Efficiently Enforcing Strong Memory Ordering in GPUs

Abhayendra Singh*, Shaizeen Aga, Satish Narayanasamy

Google, University of Michigan, Ann Arbor

Dec 9, 2015

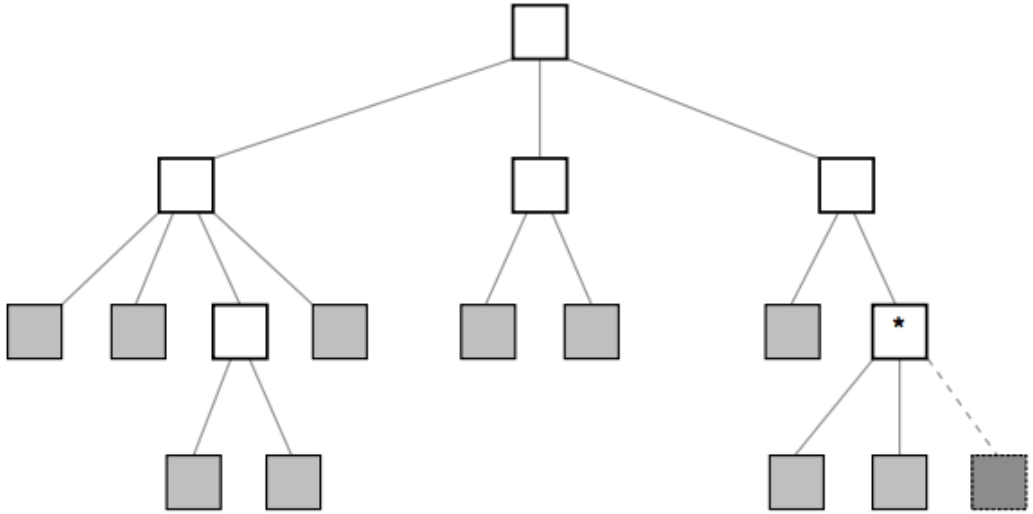
University of Michigan
Electrical Engineering and Computer Science

*author performed the work at the University of Michigan, Ann Arbor

Increasing communication between threads in GPGPU applications

More irregular applications run on GPUs

- data-dependent,
- higher communication

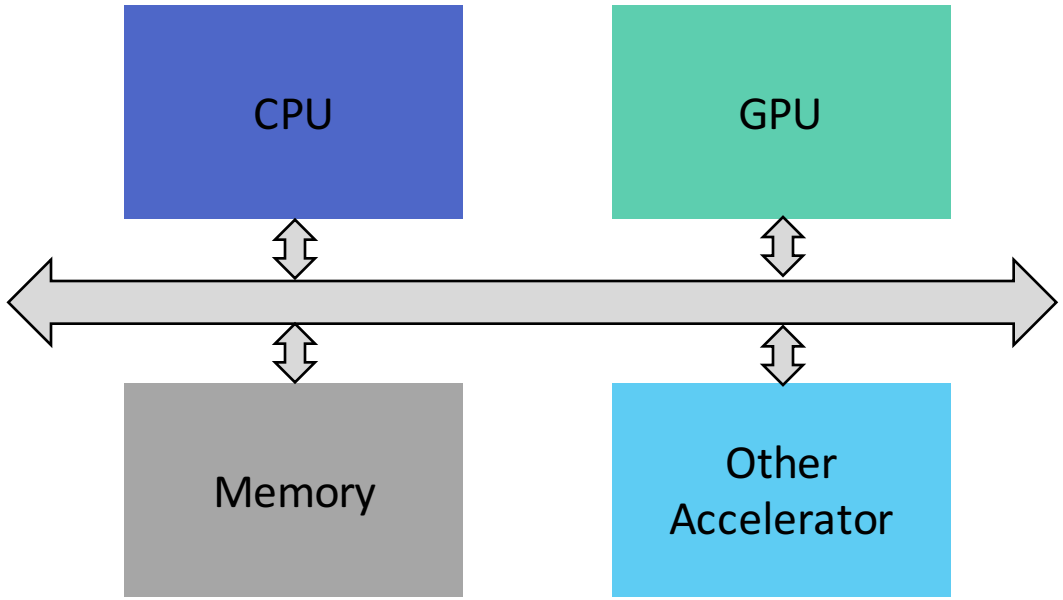


TreeBuilding kernel in barneshut
(Burtscher et al., IISWC'12)

Heterogeneous systems will have more fine-grained communication

Fine-grain communication between CPU and GPU

- Unified virtual memory
- Cache coherence [Power et al., MICRO'13]



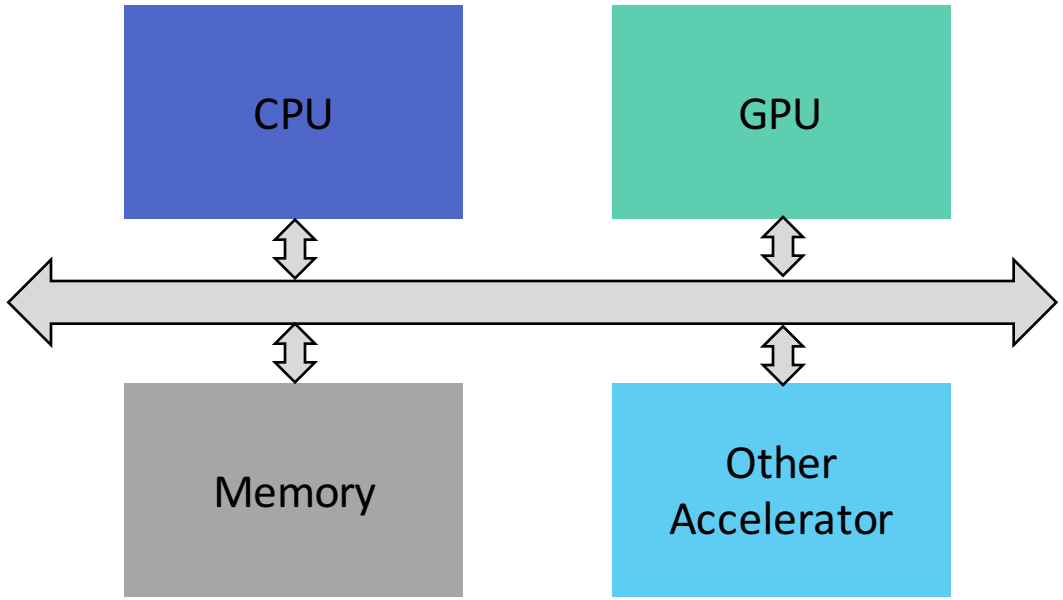
Heterogeneous systems will have more fine-grained communication

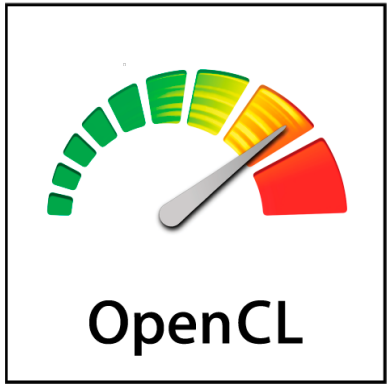
Fine-grain communication between CPU and GPU

- Unified virtual memory
- Cache coherence [Power et al., MICRO'13]

OpenCL supports fine-grain sharing

More irregularity in applications





Memory Consistency Model

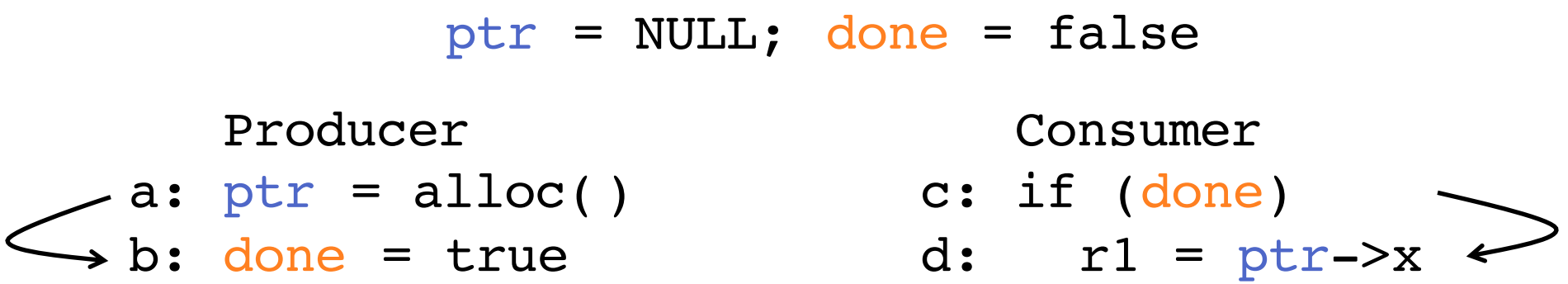
Defines rules that a programmer can use to reason about a parallel execution

Memory Consistency Model

Defines rules that a programmer can use to reason about a parallel execution

Sequential Consistency (SC)

"program-order"



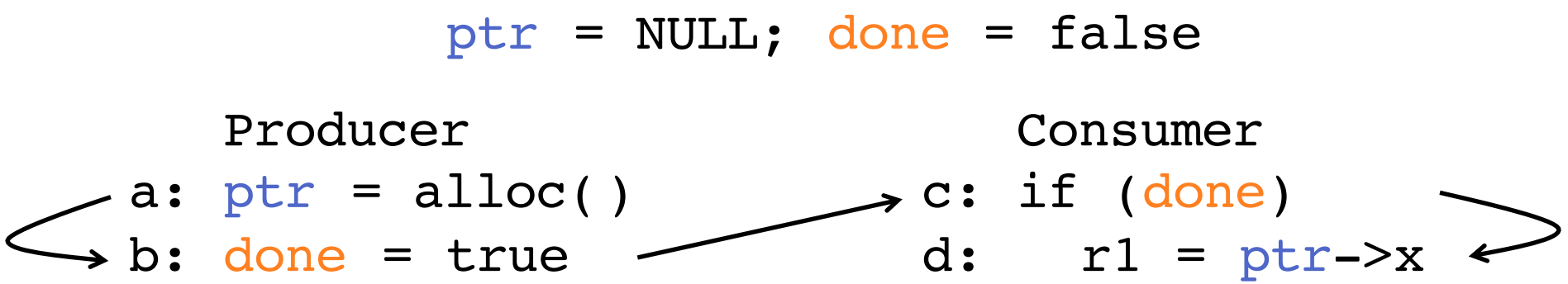
Memory Consistency Model

Defines rules that a programmer can use to reason about a parallel execution

Sequential Consistency (SC)

"program-order" +

"atomic memory"



Data-race-free-0 (DRF-0) Memory Model

C++, Java

OpenCL, CUDA

- Heterogeneous-race-free (HRF) (Hower et al., ASPLOS'14)

Data-race-free-0 (DRF-0) Memory Model

C++, Java

OpenCL, CUDA

Heterogeneous-race-free (HRF) (Hower et al., ASPLOS'14)

SC if data-race-free

Programmers annotate synchronization variables

```
ptr = NULL; atomic done = false

   Producer                    Consumer
a: ptr = alloc()          c: if (done)
b: done = true            d:   r1 = ptr->x
```

Data-race-free-0 (DRF-0) Memory Model

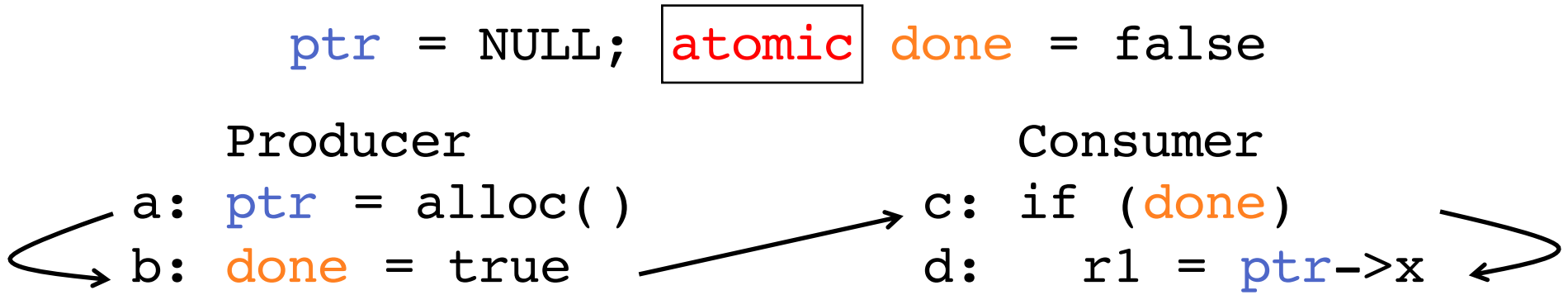
C++, Java

OpenCL, CUDA

- Heterogeneous-race-free (HRF) (Hower et al., ASPLOS'14)

SC if data-race-free

- Programmers annotate synchronization variables
- Compiler and runtime guarantee total order on synchronization operations



Data-race-free-0 (DRF-0) Memory Model

C++, Java

OpenCL, CUDA

- Heterogeneous-race-free (HRF) (Hower et al., ASPLOS'14)

SC if data-race-free

- Programmers annotate synchronization variables
- Compiler and runtime guarantee total order on synchronization operations

```
          ptr = NULL; done = false

   Producer                    Consumer
a: ptr = alloc()          c: if (done)
b: done = true            d:   r1 = ptr->x
```

reordering could lead to ptr being NULL

Data-race-free-0 (DRF-0) Memory Model

C++, Java

OpenCL, CUDA

- Heterogeneous-race-free (HRF) (Hower et al., ASPLOS'14)

SC if data-race-free

- Programmers annotate synchronization variables
- Compiler and runtime guarantee total order on synchronization operations

Undefined semantics for programs with a data-race

Documented data-races in GPGPU programs

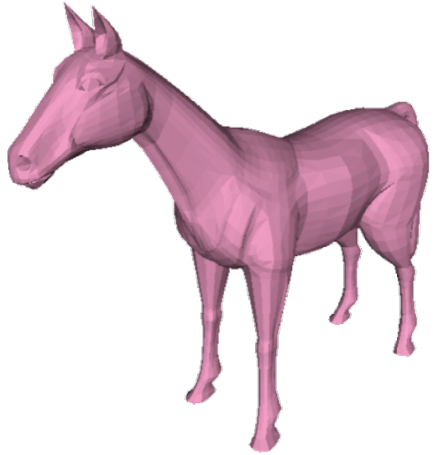


Image source: [Alglave et al., ASPLOS 2015]

Bug: a data-race in code for
dynamic load balancing
[Tyler Sorensen, MS thesis, 2014]

Other data-races:

- N-body simulation [Betts et al., OOPSLA 2012]
- RadixSort [Li et al., PPoPP 2012]
- Efficient Synchronization Primitives for GPUs [Tyler Sorensen, MS thesis, 2014]

Is there a motivation for DRF-0 over SC?

Performance of DRF-0 better than SC?

Very little for CPUs

IEEE Computer'98, PACT'02, ISCA'12

Is there a performance justification for DRF-0 (or TSO) over SC in GPUs?

Goals

Identify sources of SC violation in GPUs

Understand overhead of various memory ordering constraints in GPUs

DRF-0, TSO, SC

Bridge the gap between SC and DRF-0

Access-type aware GPU architecture

How can GPU violate SC?

Instructions are executed in-order

How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order

- Caching at L1
- Reordering in interconnect
- Partitioned address space

How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order

- Caching at L1
- Reordering in interconnect
- Partitioned address space

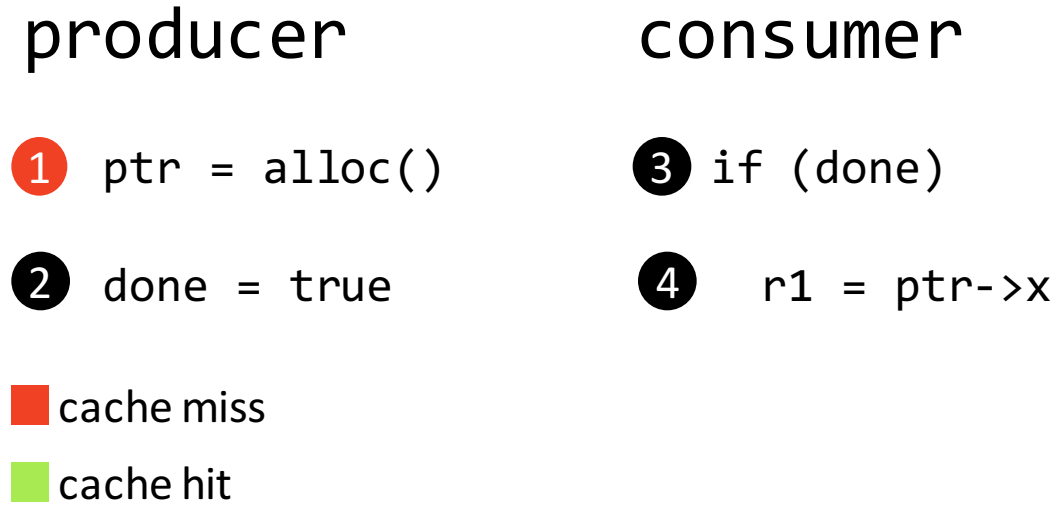
producer	consumer
❶ `ptr = alloc()`	❸ `if (done)`
❷ `done = true`	❹ `r1 = ptr->x`

How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order

- Caching at L1
- Reordering in interconnect
- Partitioned address space

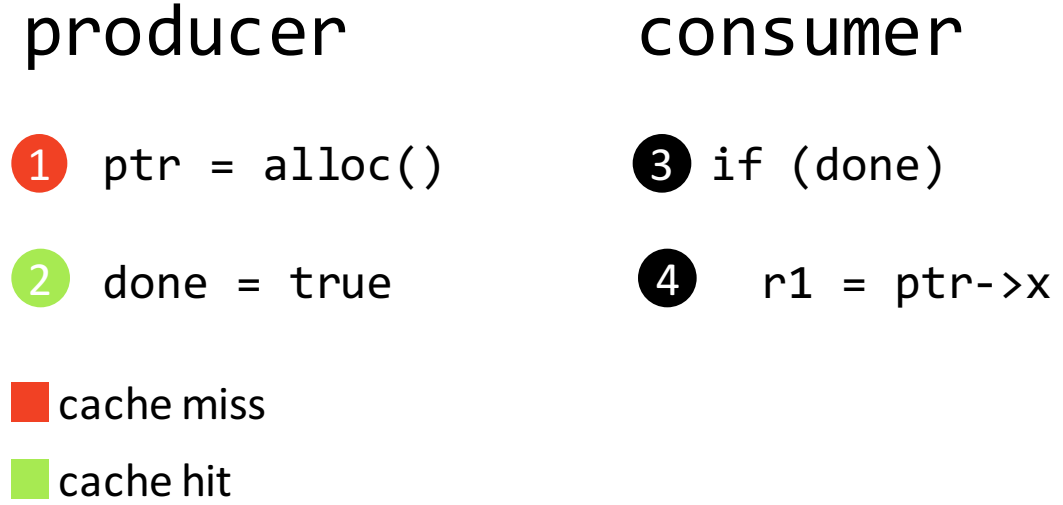


How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order

- Caching at L1
- Reordering in interconnect
- Partitioned address space

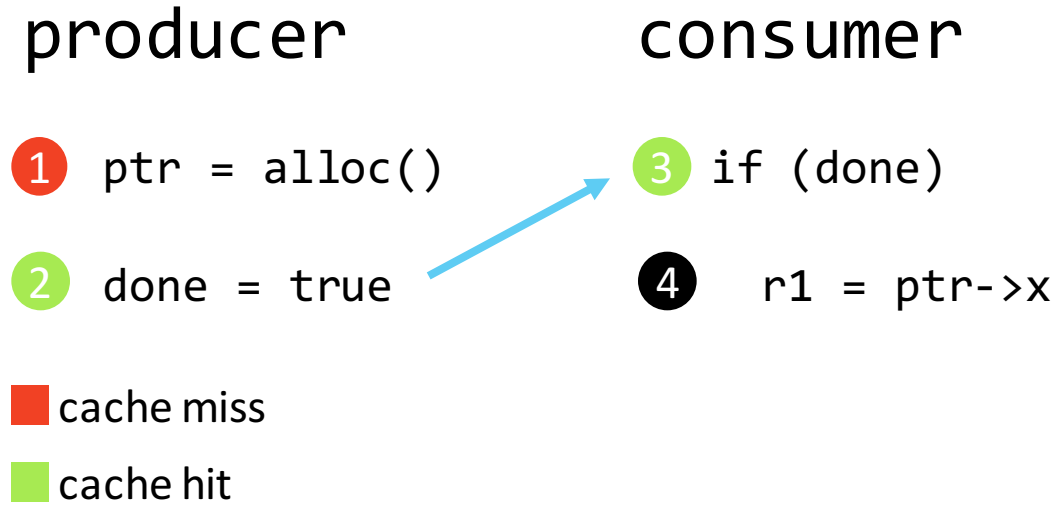


How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order

- Caching at L1
- Reordering in interconnect
- Partitioned address space

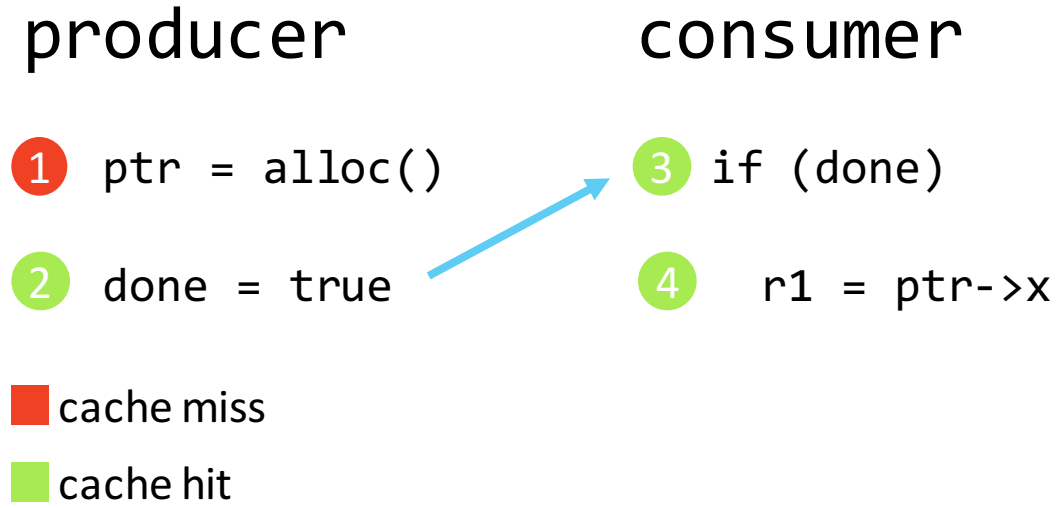


How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order

- Caching at L1
- Reordering in interconnect
- Partitioned address space



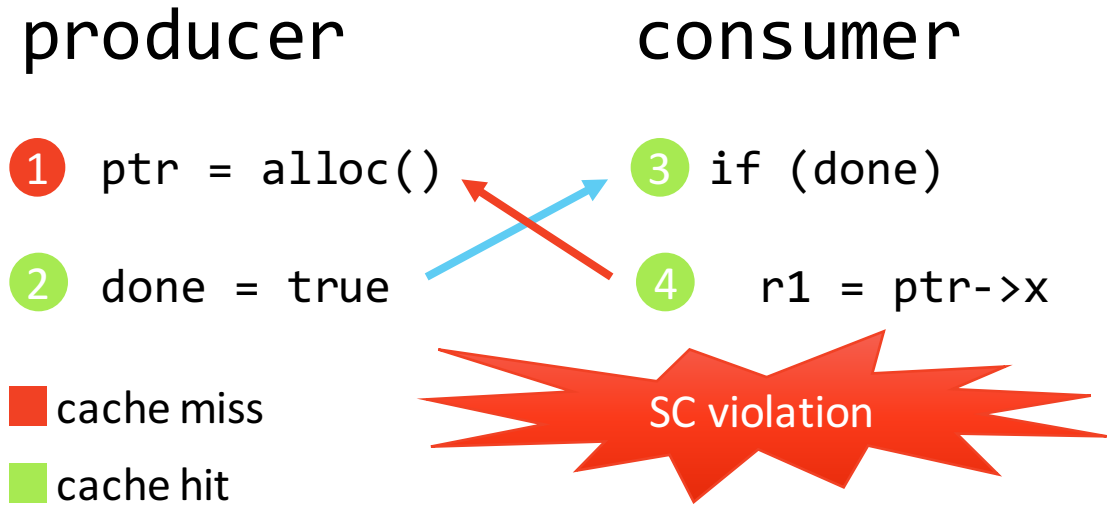
How can GPU violate SC?

Instructions are executed in-order

But, can complete out-of-order

- Caching at L1
- Reordering in interconnect
- Partitioned address space

⟹ Can violate SC



Roadmap

Identify sources of SC violation

Understand overhead of various memory ordering constraints in GPUs

- DRF-0, TSO, SC

Bridge the gap between SC and DRF-0

- Access-type aware GPU architecture

Fences for various memory models

DRF-0

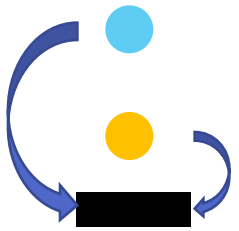
fences only for synchronization

SC

any shared or global access behaves like a fence

Naïvely Enforcing Fence Constraints

Delay a warp till non-local memory accesses preceding a fence are complete



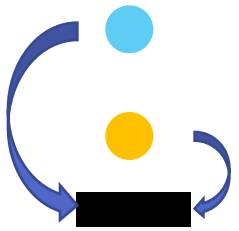
Shared memory access

Global memory access

Fence

Naïvely Enforcing Fence Constraints

Delay a warp till non-local memory accesses preceding a fence are complete



Shared memory access

Global memory access

Fence

GPU extension:

- Two counters per warp track its pending global loads and stores
- No need to track pending shared memory accesses

warp id	pending loads	pending stores
w0	0	1
…	…	…
…	…	…

Experimental Methodology

Simulator: GPGPU-sim v3.2.1

- extended with Ruby memory hierarchy
- 16 SMs, crossbar interconnect

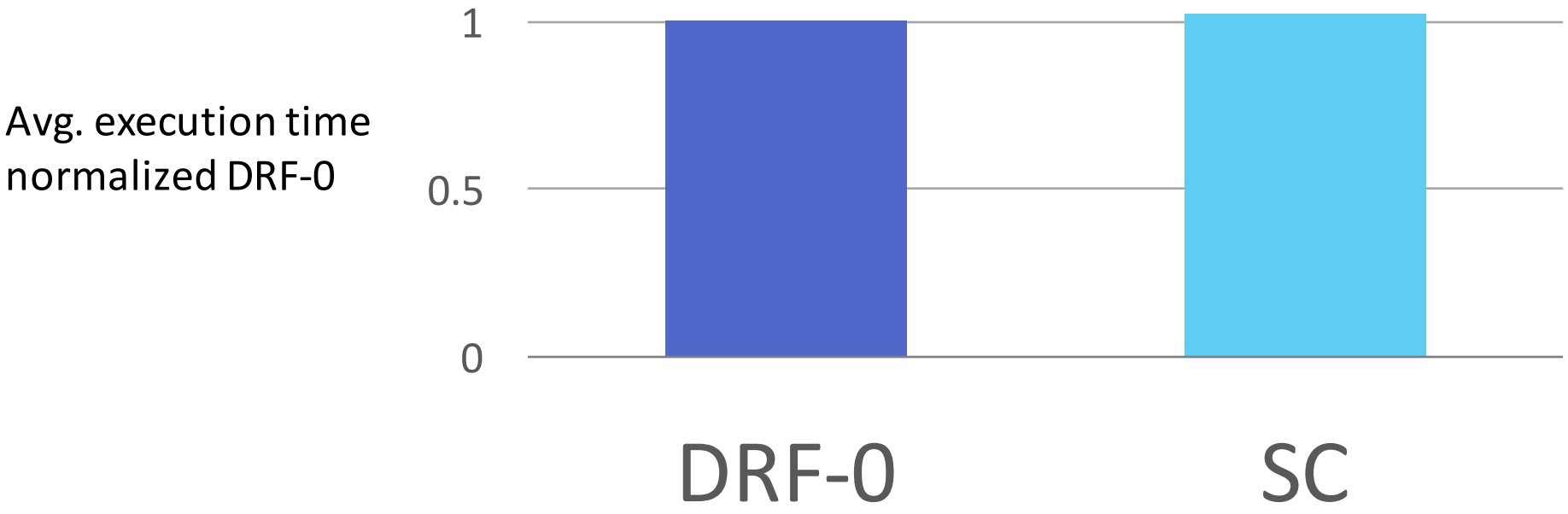
L1 Cache Coherence protocol

- MESI for write-back
- Valid/Invalid for write-through

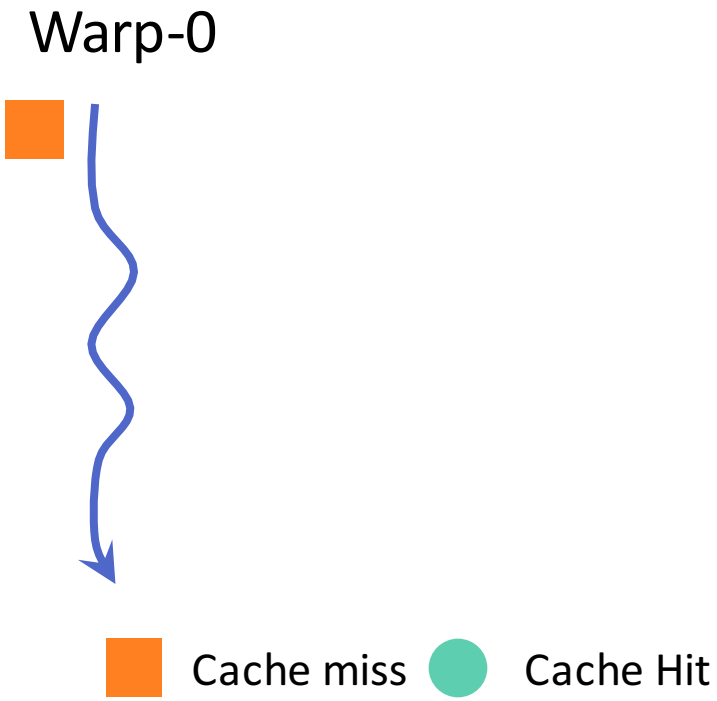
Benchmarks

- applications from Rodinia, Polybench benchmark suite
- Applications used in GPU coherence [Singh et al., HPCA'13]

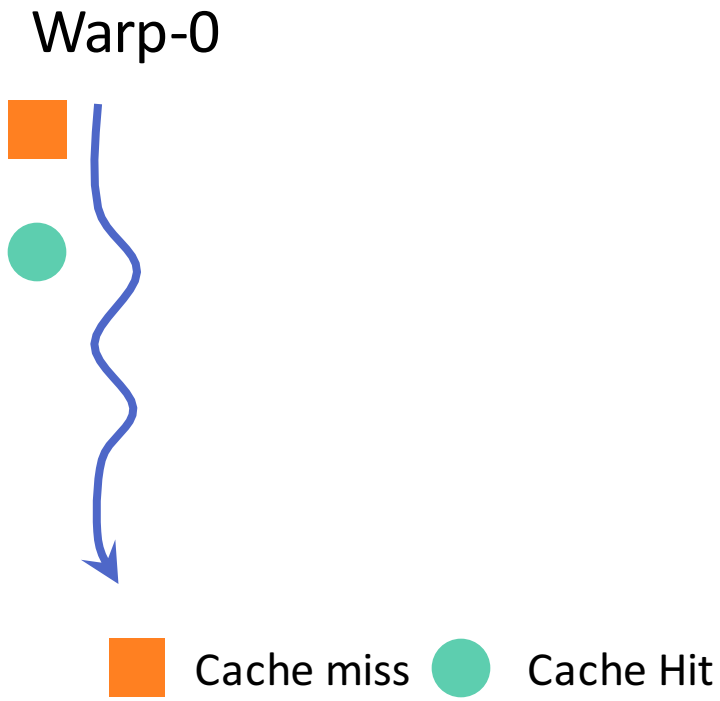
18 out of 22 applications incur insignificant SC overhead



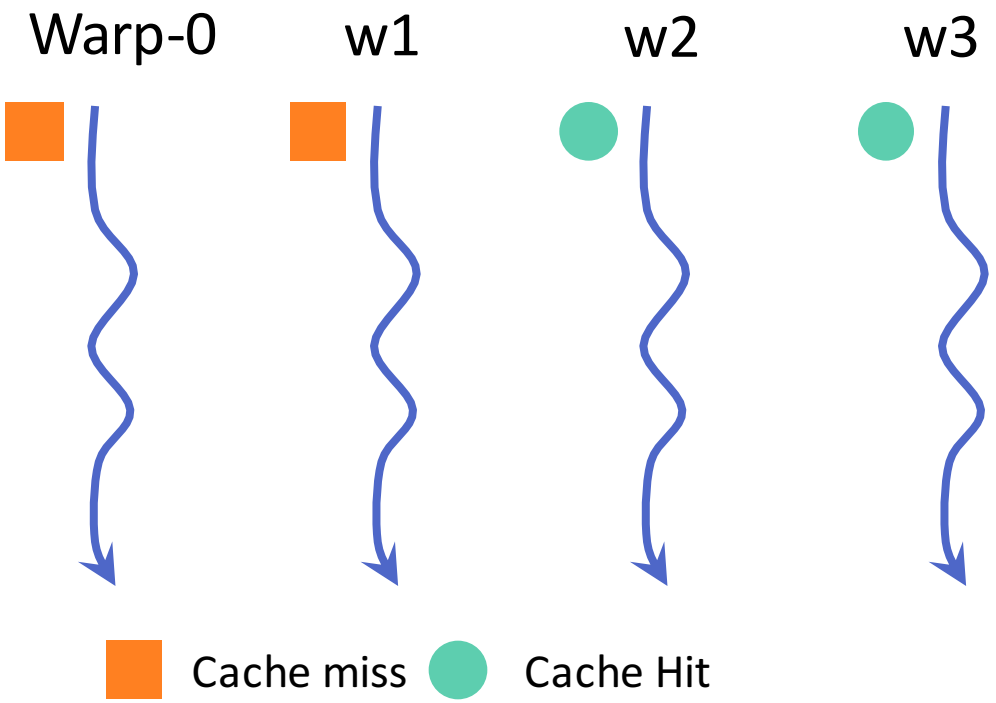
Warp-level-parallelism (WLP) masks SC overhead



Warp-level-parallelism (WLP) masks SC overhead

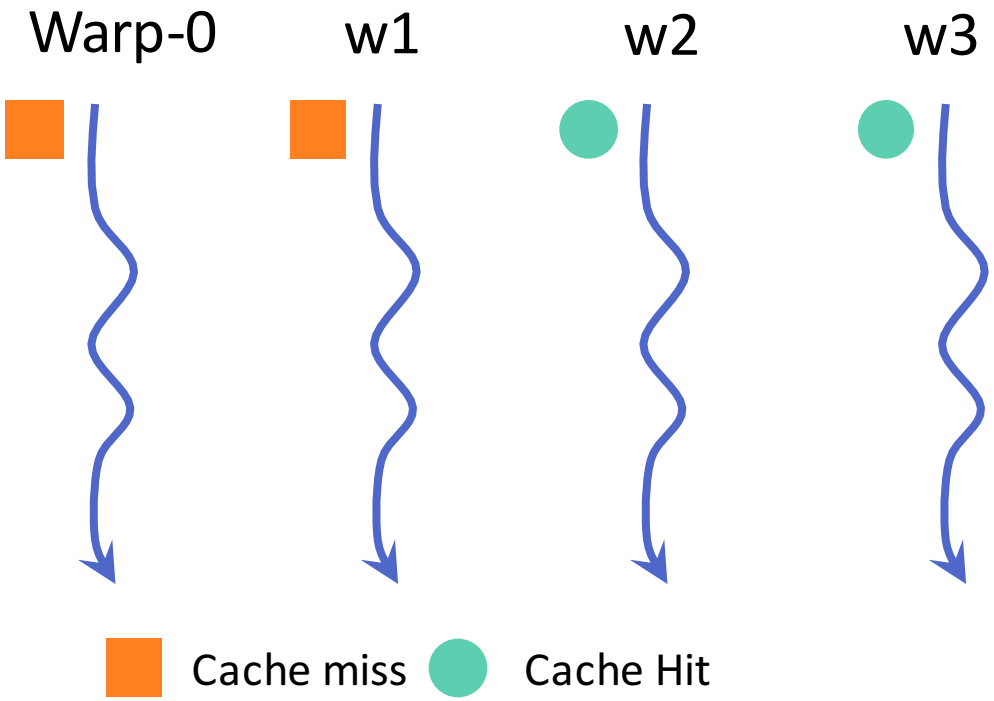


Warp-level-parallelism (WLP) masks SC overhead



SC can exploit inter-warp MLP

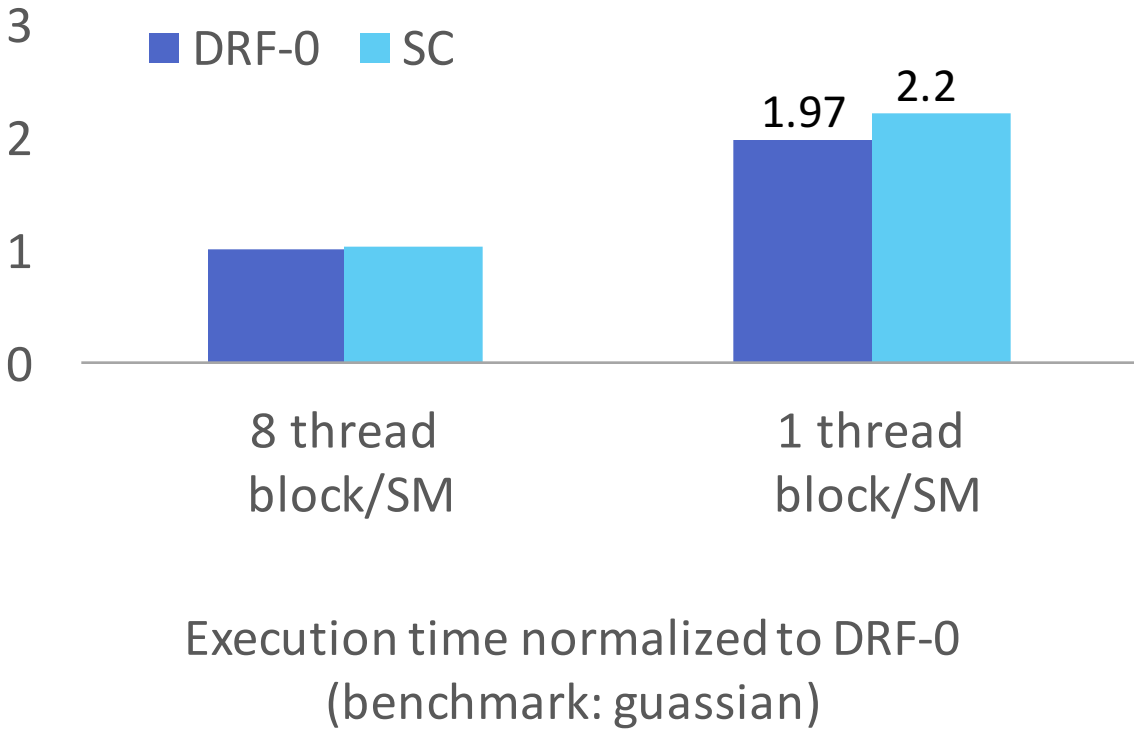
Warp-level-parallelism (WLP) masks SC overhead



SC can exploit inter-warp MLP

Adequate WLP => Low SC overhead

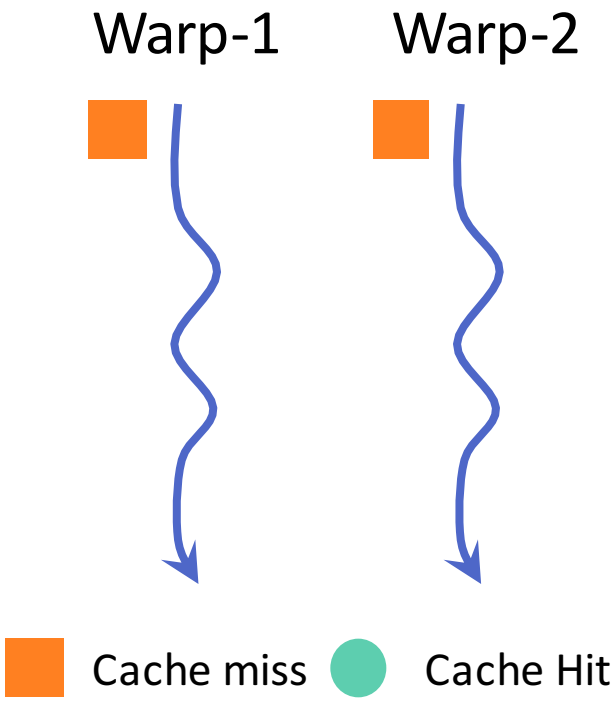
Warp-level-parallelism (WLP) masks SC overhead



SC can exploit inter-warp MLP

Adequate WLP => Low SC overhead

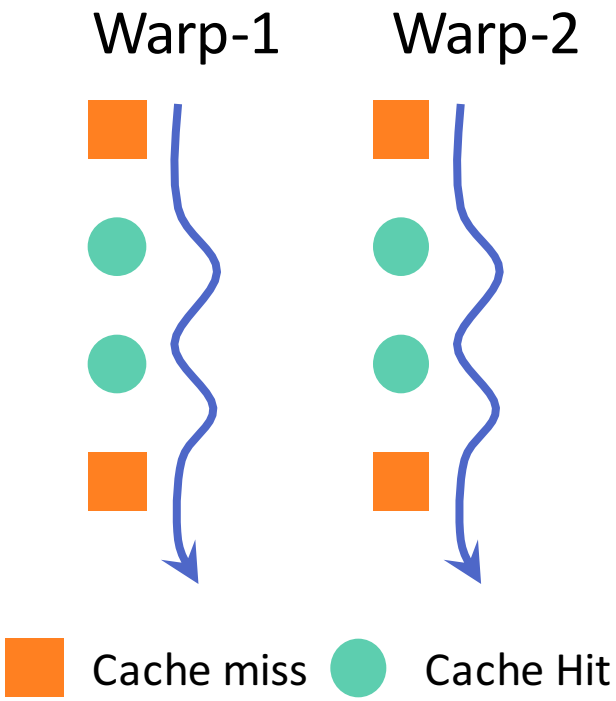
Higher SC overhead in apps where intra-warp MLP is important



Need for intra-warp MLP

- App has fewer warps
- Want fewer warps to avoid cache thrashing

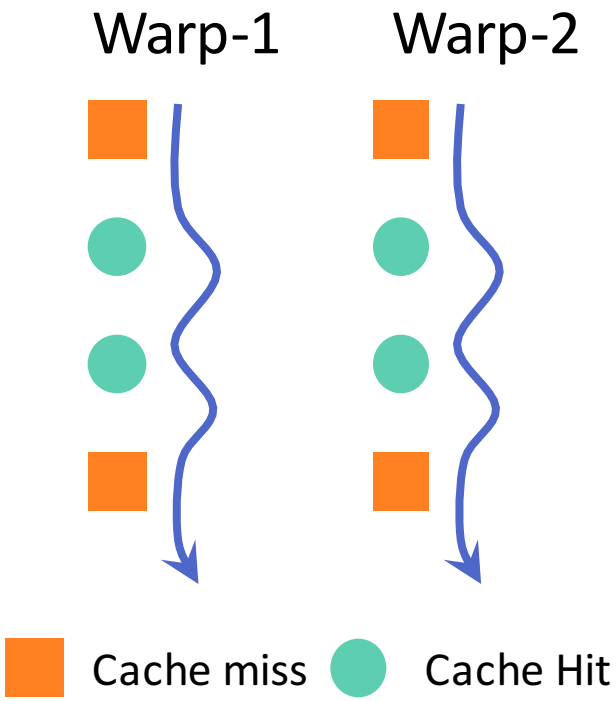
Higher SC overhead in apps where intra-warp MLP is important



Need for intra-warp MLP

- App has fewer warps
- Want fewer warps to avoid cache thrashing

Higher SC overhead in apps where intra-warp MLP is important

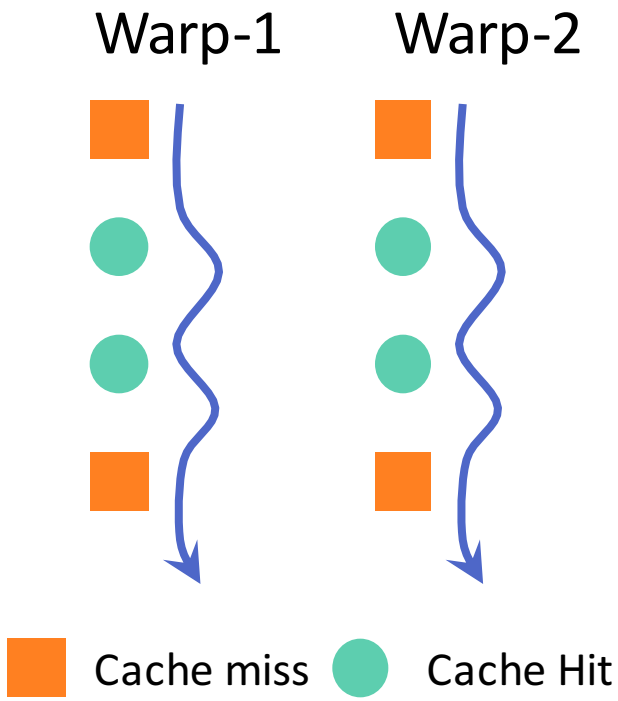


Need for intra-warp MLP

- App has fewer warps
- Want fewer warps to avoid cache thrashing

In-order execution limits the ability to exploit intra-warp MLP in DRF-0

Higher SC overhead in apps where intra-warp MLP is important



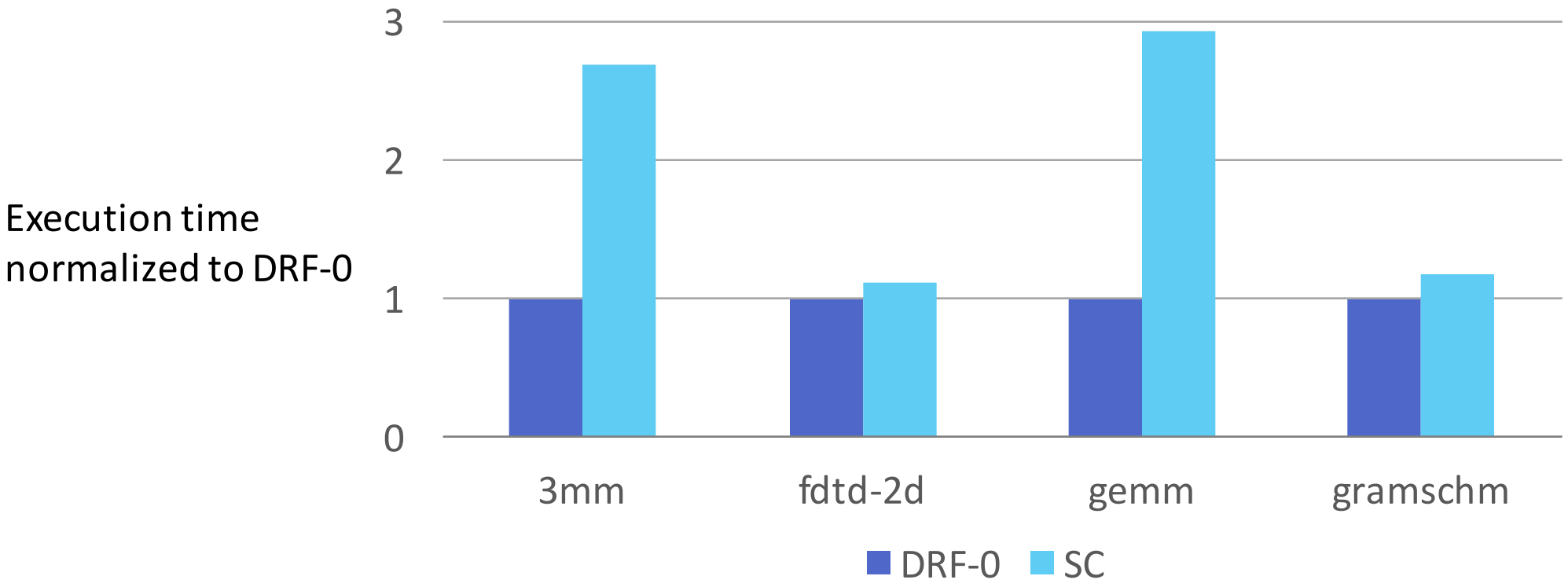
Unlike DRF-0, SC cannot exploit intra-warp MLP

Need for intra-warp MLP

- App has fewer warps
- Want fewer warps to avoid cache thrashing

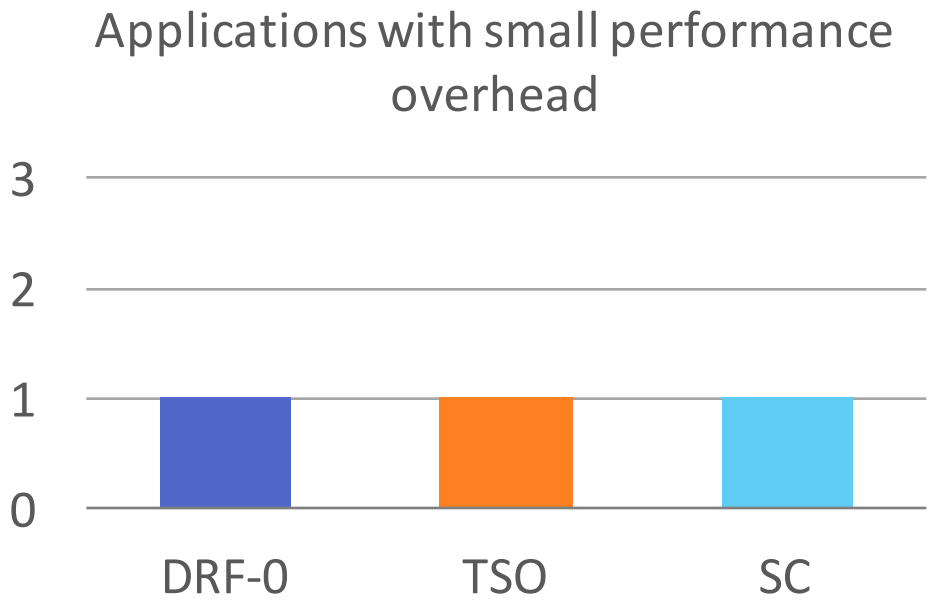
In-order execution limits the ability to exploit intra-warp MLP in DRF-0

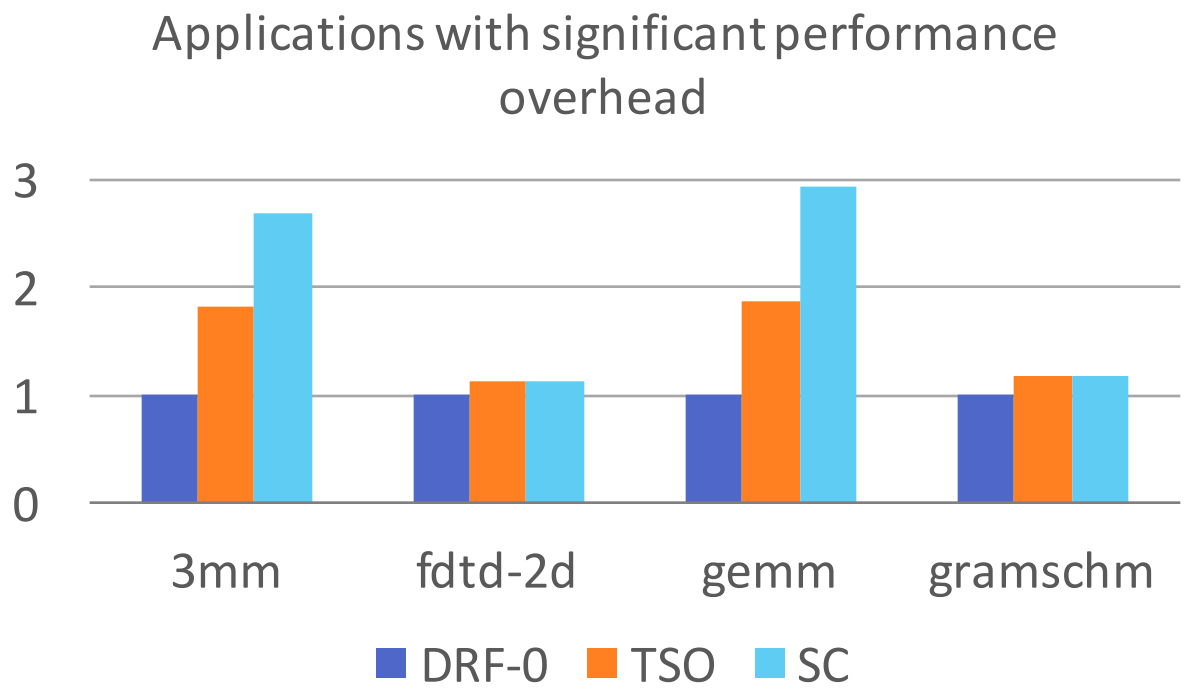
4 out of 22 applications exhibit significant SC overhead



Reason: Unlike DRF-0, SC cannot exploit intra-warp MLP

TSO is not suitable for GPUs





Execution time normalized to baseline (DRF-0)

TSO does not offer much performance or programmability advantage over SC

Roadmap

How GPU optimizations can violate memory ordering constraints?

Understand overhead of various memory ordering constraints in GPUs

- DRF-0, TSO, SC

Bridge the gap between SC and DRF-0

- Access-type aware GPU architecture

Access-type Aware Optimization for GPU

Relax ordering constraint for safe accesses

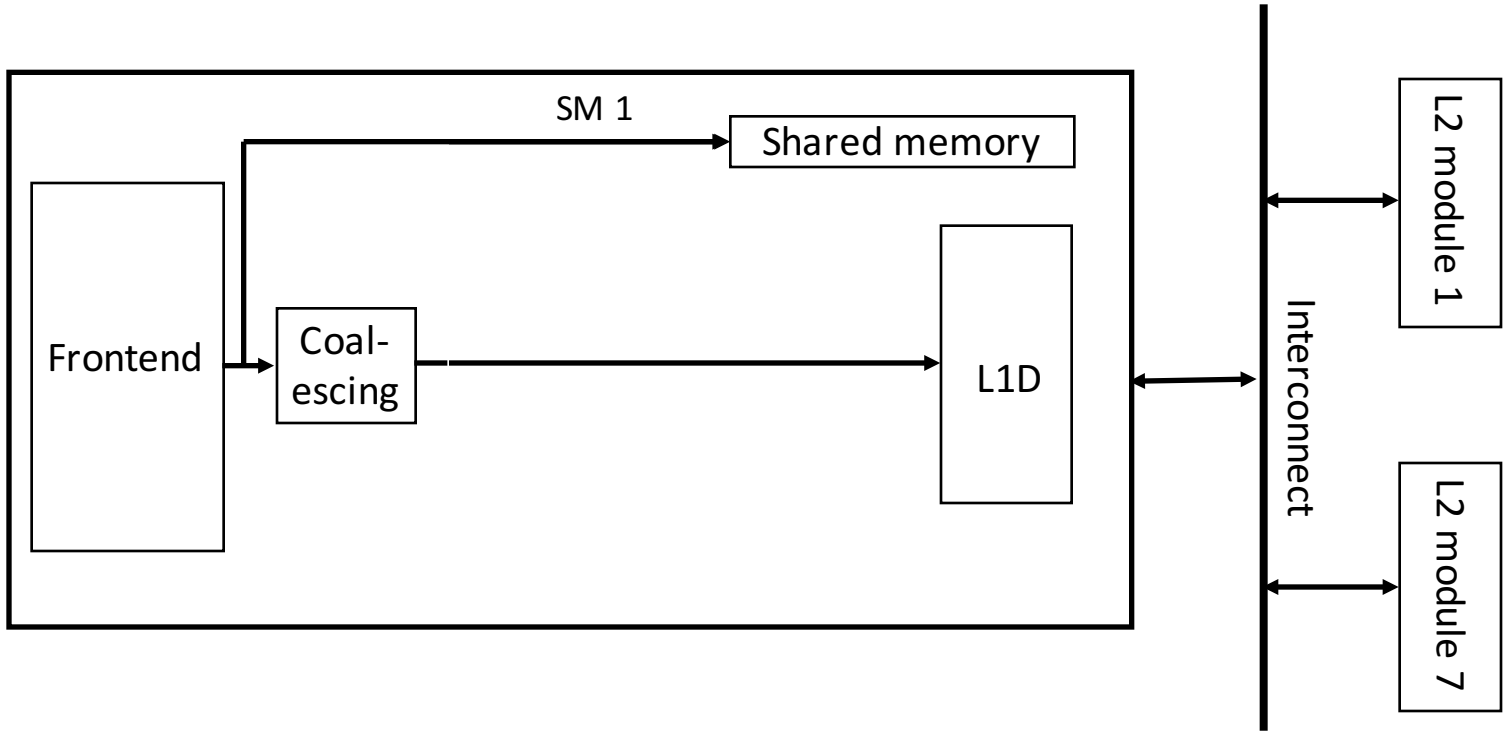
- Accesses to thread-private or read-only location are safe
- (Shasha & Snir, TOPLAS'88, Singh et al., ISCA'12)

Thread-level classification is prohibitively expensive

- Classify accesses as unsafe or SM-safe

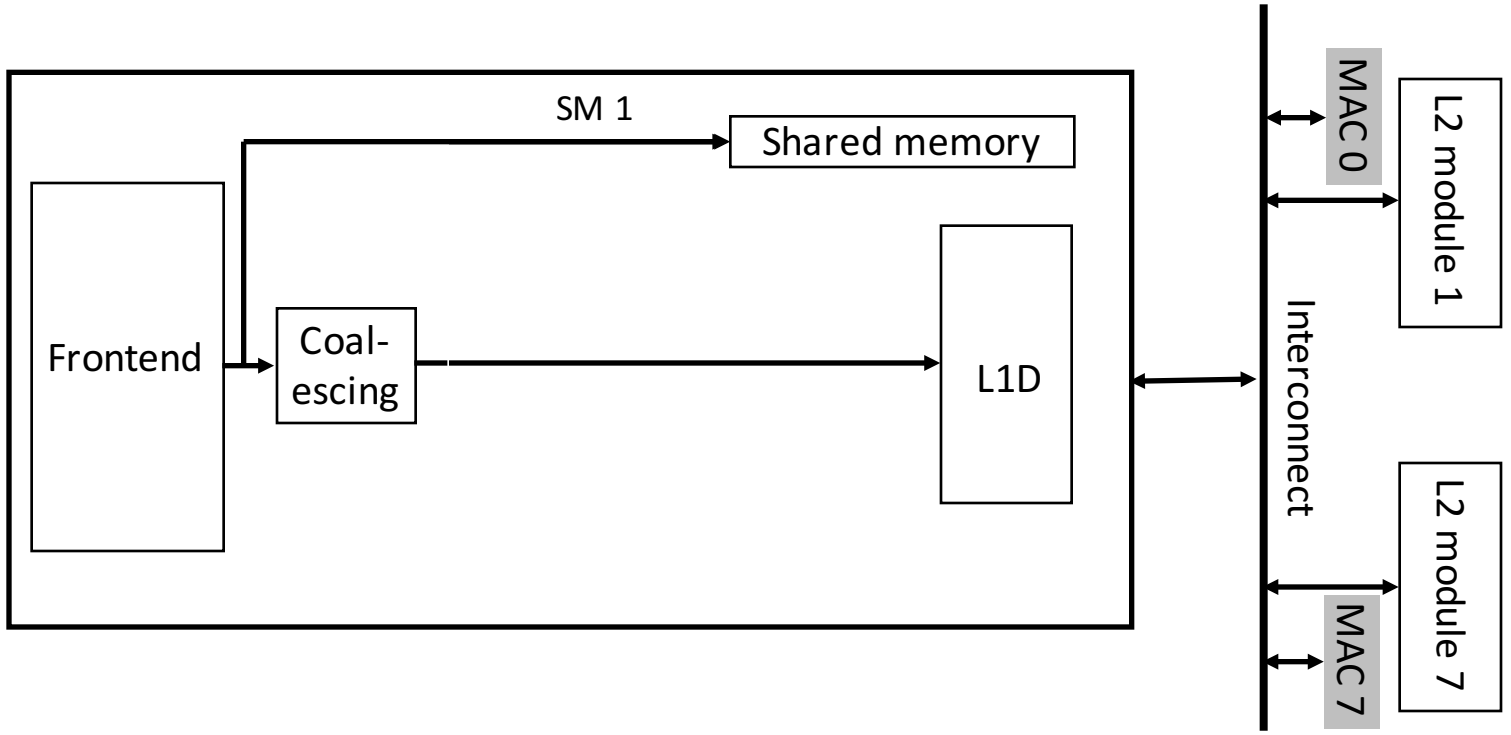
Type-aware SC Extensions to Baseline GPU

1. Classify memory accesses



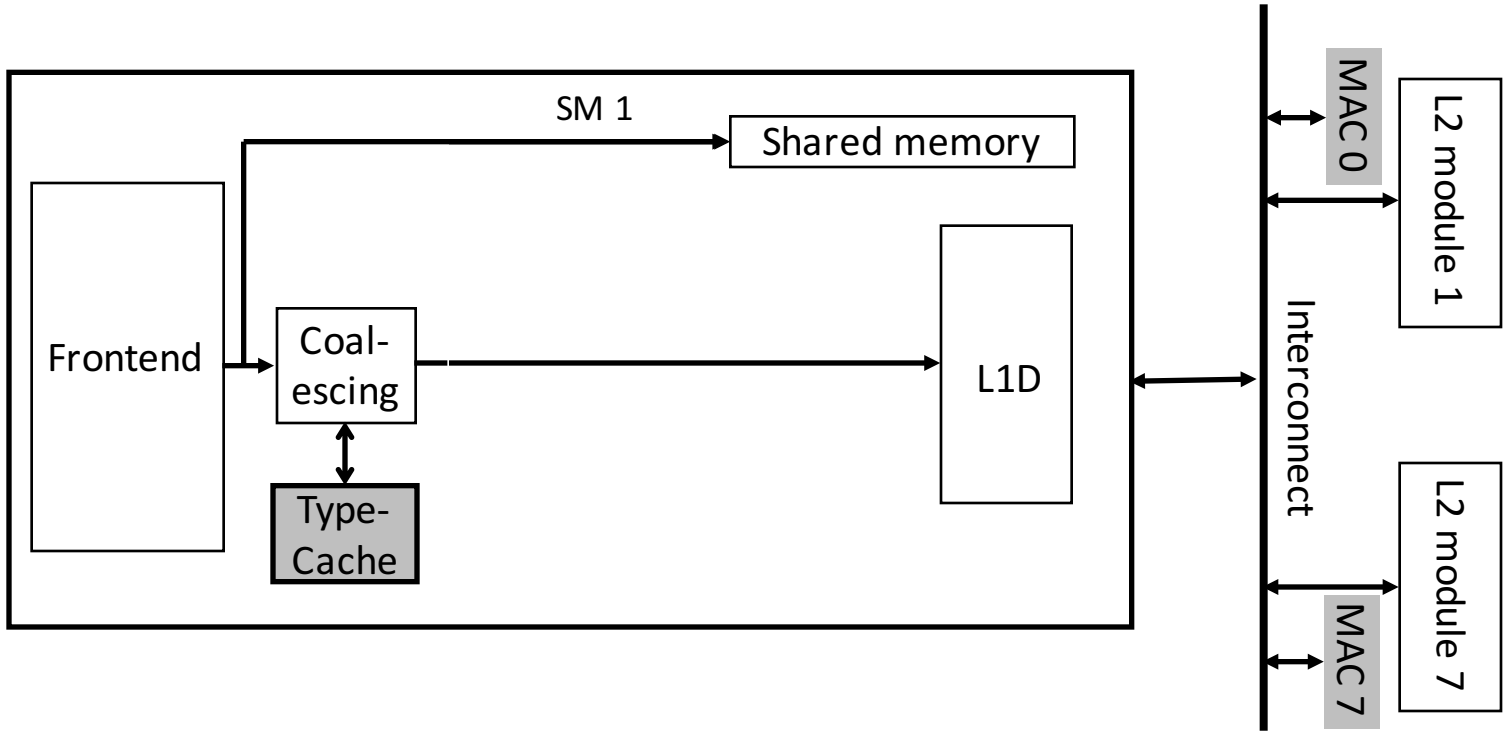
Type-aware SC Extensions to Baseline GPU

1. Classify memory accesses



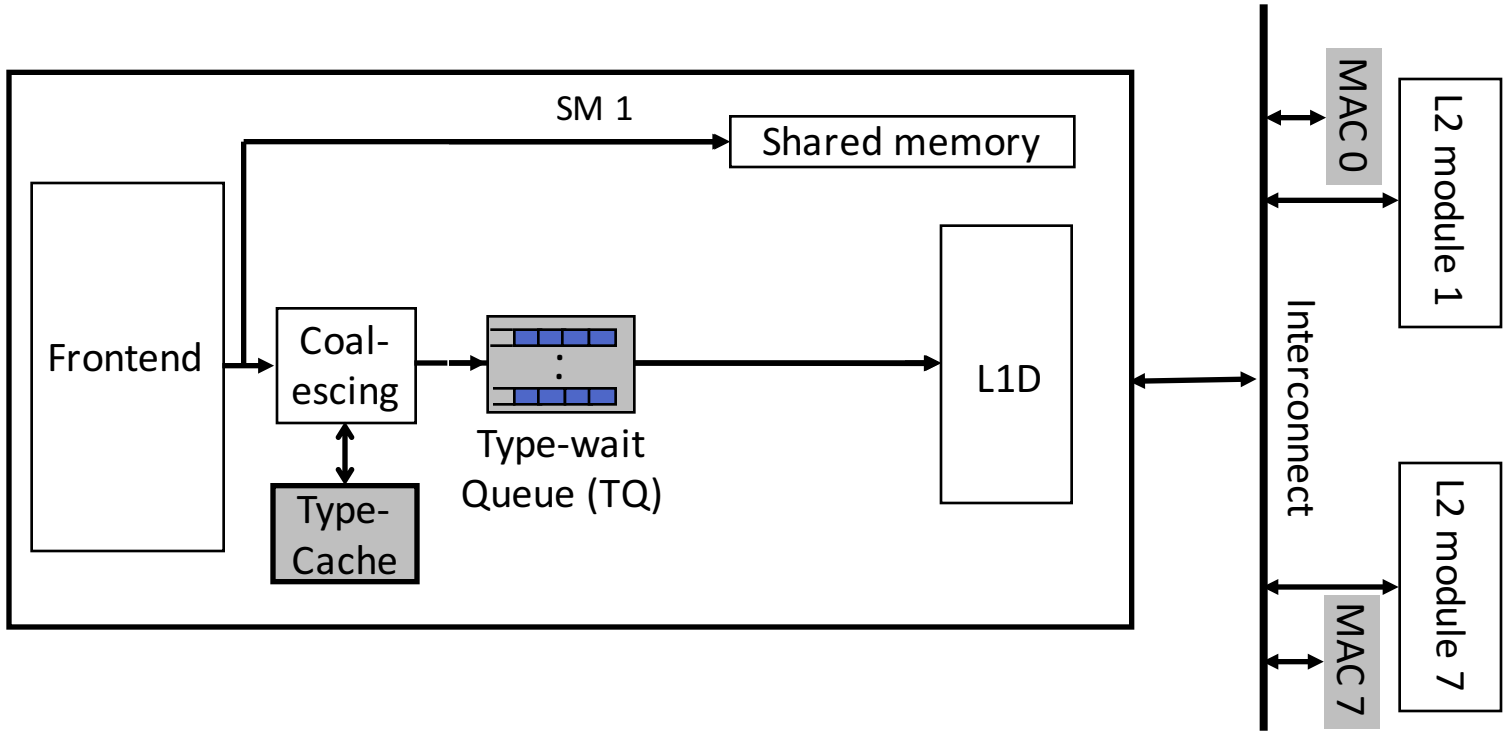
Type-aware SC Extensions to Baseline GPU

1. Classify memory accesses



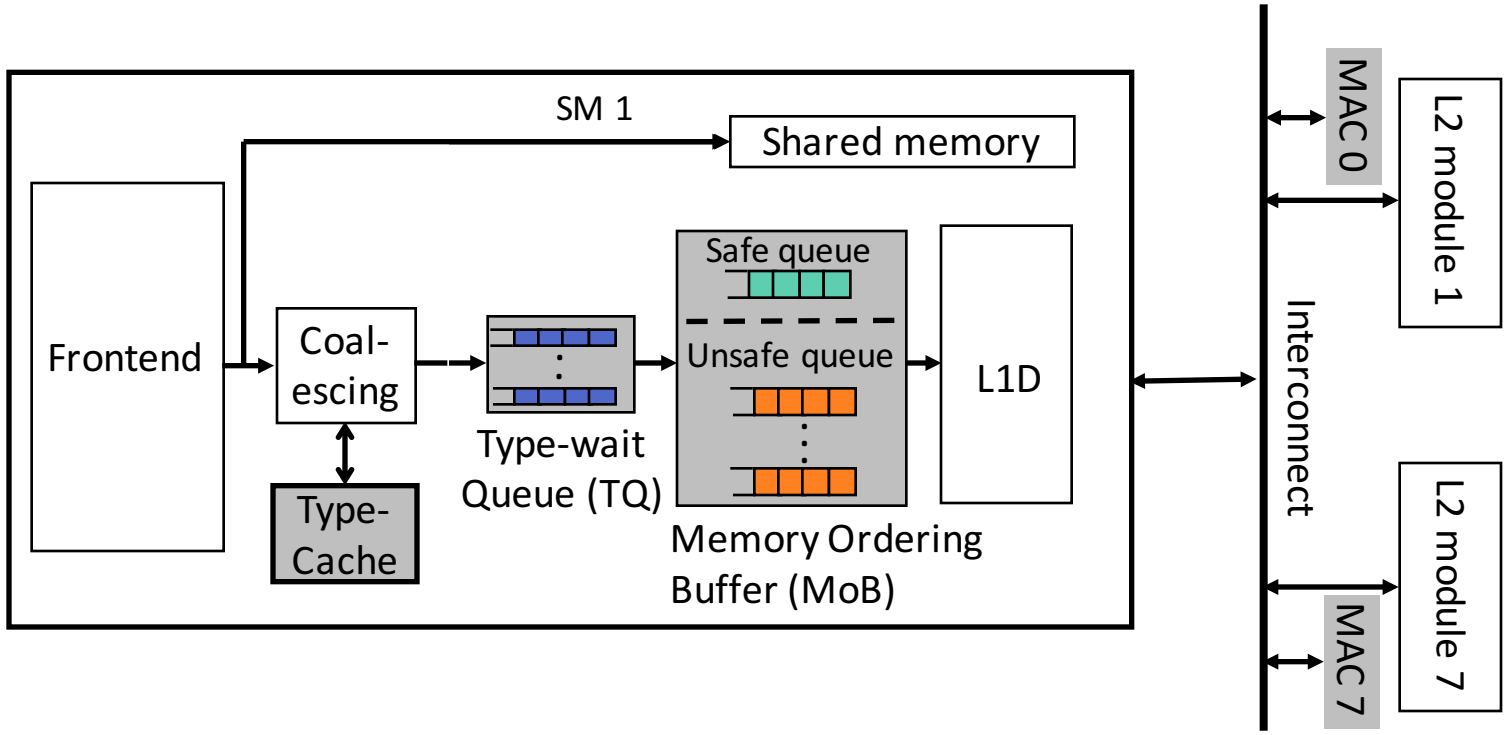
Type-aware SC Extensions to Baseline GPU

1. Classify memory accesses



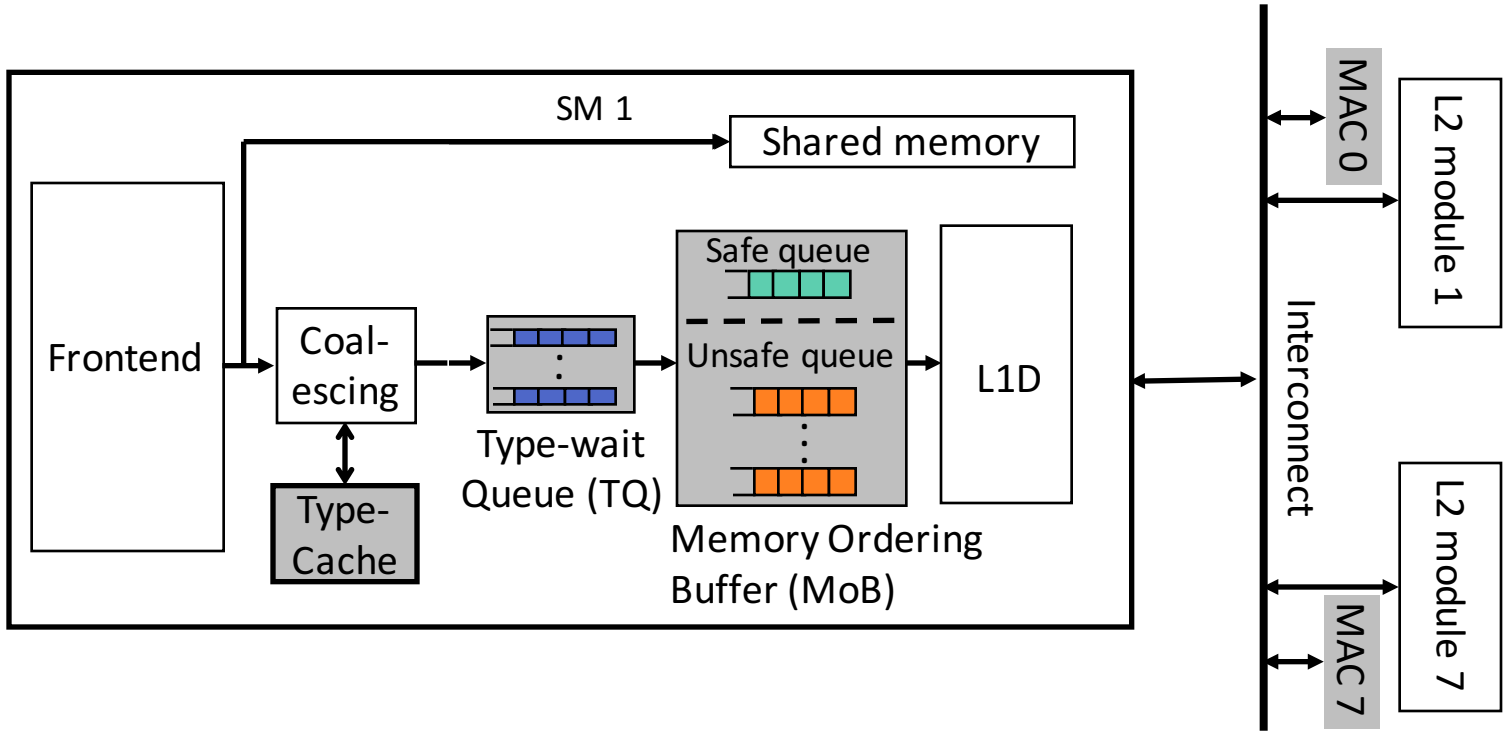
Type-aware SC Extensions to Baseline GPU

1. Classify memory accesses
2. Relax ordering constraints for SM-safe accesses



Type-aware SC Extensions to Baseline GPU

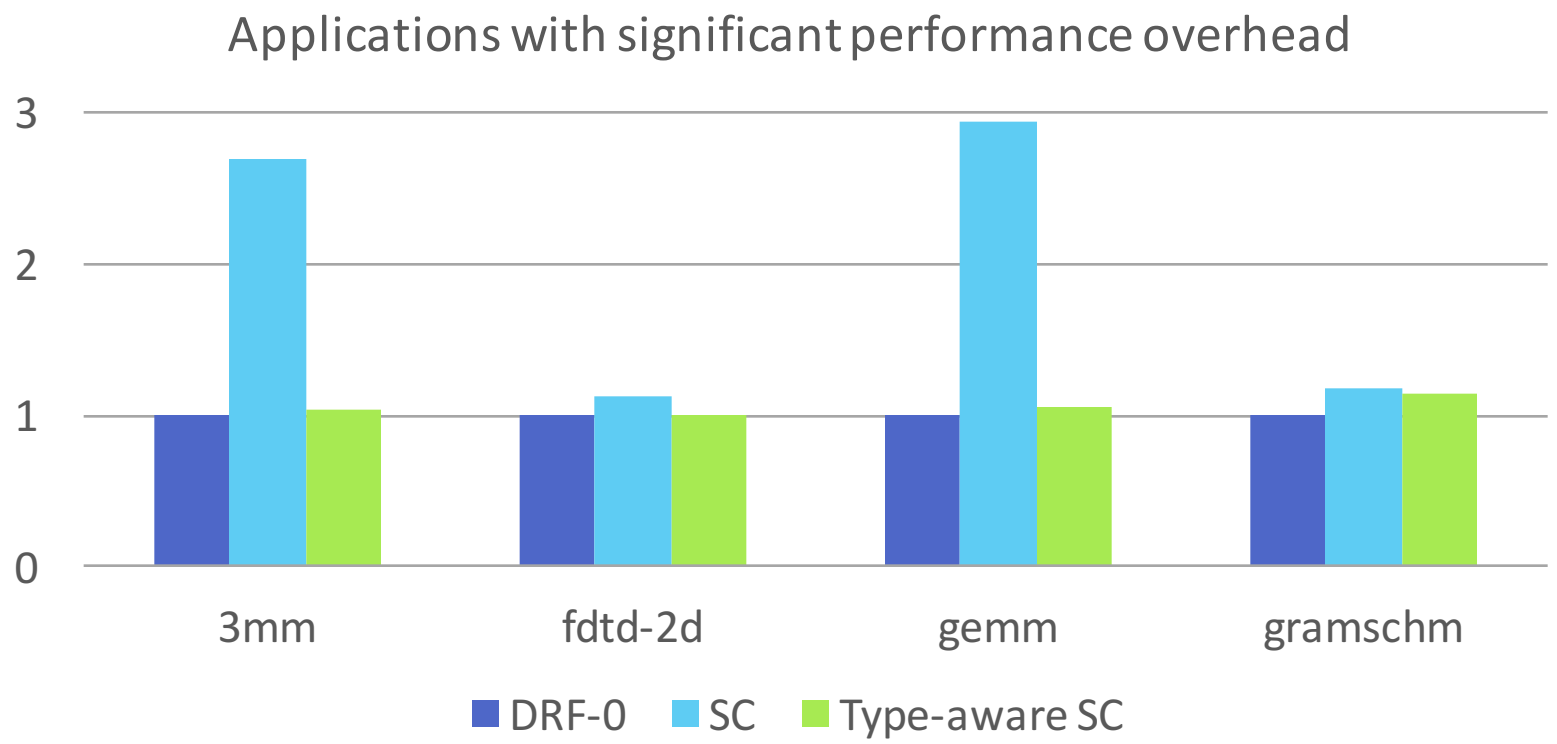
1. Classify memory accesses
2. Relax ordering constraints for SM-safe accesses



Problem: Ensuring ordering among conflicting accesses to an SM-safe location within an SM

✓ See details in the paper

Type-aware SC incurs only small performance overhead

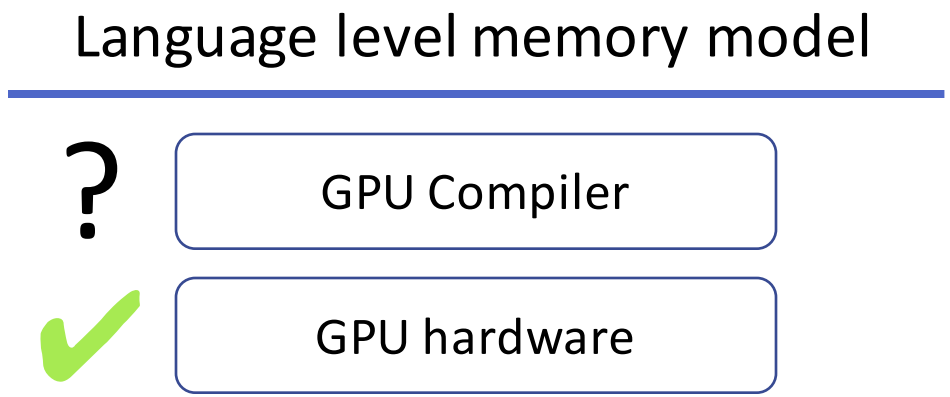


Proposed design is able to exploit intra-warp MLP for SM-safe accesses

Future research directions

Build SC-preserving GPU compiler

Overhead SC-preserving LLVM compiler for C++: ~2% [Marino et al., PLDI'11]



Conclusion

Quantified performance overhead of various memory models in GPUs

TSO is unattractive for GPUs: No performance or programmability benefits over SC

Performance gap between SC and DRF-0 is insignificant for most applications

Access type aware optimization bridges the gap in remaining applications

Questions?