Show how the theory, algorithms and tools of the verification community can be brought to bear on an interesting and important domain
Schedule

• GPUs and their programming models

• The need for semantics and verification tools

• Building a static analyser with off-the-shelf parts

• Examining the GPUVerify verification method
Organisation

• Please ask questions!

• Material online
  http://multicore.doc.ic.ac.uk/tools/GPUVerify/tutorials/POPL14

• Try GPUVerify online
  http://rise4fun.com/GPUVerify-CUDA
  http://rise4fun.com/GPUVerify-OpenCL
GPUs and their programming models
Graphics Processing Units

• Originally designed to accelerate graphics processing

• Operations on pixels are inherently parallel

• Early GPUs were specialised for graphics

• Modern GPUs are more powerful and general purpose

• Now widely used as accelerators for many domains
copy data and kernel code

host
cpu

gpu

global memory

local memory
The diagram illustrates the flow of data between the host CPU and the GPU. The process begins with the host CPU invoking a kernel function. The kernel function accesses both local and global memory on the GPU. The local memory is used for temporary data sharing between threads, while the global memory stores data that is accessible by all threads in the kernel. The diagram shows the data exchange between the host and GPU, emphasizing the importance of efficient memory management in GPU programming.
GPU

Local memory

Global memory

Copy back results

Host CPU
processing elements or cores

local memory

global memory
multiprocessors

local memory

global memory
CUDA and OpenCL

- thread / work-item
- block / workgroup
- grid / ndrange
- core
- multiprocessor
- gpu
- private
- shared / local
- global
__kernel void vadd(
    __local int *A, __local int *B,
    __local int *C) {

    int tid = get_local_id(0);
}

Summary

• Kernels are run on massively parallel GPUs

• Potentially hundreds of thousands of threads acting in concert over shared memory

• Wide variety of applications and domains
The need for semantics and verification tools
Data races

- 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
global memory

local memory

intra-group race

X
global memory

local memory

inter-group race

X
__kernel void
add_nbor(__local int *A, int offset) {
    int tid = get_local_id(0);
}

s  s+offset

\[ \begin{array}{cccccccc}
    & & & & & & & \\
    & & & & & & & \\
    & & & & & & & \\
    & & & & & & & \\
    s & s+offset & & & & & & \\
    & & & & & & & \\
    & & & & & & & \\
    & & & & & & & \\
\end{array} \]
__kernel void
add_nbor(__local int *A, int offset) {
    int tid = get_local_id(0);
}

5

X

7

t3  t5
barrier()
barrier()
barrier() = synchronisation + consistency
__kernel void
add_nbor(__local int *A, int offset) {
    int tid = get_local_id(0);

}
__kernel void
add_nbor(__local int *A, int offset) {
    int tid = get_local_id(0);
    int tmp = A[tid+offset];
    barrier();
}
__kernel void diverge1() {
    int tid = get_local_id(0);
    if (tid == 0) barrier();
    else barrier();
}

__kernel void diverge2() {
    int x = get_local_id(0) == 0 ? 4 : 1;
    int y = get_local_id(0) == 0 ? 1 : 4;

    for (int i=0; i<x; i++)
        for (int j=0; j<y; j++)
            barrier();
}

__kernel void diverge2() {
    int x = get_local_id(0) == 0 ? 4 : 1;
    int y = get_local_id(0) == 0 ? 1 : 4;

    for (int i=0; i<x; i++)
        for (int j=0; j<y; j++)
            barrier();
}

Summary

• Data races result in non-determinism

• Barrier divergence leads to undefined behaviour

• Bugs of these kind are rarely intended and not benign
Techniques for data race detection
Building a static analyser
Essential idea

GPU kernel $\xrightarrow{\text{transform}}$ SMT problem

SAT: potential counterexample

UNSAT: could not find any bugs
Satisfiability Modulo Theories

- SMT = SAT solver + Theories

- Decision procedures for
  - Equality and uninterpreted functions
  - Bitvectors
  - Arrays
  - ...

38
OpenCL kernel

CUDA kernel

Frontend (built on LLVM/CLANG)

Kernel Transformation Engine

Sequential candidate loop invariants

Boogie Verification Engine

SMT Solvers
Boogie

• Intermediate verification language target

• Imperative language with mathematical components

• Generates verification conditions for a variety of solvers

• Used by many static analysers
Invariant inference

- Abstract interpretation
- Predicate abstraction
- Candidate generation and Houdini
- Interpolation
- Abduction
Houdini Algorithm

- A form of predicate abstraction implemented in Boogie
- Takes candidate invariants (guesses) and finds the maximal conjunction of true invariants
- Requires candidate invariant generation for domain
- More predictable than refinement strategies
Summary

• The work of the verification community enables the rapid design and implementation of new tools and techniques

• We live in interesting times!
“It was a real bug, and it caused real issues in the results. It took significant debugging time to find the problem.”

Lars Nyland (Senior Architect, NVIDIA)
Some important observations
Parallel Kernel $K$ \hspace{1cm} translate \hspace{1cm} Sequential Program $P$

race-freedom
2-thread reduction
Arbitrary threads s and t

```
barrier() // b1
```

```
barrier() // b2
```
Arbitrary threads s and t

barrier() // b1

run s from b1 to b2
log all accesses

barrier() // b2
Arbitrary threads $s$ and $t$

```
barrier() // b1
```

```
run $s$ from b1 to b2
log all accesses
```

```
run $t$ from b1 to b2
check all accesses against $s$
abort on race
```

```
barrier() // b2
```
2-thread reduction gives scalable verification
barrier() // b1

run s from b1 to b2
log all accesses
run t from b1 to b2
check all accesses against s

barrier() // b2

run s from b2 to b3
log all accesses
run t from b2 to b3
check all accesses against s

barrier() // b3
\begin{itemize}
  \item \texttt{barrier()}  // b1
  \item run s from b1 to b2
  \item log all accesses
  \item run t from b1 to b2
  \item check all accesses against s
  \item \texttt{barrier()}  // b2
  \item run s from b2 to b3
  \item log all accesses
  \item run t from b2 to b3
  \item check all accesses against s
  \item \texttt{barrier()}  // b3
\end{itemize}

unsound
havoc shared state

\begin{itemize}
\item \texttt{barrier() // b1}
\item \texttt{run s from b1 to b2}
\item log all accesses
\item \texttt{run t from b1 to b2}
\item check all accesses against s
\item \texttt{barrier() // b2}
\item \texttt{run s from b2 to b3}
\item log all accesses
\item \texttt{run t from b2 to b3}
\item check all accesses against s
\item \texttt{barrier() // b3}
\end{itemize}
Shared state abstraction is necessary for soundness
Break
Kernel transformation for straight-line code
Straight-line code

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

Form of each statement:

\[
\begin{align*}
    x &= e & & x \text{ is a local variable} \\
    x &= A[e] & & e \text{ is an expression over local variables and tid} \\
    A[e] &= x & & A \text{ is a __local array parameter}
\end{align*}
\]
2 arbitrary threads

The kernel has implicit variable tid
Suppose \( N \) is the total number of threads
Model tid by introducing two global variables

\[
\text{int } \text{tid}\$1;
\text{int } \text{tid}\$2;
\]

and preconditions:

\[
\begin{align*}
\text{\texttt{\textbackslash requires } 0 \leq \text{tid}\$1 < N;} & \quad \text{\textcolor{red}{\textbullet} Threads in range} \\
\text{\texttt{\textbackslash requires } 0 \leq \text{tid}\$2 < N;} & \quad \text{\textcolor{red}{\textbullet} Threads in range} \\
\text{\texttt{\textbackslash requires } \text{tid}\$1 \neq \text{tid}\$2;} & \quad \text{\textcolor{red}{\textbullet} Threads different}
\end{align*}
\]

...but otherwise the threads are arbitrary
Race checking instrumentation

For each local array $A$ introduce four variables

\[
\begin{align*}
&\text{bool } \text{READ HAS OCCURRED}_A; \\
&\text{bool } \text{WRITE HAS OCCURRED}_A; \\
&\text{int } \text{READ OFFSET}_A; \\
&\text{int } \text{WRITE OFFSET}_A;
\end{align*}
\]

and four procedures

\[
\begin{align*}
&\text{void } \text{LOG READ}_A(\text{int offset}); \\
&\text{void } \text{LOG WRITE}_A(\text{int offset}); \\
&\text{void } \text{CHECK READ}_A(\text{int offset}); \\
&\text{void } \text{CHECK WRITE}_A(\text{int offset});
\end{align*}
\]

and do not represent $A$ in the sequential program
Example

Kernel

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

Sequential program

```c
int tid$1; int tid$2; // 2 thread ids

bool READ_HAS_OCCURRED_A;
bool WRITE_HAS_OCCURRED_A; // Instrumentation variables for array A
int READ_OFFSET_A;
int WRITE_OFFSET_A;

// \texttt{requires} 0 \leq tid$1 < N;
// \texttt{requires} 0 \leq tid$2 < N;
// \texttt{requires} tid$1 \neq tid$2;
void foo(int idx) { ... } // Constraining tid$1 and tid$2 to be arbitrary, distinct threads
```
Dualise local variables

\begin{verbatim}
int x is duplicated to become: int x$1;
int x$2;
\end{verbatim}

This reflects fact that each thread has a copy of $x$

Each non-array parameter $x$ is duplicated similarly
Initially equal for threads

\texttt{requires x$1 == x$2}
More generally

For an expression e let

\(e^1\) denotes \(e\) with every local variable \(x\) replaced by \(x^1\)
and similarly for \(e^2\)

For example: if \(e\) is \(a + \text{tid} - x\) then

\(e^2\) is \(a^2 + \text{tid}^2 - x^2\)
Translating statements

<table>
<thead>
<tr>
<th>Stmt</th>
<th>translate(Stmt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = e;</td>
<td>x$1 = e$1;</td>
</tr>
<tr>
<td></td>
<td>x$2 = e$2;</td>
</tr>
<tr>
<td>x = A[e];</td>
<td>LOG_READ_A(e$1);</td>
</tr>
<tr>
<td></td>
<td>CHECK_READ_A(e$2);</td>
</tr>
<tr>
<td></td>
<td>havoc(x$1); havoc(x$2);</td>
</tr>
</tbody>
</table>

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 assigned variables arbitrary

We removed the array A, thus we over-approximate reading from A using havoc. That is, we make no assumptions about the contents of A.
Translating statements (2)

<table>
<thead>
<tr>
<th>Stmt</th>
<th>translate(Stmt)</th>
<th>Log location to which the first thread writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[e] = x;</td>
<td>LOG_WRITE_A(e$1); CHECK_WRITE_A(e$2); // nothing</td>
<td>Check write by the second thread does not conflict with any prior read or write by the first thread</td>
</tr>
<tr>
<td>barrier();</td>
<td>barrier();</td>
<td>The write itself has no effect because the array A has been removed</td>
</tr>
<tr>
<td>S;</td>
<td>translate(S);</td>
<td>We shall give barrier() a special meaning in the translated program</td>
</tr>
<tr>
<td>T;</td>
<td>translate(T);</td>
<td></td>
</tr>
</tbody>
</table>
Example

__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 < N;
// \requires 0 <= tid$2 < N;
// \requires tid$1 != tid$2;
// \requires idx$1 == idx$2;
__kernel 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);
}
void LOG_READ_A(int offset) {
    if (*) {
        READ_HAS_OCCURRED_A = true;
        READ_OFFSET_A = offset;
    }
}
LOG_WRITE_A

```c
void LOG_WRITE_A(int offset) {
    if (*) {
        WRITE_HAS_OCCURRED_A = true;
        WRITE_OFFSET_A = offset;
    }
}
```
void CHECK_READ_A(int offset) {
    assert(WRITE_HAS_OCCURRED_A =>
        WRITE_OFFSET_A != offset);
}

CHECK_READ_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);
}
Precondition

// \texttt{\textbackslash requires}\ 0 <= \texttt{tid\$1} < N;
// \texttt{\textbackslash requires}\ 0 <= \texttt{tid\$2} < N;
// \texttt{\textbackslash requires}\ \texttt{tid\$1} \neq \texttt{tid\$2};
// \texttt{\textbackslash requires}\ \texttt{idx\$1} == \texttt{idx\$2};
// \texttt{\textbackslash requires}\ !\texttt{READ\_HAS\_OCCURRED\_A};
// \texttt{\textbackslash requires}\ !\texttt{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);
    ...
}
// \requires 0 <= tid$1 < N;
// \requires 0 <= tid$2 < N;
// \requires tid$1 != tid$2;
// \requires idx$1 == idx$2;
// \requires !READ_HAS_OCCURRED_A;
// \requires !WRITE_HAS_OCCURRED_A;
__kernel void foo(int x$1; int x$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);
}
/requires 0 <= tid$1 < N;
// \\ requires 0 <= 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);
}
// 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);
    // 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);
}

Nondeterminism ensures that some program execution checks every pair of potentially racing operations.
// 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);
}
// 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);
}
void barrier() {
    assume(!READ_HAS_OCCURRED_A);
    assume(!WRITE_HAS_OCCURRED_A);
    // Do this for every array
}
Summary

For each array A

- Introduce instrumentation variables
- Generate log and check procedures
- Remove array parameter

For each statement

- Generate corresponding statements
- Interleave two arbitrary threads using round-robin schedule
Loops and conditionals
Predicated execution

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

P = (x < 100);
Q = !(x < 100);

x = (P ? x + 1 : x);
y = (Q ? y + 1 : y);
Encoding loops and conditionals

- Apply predication so that every execution point has a predicate determining whether a thread is enabled
- Add predication to logging and checking procedures and barrier
## Statements with predication

<table>
<thead>
<tr>
<th>Stmt</th>
<th>translate(Stmt, 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>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;</td>
</tr>
<tr>
<td>A[e] = x;</td>
<td>LOG_WRITE_A(P$1, e$1); CHECK_WRITE_A(P$2, e$2);</td>
</tr>
</tbody>
</table>

- LOG and CHECK calls take predicate as parameter.
- We only havoc x$1 and x$2 if P$1 and P$2 are true, respectively.
- LOG and CHECK calls take predicate as parameter.
Generating predicates

<table>
<thead>
<tr>
<th>Stmt</th>
<th>translate(Stmt, P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (e) {</td>
<td>Q$1 = P$1 &amp;&amp; e$1;</td>
</tr>
<tr>
<td>S; } else {</td>
<td>Q$2 = P$2 &amp;&amp; e$2;</td>
</tr>
<tr>
<td>T; }</td>
<td>R$1 = P$1 &amp;&amp; !e$1;</td>
</tr>
<tr>
<td></td>
<td>R$2 = P$2 &amp;&amp; !e$2;</td>
</tr>
<tr>
<td></td>
<td>translate(S, Q);</td>
</tr>
<tr>
<td></td>
<td>translate(T, R);</td>
</tr>
<tr>
<td>while (e) {</td>
<td>Q$1 = P$1 &amp;&amp; e$1;</td>
</tr>
<tr>
<td>S; }</td>
<td>Q$2 = P$2 &amp;&amp; e$2;</td>
</tr>
<tr>
<td>}</td>
<td>while (Q$1</td>
</tr>
<tr>
<td></td>
<td>translate(S, Q);</td>
</tr>
<tr>
<td></td>
<td>Q$1 = Q$1 &amp;&amp; e$1;</td>
</tr>
<tr>
<td></td>
<td>Q$2 = Q$2 &amp;&amp; e$2;</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

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
Statements with Predicates

<table>
<thead>
<tr>
<th>Stmt</th>
<th>translate(Stmt, P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S;</td>
<td>translate(S, P);</td>
</tr>
<tr>
<td>T;</td>
<td>translate(T, P);</td>
</tr>
<tr>
<td>barrier();</td>
<td>barrier(P$1, P$2);</td>
</tr>
</tbody>
</table>

barrier now takes parameters determining whether the threads are enabled
void LOG_READ_A(bool enabled, int offset) {
    if(enabled) {
        if(1) {
            READ_HAS_OCCURRED_A = true;
            READ_OFFSET_A = offset;
        }
    }
}
void CHECK_WRITE_A(bool enabled, int offset) {
    assert(enabled && WRITE_HAS_OCCURRED_A =>
           WRITE_OFFSET_A != offset);
    assert(enabled && READ_HAS_OCCURRED_A =>
           READ_OFFSET_A != offset);
}

Predicated
CHECK_WRITE_A
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
}

Predicated barrier

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

barrier does nothing if the threads are not enabled
Loop invariants (optional)
Houdini

set of candidate invariants

refutations

empty set

maximal fixpoint
__kernel void
stride(__local int *A, uint n) {
  uint tid = get_local_id(0);
  uint size = get_group_size(0);
  for (uint i=tid; i<n; i+=size) {
    A[i] = val;
  }
}
__kernel void
stride(__local int *A, uint n) {
    uint tid = get_local_id(0);
    uint size = get_group_size(0);
    for (uint i = tid; i < n; i += size) {
        A[i] = val;
    }
}

__write(A) ==> __offset(A) % size == tid
__kernel void slice(___local int *A, uint m) {
    uint tid = get_local_id(0);
    for (uint i=0; i<m; i++) {
        A[tid*m+i] = val;
    }
}

0  1  2  3  4  5  6  7
T0 T0 T0 T0 TI TI TI TI
__kernel void
slice(__local int *A, uint m) {
    uint tid = get_local_id(0);
    for (uint i=0; i<m; i++) {
        A[tid*m+i] = val;
    }
}

__write(A) ==> tid*m <= __offset(A) < (tid+1)*m
__kernel void slice(__local int *A, uint m) {
    uint tid = get_local_id(0);
    for (uint i=0; i<m; i++) {
        A[tid*m+i] = val;
    }
}

__write(A) ==> __offset(A) / m == tid
Summary

• Using the Houdini algorithm avoids having to design abstract domains or search for invariants

• Instead we depend on heuristics for generating candidate invariants
Show how the *theory, algorithms and tools* of the verification community can be brought to bear on an interesting and important domain
What we covered

• GPUs and their programming models
• The need for semantics and verification tools
• Building a static analyser with off-the-shelf parts
• Examining the GPUVerify verification method
What we didn’t cover

- Multidimensional groups
- Inter-group race checking with global arrays
- Benign data races
- Atomics and warp synchronisation
- Data-dependent kernels
GPUVerify on YouTube
GPUVerify contributors

Alastair F. Donaldson

• Ethel Bardsley
• Adam Betts
• Nathan Chong
• Peter Collingbourne
• Pantazis Deligiannis

Shaz Qadeer

• Jeroen Ketema
• Egor Kyshtymov
• Daniel Liew
• Paul Thomson
• John Wickerson

102
EU FP7 STREP project CARP (project number 287767)  
EPSRC PSL project (EP/I006761/1)  
EPSRC First Grant (EP/K011499/1)