École de Recherche:

Semantics and Tools for Low-Level Concurrent Programming

ENS Lyon

Formal Verification Techniques for GPU Kernels

Lecture 2

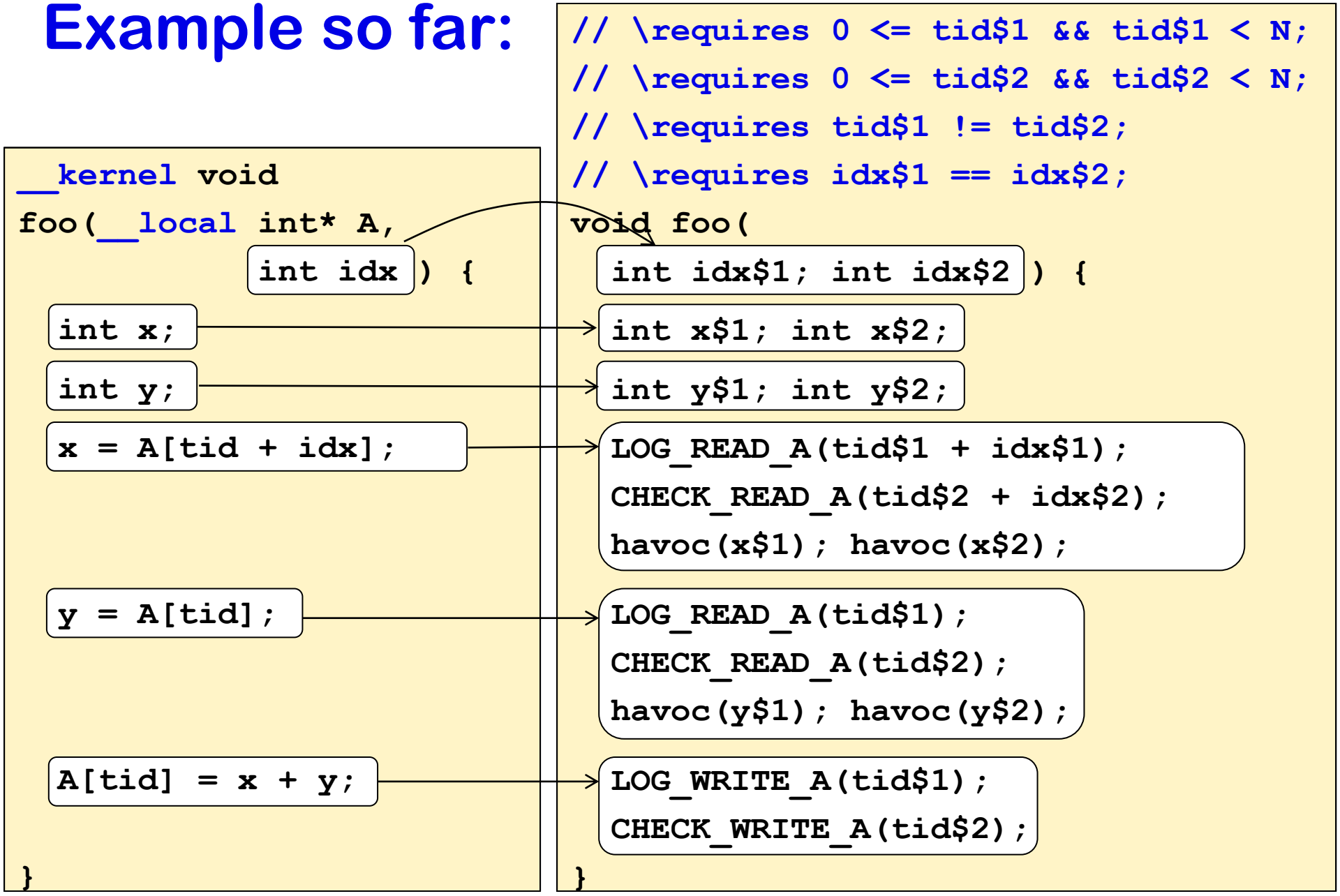
Alastair Donaldson

Imperial College London

www.doc.ic.ac.uk/~afd

afd@imperial.ac.uk

Example so far:



Implementing LOG_READ_A

Global variables **`READ_HAS_OCCURED_A`** and **`READ_OFFSET_A`** collectively log either **nothing**, or the offset of a **single** read from A by the first thread

If **`READ_HAS_OCCURED_A`** is **false**, no read from **A** by the first thread has been logged. In this case the value of **`READ_OFFSET_A`** is meaningless

If **`READ_HAS_OCCURED_A`** is **true**, a read from **A** by the first thread has been logged, and the offset associated with this read is **`READ_OFFSET_A`**

`WRITE_HAS_OCCURED_A` and **`WRITE_OFFSET_A`** are used similarly

Implementing LOG_READ_A

```
void LOG_READ_A(int offset) {
  if(*) {
    READ_HAS_OCCURRED_A = true;
    READ_OFFSET_A = offset;
  }
}
```

`*` is an expression that evaluates non-deterministically

Non-deterministically choose whether to

- log this read from **A**, in which case existing values of **`READ_HAS_OCCURRED_A`** and **`READ_OFFSET_A`** are **over-written**, or:
- Ignore this read from **A**, in which case **`READ_HAS_OCCURRED_A`** and **`READ_OFFSET_A`** are **left alone**

Implementing LOG_WRITE_A

```
void LOG_WRITE_A(int offset) {
   if(*) {
      WRITE_HAS_OCCURRED_A = true;
      WRITE_OFFSET_A = offset;
   }
}
```

Similar to **LOG_READ_A**

Implementing CHECK_READ_A

```
void CHECK_READ_A(int offset) {
  assert(WRITE_HAS_OCCURRED_A =>
         WRITE_OFFSET_A != offset);
}
```

A read from **A** at **offset** by second thread is OK unless first thread has logged a write to **A** at this offset

Whether first thread has logged a write to **A** is determined by **`WRITE_HAS_OCCURRED_A`**

If **`WRITE_HAS_OCCURRED_A`** is true then **`WRITE_OFFSET_A`** records the offset that was written to. This must be different from **offset**

Implementing CHECK_WRITE_A

```
void CHECK_WRITE_A(int offset) {
  assert(WRITE_HAS_OCCURRED_A =>
         WRITE_OFFSET_A != offset);
  assert(READ_HAS_OCCURRED_A =>
         READ_OFFSET_A != offset);
}
```

This is similar to **`CHECK_READ_A`**, but there is a little more to check:

We must check that write by second thread does not conflict with a write **or** a read by first thread

Initially, no reads or writes are logged

We specify this via the precondition:

```
// \requires !READ_HAS_OCCURRED_A
// \requires !WRITE_HAS_OCCURRED_A
```

for each array **A**

Example including precondition

```
// \requires 0 <= tid$1 && tid$1 < N;
// \requires 0 <= tid$2 && tid$2 < N;
// \requires tid$1 != tid$2;
// \requires idx$1 == idx$2;
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
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);
}
```

Example restricted to LOG and CHECK calls

```
// \requires 0 <= tid$1 && tid$1 < N;
// \requires 0 <= tid$2 && tid$2 < N;
// \requires tid$1 != tid$2;
// \requires idx$1 == idx$2;
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
void foo(int idx$1, int idx$2) {
  LOG_READ_A(tid$1 + idx$1);
  CHECK_READ_A(tid$2 + idx$2);
  LOG_READ_A(tid$1);
  CHECK_READ_A(tid$2);
  LOG_WRITE_A(tid$1);
  CHECK_WRITE_A(tid$2);
}
```

What happens to **x** and **y** is irrelevant in this example. Let's omit these details to really focus on what the **LOG** and **CHECK** calls are doing

Inlining all log and check calls

```
// other preconditions same as before
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
void foo(int idx$1, int idx$2) {
  // LOG_READ_A(tid$1 + idx$1);
  if(*) { READ_HAS_OCCURRED_A = true;
          READ_OFFSET_A = tid$1 + idx$1; }
  // CHECK_READ_A(tid$2 + idx$2);
  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2 + idx$2);
  // LOG_READ_A(tid$1);
  if(*) { READ_HAS_OCCURRED_A = true;
          READ_OFFSET_A = tid$1; }
  // CHECK_READ_A(tid$2);
  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2);
  // LOG_WRITE_A(tid$1);
  if(*) { WRITE_HAS_OCCURRED_A = true;
          WRITE_OFFSET_A = tid$1; }
  // CHECK_WRITE_A(tid$2);
  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2);
  assert(READ_HAS_OCCURRED_A => READ_OFFSET_A != tid$2);
}
```

The non-determinism ensures that some program execution checks every pair of potentially racing operations

Checking read from A[tid$1] against write to A[tid$2]

```
// other preconditions same as before
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
void foo(int idx$1, int idx$2) {

  if(*) { READ_HAS_OCCURRED_A = true;
          READ_OFFSET_A = tid$1 + idx$1; }

  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2 + idx$2);

  if(*) { READ_HAS_OCCURRED_A = true;
          READ_OFFSET_A = tid$1;          }

  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2);

  if(*) { WRITE_HAS_OCCURRED_A = true;
          WRITE_OFFSET_A = tid$1; }

  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2);
  assert(READ_HAS_OCCURRED_A => READ_OFFSET_A != tid$2);
}
```

Possible race checked by choosing to log the read, then executing the assert. In this case there is no race

Checking read from A[tid$1 + idx$1] against write to A[tid$2]

```
// other preconditions same as before
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
void foo(int idx$1, int idx$2) {

  if(*) { READ_HAS_OCCURRED_A = true;
          READ_OFFSET_A = tid$1 + idx$1; }

  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2 + idx$2);

  if(*) { READ_HAS_OCCURRED_A = true;
          READ_OFFSET_A = tid$1;        }

  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2);

  if(*) { WRITE_HAS_OCCURRED_A = true;
          WRITE_OFFSET_A = tid$1; }

  assert(WRITE_HAS_OCCURRED_A => WRITE_OFFSET_A != tid$2);
  assert(READ_HAS_OCCURRED_A => READ_OFFSET_A != tid$2);
}
```

Possible race checked similarly – **in this case a potential race is detected**

Implementing barrier()

```
void barrier() {
  assume(!READ_HAS_OCCURRED_A);
  assume(!WRITE_HAS_OCCURRED_A);
  // Do this for every array
}
```

The **`if(*) { … }`** construction in **`LOG_READ`** and **`LOG_WRITE`** means that there is **one** path along which **nothing** was logged

`barrier()` has the effect of **killing** all paths **except this one**

Informal argument that encoding is sound

If there can be a write-write race on array **A** between threads ***i*** and ***j*** there must two statements:

A[**d**] = **x**; // executed by thread ***i***

… // no barrier()

A[**e**] = **y**; // executed by thread ***j***

such that threads ***i*** and ***j*** evaluate **d** and **e** respectively to the same value

Informal argument that encoding is sound

After insertion of calls to **`LOG/CHECK_WRITE_A`** there exists an execution trace where:

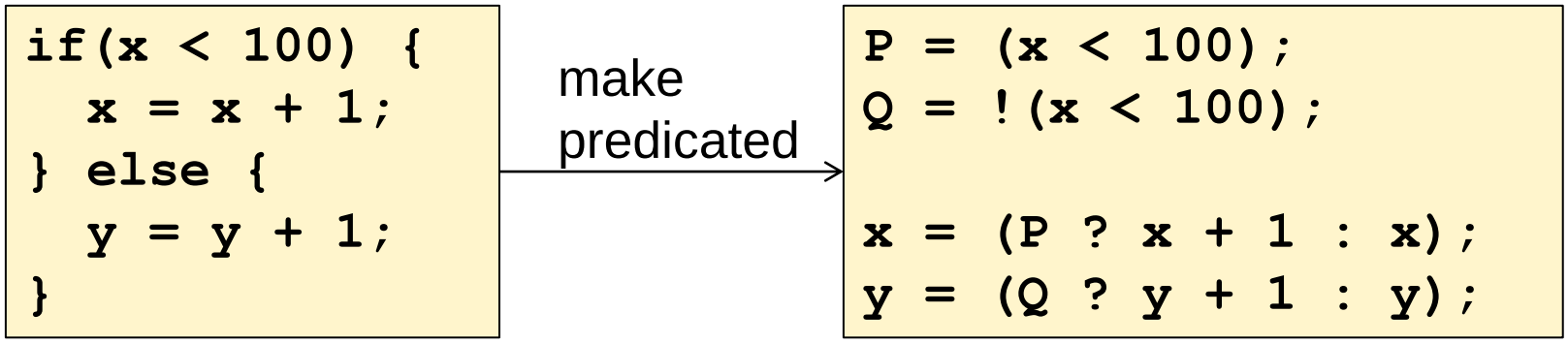
- **tid$1** == ***i***
- **tid$2** == ***j***
- **`LOG_WRITE_A`**(**d$1**) is called and a **non-deterministic** choice is made to track the write to **d$1**
- In subsequent calls to **`LOG_WRITE_A`** a non-deterministic choice is made **not** to track the associated writes (so write to **d$1** is still tracked)
- Assertion in **`CHECK_WRITE_A`**(**e$2**) fails because **e$2** evaluates to the **same** value as **d$1**, which is the value that was logged

Handling loops and conditionals

Use **predicated execution**

Essence of predicated execution: flatten **conditional** code into **straight line code**

Example:



Handling loops and conditionals

Apply **predication** to kernel so that at every execution point there is a predicate determining whether each of the threads are enabled

Add parameters to **`LOG_READ`**/**`WRITE_A`** and **`CHECK_READ`**/**`WRITE_A`** recording whether first or second thread, respectively, is enabled

Translating statements with predicate

We revise the encoding rules to incorporate a **predicate of execution** for each thread; these are initially **true**

Stmt	translate(**Stmt**, **P**)
`x = e;`	`x$1 = P$1 ? e$1 : x$1;` `x$2 = P$2 ? e$2 : x$2;`

`x = A[e];`	`LOG_READ_A(P$1, e$1);`
	`CHECK_READ_A(P$2, e$2);`
	`x$1 = P$1 ? * : x$1;`
	`x$2 = P$2 ? * : x$2;`

`A[e] = x;`	`LOG_WRITE_A(P$1, e$1);` `CHECK_WRITE_A(P$2, e$2);`

LOG and **CHECK** calls take predicate as parameter

We only havoc **x$1** and **x$2** if **P$1** and **P$2**, respectively, are **true**

LOG and **CHECK** calls take predicate as parameter

Translating statements with predicates

The predicates come from conditionals and loops

Stmt	translate(**Stmt**, **P**)

`if(e) {`
`  S;`
`} else {`
`  T;`
`}`
	`Q$1 = P$1 && e$1;`
	`Q$2 = P$2 && e$2;`
	`R$1 = P$1 && !e$1;`
	`R$2 = P$2 && !e$2;`
	translate(**S**, **Q**);
	translate(**T**, **R**);

`while(e) {`
`  S;`
`}`
	`Q$1 = P$1 && e$1;`
	`Q$2 = P$2 && e$2;`
	`while(Q$1 || Q$2) {`
	translate(**S**, **Q**);
	`  Q$1 = Q$1 && e$1;`
	`  Q$2 = Q$2 && e$2;`
	`}`

Q and **R** are fresh

Code for both threads becomes predicated

Threads compute loop guard into predicate

Loop until both threads are done

Translate loop body using loop predicate

Re-evaluate loop guard

Translating statements with predicates

Stmt	translate(**Stmt**, **P**)
S; **T;**	translate(**S**, **P**); translate(**T**, **P**);
`barrier();`	`barrier(P$1, P$2);`

barrier now takes parameters determining whether the threads are **enabled**

Implementing predicated LOG_READ_A

Records whether first thread is enabled

```
void LOG_READ_A(bool enabled, int offset) {
  if(enabled) {
    if(*) {
      READ_HAS_OCCURRED_A = true;
      READ_OFFSET_A = offset;
    }
  }
}
```

If first thread is not enabled it does not execute the read, thus there is nothing to log

LOG_WRITE_A is similar

Implementing predicated CHECK_WRITE_A

Records whether second thread is enabled

```
void CHECK_WRITE_A(bool enabled, int offset) {
  assert(enabled && WRITE_HAS_OCCURRED_A =>
         WRITE_OFFSET_A != off);
  assert(enabled && READ_HAS_OCCURRED_A =>
         READ_OFFSET_A != off);
}
```

If second thread is not enabled it did not execute the write, thus there is nothing to check

CHECK_READ_A is similar

Implementing barrier with predicates

```
void barrier(bool enabled$1, bool enabled$2) {
  assert(enabled$1 == enabled$2);
  if(!enabled$1) {
    return;
  }
  // As before:
  assume(!READ_HAS_OCCURRED_A);
  assume(!WRITE_HAS_OCCURRED_A);
  // Do this for every array
}
```

The threads must agree on whether they are enabled – otherwise we have **barrier divergence**

barrier does nothing if the threads are not enabled

Otherwise it behaves as before

Summary

For each array parameter A:

- Introduce instrumentation variables to log reads from and writes to A
- Generate procedures to log and check reads and writes, using non-determinism to consider all possibilities
- Remove array parameter, and model reads from A using non-determinism

For statements in kernel **K**: generate corresponding statements in sequential program **P**

- Interleave two arbitrary threads using round-robin schedule
- Use predication to handle conditionals and loops

Summary

All together this gives a procedure for turning **K** into a sequential program **P** such that we **almost** have:

P is correct	=>	**K** is free from data races and barrier divergence

Actually we have something **weaker**:

P is correct	=>	All terminating executions of **K** are free from data races and barrier divergence

Exercise: why is this the case?

Worked example using Boogie

Live demo (!!!)

Find out more

Check out: GPUVerify:

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

My web page:

http://www.doc.ic.ac.uk/~afd

My group's page:

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

If you would like to talk about doing a PhD at Imperial, please email me: **afd@imperial.ac.uk**

Bibliography

From my group:

- A. Betts, N. Chong, A. Donaldson, S. Qadeer, P. Thomson, *GPUVerify, a Verifier for GPU Kernels*, OOPSLA 2012
- P. Collingbourne, A. Donaldson, J. Ketema, S. Qadeer, *Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels*, ESOP 2013

University of Utah:

- G. Li, G. Gopalakrishnan, *Scalable SMT-Based Verification of GPU Kernel Functions*, FSE 2010
- G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, S. Rajan, *GKLEE: Concolic Verification and Test Generation for GPUs*, PPoPP 2012

University of California, San Diego

- A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, S. Lerner, *Verifying GPU Kernels by Test Amplification*. PLDI 2012

Thank you for your attention!