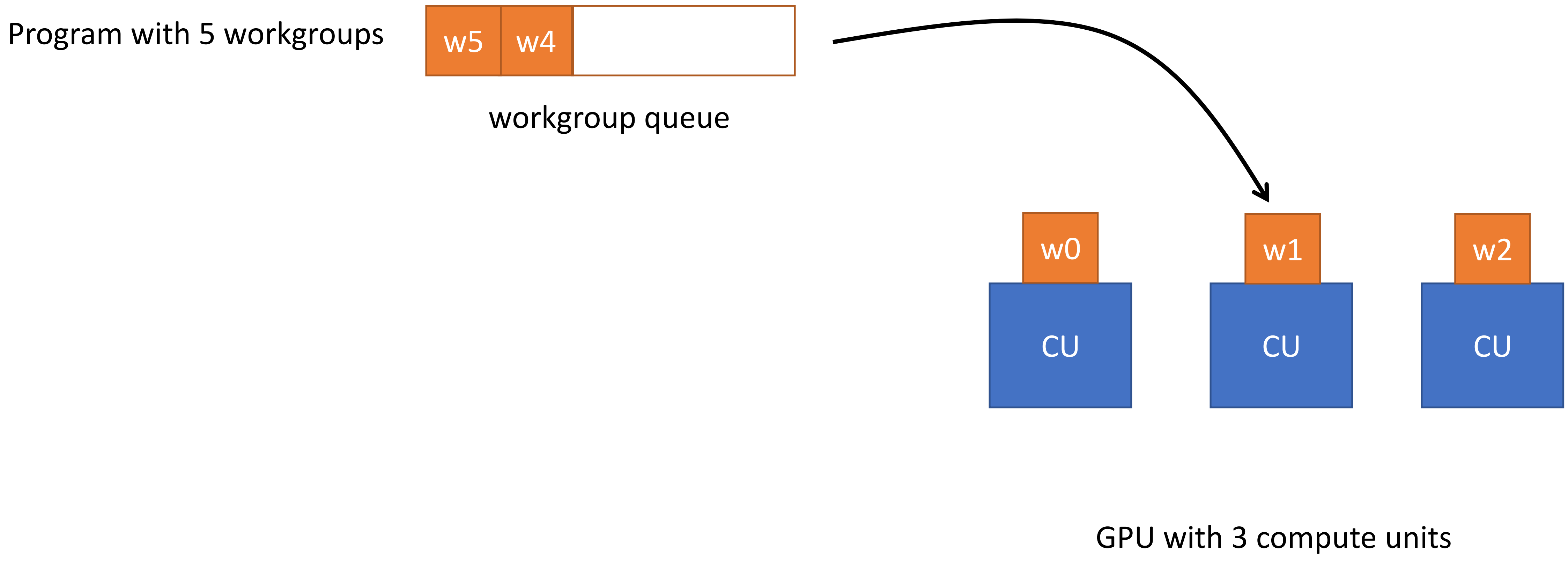


GPU with 3 compute units

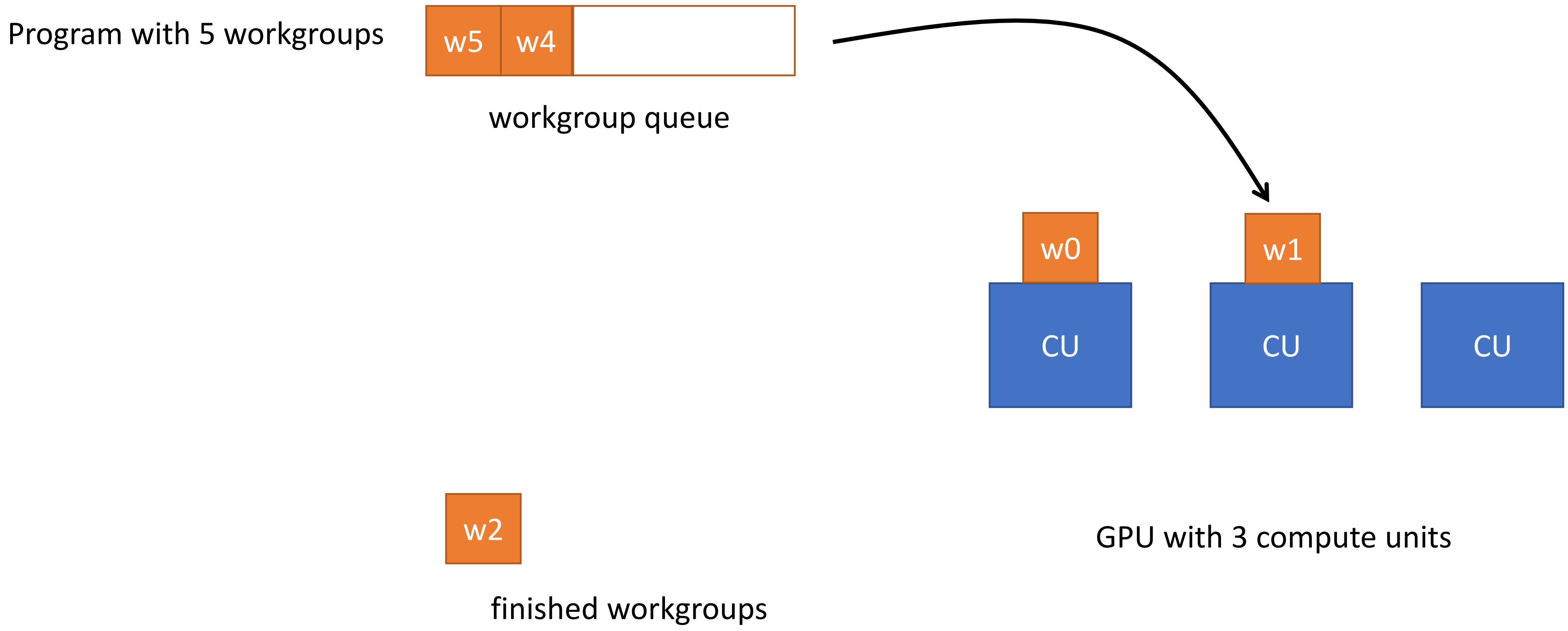
Occupancy bound execution



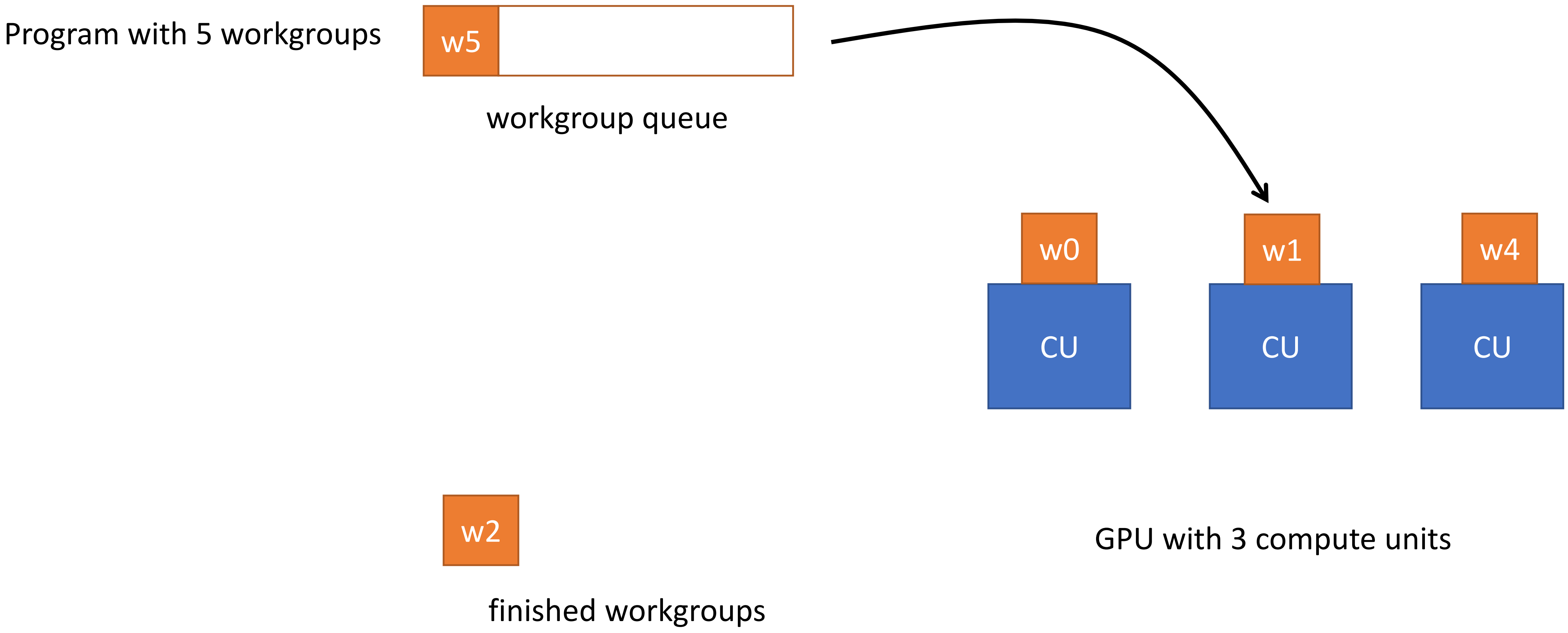
Occupancy bound execution



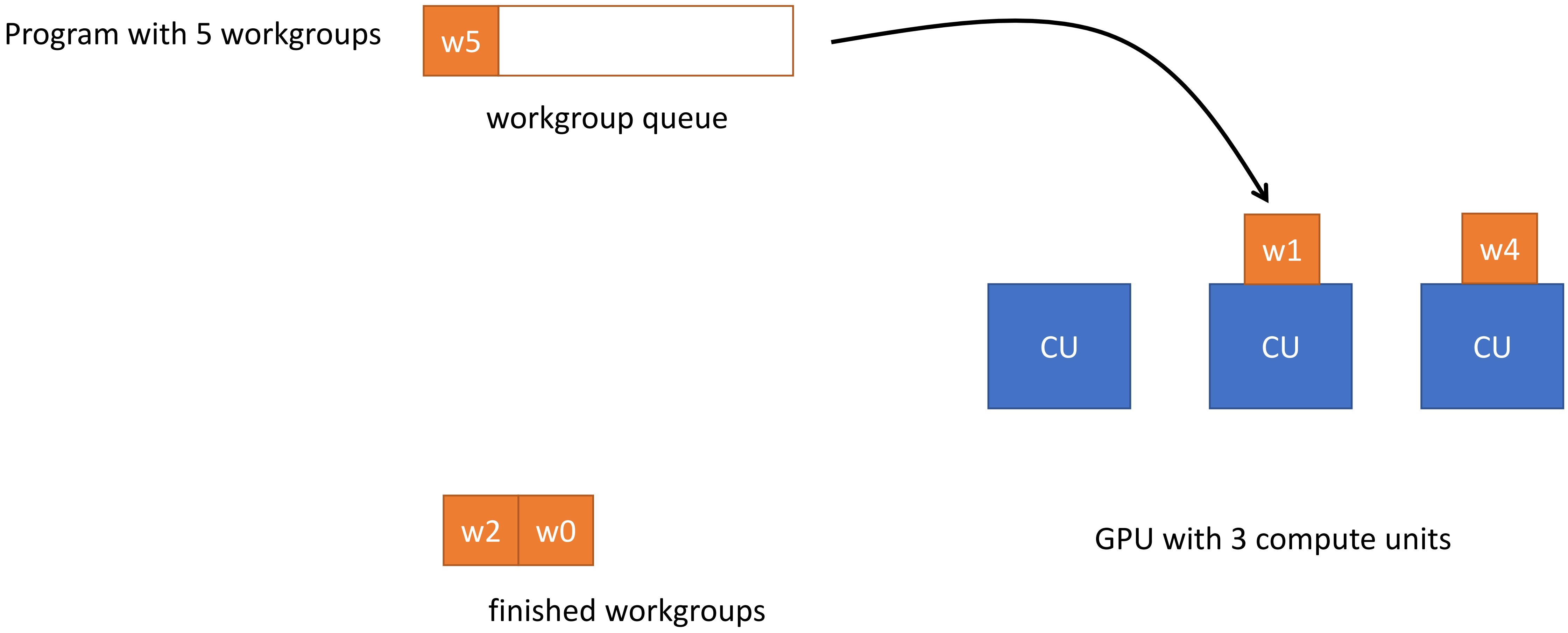
Occupancy bound execution



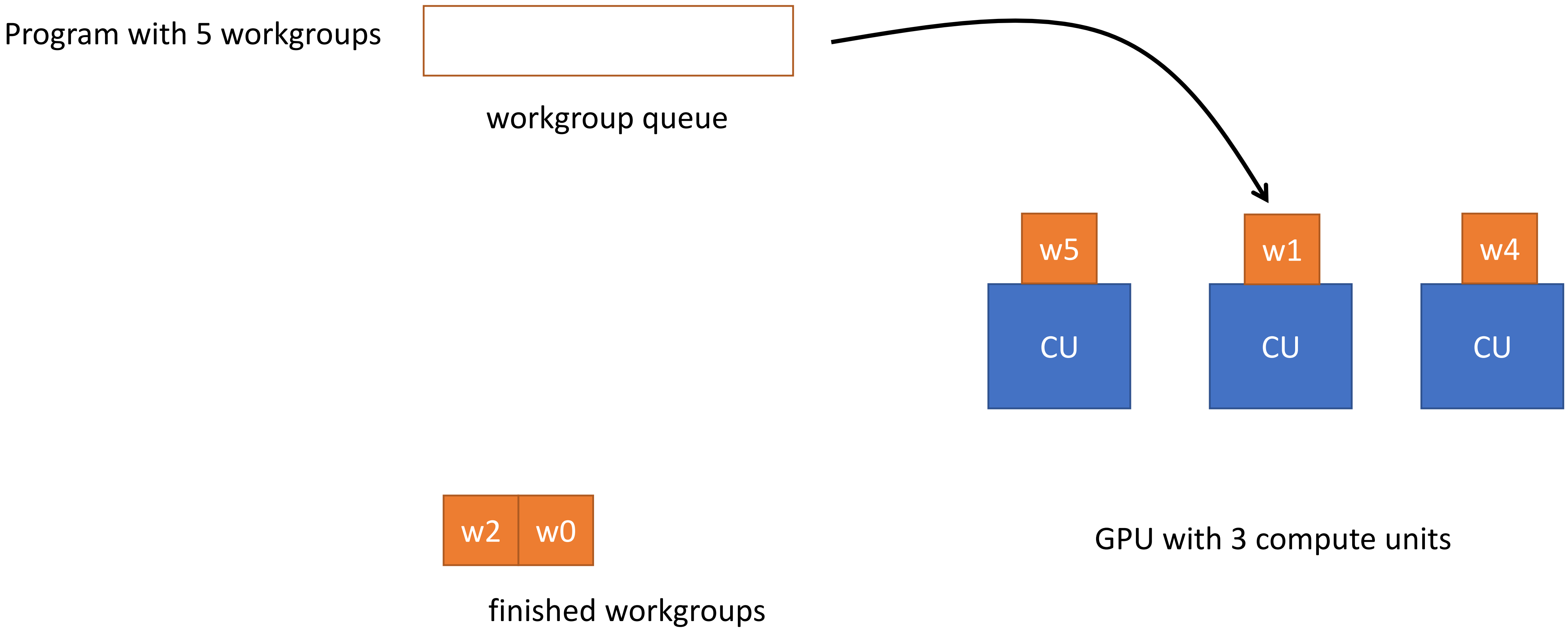
Occupancy bound execution



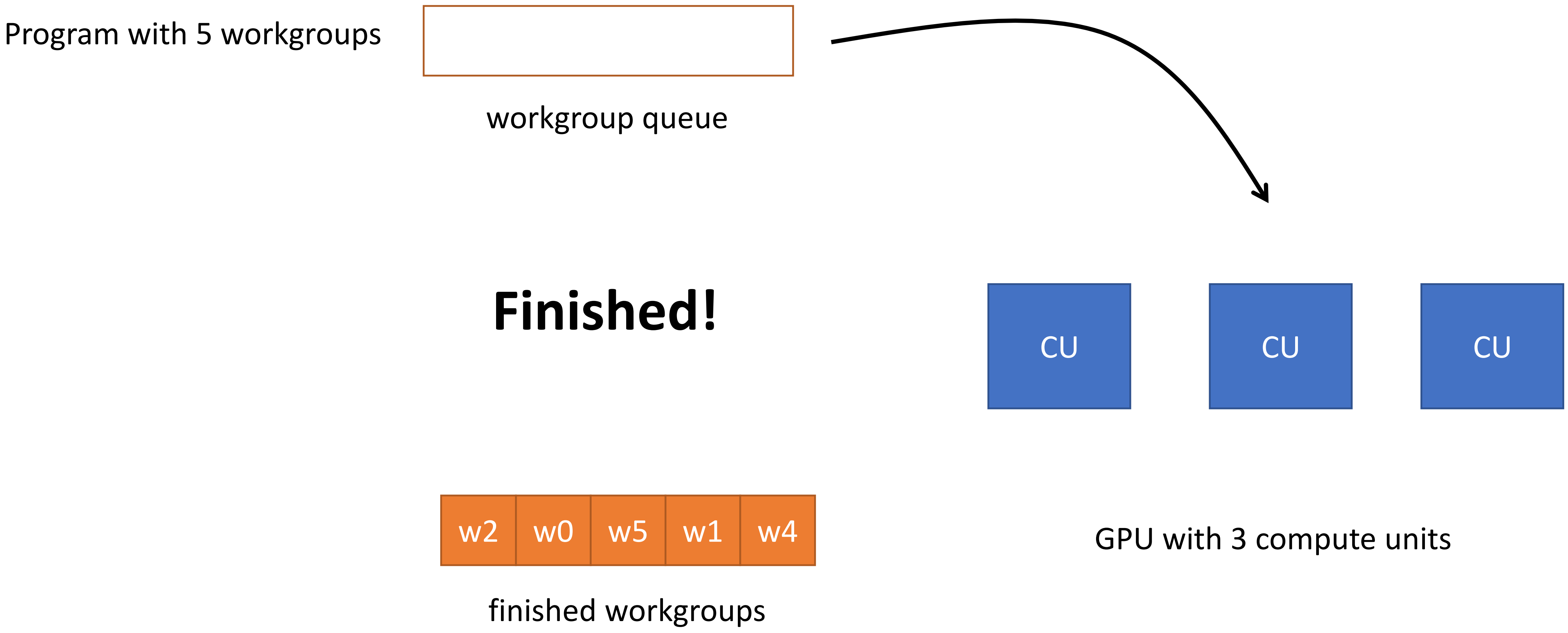
Occupancy bound execution



Occupancy bound execution



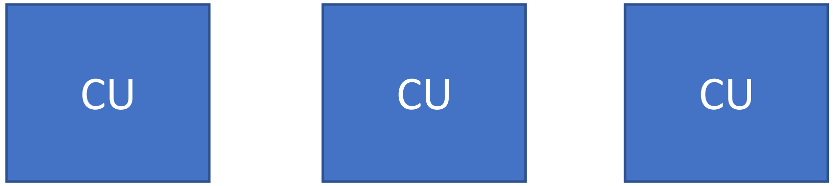
Occupancy bound execution



Occupancy bound execution

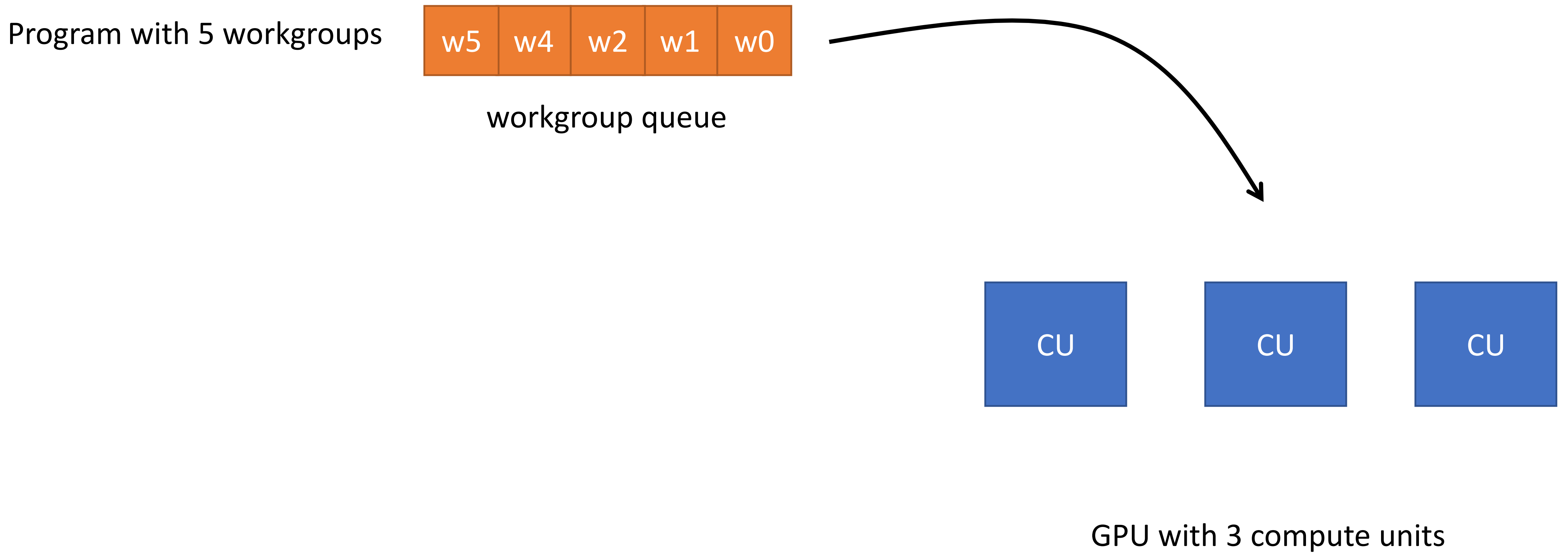
Program with 5 workgroups

workgroup queue



GPU with 3 compute units

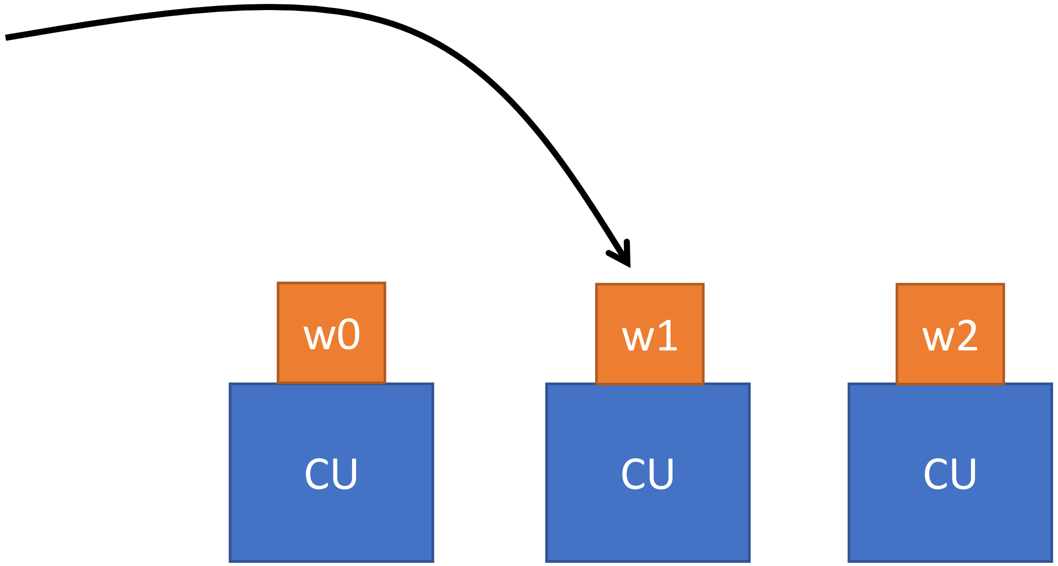
Occupancy bound execution



Occupancy bound execution

Program with 5 workgroups

workgroup queue



Cannot synchronise with workgroups in queue

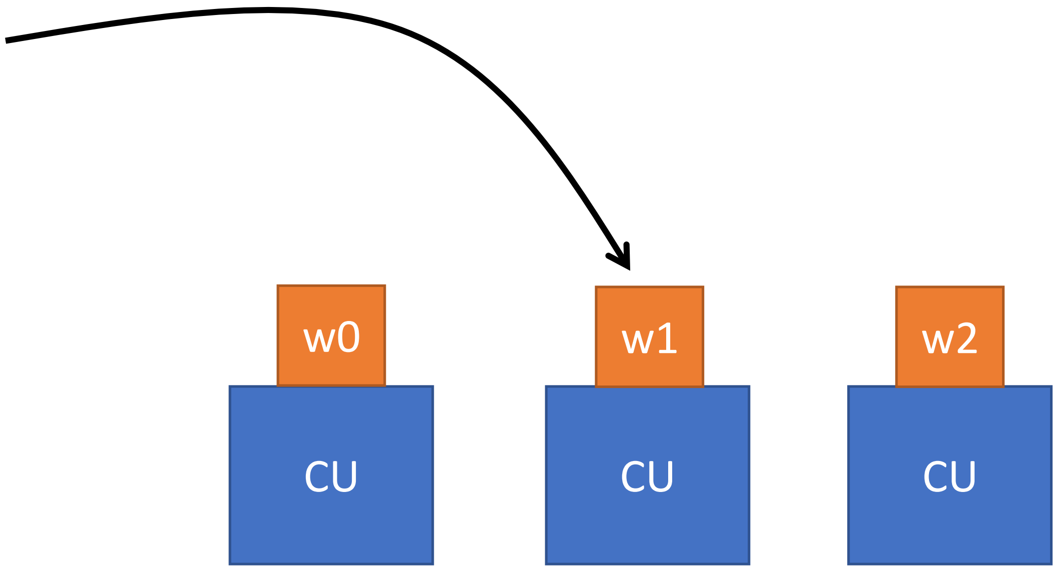
Barrier gives deadlock!

GPU with 3 compute units

Occupancy bound execution

Program with 5 workgroups

workgroup queue



Barrier is possible if
we know the occupancy

GPU with 3 compute units

Occupancy bound execution

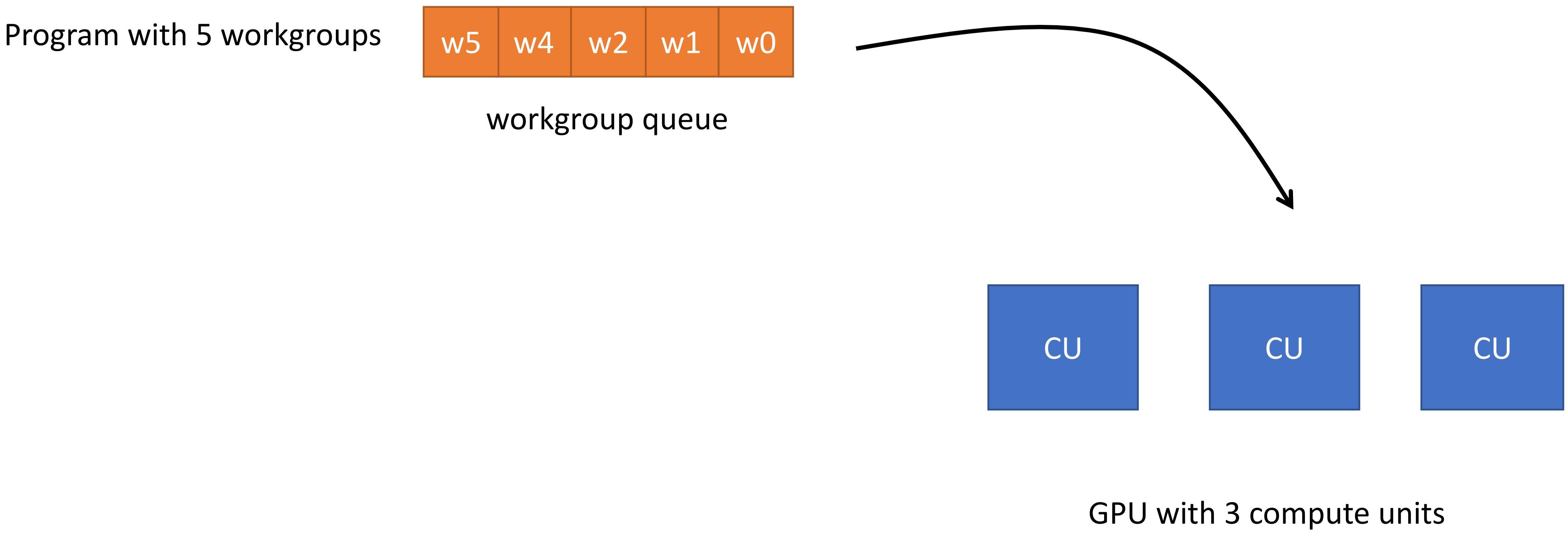
- Launch as many workgroups as compute units?

Occupancy bound execution

- Launch as many workgroups as compute units?

Occupancy bound execution

- Launch as many workgroups as compute units?

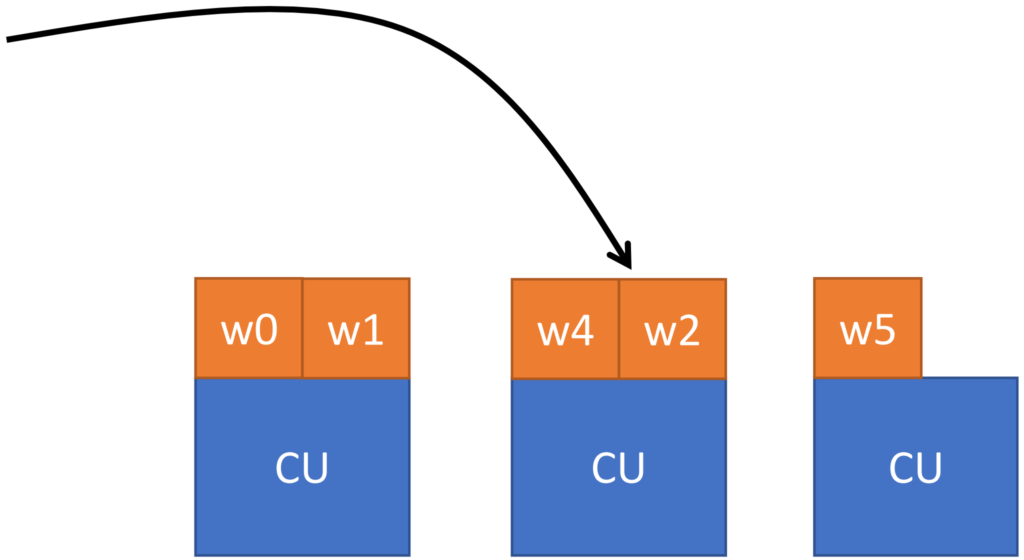


Occupancy bound execution

- Launch as many workgroups as compute units?

Program with 5 workgroups

workgroup queue



Depending on resources, multiple wgs can execute on CU

GPU with 3 compute units

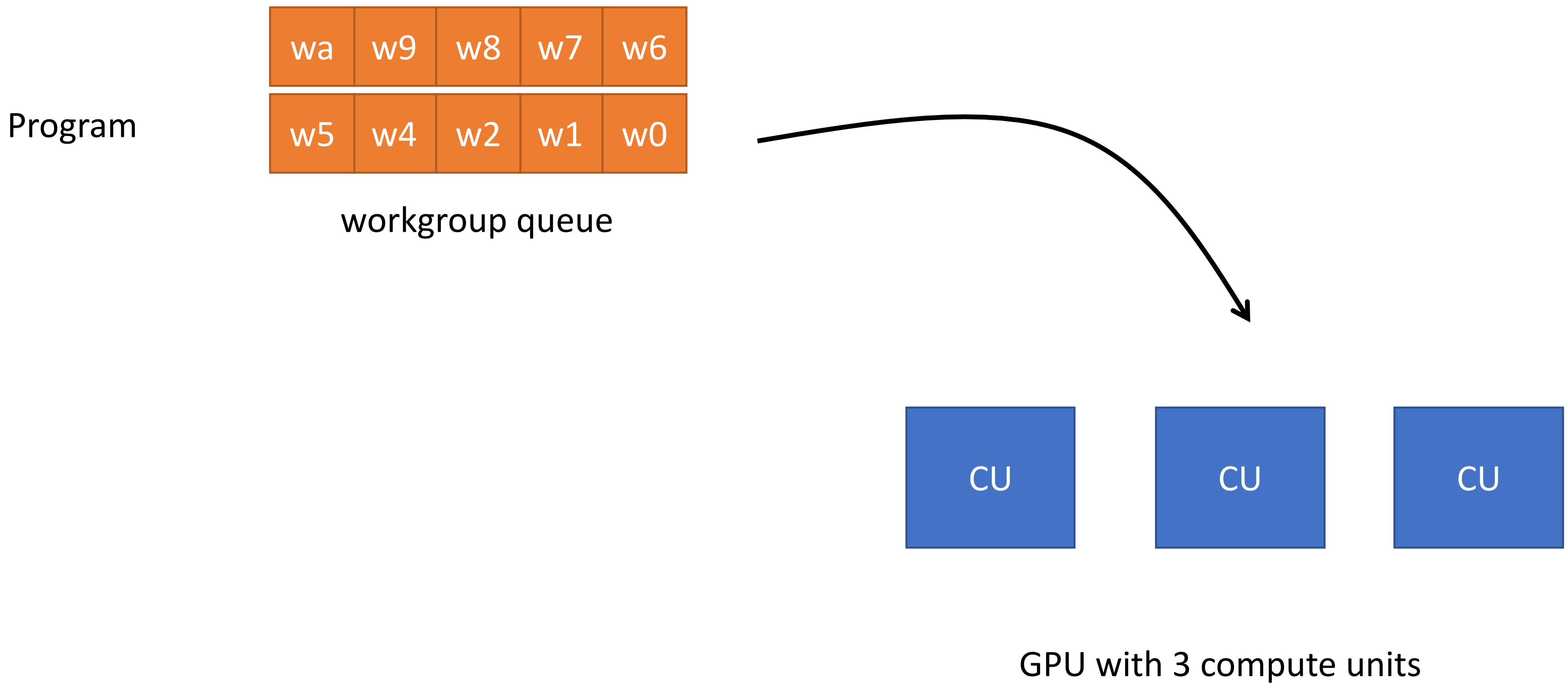
Recall of occupancy discovery

Chip	Compute Units	Occupancy Bound
GTX 980	16	
Quadro K500	12	
Iris 6100	47	
HD 5500	24	
Radeon R9	28	
Radeon R7	8	
T628-4	4	
T628-2	2	

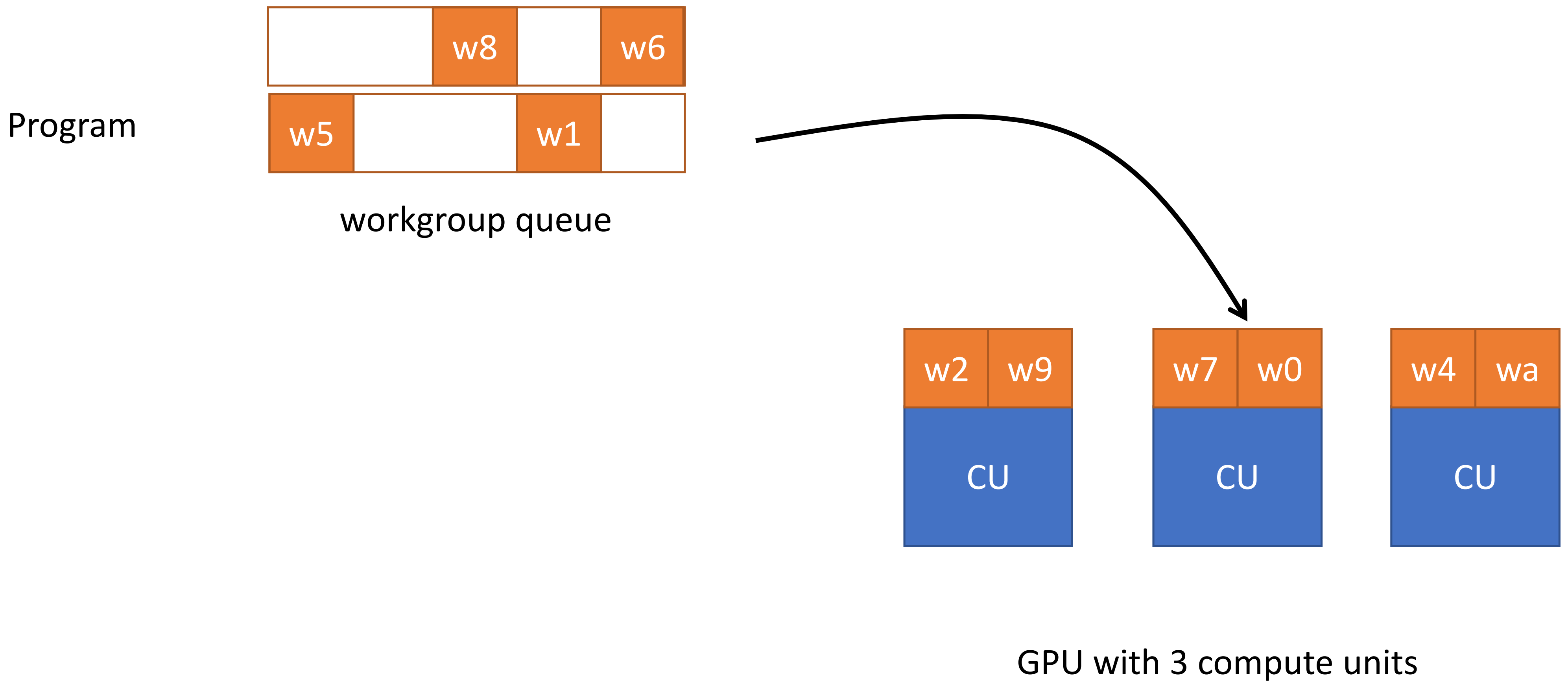
Recall of occupancy discovery

Chip	Compute Units	Occupancy Bound
GTX 980	16	32
Quadro K500	12	12
Iris 6100	47	6
HD 5500	24	3
Radeon R9	28	48
Radeon R7	8	16
T628-4	4	4
T628-2	2	2

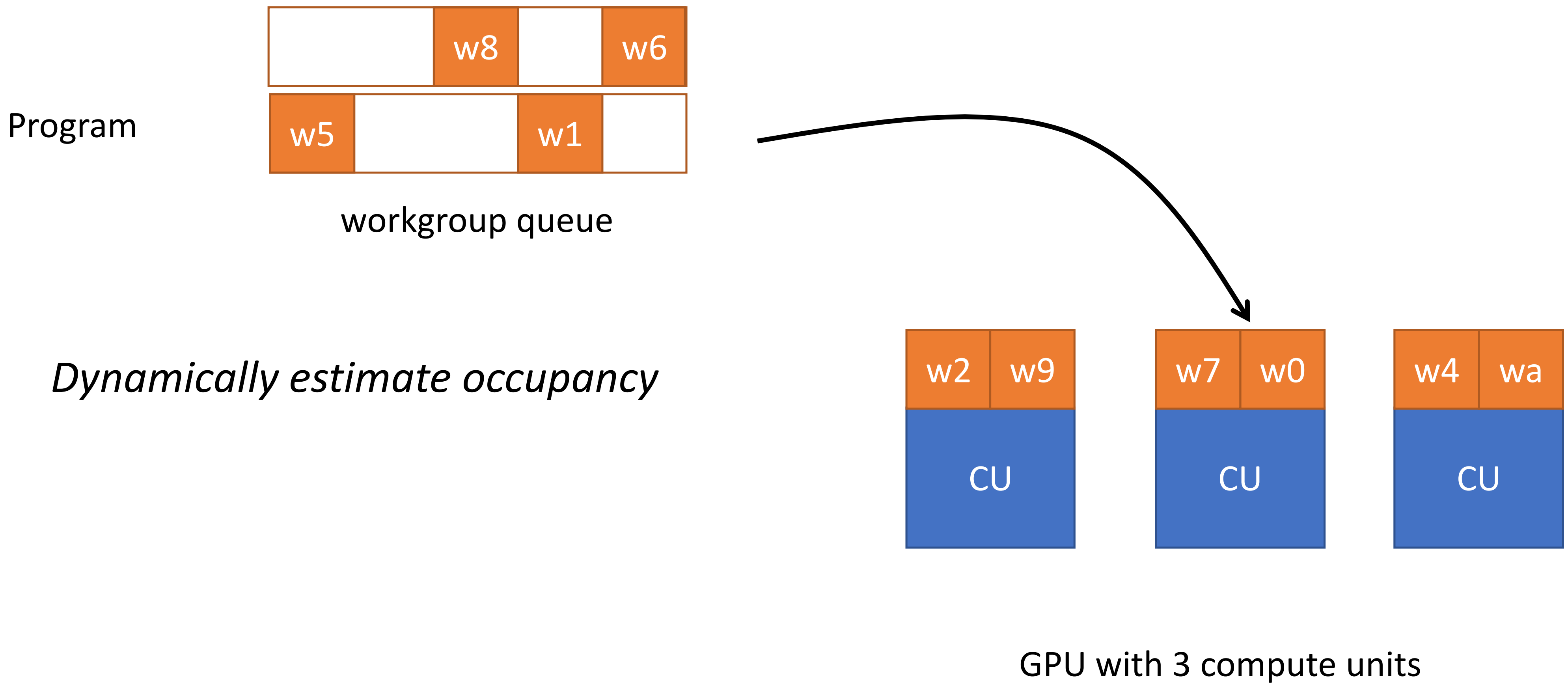
Our approach (scheduling)



Our approach (scheduling)



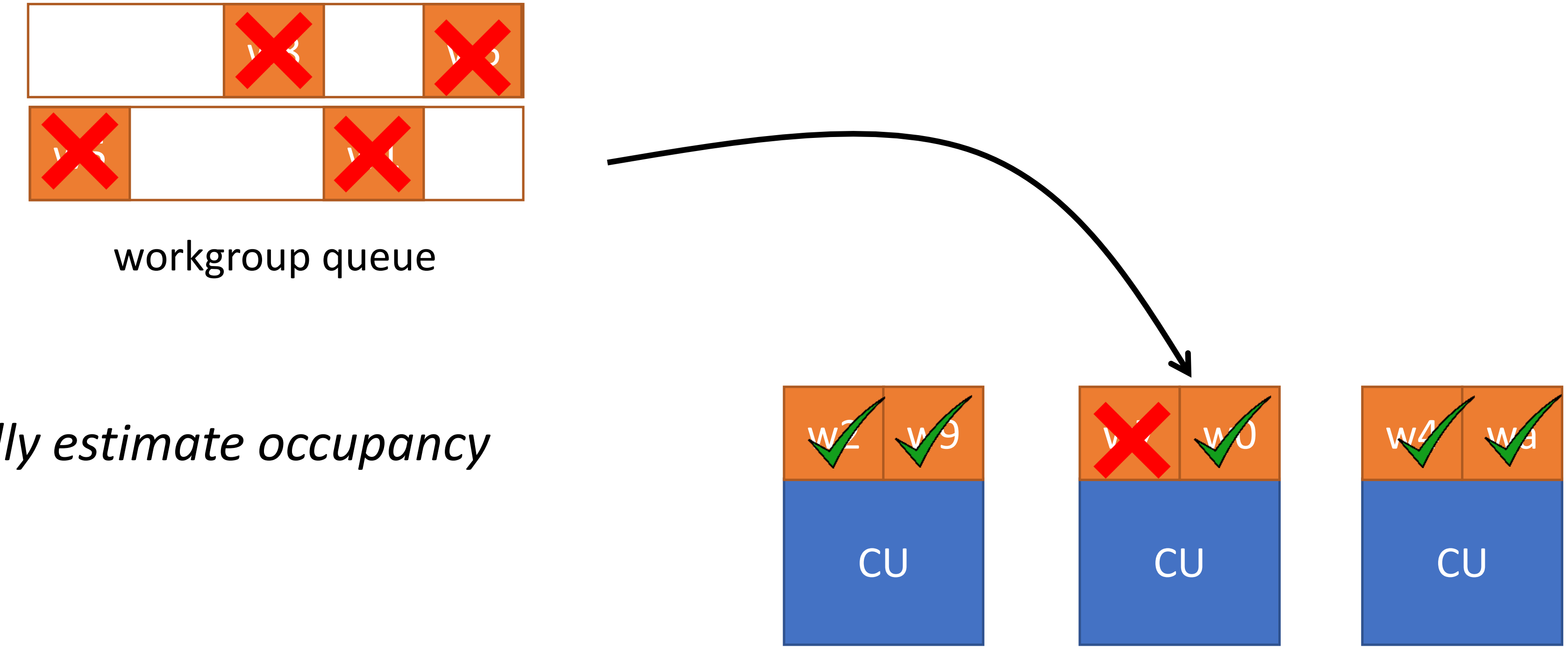
Our approach (scheduling)



Dynamically estimate occupancy

Our approach (scheduling)

Program



workgroup queue

Dynamically estimate occupancy

GPU with 3 compute units

Finding occupant workgroups

- Executed by 1 thread per workgroup
- Two phases:
  - Polling
  - Closing

Finding occupant workgroups

- Executed by 1 thread per workgroup
- Three global variables:
  - Mutex: `m`
  - Bool poll flag: `poll_open`
  - Integer counter: `count`

Finding occupant workgroups

- Executed by 1 thread per workgroup
- Three global variables:
  - Mutex: `m`
  - Bool poll flag: `poll_open`
  - Integer counter: `count`

```
lock(m)
if (poll_open) {
    count++;
    unlock(m);
}
else {
     unlock(m);
     return false;
}

lock(m)
if (poll_open) {
    poll_open = false
}
unlock(m)
return true;
```

Finding occupant workgroups

Polling phase

- Executed by 1 thread per workgroup
- Three global variables:
  - Mutex: `m`
  - Bool poll flag: `poll_open`
  - Integer counter: `count`

```
lock(m)
if (poll_open) {
    count++;
    unlock(m);
}
else {
     unlock(m);
     return false;
}

lock(m)
if (poll_open) {
    poll_open = false
}
unlock(m)
return true;
```

Finding occupant workgroups

Polling phase

```
lock(m)
if (poll_open) {
    count++;
    unlock(m);
}
else {
     unlock(m);
     return false;
}
```

- Executed by 1 thread per workgroup
- Three global variables:
  - Mutex: `m`
  - Bool poll flag: `poll_open`
  - Integer counter: `count`

Closing phase

```
lock(m)
if (poll_open) {
    poll_open = false
}
unlock(m)
return true;
```

Finding occupant workgroups

- Executed by 1 thread per workgroup
- Three global variables:
  - Mutex: `m`
  - Bool poll flag: `poll_open`
  - Integer counter: `count`

```
lock(m)
if (poll_open) {
    count++;
    unlock(m);
}
else {
     unlock(m);
     return false;
}

lock(m)
if (poll_open) {
    poll_open = false
}
unlock(m)
return true;
```

Finding occupant workgroups

- Executed by 1 thread per workgroup
- Three global variables:
  - Mutex: `m`
  - Bool poll flag: `poll_open`
  - Integer counter: `count`

```
lock(m)
if (poll_open) {
    count++;
    unlock(m);
}
else {
     unlock(m);
     return false;
}

lock(m)
if (poll_open) {
    poll_open = false
}
unlock(m)
return true;
```

Let’s implement the discovery protocol

Let’s implement a portable barrier using the discovery protocol

Further reading

Our proposal for a portable inter-workgroup barrier:

- Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, Zvonimir Rakamaric: Portable inter-workgroup barrier synchronisation for GPUs. OOPSLA 2016: 39-58

Our proposal for enabling fair scheduling on GPUs:

- Tyler Sorensen, Hugues Evrard, Alastair F. Donaldson: *Cooperative kernels: GPU multitasking for blocking algorithms*. ESEC/SIGSOFT FSE 2017: 431-441

Summary

- OpenCL provides low-level control over GPU architectural features
- Traditional hierarchical execution model uses barriers to synchronize inside workgroups, with no communication between workgroups
- OpenCL 2.0 provides atomic operations and memory model to facilitate inter workgroup communication
- But the OpenCL execution model provides few guarantees – makes it hard to build reliable concurrency primitives such as barriers

Current research directions

- Theoretical study of execution model hierarchy
- Empirical study of execution model characteristics provided by current GPUs
- Cooperative kernels for GPU multi-tasking (presented at ESEC/FSE tomorrow)