École de Recherche:
Semantics and Tools for Low-Level Concurrent Programming
ENS Lyon

Formal Verification Techniques for GPU Kernels

Lecture 1

Alastair Donaldson
Imperial College London
www.doc.ic.ac.uk/~afd
afd@imperial.ac.uk

The challenges of concurrency

Let’s consider a simple concurrent program with three threads:

A **data stream** thread which repeatedly writes random values to a memory location

A **sampler** thread which repeatedly reads from this memory location, printing the value that was read to the console

A **main** thread which launches the data stream and sampler threads

```c
#include <stdio.h>
#include <pthread.h>

volatile int* volatile p = NULL;

void* dataStream(void* unused) {
  usleep(1);
  p = (int*)malloc(sizeof(int));
  while(1) {
    *p = rand();
  }
}

void* sampler(void* unused) {
  while(1) {
    printf("%d\n", *p);
  }
}
```

volatile necessary to tell the compiler that **p**, and/or its contents, are subject to change by other threads

What can go wrong with this program?

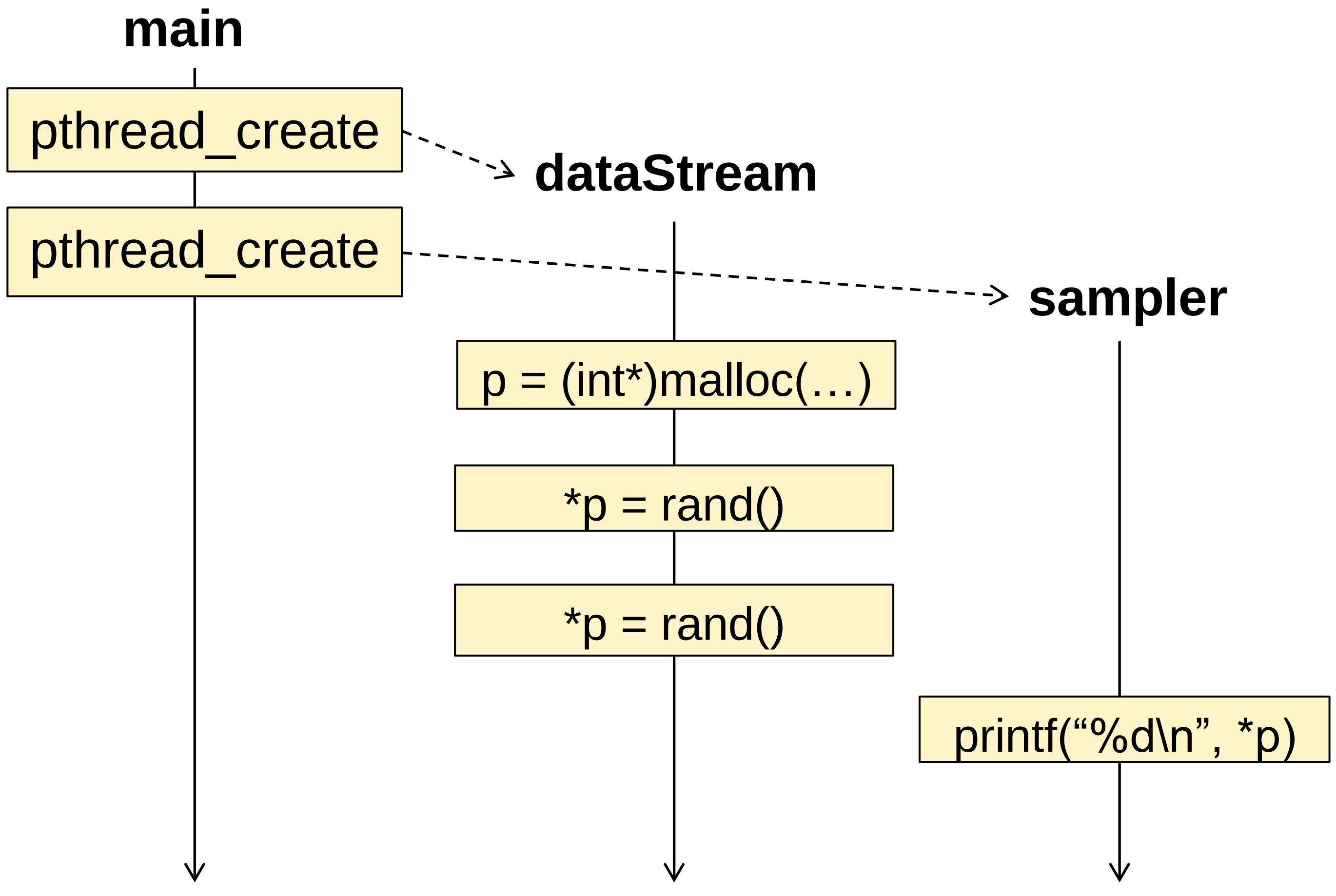
Possible for **sampler** to begin sampling before **p** has been allocated

Big problem: the bug does not always manifest! (Adding **usleep**(…) helps to expose it for illustration purposes)

```c
int main() {
  pthread_t dataStreamHandle;
  pthread_t samplerHandle;
  pthread_create(&dataStreamHandle, NULL, dataStream, NULL);
  pthread_create(&samplerHandle, NULL, sampler, NULL);
  pthread_join(dataStreamHandle, NULL);
  pthread_join(samplerHandle, NULL);
}
```

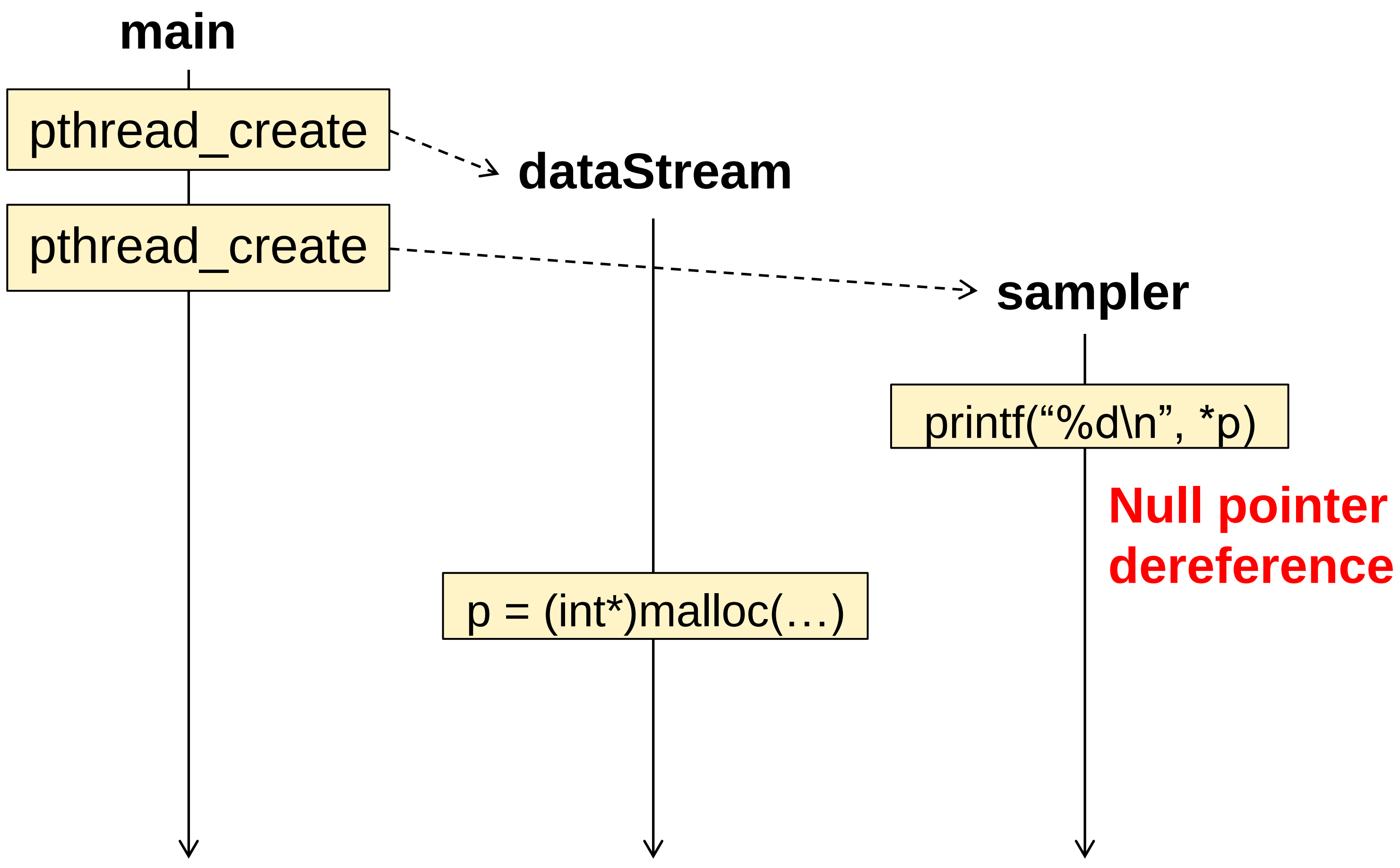
Buggy behaviour depends on schedule

Program will not crash if malloc in **dataStream** happens before any `printf("%d\n", *p)` in **sampler**



Buggy behaviour depends on schedule

…but if **`printf("%d\n", *p)`** the program dereferences a null pointer



Try it for yourself

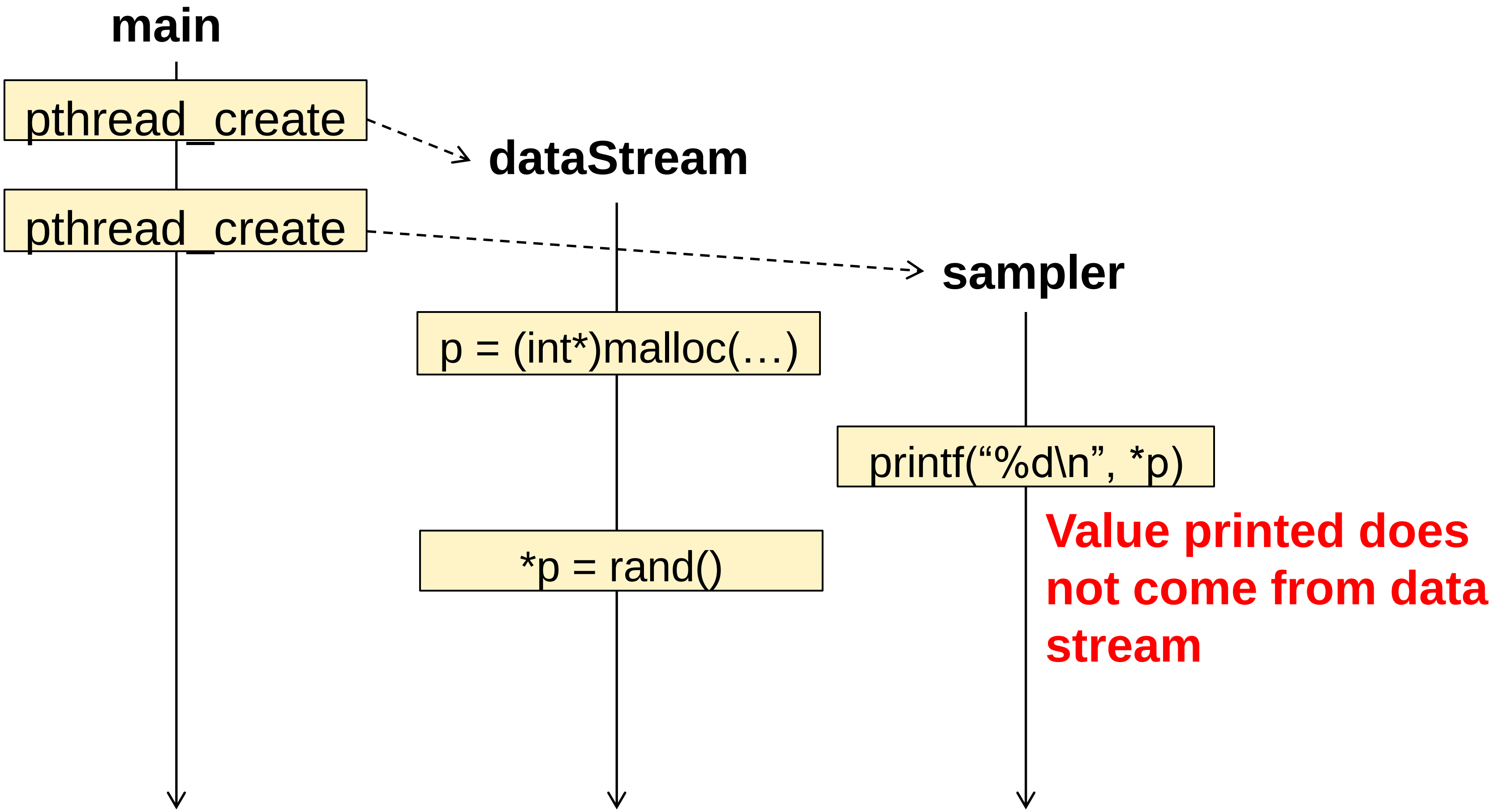
Compile, then run many times:

```
$ gcc -o main test.c
$ ./main
Segmentation fault (core dumped)
$ ./main
1270744533
1670651648
364481212
…
$ ./main
Segmentation fault (core dumped)
```

etc.

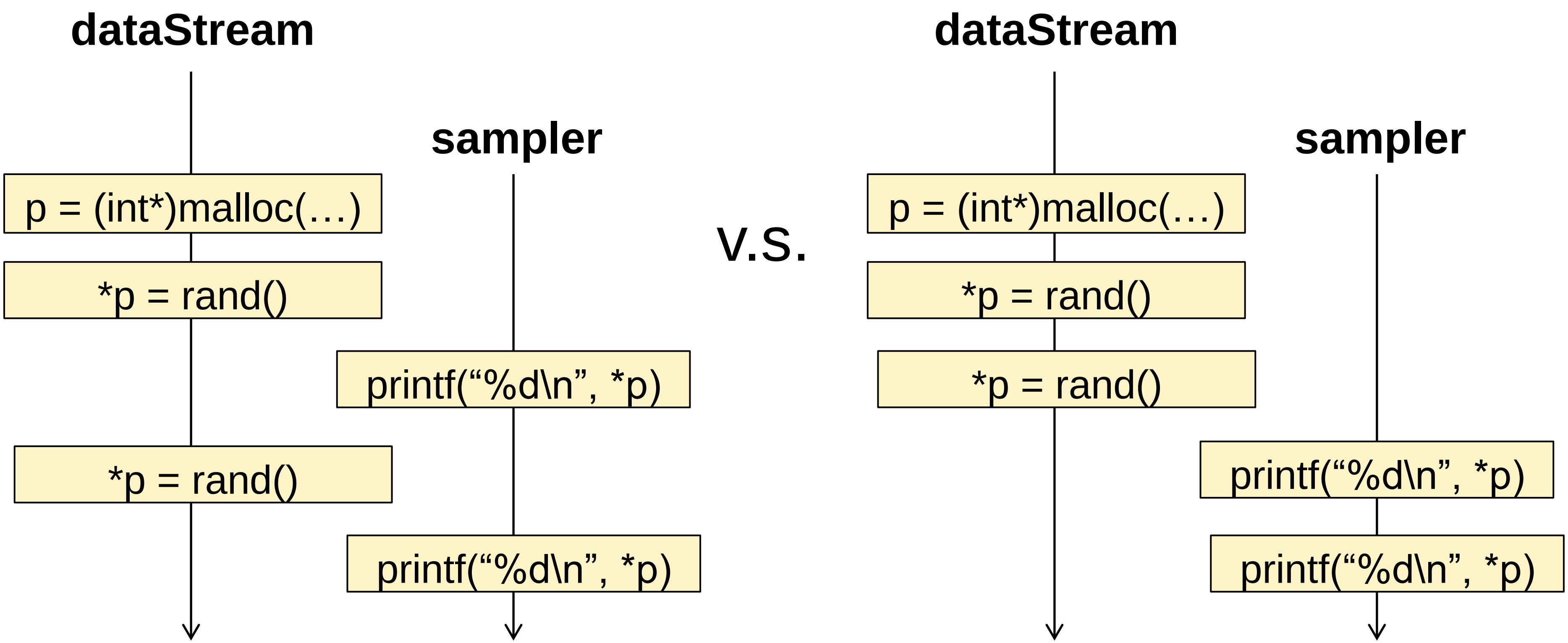
Other issues related to the example:

Sampler may read from data stream after allocation but before a data value is written – **data race**:



Other issues related to the example:

Which values are sampled depends on thread schedule:



…thus there are further data races on ***p**, but since sampling is random anyway these are **benign** – they do not matter

Analysing concurrent programs: challenge

Something simple like:

```
x = 0;
x = x + 1;
assert(x == 1);
```

may not be correct if **x** can be modified by other threads!

Analysing concurrent programs: solutions

Four main approaches have been explored by researchers:

Stress testing

Run concurrent program 100s of times, injecting **random delays** so that a variety of schedules are exhibited

Schedule enumeration

Run concurrent program using a **controlled scheduler**

Systematically explore all possible schedules

Static concurrent program verification

Extend program verification techniques (pre- and post-conditions, loop invariants) to take account of concurrency

Reducing concurrent program verification to sequential program verification

Transform concurrent program into semantically equivalent **sequential program**, then apply existing techniques

We will focus on:

Reducing concurrent program verification to sequential program verification

Allows re-use of techniques for sequential program analysis which are **well understood**

Only works in restricted circumstances

We shall study an approach for reducing verification of concurrent programs to a sequential program verification task for a certain class of software: **GPU kernels**

Graphics processing units (GPUs)

Originally designed to accelerate graphics processing

GPU has many parallel processing elements: graphics operations on sets of pixels are **inherently parallel**

Early GPUs: limited functionality, tailored specifically towards graphics

Recently GPUs have become more powerful and general purpose. Widely used in parallel programming to accelerate tasks including:

Medical imaging	**Financial simulation**	**Computer vision**
Computational fluid dynamics	**DNA sequence alignment**	…and many more

Graphics processing units (GPUs)

Many PEs

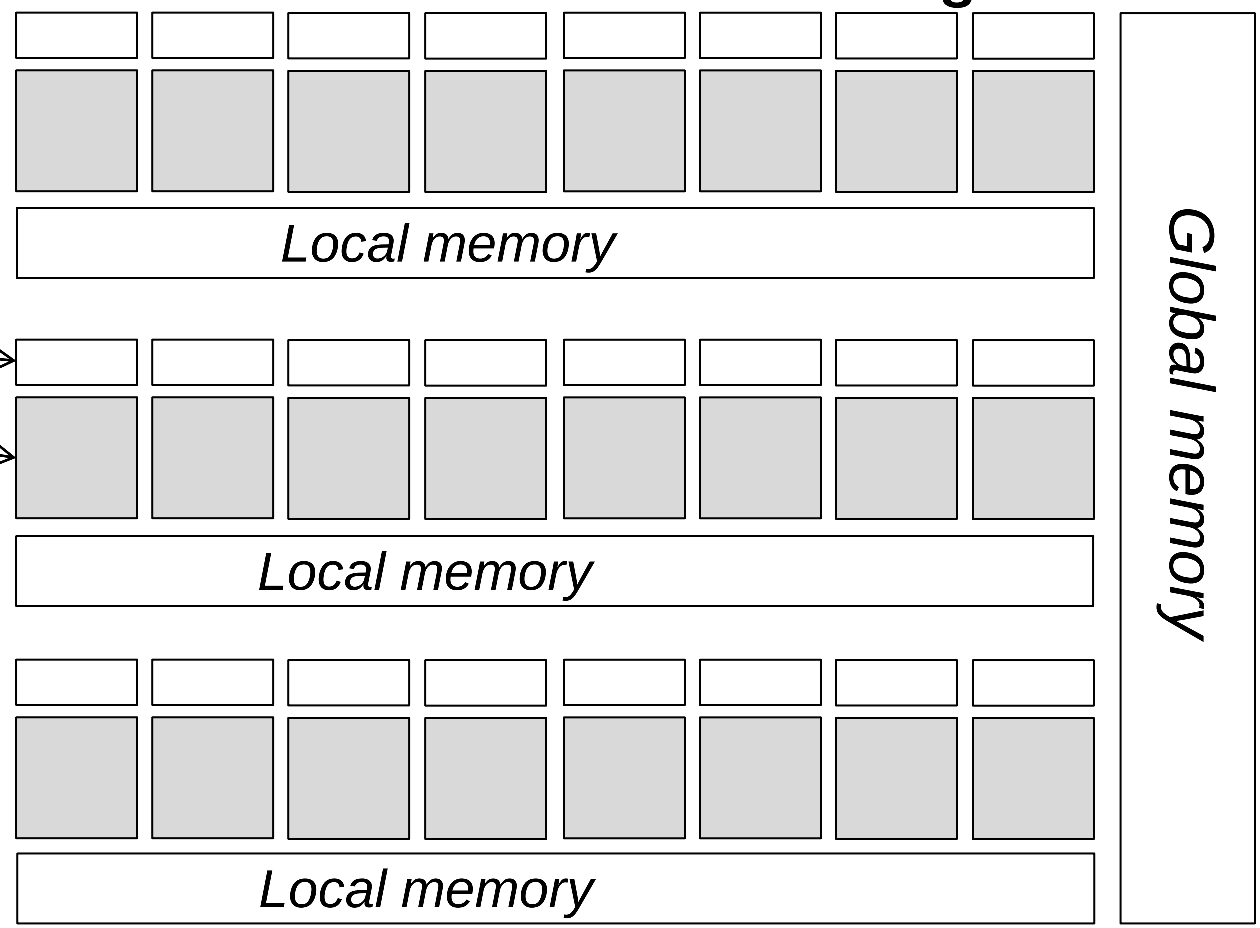
All PEs share global memory

Organised into groups

Private memory

Processing element (PE)

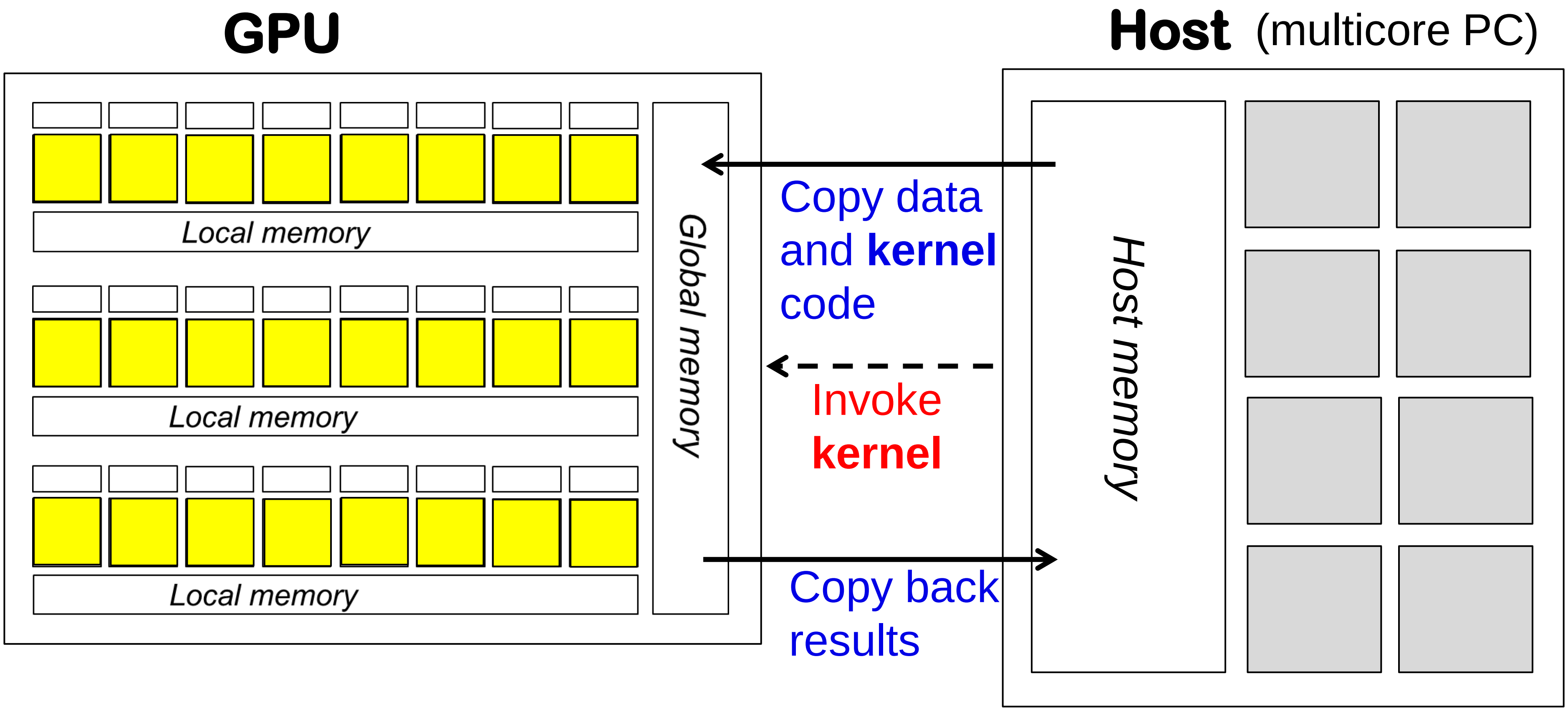
PEs in same group share memory



GPU-accelerated systems

Host PC copies data and code into GPU memory

Code is a **kernel** function which is executed by each PE



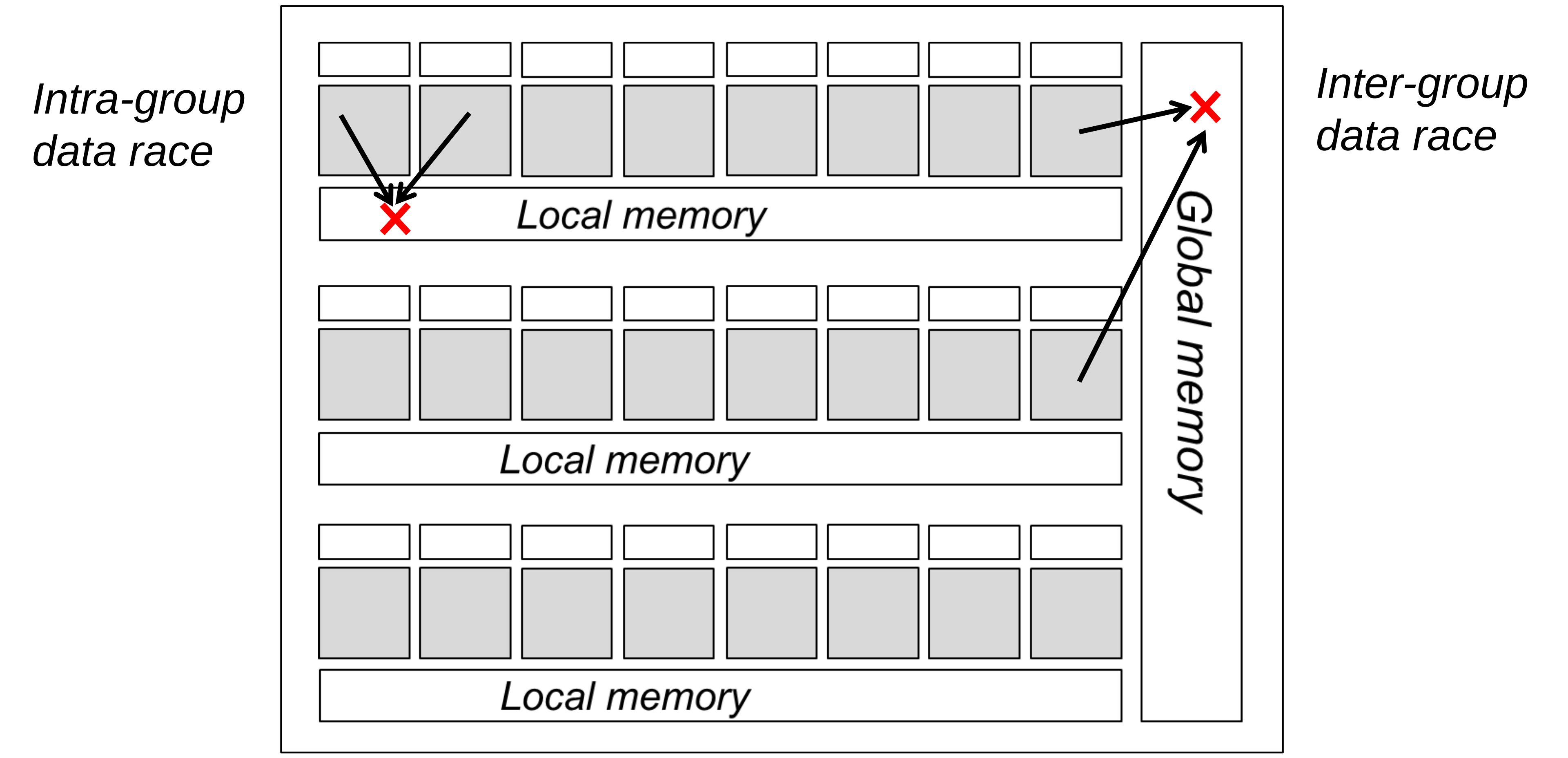
Data races in GPU kernels

A **data race** occurs if:

- two **distinct** threads access the **same** memory location
- at least one of the accesses is a **write**
- the accesses are **not** separated by a barrier synchronisation operation

More on this later

Data races in GPU kernels



Lead to **all kinds of problems**!

Almost always **accidental** and **unwanted**: data races in GPU kernels are not usually benign

Data races in GPU kernels

We shall look at a technique for analysing whether a GPU kernel can exhibit data races

We shall restrict attention to **intra-group** data races

Henceforth, let’s assume that all threads are in the same group

GPU kernel example

Indicates that function is the kernel's entry point

Indicates that **A** is an array stored in group's local memory

```
__kernel void
add_neighbour(__local int* A, int offset) {
  A[tid] = A[tid] + A[tid + offset];
}
```

Read/write data race

Built-in variable which contains thread's id

Syntax used here is (more or less) OpenCL, an industry standard for multicore computing

All threads execute **add_neighbour** – host specifies how many threads should run

Illustration of data race

```
__kernel void
add_neighbour(__local int* A, int offset) {
  A[tid] = A[tid] + A[tid + offset];
}
```

Suppose **offset** == 1

Thread 0: **reads** from **A**[**tid** + **offset**], i.e., **A**[1]

Thread 1: **writes** to **A**[**tid**], i.e., **A**[1]

Similar data races possible between other pairs of adjacent threads

No guarantee about the **order** in which these accesses will occur

Illustrating the effects of a data race

Suppose:

- **offset** == 1
- **A** initially contains { 1, 1, 1, 1, 1 }
- there are four threads

Let’s see how A evolves for two particular schedules

$$\{1, 1, 1, 1, 1\} \xrightarrow{\textit{thread 0}} \{2, 1, 1, 1, 1\} \xrightarrow{\textit{thread 1}} \{2, 2, 1, 1, 1\} \xrightarrow{\textit{thread 2}} \{2, 2, 2, 1, 1\} \xrightarrow{\textit{thread 3}} \{2, 2, 2, 2, 1\}$$

$$\{1, 1, 1, 1, 1\} \xrightarrow{\textit{thread 3}} \{1, 1, 1, 2, 1\} \xrightarrow{\textit{thread 2}} \{1, 1, 3, 2, 1\} \xrightarrow{\textit{thread 1}} \{1, 4, 3, 2, 1\} \xrightarrow{\textit{thread 0}} \{5, 4, 3, 2, 1\}$$

Completely different results!

Barrier synchronisation

`barrier()`

Related to, but **different** from memory barrier in CPU instruction set

Used to **synchronise** threads

When a thread reaches **barrier()** it waits until **all** threads reach the barrier

Note: all threads must reach the same barrier – illegal for threads to reach different barriers

When all threads have reached the barrier, the threads can proceed past the barrier

Reads and writes before the barrier are **guaranteed to have completed** after the barrier

Using barrier to avoid a data race

```
__kernel void
add_neighbour(__local int* A, int offset) {
  int temp = A[tid + offset];
  barrier();
  A[tid] = A[tid] + temp;
}
```

Accesss cannot be concurrent

Focussing data race analysis

All threads are always executing in a region **between two barriers**:

```
...
barrier();
```

Barrier-free code region

Race may be due to two threads executing statements **within** the region

We **cannot** have a race caused by a statement in the region and a statement **outside** the region

```
barrier();
...
```

Data race analysis can be localised to focus on **regions between barriers**

Reducing thread schedules

With ***n*** threads, roughly how many possible thread schedules are there between these barriers, assuming each statement is atomic?

```
barrier();
    S1;
    S2;
    ...
    Sk
barrier();
```

Total execution length is $n \times k$

Thread 1 executes k statements:

$\binom{n \times k}{k}$ choices for these

Thread 2 executes k statements:

$\binom{(n-1) \times k}{k}$ choices for these

etc.

Number of possible schedules: in the order of n^k

Reducing thread schedules

Do we really need to consider **all** of these schedules to detect data races?

No: actually is suffices to consider **just one schedule**, and it can be **any schedule**

Any schedule will do! For example:

```
barrier(); // A
```

Run thread 0 from A to B
Log all accesses

Run thread 1 from A to B
Log all accesses
Check against thread 0

Run thread 2 from A to B
Log all accesses
Check against threads 0 and 1

. . .

Run thread *N-1* from A to B
Log all accesses
Check against threads 0..*N*-2

```
barrier(); // B
```

If data race exists it will be detected: **abort**

No data races: chosen schedule **equivalent to all others**

Abort on race

Completely avoids reasoning about interleavings!

Reducing thread schedules

Because we can choose a **single** thread schedule, we can view a barrier region containing **k** statements as a sequential program containing **n** $\times$ **k** statements

This is good: it means we are back in the world of sequential program analysis

But in practice it is quite normal for a GPU kernel to be executed by e.g. 1024 threads

Leads to an **intractably large** sequential program

Can we do better?

Yes: just two threads will do!

Choose *arbitrary* ***i***, ***j*** ∈ { 0, 1, …, *N*-1 } with ***i*** ≠ ***j***

```
barrier(); // A
```

Run thread ***i*** from A to B
Log all accesses

Run thread ***j*** from A to B
Check all accesses against thread ***i***
Abort on race

```
barrier(); // B
```

If data race exists it will be exposed for ***some*** choice of ***i*** and ***j***. If we can prove data race freedom for arbitrary ***i*** and ***j*** then the region must be data race free

Is this sound?

havoc(**x**) means "set **x** to an *arbitrary* value"

```
barrier(); // A
```

Run thread ***i*** from A to B
Log all accesses
Run thread ***j*** from A to B
Check all accesses against thread ***i***
Abort on race

```
barrier(); // B
```

Run thread ***i*** from B to C
Log all accesses
Run thread ***j*** from B to C
Check all accesses against thread ***i***
Abort on race

```
barrier(); // C
```

No: it is as if only ***i*** and ***j*** exist, and other threads have no effect!

Solution: make shared state **abstract**

- simple idea: **havoc** the shared state at each barrier
- even simpler: remove shared state completely

GPUVerify technique and tool

Exploit:

	any schedule will do
+	two threads will do
+	shared state abstraction

to compile **massively parallel** kernel **K** into **sequential** program **P** such that (roughly):

P correct (no assertion failures) **=>** **K** free from **data races**

Next: technical details of how this works

Demo of GPUVerify

Also try it yourself:

http://multicore.doc.ic.ac.uk/tools/GPUVerify

Data race analysis for straight line kernels

Assume kernel has form:

```
__kernel void foo( <parameters, including __local arrays> ) {
   <local variable declarations>
   S1;
   S2;
   ...
   Sk;
}
```

where each statement $\mathbf{S}_i$ has one of the following forms:

x = **e**
x = **A**[**e**]
A[**e**] = **x**
barrier()

where:

- **x** denotes a local variable
- **e** denotes an expression over local variables
- **A** denotes a `__local` array parameter

Data race analysis for straight line kernels

Restricting statements to these forms:

x = **e**
x = **A**[**e**]
A[**e**] = **x**
barrier()

where:

- **x** denotes a local variable
- **e** denotes an expression over local variables and **tid**
- **A** denotes a `__local` array parameter

means:

Easy to enforce by pre-processing the code

- A statement involves **at most one** load from / stores to local memory
- There is no conditional or loop code

We will drop this restriction later

Our aim

We want to translate kernel into sequential program that:

- Models execution of two arbitrary threads using some fixed schedule
- Detects data races
- Treats shared state abstractly

Call original GPU kernel **K**

Call resulting sequential program **P**

Introducing two arbitrary threads

K has implicit variable **tid** which gives the id of a thread

Suppose **N** is the total number of threads

In **P**, introduce two global variables:

```
int tid$1;
int tid$2;
```

and preconditions:

```
\requires 0 <= tid$1 && tid$1 < N;
\requires 0 <= tid$2 && tid$2 < N;
\requires tid$1 != tid$2;
```

Threads must both be in range

Threads must be different

…but otherwise the threads are arbitrary

Race checking instrumentation

For each `__local` array parameter **A** in **K** introduce four global variables:

```
bool READ_HAS_OCCURRED_A;
bool WRITE_HAS_OCCURRED_A;
int READ_OFFSET_A;
int WRITE_OFFSET_A;
```

We shall shortly discuss the purpose of these

and four procedures:

```
void LOG_READ_A(int offset);

void LOG_WRITE_A(int offset);

void CHECK_READ_A(int offset);

void CHECK_WRITE_A(int offset);
```

We shall shortly discuss the implementation of these

Get rid of parameter **A** in **P**

Example illustrating concepts so far:

Form of **K**

```
__kernel void foo(__local int* A, __local int* B,
                  int idx) {
  ...
}
```

Form of **P**

```
int tid$1;
int tid$2;

bool READ_HAS_OCCURRED_A;     bool READ_HAS_OCCURRED_B;
bool WRITE_HAS_OCCURRED_A;    bool WRITE_HAS_OCCURRED_B;
int READ_OFFSET_A;            int READ_OFFSET_B;
int WRITE_OFFSET_A;           int WRITE_OFFSET_B;

// \requires 0 <= tid$1 && tid$1 < N;
// \requires 0 <= tid$2 && tid$2 < N;
// \requires tid$1 != tid$2;
void foo(int idx) {
  ...
}
```

Ids for two threads

Instrumentation variables for **A**

Instrumentation variables for **B**

Constraining **tid$1** and **tid$2** to be arbitrary, distinct threads

A and **B** have gone

Duplicating local variable declarations

Local variable declaration:

`int x` duplicated to become:

```
int x$1;
int x$2;
```

Reflects fact that each thread has a copy of **x**

Non-array parameter declaration duplicated similarly.

Non-array parameter **x** initially assumed to be equal between threads: **\requires x$1** == **x$2**

Notation: for an expression **e** over local variables and **tid** we use **e$1** to denote **e** with every occurrence of a variable **x** replaced with **x$1**

e$2 is similar

E.g., if **e** is **a** + **tid** - **x** **e$2** is **a$2** + **tid$2** - **x$2**

Translating statements of K

Encode the statements of **K** for both threads using **round-robin schedule** for the two threads being modelled

Stmt	translate(**Stmt**)
`x = e;`	`x$1 = e$1;` `x$2 = e$2;`

`x = A[e];`	`LOG_READ_A(e$1);`
	`CHECK_READ_A(e$2);`
	`havoc(x$1);`
	`havoc(x$2);`

Log location from which first thread reads

Check read by second thread does not conflict with **any** prior write by first thread

Over-approximate effect of read by making receiving variables **arbitrary**

We have **removed** array **A**. Thus we over-approximate the effect of reading from **A** using **havoc**. We make no assumptions about what **A** contains

Translating statements of K (continued)

Stmt	translate(**Stmt**)

`A[e] = x;`	`LOG_WRITE_A(e$1);`
	`CHECK_WRITE_A(e$2);`
	`// nothing`

`barrier();`	`barrier();`
S; **T**;	translate(**S**); translate(**T**);

Log location to which first thread writes

Check write by second thread does not conflict with **any** prior read or write by first thread

The write itself has no effect in because the array **A** has been removed

We shall give **barrier()** a special meaning in translated program

Example so far:

```
__kernel void
foo(__local int* A,
              int idx ) {
  int x;
  int y;
  x = A[tid + idx];

  y = A[tid];

  A[tid] = x + y;

}
```

```
// \requires 0 <= tid$1 && tid$1 < N;
// \requires 0 <= tid$2 && tid$2 < N;
// \requires tid$1 != tid$2;
// \requires idx$1 == idx$2;
void foo(
  int idx$1; int idx$2 ) {
  int x$1; int x$2;
  int y$1; int y$2;
  LOG_READ_A(tid$1 + idx$1);
  CHECK_READ_A(tid$2 + idx$2);
  havoc(x$1); havoc(x$2);

  LOG_READ_A(tid$1);
  CHECK_READ_A(tid$2);
  havoc(y$1); havoc(y$2);

  LOG_WRITE_A(tid$1);
  CHECK_WRITE_A(tid$2);
}
```