Automatic verification of GPU kernels

Lecture by John Wickerson, Imperial College London

Based on work by the GPUVerify team: Adam Betts, Nathan Chong, Peter Collingbourne, Alastair Donaldson, Jeroen Ketema, Egor Kyshtymov, Shaz Qadeer, Paul Thomson
Aims of this lecture

‣ Show a real-world application of Hoare Logic

‣ Introduce GPU programming: data races and barrier divergence

‣ Demonstrate GPUVerify, a tool for statically analysing GPU kernels to check for these kinds of defects

‣ Explain some of the novel verification techniques underlying GPUVerify
GPUs and GPU programming
GPUs

- Many parallel processing elements
- Originally designed to accelerate graphics processing, limited functionality, hard to program
- Recently, more general-purpose functionality. Accelerate such tasks as:
  - Medical imaging
  - Computational fluid dynamics
  - Financial simulation
  - DNA sequence alignment
  - Computer vision
  - ... and many more
GPU Architecture

GPU

Private memory
Processing element (PE)
Local memory
Global memory
Local memory
Local memory

Host (CPU)
Host memory

1. Copy data and kernel code
2. Invoke kernel
3. Copy back results
Data races

- A **data race** occurs when:
  - two **different** threads access the **same** memory location
  - at least one of the accesses is a **write**
  - the accesses are **not** separated by a **barrier**
Data races

Intra-group data race

Inter-group data race
Data races

- A **data race** occurs when:
  - two **different** threads access the **same** memory location
  - at least one of the accesses is a **write**
  - the accesses are **not** separated by a **barrier**

- Data races can cause **undefined behaviour**

- Almost always **accidental** and **unwanted**
Data races

Intra-group data race

Inter-group data race
Data races

Intra-group data race

Local memory
A GPU kernel

This function is the kernel's entry point

The array A is stored in the group's local memory

```
#define tid (get_local_id(0))

__kernel void add_neighbour(__local int* A, int offset) {
}

Identifies the current thread
```
A GPU kernel

```c
__kernel void add_neighbour(__local int* A, int offset) {
}
```

- Suppose `offset = 1`, and that there are four threads

<table>
<thead>
<tr>
<th>Thread:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
</table>

Data race
### Effects of a data race

- Suppose offset $= 1$, and that there are four threads

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Barrier synchronisation

- No thread can proceed beyond a `barrier()` until all threads have reached it.

- Reads and writes from before the barrier are guaranteed to have completed after the barrier.

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

- Threads must reach the same barrier

```
__kernel void foo() {
    if (tid == 0)
        barrier();
    else
        barrier();
}
```
Barrier divergence

- Threads must reach the same barrier
- If the barrier is in a loop, threads must have performed the same number of iterations upon reaching it

```c
__kernel void foo() {
    int i_max = (tid==0 ? 4 : 1);
    int j_max = (tid==0 ? 1 : 4);
    for (int i = 0; i < i_max; i++)
        for (int j = 0; j < j_max; j++)
            barrier();
}
```

**NOT ALLOWED**
The GPUVerify tool
The GPUVerify tool

- A verifier for GPU kernels
- Analyses the source code of OpenCL and CUDA kernels to check for:
  - Intra-group and inter-group data races
  - Barrier divergence
  - Violations of user-specified assertions
- Download from multicore.doc.ic.ac.uk/tools/GPUVerify
- Or try it online at rise4fun.com/GPUVerify-OpenCL
## Parallel reduction

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0+4</td>
<td>1+5</td>
<td>2+6</td>
<td>3+7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0+2</td>
<td>1+3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0+1+2+3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0+4+5+6+7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table illustrates the parallel reduction process, where each element is added to the next in a diagonal fashion. The final result is obtained through successive additions.
How the GPUVerify tool works
Architecture of GPUVerify

- Frontend, built on CLANG/LLVM
- Kernel transformation engine
- Boogie verification engine
- Z3 SMT solver

OpenCL or CUDA kernel → LLVM bytecode → sequential Boogie program → verification conditions
Plan:

Transform massively-parallel kernel $K$ into a sequential program $P$ such that if $P$ is correct then $K$ has no data races and no barrier divergence.
Making the problem tractable

- Data race analysis focuses on each barrier-separated region separately

```c
barrier();
S_0;
S_1;
...
S_{k-1};
barrier();
```
Making the problem tractable

- There are about $N^k$ possible interleavings ... but **any one of them** will do!

Thread 0

```plaintext
barrier();
S_0;
S_1;
...
S_{k-1};
barrier();
```

Thread 1

```plaintext
barrier();
S_0;
S_1;
...
S_{k-1};
barrier();
```

Thread N-1

```plaintext
barrier();
S_0;
S_1;
...
S_{k-1};
barrier();
```
Reduction to two threads

- We can do better still!

- Pick **arbitrary** threads $i$ and $j$ (ensuring $i \neq j$)
Reduction to two threads

- We can do better still!
- Pick *arbitrary* threads $i$ and $j$ (ensuring $i \neq j$)
- Problem: it’s like threads $i$ and $j$ are the *only threads*
- Account for the effects of other threads by randomising the *shared state* at each barrier
Verification technique

Plan:

Transform massively-parallel kernel $K$ into a sequential program $P$ such that if $P$ is correct then $K$ has no data races and no barrier divergence.

Three key observations:
- any schedule will do
- two threads will do
- abstracting the shared state
Details of the two-thread reduction
The two-thread reduction

- Assume kernel has this form:

  ```
  __kernel void foo(<parameters>) {
      <declare local variables>
      S_0; S_1; ...; S_{k-1};
  }
  ```

- where each statement $S_k$ has one of these forms:

  ```
  x = e  
  x = A[e]  
  A[e] = e'
  ```
Our example kernel

Kernel $K$:

```c
__kernel void foo(
    __local int* A,
    __local int* B,
    int idx)
{
    int x, y;
    x = A[tid + idx];
    y = A[tid];
    A[tid] = x + y;
}
```
Picking two arbitrary threads

- Introduce two global variables:

\[
\begin{align*}
\text{var } & \text{tid}\$1 : \text{int}; \\
\text{var } & \text{tid}\$2 : \text{int};
\end{align*}
\]

and assume that they are in-range and different:

\[
\begin{align*}
\text{requires } & 0 \leq \text{tid}\$1 \land \text{tid}\$1 < N; \\
\text{requires } & 0 \leq \text{tid}\$2 \land \text{tid}\$2 < N; \\
\text{requires } & \text{tid}\$1 \neq \text{tid}\$2;
\end{align*}
\]
Logging reads and writes

- Replace each __local array parameter \( A \) with four global variables:

\[
\begin{align*}
\text{var} & \quad \text{READ\_HAS\_OCCURRED\_A} : \text{bool}; \\
\text{var} & \quad \text{WRITE\_HAS\_OCCURRED\_A} : \text{bool}; \\
\text{var} & \quad \text{READ\_OFFSET\_A} : \text{int}; \\
\text{var} & \quad \text{WRITE\_OFFSET\_A} : \text{int};
\end{align*}
\]

- and four procedures:

\[
\begin{align*}
\text{procedure} & \quad \text{LOG\_READ\_A}(\text{offset} : \text{int}); \\
\text{procedure} & \quad \text{LOG\_WRITE\_A}(\text{offset} : \text{int}); \\
\text{procedure} & \quad \text{CHECK\_READ\_A}(\text{offset} : \text{int}); \\
\text{procedure} & \quad \text{CHECK\_WRITE\_A}(\text{offset} : \text{int});
\end{align*}
\]
The transformation so far...

**Kernel \( \mathbf{K} \):**

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

**Sequential program \( \mathbf{P} \):**

```c
var tid$1, tid$2 : int;
var READ_HAS_OCCURRED_A : bool;
var READ_HAS_OCCURRED_B : bool;
var WRITE_HAS_OCCURRED_A : bool;
var WRITE_HAS_OCCURRED_B : bool;
var READ_OFFSET_A, READ_OFFSET_B : int;
var WRITE_OFFSET_A, WRITE_OFFSET_B : int;
procedure foo(idx : int)
    requires 0 <= tid$1 && tid$1 < N;
    requires 0 <= tid$2 && tid$2 < N;
    requires tid$1 != tid$2;
{
    ...
}
```
Duplicating local variables

- Both threads need a copy of each local variable
- E.g. `int x;` becomes `var x$1, x$2 : int;`
- Same goes for non-array parameters
- Note that the values of the parameters are the same across all threads:

```plaintext
requires param$1 == param$2;
```
Translating statements

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\text{translate}(S)$</th>
</tr>
</thead>
</table>
| $x = e;$ | $x$1 := $e$1;  
$x$2 := $e$2; |
| $x = A[e]$; | $\text{call }$ LOG_READ_A($e$1);  
$\text{call }$ CHECK_READ_A($e$2);  
$\text{havoc } x$1, $x$2; |
| $A[e] = e'$; | $\text{call }$ LOG_WRITE_A($e$1);  
$\text{call }$ CHECK_WRITE_A($e$2); |
| barrier(); | $\text{call }$ barrier(); |
| $S_1; S_2;$ | $\text{translate}(S_1); \text{translate}(S_2);$ |
The transformation so far...

Kernel $K$:

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

Sequential program $P$:

```c
var tid$1, tid$2 : int;
var READ_HAS_OCCURRED_A : bool;
var READ_HAS_OCCURRED_B : bool;
var WRITE_HAS_OCCURRED_A : bool;
var WRITE_HAS_OCCURRED_B : bool;
var READ_OFFSET_A, READ_OFFSET_B : int;
var WRITE_OFFSET_A, WRITE_OFFSET_B : int;
procedure foo(idx$1 : int, idx$2 : int)
    requires 0 <= tid$1 && tid$1 < N;
    requires 0 <= tid$2 && tid$2 < N;
    requires tid$1 != tid$2;
    requires idx$1 == idx$2;
{
    var x$1, x$2, y$1, y$2 : int;
    call LOG_READ_A(tid$1 + idx$1);
    call CHECK_READ_A(tid$2 + idx$2);
    havoc x$1, x$2;
    call LOG_READ_A(tid$1);
    call CHECK_READ_A(tid$2);
    havoc y$1, y$2;
    call LOG_WRITE_A(tid$1);
    call CHECK_WRITE_A(tid$2);
}
```
procedure LOG_READ_A(offset : int) {
    if (*) {
        READ_HAS_OCCURRED_A := true;
        READ_OFFSET_A := offset;
    }
}

procedure LOG_WRITE_A(offset : int) {
    if (*) {
        WRITE_HAS_OCCURRED_A := true;
        WRITE_OFFSET_A := offset;
    }
}
The checking functions

procedure CHECK_READ_A(offset : int) {
    assert (WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_A != offset);
}

procedure CHECK_WRITE_A(offset : int) {
    assert (WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_A != offset);
    assert (READ_HAS_OCCURRED_A ==> READ_OFFSET_A != offset);
}
The transformation so far...

Kernel $K$: 

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

Sequential program $P$: 

```c
var tid$1, tid$2 : int;
var READ_HAS_OCCURRED_A : bool;
var READ_HAS_OCCURRED_B : bool;
var WRITE_HAS_OCCURRED_A : bool;
var WRITE_HAS_OCCURRED_B : bool;
var READ_OFFSET_A, READ_OFFSET_B : int;
var WRITE_OFFSET_A, WRITE_OFFSET_B : int;
procedure foo(idx$1 : int, idx$2 : int);
    requires 0 <= tid$1 && tid$1 < N;
    requires 0 <= tid$2 && tid$2 < N;
    requires tid$1 != tid$2;
    requires !READ_HAS_OCCURRED_A;
    requires !WRITE_HAS_OCCURRED_A;
{
    var x$1, x$2, y$1, y$2 : int;
    call LOG_READ_A(tid$1 + idx$1);
    call CHECK_READ_A(tid$2 + idx$2);
    havoc x$1, x$2;
    call LOG_READ_A(tid$1);
    call CHECK_READ_A(tid$2);
    havoc y$1, y$2;
    call LOG_WRITE_A(tid$1);
    call CHECK_WRITE_A(tid$2);
}
```
Non-deterministic logging

```
var x$1, x$2, y$1, y$2 : int;
call LOG_READ_A(tid$1 + idx$1);
call CHECK_READ_A(tid$2 + idx$2);
havoc x$1, x$2;
call LOG_READ_A(tid$1);
call CHECK_READ_A(tid$2);
havoc y$1, y$2;
call LOG_WRITE_A(tid$1);
call CHECK_WRITE_A(tid$2);
```
Non-deterministic logging

call LOG_READ_A(tid$1 + idx$1);
call CHECK_READ_A(tid$2 + idx$2);
call LOG_READ_A(tid$1);
call CHECK_READ_A(tid$2);
call LOG_WRITE_A(tid$1);
call CHECK_WRITE_A(tid$2);
Non-deterministic logging

//call LOG_READ_A(tid$1 + idx$1);
if (*) { READ_HAS_OCCURRED_A := true; READ_OFFSET_A := tid$1 + idx$1; }
//call CHECK_READ_A(tid$2 + idx$2);
assert (WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_A != tid$2 + idx$2);
//call LOG_READ_A(tid$1);
if (*) { READ_HAS_OCCURRED_A := true; READ_OFFSET_A := tid$1; }
//call CHECK_READ_A(tid$2);
assert (WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_A != tid$2)
//call LOG_WRITE_A(tid$1);
if (*) { WRITE_HAS_OCCURRED_A := true; WRITE_OFFSET_A := tid$1; }
//call 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);
Non-deterministic logging

//call LOG_READ_A(tid$1 + idx$1);
if (*) { }
//call CHECK_READ_A(tid$2 + idx$2);
assert (WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_A != tid$2 + idx$2);
//call LOG_READ_A(tid$1);
if (*) { READ_HAS_OCCURRED_A := true; READ_OFFSET_A := tid$1; }
//call CHECK_READ_A(tid$2);
assert (WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_A != tid$2)
//call LOG_WRITE_A(tid$1);
if (*) { WRITE_HAS_OCCURRED_A := true; WRITE_OFFSET_A := tid$1; }
//call 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);
Non-deterministic logging

```c
//call LOG_READ_A(tid$1 + idx$1);
if (*) { READ_HAS_OCCURRED_A := true; READ_OFFSET_A := tid$1 + idx$1; }
//call CHECK_READ_A(tid$2 + idx$2);
assert (WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_A != tid$2 + idx$2);
//call LOG_READ_A(tid$1);
if (*) { READ_HAS_OCCURRED_A := true; READ_OFFSET_A := tid$1; }
//call CHECK_READ_A(tid$2);
assert (WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_A != tid$2);
//call LOG_WRITE_A(tid$1);
if (*) { WRITE_HAS_OCCURRED_A := true; WRITE_OFFSET_A := tid$1; }
//call 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 barrier() function

```plaintext
procedure barrier() {
    assume (!READ_HAS_OCCURRED_A);
    assume (!WRITE_HAS_OCCURRED_A);
    // Do this for every array
}
```
Summary so far

- For each array parameter \( A \):
  - Add variables to log \( A \)'s reads and writes
  - Generate procedures to log and check reads and writes, using non-determinism to consider all possibilities
  - Remove \( A \), and model reads from \( A \) using non-determinism

- For each statement in kernel \( K \):
  - generate corresponding statement(s) in sequential program \( P \)
  - interleave two arbitrary threads using round-robin schedule

- Next up: conditionals and loops
Handling conditionals

- Use predicated execution to flatten **conditional code** into **straight line code**

```c
if (x < 100) {
    x = x+1;
} else {
    y = y+1;
}
```

```c
P := (x < 100);
Q := !(x < 100);
x := (P ? x+1 : x);
y := (Q ? y+1 : y);
```
Handling conditionals

- Use predicated execution to flatten conditional code into straight line code
- Each statement is tagged with a predicate that determines which threads are enabled
- This complicates the translation...
Translating statements (revised)

<table>
<thead>
<tr>
<th>S</th>
<th>translate(S,P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = e;</td>
<td>x$1 := P$1 ? e$1 : x$1; x$2 := P$2 ? e$2 : x$2;</td>
</tr>
<tr>
<td>x = A[e];</td>
<td>call LOG_READ_A(P$1, e$1); call CHECK_READ_A(P$2, e$2); x$1 := P$1 ? * : x$1; x$2 := P$2 ? * : x$2;</td>
</tr>
<tr>
<td>A[e] = e';</td>
<td>call LOG_WRITE_A(P$1, e$1); call CHECK_WRITE_A(P$2, e$2);</td>
</tr>
<tr>
<td>barrier();</td>
<td>call barrier(P$1, P$2);</td>
</tr>
<tr>
<td>S₁; S₂;</td>
<td>translate(S₁,P); translate(S₂,P);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>translate(S,P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(e) {</td>
<td>Q$1 := P$1 &amp;&amp; e$1; Q$2 := P$2 &amp;&amp; e$2;</td>
</tr>
<tr>
<td>S₁; } else {</td>
<td>R$1 := P$1 &amp;&amp; !e$1; R$2 := P$2 &amp;&amp; !e$2;</td>
</tr>
<tr>
<td>}</td>
<td>translate(S₁,Q); translate(S₂,R);</td>
</tr>
<tr>
<td>while(e) {</td>
<td>Q$1 := P$1 &amp;&amp; e$1; Q$2 := P$2 &amp;&amp; e$2;</td>
</tr>
<tr>
<td>}</td>
<td>while (Q$1</td>
</tr>
<tr>
<td>}</td>
<td>Q$1 := P$1 &amp;&amp; e$1; Q$2 := P$2 &amp;&amp; e$2;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
The logging functions (revised)

procedure LOG_READ_A(enabled : bool, offset : int) {
    if (enabled && *) {
        READ_HAS_OCCURRED_A := true;
        READ_OFFSET_A := offset;
    }
}

procedure LOG_WRITE_A(enabled : bool, offset : int) {
    if (enabled && *) {
        WRITE_HAS_OCCURRED_A := true;
        WRITE_OFFSET_A := offset;
    }
}
The checking functions (revised)

```plaintext
procedure CHECK_READ_A(enabled : bool, offset : int) {
    assert (enabled && WRITE_HAS_OCCURRED_A
        ==> WRITE_OFFSET_A != offset);
}

procedure CHECK_WRITE_A(enabled : bool, offset : int) {
    assert (enabled && WRITE_HAS_OCCURRED_A
        ==> WRITE_OFFSET_A != offset);
    assert (enabled && WRITE_HAS_OCCURRED_A
        ==> WRITE_OFFSET_A != offset);
}
```
The barrier() function (revised)

```plaintext
procedure barrier(enabled$1 : bool, enabled$2 : bool) {
    assert (enabled$1 == enabled$2);
    if (!enabled$1) return;

    assume (!READ_HAS_OCCURRED_A);
    assume (!WRITE_HAS_OCCURRED_A);
    // Do this for every array
}
```
Find out more

- Download GPUVerify:  
  - multicore.doc.ic.ac.uk/tools/GPUVerify

- Or try it online:  
  - rise4fun.com/GPUVerify-OpenCL

- The Multicore Group at Imperial  
  - multicore.doc.ic.ac.uk
Further reading


- P. Collingbourne, A. Donaldson, J. Ketema, S. Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels, ESOP 2013

- 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