Sequential Consistency for Heterogeneous-Race-Free

DEREK R. HOWER, BRADFORD M. BECKMANN, BENEDICT R. GASTER,
BLAKE A. HECHTMAN, MARK D. HILL, STEVEN K. REINHARDT, DAVID A. WOOD

JUNE 12, 2013

EXECUTIVE SUMMARY

AMD

- Existing GPU memory models ambiguous, dense for programmers
- CPUs: Sequential Consistency for Data-Race-Free (SC for DRF)
  - Relaxed HW, precise semantics, programmer-friendly
  - Problem: GPUs use scoped synchronization
- Sequential Consistency for Heterogeneous-Race-Free (SC for HRF)
  - SC for DRF + Scopes
- **HRF0**: Basic scope synchronization
  - Two threads communicate → use *identical* scope synchronization
  - Works well with existing, regular GPU codes
- Beyond HRF0?
  - There are limits

OUTLINE

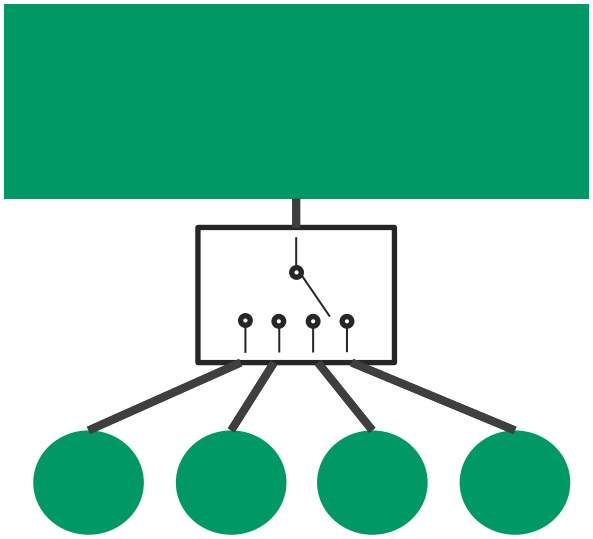
AMD

- Background and Setup
- HRF0: Basic scope synchronization
- Future directions

BACKGROUND AND SETUP

AMD

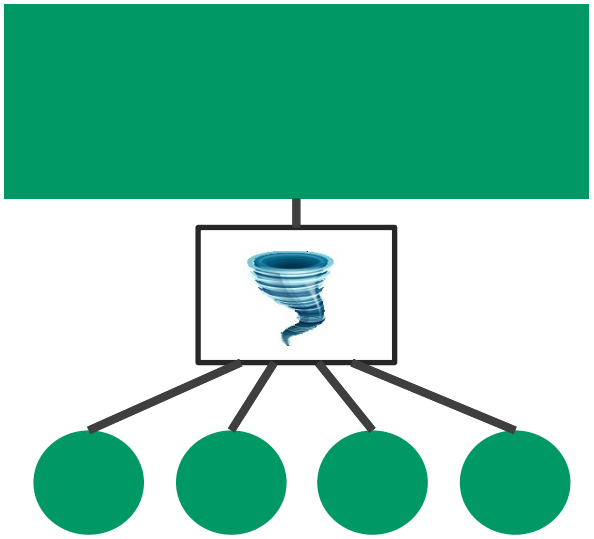
- CPU Programmers use *memory model* to understand memory behavior
  - Sequential Consistency (SC) [1979]: threads interleave like multitasking uniprocessor



BACKGROUND AND SETUP

AMD

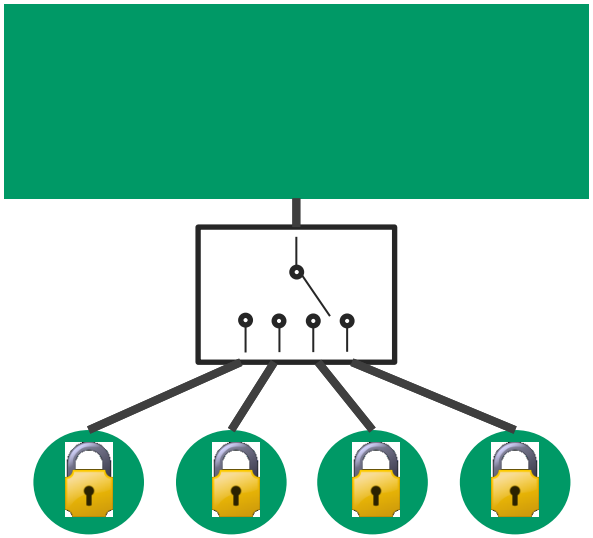
- CPU Programmers use *memory model* to understand memory behavior
  - Sequential Consistency (SC) [1979]: threads interleave like multitasking uniprocessor
  - HW/Compiler actually implements TSO [1991] or more relaxed model



BACKGROUND AND SETUP

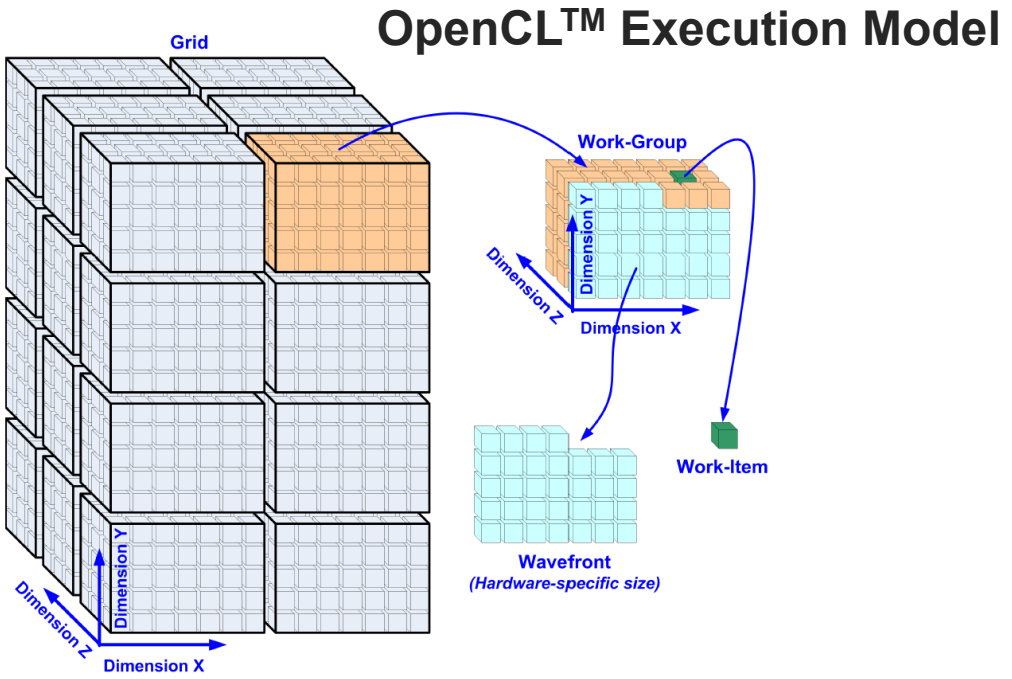
AMD

- CPU Programmers use *memory model* to understand memory behavior
  - Sequential Consistency (SC) [1979]: threads interleave like multitasking uniprocessor
  - HW/Compiler actually implements TSO [1991] or more relaxed model
  - Java™ [2005] and C++ [2008] insure SC for data-race-free (DRF) programs
- Programmers need a GPU memory model for abstraction and portability
  - Currently GPU models expose ad hoc HW mechanisms
  - SC for DRF is a start, BUT…



BACKGROUND AND SETUP

- CPU Programmers use *memory model* to understand memory behavior
  - Sequential Consistency (SC) [1979]: threads interleave like multitasking uniprocessor
  - HW/Compiler actually implements TSO [1991] or more relaxed model
  - Java™ [2005] and C++ [2008] insure SC for data-race-free (DRF) programs
- Programmers need a GPU memory model for abstraction and portability
  - Currently GPU models expose ad hoc HW mechanisms
  - SC for DRF is a start, BUT…



SEQUENTIAL CONSISTENCY FOR DATA-RACE-FREE

AMD

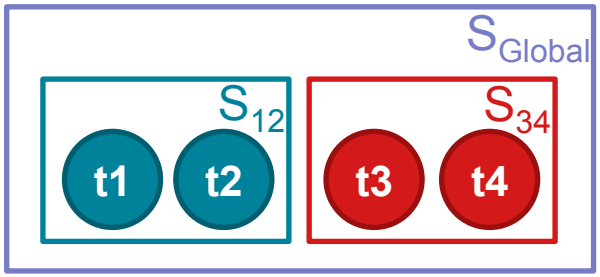
- Two memory accesses participate in a **data race** if they
  - access the same location
  - at least one access is a store
  - can occur simultaneously
    - i.e. appear as adjacent operations in interleaving.

GPUs: Not good enough!

- A program is **data-race-free** if no possible execution results in a data race.
- Sequential consistency for data-race-free programs
  - Avoid everything else

DATA-RACE-FREE IS NOT ENOUGH

AMD



Workgroup #1-2

Workgroup #3-4

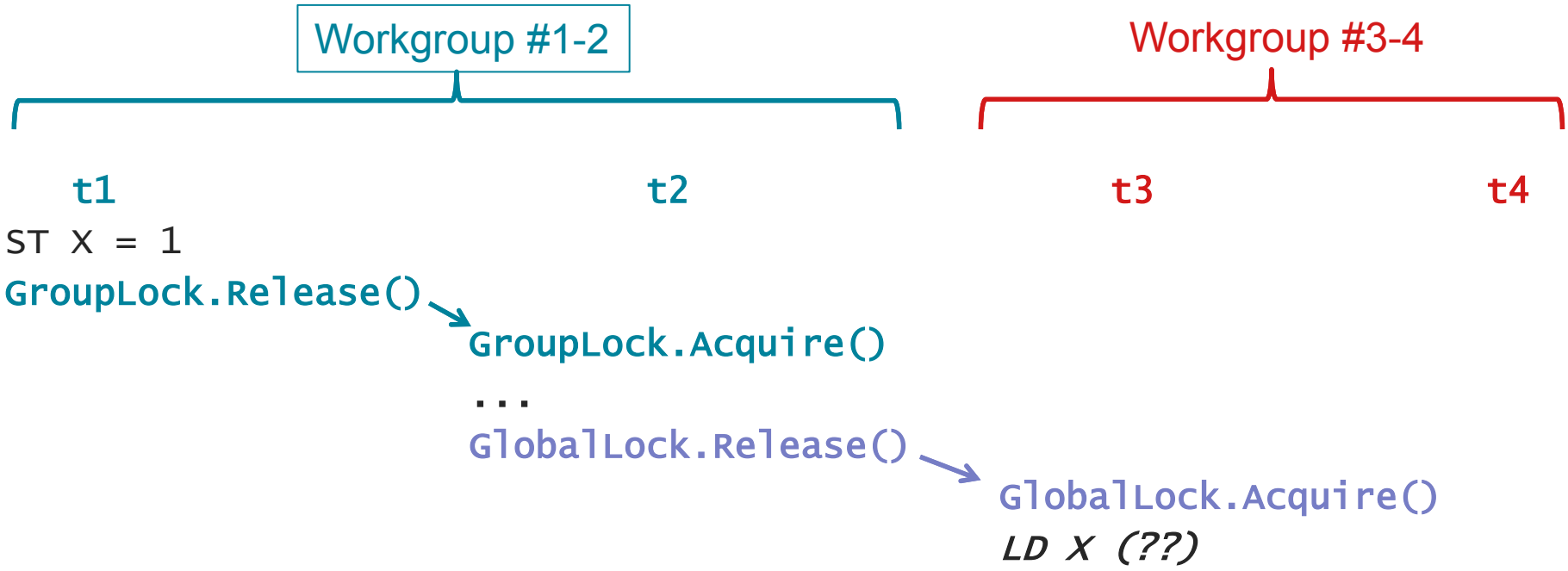
t1 t2 t3 t4

```
ST X = 1
GroupLock.Release()
                    GroupLock.Acquire()
                    ...
                    GlobalLock.Release()
                                        GlobalLock.Acquire()
                                        LD X (??)
```

DATA-RACE-FREE IS NOT ENOUGH

AMD

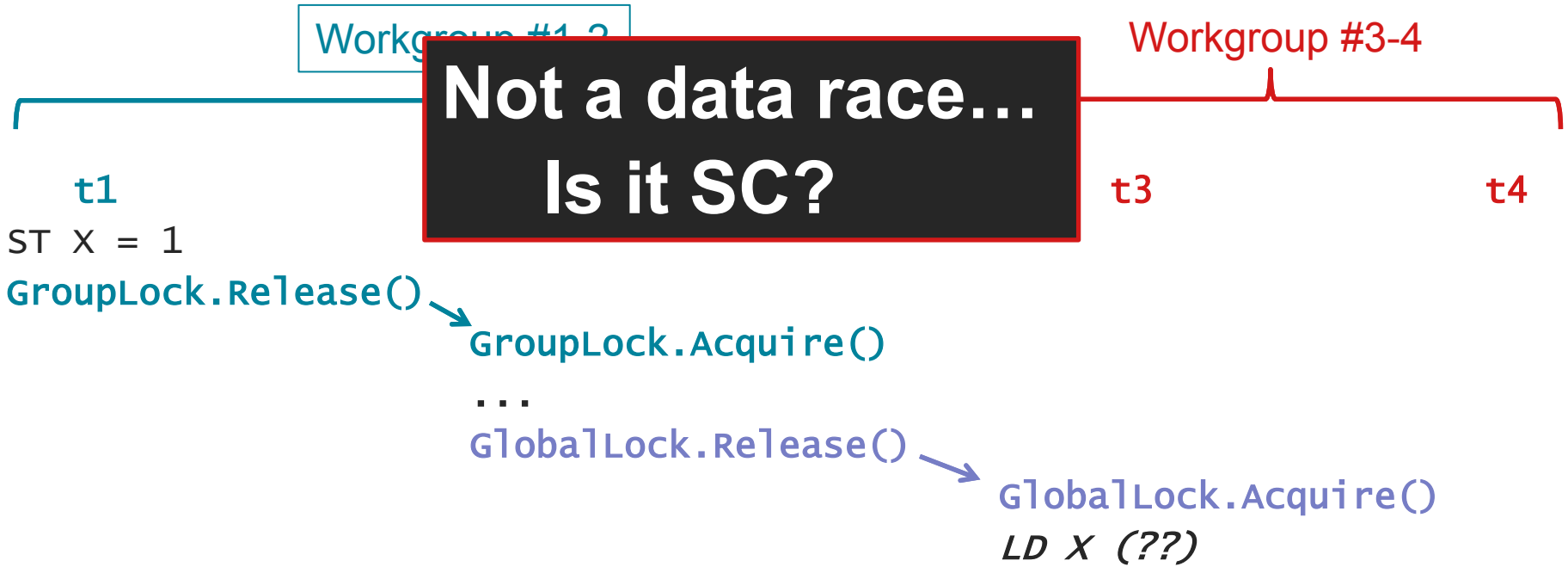
- Two ordinary memory accesses participate in a **data race** if they
  - Access same location
  - At least one is a store
  - Can occur simultaneously



DATA-RACE-FREE IS NOT ENOUGH

AMD

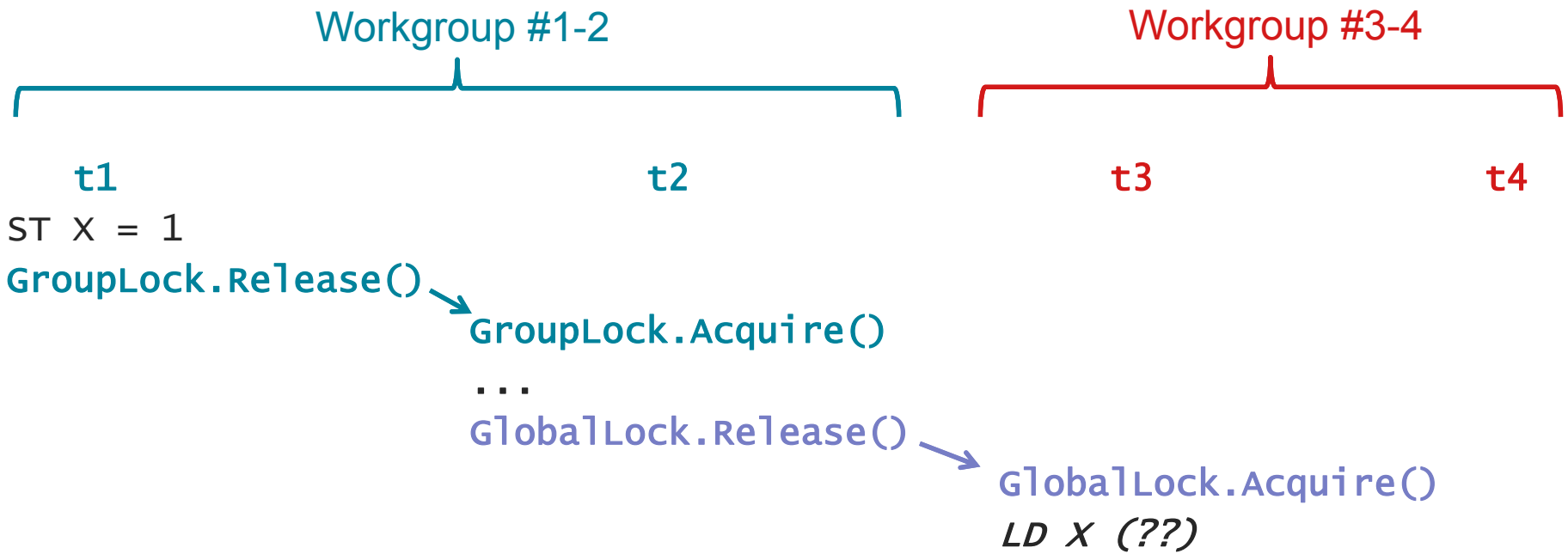
- Two ordinary memory accesses participate in a **data race** if they
  - Access same location
  - At least one is a store
  - ~~Can occur simultaneously~~



WHAT WILL HAPPEN?

CUDA™ `__threadfence{_block}` definition:

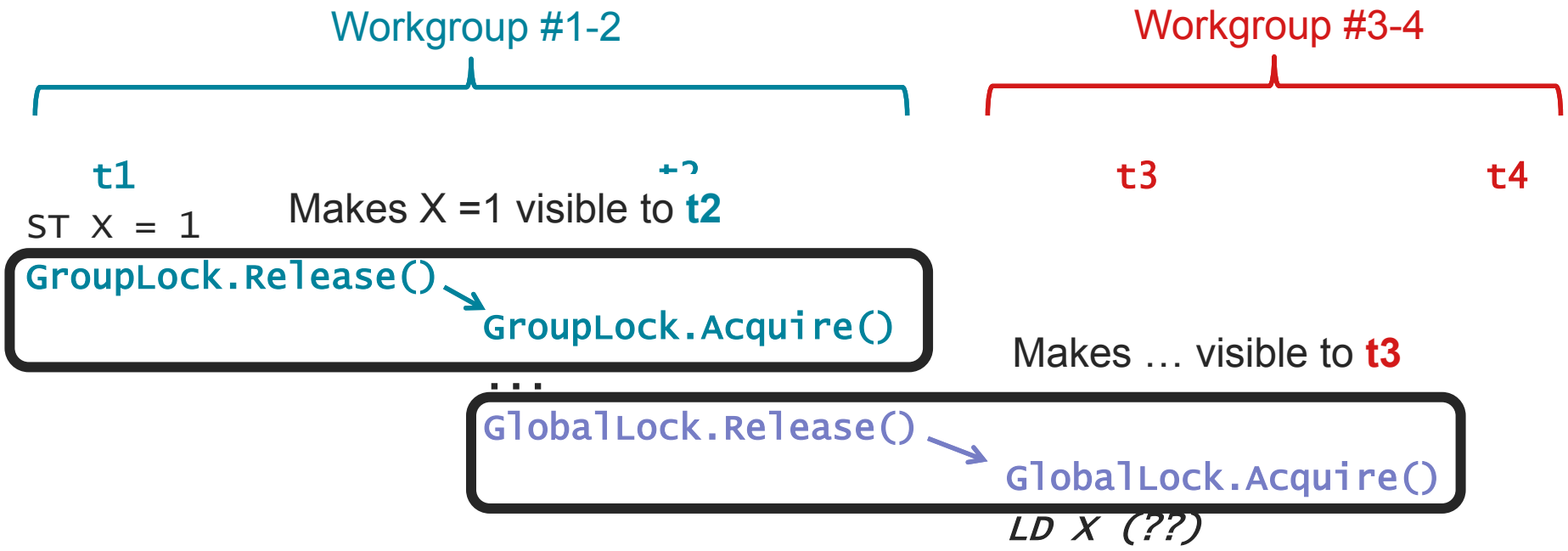
> "waits until all…memory accesses **made by the calling thread** prior to [{Group, Global}Lock.Release()] are visible to…all threads in the {group, device}"



WHAT WILL HAPPEN?

AMD

Heterogeneous Race!



SEQUENTIAL CONSISTENCY FOR DATA-RACE-FREE

AMD

- Two memory accesses participate in a **data race** if they
  - access the same location
  - at least one access is a store
  - can occur simultaneously
    - i.e. appear as adjacent operations in interleaving.
- A program is **data-race-free** if no possible execution results in a data race.
- Sequential consistency for data-race-free programs
  - Avoid everything else

SEQUENTIAL CONSISTENCY FOR HETEROGNEOUS-RACE-FREE

AMD

- Two memory accesses participate in a **heterogeneous race** if
  - access the same location
  - at least one access is a store
  - can occur simultaneously
    - i.e. appear as adjacent operations in interleaving.
  - Are not synchronized with “enough” scope
- A program is **heterogeneous-race-free** if no possible execution results in a heterogeneous race.
- Sequential consistency for heterogeneous-race-free programs
  - Avoid everything else

A FIRST CUT

AMD

HRF0: Basic Scope Synchronization

A FIRST CUT

AMD

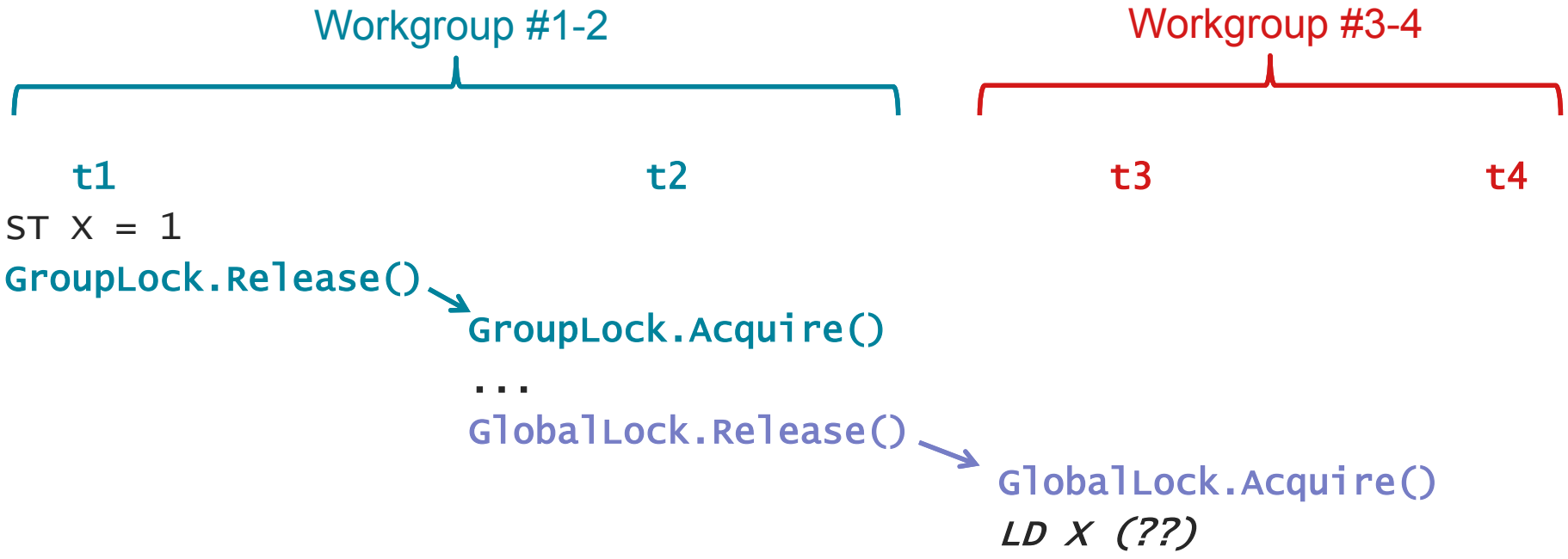
- **HRF0: Basic Scope Synchronization**
  - "enough" = both threads synchronize using ***identical*** scope

A FIRST CUT

AMD

- **HRF0: Basic Scope Synchronization**
  - "enough" = both threads synchronize using ***identical*** scope

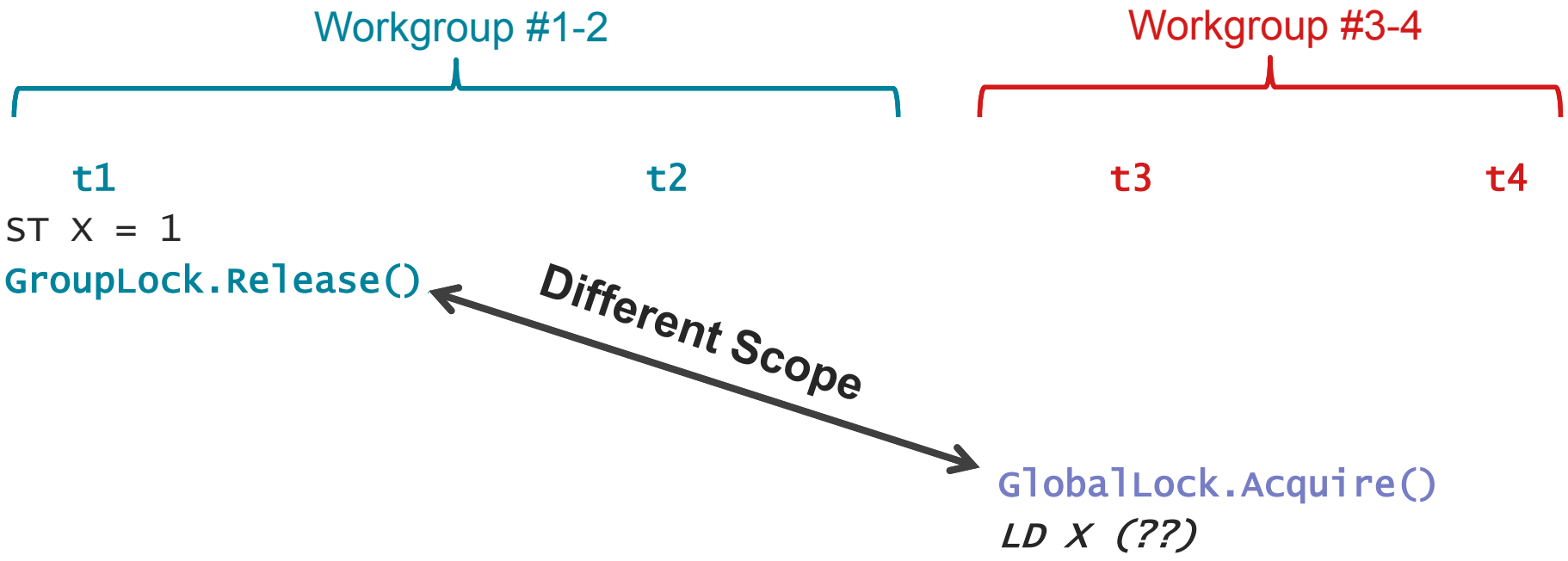
- Recall example:
  - Contains a heterogeneous race in HRF0



AMD

- **HRF0: Basic Scope Synchronization**
  - "enough" = both threads synchronize using ***identical*** scope

- Recall example:
  - Contains a heterogeneous race in HRF0

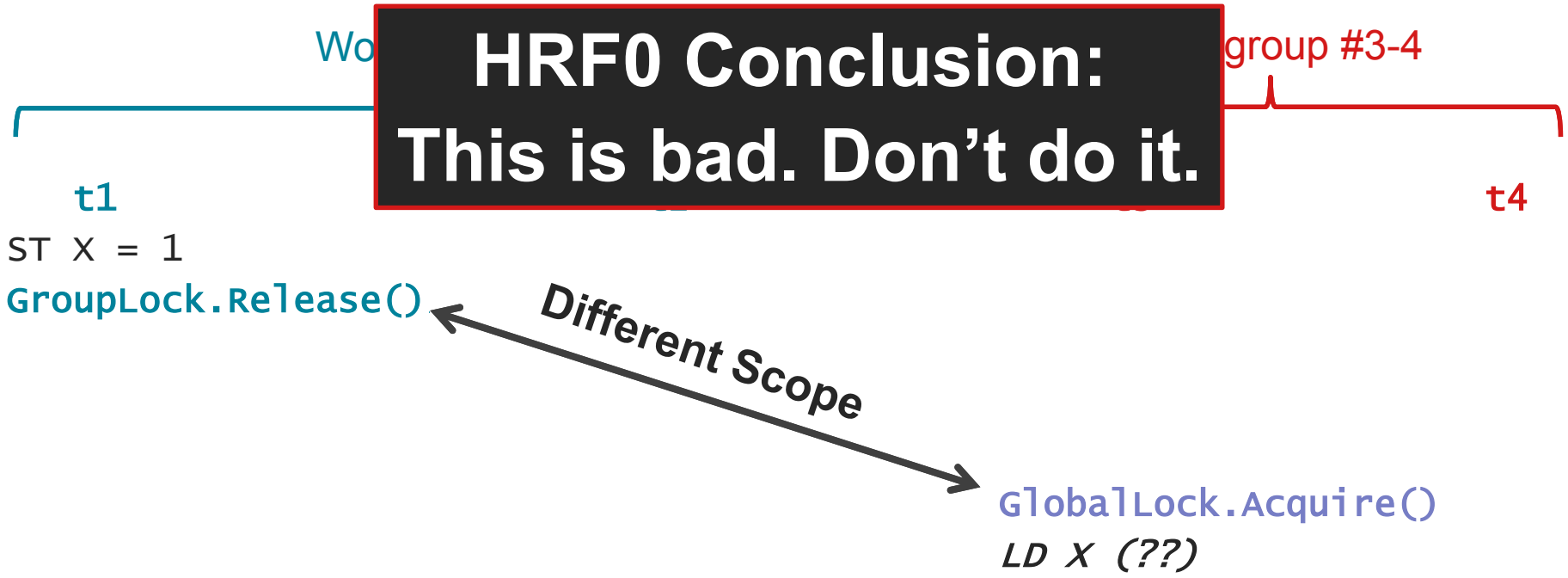


A FIRST CUT

AMD

- **HRF0: Basic Scope Synchronization**
  - "enough" = both threads synchronize using ***identical*** scope

- Recall example:
  - Contains a heterogeneous race in HRF0



IS HRF0 USEFUL?

AMD

- Want: For performance, use smallest scope possible
  - What is safe in HRF0?

Use smallest scope that includes all producers/consumers of shared data

HRF0 Scope Selection Guideline

IS HRF0 USEFUL?

AMD

- Want: For performance, use smallest scope possible
  - What is safe in HRF0?

Use smallest scope that includes all producers/consumers of shared data

HRF0 Scope Selection Guideline

Implication:

Producers/consumers must be known at synchronization time

IS HRF0 USEFUL?

AMD

- Want: For performance, use smallest scope possible
  - What is safe in HRF0?

Use smallest scope that includes all producers/consumers of shared data

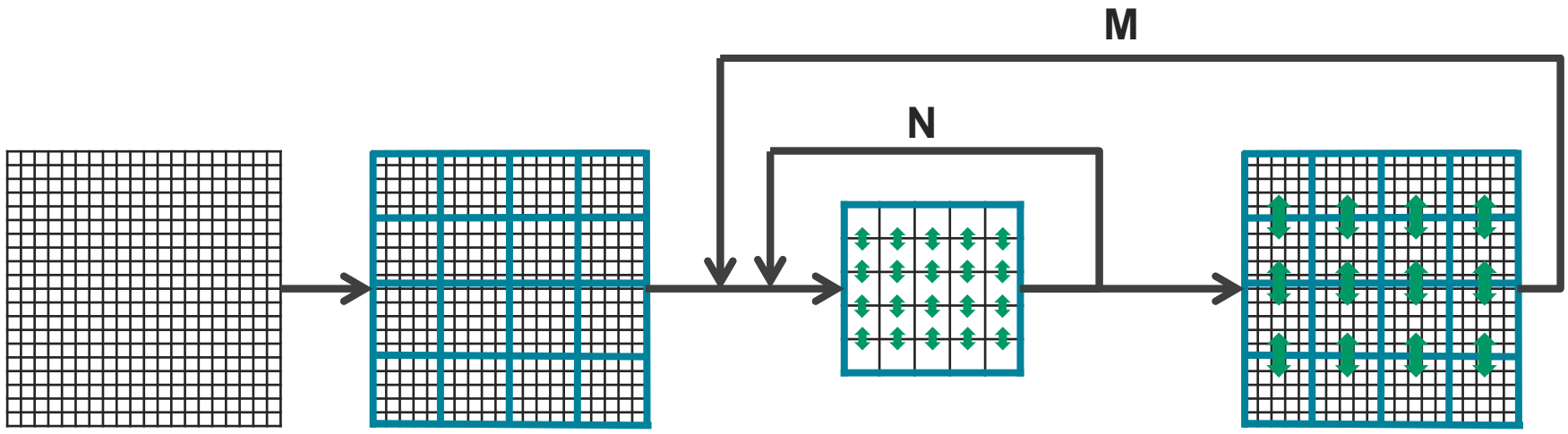
HRF0 Scope Selection Guideline

Is this a valid assumption?

Implication:

Producers/consumers must be known at synchronization time

REGULAR GPGPU WORKLOADS

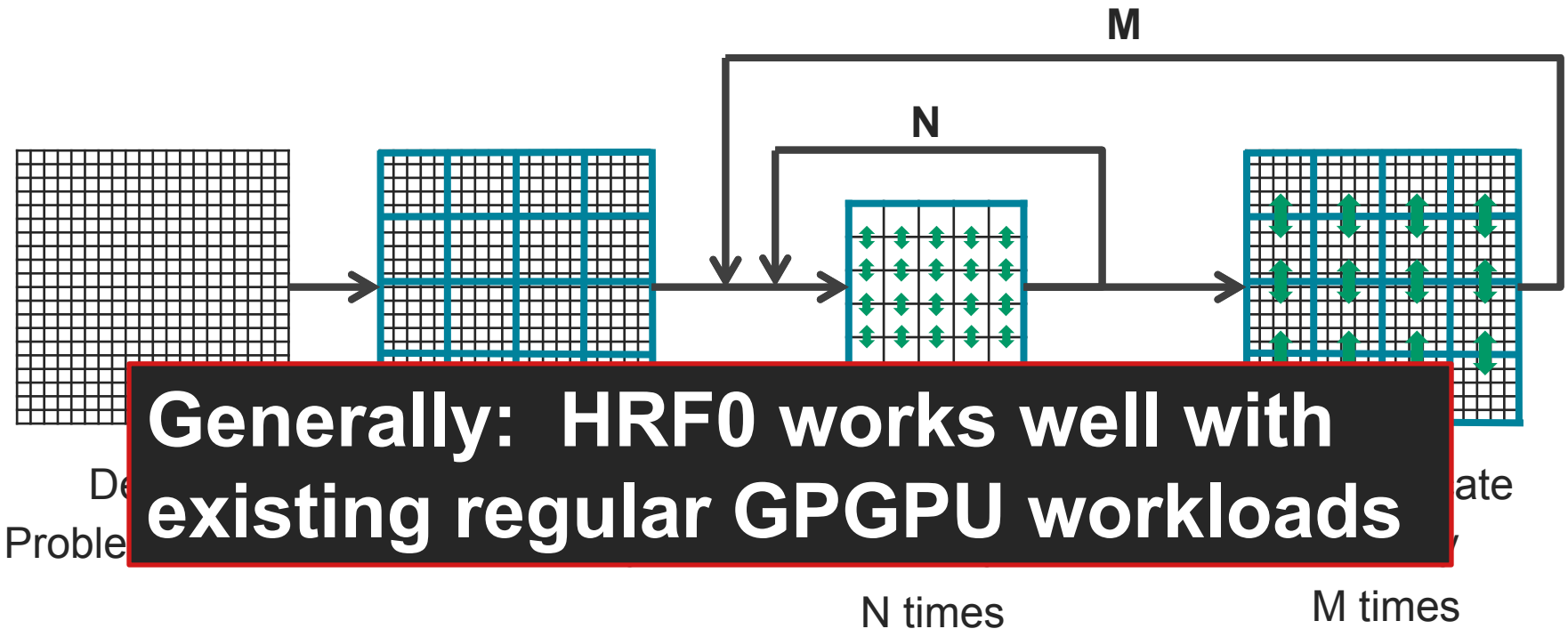


Well defined (regular) data partitioning +

Well defined (regular) synchronization pattern =

- *Producer/consumers are always known*

REGULAR GPGPU WORKLOADS



Generally: HRF0 works well with existing regular GPGPU workloads

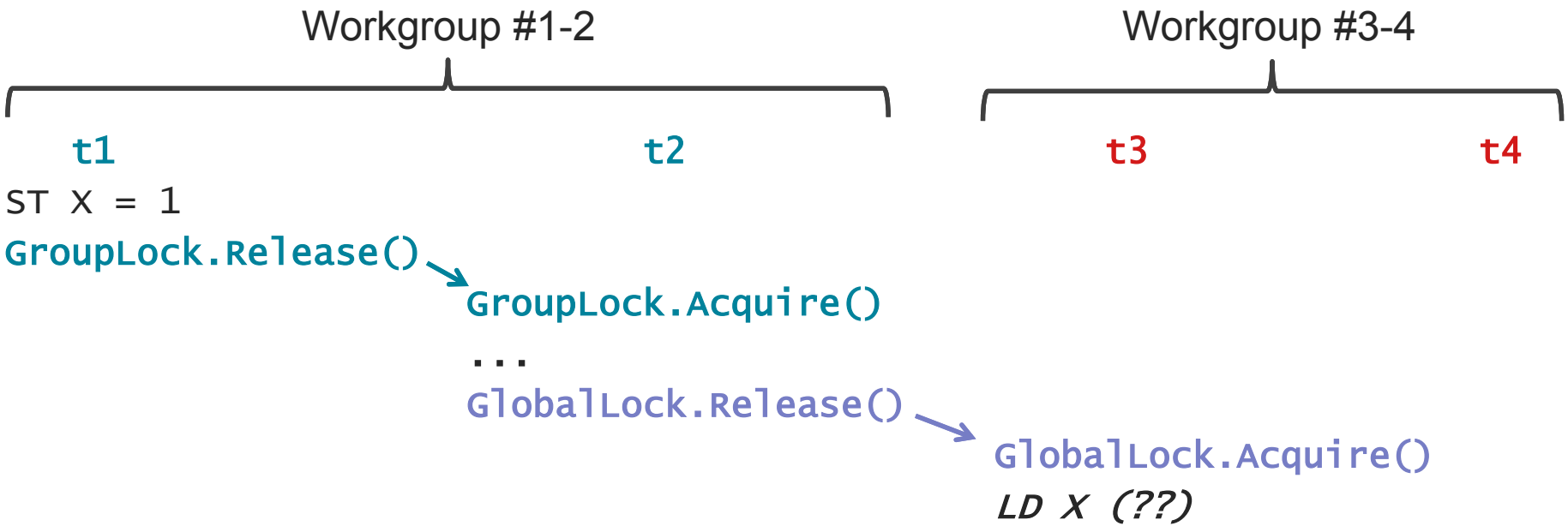
Well defined (regular) data partitioning +

Well defined (regular) synchronization pattern =

- *Producer/consumers are always known*

IRREGULAR GPGPU WORKLOADS

- HRF0: example is race
  - Must upgrade Group -> Global
- Current hardware:
  - LD X will see value (1)!



HRFX: THE NEXT ITERATION

AMD

- HRF0 is overly conservative on existing HW
  - Makes fast irregular parallelism hard
- Other HRF definitions are possible
  - e.g., define behavior when different scopes interact
- What are the gotchas? (there will be many…)

The answer: ???

CONCLUSIONS & FUTURE DIRECTIONS

AMD

- GPUs Currently expose ad-hoc scoped synchronization
- From CPU world: mask low-level details with SC for DRF
  - GPU scope synchronization incompatible w/ SC for DRF
- GPUs: SC for HRF
  - HRF0: Basic scope synchronization
    - + Easy-ish to define/understand
    - + Safe interpretation of existing models
    - + Permits most HW optimizations
    - − Prohibits some SW opts in current hardware
  - HRFx: models to exploit hierarchy
    - What happens when different scopes interact?
- *Let's not wait 30 years this time*

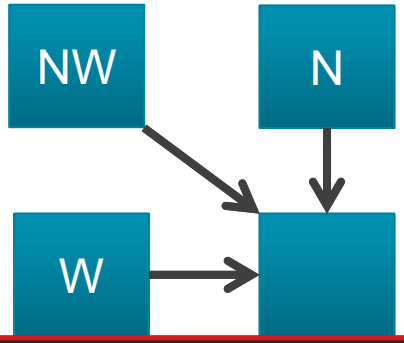
AMD

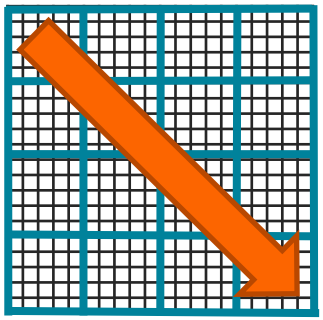
THANKS!
QUESTIONS?

AMD

BACKUP

BASIC APPLICATION EXAMPLE – SEQUENCE ALIGNMENT



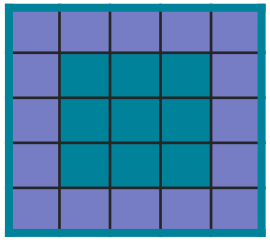


Generally: HRF0 works well with existing regular GPGPU workloads

Well defined (regular) synchronization pattern +

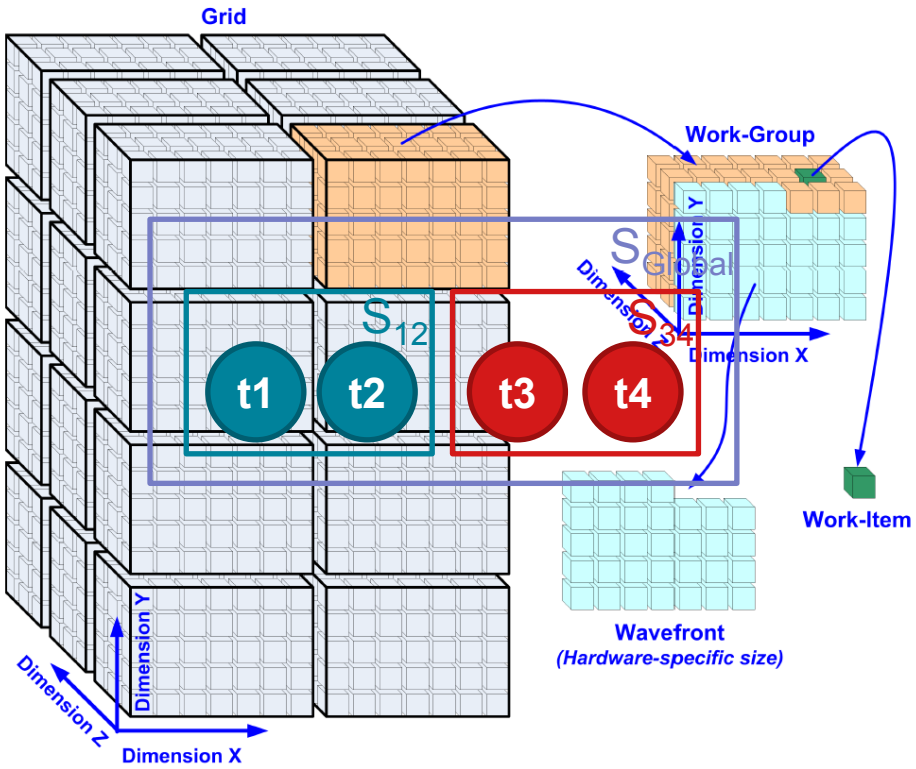
Well defined (regular) data partitioning =

- *Producer/consumers are always known*



SCOPED SYNCHRONIZATION

- Some operations are not global, have visibility limited to *workgroup* or *device*
  - **HSAIL:** `st_{rel, part_rel}, ld_{acq, part_acq}`, etc.
  - **CUDA™:** `threadfence_{block, system}`, `__syncthreads`, etc.
  - **PTX**: `membar.{cta, gl, sys}`, `bar`, etc.

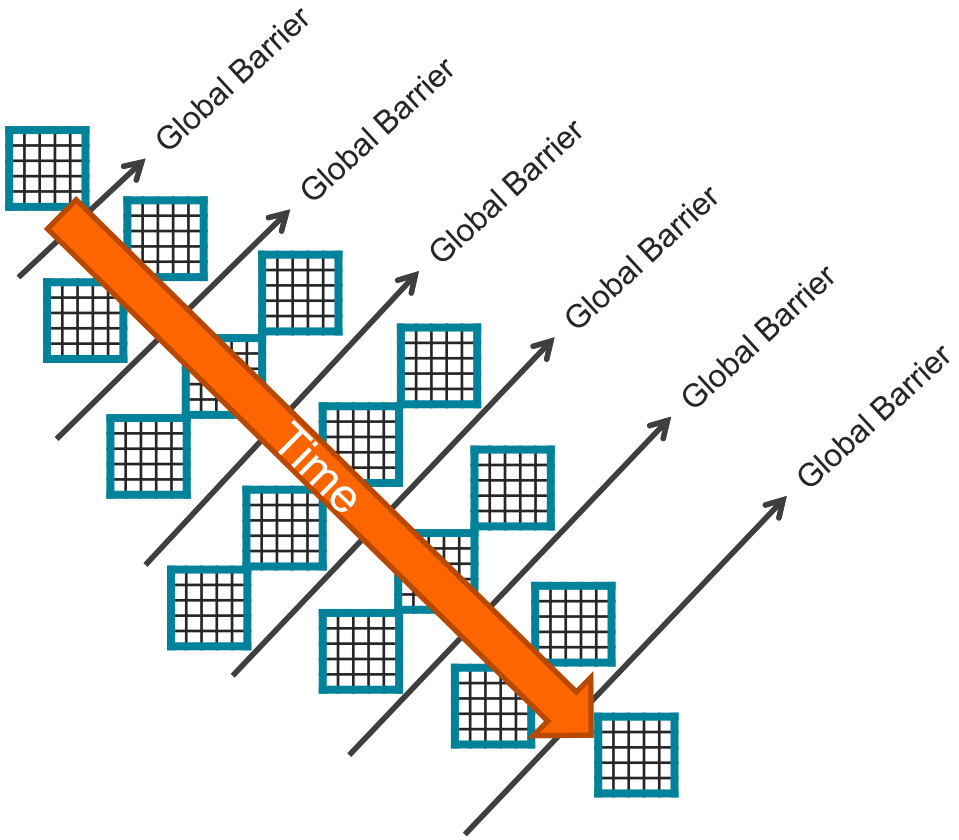


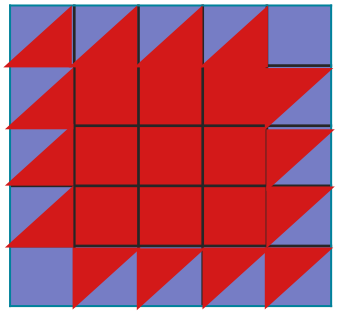
EXECUTIVE SUMMARY

AMD

- GPUs: streaming memory system, hierarchical programming model
  - Use partial (scoped) synchronization
  - Existing GPU memory models ambiguous, dense for programmers
- From the CPU world: SC for DRF
  - Relaxed HW, precise semantics, programmer-friendly
  - *Let's not wait 30 years this time*
- This work: apply same principle to GPU world
  - Sequential Consistency for Heterogeneous-Race-Free (SC for HRF)
- **HRF0**: Basic scope synchronization
  - Two threads communicate → use *identical* scope synchronization
  - Works well with existing GPU codes
- Others possible. HRF0 is:
  - Difficult to use efficiently w/ irregular synchronization
  - Overly conservative *for current implementations*

INSERTING SCOPED SYNCHRONIZATION





```
Acquire_S?
cell[id] = fn(cell[N],
              cell[NW],
              cell[W])
Release_S?
```

Synchronization Scope:

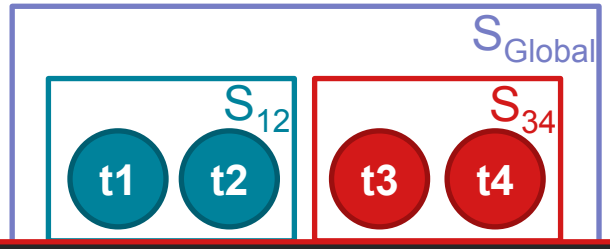
WF on N, W edge of WG use **global** acquire

WF on S, E edge of WG use **global** release

All other sync is **local**

EXAMPLE AMBIGUITY

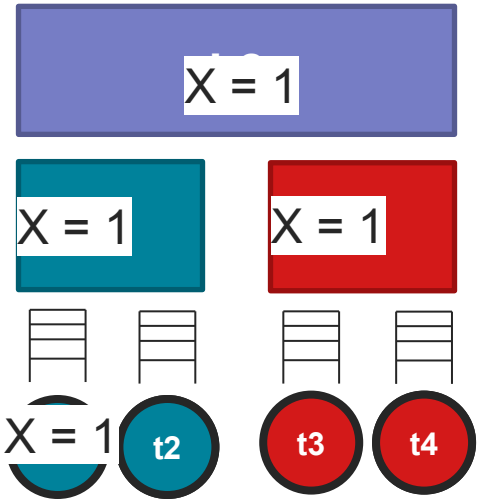
- What value of X does t3 see?



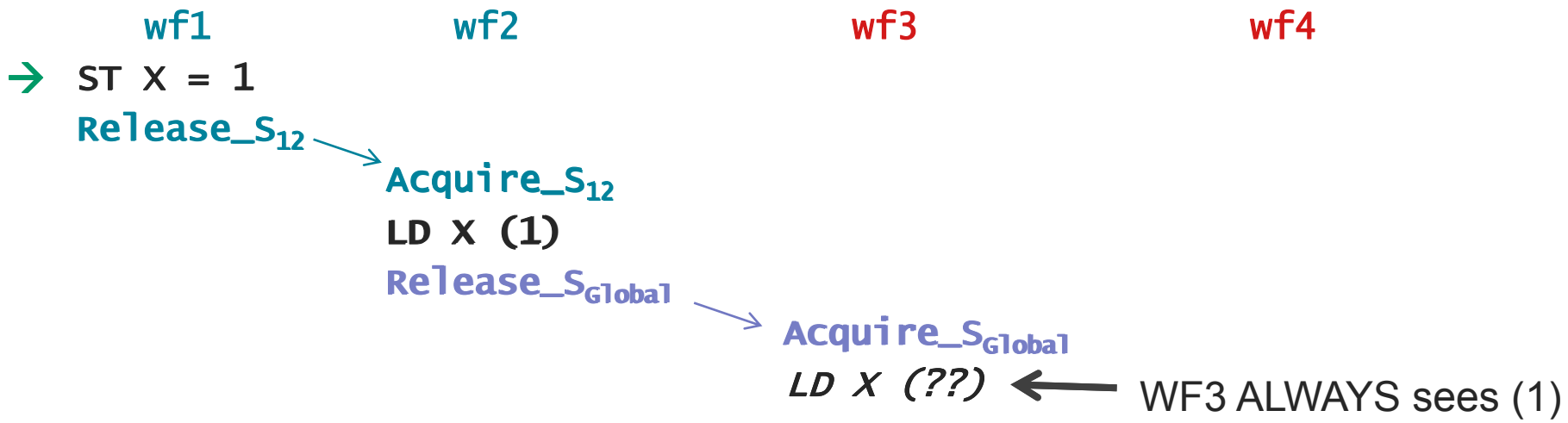
Answer not obvious -- depends on system

t1	t2	t3	t4
ST X = 1			
$Release\_S_{12}$ →	$Acquire\_S_{12}$		
	LD X (1)		
	$Release\_S_{Global}$ →	$Acquire\_S_{Global}$	
		LD X (??)	

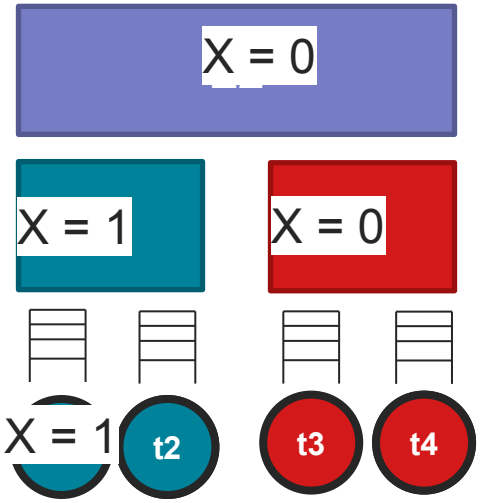
EXAMPLE: CONVENTIONAL BASE SYSTEM



- Conventional write-through/combining cache hierarchy:
  - Local release → flush stores from queue
  - Local acquire → stall until queue is empty
  - Global acquire → Invalidate *all valid locations* in L1 cache
  - Global release → Flush *all dirty locations* in L1 cache



EXAMPLE: OPTIMIZED BASE SYSTEM



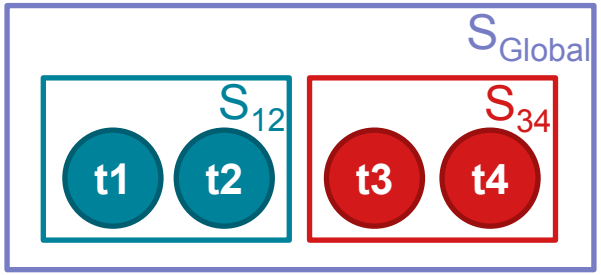
- Optimized write-combining cache hierarchy
  - Local release → flush stores from queue
  - Local acquire → stall until queue is empty
  - Global acquire → Inv. *locations read by acquiring WF* in L1
  - Global release → Flush *locations written by releasing WF* in L1

```
t1                t2                t3                wf4
ST X = 1
Release_S12 ──→
                  Acquire_S12
                  LD X (1)
                  Release_SGlobal ──→
                                    Acquire_SGlobal
                                    LD X (??) ←── WF3 sees (0)
```

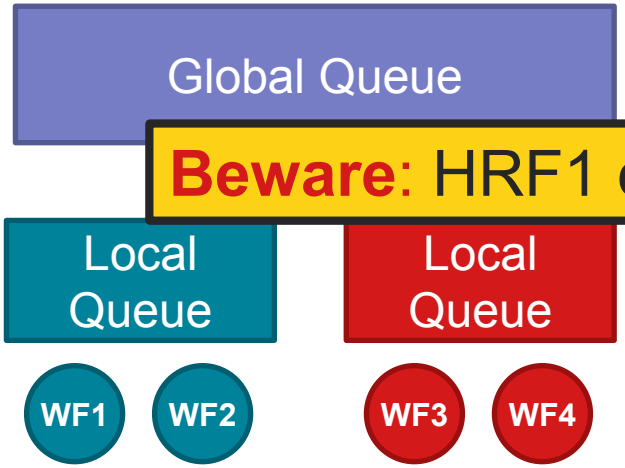
ASSUMPTIONS/SIMPLIFICATIONS

AMD

1. Ignore scratchpad (local/group/shared) memory
   - i.e., all memory in single, global shared address space
2. Use scoped **acquire/release** synchronization
   - Generalizes to other forms of synchronization
3. Examples: two-level scope hierarchy with simple threads



APPLICATION EXAMPLE – TASK PARALLEL RUNTIME



Hierarchical task queue:

- WF produce/consume tasks independently
- Use local queue until:
  - Local queue empty → pull from global

Beware: HRF1 consumes significant brain power

Observation:

- Push/pull performed on behalf of **workgroup**

HRF0:

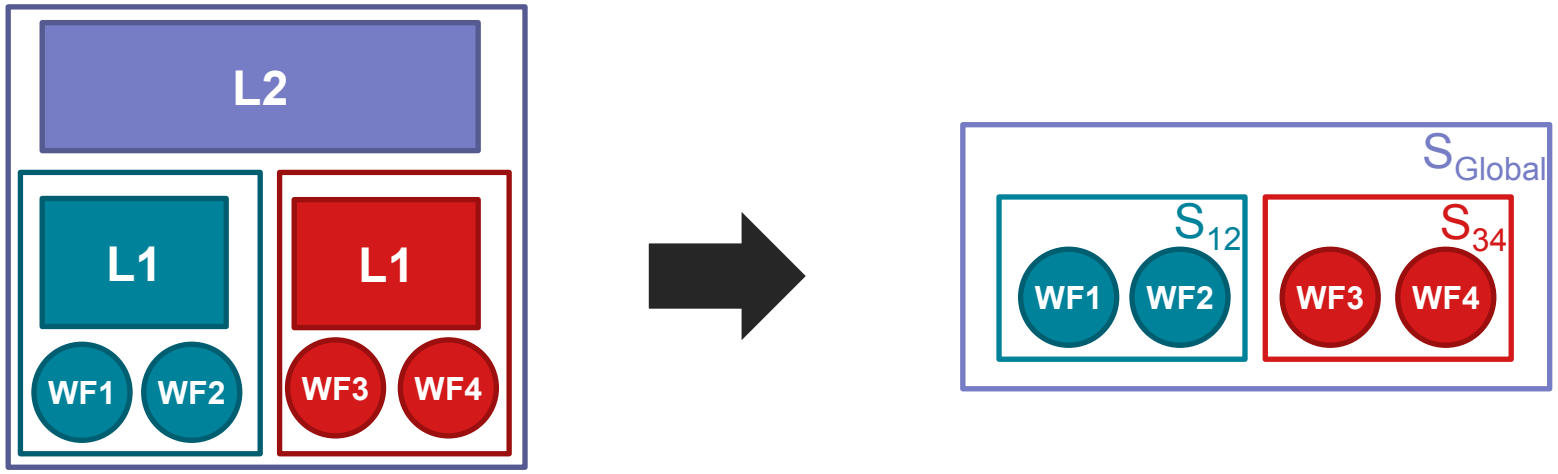
- Either all sync has to be global or WFs coordinate to push/pull

HRF1:

- Single WF can push/pull independently

SCOPES

- Scope: A subset of threads
- **Scoped synchronization**: synchronization w.r.t. a scope
  - **OpenCL™**: `mem_fence`
  - **HSAIL**: `st_{rel, part_rel}`, `ld_{acq, part_acq}`, etc.
  - **CUDA™**: `threadfence_{block, system}`, `__syncthreads`, etc.
  - **PTX**: `membar.{cta, gl, sys}`, `bar`, etc.
- Scopes introduce new class of races:
  - What happens when threads (wavefronts/warps) use different scopes?



DISCLAIMER & ATTRIBUTION

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION