Testing GPU Memory Models

Daniel Poetzl

Joint work with

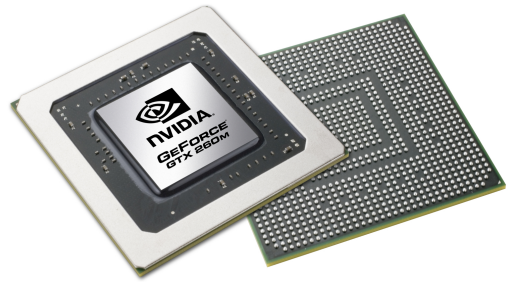
Jade Alglave (UCL), Mark Batty (Cambridge), Alastair Donaldson (Imperial), Ganesh Gopalakrishnan (Utah), Tyler Sorensen (Utah), John Wickerson (Imperial)

Outline

Introduction

Graphics Processing Units (GPUs)

- GPUs have traditionally been designed to accelerate graphics applications
  - 3D games
  - Video processing



- General-purpose computing on GPUs (GPGPU) is becoming increasingly widespread
  - Regular applications:
    - Weather forecasting
    - Brute-force password cracking
  - Irregular applications:
    - Graph traversal
- Numerous papers are published each year that aim to accelerate traditional algorithms using GPUs

Graphics Processing Units (GPUs)

GPUs have found their way into many types of computer systems:

- Desktops and Laptops
- Game consoles
- Mobile devices:
  - iPhone 5S
  - Samsung Galaxy S
- Cars:
  - Audi self-driving car
    - Video processing
    - Safety-critical (!)
  - Tesla Motors Model S
    - Infotainment system
- Supercomputers

GPUs in Supercomputers

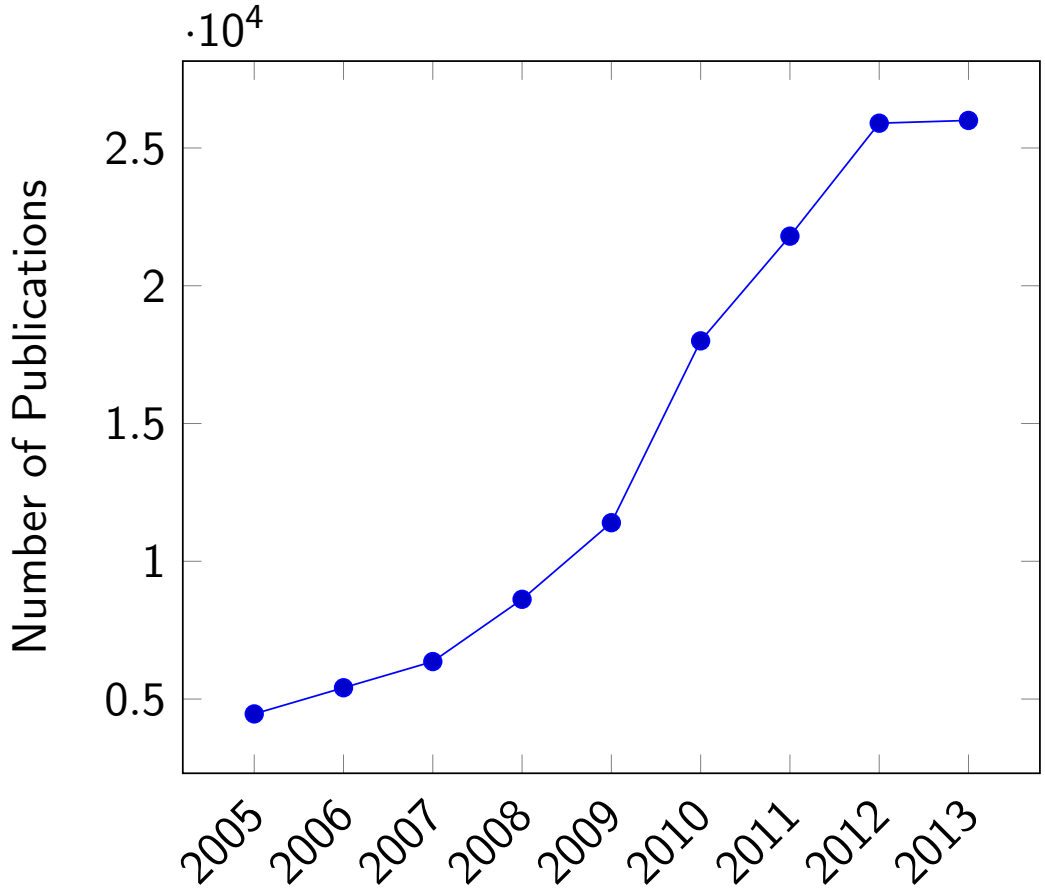
- Green500 list of most energy-efficient supercomputers
- All top ten places are occupied by systems using GPUs

Rank	GFLOPS/W	Name	Site
1	4.5	Tsubame	Tokyo
2	3.6	Wilkes	Cambridge
3	3.5	HA-PACS	Tsukuba
4	3.2	Piz Daint	Lugano
5	3.1	Romeo	Champagne-Ardenne

- For comparison:
  - Tianhe-2 (Guangzhou, 1st in Top500): 1.9 GFLOPS/W
  - Stampede (Austin, 7th in Top500): 1.1 GFLOPS/W

GPU Research

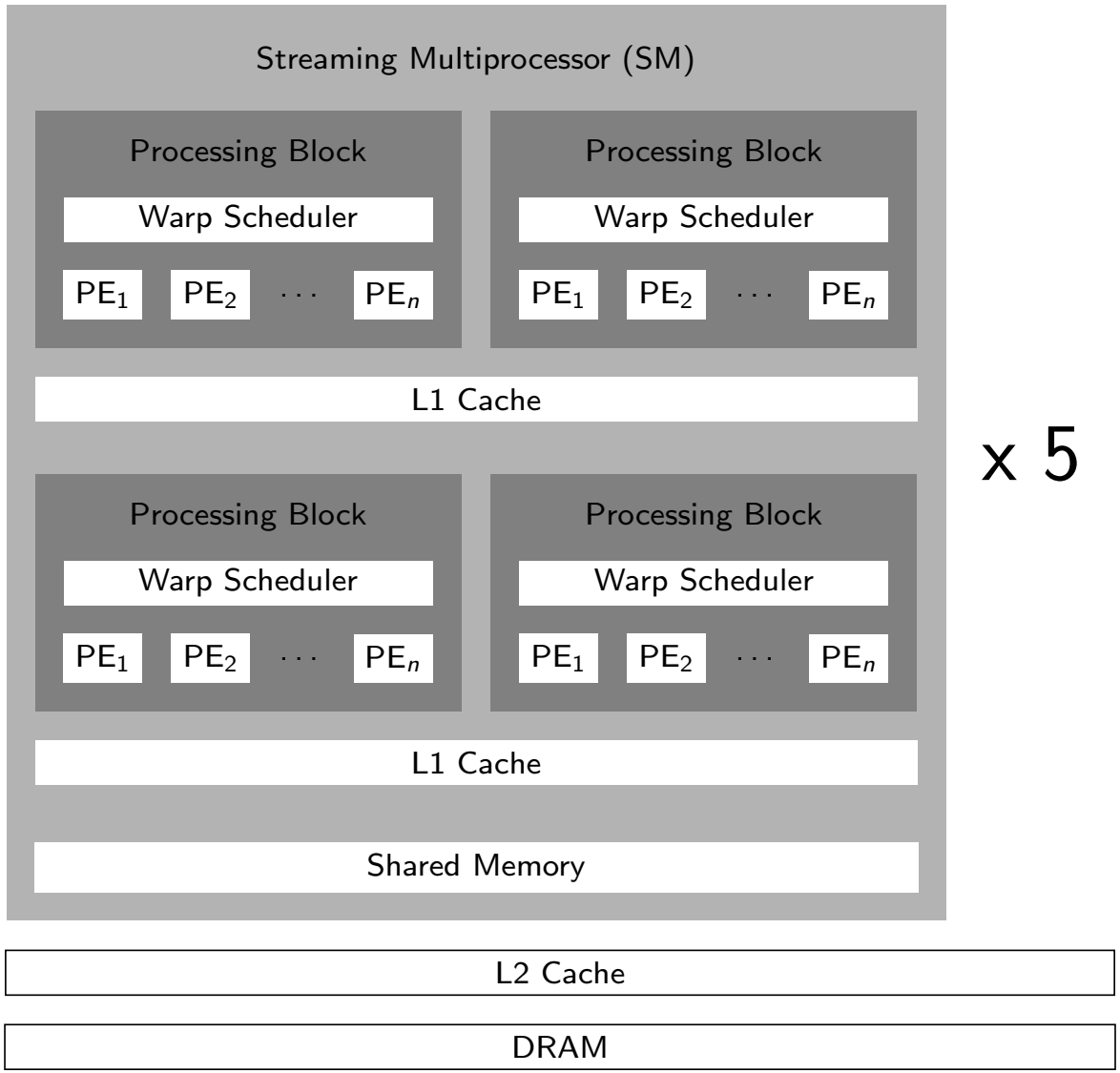
- The number of publications/year on GPU programming has continuously grown over the last years (Source: Google Scholar):



GPU architectures

GPU Architectures

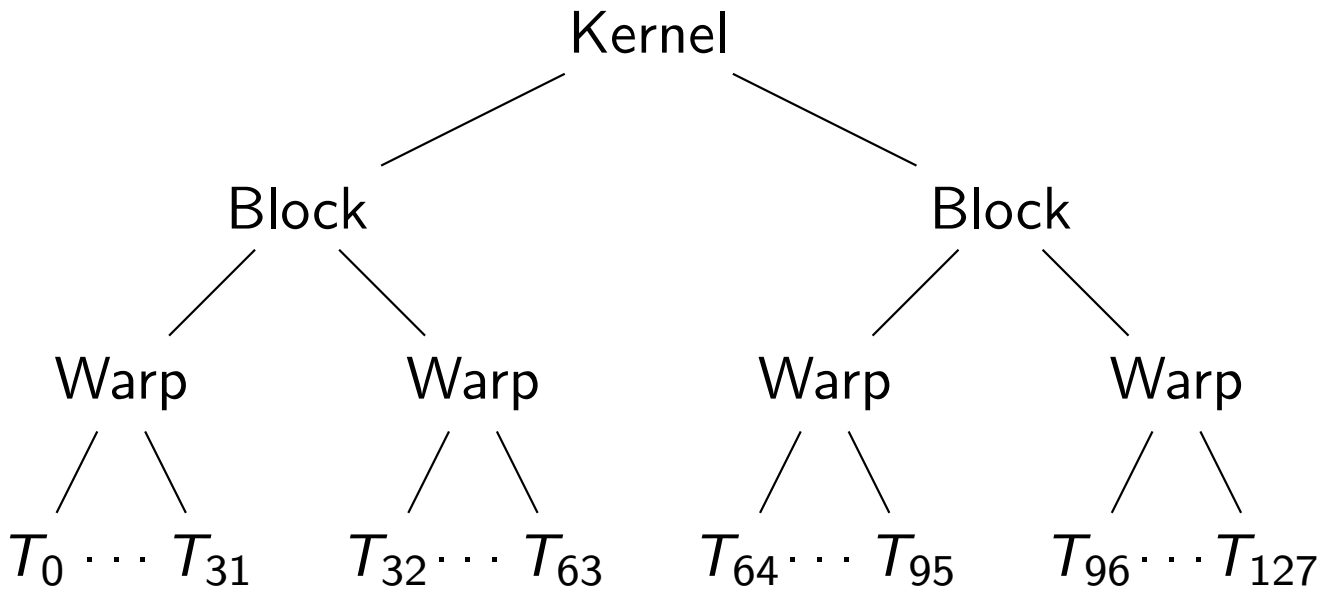
Nvidia's Maxwell (2014)



Programming Model

CUDA

- CUDA is Nvidia's framework for general-purpose computing on GPUs
- Threads are hierarchically organized:



- Different memory spaces: global, shared, local, constant, texture, parameter

Vector Addition

CPU implementation

- Summing two vectors of size N in C:

```
void add(int *a, int *b, int *c) {
  for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
  }
}
```

- Runtime: $\mathcal{O}(N)$

Vector Addition

CUDA C implementation

- Summing two vectors of size N in CUDA C:

```
__global__ void add(int *a, int *b, int *c) {
  int tid = blockIdx.x;
  if (tid < N)
    c[tid] = a[tid] + b[tid];
}
```

- If number of processing elements is greater or equal to N
- Runtime: $\mathcal{O}(1)$

CPUs vs. GPUs

- Key differences between CPUs and GPUs:

	CPUs	**GPUs**
Cores	Few	Many
Core complexity	High	Low
Caches	Large	Small
Memory bandwidth	Low	High
Context switches	Slow	Fast
Explicit concurrency hierarchy	No	Yes
Different memory spaces	No	Yes

Weak Memory 101

Interleaved Execution

- A simple model of concurrency is Lamport's *sequential consistency* (SC), i. e. interleaved execution

```
1 // Init
2 data = flag = 0
```

```
1 // Producer
2 data = 0x7f
3 flag = 1
```

```
1 // Consumer
2 while (flag == 0) {}
3 assert(data != 0)
```

Interleaved Execution

- A simple model of concurrency is Lamport's *sequential consistency* (SC), i. e. interleaved execution

```
1 // Init
2 data = flag = 0
```

```
1 // Producer
2 data = 0x7f
3 flag = 1
```

```
1 // Consumer
2 while (flag == 0) {}
3 assert(data != 0)
```

- Example interleaving:

```
1 data = 0x7f
    1 flag == 0 ?
2 flag = 1
    1 flag == 0 ?
    2 assert(data != 0)
```

- Assertion is satisfied on all interleavings.

Interleaved Execution

- A simple model of concurrency is Lamport's *sequential consistency* (SC), i. e. interleaved execution

```
1 // Init
2 data = flag = 0
```

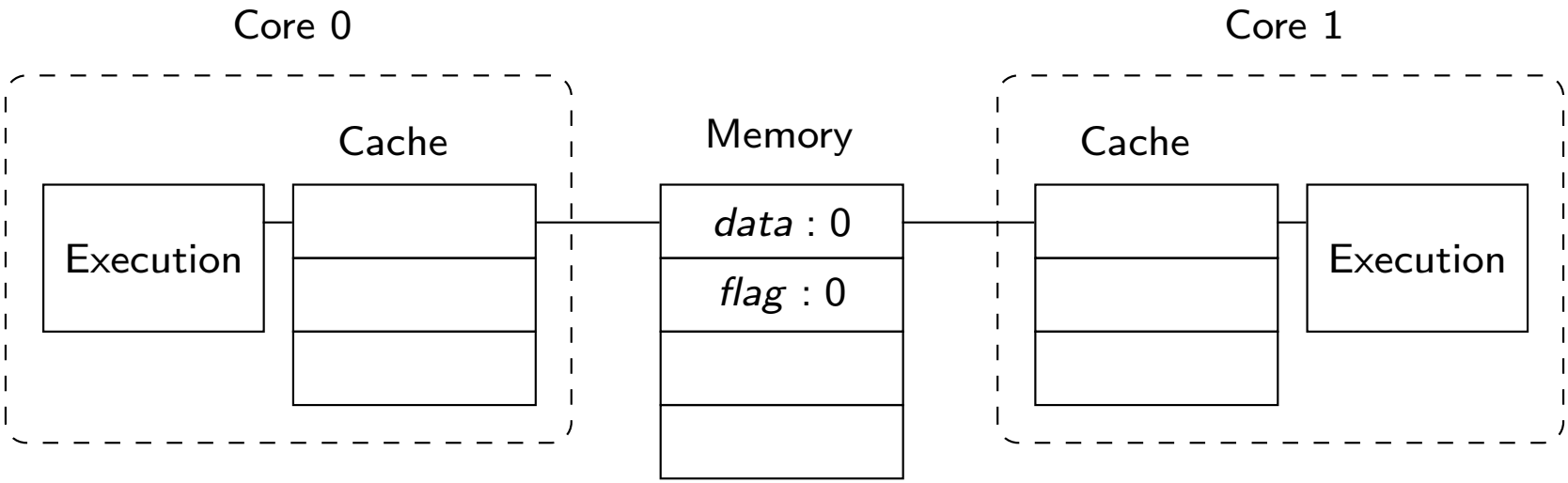
```
1 // Producer
2 data = 0x7f
3 flag = 1
```

```
1 // Consumer
2 while (flag == 0) {}
3 assert(data != 0)
```

- Multi- and manycore processors exhibit weak memory consistency:
  - Out-of-order execution
  - Speculative execution
  - Caching
- Assertion can fail on those systems!
- Synchronization algorithms (Dekker, Peterson, . . .) we've been taught in school do not work on multicore systems

Weak Memory Consistency

Caching

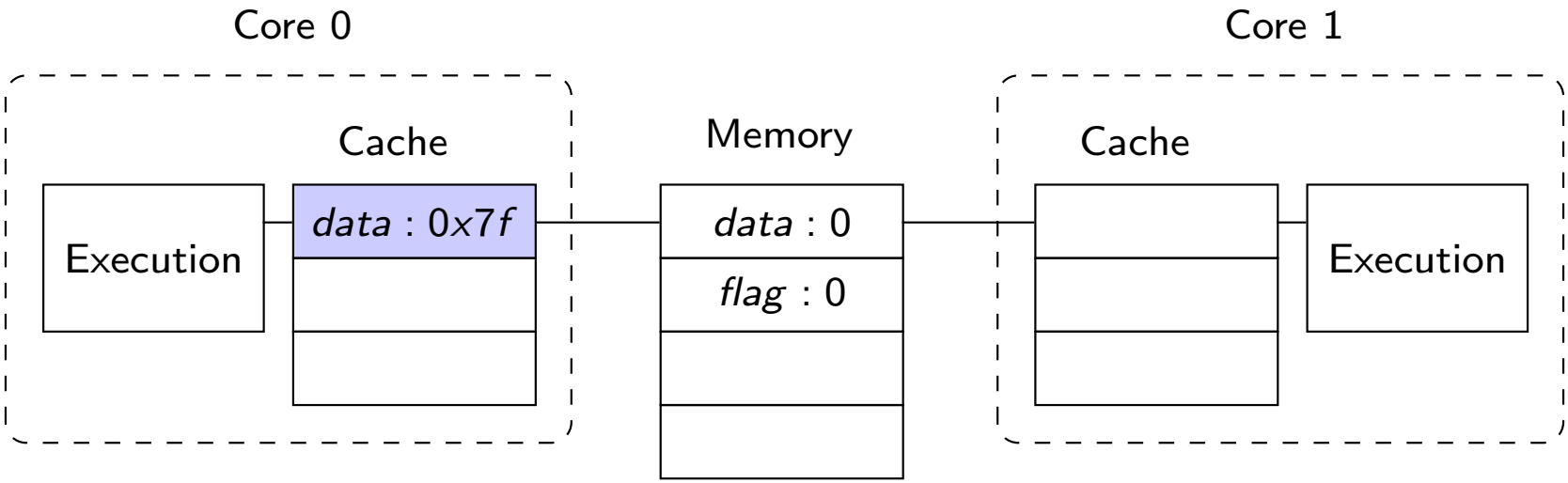



```
// Init
flag = data = 0

// Producer
data = 0x7f
flag = 1
```

```
// Consumer
while (flag == 0) {}
assert(data != 0)
```

Weak Memory Consistency

Caching

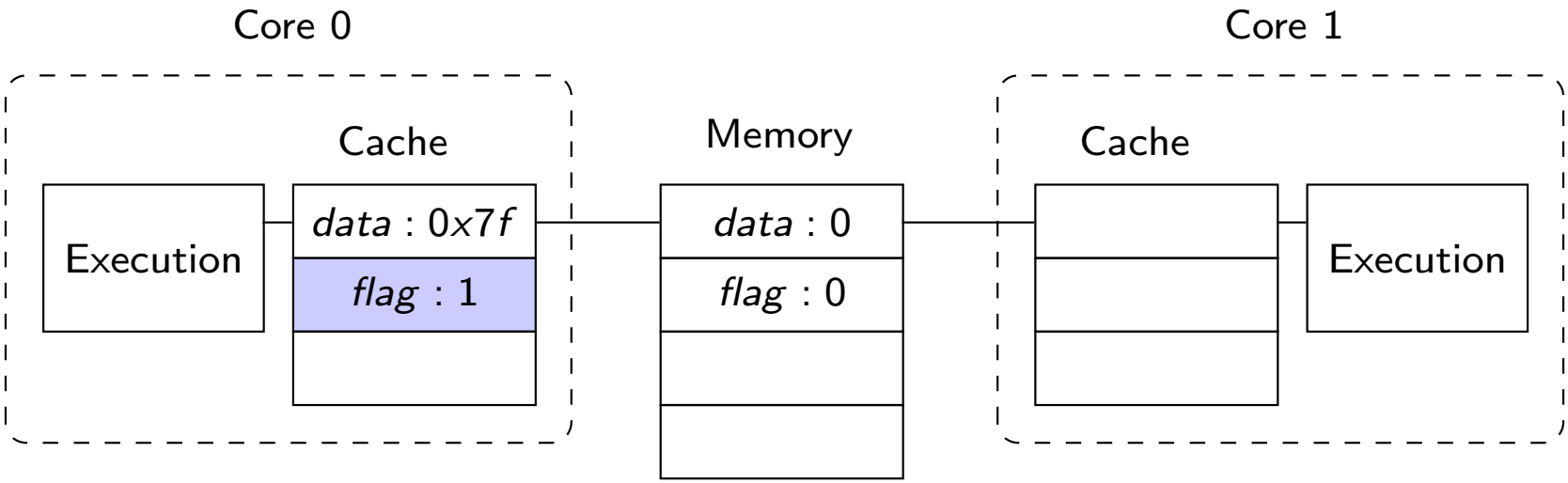



```
// Init
flag = data = 0

// Producer
data = 0x7f
flag = 1
```

```
// Consumer
while (flag == 0) {}
assert(data != 0)
```

Weak Memory Consistency

Caching

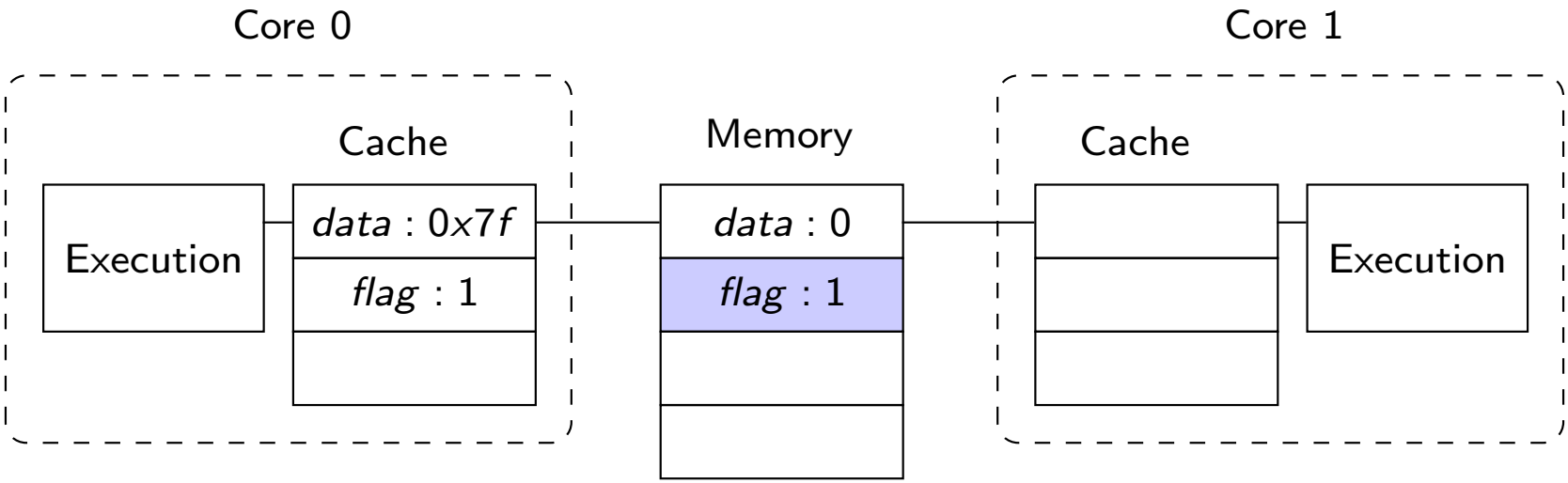



```
// Init
flag = data = 0

// Producer
data = 0x7f
flag = 1
```

```
// Consumer
while (flag == 0) {}
assert(data != 0)
```

Weak Memory Consistency

Caching



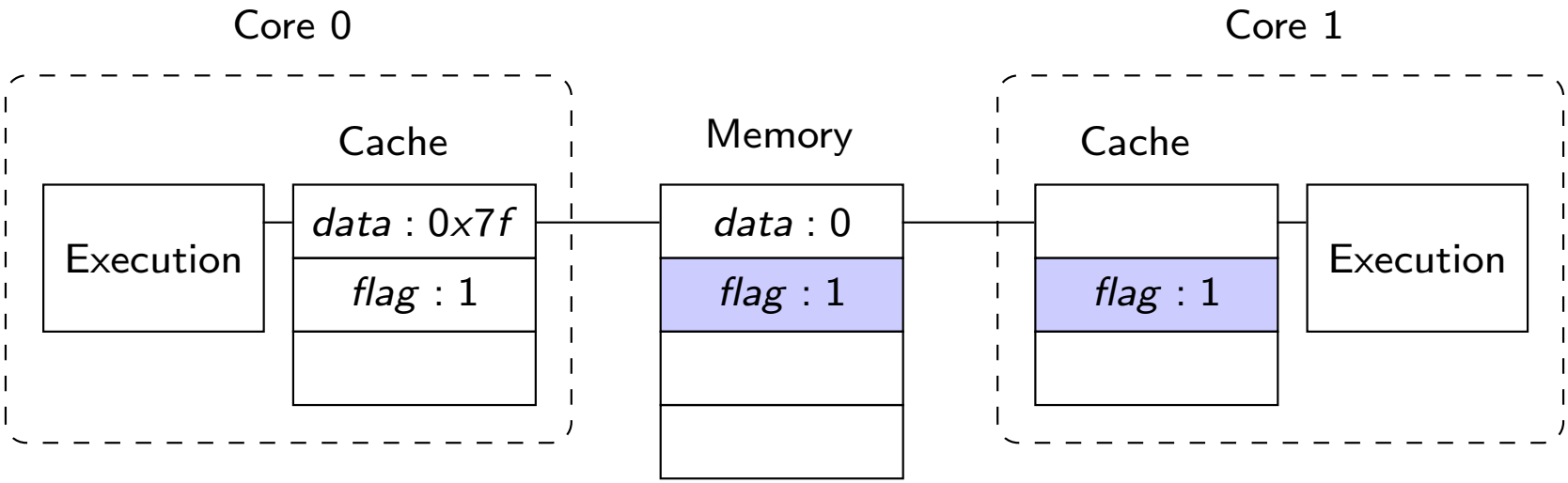

```
// Init
flag = data = 0

// Producer
data = 0x7f
flag = 1
```

```
// Consumer
while (flag == 0) {}
assert(data != 0)
```

Cache coherency protocol commits **flag** before **data** to main memory.

Weak Memory Consistency

Caching

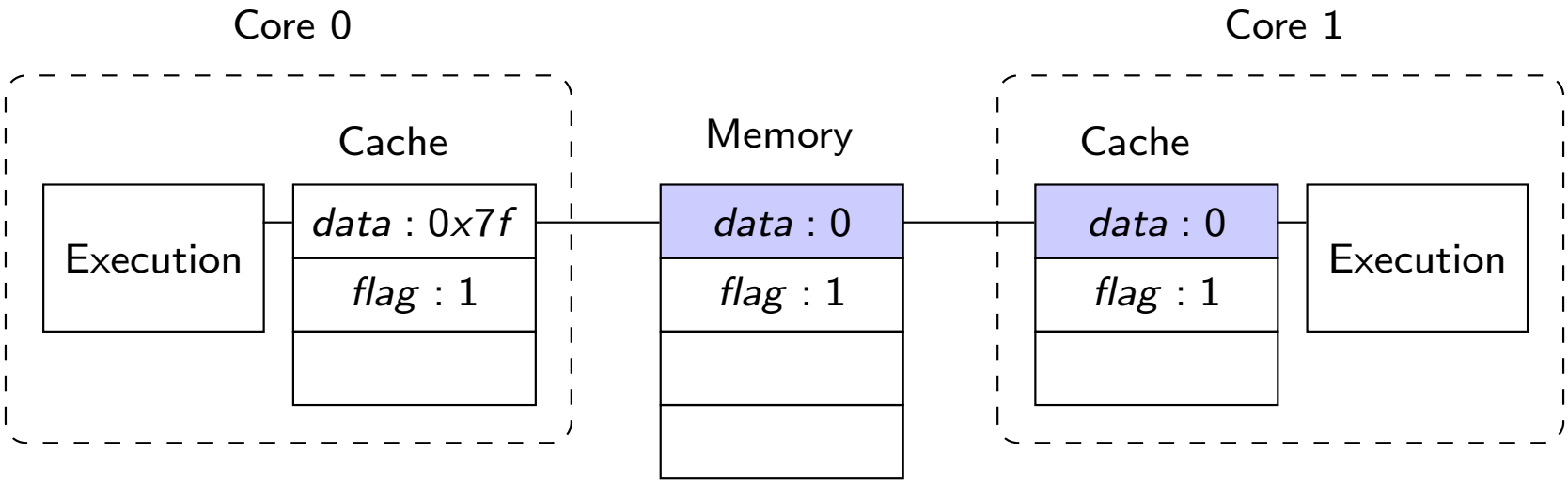



```
// Init
flag = data = 0

// Producer
data = 0x7f
flag = 1
```

```
// Consumer
while (flag == 0) {}
assert(data != 0)
```

Weak Memory Consistency

Caching




```
// Init
flag = data = 0

// Producer
data = 0x7f
flag = 1
```

```
// Consumer
while (flag == 0) {}
assert(data != 0) ↯
```

Memory Barriers

- CPUs/GPUs provide memory barrier instructions to enforce ordering constraints on memory accesses.
- Expensive: 100s of clock cycles
- Different types of barriers

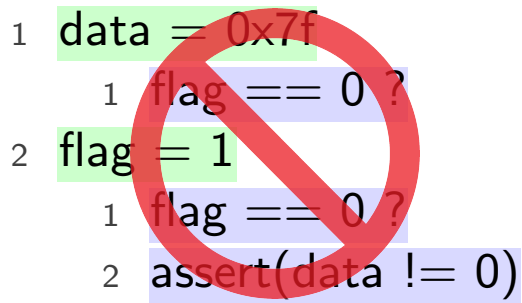
- Fix for the example (on Nvidia GPUs; assuming the producer and consumer are in different blocks):

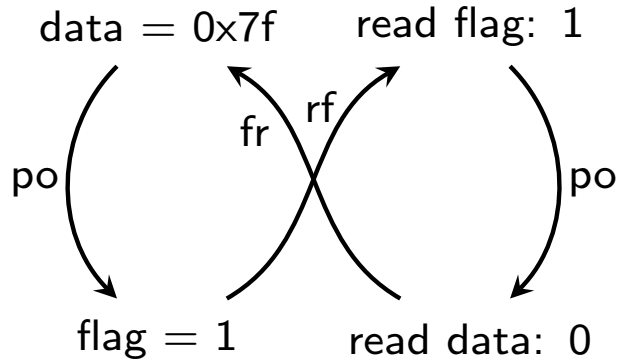
```
1  // Producer
2  data = 0x7f
3  asm ("membar.gl")
4  flag = 1
```

```
1  // Consumer
2  while (flag == 0) {}
3  asm ("membar.gl")
4  assert(data != 0)
```

Axiomatic Memory Models

- Executions are not represented as interleavings, but as execution graphs:





- An execution graph is acyclic if and only if it corresponds to an interleaving
- Axiomatic memory models: Give a set of formal rules defining which executions are possible on a certain architecture
- Full details:
  - Herding Cats. Alglave et al. TOPLAS '14

Testing GPU Memory Models

Weak Memory Models

Which behaviors can be observed when threads concurrently access shared memory?

- As we've seen, we cannot expect sequential consistency (interleaved execution) on GPUs
- But what exactly can we expect?
  - Consult the manual: prose, ambiguous, little detail, sometimes plain wrong
  - We want a formal memory model!

Weak Memory Models

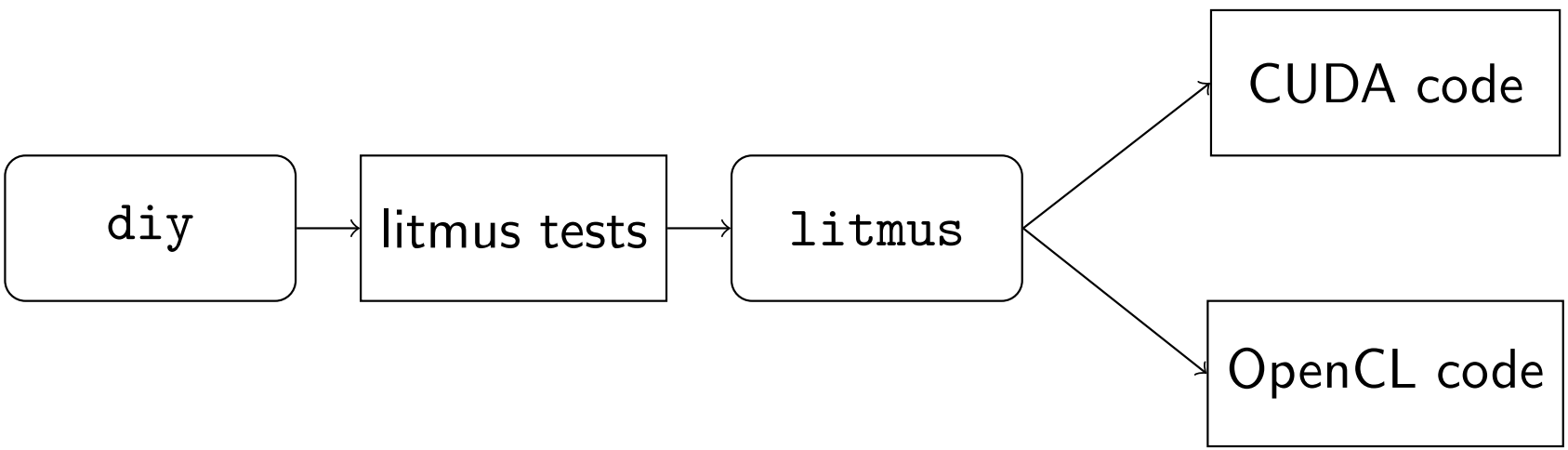
Which behaviors can be observed when threads concurrently access shared memory?

- As we've seen, we cannot expect sequential consistency (interleaved execution) on GPUs
- But what exactly can we expect?
  - Consult the manual: prose, ambiguous, little detail, sometimes plain wrong
  - We want a formal memory model!

- Formal memory model based on:
  - Vendor documentation
  - Testing
  - Discussion with industry contacts

Test Framework

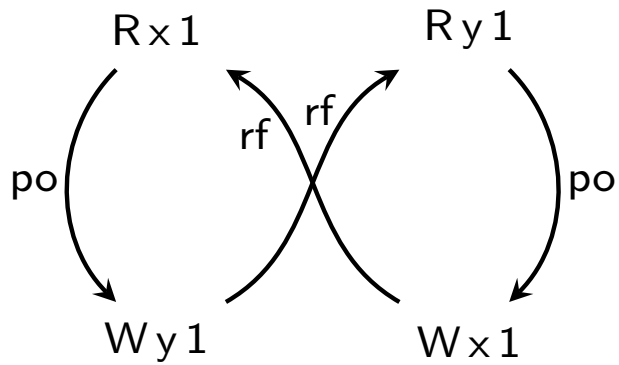
- We extended the `diy` and `litmus` tools to generate and run GPU litmus tests
- `diy` to generate tests
  - Short assembly code snippets called litmus tests
  - Test generation based on an axiomatic modeling framework
- `litmus` to run tests
  - Runs tests produced by `diy` many times
  - Adds additional code to create noise ("incantations") to make weak behaviors appear



Test Generation

`diy`

- Executions are represented as directed graphs
- Non-SC executions have cycles

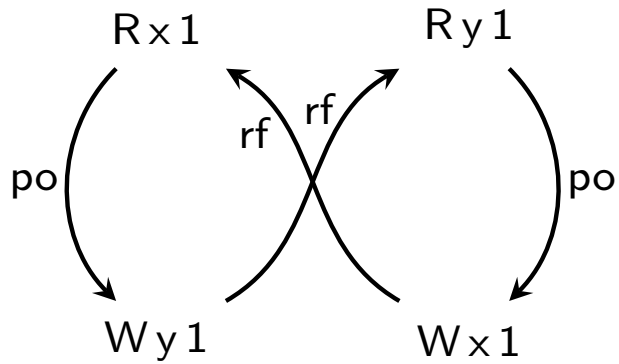


- Which non-SC executions are possible on a certain chip?

Test Generation

`diy`

- Executions are represented as directed graphs
- Non-SC executions have cycles



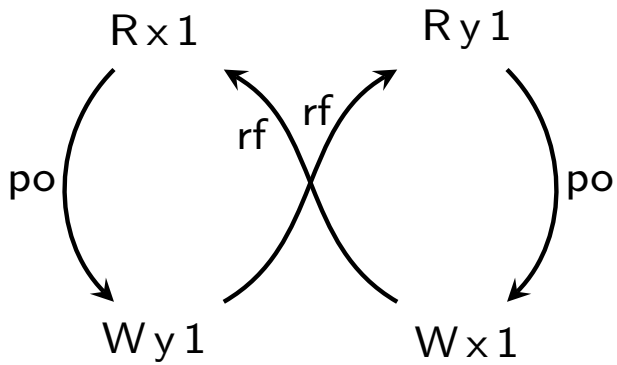
- Which non-SC executions are possible on a certain chip?

- Key idea of `diy`:
  - Enumerate non-SC executions (i. e. cyclic execution graphs)
  - From each such graph, generate a test such that one of its executions is the execution from which it was generated

Test Generation

Example

- Execution graph:



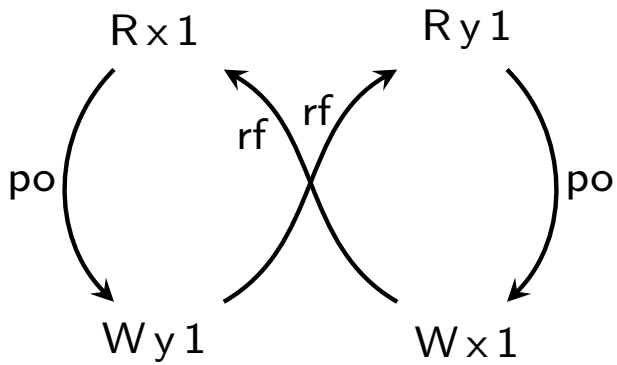
- Generated litmus test:

P_0	P_1

Test Generation

Example

- Execution graph:



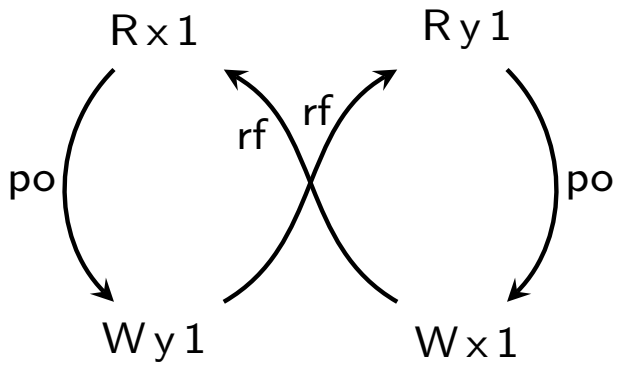
- Generated litmus test:

P_0	P_1
r1 = x	

Test Generation

Example

- Execution graph:



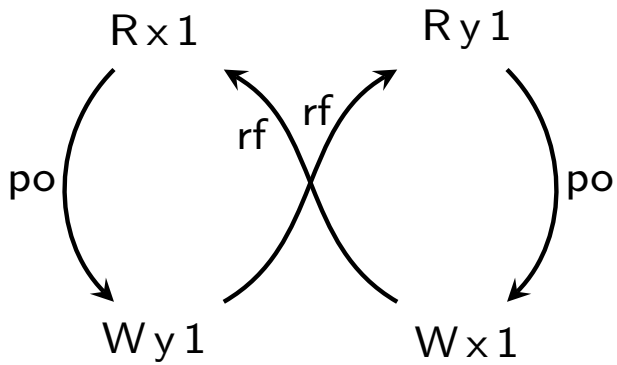
- Generated litmus test:

P_0	P_1
r1 = x y = 1	

Test Generation

Example

- Execution graph:



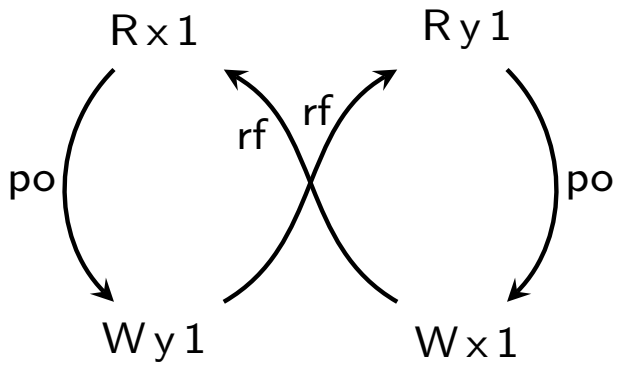
- Generated litmus test:

P_0	P_1
r1 = x y = 1	r2 = y

Test Generation

Example

- Execution graph:



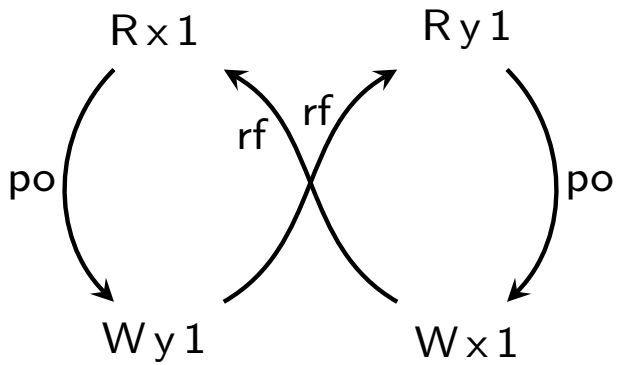
- Generated litmus test:

P_0	P_1
r1 = x	r2 = y
y = 1	x = 1

Test Generation

Example

- Execution graph:



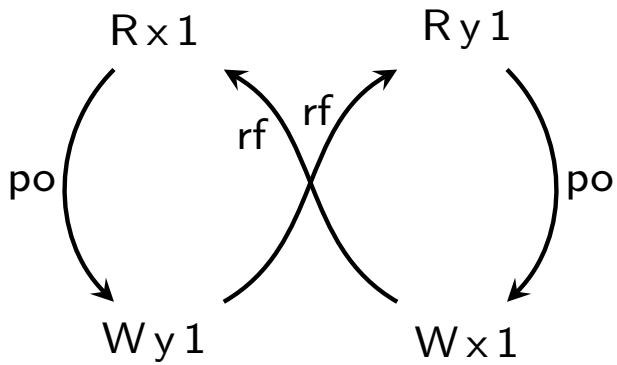
- Generated litmus test:

P_0	P_1
r1 = x	r2 = y
y = 1	x = 1
r1 == 1	

Test Generation

Example

- Execution graph:



- Generated litmus test:

P_0	P_1
r1 = x	r2 = y
y = 1	x = 1
r1 == 1 $\wedge$ r2 == 1	

Running Tests

`litmus`

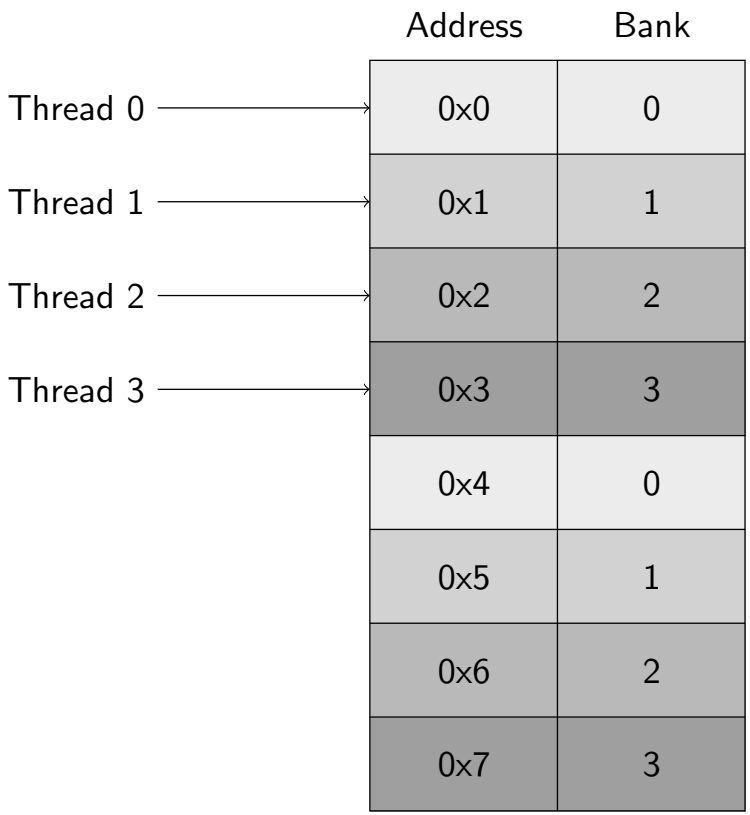
- Now that we can generate tests, we want to run them on the hardware!
- To make the weak behaviors appear, we need "incantations":
  - Put variables on different cache lines
  - Noise maker threads that write random memory locations
  - Random launch parameters
  - Trigger bank conflicts
  - ...

Running Tests

Bank Conflicts

- Memory is divided into banks
- Banks are interleaved, not contiguous
- Accesses to the same bank are serialized

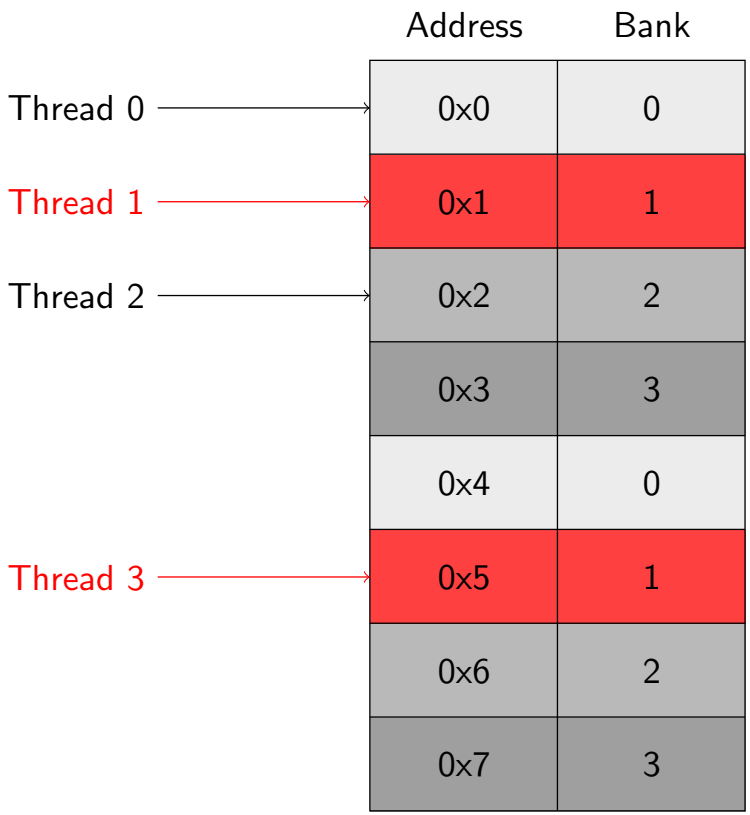
- No bank conflict:



Running Tests

Bank Conflicts

- Memory is divided into banks
- Banks are interleaved, not contiguous
- Accesses to the same bank are serialized
- Bank conflict:



Running Tests

Bank Conflicts

- Accesses to the same bank are serialized
- Accesses can be delayed due to a bank conflict

P_0	P_1
x = 1	r3 = y
y = 1	r4 = x
r3 == 1 $\wedge$ r4 == 0	

- If x = 1 has a bank conflict, it may be delayed leading to y = 1 being executed first
- The order in which accesses to the same bank are serialized is unspecified

Test Results

Read-Read-Coherence

- Consider the following test, with P_0 and P_1 in different blocks, and initially $x = 0$:

P_0	P_1
$x = 1$	$r1 = x$
	$r2 = x$
$r1 == 1 \wedge r2 == 0$	

Read-Read-Coherence

- Consider the following test, with P_0 and P_1 in different blocks, and initially x = 0:

P_0	P_1
x = 1	r1 = x
	r2 = x
r1 == 1 $\wedge$ r2 == 0	

- Running this test 100,000 times with `litmus` on the GeForce GTX 660 yields the following histogram:

```
Test CoRR Allowed
Histogram (4 states)
59875 :>1:r0=0; 1:r2=0;
828   *>1:r0=1; 1:r2=0;
2422  :>1:r0=0; 1:r2=1;
36875 :>1:r0=1; 1:r2=1;
```

Read-Read-Coherence

- Consider the following test, with P_0 and P_1 in different blocks, and initially x = 0:

P_0	P_1
x = 1	r1 = x
	r2 = x
r1 == 1 $\wedge$ r2 == 0	

- Behavior is considered a bug:
  - Does not guarantee what is typically required by programming language standards (OpenCL, C++11)
  - OpenCL and C++11 require that there is a total order on all writes to a memory location (coherence order)
  - No thread shall read values that contradict this order
- Bug occured in all Nvidia chips of the Fermi and Kepler generations we tested
- Fixed in the new Maxwell architecture

Read-Read-Coherence

- Consider the following test, with P_0 and P_1 in different blocks, and initially x = 0:

P_0	P_1
x = 1	r1 = x
	r2 = x
r1 == 1 $\wedge$ r2 == 0	

- GPUs are fairly deterministic (compared to CPUs)
- By fixing the random test parameters, we can make the bug deterministically show up (on Fermi and Kepler GPUs):

```
Test CoRR Allowed
Histogram (1 state)
100000    *>1:r0=1; 1:r2=0;
```

Compare-and-swap

Mutex idiom

- Consider the following test, with P_0 and P_1 in different blocks, and initially x = 0 and mutex = 1:

P_0	P_1
x = 1 membar.gl mutex = 0	b = cas(&mutex, 0, 1) r = x
b == true $\wedge$ r == 0	

- P_0: Write data to x, then unlock the mutex
- P_1: Attempt to lock the mutex; read x if successful
- Can P_1 read a stale value from x when the CAS succeeds?

Compare-and-swap

Mutex idiom

- Consider the following test, with P_0 and P_1 in different blocks, and initially x = 0 and mutex = 1:

P_0	P_1
x = 1 membar.gl mutex = 0	b = cas(&mutex, 0, 1) r = x
b == true $\wedge$ r == 0	

- P_0: Write data to x, then unlock the mutex
- P_1: Attempt to lock the mutex; read x if successful
- Can P_1 read a stale value from x when the CAS succeeds?
  - Yes! (on Fermi and Kepler)
  - CAS does not imply a memory fence on Nvidia GPUs
  - Several papers assume this and are thus wrong (among them the textbook *CUDA by Example*)

Summary

- `diy` to generate GPU litmus tests
- `litmus` to run GPU litmus tests
- Testing the hardware is a necessary first step towards building a formal memory model:
  - Documentation is insufficient:
    - ambiguous
    - little detail
    - sometimes wrong
  - Side effect: We find bugs in the hardware
  - Test results serve as a basis for communication with industry contacts

Thank you!