



MEMORY CONSISTENCY MODEL

Girish Bharambe

AGENDA

- ❖ Motivation
- ❖ What is Memory Consistency Model
- ❖ HW Memory Consistency Models
- ❖ C++ 11 atomics
- ❖ NVIDIA GPU Memory Consistency Model

CREDITS

Content influenced with many references

- ❖ A Primer on Memory Consistency and Cache Coherence - Daniel Sorin et al.
- ❖ Shared Memory Consistency Models: A Tutorial. WRL Research Report, 1995 - Sarita Adve
- ❖ Talks in CppCon - Hans Boehm, Michael Wong, Fedor Pikus, Olivier Giroux
- ❖ CMU Course Parallel Computer Architecture and Programming

SINGLE THREADED PROGRAMS

- ❖ Execute one statement after another
- ❖ One memory access after another
- ❖ A read after a write returns last written value

MULTI-THREADED PROGRAMS


- ❖ Statements executed by threads are interleaved
- ❖ Exponential number of possible interleaving
- ❖ Memory accesses are interleaved
- ❖ What will a read return in multi-threaded program?
 - ❖ Intuitively : A read should return latest value written to it (similar to single threaded execution)
 - ❖ What does “latest” mean?
 - ❖ Within a thread?
 - ❖ Across threads?
 - ❖ Most recent value based on actual time?

EXAMPLE

<code>data = 0; data_ready = 0;</code>	
Thread T1 (on P1)	Thread T2 (on P2)
<code>data = 5;</code> <code>data_ready = true;</code>	<code>...</code> <code>while (!data_ready) ;</code> <code>y = data;</code>
Is <code>y == 5</code> always?	

EXAMPLE

Compiler Optimizations

data = 0; data_ready = 0;	
Thread T1 (on P1)	Thread T2 (on P2)
 data = 5; data_ready = true;	... while (!data_ready) ; y = data;
y == 0	

EXAMPLE

Let's disable Compiler Optimizations

data = 0; data_ready = 0;	
Thread T1 (on P1)	Thread T2 (on P2)
data = 5; data_ready = true;	... while (!data_ready) ; y = data;
Can y == 0?	

❖ Hardware out of order execution

PROGRAMMER EXPECTATIONS

- ❖ Programmer expects processor *atomically* performs one instruction at a time, *in program order*
- ❖ Reality:
 - ❖ If processor actually operated this way, it will be very slow
 - ❖ Language Implementation (compilers, processors) aggressively reorder instructions for performance
- ❖ Bottom Line:
 - ❖ Within a single thread, compilers/processors will preserve program order illusion
 - ❖ From perspective of other threads, all bets are off

WHY REORDERING

- ❖ Fundamental Issue: Memory loads and stores are expensive
- ❖ Hiding Latency of memory operations is important for performance
- ❖ How to handle?
 - ❖ Idea: Overlap memory accesses with other accesses and computation



❖ Possible Reordering

- ❖ W -> W
- ❖ R -> R
- ❖ R -> W
- ❖ W-> R

WHAT IS A MEMORY CONSISTENCY MODEL?

- ❖ Determines order in which memory operations appear to execute
 - ❖ Specifies the values that can be returned by loads

TERMINOLOGY IN MEMORY MODEL

❖ (Single Copy) Atomicity

- ❖ Operations are indivisible
- ❖ To other threads, operation appears complete or not performed at all

❖ Visibility

- ❖ When does a memory write become visible to others i.e. Once a memory write becomes visible if a processor reads value, it will see that update.

❖ Ordering

- ❖ Ordering of memory operations
- ❖ Program Order: Order of memory operations in a single thread as written in program

WHY DO WE NEED MEMORY MODEL?

- ❖ For the same reason we need any other technical specification
- ❖ It is one specific part of contract between the software and the implementation about legal behaviors
- ❖ It affects
 - ❖ Programmability
 - ❖ Performance
 - ❖ Portability

The background is a dark blue gradient with a complex network of thin, glowing green lines. These lines connect various points, some of which are highlighted as bright green dots. The overall effect is a sense of a dynamic, interconnected system, possibly representing a network or a data structure.

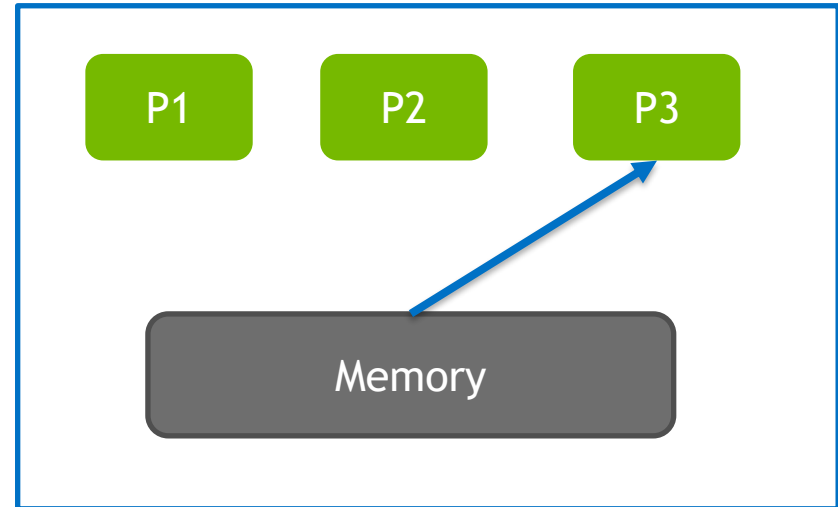
HARDWARE MEMORY MODELS

UNIPROCESSOR MEMORY MODEL

- ❖ Memory operations are *atomic* and occur *in program order*
 - ❖ Only maintain data and control dependencies
 - ❖ HW and Compiler optimizations must respect these
 - ❖ Can **reorder accesses to independent locations**
- ❖ Read from memory returns value from last write in program order
- ❖ Easy to program and to Optimize

SEQUENTIAL CONSISTENCY (SC)

- ❖ Formalized by Lamport
- ❖ *“The result of any execution is the same as if reads and writes occurred in some order and the operations of each individual processor appear in the order specified by its program”*
- ❖ Accesses of each processor in program order
- ❖ All accesses appear in sequential order
- ❖ Preserves all four ordering $W \rightarrow W$, $R \rightarrow R$, $R \rightarrow W$, $W \rightarrow R$



EXAMPLE (REVISITED)

data = 0; data_ready = 0; x = 0; y = 0;	
Thread T1	Thread T2
data = 1; data_ready = 1;	... x = data_ready; y = data;

- ❖ Under Sequential consistency
 - ❖ Possible outcomes for (x, y)
 - ❖ (0, 0), (0, 1), (1, 1)
 - ❖ Forbidden outcome
 - ❖ (1, 0)

ISSUES WITH SEQUENTIAL CONSISTENCY

- ❖ Performance

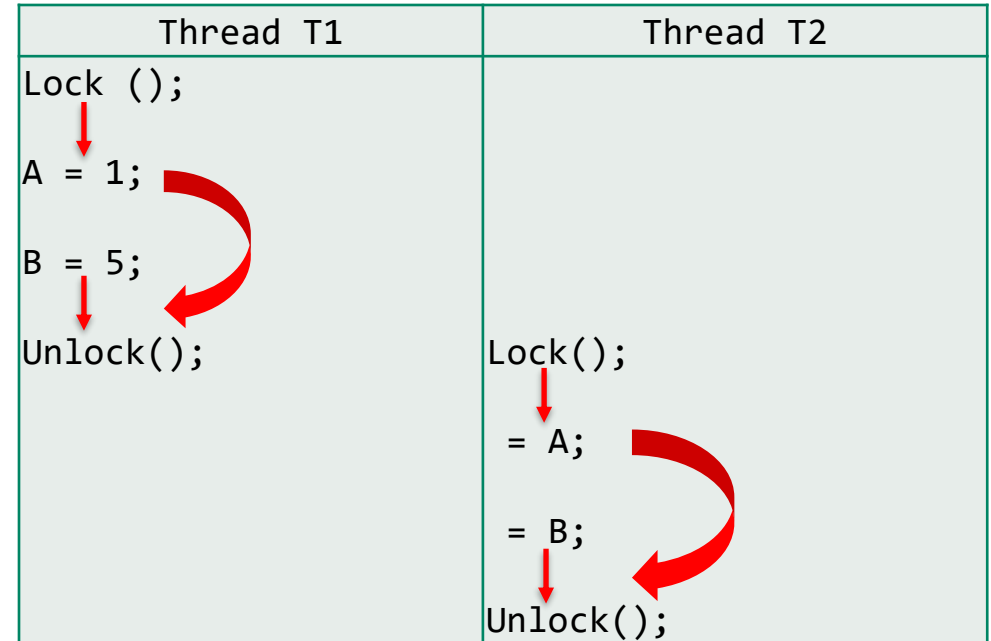
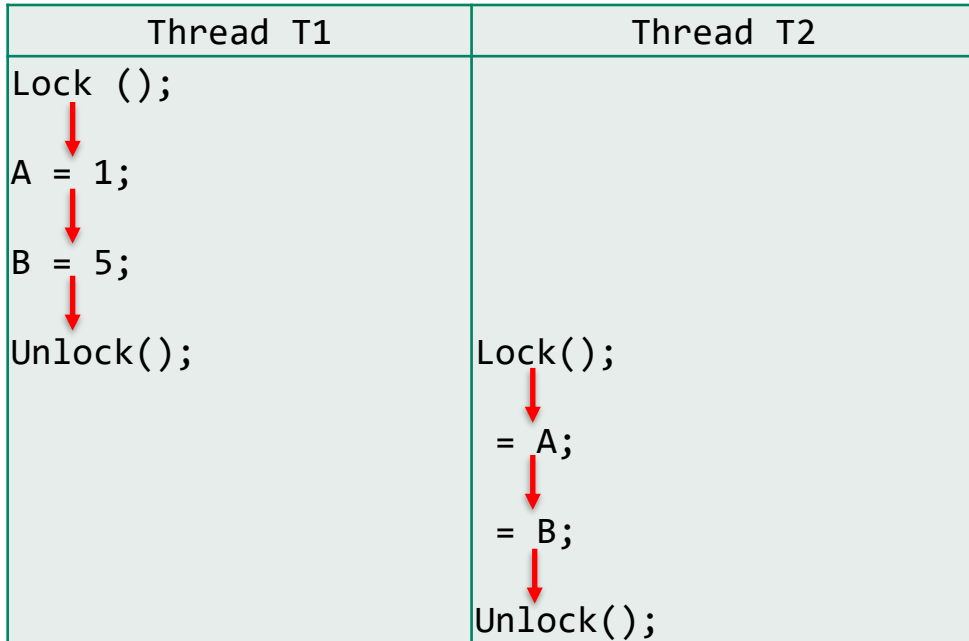
- ❖ Lot of optimizations being done by compiler and HW will be illegal

- ❖ Harder to implement:

- ❖ Requires cache coherence such that writes to same location are observed in same order by all processors
 - ❖ For each processor, delay start of memory access until previous one completes

RELAXING SC

- ❖ Relax constraints on memory ordering
 - ❖ E.g. relax any or some of orderings from $W \rightarrow W$, $R \rightarrow R$, $R \rightarrow W$, $W \rightarrow R$
- ❖ Will programs be still useful with such relaxed orderings?
- ❖ Do programs require strict ordering **all of the time**?



RELAX W -> R

TSO and PC

- ❖ Allowing reads to move ahead of writes
- ❖ Allows processor to hide latency of writes

ALTERNATIVES TO SC : TSO AND PC

❖ Total Store Order (TSO)

- ❖ Processor P can read B before its write to A is visible to other processors i.e. processor can move its read ahead of its writes
- ❖ Reads by other processors cannot return new value of “A” until write to “A” is visible to all processors
- ❖ x86 supports this

❖ Processor Consistency (PC)

- ❖ Any processor can read new value of “A” before write to “A” is visible to all processors

TSO AND PC EXAMPLES

data = 0; data_ready = 0;	
Thread T1 (on P1)	Thread T2 (on P2)
data = 5; data_ready = true;	while (!data_ready); print data;

- ❖ On TSO, will execution results match execution of SC? **Yes**
- ❖ On PC, will execution results match execution of SC? **Yes**

TSO AND PC EXAMPLES

A = 0; B = 0;		
Thread T1 (on P1)	Thread T2 (on P2)	Thread T3 (on P3)
A = 1;	while (A == 0); B = 1;	while (B == 0); print A;

- ❖ On TSO, will execution results match execution of SC? **Yes**
- ❖ On PC, will execution results match execution of SC? **No**

TSO AND PC EXAMPLES

A = 0; B = 0;	
Thread T1 (on P1)	Thread T2 (on P2)
A = 1; print B;	B = 1; print A;

- ❖ On TSO, will execution results match execution of SC? **No**
- ❖ On PC, will execution results match execution of SC? **No**

DEKKER'S ALGORITHM

Why do I care for W -> R Ordering?

A = 0; B = 0;	
Thread T1 (on P1)	Thread T2 (on P2)
<pre>A = 1; while (B == 1) ; // critical section A = 0;</pre>	<pre>B = 1; while (A == 1) ; // critical section B = 0;</pre>

RELAXED MEMORY CONSISTENCY MODELS

Weak Ordering (WO)

- ❖ Allow all reordering
- ❖ Distinguishes between data and synchronization operations
- ❖ A synchronization operation is not issued until all previous operations complete
- ❖ No operation issued until previous synchronization operation completes

Reorderable reads and writes here
memory fence
Reorderable reads and writes here
memory fence
Reorderable reads and writes here

RELAXED MEMORY CONSISTENCY MODELS

Release Ordering (RC)

- ❖ Allow all reordering
- ❖ Distinguishes between acquire and release operations
- ❖ RCsc - maintains SC between synchronization operations
- ❖ Acquire
 - ❖ Prevent reordering of any subsequent read/write
 - ❖ E.g. taking lock has acquire semantics
- ❖ Release
 - ❖ Prevent reordering of any prior read/write
 - ❖ E.g. releasing lock has release semantics

Read/writes that can move below
acquire

Acquire

Read/writes that **CANNOT** move above
acquire

Read/writes that **CANNOT** move below
release

Release

Read/writes that can move above
release

COHERENCE V/S CONSISTENCY

- ❖ Memory Coherence

- ❖ Requirements for observed behavior of read and writes to the *same* memory location

- ❖ Memory Consistency

- ❖ Requirements for observed behavior of read and writes to *different* memory location (as observed by other processors)

TRADE OFFS IN DESIGNING MEMORY MODEL

- Implementation cost for HW
- Performance expectations
- Easy of use for Programmers

An abstract network diagram with several glowing green nodes connected by thin, intersecting green lines. The background is dark blue/black with some faint, larger glowing blue-green circular shapes. The overall aesthetic is high-tech and digital.

C++11 MEMORY MODEL

C++11 MEMORY MODEL

❖ C++03: What's a thread???

int x = 0;	
Thread T1	Thread T2
++x;	++x;
Is x==2 always?	

❖ C++11: std::atomic

std::atomic<int> x(0);	
Thread T1	Thread T2
++x;	++x;
Is x==2 always? YES	

WHAT TYPES CAN BE MADE ATOMIC?

- ❖ Any trivially copyable type can be made atomic
 - ❖ Contiguous chunk of memory
 - ❖ Can be copied via memcpy
 - ❖ No virtual functions, noexcept constructor
- ❖ What all operations are supported:
 - ❖ Assignment
 - ❖ Operations based on type : increment, decrement etc.
 - ❖ Note ++x and x = x+1 are different for atomics
 - ❖ Special operations : compare_exchange etc.

C++11 MEMORY MODEL

- ❖ `atomic<T>` also has memory ordering semantics for operations before and after atomic operations
 - ❖ By default, sequential consistency
- ❖ Note C++ `volatile` does not establish inter-thread synchronization, not atomic
 - ❖ C++ `volatile` indicates something from env may also change this
 - ❖ JAVA `volatile` is different and is equivalent to C++ `atomic`

<pre>atomic<int> data_ready; int data = 0;</pre>	
Thread T1 (on P1)	Thread T2 (on P2)
<pre>data = 5; data_ready.store(1);</pre>	<pre>while (data_ready.load()==0); print data;</pre>

C++11 MEMORY MODEL

- ❖ C++11 `atomic<T>` provides atomic read, write and read-modify-write of entire objects
- ❖ Provides memory ordering semantics for operations before and after atomic operations
 - ❖ By default, sequential consistency
- ❖ Note C++ `volatile` does not establish inter-thread synchronization, not atomic

<pre>atomic<int> data_ready; int data = 0;</pre>	
Thread T1 (on P1)	Thread T2 (on P2)
<pre>data = 5; data_ready.store(1, memory_order_release);</pre>	<pre>while (data_ready.load(memory_order_acquire)==0); print data;</pre>

- ❖ No fence required on x86 for above program

C++11 MEMORY ORDERS

- ❖ `memory_order_seq_cst`
 - ❖ Default
 - ❖ Sequentially consistent
- ❖ `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`
 - ❖ Relax W -> R ordering
 - ❖ Avoids fences after store, allows some fences to be weekend
- ❖ `memory_order_relaxed`
 - ❖ Relax all ordering but preserve atomicity
 - ❖ Only ensures ordering for operations on that memory location
 - ❖ Useful for single word data structures e.g. counters, accumulation in bit vector
- ❖ `memory_order_consume`
 - ❖ Out of scope for today's discussion

SEQUENCING OF OPERATIONS

- ❖ `sequenced-before`
 - ❖ Per thread ordering of operations
 - ❖ Old C++ std, use sequence points for these
- ❖ `synchronizes-with`
 - ❖ Between operations on `atomic` types
 - ❖ Store-release synchronized with load-acquire
 - ❖ Provides inter-thread ordering
- ❖ `happens-before`
 - ❖ Between memory operations in a different thread

DATA RACES

- ❖ Conflicting data accesses:

- ❖ Two memory accesses conflict if:

- ❖ They access same memory location
 - ❖ At least one is write

- ❖ Data Race:

- ❖ Conflicting data access that are not ordered by synchronization (e.g. fence, release/acquire, barrier etc.)

C++11 & DATA RACES

- ❖ C++11 is SC for DRF
 - ❖ Sequential consistency for data race free programs
 - ❖ Undefined behavior for programs with data races

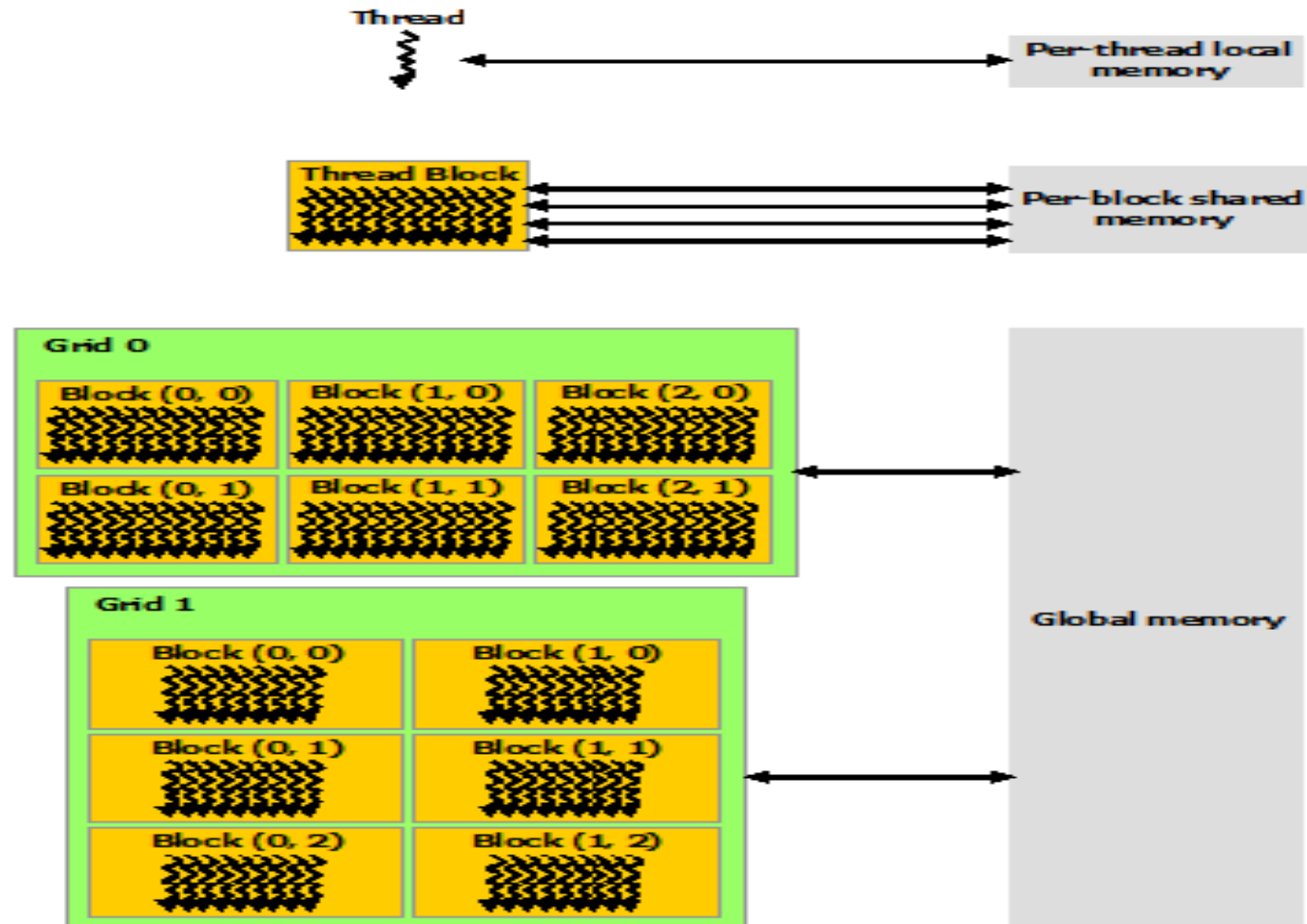


NVIDIA GPU MEMORY MODEL

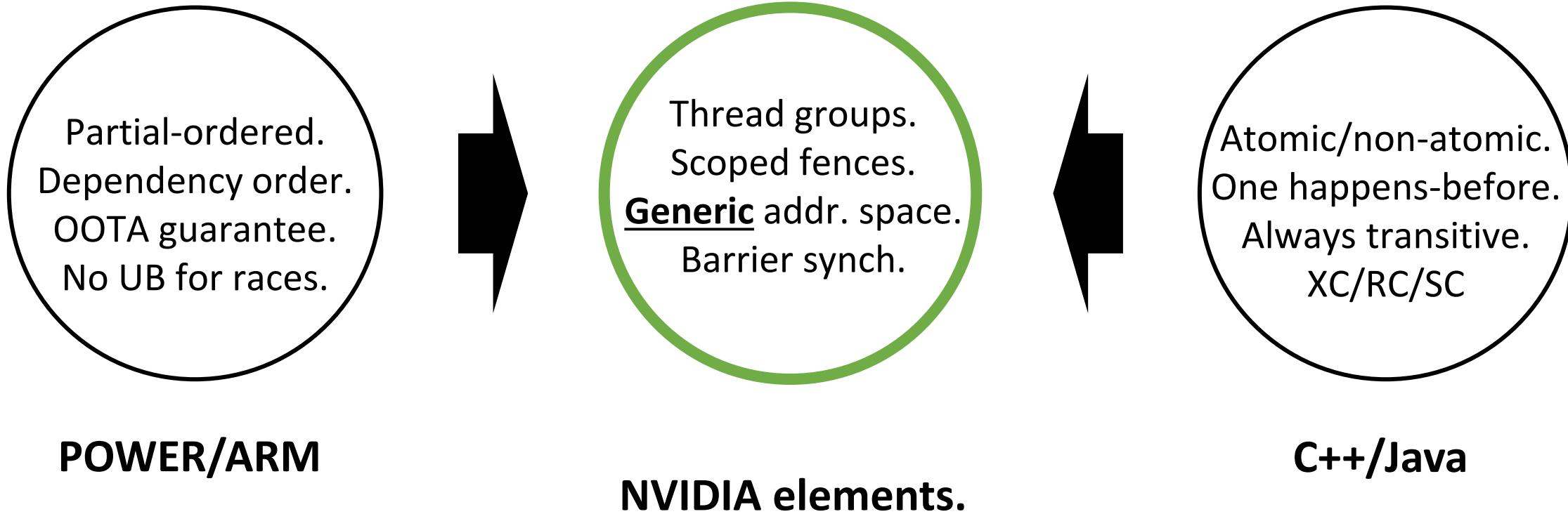
WHY DO GPUS NEED MEMORY MODEL?

- Memory model required for reasoning about and writing multi-threaded programs
- GPUs supporting closer to traditional shared memory programming model
- Enable more concurrent Algorithms and Data structures on GPUs that were previously unavailable

MEMORY HIERARCHY IN GPU



INSPIRATION FOR NVIDIA MEMORY MODEL



NVIDIA MEMORY MODEL IN NUTSHELL

- Weakly ordered
- Single copy atomicity
- Scoped fences
- Does not require data race freedom
- Described as axiomatic memory model
 - Axiomatic : Define a set of criteria (“axioms”) to be satisfied
 - Executions permitted unless they fail one or more axioms
 - Operational: define a golden abstract machine model
 - Executions forbidden unless producible when executing this model

CONCEPTS IN NVIDIA MEMORY MODEL

- Weak operations
 - Used for ordinary data
- Strong operations
 - Provide synchronization semantics
- Scopes
 - `.cta`, `.gpu`, `.sys`

CONCEPT: MORALLY STRONG

Two operations are morally strong relative to each other if

- Operations are related in program order or each operation is strong and specifies a scope that includes the other thread
- If both are memory operations, they overlap completely

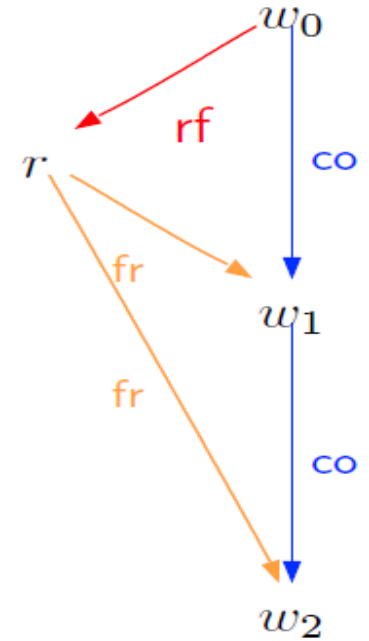
Generally, only pair of morally strong operations may be used to synchronize between the threads

SYNCHRONIZATION PATTERNS

- Release Acquire consistency
- Barrier synchronization
- Synchronization of SC fences

RELATIONS AND ORDERINGS

- **po** “program order” relates operations in sequence order within each thread.
- **rf** “read from” (W->R) from writes to reads that take their value.
 - **rfe** “read from external” when across threads
- **co** “coherence order” (W-> R) from writes to other writes that overwrote their value.
- **fr** “from read” (R-> W) *derived*, from reads to writes that overwrote the value taken.
 - **fre** “from read external” when across threads
- **sc** Partial order of all morally strong sc fences



AXIOMS IN PTX MEMORY MODEL

Causality: Message Passing (MP)

- <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#causality-axiom>

<code>.global .u32 data = 0; .global .u32 flag = 0;</code>	
T1	T2
W1: <code>st.global.u32 [data], 1;</code>	R1: <code>ld.global.relaxed.sys.u32 %r0, [flag];</code>
F1: <code>fence.sys;</code>	F2: <code>fence.sys;</code>
W2: <code>st.global.relaxed.sys.u32 [flag], 1;</code>	R2: <code>ld.global.u32 %r1, [data];</code>
IF %r0 == 1 THEN %r1 == 1	

AXIOMS IN PTX MEMORY MODEL

Causality: Store Buffering (SB)

- <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#causality-axiom>

<code>.global .u32 x = 0; .global .u32 y = 0;</code>	
T1	T2
W1: <code>st.global.u32 [x], 1;</code>	R1: <code>st.global.u32 [y], 1;</code>
F1: <code>fence.sc.sys;</code>	F2: <code>fence.sc.sys;</code>
W2: <code>ld.global.u32 %r0, [y];</code>	R2: <code>ld.global.u32 %r1, [x];</code>
<code>%r0 == 1 OR %r1 == 1</code>	

AXIOMS IN PTX MEMORY MODEL

SC Per Location: Coherent Read Read (coRR)

- <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#sc-per-loc-axiom>

.global .u32 x = 0;	
T1	T2
W1: st.global.relaxed.sys.u32 [x], 1;	R1: ld.global.relaxed.u32 %r0, [x]; R2: ld.global.relaxed.u32 %r1, [x];
IF %r0 == 1 THEN %r1 == 1	

AXIOMS IN PTX MEMORY MODEL

No Thin Air: Load Buffering (LB)

- <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#no-thin-air-axiom>
- No Thin Air: Litmus test: LB

<code>.global .u32 x = 0; .global .u32 y = 0;</code>	
T1	T2
A1: <code>ld.global.u32 %r0, [x];</code> B1: <code>st.global.u32 [y], %r0;</code>	A2: <code>ld.global.u32 %r0, [y];</code> B2: <code>st.global.u32 [x], %r0;</code>
FINAL STATE: <code>x == 0 AND y == 0</code>	

MOTIVATING EXAMPLE

Legacy Code

```
__device__ void
signal (volatile int &flag)
{
    __threadfence_system();
    atomicExch((int*)&flag, 1);
}

__device__ int
poll(volatile int &flag, int &data)
{
    while (1 != atomicAdd((int*)&flag, 0))
        ;
    __threadfence_system();
    return data;
}
```

LIBCU++

```
__device__ void
signal(atomic<bool> &flag)
{
    flag = true;
    // flag.store(true, memory_order_release);
}

__device__ int
poll(atomic<int> &flag, int &data)
{
    while (!flag)
        // OR (!flag.load(memory_order_acquire))
        ;
    return data;
}
```

DATA RACES

- Two overlapping operations conflict if one of them is a write
- Two conflicting operations are in a data race if they are not related in causality order and they're not morally strong
- PTX memory model axioms describe behavior when program has *uniform size data race*
 - Currently don't specify behavior for programs with mixed-size data race

FORMAL ANALYSIS OF PTX MEMORY MODEL

- Formal axiomatic model of PTX in Alloy DSL
- Mapping Scoped C++ RC11 model to PTX
- Formal proof of mapping Scoped C++ RC11 model to PTX using Alloy to Coq compiler
- <https://dl.acm.org/citation.cfm?id=3304043>

FUTURE WORK

- Mixed size data races
 - Still very new and not well understood problem
 - <https://www.cl.cam.ac.uk/~pes20/popl17/mixed-size.pdf>
- Specifying memory consistency model for textures, surfaces
- Scalability for large GPU systems and clusters