FAT-GPU: Formal Analysis Techniques for GPU kernels

Tutorial 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

Supported by the FP7 project CARP: Correct and Efficient Accelerator Programming
Aims of this tutorial

- Explain two pitfalls of GPU programming: \textbf{data races} and \textbf{barrier divergence}
- Demonstrate \textbf{GPUVerify}, a tool for statically analysing GPU kernels to check for these kinds of defects
- Introduce some of the verification techniques underlying \textbf{GPUVerify}
- (Compare with another GPU verification tool)
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
CUDA Architecture

GPU

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

Local memory

Global memory

GPU
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

Identifies the current thread

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

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

\[
\begin{array}{ccccc}
1 & 1 & 1 & 1 & 1 \\
\text{Thread 0} \\
1 & 1 & 1 & 1 & 1 \\
\text{Thread 1} \\
1 & 1 & 1 & 1 & 1 \\
\text{Thread 2} \\
1 & 1 & 1 & 1 & 1 \\
\text{Thread 3} \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]
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

```kernel
__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();
}
```

NOT ALLOWED
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 + 5 + 7 + 4 + 6 + 3 + 2 + 0 = 28
## Striding

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
How the GPUVerify tool works
Architecture of GPUVerify

OpenCL or CUDA kernel

Frontend, built on CLANG/LLVM

Kernel transformation engine

Boogie verification engine

Z3 SMT solver

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

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

Thread N-1
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!

- Check each statement for races with the already-executed statements of all threads with lower tids
Reduction to two threads

- We can do better still!

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

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

```
Thread j
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 \(i \neq j\))
- 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:

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

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

  - $x = e$
  - $x = A[e]$
  - $A[e] = e'$
  - `barrier()`

local variable  
expression over local variables  
__local array parameter
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:

```
var tid$1 : int;
var tid$2 : int;
```

and assume that they are in-range and different:

```
requires 0 <= tid$1 && tid$1 < N;
requires 0 <= tid$2 && tid$2 < N;
requires tid$1 != tid$2;
```
Logging reads and writes

- Replace each `__local` array parameter `A` with four global variables:

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

- and four procedures:

  ```
  procedure LOG_READ_A(offset : int);
  procedure LOG_WRITE_A(offset : int);
  procedure CHECK_READ_A(offset : int);
  procedure CHECK_WRITE_A(offset : int);
  ```
The transformation so far...

Kernel $K$:

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

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 : 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:

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

<table>
<thead>
<tr>
<th>S</th>
<th>translate(S)</th>
</tr>
</thead>
</table>
| x = e; | x$1 := e$1;  
x$2 := e$2; |
| x = A[e]; | call LOG_READ_A(e$1);  
call CHECK_READ_A(e$2);  
havoc x$1, x$2; |
| A[e] = e'; | call LOG_WRITE_A(e$1);  
call CHECK_WRITE_A(e$2); |
| barrier(); | call barrier(); |
| S₁; S₂; | translate(S₁); translate(S₂); |
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 tid1, tid2 : 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(idx1 : int, idx2 : int)
    requires 0 <= tid1 && tid1 < N;
    requires 0 <= tid2 && tid2 < N;
    requires tid1 != tid2;
    requires idx1 == idx2;
{
    var x1, x2, y1, y2 : int;
    call LOG_READ_A(tid1 + idx1);
    call CHECK_READ_A(tid2 + idx2);
    havoc x1, x2;
    call LOG_READ_A(tid1);
    call CHECK_READ_A(tid2);
    havoc y1, y2;
    call LOG_WRITE_A(tid1);
    call CHECK_WRITE_A(tid2);
}
```
The logging functions

```plaintext
procedure LOG_READ_A(offset : int) {
    if (*) {
        READ_HAS_OCCURRED_A := true;
        READ_OFFSET_A := offset;
    }
}
```

```plaintext
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 tid1, tid2 : 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(idx1 : int, idx2 : int);
    requires 0 <= tid1 && tid1 < N;
    requires 0 <= tid2 && tid2 < N;
    requires tid1 != tid2;
    requires !READ_HAS_OCCURRED_A;
    requires !WRITE_HAS_OCCURRED_A;
    { var x1, x2, y1, y2 : int;
      call LOG_READ_A(tid1 + idx1);
      call CHECK_READ_A(tid2 + idx2);
      havoc x1, x2;
      call LOG_READ_A(tid1);
      call CHECK_READ_A(tid2);
      havoc y1, y2;
      call LOG_WRITE_A(tid1);
      call CHECK_WRITE_A(tid2);
    }
```
Non-deterministic logging

```plaintext
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

```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);
```
Non-deterministic logging

```c
//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

//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 (*) { WRITE_HAS_OCCURRED_A := true; WRITE_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

```c
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**

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

```plaintext
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>\text{call LOG_READ_A}(P$1, e$1); \text{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>\text{call LOG_WRITE_A}(P$1, e$1); \text{call CHECK_WRITE_A}(P$2, e$2);</td>
</tr>
<tr>
<td>barrier();</td>
<td>\text{call barrier}(P$1, P$2);</td>
</tr>
<tr>
<td>S(_1); S(_2);</td>
<td>translate(S(_1),P); translate(S(_2),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>} else {</td>
<td>(R$1 := P$1 &amp;&amp; \neg e$1; ) (R$2 := P$2 &amp;&amp; \neg e$2; )</td>
</tr>
<tr>
<td>S(_1);</td>
<td>translate(S(_1),Q); translate(S(_2),R);</td>
</tr>
<tr>
<td>S(_2);</td>
<td>translate(S(_2),P);</td>
</tr>
<tr>
<td>while(e) {</td>
<td>\text{while (Q$1</td>
</tr>
<tr>
<td>S;</td>
<td></td>
</tr>
</tbody>
</table>
The logging functions (revised)

```plaintext
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)

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)

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


- 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