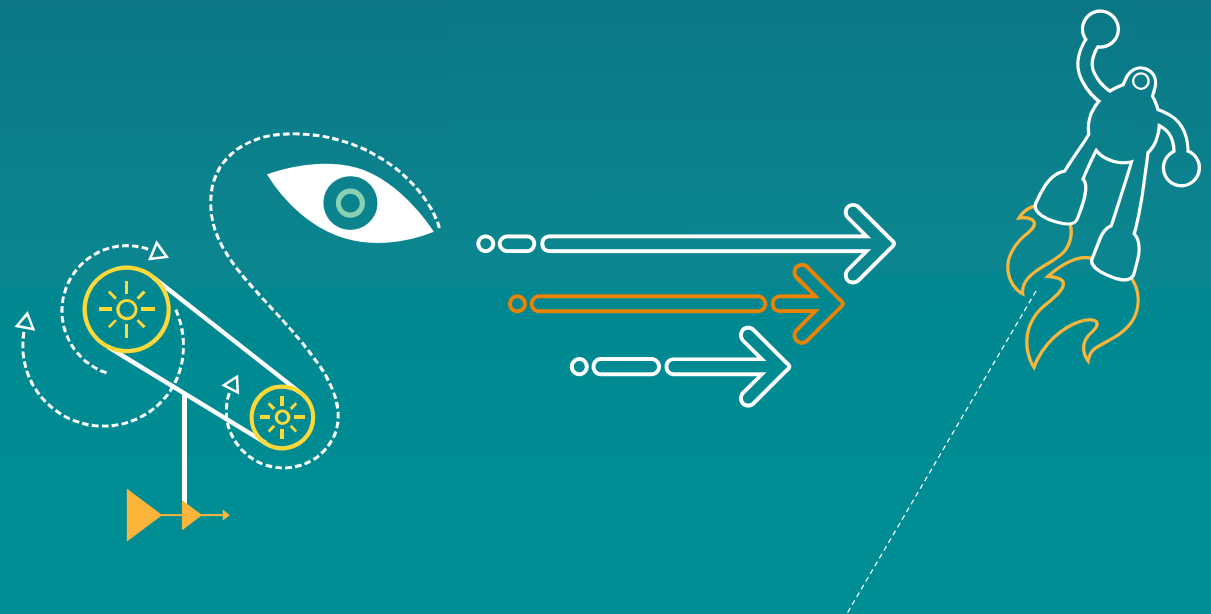
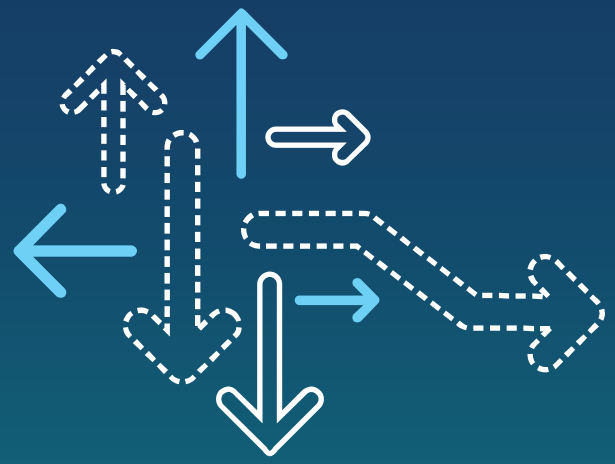
Lee Howes
2015-02-07, Qualcomm Technologies Inc.

Dark Secrets: Heterogeneous Memory Models

QUALCOMM®



For more information on Qualcomm, visit us at: www.qualcomm.com & www.qualcomm.com/blog



Introduction – why memory consistency

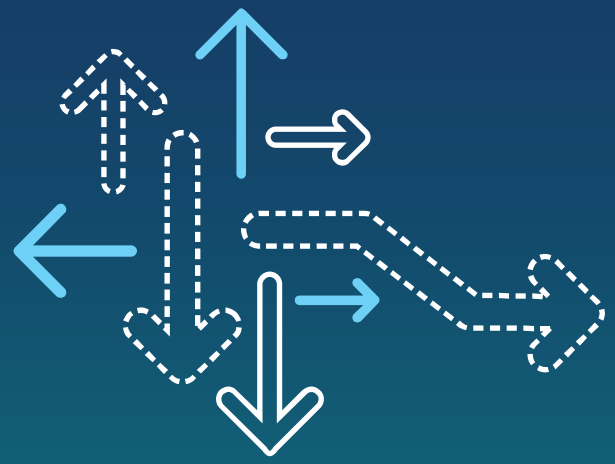
Introduction | Current restrictions | Basics of OpenCL 2.0 | Heterogeneity in OpenCL 2.0 | Heterogeneous memory ordering | Summary

Weak memory models

- Many programming languages have coped with weak memory models
- These are fixed in various ways:
  - Platform-specific rules
  - Conservative compilation behavior
- A clear memory model allows developers to understand their program behavior!
  - Or part of it anyway. It's not the only requirement but it is a start.
- Many current models are based on the Data-Race-Free work
  - Special operations are used to maintain order
  - Between special instructions reordering is valid for efficiency

Better memory models

- Many languages have tried to formalize their memory models
  - Java, C++, C…
  - Why?
- OpenCL and HSA are not exceptions
  - Both have developed models based on the DRF work, via C11/C++11
- Ben Gaster, Derek Hower and I have collaborated on HRF-Relaxed
  - An extension of DRF models for heterogeneous systems
  - To appear in ACM TACO
- Unfortunately, even a stronger memory model doesn't solve all your problems…



Current restrictions

Introduction | **Current restrictions** | Basics of OpenCL 2.0 | Heterogeneity in OpenCL 2.0 | Heterogeneous memory ordering | Summary

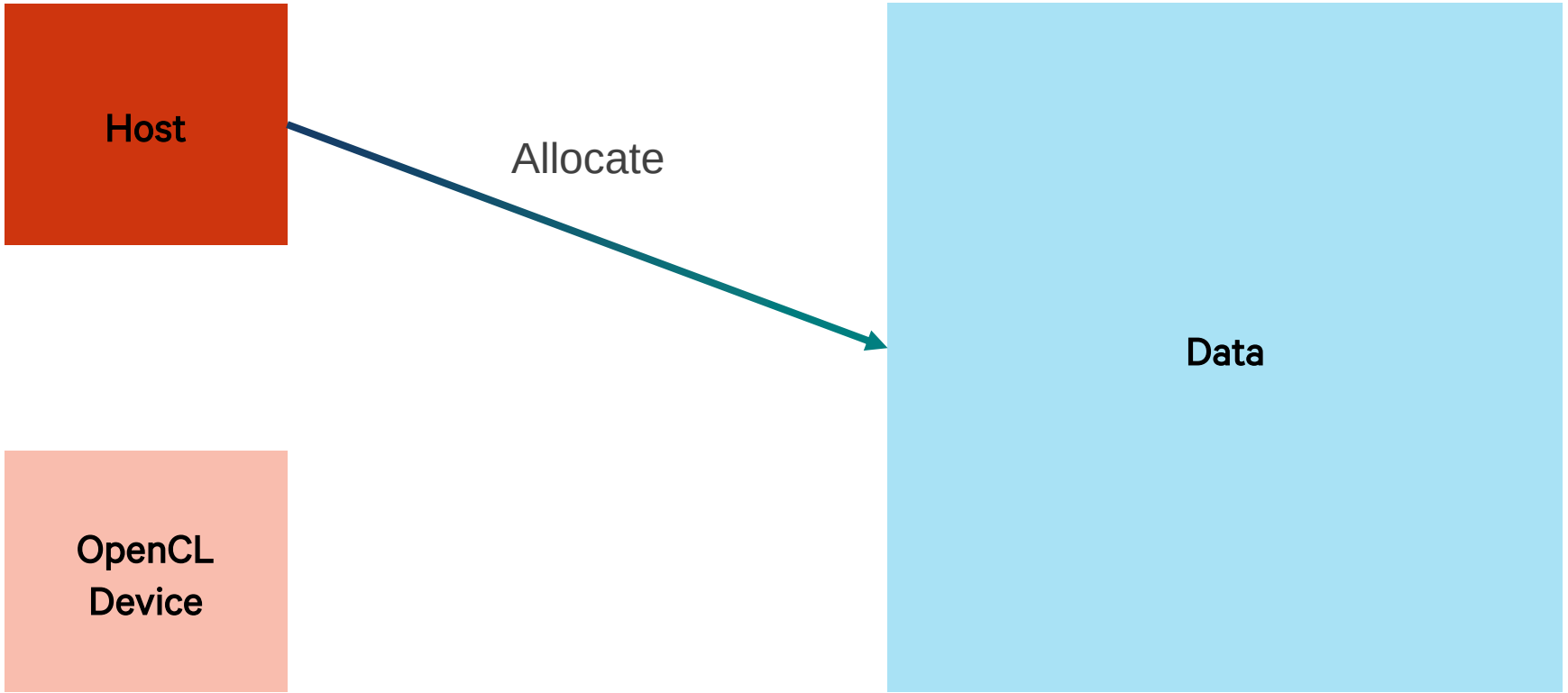
Weakness in the OpenCL 1.x memory model

- Three specific issues:
  - Coarse grained access to data
  - Separating addressing between devices and the host process
  - Weak and poorly defined ordering controls

- Any of these can be worked around with vendor extensions or device knowledge

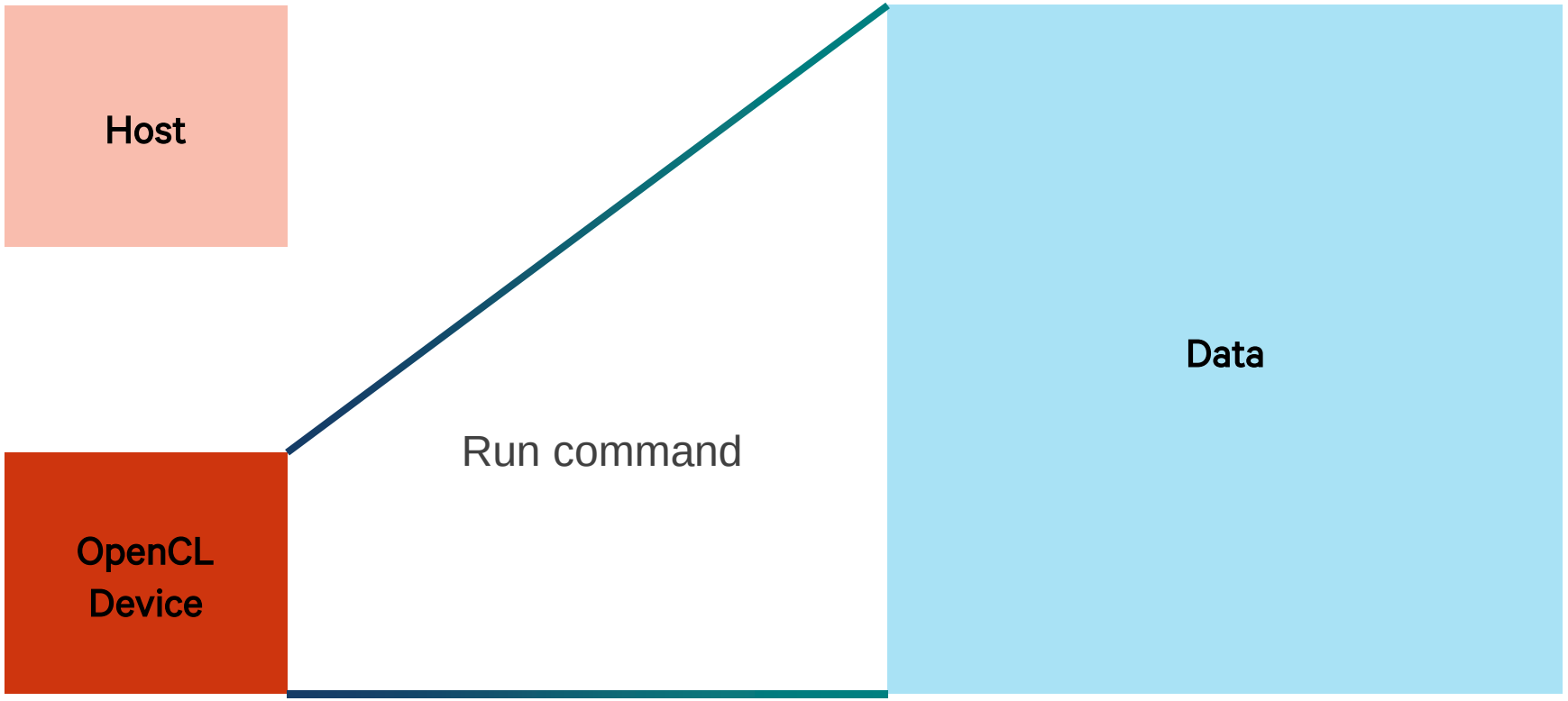
Coarse host->device synchronization

- Host controls data



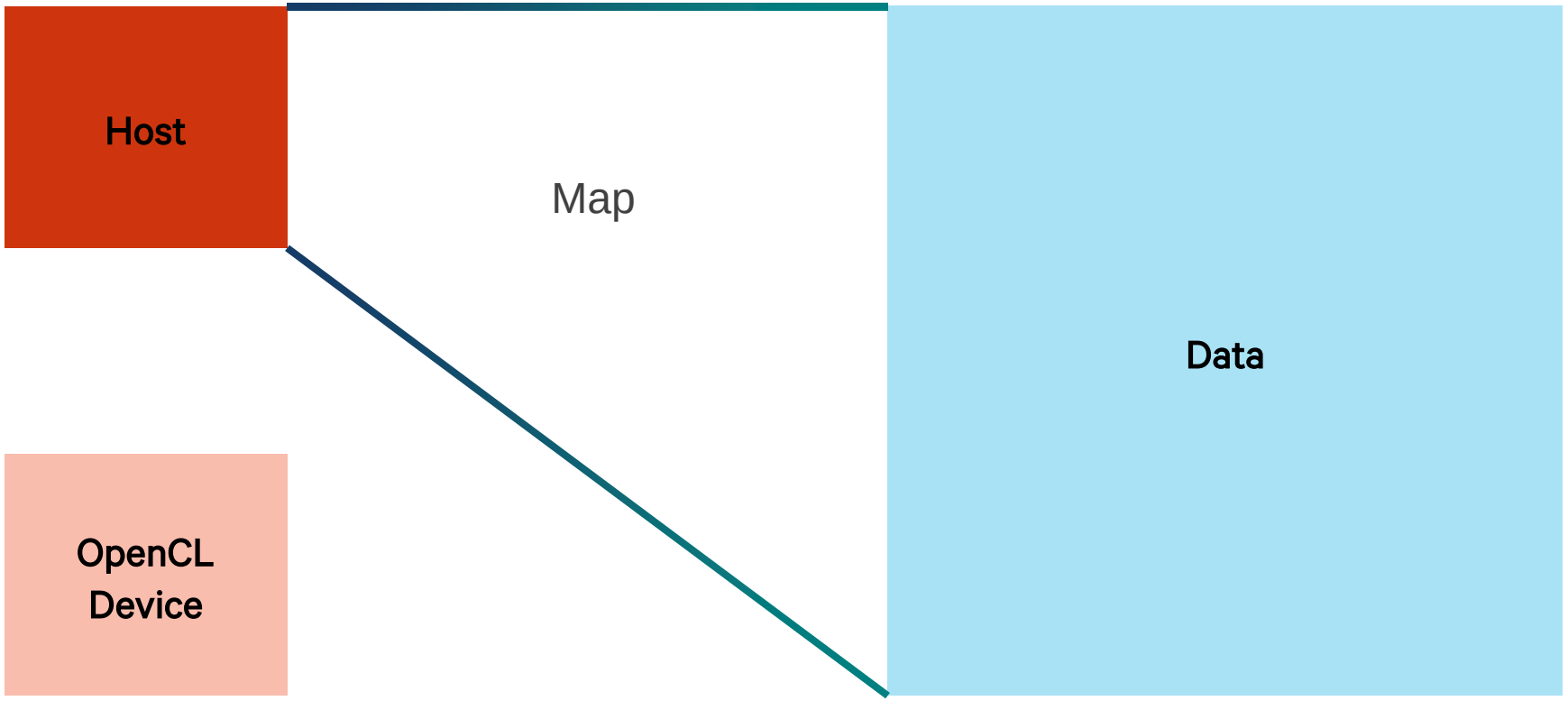
Coarse host->device synchronization

- A single running command owns an entire allocation

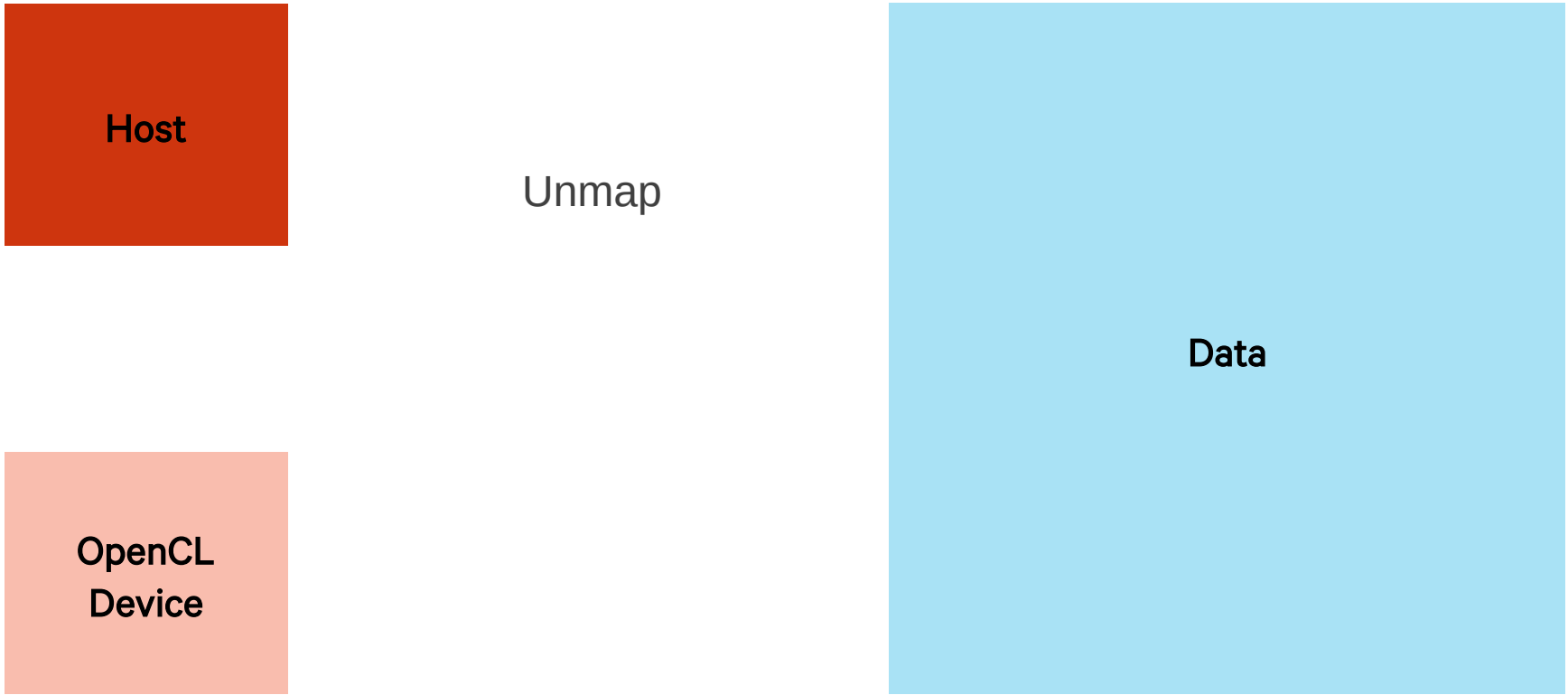


Coarse host->device synchronization

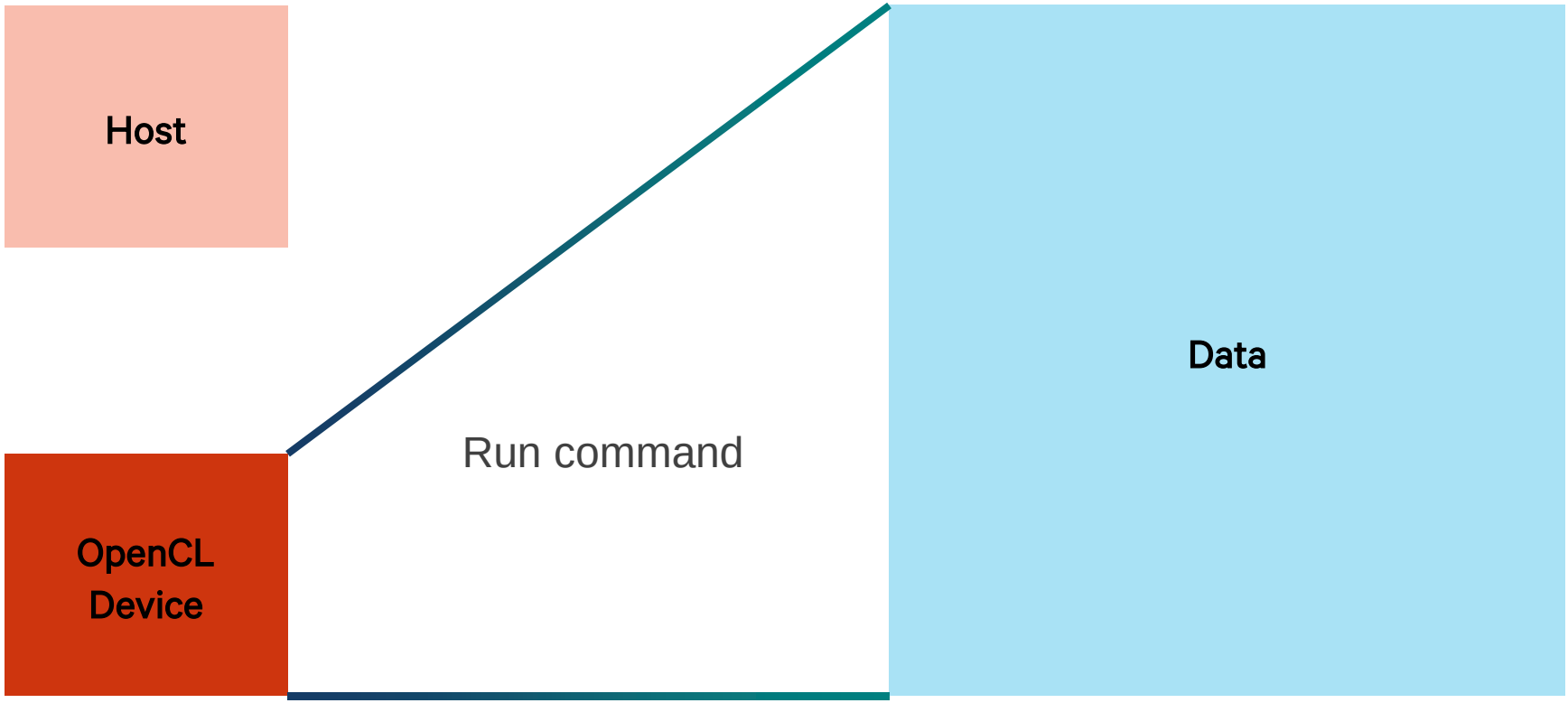
- The host can access it using map/unmap operations



Coarse host->device synchronization

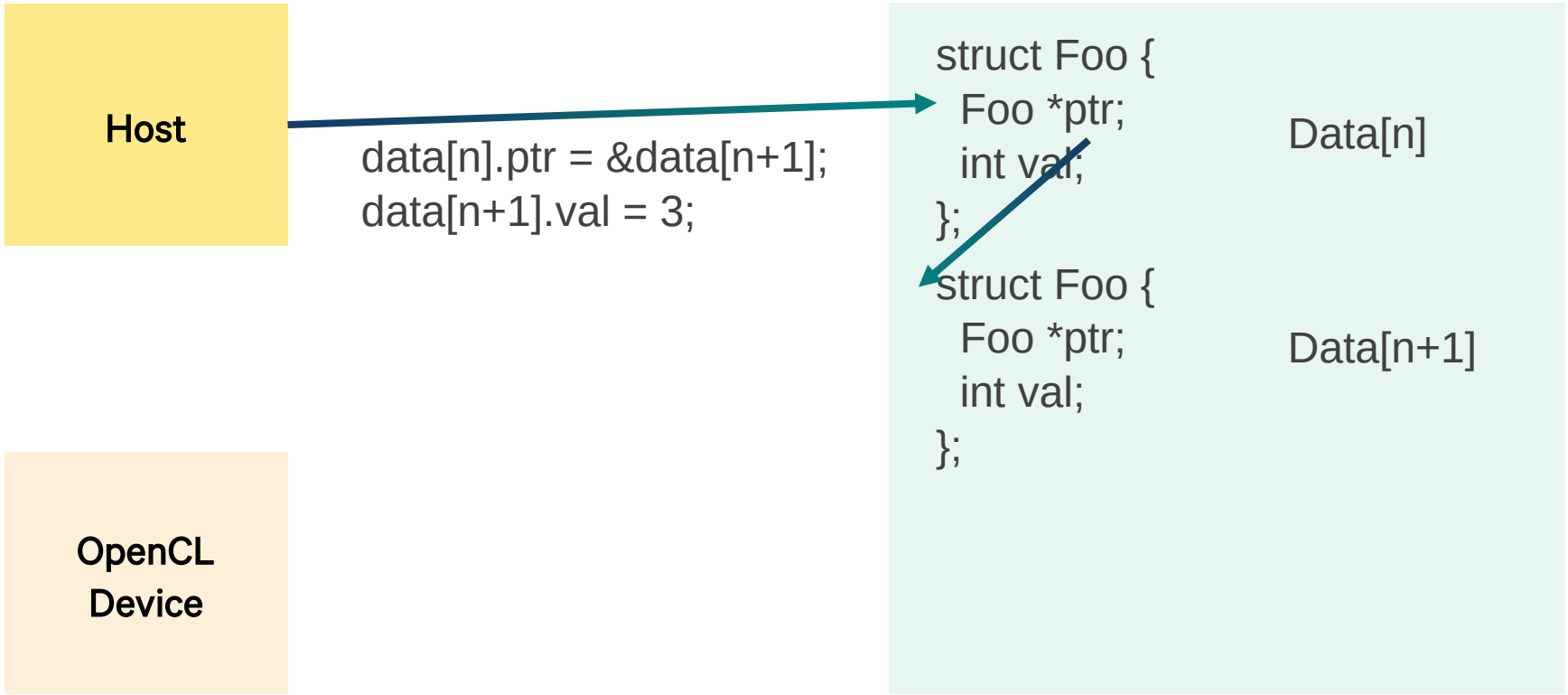


Coarse host->device synchronization



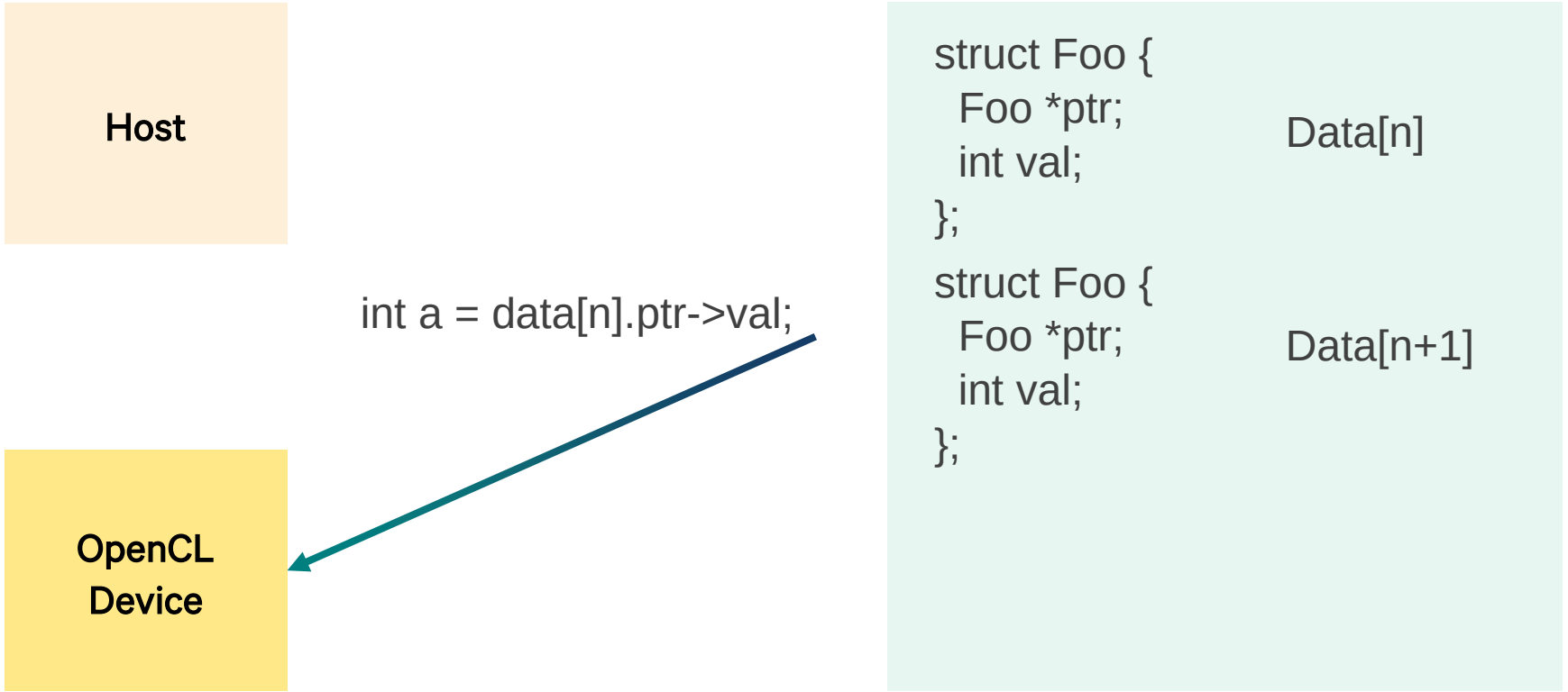
Separate addressing

We can update memory with a pointer



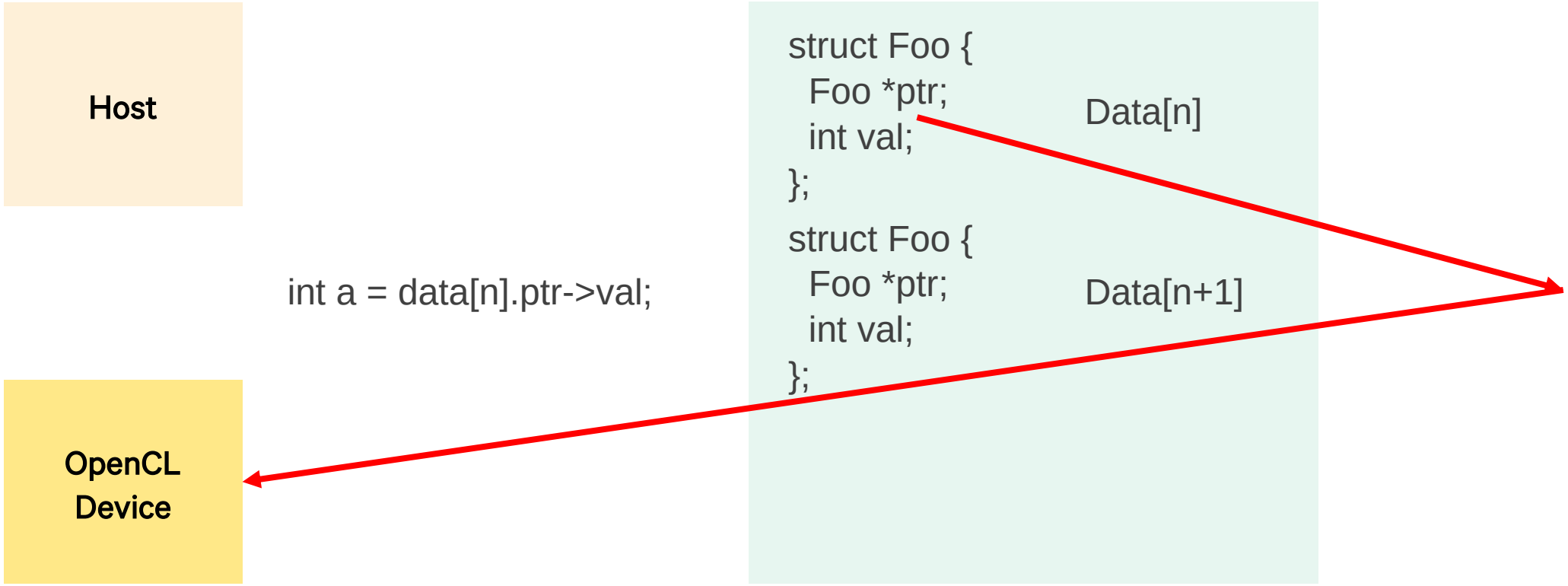
Separate addressing

We try to read it – but what is the value of **a**?



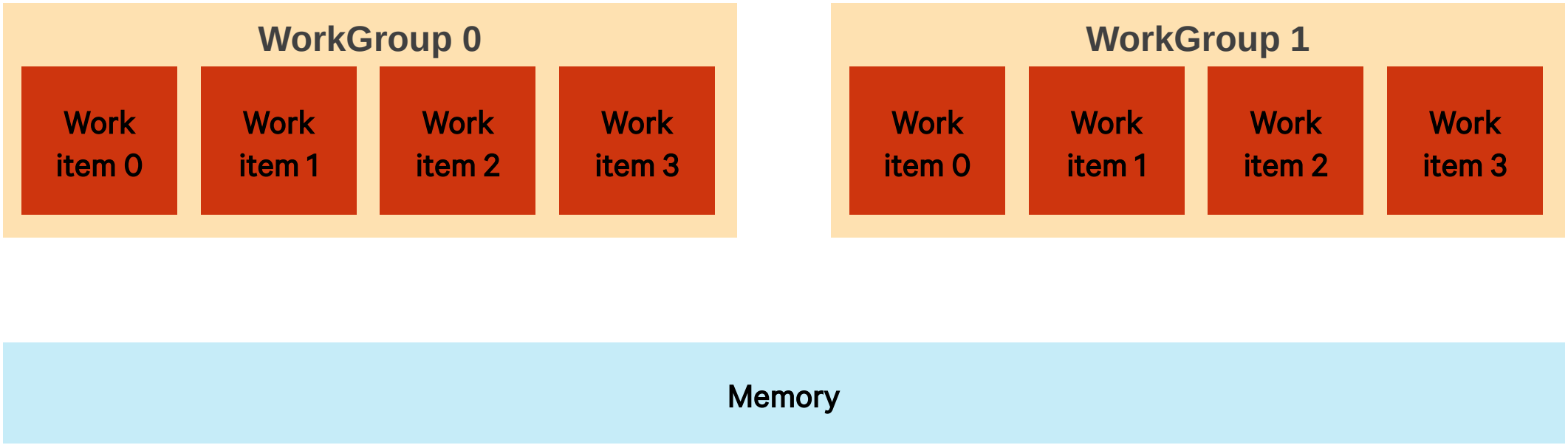
Separate addressing

Unfortunately, the address of **data** may change



Bounds on visibility

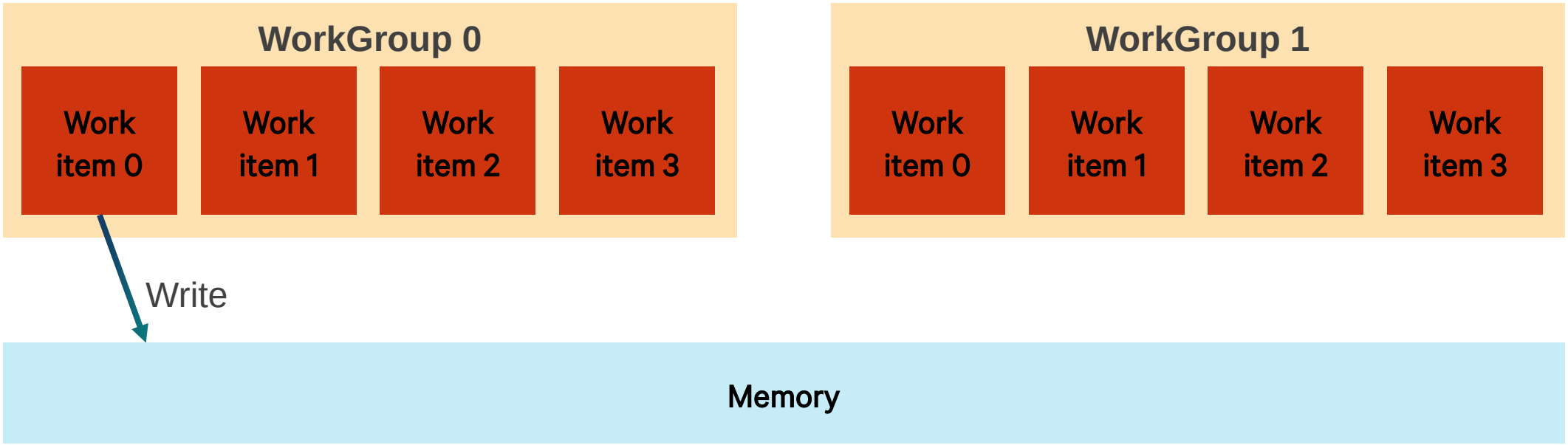
Controlling memory ordering is challenging



Bounds on visibility

Within a group we can synchronize

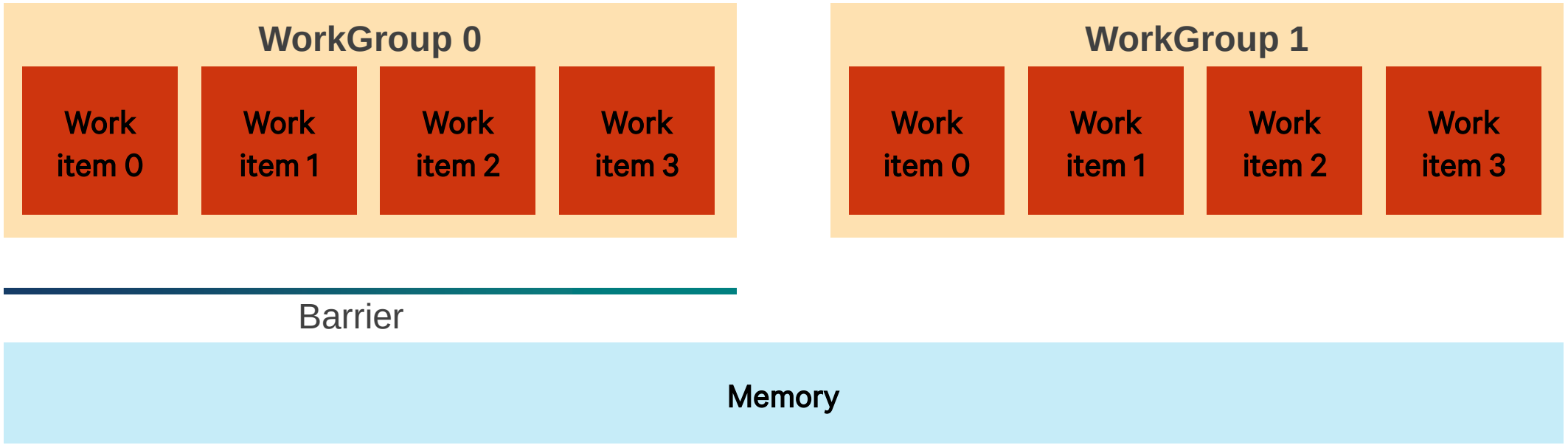
- Physically, this probably means within a single core



Bounds on visibility

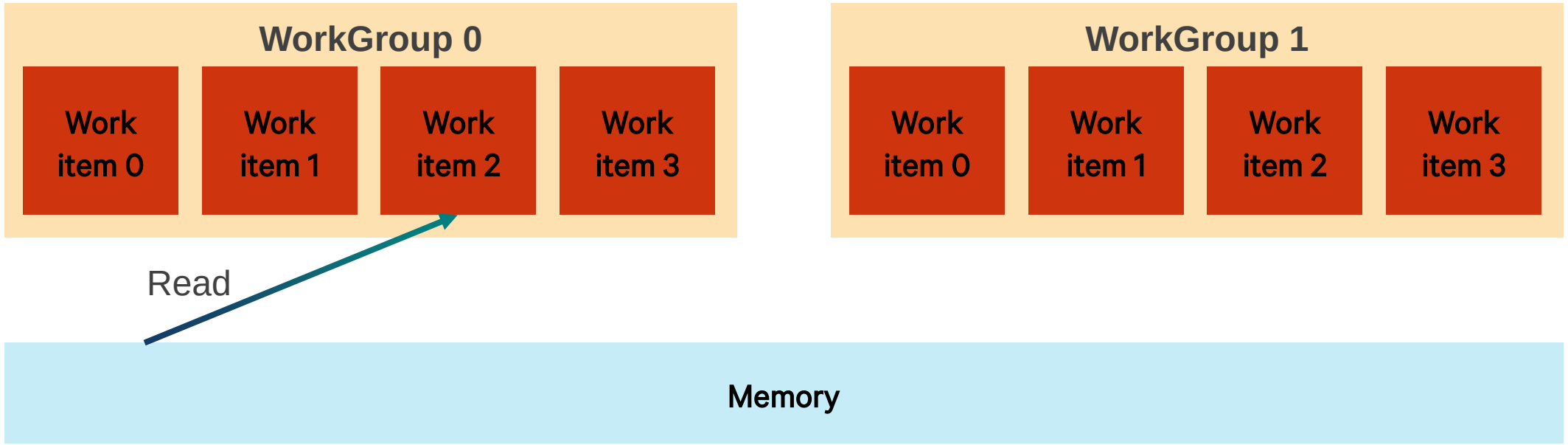
Within a group we can synchronize

- Barrier operations synchronize active threads and constituent work-items



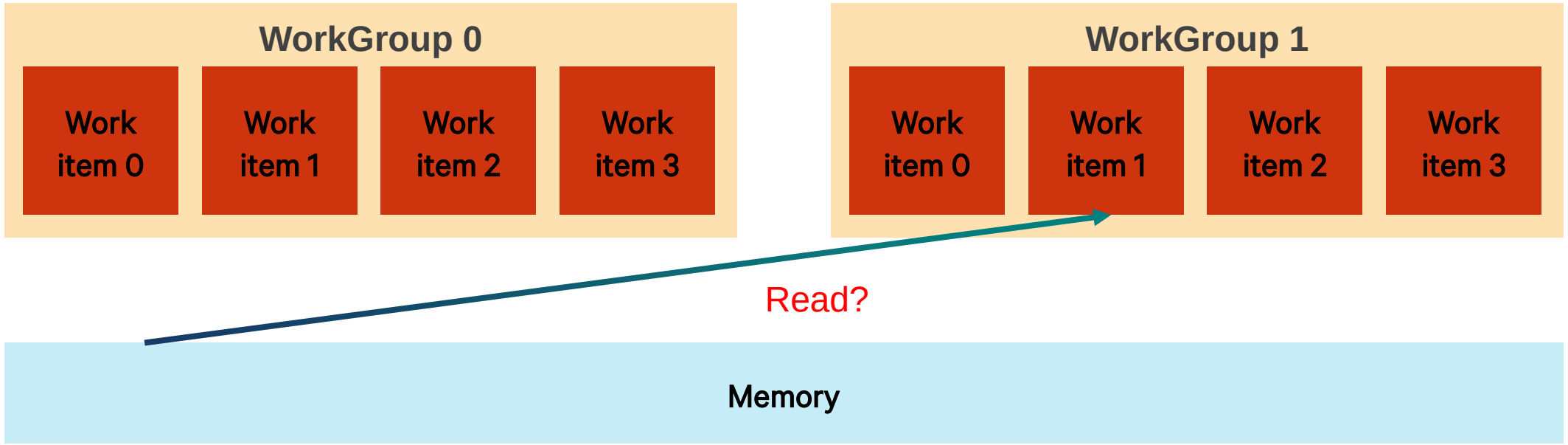
Bounds on visibility

Within a group we can synchronize



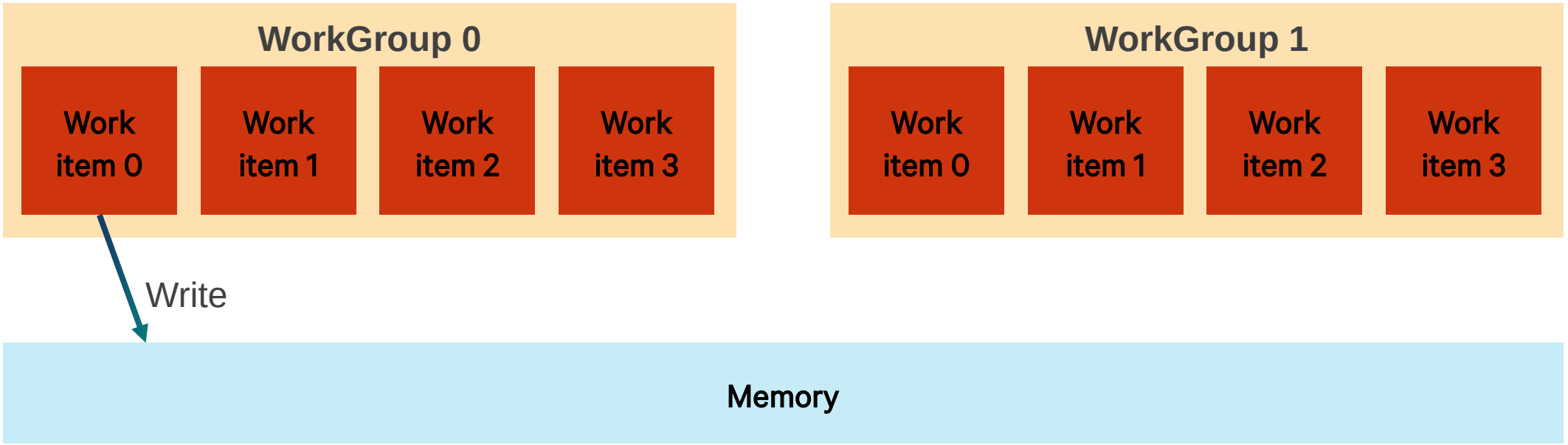
Bounds on visibility

Between groups we can't



Bounds on visibility

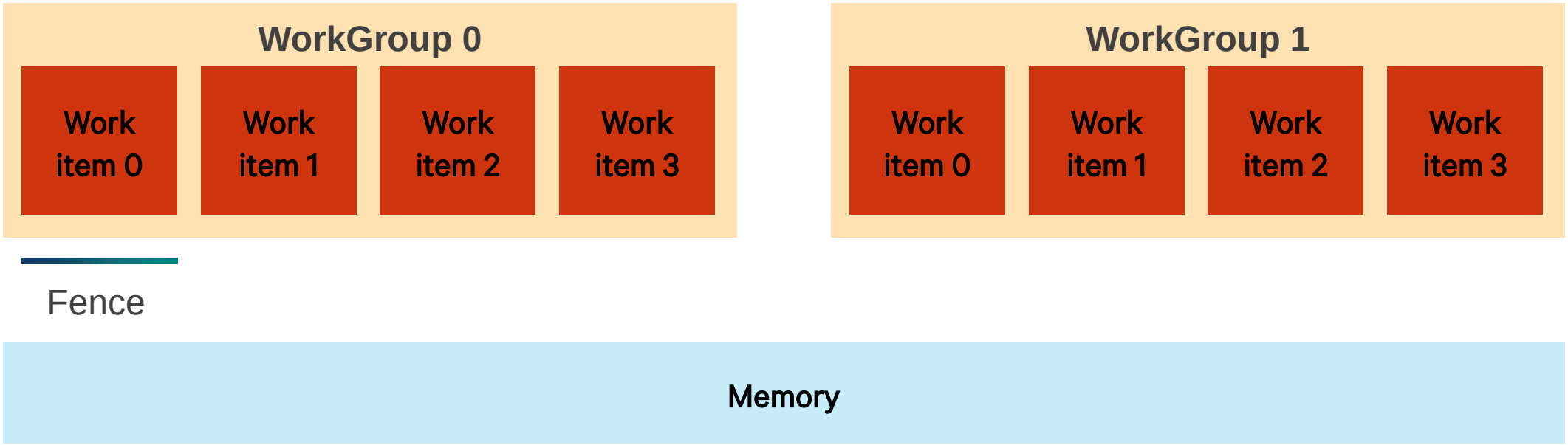
What if we use fences?



Bounds on visibility

What if we use fences?

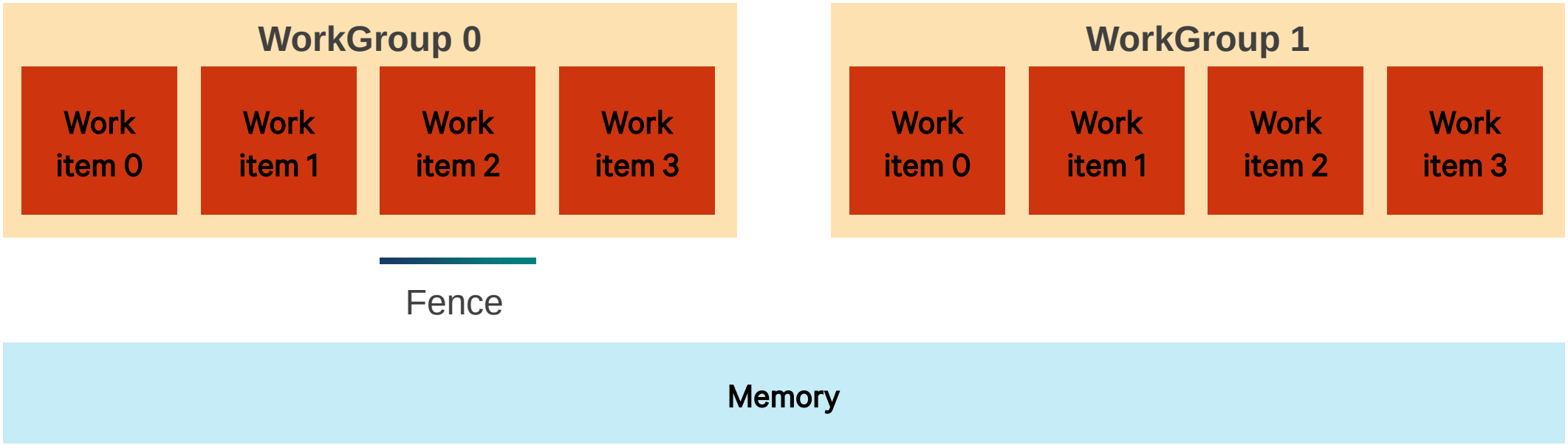
- Ensure the write completes for a given work-item



Bounds on visibility

What if we use fences?

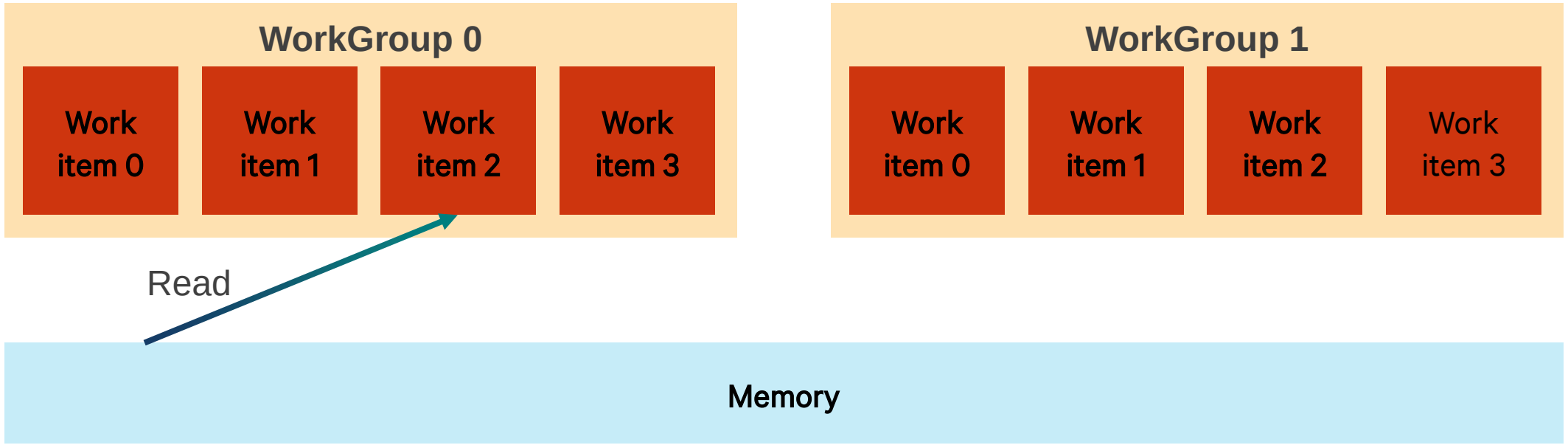
- Fence at the other end



Bounds on visibility

Within a group we can synchronize

- Ensure a read is after the fence

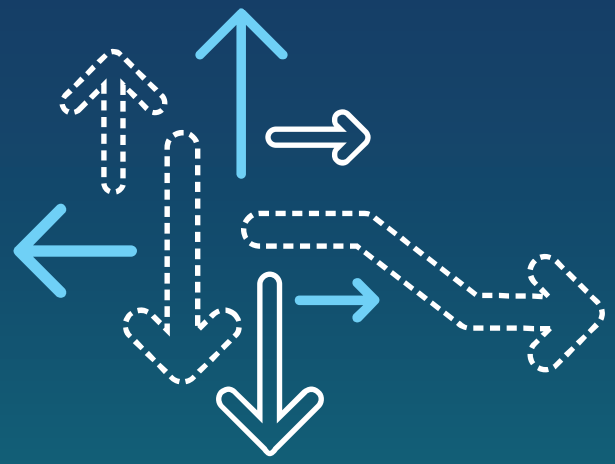


Meaning of fences

When did the fence happen?

- **Probably** seeing a write that was written after a fence guarantees that the fence completed
  - The spec is not very clear on this
- There is no coherence guarantee
  - Need the write ever complete?
  - If it doesn't complete, who can see it, who can know that the fence happened?
- Can the flag be updated without a race?
  - For that matter, what is a race?
- Spliet et al: KMA.
  - Weak ordering differences between platforms due to poorly defined model.

```
{{
  data[n] = value;
  fence(…);
  flag = trigger;
||
  if(flag) {
    fence(…);
    value = data[n];
}}
```

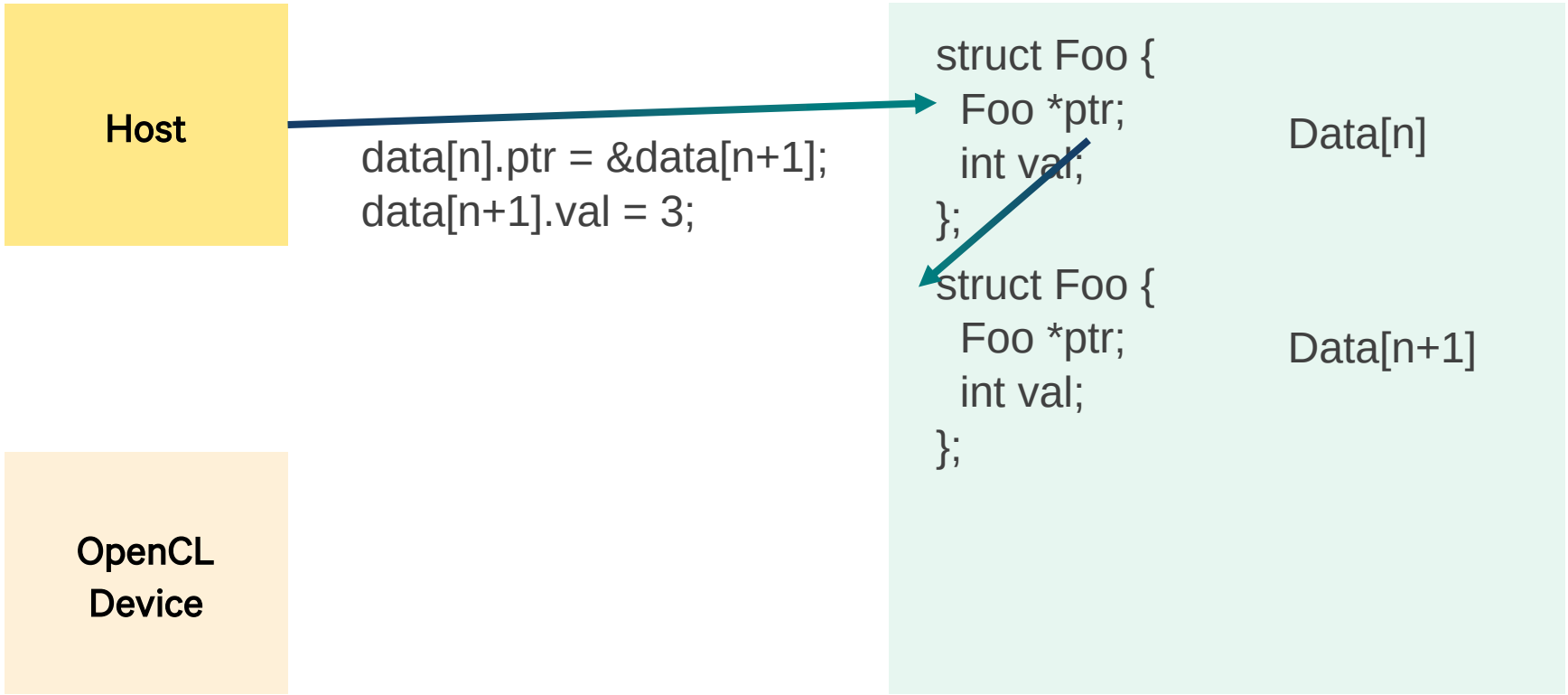


Basics of OpenCL 2.0

Introduction | Current restrictions | **Basics of OpenCL 2.0** | Heterogeneity in OpenCL 2.0 | Heterogeneous memory ordering | Summary

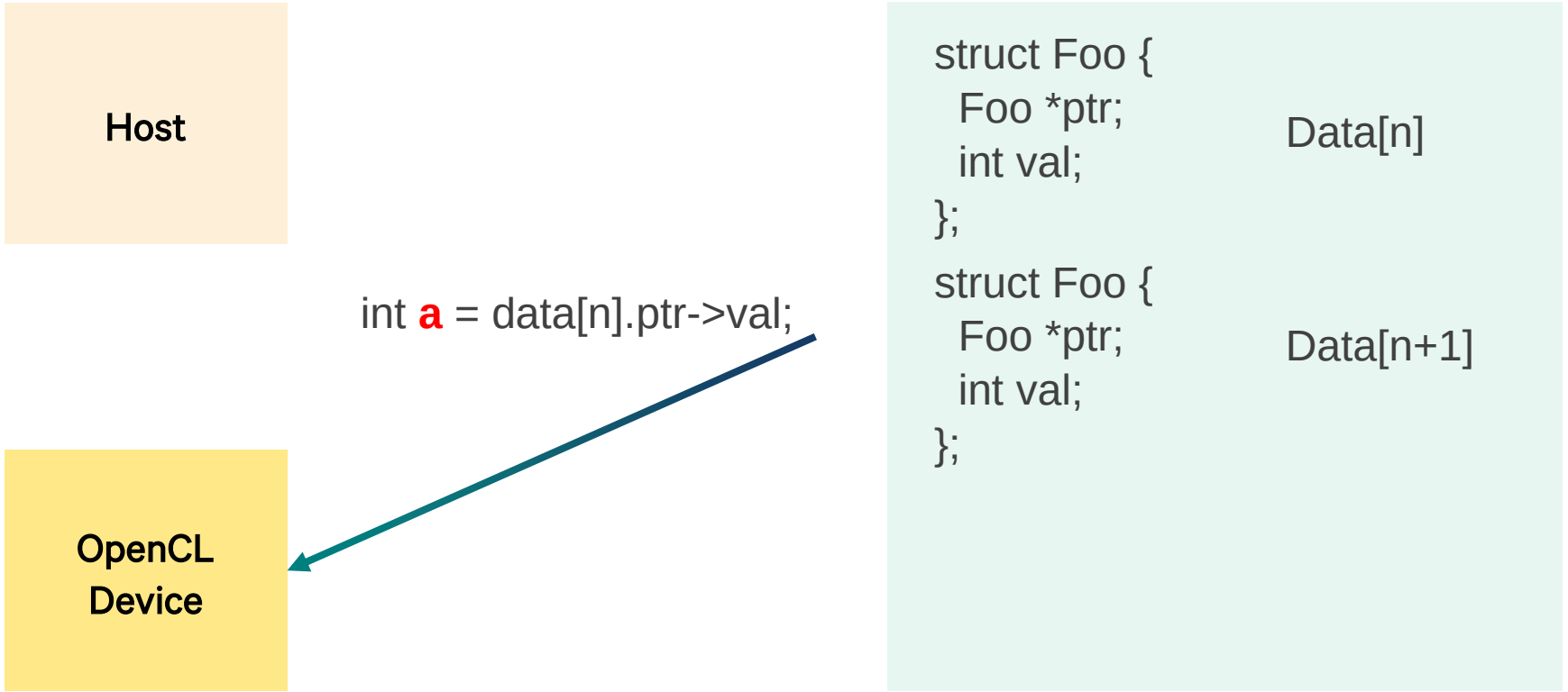
Sharing virtual addresses

We can update memory with a pointer



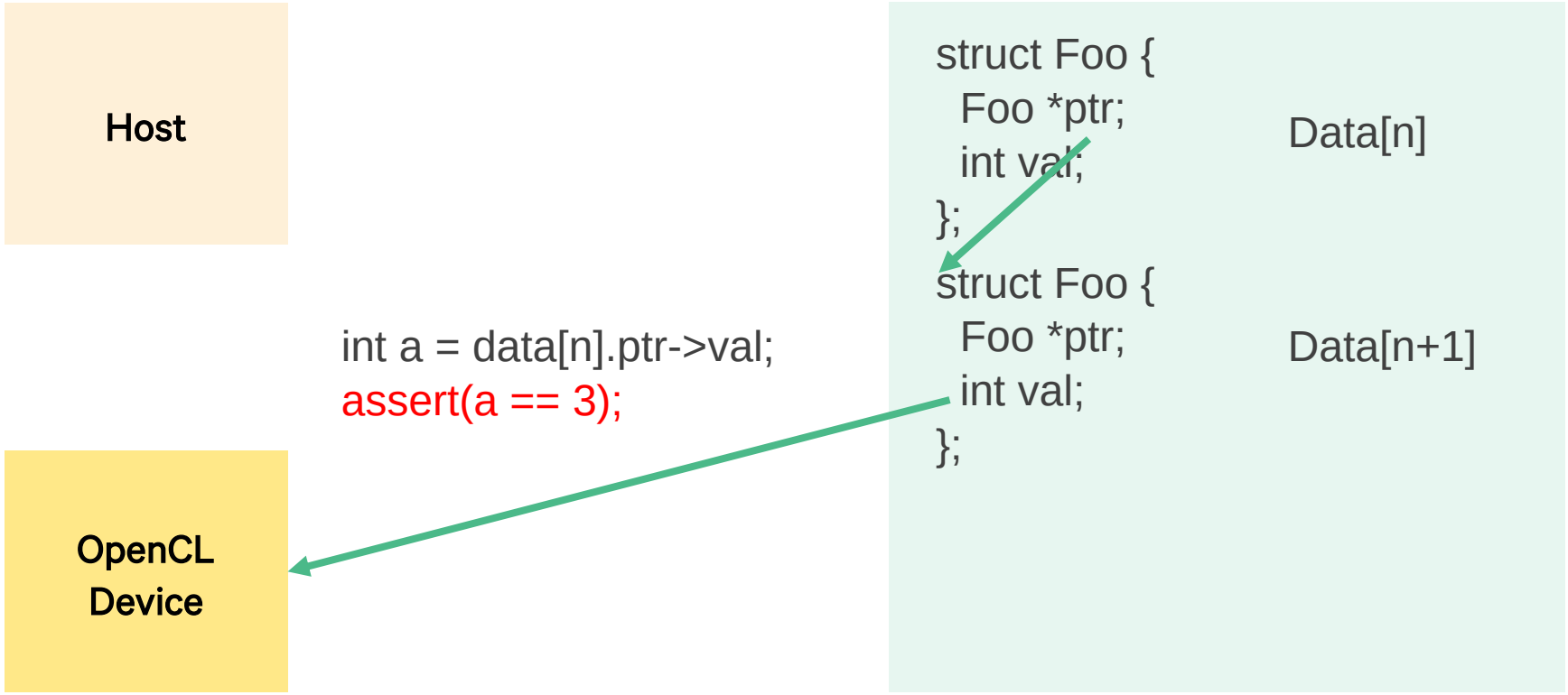
Sharing virtual addresses

We try to read it – but what is the value of **a**?



Sharing virtual addresses

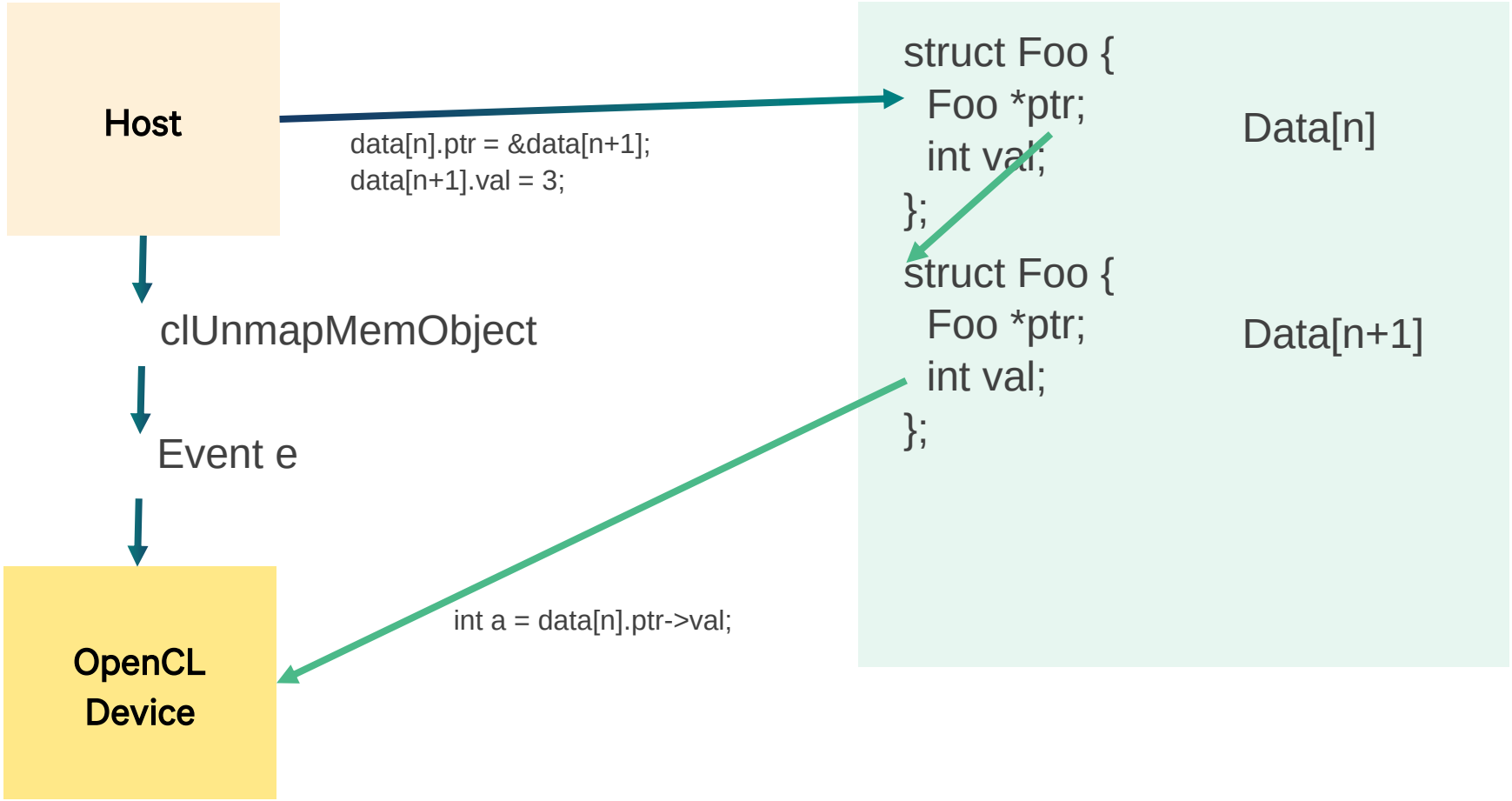
Now the address does not change!



Sharing data – **when** does the value change?

Coarse grained

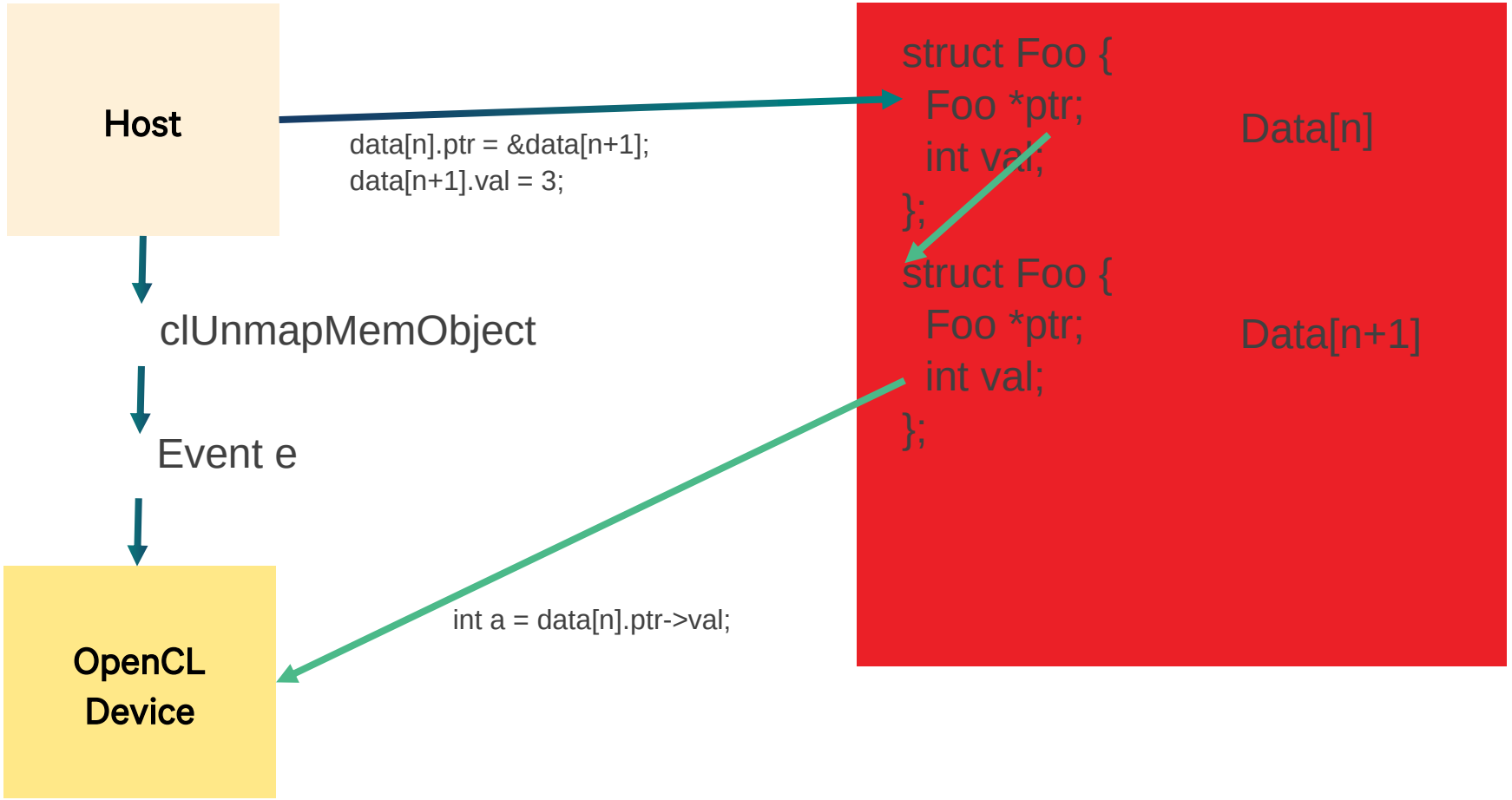
- Unmap on the host, event dependency on device



Sharing data – when does the value change?

Coarse grained

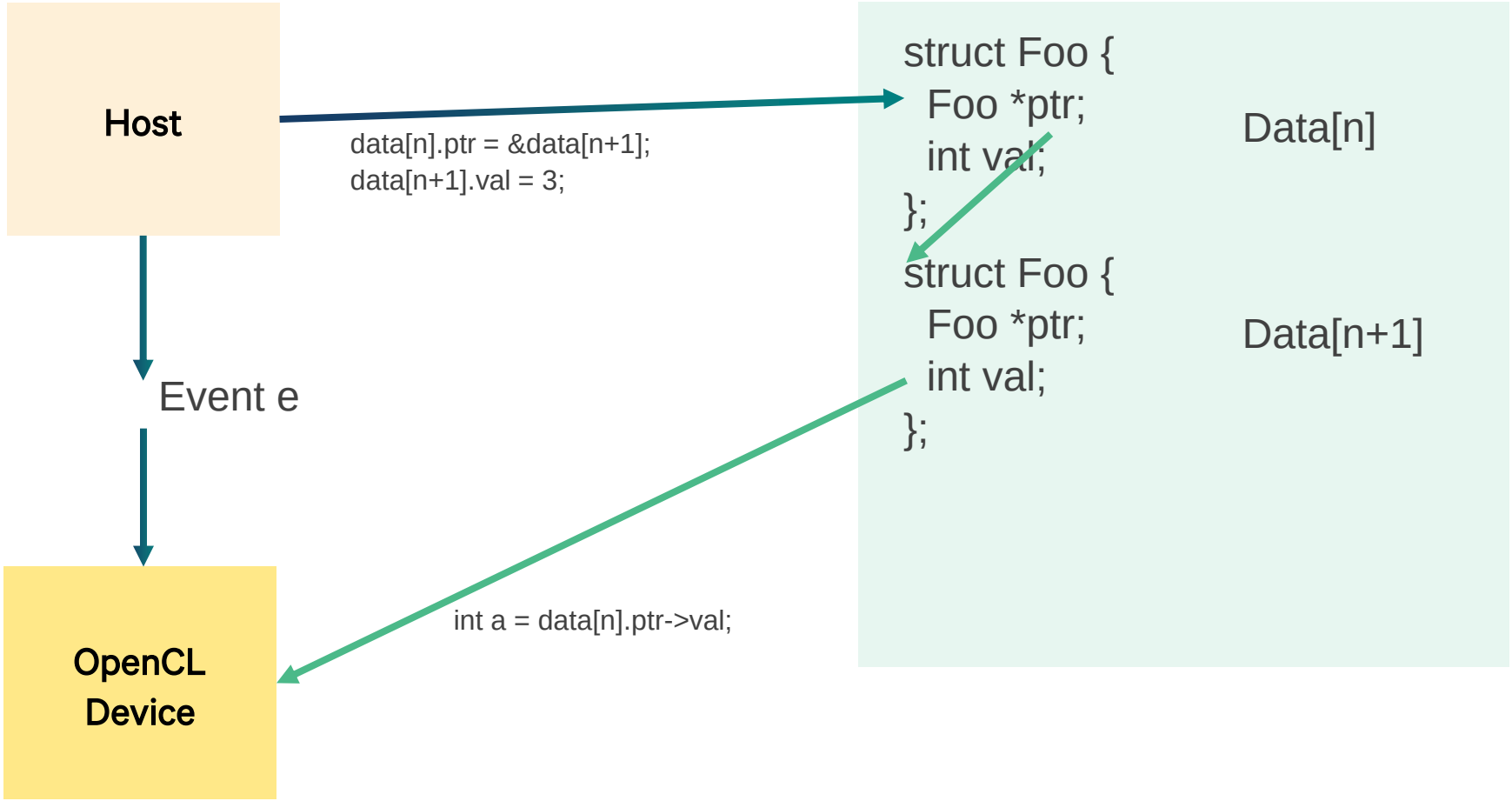
- Granularity of data race covers the whole buffer



Sharing data – when does the value change?

Fine grained

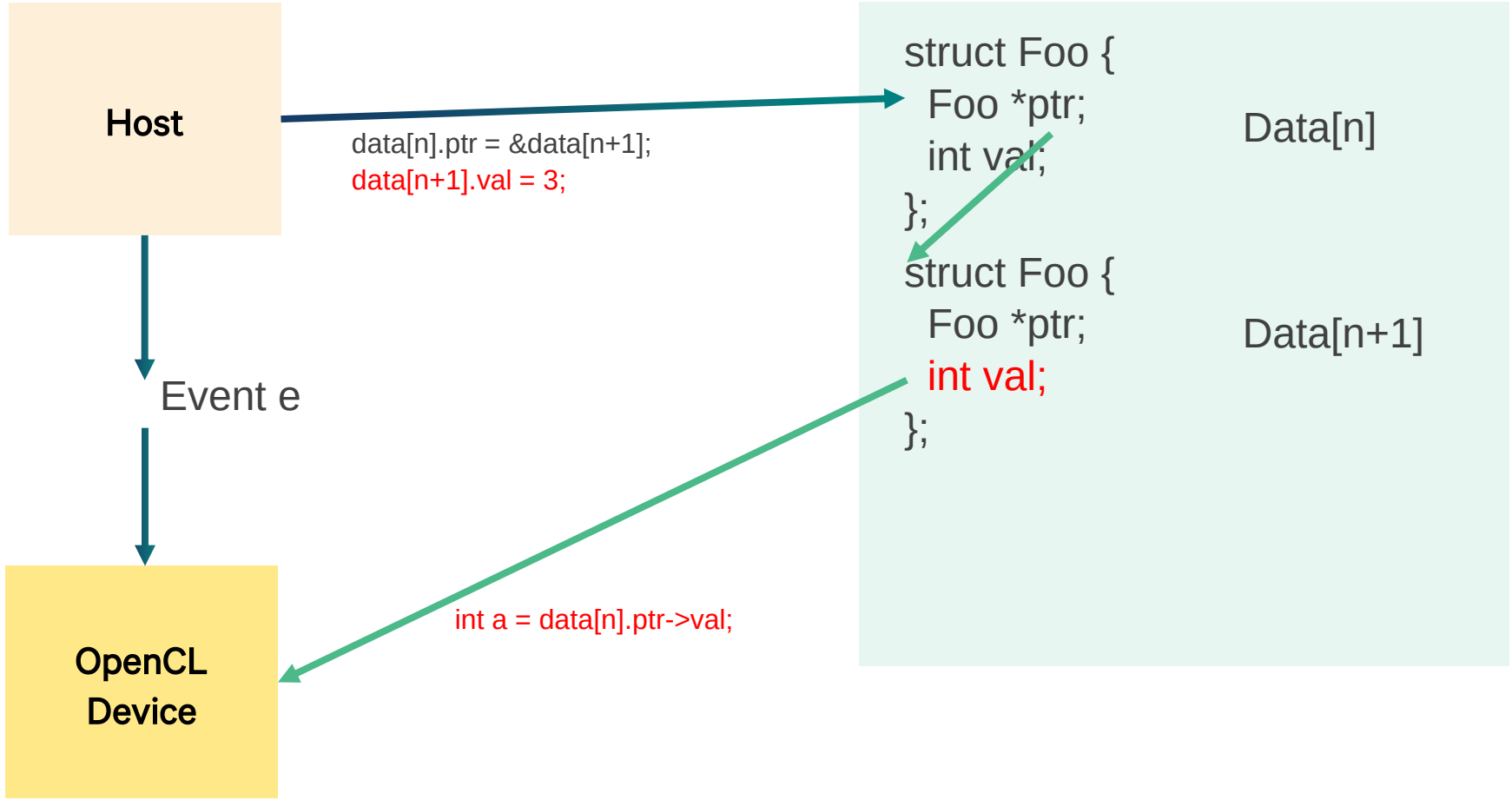
- Event dependency on device – caches will flush as necessary



Sharing data – when does the value change?

Fine grained

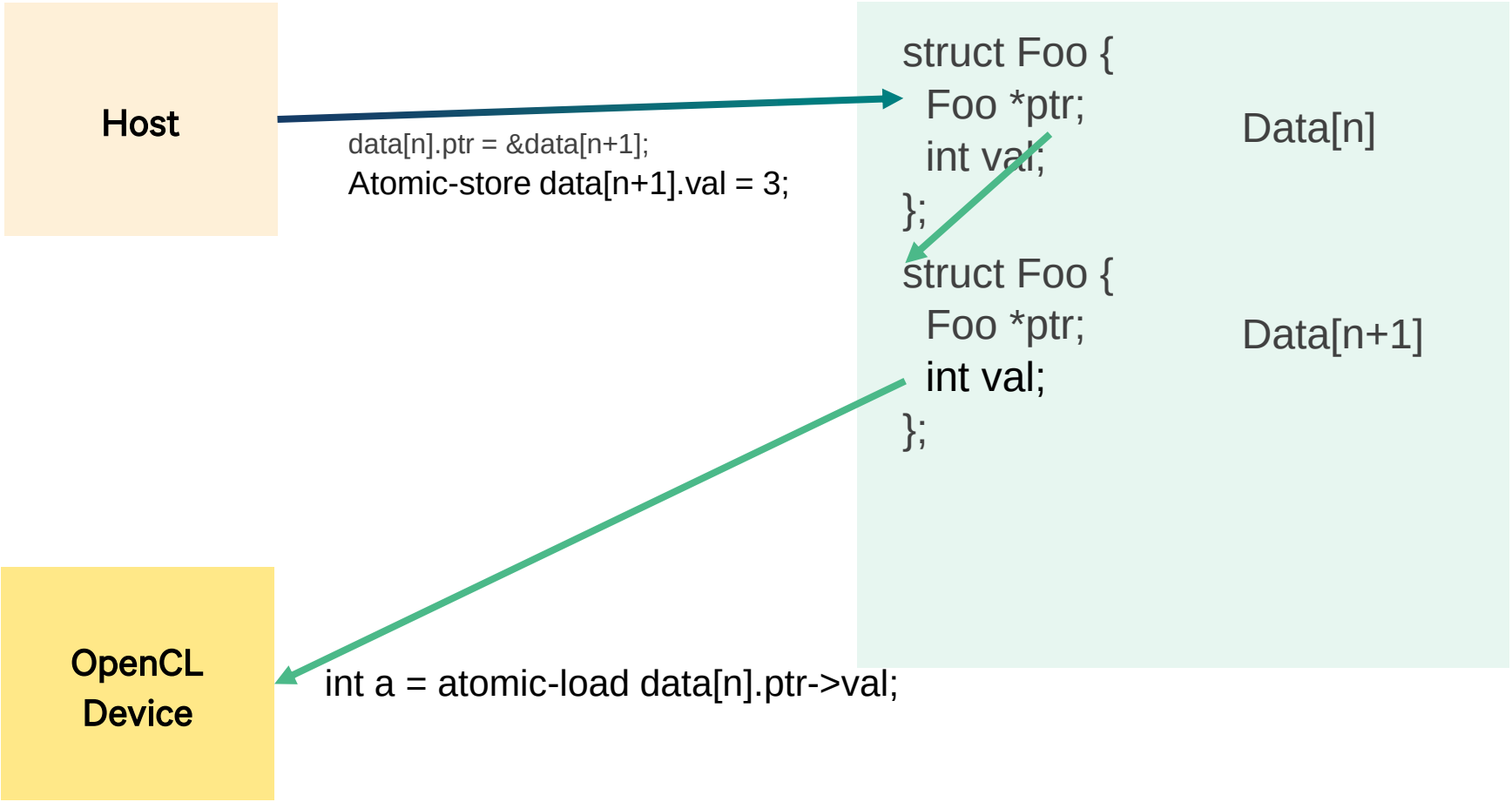
- Data will be merged – data race at byte granularity



Sharing data – when does the value change?

Fine grained with atomic support

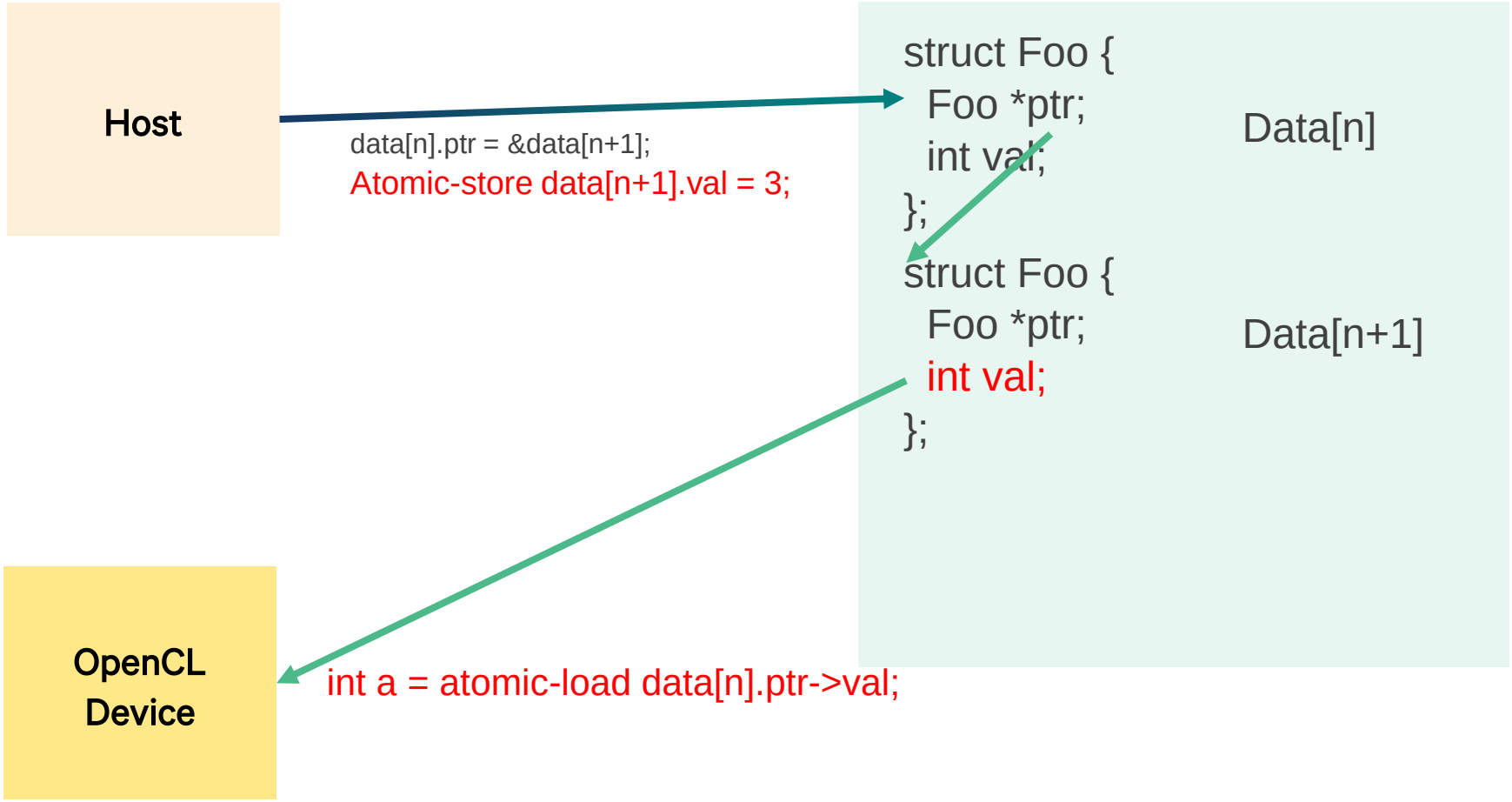
- No dispatch-level ordering necessary



Sharing data – when does the value change?

Fine grained with atomic support

- Races at byte level – avoided using the atomics in the memory model



Sharing virtual addresses

- This is an ease of programming concern
  - Complex apps with complex data structures can more effectively work with data
  - Less work to package and repackage data
- It can also improve performance
  - Less overhead in updating pointers to convert to offsets
  - Less overhead in repacking data
  - Lower overhead of data copies when appropriate hardware support present

Sharing virtual addresses

SC-for Data-Race-Free by default – release consistency for flexibility

- Most memory operations are entirely unordered
  - The compiler can reorder
  - The caches can reorder
- Unordered relations to the same location are *races*
  - Behaviour is undefined
- Ordering operations (atomics) may update the same location
  - Ordering operations order other operations *relative to the ordering operation*
- Ordering operations default to sequentially consistent semantics

So what can we safely do with synchronization

Take a spin wait…

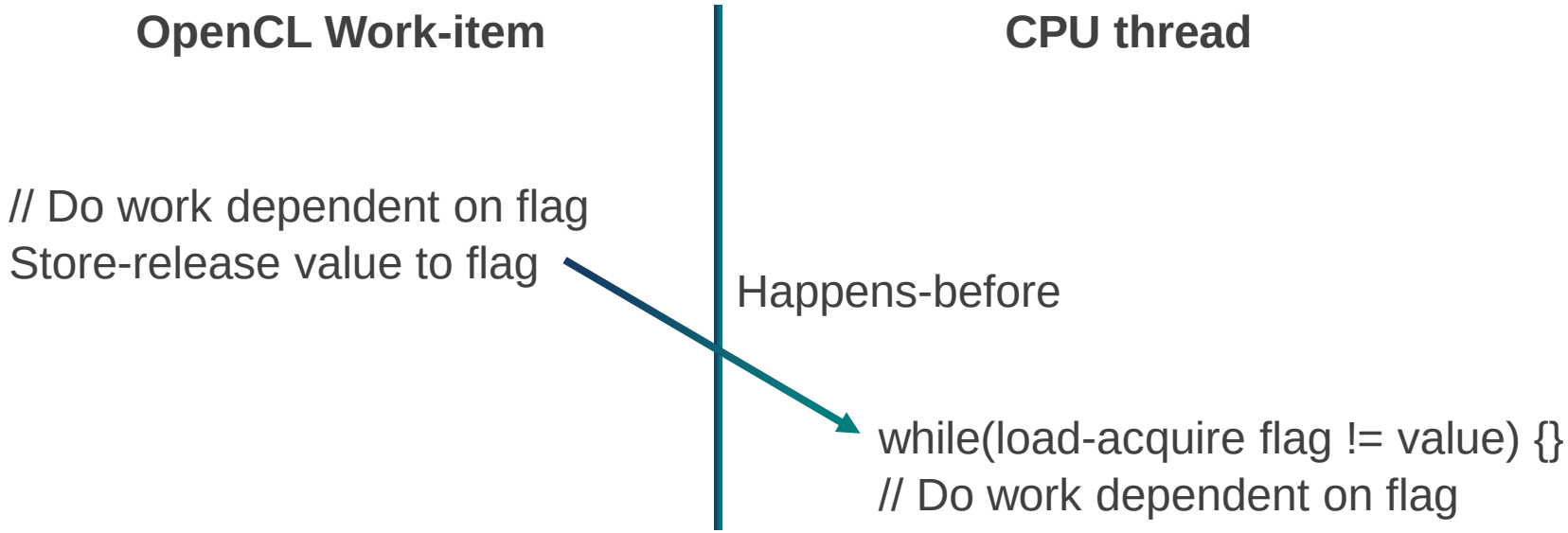
- Commonly tried on OpenCL 1.x
  - Not at all portable!
  - Due to: weak memory ordering, lack of forward progress

- Is the situation any better for OpenCL 2.0?
  - Yes, memory ordering is now under control!
  - Is forward progress?

So what can we safely do with synchronization

CPU thread waits on work-item – works?

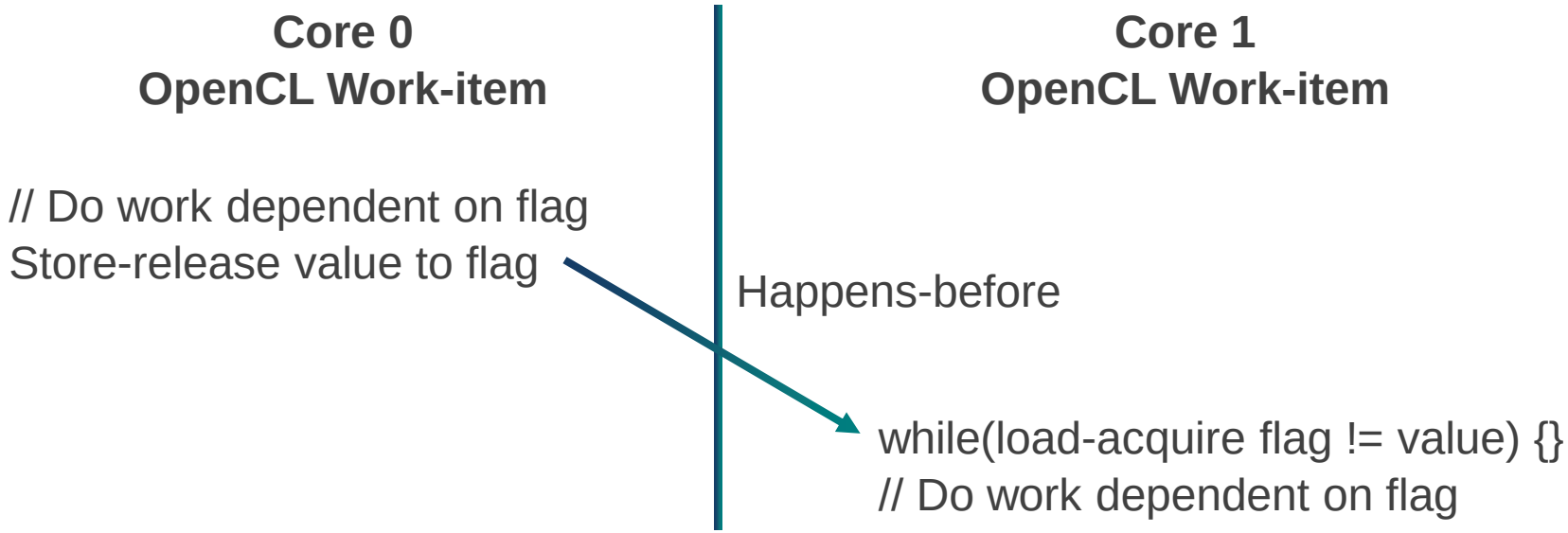
- Yes and no
  - Conceptually similar to CPU waiting on an event – ie all work-items to complete
  - Other app could occupy device, graphics could consume device, work may interfere with graphics
  - Risk of whole device being occupied elsewhere so work on the assumption that this context owns the device



So what can we safely do with synchronization

Work-item on one core waits for work-item on another core – works?

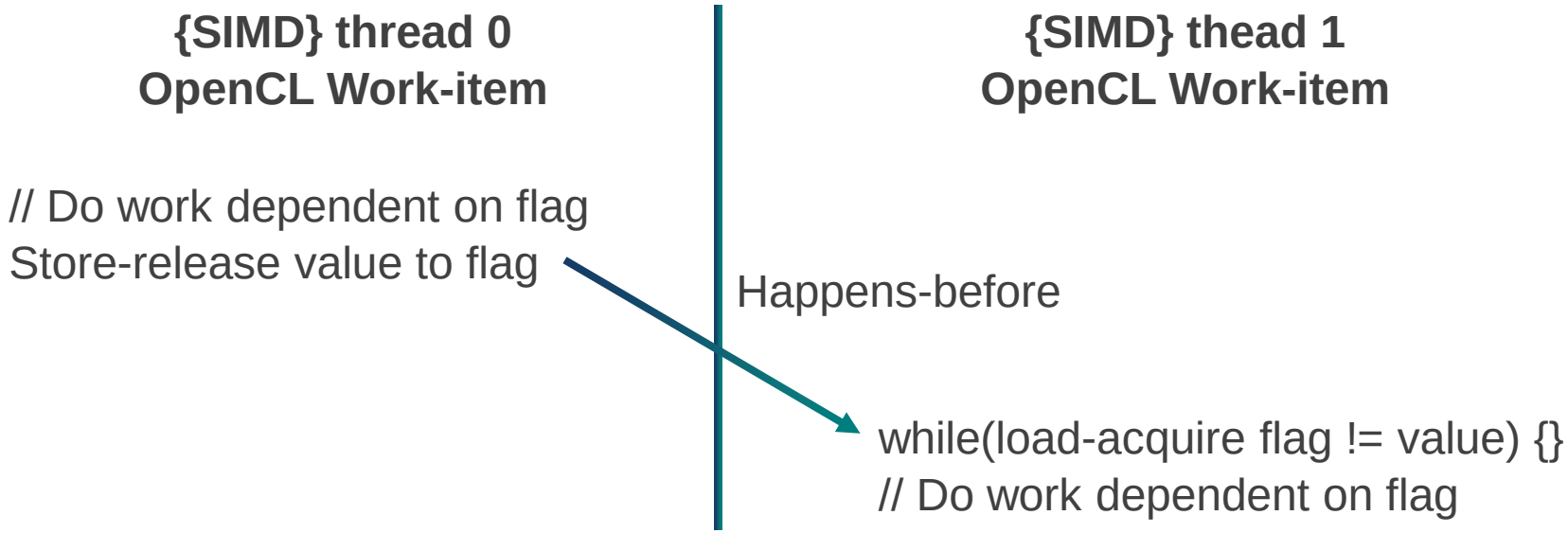
- Probably
  - The spec doesn't guarantee this
  - How do you know what core your work-item is on?
  - All you know is which work-group it is in.



So what can we safely do with synchronization

Work-item on one thread waits for work-item on another thread – works?

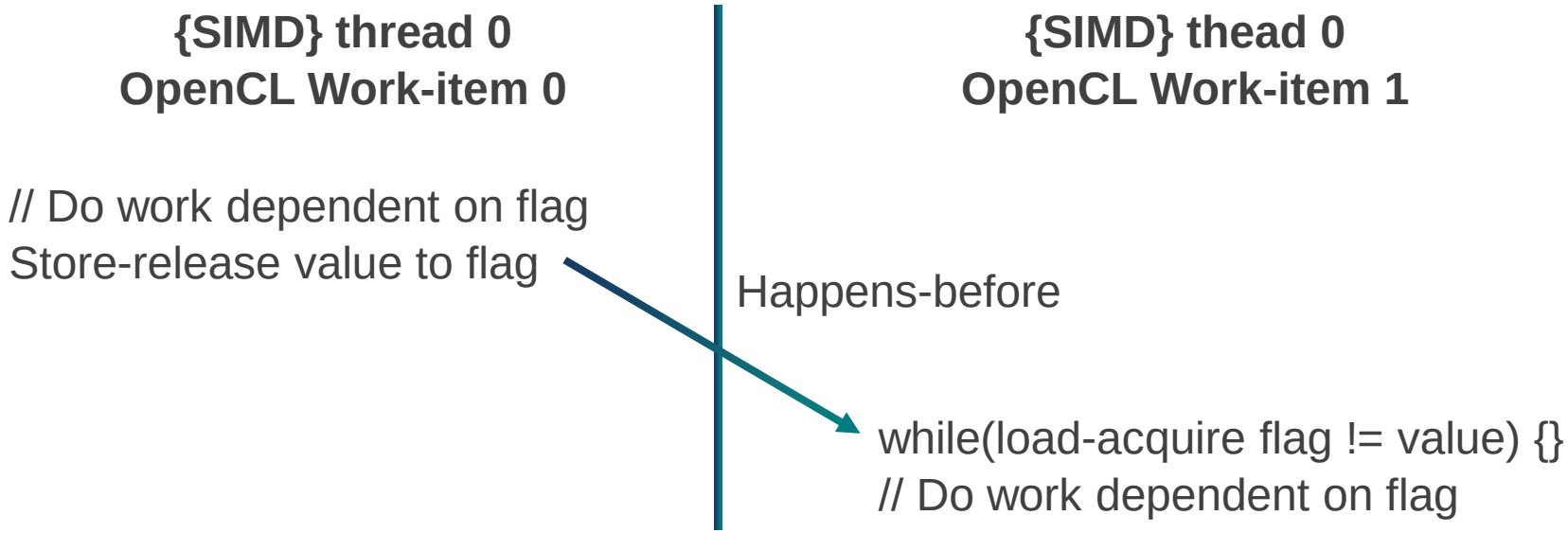
- Sometimes
  - On some architectures this is fine
  - On other architectures if both SIMD threads are on the same core: **starvation**
  - Thread 0 may never run to satisfy thread 1's spin wait



So what can we safely do with synchronization

Work-item waits for work-item in the same SIMD thread – works?

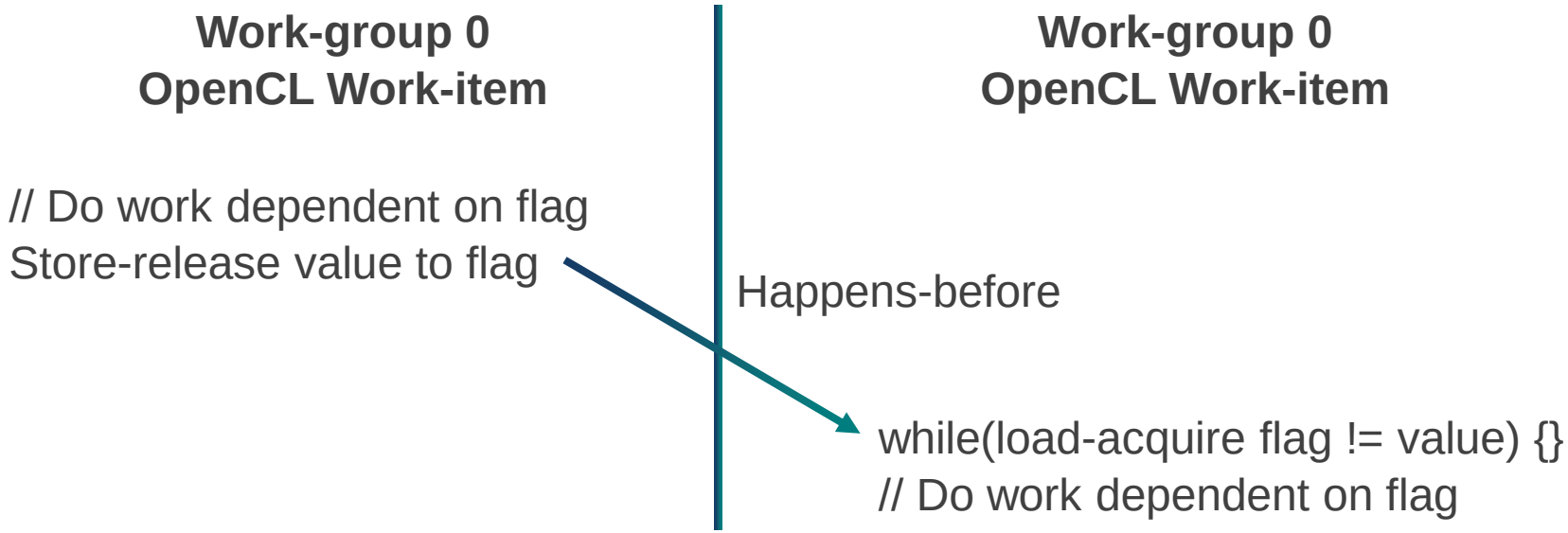
- No (well, sometimes yes, but rarely)
  - Fairly widely understood, but sometimes hard for new developers
  - If you think about the mapping to SIMD it is fairly obvious
  - A single program counter can't be in two places at once – some architectures can track multiple



So what can we safely do with synchronization

Work-items in a work-group

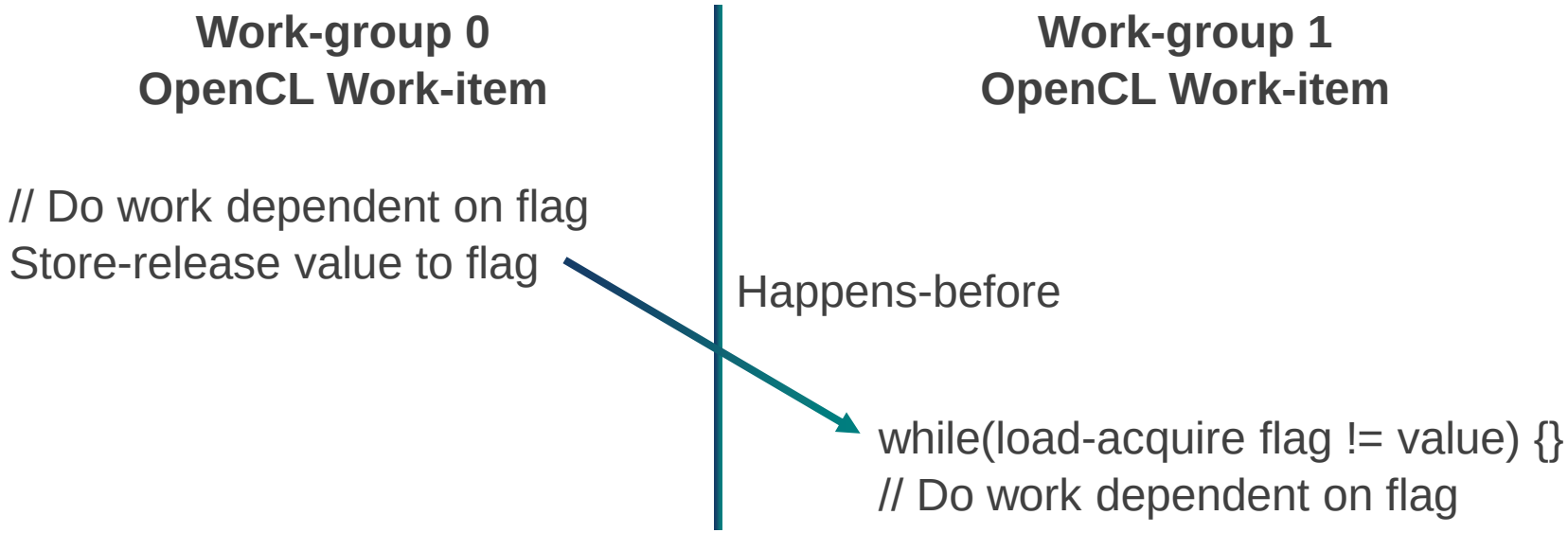
- Maybe, maybe not
  - It depends entirely on where the work-items are mapped in the group
  - Same thread – **no**
  - Different threads – **maybe**
  - The developer can often tell, but it isn't portable and the compiler can easily break it



So what can we safely do with synchronization

Work-groups

- Maybe, maybe not
  - It depends entirely on where the work-groups are placed on the device
  - Two work-groups on the same core – you have the thread to thread case
  - Two work-groups on different cores – it probably works
  - No way to control the mapping!

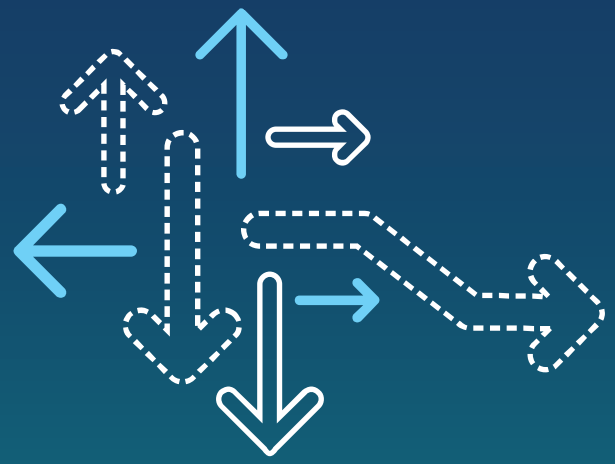


So what can we safely do with synchronization

Overall, a fairly poor situation

- Realistically
  - Spin waits on other OpenCL work-items are just not portable
  - Very limited use of the memory model

- So what can you do?
  - Communicating that work has passed a certain point
  - Updating shared data buffers with flags
  - Lock-free FIFO data structures to share data
  - OpenCL 2.0's sub-group extension provides limited but important forward progress guarantees



Heterogeneity in OpenCL 2.0

Introduction | Current restrictions | Basics of OpenCL 2.0 | **Heterogeneity in OpenCL 2.0** | Heterogeneous memory ordering | Summary

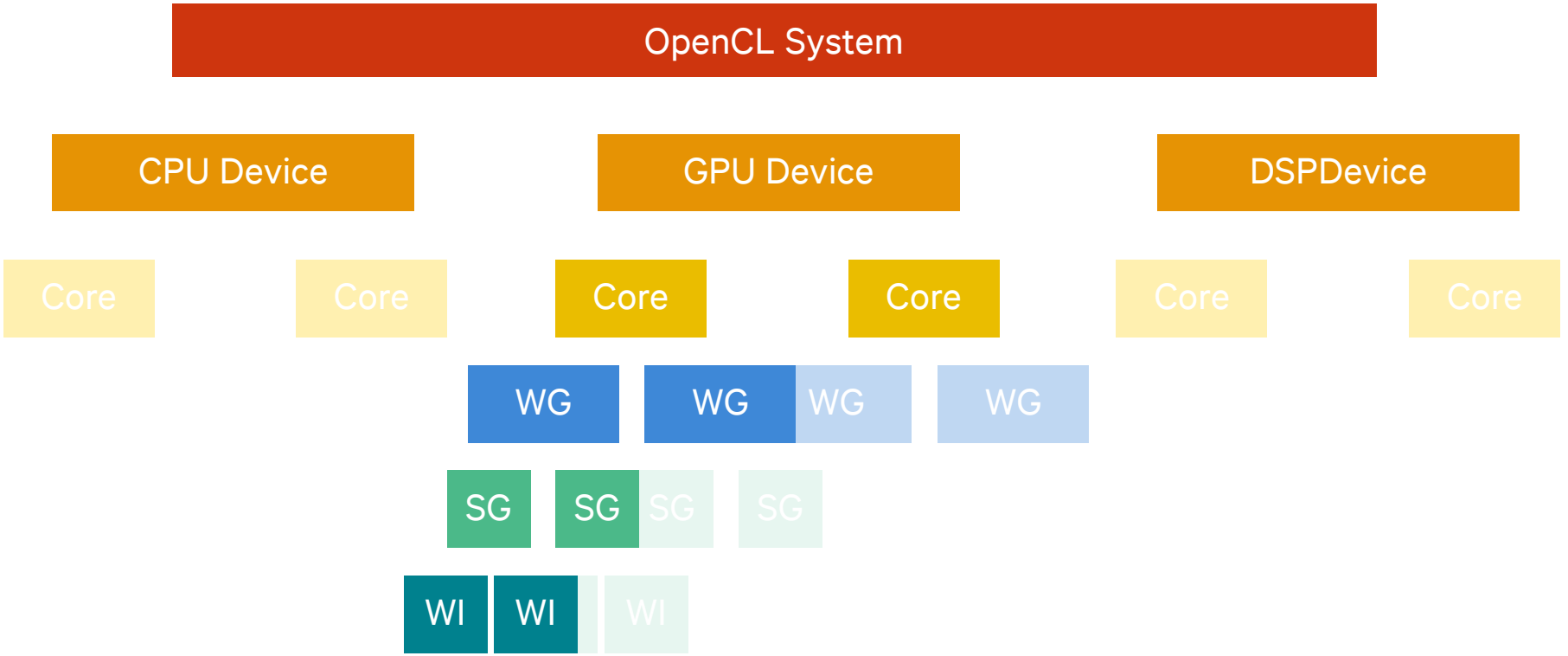
Lowering implementation cost

- Even acquire-release consistency can be expensive
- In particular, always synchronizing the whole system is expensive
  - A discrete GPU does not want to always make data visible across the PCIe interface
  - A single core shuffling data in local cache does not want to interfere with DRAM-consuming throughput tasks
- OpenCL 2 optimizes this using the concept of synchronization scopes

Hierarchical memory synchronization

Synchronization is expensive

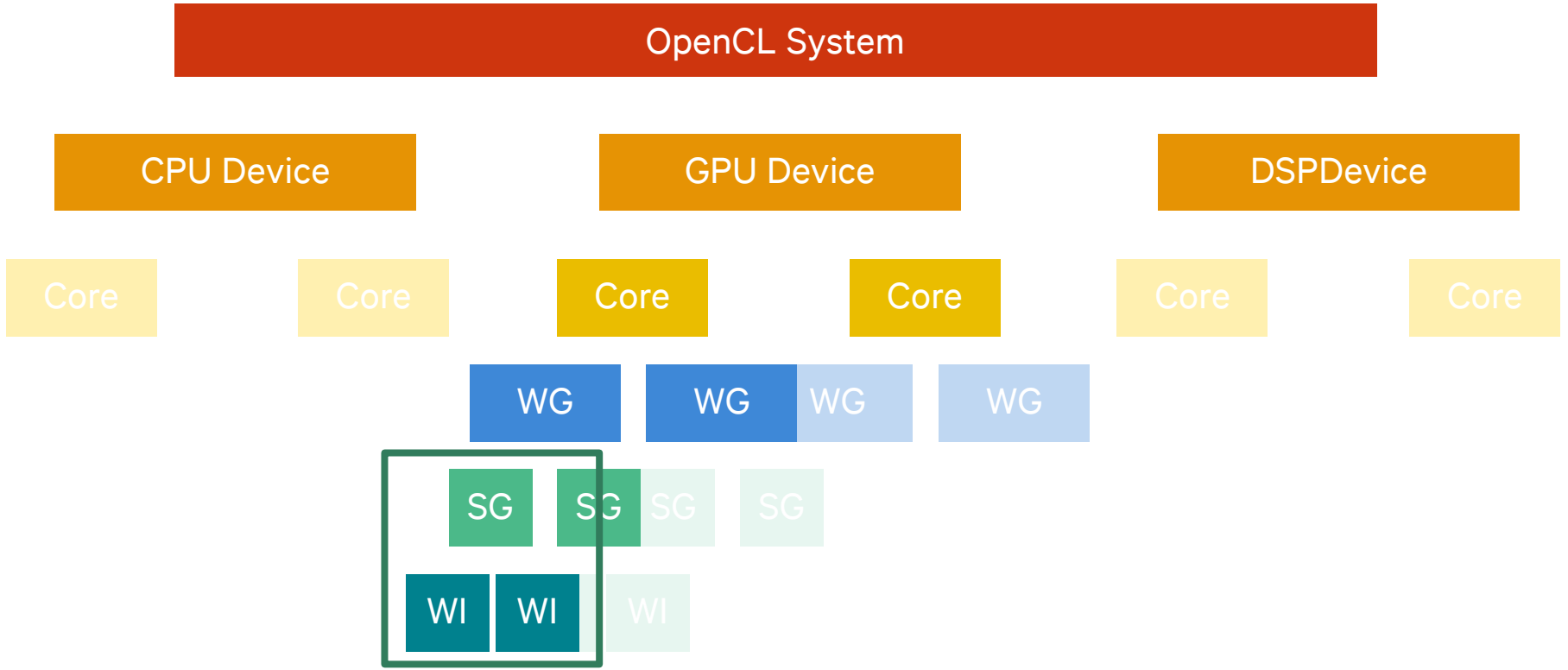
- Not all communication is global – so bound it



Hierarchical memory synchronization

Scopes!

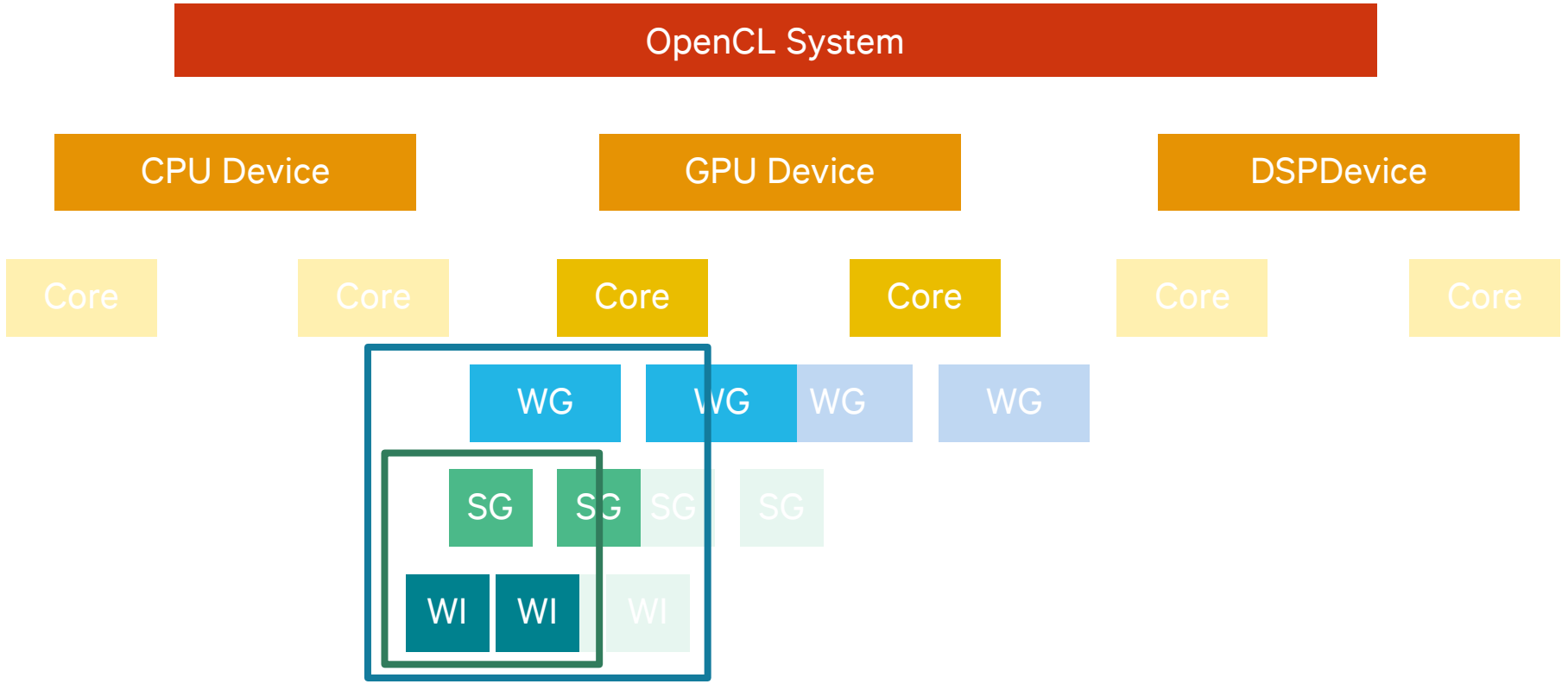
- Sub-group scope



Hierarchical memory synchronization

Scopes!

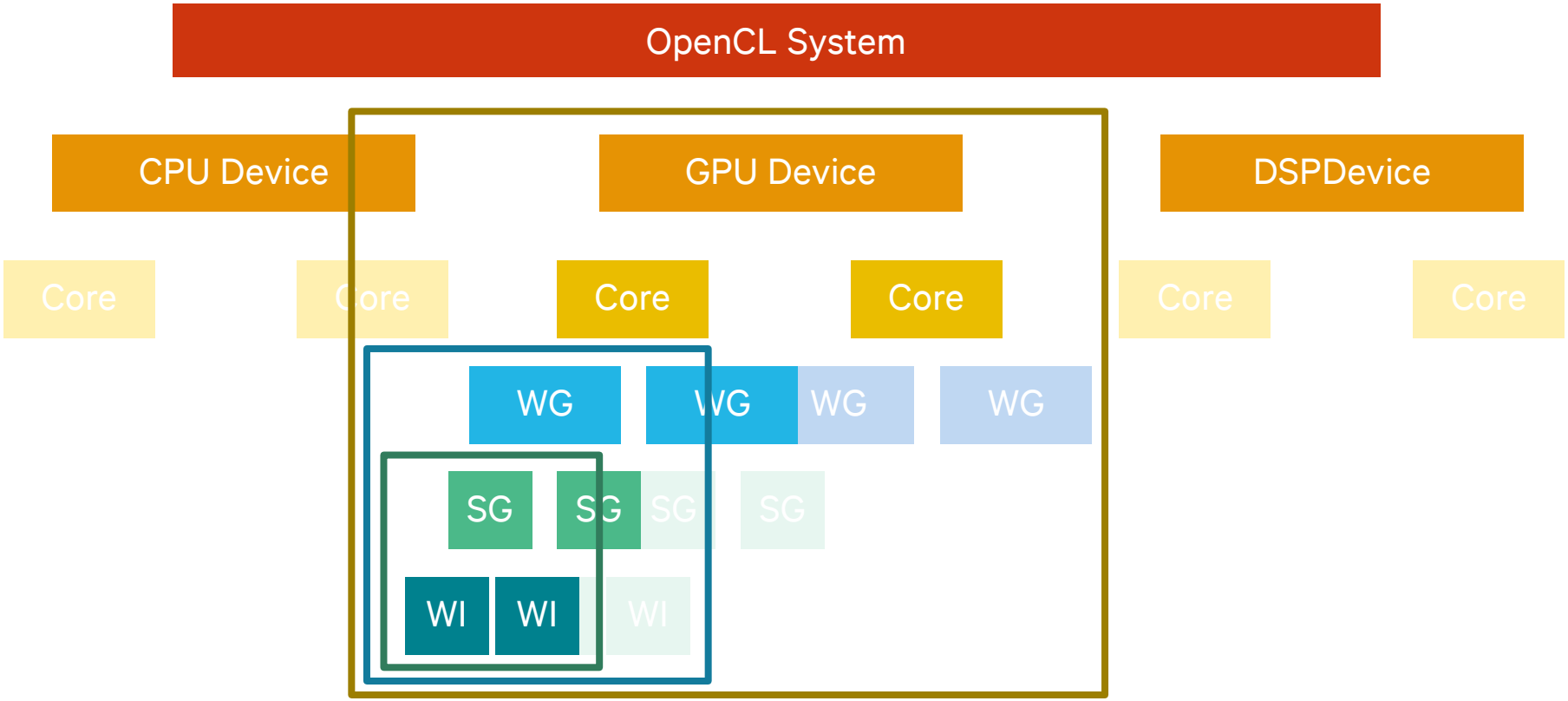
- Sub-group scope; Work-group scope



Hierarchical memory synchronization

Scopes!

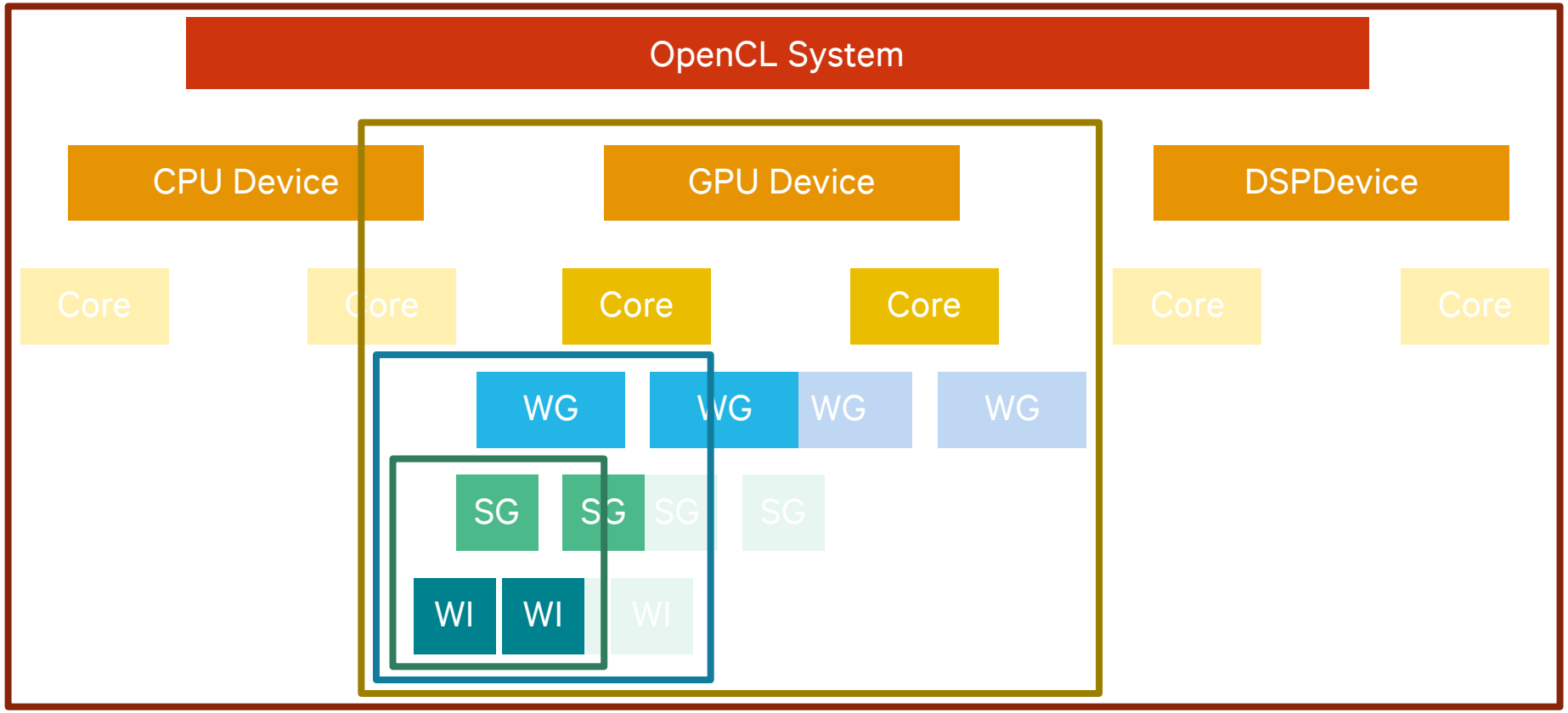
- Sub-group scope; Work-group scope; Device scope



Hierarchical memory synchronization

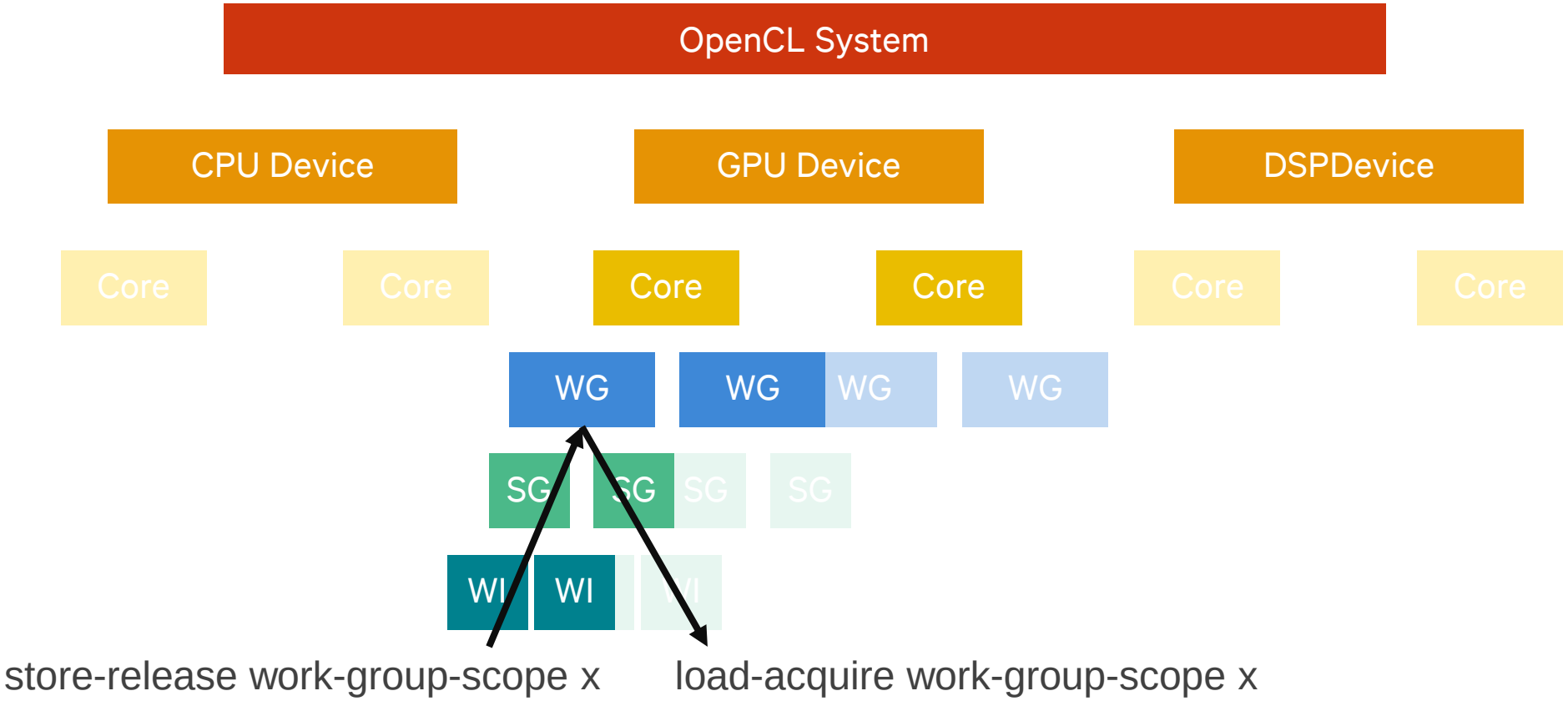
Scopes!

- Sub-group scope; Work-group scope; Device scope; All-SVM-Devices scope



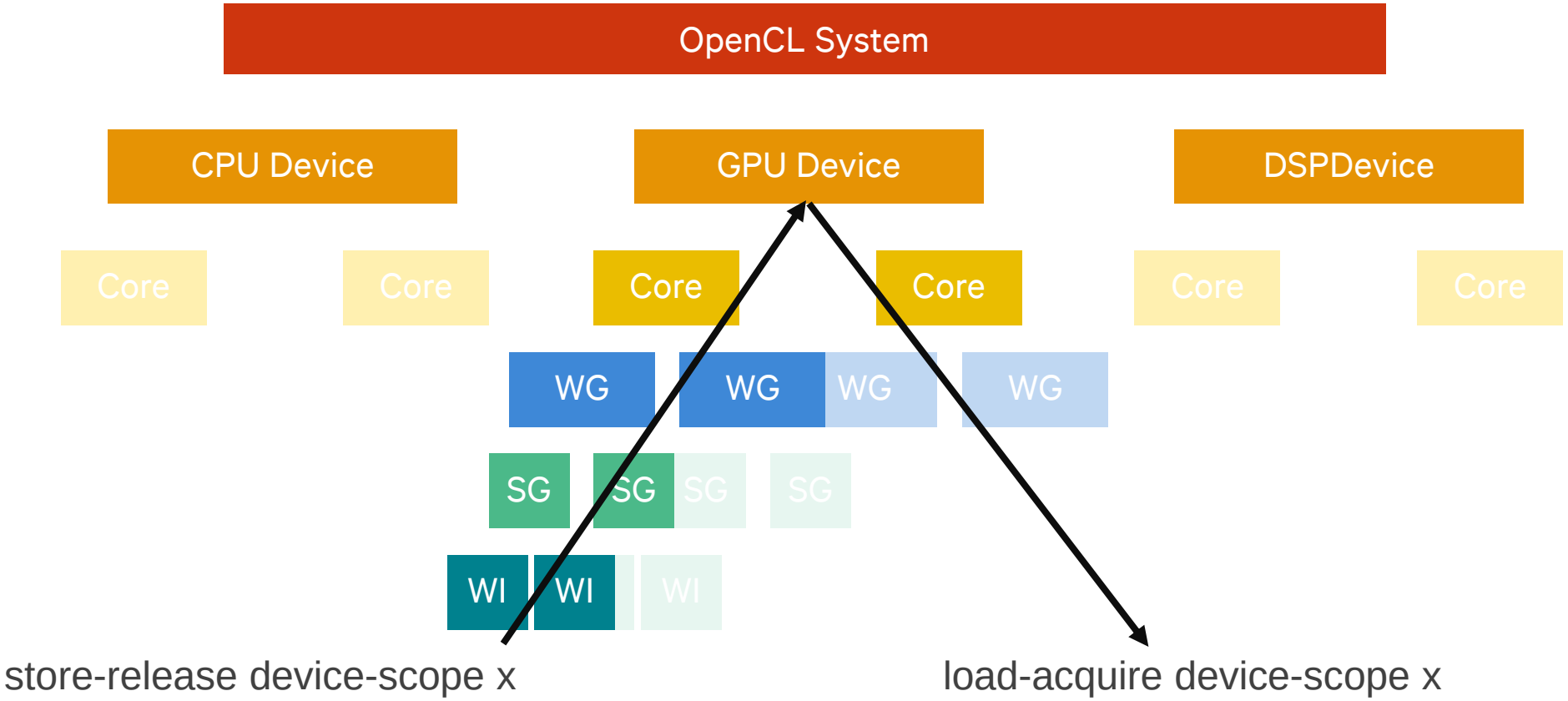
Hierarchical memory synchronization

Release to the appropriate scope, acquire from the matching scope



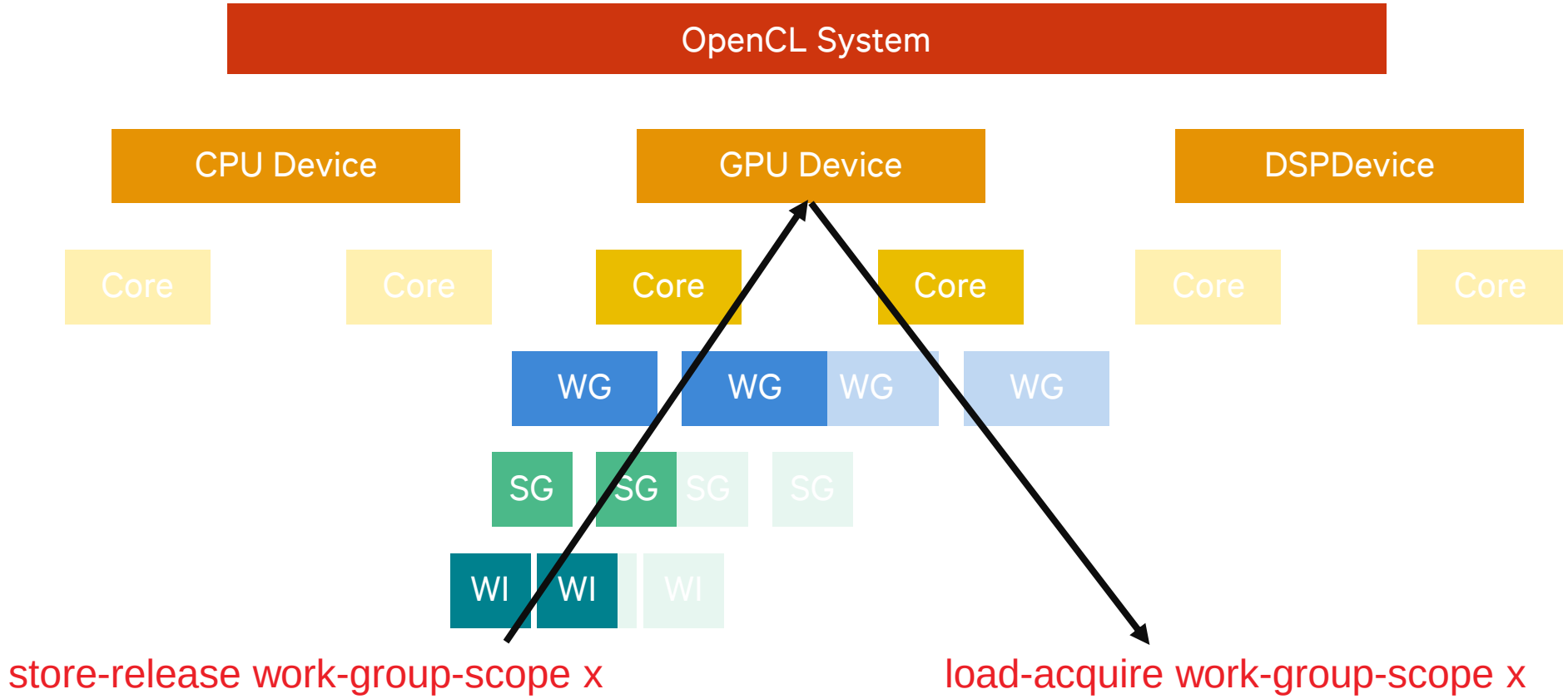
Hierarchical memory synchronization

Release to the appropriate scope, acquire from the matching scope



Hierarchical memory synchronization

If scopes do not reach far enough, this is a race

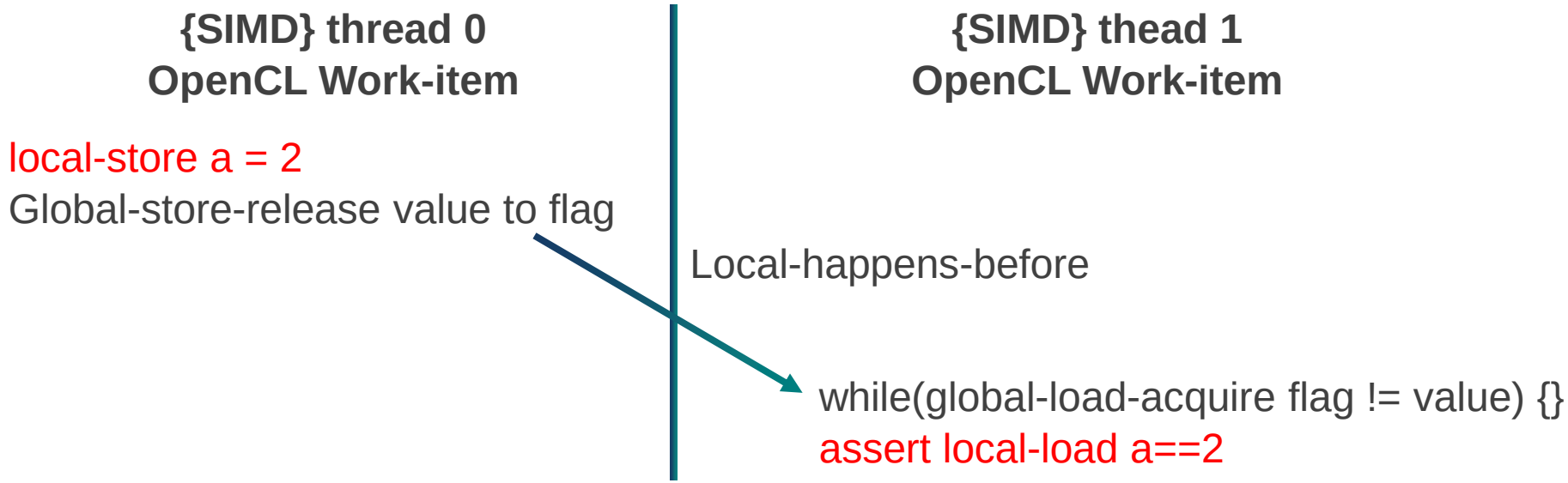


Scoped synchronization

- Allows aggressive hardware optimization to coherence traffic
  - GPU coherence in particular is often expensive – GPUs generate a lot of memory traffic
- The memory model defines synchronization rules in terms of scopes
  - Insufficient scope is a race
  - Non-matching scopes race (in the OpenCL 2.0 model, this isn't necessary)

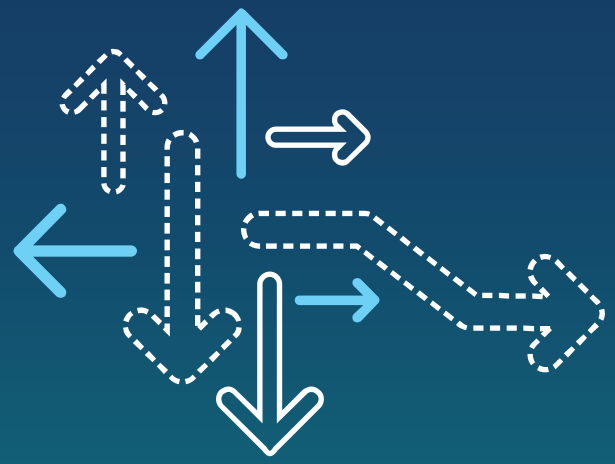
Address space orderings

- Four address spaces in OpenCL 2.0
  - Constant and private are not relevant for communication
  - Global and local maintain separate orders in the memory model
- Synchronization, acquire/release behavior etc apply only to local OR global, not both
  - The global release->acquire order below does not order the updates to a!



Multiple orderings in the presence of generic pointers

- Take an example like this:
  - void updateRelease(int *flag, int *data);
- If I lock the flag, do I safely update data or not?
  - The function takes pointers with no address space
  - The ordering depends on the address space
  - The address space depends on the type of the pointers at the call site, or even earlier!
- There are ways to get the fence flags, but it is messy, care must be taken



Heterogeneous memory ordering

Introduction | Current restrictions | Basics of OpenCL 2.0 | Heterogeneity in OpenCL 2.0 | Heterogeneous memory ordering | Summary

Limits of sequential consistency

- Sequential consistency aims to be a simple, easy to understand, model
  - Behave as if the ordering operations are simply interleaved

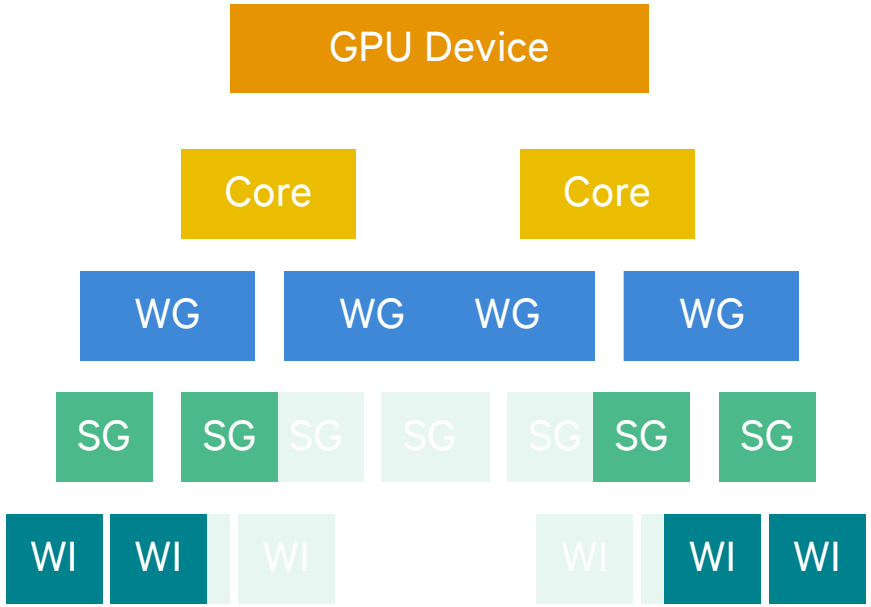
- In the OpenCL model, sequential consistency is guaranteed only in specific cases:
  - All memory_order_seq_cst operations have the scope **memory_scope_all_svm_devices** and all affected memory locations **are** contained in system allocations or fine grain SVM buffers with atomics support
  - All memory_order_seq_cst operations have the scope **memory_scope_device** and all affected memory locations **are not** located in system allocated regions or fine-grain SVM buffers with atomics support

- Consider what this means…
  - You start modifying your app to have more fine-grained sharing of some structures
  - Suddenly your atomics are not sequentially consistent at all!
  - What about SC operations to local memory?

Is such a limit necessary?

First, scopes…

- These are data-race-free memory models
- They only guarantee ordering in the absence of races
  - So we only actually order things we can observe! Order can be relaxed between atomics.



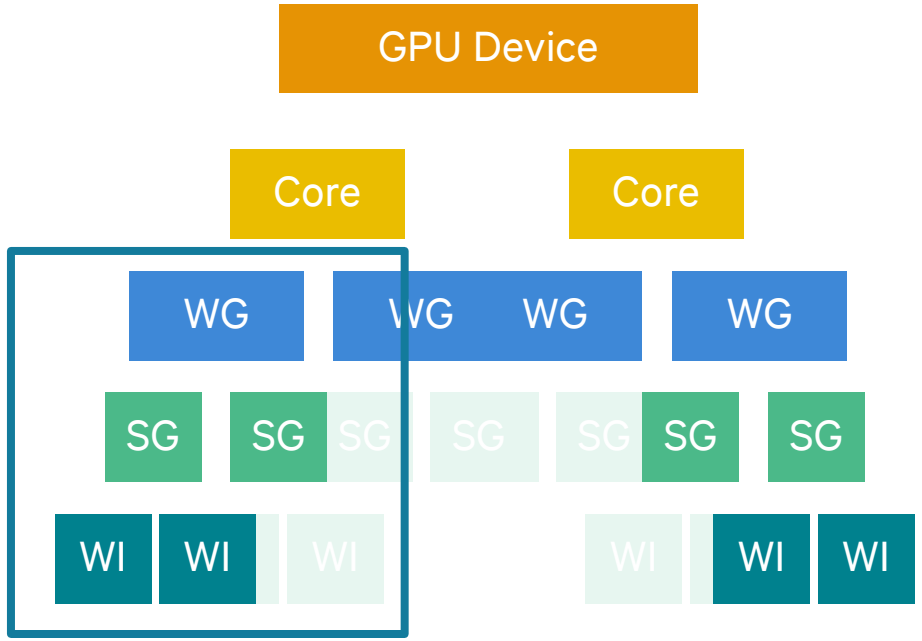
Is such a limit necessary?

First, scopes…

- In one part of the execution, we have SC operations at device scope
  - Let's assume this is valid

WG scope SC Operations.

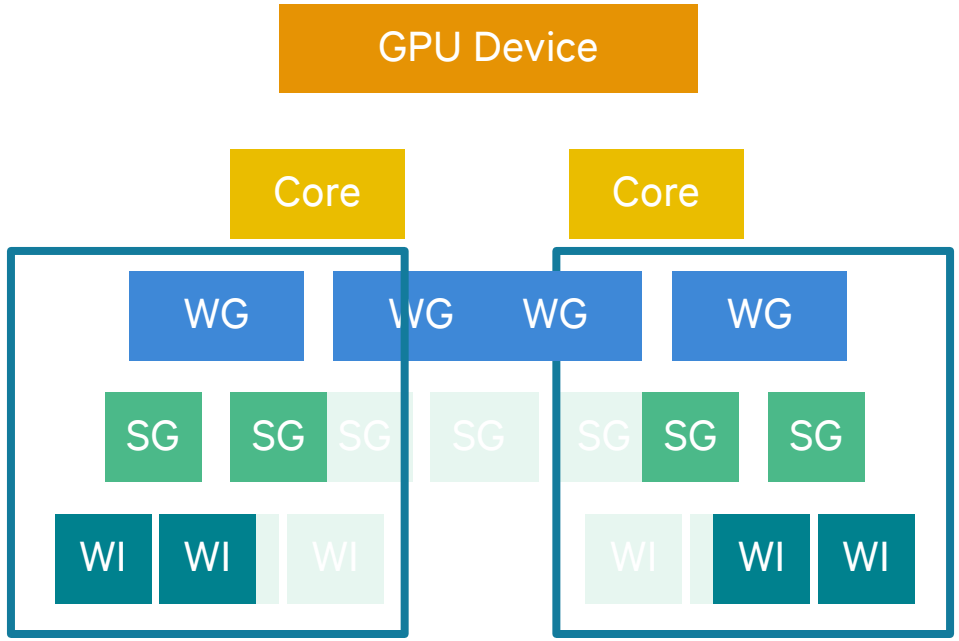
Ordered SC.



Is such a limit necessary?

First, scopes…

- Elsewhere we have another set of SC operations



WG scope SC Operations.

Ordered SC.

WG scope SC Operations.

Ordered SC.

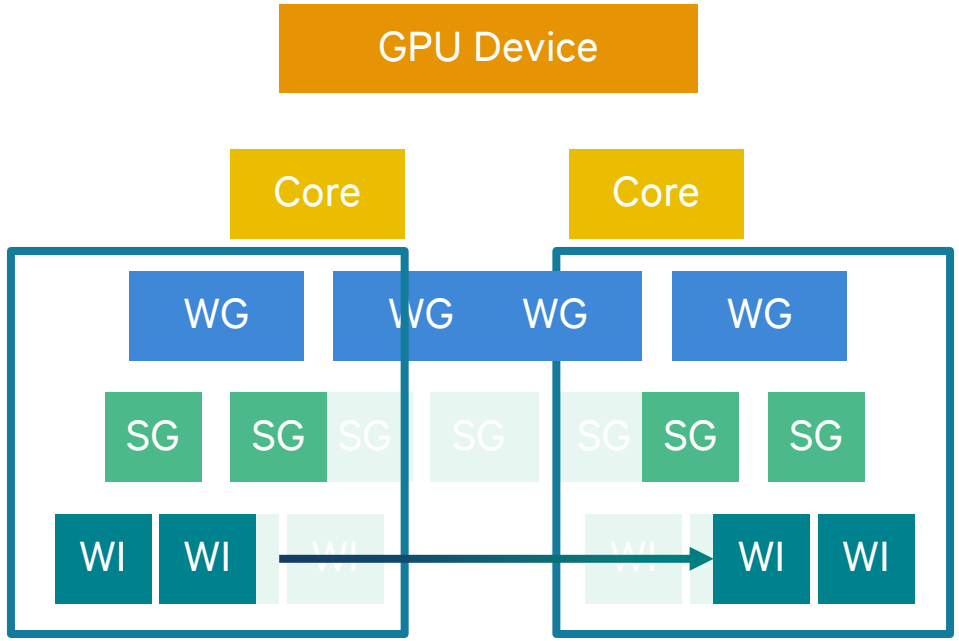
Is such a limit necessary?

First, scopes…

- Any access across from one work-group to another is a race
  - It is equivalent to a non-atomic operation
  - Therefore it is invalid

WG scope SC Operations.

Ordered SC.



WG scope SC Operations.

Ordered SC.

An access like this is a race

Making sequential consistency a partial order

- In Hower et. al.'s work on Heterogeneous-Race-Free memory models this is made explicit
- Sequential consistency can be maintained with scopes
- Access to invalid scope
  - Is unordered
  - Is not observable
  - So this is still valid sequential consistency: everything observable, ie that is race-free, is SC

Extending to coarse-grained memory

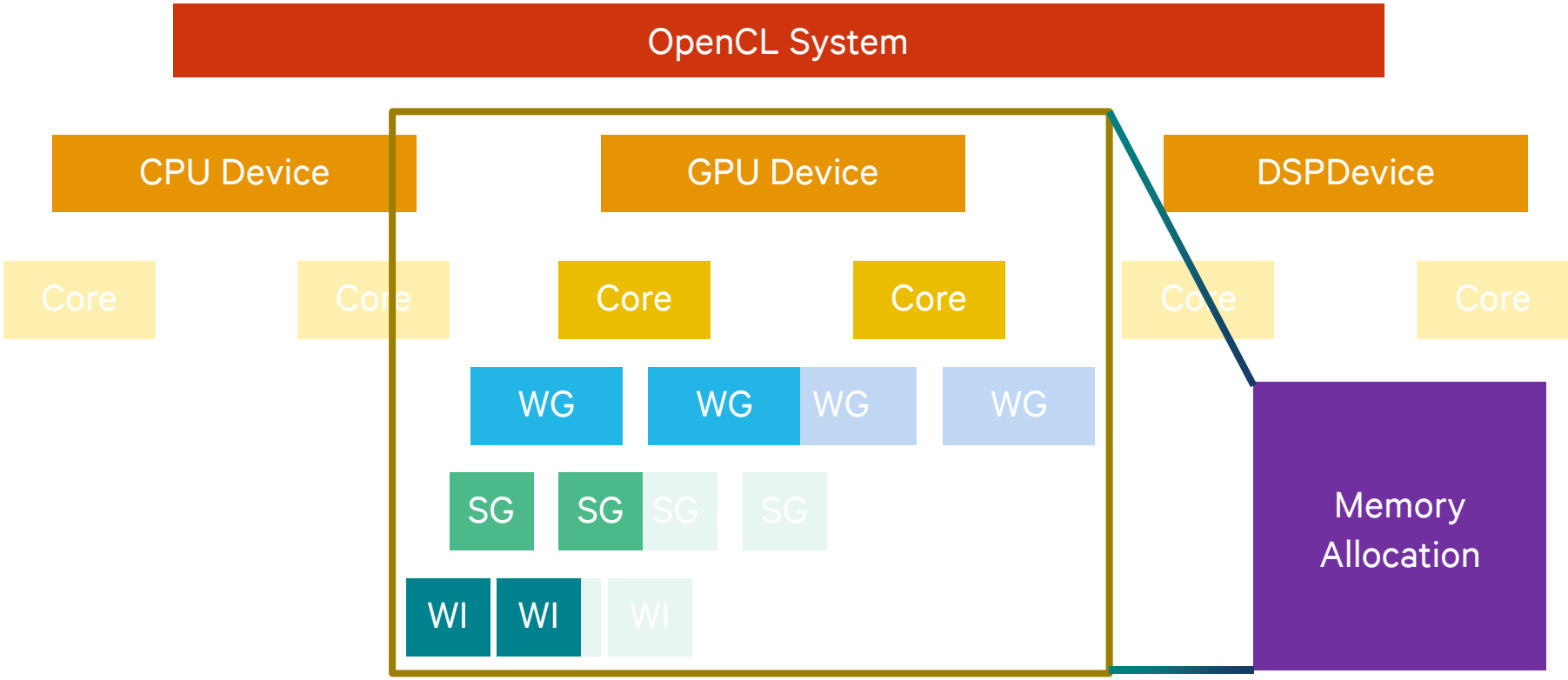
- We can apply SC semantics here for the same reason
  - Actions to coarse-grained memory is not visible to other clients
  - However – coarse buffers don't fit cleanly in a hierarchical model

- In Gaster et al. (to appear ACM TACO) we use the concept of *observability* as a memory model extension
  - *"At any given point in time a given location will be available in a particular set of scope instances out to some maximum instance and by some set of actors. Only memory operations that are inclusive with that maximal scope instance will observe changes to those locations."*

- Observability can migrate using API actions
  - Map, unmap, and event dependencies

Observability

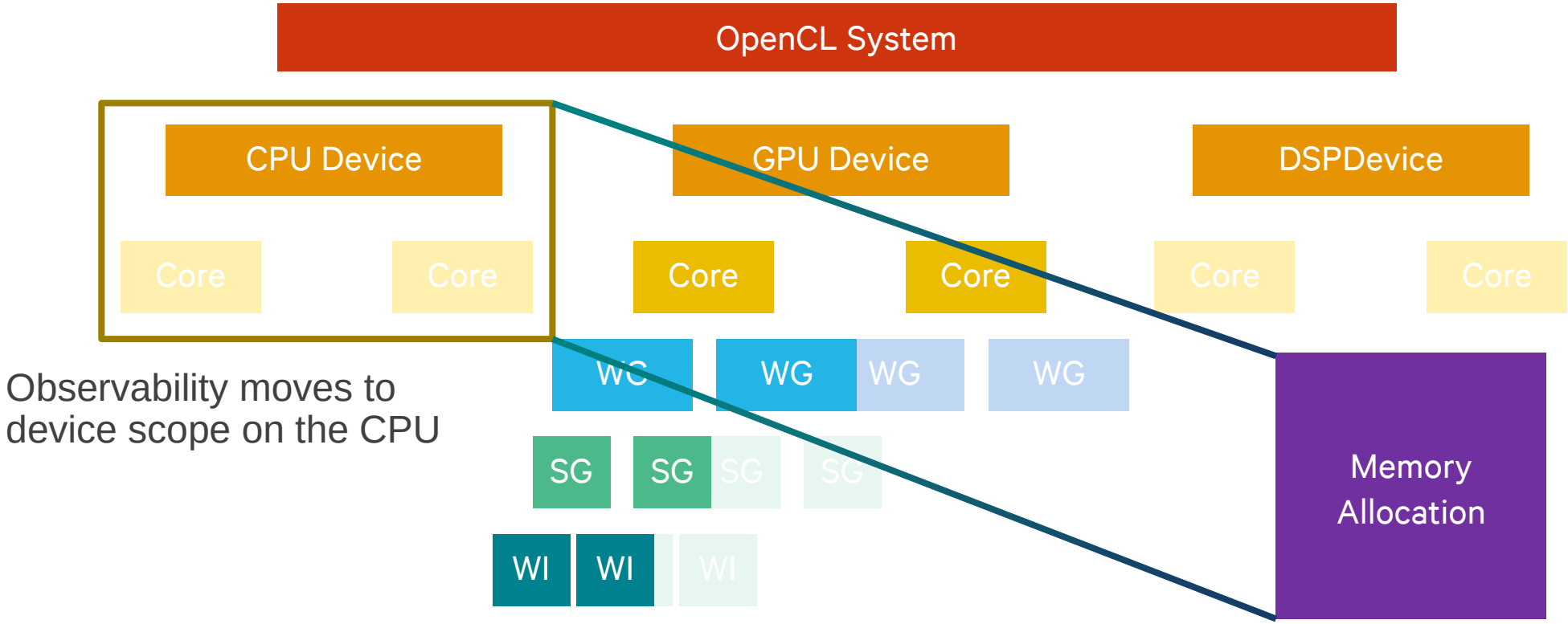
Initial state



Observability bounds – device scope on the GPU

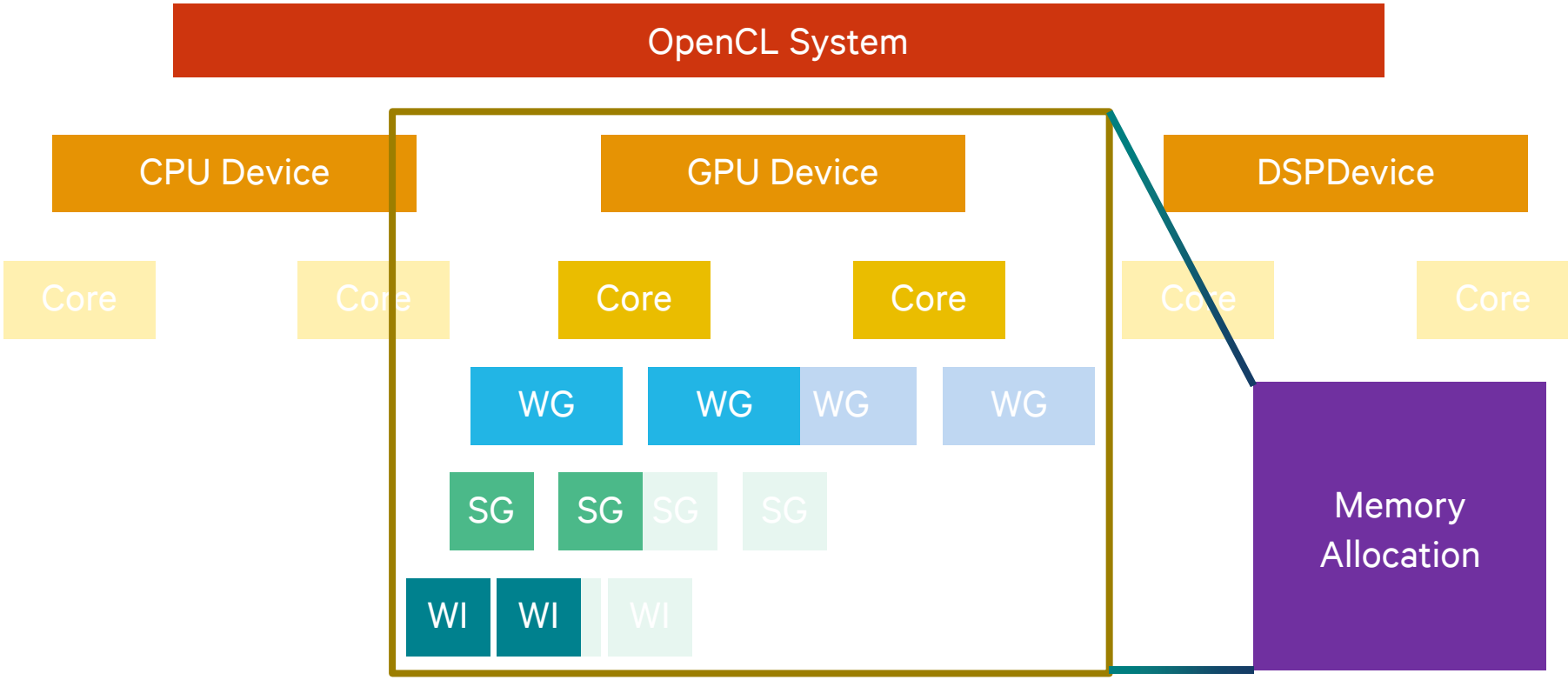
Observability

Map operation



Observability

Unmap



Unmap will transfer dependence back to an OpenCL device

The point

- We can cleanly put scopes and coarse memory in the same memory consistency model
  - It is more complicated than, say, C++
  - It is practical to formalize, and hopefully easy to explain
- There is no need for quirky corner cases
- We merely rethink slightly
  - Instead of a total order S for all SC operations we have a total order Sa for each agent a of all SC operations observable by that agent such that all these orders are consistent with each other.
  - This is a relatively small step from the DRF idea, which is effectively that non-atomic operations are not observable, and thus do not need to be strictly ordered.

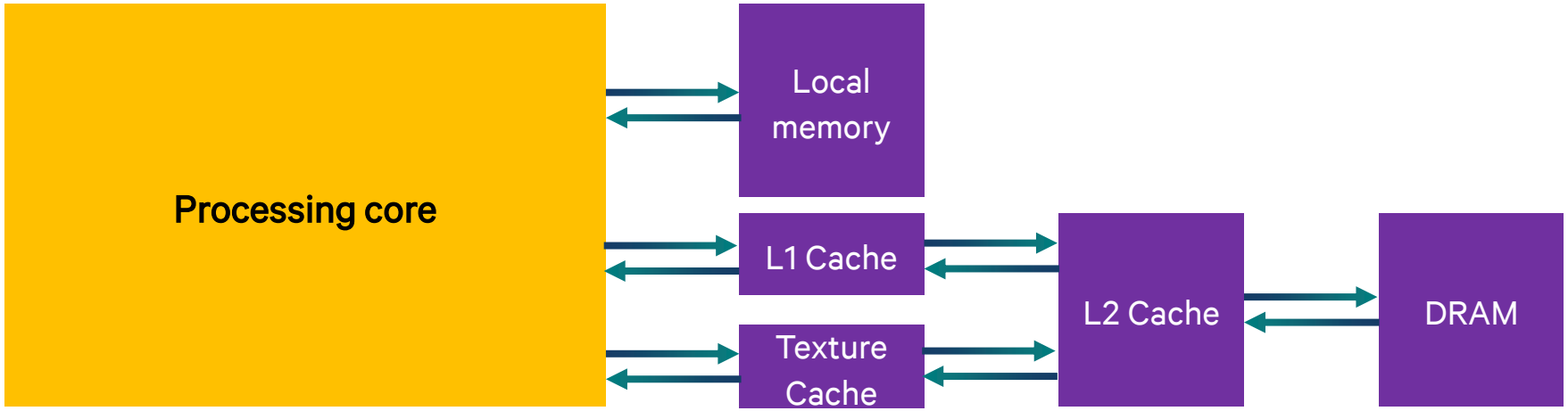
Joining address spaces

- Separate orders for local and global memory are likely to be painful for programmers

- Do we need them?
  - We have formalized the concept (also in the TACO paper) using multiple *happens-before* orders that subset memory operations
  - We also describe *bridging-synchronization-order* as a formal way to allow join points
  - It is messy as a formalization. The harm to the developer is probably significant.

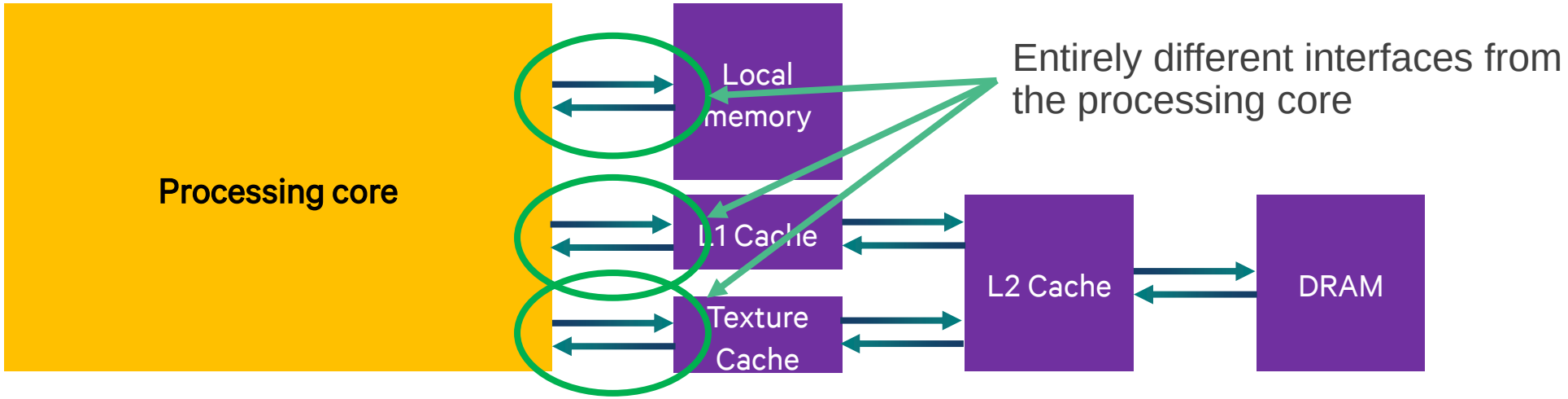
Joining address spaces

- However!
  - Hardware really does have different instructions and different timing to access different memories
  - Can all hardware efficiently synchronize these memory interfaces?



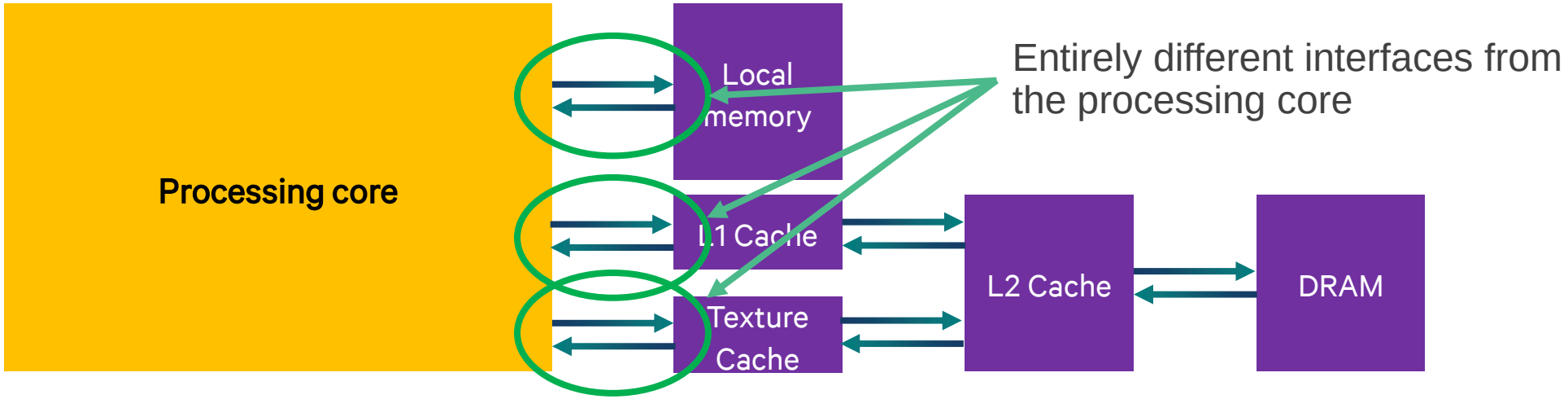
Joining address spaces

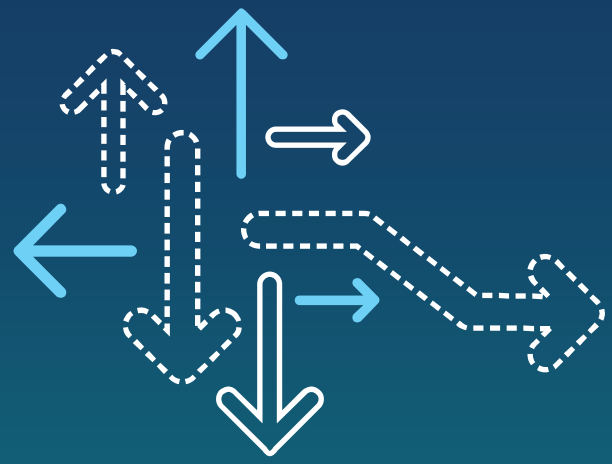
- However!
  - Hardware really does have different instructions and different timing to access different memories
  - Can all hardware efficiently synchronize these memory interfaces?



Joining address spaces

- However!
  - Hardware really does have different instructions and different timing to access different memories
  - Can all hardware efficiently synchronize these memory interfaces?
  - We will have to see how this plays out
  - Consider this a warning to take care if you try to use this aspect of OpenCL 2.0





Summary

Introduction | Current restrictions | Basics of OpenCL 2.0 | Heterogeneity in OpenCL 2.0 | Heterogeneous memory ordering | Summary

Things are improving

- Like mainstream languages, heterogeneous programming models are adopting firm memory models
- Without fundamental execution model guarantees the usefulness is limited
- We are making progress on both these counts

Thank you

Follow us on:

For more information on Qualcomm, visit us at:
www.qualcomm.com & www.qualcomm.com/blog