CS 758: Advanced Topics in Computer Architecture

Lecture #8: The GPU Memory System (Part 2)

Professor Matthew D. Sinclair

Backup of these slides were developed by Tim Rogers at the Purdue University.

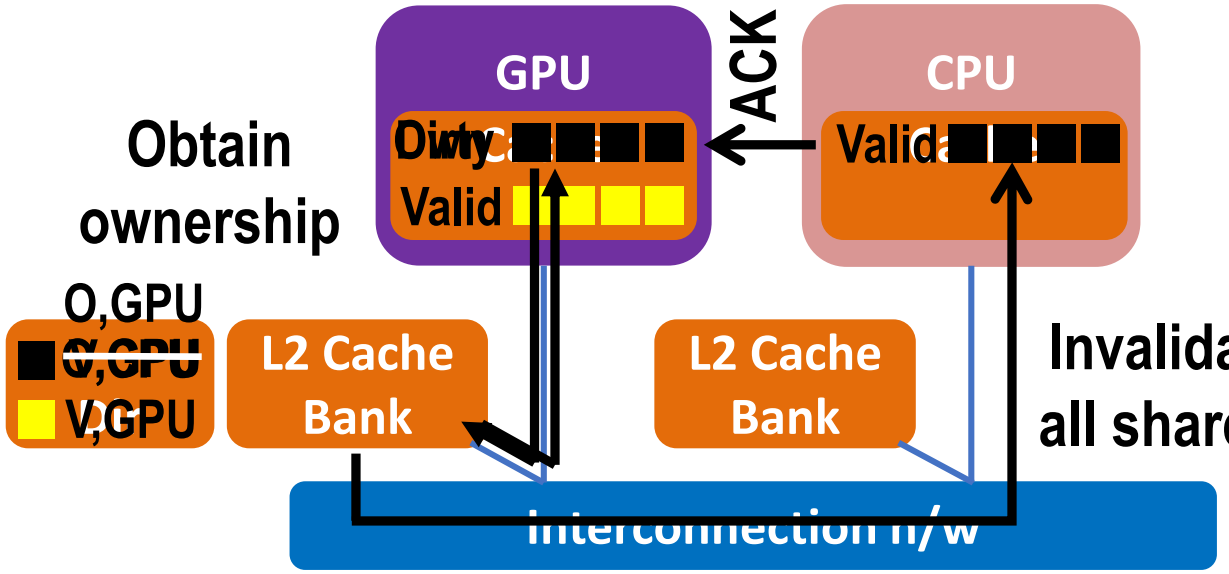
Slides enhanced by Matt Sinclair

In-Class Activity

- With a partner, work on the 3 problems for 5 minutes
  - Problem 1: GPU MCM
  - Problem 2: GPU MCM with scopes
  - Problem 3: bugs with writing synchronization code on GPUs

- Then we'll come together as a group and discuss

CPU Coherence: MESI




```
// each thread
for i = r[tid]:r[tid+1]
    LOCK
    LD R1, A[i];
    LD R2, B[i];
    R3 ← Math(R1, R2);
    ST B[i], R3;
    UNLOCK
```

- Write miss: Get ownership, invalidate all sharers
- Read miss: Update sharers list
- Synchronization points are cheap
- BUT poor fit for GPUs: [Singh13, Hechtman14]
  - Directory overhead, transient states, excessive traffic, indirection

Complex coherence, simple consistency

Atomics Background

- Default: Data-race-free-0 (DRF0) [ISCA '90]
  - Identify all races as synchronization accesses (C++: atomics)

```
// each thread
for i = 0:n
    …
    ADD R4, A[i], R1     synch (atomic)
    ADD R5, B[i], R1     synch (atomic)
```

  - All atomics order data accesses
  - Atomics order other atomics

⇒Ensures SC semantics if no data races

Atomics Background (Cont.)

- Default: Data-race-free-0 (DRF0) [ISCA ‘90]
  - All atomics order data accesses
  - Atomics order other atomics

  ⇒Ensures SC semantics if no data races
- Data-race-free-1 (DRF1): unpaired atomics [TPDS ‘93]

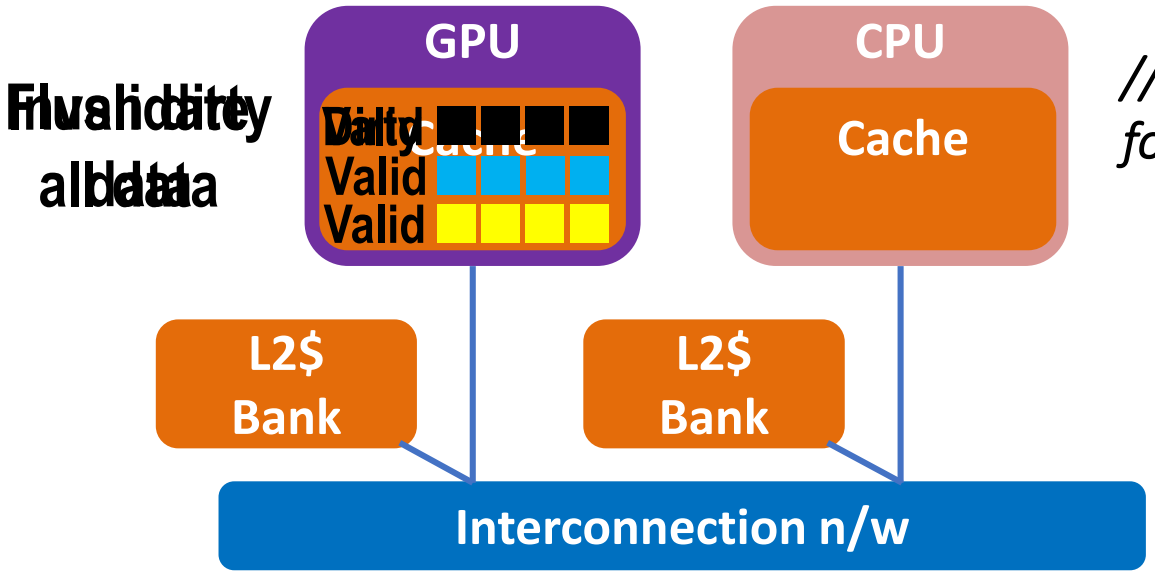
  + Unpaired atomics do not order data accesses
  - Atomics order other atomics

  ⇒Ensures SC semantics if no data races
- Relaxed atomics [PLDI ‘08]

  + Do not order data or other atomics

  ⇒But can violate SC and no formal specification

Traditional GPU Coherence




```
// unique per thread
for i = start:end
    LD R1, A[i];
    LD R2, B[i];
    R3 ← Math(R1, R2);
    ST C[i], R3;
```

Each thread accesses independent data (no races)

No data reuse or data sharing

Coarse-grained synchronization

Optimized for streaming, data parallel applications

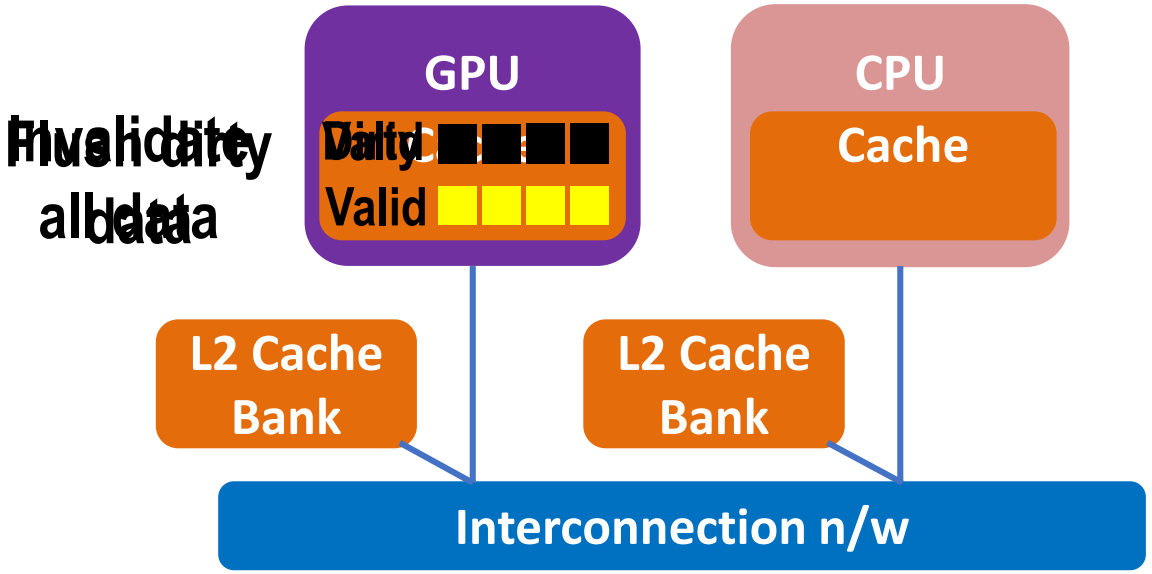
GPU Memory Consistency Model

- Active area of research
- Tightly tied in with coherence protocol

- Provides very weak guarantees
  - Respect program order within a single thread
  - Easy to design hardware
  - Programmers add ***fences*** to provide extra guarantees
    - Fence guarantee all previous accesses are visible before proceeding
    - … usually

- Most GPUs use a ***scoped*** memory consistency model
  - __threadfence_block – local synchronization (usually at L1)
  - __threadfence – GPU global synchronization (usually at L2)
  - __threadfence_system – CPU-GPU global synchronization (flush GPU)

GPU Coherence with DRF

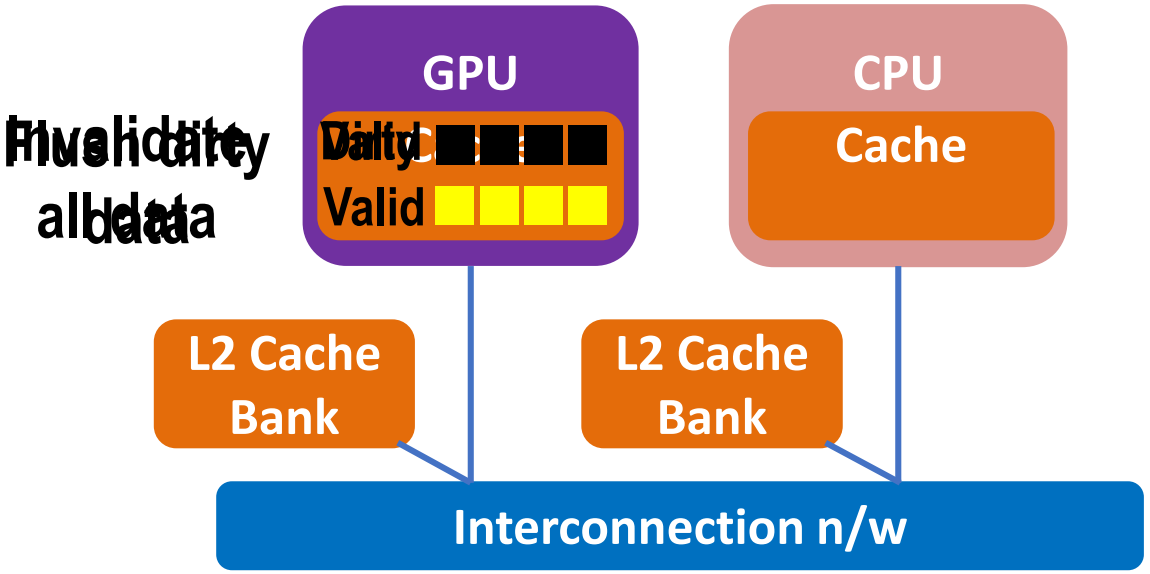



```
// each thread
for i = r[tid]:r[tid+1]
    LOCK
    LD R1, A[i];
    LD R2, B[i];
    R3 ← Math(R1, R2);
    ST B[i], R3;
    UNLOCK
```

- With data-race-free (DRF) memory model
  - No data races; synchs must be explicitly distinguished
  - Synchronization accesses (atomics) go to last level cache (LLC)
  - Synchronization points are expensive, preclude reuse

Simple but inefficient coherence, simple consistency

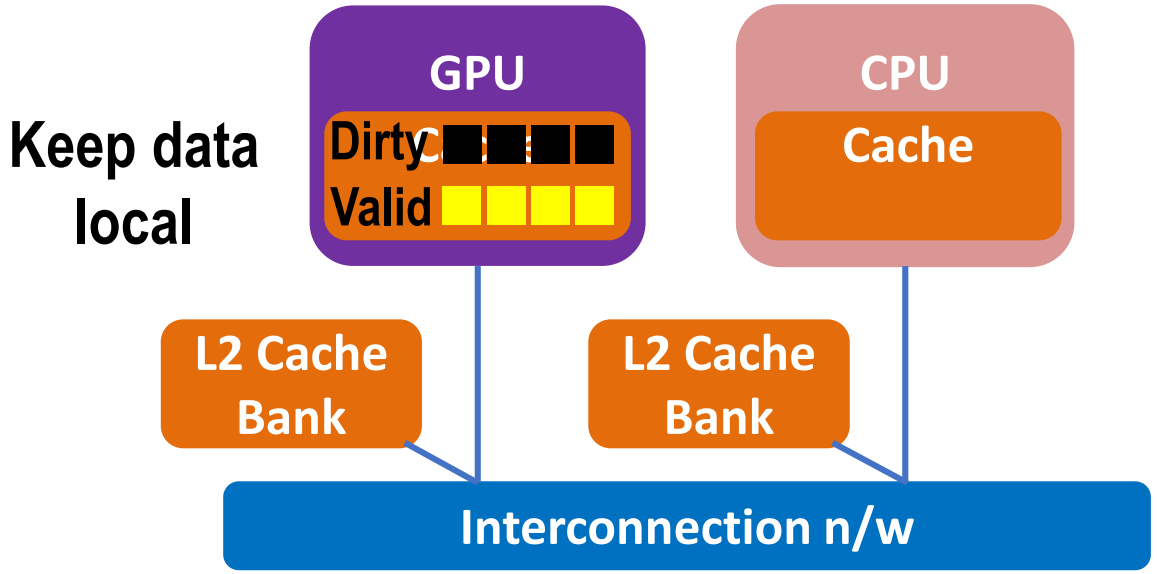
GPU Coherence with HRF




```
// each thread
for i = r[tid]:r[tid+1]
    LOCK            global
    LD R1, A[i];    ■
    LD R2, B[i];    ■
    R3 ← Math(R1, R2);
    ST B[i], R3;
    UNLOCK          global
```

- New memory model: Heterogeneous-race-free (HRF) [ASPLOS '14]
  - Adds scoped synchronization

GPU Coherence with HRF




```
// each thread
for i = r[tid]:r[tid+1]
    LOCK                global
    LD R1, A[i];
    LD R2, B[i];
    R3 ← Math(R1, R2);
    ST B[i], R3;
    UNLOCK              global
```

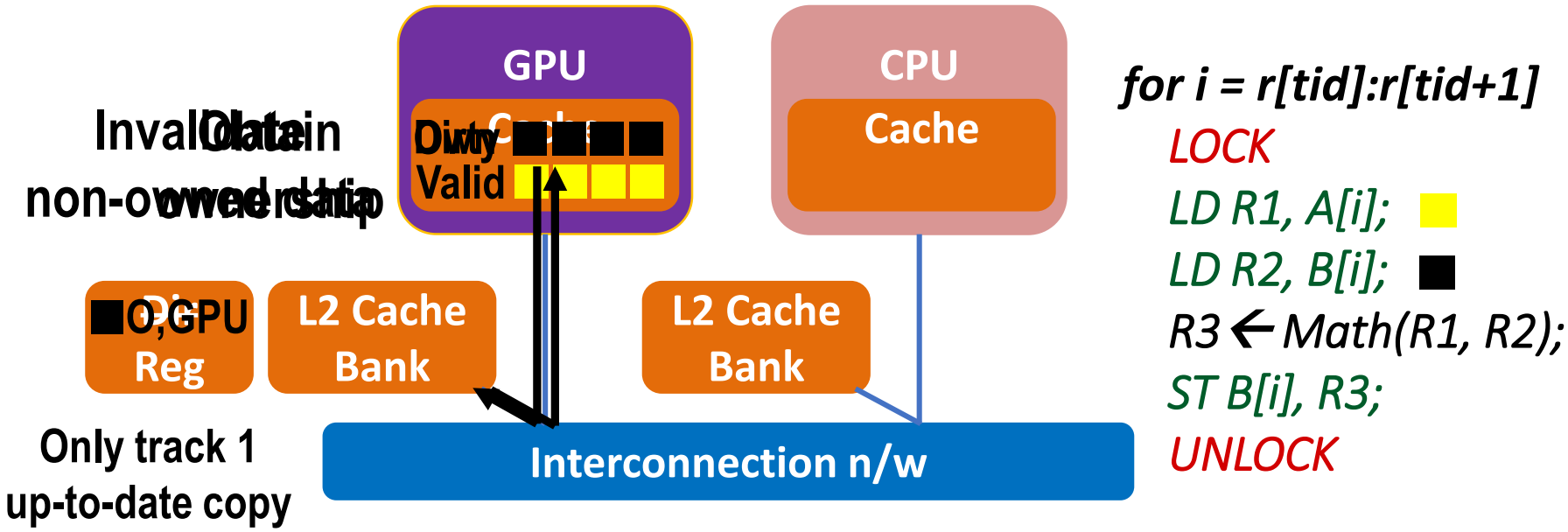
- New memory model: Heterogeneous-race-free (HRF) [ASPLOS '14]
  - Adds scoped synchronization
  - No overhead for locally scoped synchronizations
- But higher programming complexity

More efficient coherence, complex consistency

GPU Coherence with HRF

- With ~~data~~ race-free (~~DRF~~) memory model **[ASPLOS '14]**
  - No ~~data~~ races; synchs must be explicitly distinguished
    - At all synch points
      - Flush all dirty data: Unnecessary writethroughs
      - Invalidate all data: Can't reuse data across synch points
    - Synchronization accesses must go to last level cache (LLC)
    - No overhead for locally scoped synchs
- But higher programming complexity

DeNovo Coherence with DRF



- Reuse dirty data across synch points – more data reuse
- Synchronization accesses can be performed at L1 – synch reuse

3% area overhead vs. GPU Coherence + HRF

Efficient coherence, simple consistency (except relaxed atomics)

Additional Topics

- Extending Coherence across accelerators
  - CCIX, ACE, Spandex, CXL,
  - Challenge: Different accelerators have different coherence requirements
    - E.g., different data widths
  - Especially important as number of accelerators increases
- Multi-GPU/Multi-Chiplet Coherence
  - Challenge: NUMA accesses (ala SMPs), page migration, partitioning
  - Potentially can extend existing mechanisms
- DRFrlx [Sinclair ISCA '17]
  - Extend MCM to provide sane semantics for relaxed atomics
- Timestamps
  - Avoid traditional coherence overheads, but some bookkeeping/delay instead

Additional Topics (Cont.)

- Coherence Granularity
  - HSC [Power MICRO ‘13] – large granularity for traditional GPGPU apps
  - hUVM [Koukos TACO ‘16] – page granularity coherence
    - Exploits traditional GPGPU apps nature
  - Some relationship with subsequent work on ACE, CCIX, Spandex, etc.
- Coherent Scratchpads
  - Best of both worlds!
- Remote Scopes [Orr ASPLOS ‘15]
  - Dynamically vary scope granularity to reduce overhead
- hLRC [Alsop MICRO ‘17]

Conclusion

- GPU coherence and consistency are hot, recent topics
  - Lots of ongoing research in the area
- Goal: avoid replicating 20-30 years of CPU MCM research
- Idea: evolve from that CPUs already have now
  - Design Goal: keep GPU coherence protocols simple
  - GPU apps don’t need more complex coherence protocols
  - This has implications on MCM
- State-of-the-art (products): GPU + HRF
  - CPU-GPU coherence in real products assumes this for the GPU component

Backup

Coherence & Consistency Qualitative Analysis

Coherence + Consistency	Reuse Data		Do Synchs at L1
	Owned	Valid	
GPU + DRF(GD)	X	X	X
GPU + HRF(GH)	local	local	local
DeNovo + DRF (DD)	✓	X	✓
DeNovo-RO + DRF (DD+RO)	✓	read-only	✓
DeNovo + HRF (DH)	✓	local	✓

Incorporating Other Use Cases Into DRFrlx

Work Queues

Seqlocks

Flags

Ref Counters

How to incorporate SC violations in approximable applications?

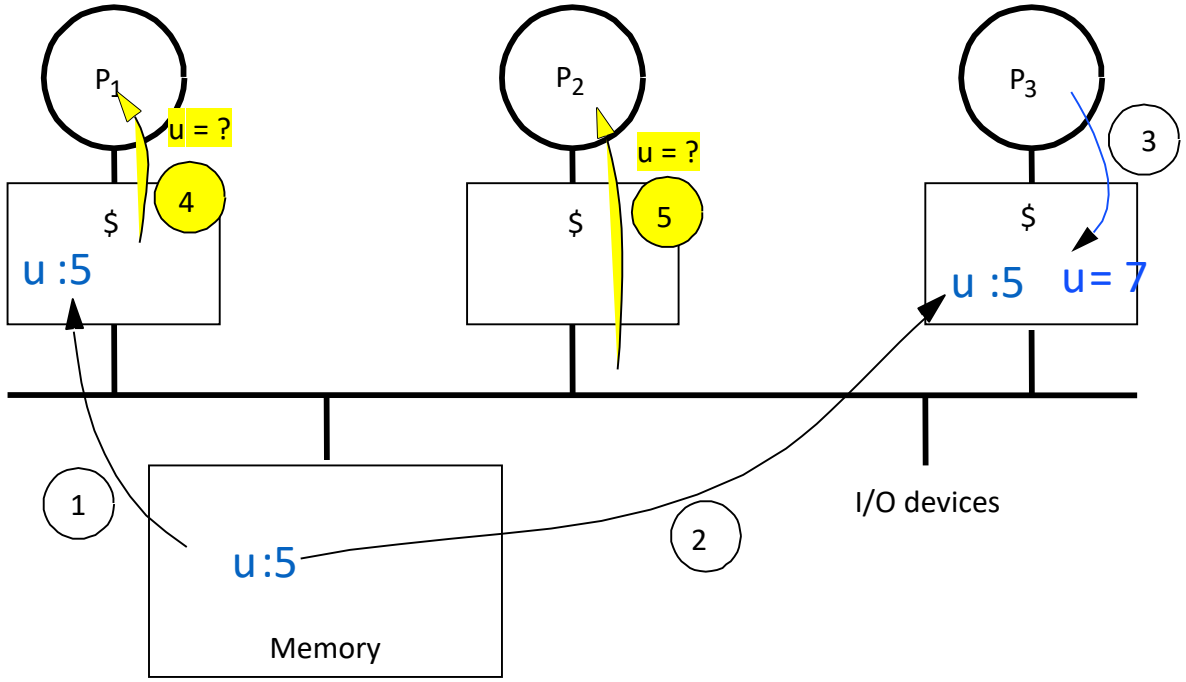
How to incorporate overlapped atomics that do not order?

How to incorporate violations of SC that do not affect final result?

How to incorporate relaxed atomics that do not order data?

Use Case	Category	Semantics
Work Queues Flags	Unpaired Non-Ordering	SC
Event Counters Seqlocks	Commutative Speculative	Final result always SC
Split Counters Ref Counters	Quantum	SC-centric: non-SC parts isolated

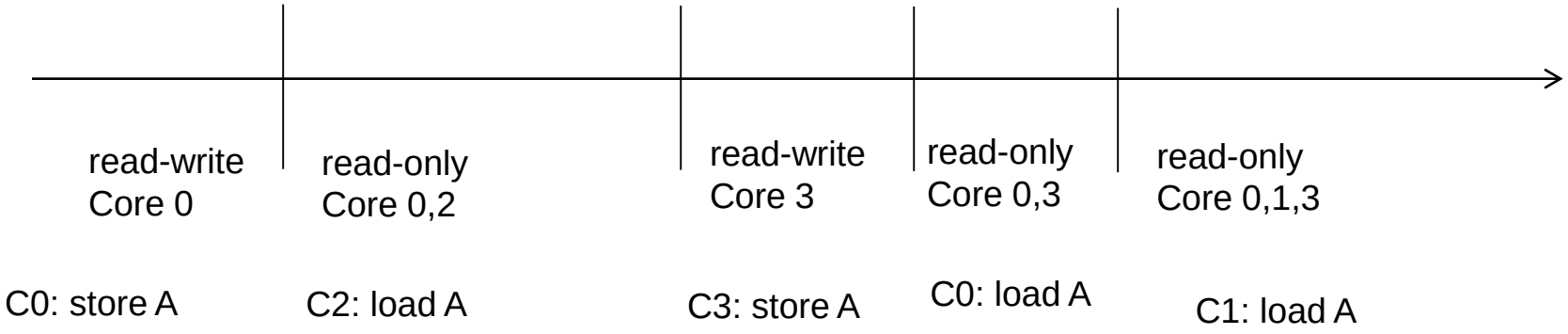
Review: Cache Coherence Problem



- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on order of which cache writes back value first
- Unacceptable situation for programmers

Coherence Invariants

1. **Single-Writer, Multiple-Reader (SWMR) Invariant**



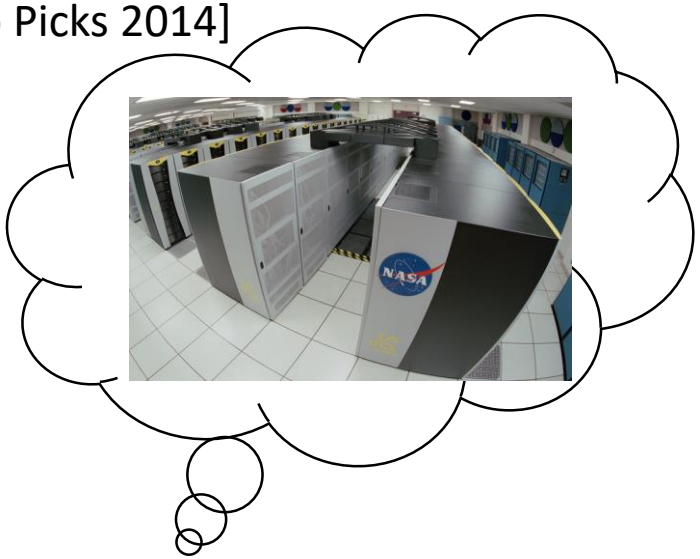
2. **Data-Value Invariant.** The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

Coherence States

- How to design system satisfying invariants?
- Track “state” of memory block copies and ensure states changes satisfy invariants.
- Typical states: “modified”, “shared”, “invalid”.
- Mechanism for updating block state called a coherence protocol.

Intra-GPU Coherence

[Singh et al., HPCA 2013, IEEE Micro Top Picks 2014]



Coherent memory space

- Efficient critical sections
- Load balancing

lock shared structure

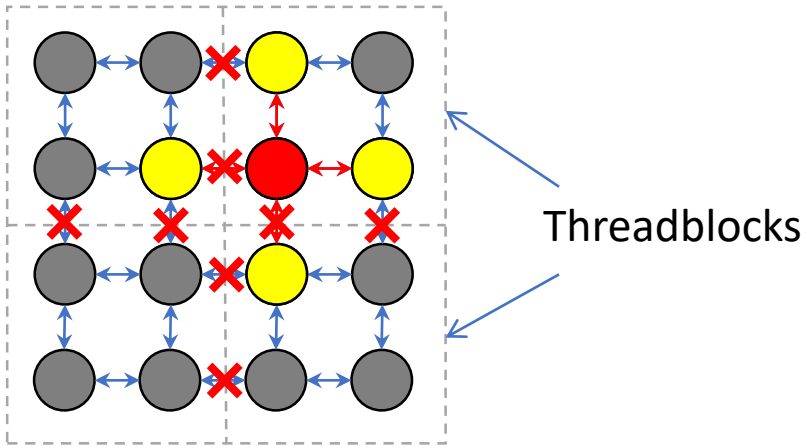
…

computation

…

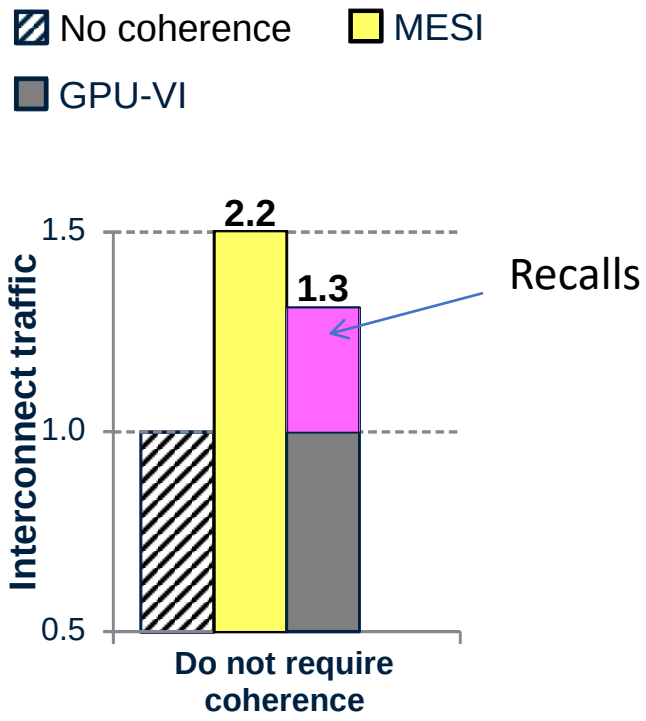
unlock

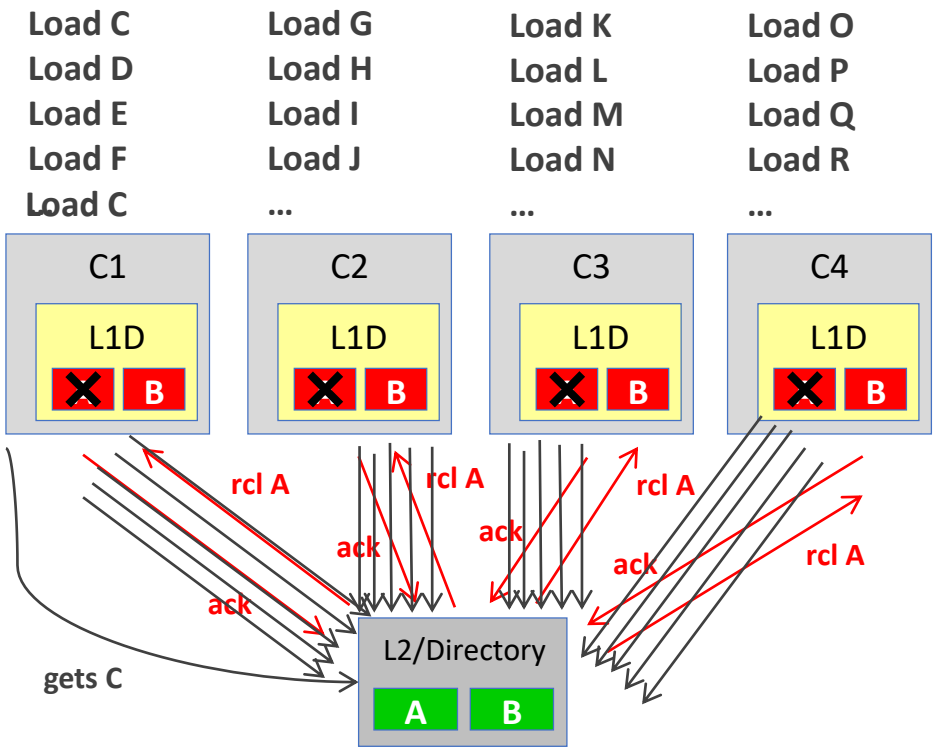
Stencil computation



GPU Coherence Challenges

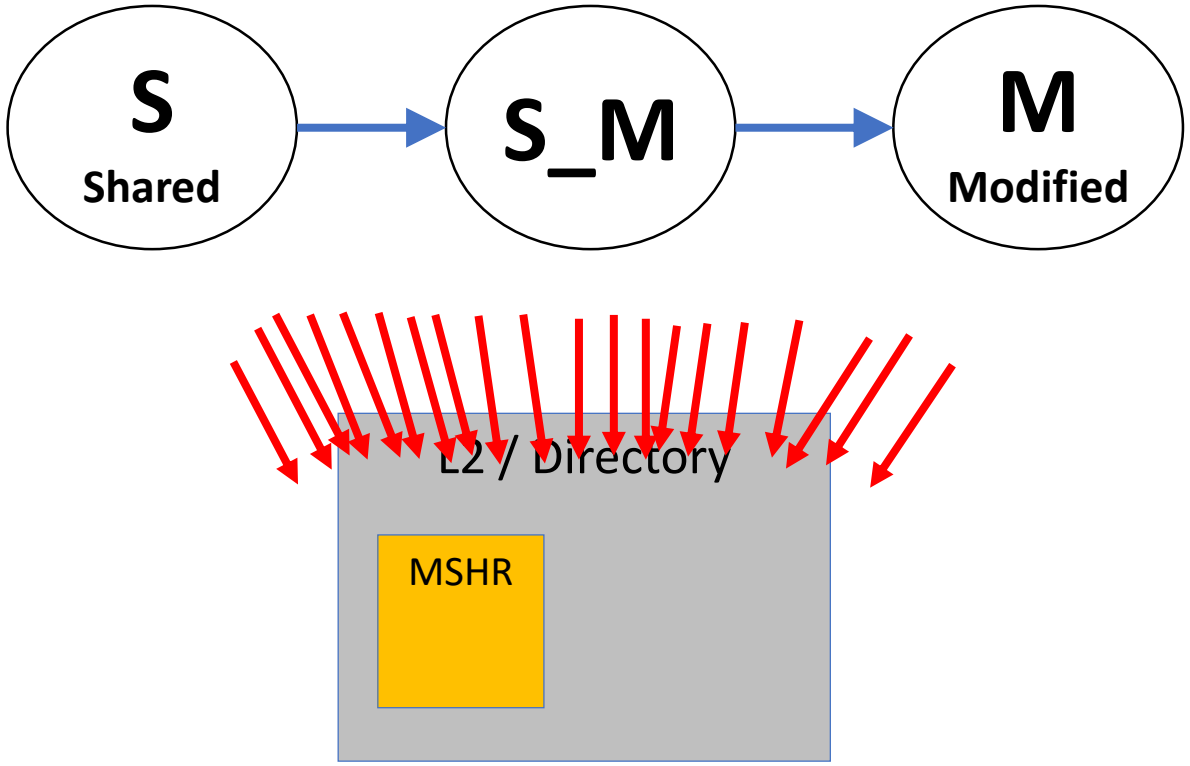
- Challenge 1: Coherence traffic





GPU Coherence Challenges

- Challenge 2: Tracking in-flight requests
  - Significant % of L2



GPU Coherence Challenges

- Challenge 3: Complexity

Non-coherent L1

	Load	L1 WThru	L1 Atomic	L1 Replacement	Data	Data Done	WBorAtomic	WBorAtomic Done
I	o i pr+ a k /I S	i pw+ ds k /I I	i pw+ ds a k /I I					
S	h k	i pw+ ds f k /I I	i pw+ ds a f k /I I	f /I				
I_S		pw+ ds f k /I I	pw+ ds a f k /I I	z		pr- u h s o /S		
I I	pr+ a k	pw+ ds k	pw+ ds a k	z	pr- h o	pr- h s o /I	pw- h o	pw- h s o /I

Events

States

Non-coherent L2

	L1 GETS	WB Data	L2 Atomic	L2 Replacement	L2 Replacement clean	Mem Data
NP	q lpB i s a j /ISS	q lpB d i x as j /IM	q lpB d i x as j /IMA			
SS	lpR ds set j	f lpW d de mr set j	f lpW d ds a mr set j	f lpE c r /NP	f lpE r /NP	
ISS	lpR s j	z	z	z	z	m e s o /SS
IM	z	z	z	z	z	m mt ee s o /SS
IMA	z	z	z	z	z	m mt e a s o /SS

MESI L2 States

	L1 GET INSTR	L1 GETS	L1 GETX	L1 UPGRADE	L1 PUTX	L1 PUTX old	L2 Replacement	L2 Replacement clean	Mem Data	WB Data	WB Data clean	Ack	Ack all	Unblock	Unblock Cancel	Exclusive Unblock
NP	q l n i s a u j /IS	q l n i s a u j /ISS	q l i x u j /IM		t j	t j										
SS	ds n u set j	ds n u set j	d fwm u set j /SS MB	fwm ts u set j /SS MB	t j	t j	i f r /S I	i f r /I I								
M	d n u set j /SS	dd u set j /MT MB	d u set j /MT MB		t j	t j	i c r s /NP	i r s /NP								
MT	b u set j /MT IIB	b u set j /MT IIB	b u set j /MT MB		l mr t j /M	t j	i f r /MT I	i f r /MCT I								
MT I	zz	zz	zz	zz	zz	zz				qq ct s o /NP	s o /NP		ct s o /NP			
MCT I	zz	zz	zz	zz	zz	zz				qq ct s o /NP	s o /NP		s o /NP			
I I	zz	zz	zz	zz	t j	t j						q o	s o /NP			
S I	zz	zz	zz	zz	t j	t j						q o	ct s o /NP			
ISS	n s u j /IS	n s u j /IS	zz		t j	t j	zz	zz	md ex s od /MT MB							
IS	n s u j	n s u j	zz		t j	t j	zz	zz	md e s od /SS							
IM	zz	zz	zz		t j	t j	zz	zz	md ee s od /MT MB							
SS MB	zz	zz	zz	zz	zz	zz	zz	zz							k /SS	mu k /MT
MT MB	zz	zz	zz	zz	zz	zz	zz	zz							k /MT	mu k /MT
M MB	zz	zz	zz	zz	t j	t j	zz	zz								mu k /MT
MT IIB	zz	zz	zz	zz	zz	zz	zz	zz		m o /MT SB	m o /MT SB			nu k /MT IB		
MT IB	zz	zz	zz	zz	t j	t j	zz	zz		m o /SS	m o /SS				k /MT	
MT SB	zz	zz	zz	zz	t j	t j	zz	zz						nu k /SS		

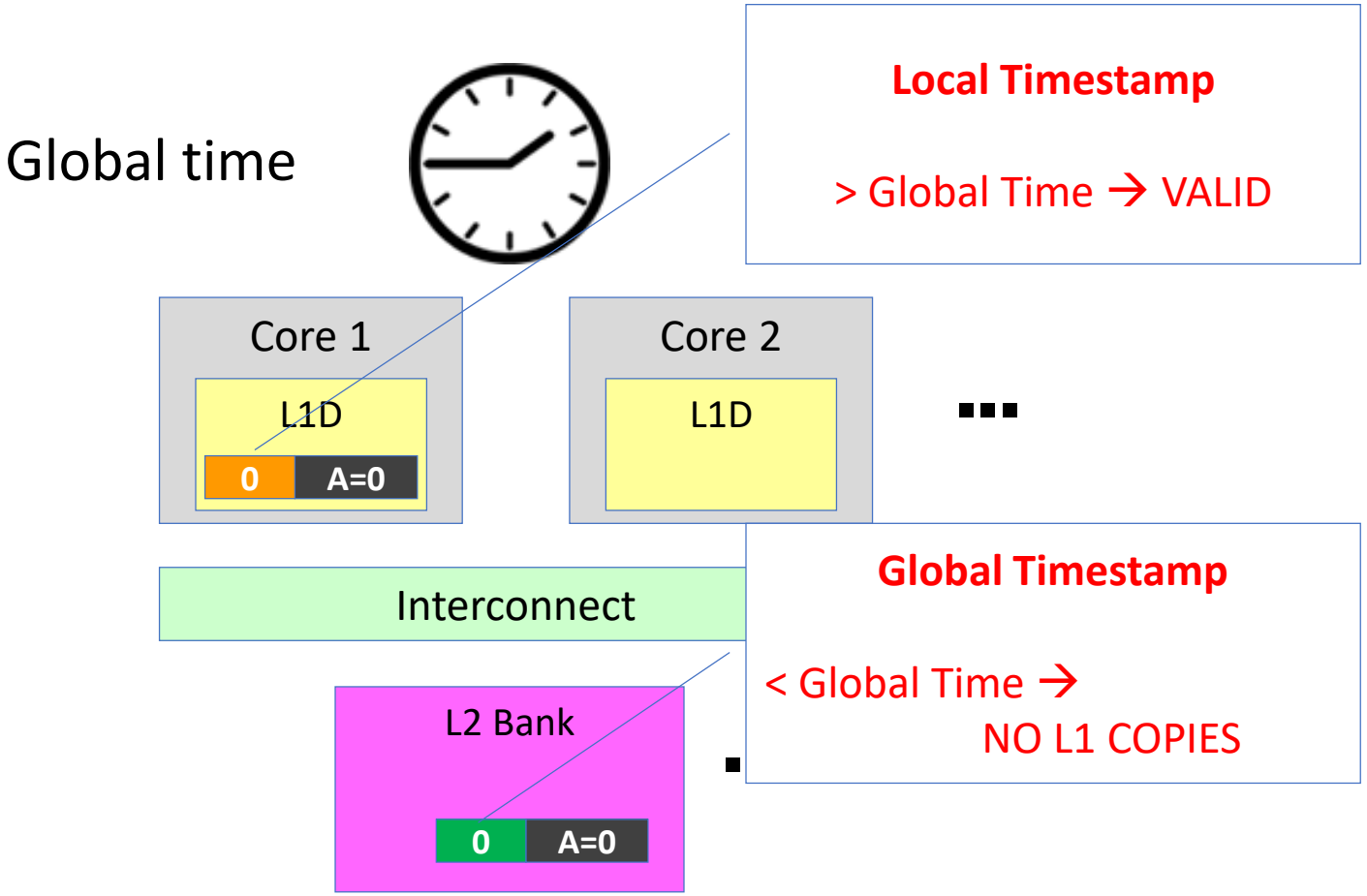
Coherence Challenges

- Challenges of introducing coherence messages on a GPU
    1. Traffic: transferring messages
    2. Storage: tracking message
    3. Complexity: managing races between messages

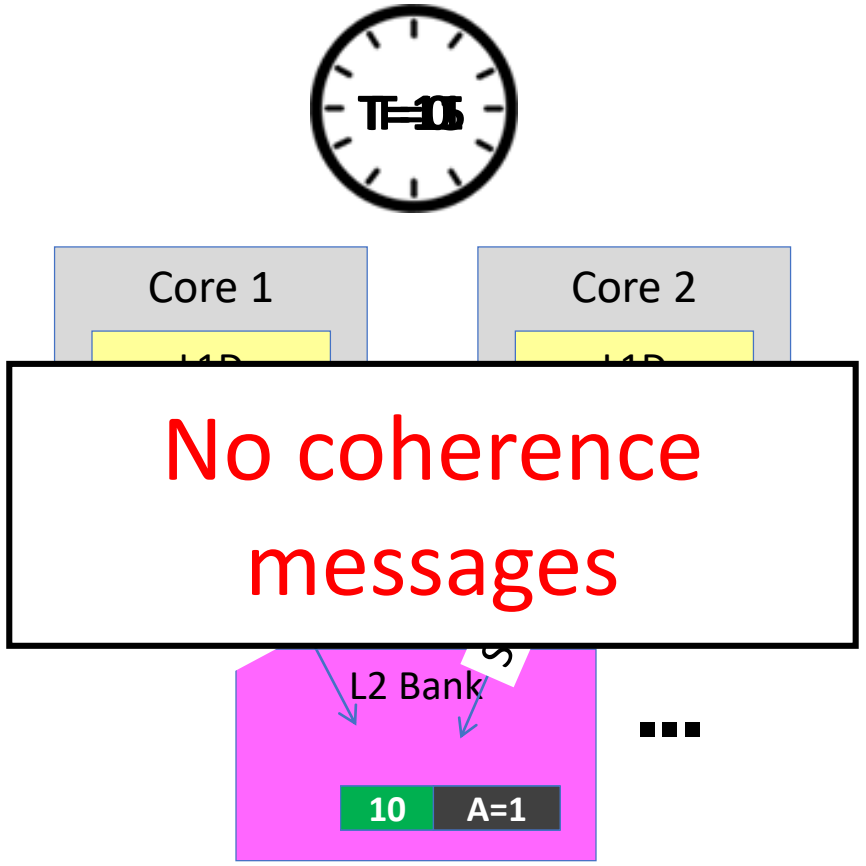
- GPU cache coherence without coherence messages?
    - YES – using global time

Temporal Coherence

Related: Library Cache Coherence

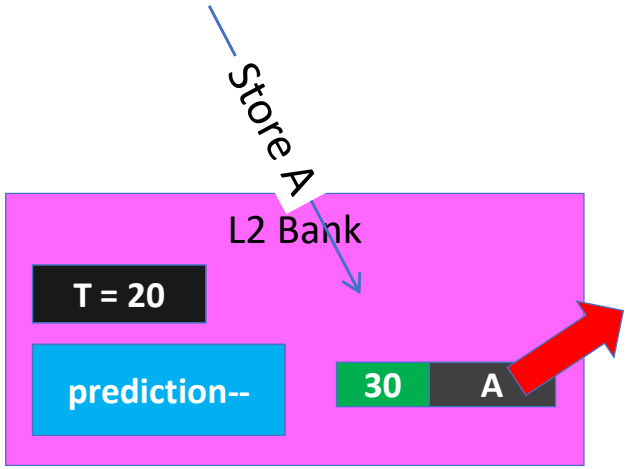


Temporal Coherence Example



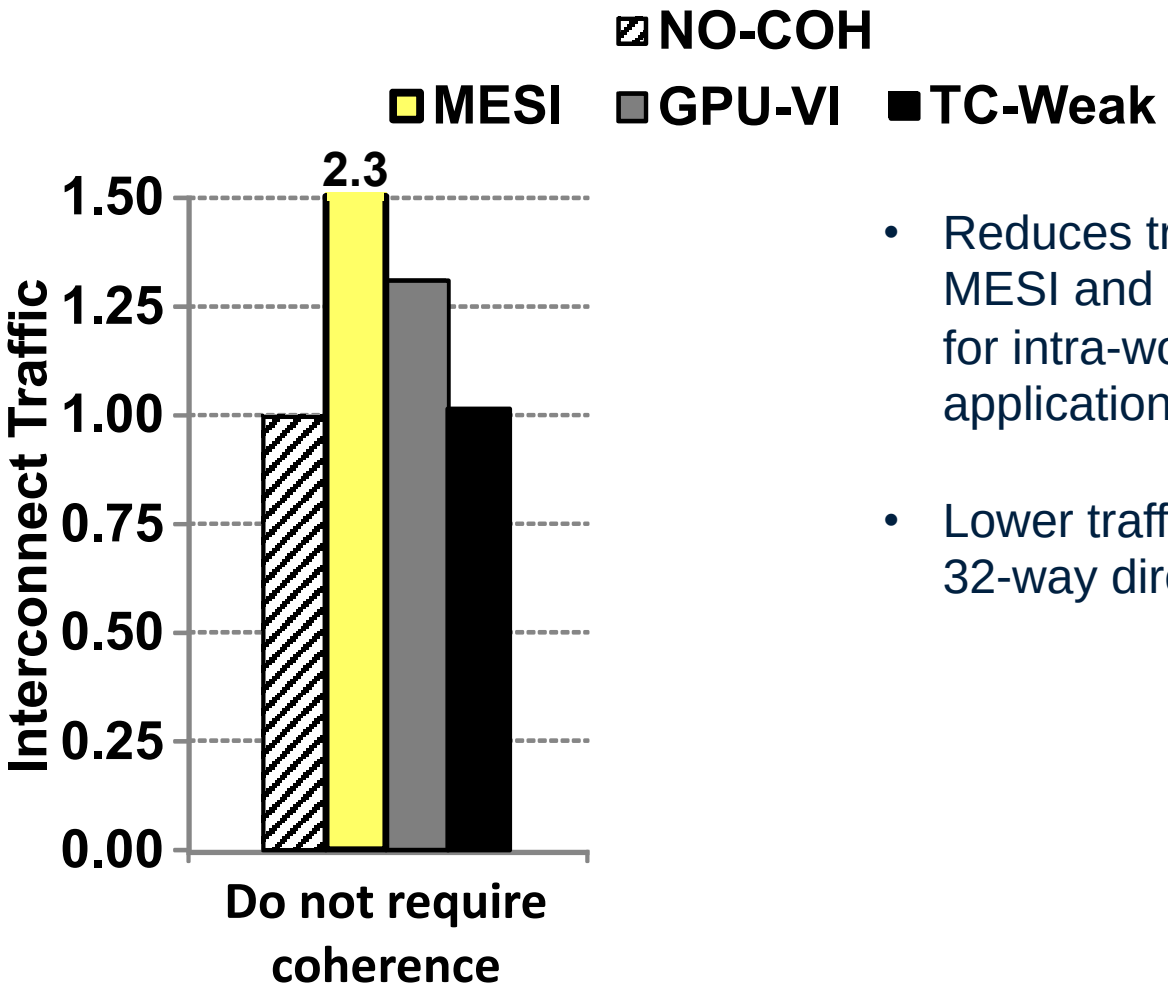
Lifetime Predictor

- One prediction value per L2 bank
- Events local to L2 bank update prediction value



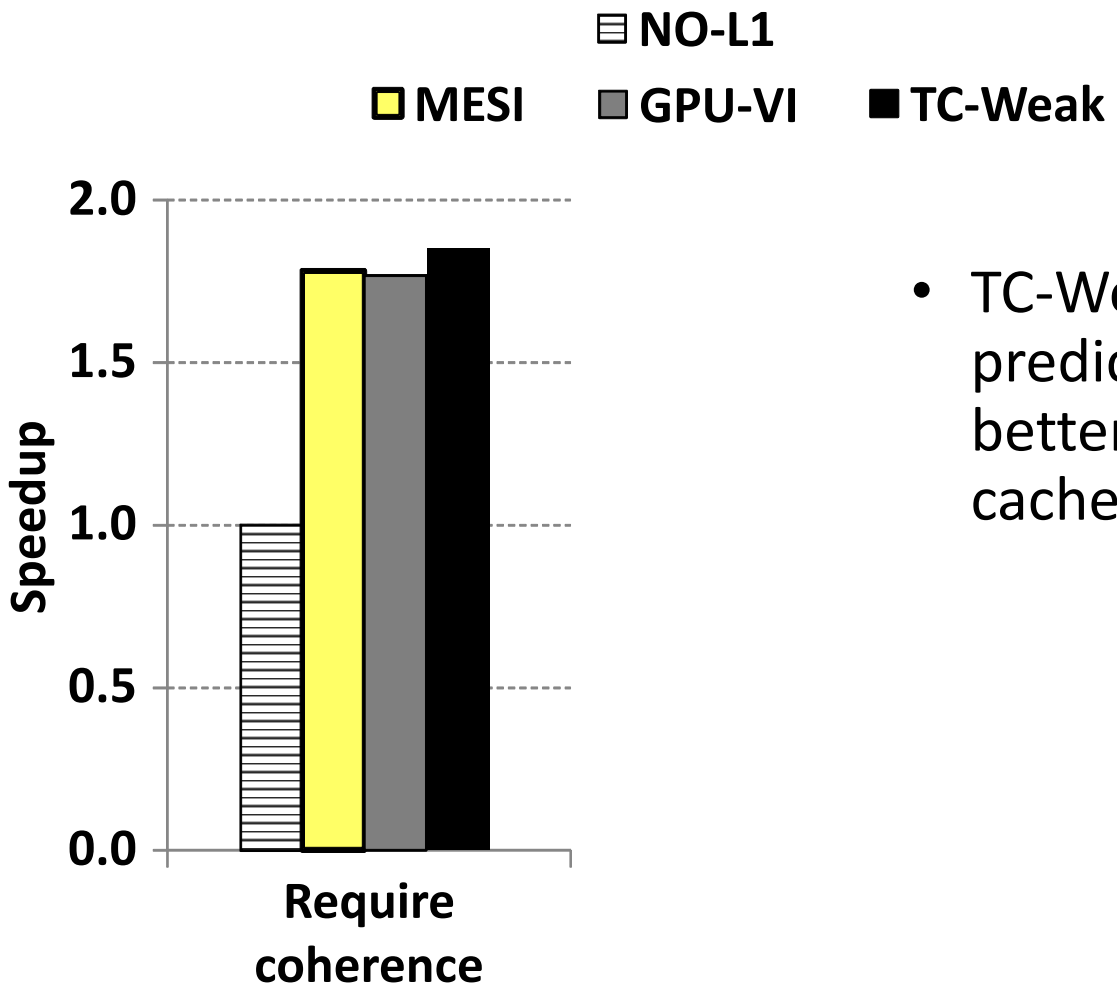
Events	Prediction
1. Expired load:	↑
2. Unexpired store:	↓
3. Unexpired eviction:	↓

Interconnect Traffic



- Reduces traffic by 53% over MESI and 23% over GPU-VI for intra-workgroup applications
- Lower traffic than 16x-sized 32-way directory

Performance



- TC-Weak with simple predictor performs 85% better than disabling L1 caches

CPU-GPU Coherence?

- Many vendors have introduced chips with both CPU and GPU (e.g., AMD Fusion, Intel Core i7, NVIDIA Tegra, etc…)
- What are the challenges with maintaining coherence across CPU and GPU?
- One important one: GPU has higher cache miss rate than CPU. Can place pressure on directory impacting performance.
- Power et al., *Heterogeneous System Coherence for Integrated CPU-GPU Systems*, MICRO 2013: Use "region coherence" to reduce number of GPU requests that need to access directory.

Synchronization

- Locks are not encouraged in current GPGPU programming manuals.
- Interaction with SIMT stack can easily cause deadlocks:

```
  while( atomicCAS(&lock[a[tid]],0,1) != 0 )
    ;  // deadlock here if a[i] = a[j] for any i,j = tid in
warp

  // critical section goes here

  atomicExch (&lock[a[tid]], 0) ;
```

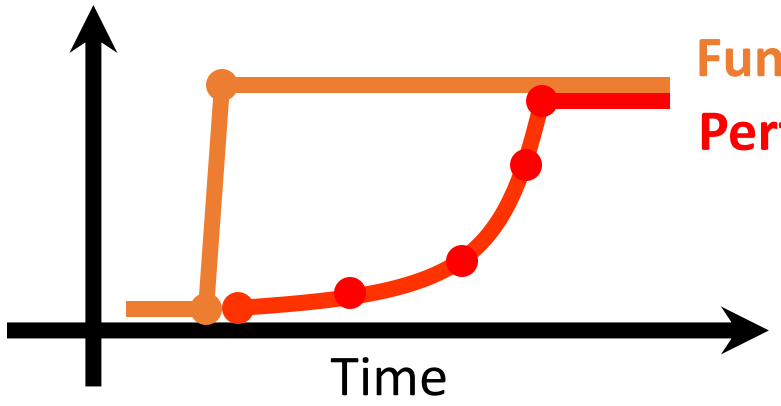
Correct way to write critical section for GPGPU:

```
done = false;
while( !done ) {
  if( atomicCAS (&lock[a[tid]], 0 , 1 )==0 ) {

    // critical section goes here

    atomicExch(&lock[a[tid]], 0 ) ;
  }
}
```

Most current GPGPU programs use barriers within thread blocks and/or lock-free data structures.

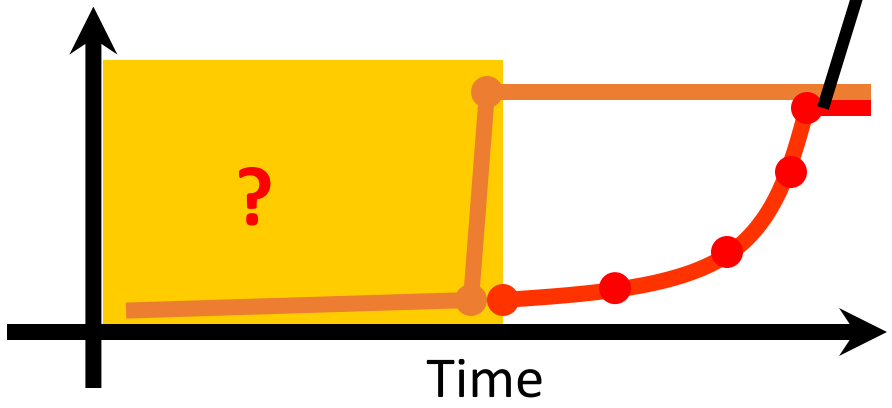
This leads to the following picture…

- Lifetime of GPU Application Development

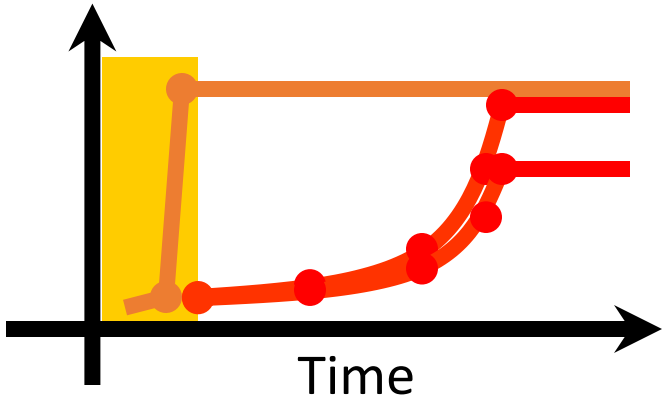


E.g. N-Body with 5M bodies
CUDA SDK: $O(n^2)$ – 1640 s (barrier)
Barnes Hut: O(nLogn) – 5.2 s (locks)

Fine-Grained Locking/Lock-Free



Transactional Memory



Wilson Fung, Inderpeet Singh, Andrew Brownsword, Tor Aamodt