Formal Analysis Techniques for GPU kernels

Nathan Chong (nyc04@imperial.ac.uk)
Leap Conference, 22 May 2013
“It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics.”

Many people have argued that computer programming should strive to become more like mathematics. Maybe so, but not in the way they seem to think. The aim of program verification, an attempt to make programming more mathematics-like, is to increase dramatically one’s confidence in the correct functioning of a piece of software, and the device that verifiers use to achieve this goal is a long chain of formal, deductive logic. In mathematics, the aim is to increase one’s confidence in the correctness of a theorem, and it’s true that...
Verification as a powerful and practical complement to Testing
“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)
Schedule

• Data races and Barrier Divergence
• Examples, Examples, Examples

• Anatomy of GPUVerify
• Further Examples
• Close and Questions
Data Races and Barrier Divergence
global memory

local memory

intra-group race

X
__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);
}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{image}
\caption{Diagram of an array with indices s and s+\text{offset}}
\end{figure}
__kernel void
add_nbor(__local int *A, int offset) {
    int tid = get_local_id(0);
}

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

s
ts + offset

s
t

t + offset
```c
__kernel void add_nbor(__local int *A, int offset) {
    int tid = get_local_id(0);
}
```
__kernel void diverge() {
    int tid = get_local_id(0);
    if (tid == 0) barrier();
    else barrier();
}

If barrier is inside a conditional statement, then all threads must enter the conditional if any thread enters the conditional statement and executes the barrier.

If barrier is inside a loop, all threads must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier.

OpenCL Specification
(6.12.8 Synchronization Functions)
Reduction Demo

<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>0,2,4,6</td>
<td>1,3,5,7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SUM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Examples, Examples, Examples, Examples
Be Skeptical

• Is the verification easier or harder than building a test harness?

• A common or rare type of bug?
• The impact of not catching this bug
• Limitations of technique
1 Races
__kernel void
add_nbor(__local int *A, int offset) {
    int tid = get_local_id(0);
}

\[
\begin{array}{c}
\text{s} & \text{s+offset} \\
\hline
\text{t} & \text{t+offset}
\end{array}
\]
• Run GPUVerify on nbor.cl

$ cd 1_simple_race
$ gpuverify --local_size=8 --num_groups=1 nbor.cl

• Can you fix the datarace?
• Does GPUVerify like your fix?
• Are there more problems with this kernel?
Lessons

• GPUVerify can find possible data races, giving a counterexample for you to evaluate

• By fixing bugs, you increase your confidence in the verification result

• But still, the verification is limited. For example, we don’t prove absence of array-bounds or functional correctness
2 Benign Races
__kernel void
allsame(__local int *p, int val) {
    *p = val;
}

• Run GPUVerify on allsame.cl

$ cd 2_benign_race
$ gpuverify --local_size=8 --num_groups=1 allsame.cl

• Try adding “--no-benign” to the command
• Change “val” to “get_local_id(0)”
• Have a look at the example in find.cl
Lessons

• Benign data races do not lead to nondeterminism

• Use `--no-benign` flag to warn about benign data races
3 Barrier Divergence
__kernel void diverge() {
    int tid = get_local_id(0);
    if (tid == 0) barrier();
    else barrier();
}

__kernel void inloop() {
    int x = tid == 0 ? 4 : 1;
    int y = tid == 0 ? 1 : 4;

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

• Run GPUVerify on these examples

$ cd 3_barrier_divergence
$ gpuverify --local_size=8 --num_groups=1 diverge.cl
$ gpuverify --local_size=8 --num_groups=1 inloop.cl

• Is the inloop kernel barrier divergent?
• What does the inloop kernel try to do?
If barrier is inside a conditional statement, then all threads must enter the conditional if any thread enters the conditional statement and executes the barrier.

If barrier is inside a loop, all threads must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier.

OpenCL Specification
(6.12.8 Synchronization Functions)
\[ A = \{\{0, 1, 2, 3\}, \{\_, \_, \_, \_\}\} \rightarrow \{\{0, 1, 2, 3\}, \{1, 2, 3, 0\}\} \rightarrow \{\{2, 3, 0, 1\}, \{1, 2, 3, 0\}\} \rightarrow \{\{2, 3, 0, 1\}, \{3, 0, 1, 2\}\} \rightarrow \{\{0, 1, 2, 3\}, \{3, 0, 1, 2\}\} \]

<table>
<thead>
<tr>
<th>GPU</th>
<th>Final state of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA Tesla C2050</td>
<td>{0, 1, 0, 1}, {1, 0, 1, 0}</td>
</tr>
<tr>
<td>AMD Tahiti</td>
<td>{0, 1, 2, 3}, {1, 2, 3, 0}</td>
</tr>
<tr>
<td>ARM Mali-T600</td>
<td>{0, 1, 2, 3}, {3, 0, 1, 2}</td>
</tr>
<tr>
<td>Intel Xeon X5650</td>
<td>{*,*,*,1}, {3,0,1,2}</td>
</tr>
</tbody>
</table>
Lessons

• Barrier divergence results in undefined behaviour
• GPUVerify can detect such problems
• Arguably, this is a rare bug?
4 Asserts and Assumes
__kernel void simple(__local int *A) {
    A[tid] = tid;
    __assert(A[tid] == tid);
    __assert(A[tid] != get_local_size(0));
    __assert(__implies(
        __write(A),
        __write_offset(A)/sizeof(int) == tid));
}

• Run GPUVerify on these examples

$ cd 4_asserts_and_assumes
$ gpuverify --local_size=8 --num_groups=1 assert.cl

• Try writing your own assertions
• Have a look at vacuous.cl
• Does this surprise you?
Lessons

• Use asserts to state expected details of your kernel at a particular program point

• The dangers of inconsistent assumptions

• Use __assert(false) to test for inconsistency
5 Loops
__kernel void inc(int x) {

    int i = 0;
    while (i < x) {
        i = i + 1;
    }
    __assert(i == x);
}

__kernel void inc(int x) {
    __requires (0 < x);

    int i = 0;
    while (i < x) {
        i = i + 1;
    }
    __assert(i == x);
}
__kernel void inc(int x) {
    __requires (0 < x);

    int i = 0;
    while (__invariant(?), i < x) {
        i = i + 1;
    }
    __assert(i == x);
}

• Run GPUVerify on these examples

$ cd 5_loops
$ gpuverify --local_size=8 --num_groups=1 inc.cl

• Try running with the “--findbugs” flag
• Can you find an invariant for the loop?
• Take a look at stride.cl
Lessons

• Loop invariants are assertions that are true at every loop iteration
• GPUVerify attempts to guess invariants
• They may be necessary to strengthen verification to avoid false-positives
• Use --findbugs to do loop unwinding
Anatomy of GPUVerify
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
Translate parallel kernel $K$ into sequential program $P$ such that $P$ correct implies $K$ is race-free
OpenCL kernel

CUDA kernel

Frontend (built on LLVM/CLANG)

Kernel Transformation Engine

sequential Boogie program

candidate loop invariants

Boogie Verification Engine

Widely used, very robust

Z3 SMT Solver

The only magic is here
Further Examples
__kernel void dbl_indirect(__local int *A) {

    A[tid] = tid;
    barrier();
    A[A[(tid+1)%N]] = tid;

}
__kernel void dbl_indirect(__local int *A) {
    A[tid] = tid;
    barrier();
    A[A[(tid+1)%N]] = tid;
}

7 0 1 2 3 4 5 6
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
barrier() // b1
  \(\exists\) run \(s\) from b1 to b2
  \(\exists\) log all accesses
  \(\exists\) run \(t\) from b1 to b2
  \(\exists\) check all accesses against \(s\)
barrier() // b2
  \(\exists\) run \(s\) from b2 to b3
  \(\exists\) log all accesses
  \(\exists\) run \(t\) from b2 to b3
  \(\exists\) check all accesses against \(s\)
barrier() // b3
havoc shared state

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
Shared state abstraction is necessary for soundness
GPUVerify: A Verifier for GPU Kernels

Adam Betts¹  Nathan Chong¹  Alastair F. Donaldson¹  Shaz Qadeer²  Paul Thomson¹

¹Department of Computing, Imperial College London, UK  ²Microsoft Research, Redmond, USA
{abetts,nyc04,afd,pt1110}@imperial.ac.uk  qadeer@microsoft.com

Abstract

We present a technique for verifying race- and divergence-freedom of GPU kernels that are written in mainstream kernel programming languages such as OpenCL and CUDA. Our approach is based on a novel formal operational semantics for GPU programming termed synchronous, delayed visibility (SDV) semantics. The SDV semantics provides a precise definition of barrier divergence in GPU kernels and allows kernel verification to be reduced to analysis of a sequential program, thereby completely avoiding the need to reason about thread interleavings, and allowing existing modular techniques for program verification to be leveraged. We describe an efficient encoding for data race detection and propose a method for automatically inferring loop invariants required for verification. We have implemented these techniques as a practical verification tool, GPUVerify, which can be applied directly to OpenCL and CUDA source code. We evaluate GPUVerify with respect to a set of 163 kernels drawn from public and commercial sources. Our evaluation demonstrates that GPUVerify is capable of efficient, automatic verification of a large number of real-world kernels.

Categories and Subject Descriptors F3.1 [Logics and Meanings of Programs]: Specifying, Verifying & Reasoning

In contrast to the well-understood notion of data races, there does not appear to be a formal definition of barrier divergence for GPU programming. Our work begins by giving such a definition. Barriers are common synchronization points in OpenCL and CUDA and are used to synchronize the execution of threads in a workgroup. In contrast to the well-understood notion of data races, there does not appear to be a formal definition of barrier divergence for GPU programming. Our work begins by giving such a definition. Barriers are common synchronization points in OpenCL and CUDA and are used to synchronize the execution of threads in a workgroup.

In OOPSLA'12 63
__kernel void iterall_edges(
  __local uint2 *edges,
  __local uint *edgecolour,
  __local float *node_val
) {

  __requires(?);

  for (uint c=0; c < MAX_COLOUR; c++) {
    if (c == edgecolour[tid]) {
      node_val[edges[tid].lo] = ...;
      node_val[edges[tid].hi] = ...
    }
    barrier();
  }
}
• Write a precondition that satisfies the colouring requirement

$ cd 6\_further$

$ gpuverify --local\_size=8 --num\_groups=1 graph.cl$

• Preconditions and assertions are a kind of executable documentation
Row Major

$A_{ij}$ stored at $i + (\text{width} \times j)$

<table>
<thead>
<tr>
<th></th>
<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</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<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>2</td>
<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>3</td>
<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>4</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>6</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

height = 8

width = 8
• Check out transpose.cu

$ cd 6_further
$ gpuverify --blockDim=[4,2] --gridDim=[2,2] -DWIDTH=8 -DHEIGHT=8 -DTILE_DIM=4 -DBLOCK_ROWS=2 transpose.cu

• Involves tricky loop invariants for reasoning about data accesses of individual threads

• More invariants than lines of code!
Lessons

• Valuable to know the limitations of the tools you use
• Discovering loop invariants can be time-consuming (but rewarding!)
• It is possible to reason about complicated kernels if the engineering investment is worthwhile
Closing
Verification as a powerful and practical complement to Testing
Formal reasoning as a valuable discipline
Search ‘GPUVerify’ on YouTube
GPUVerify: a Verifier for GPU Kernels

What is GPUVerify?

GPUVerify is a tool for formal analysis of GPU kernels written in OpenCL and CUDA. The tool can prove that kernels are free from certain types of defect, including data races.

```c
__kernel void add_neighbour(__local int *A, int offset) {
    int tid = get_local_id(0);
    int temp = A[tid + offset];
    barrier(CLK_LOCAL_MEM_FENCE);
    A[tid] += temp;
}
```

gpuverify --local_size=64 --num_groups=128 add-neighbour-correct.cl

Verified: add-neighbour-correct.cl
- no data races within work groups
- no data races between work groups
- no barrier divergence
- no assertion failures
- no other defects detected

http://multicore.doc.ic.ac.uk/tools/GPUVerify
<table>
<thead>
<tr>
<th>Role</th>
<th>Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Research</td>
<td>Shaz Qadeer</td>
</tr>
<tr>
<td>Frontend</td>
<td>Adam Betts, Peter Collingbourne</td>
</tr>
<tr>
<td>Semantics heavy lifting</td>
<td>Jeroen Ketema, Paul Thomson</td>
</tr>
<tr>
<td>PhD students</td>
<td>Nathan Chong, Dan Liew</td>
</tr>
<tr>
<td>UROP students</td>
<td>Egor Kyshtymov, Cassie Epps</td>
</tr>
</tbody>
</table>

Work supported by EU FP7 STREP project CARP (project number 287767) and EPSRC PSL project (EP/I006761/1).