Semantics and Verification for GPU Kernels

Nathan Chong and Jeroen Ketema
{nyc04, j.ketema}@imperial.ac.uk

Abstract. Semantics and Verification for GPU Kernels is a tutorial on recent and promising progress in the semantics and verification of massively-parallel GPU kernels. We will focus on how the theory, algorithms and tools of the verification community can be brought to bear on this interesting and important new domain. The tutorial will cover the formal semantics of GPU programs as well as domain-specific features and modelling techniques that can be exploited to make race-checking in this domain practical. Despite the parallel verification context we will show how the problem can be reduced to a sequential setting, therefore enabling scalable and automatic verification.

This tutorial is aimed at researchers and practitioners in software verification who would be interested in seeing how a practical implementation of automated Hoare Logic can be built using standard components in the verification community. These standard components include abstraction, invariant generation, the Boogie intermediate verification language, and SMT solvers. The tutorial will not assume a background in GPU programming.

Reading List

GPUs and their programming models
- J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach.
- NVIDIA. CUDA C Programming Guide.
- The Khronos Group. The OpenCL Specification.

Techniques for data race detection

Dynamic techniques using race instrumentation
- M. Zheng, V.T. Ravi, F. Qin and G. Agrawal. GRace: A Low-Overhead Mechanism for Detecting Data Races in GPU Programs. PPoPP 2011.

Dynamic techniques using symbolic execution
- G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. PPoPP 2012.
- P. Collingbourne, C. Cadar and P.H.J. Kelly. Symbolic Testing of OpenCL Code. HVC 2011.

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

Static techniques
- A. Betts, N. Chong, A.F. Donaldson, S. Qadeer and P. Thomson. GPUVerify: a Verifier for GPU Kernels. OOPSLA 2012
- G. Li and G. Gopalakrishnan. Scalable SMT-Based Verification of GPU Kernel Functions. FSE 2010

Building a static analyser
- K.R.M Leino. This is Boogie 2.
- L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. TACAS 2008.
- C. Barrett, C.L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli. CVC4. CAV 2011.
- C. Flanagan and K.R.M. Leino. Houdini, an Annotation Assistant for ESC/Java. FME 2001.

Code

Vector Add (http://rise4fun.com/GPUVerify-OpenCL/bT)

```
__kernel void vadd(__local float *A, __local float *B, __local float *C) {
  int tid = get_local_id(0);
  C[tid] = A[tid] + B[tid];
}
```

Add Neighbour, racy (http://rise4fun.com/GPUVerify-OpenCL/add_neighbour_bug.cl)

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

Add Neighbour, fixed (http://rise4fun.com/GPUVerify-OpenCL/add_neighbour_correct.cl)

```
__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;
}
```

Barrier Divergence, syntactic (http://rise4fun.com/GPUVerify-OpenCL/simple_barrier_divergence.cl)

```
__kernel void diverge() {
  int tid = get_local_id(0);
  if (tid == 0) barrier(CLK_LOCAL_MEM_FENCE);
  else          barrier(CLK_LOCAL_MEM_FENCE);
}
```

Barrier divergence, loop (http://rise4fun.com/GPUVerify-OpenCL/in_loop_barrier_divergence.cl)

```
__kernel void inloop() {
  __local int A[2][4];
  int buf, i, j;

  int tid = get_local_id(0);
  int x = tid == 0 ? 4 : 1;
  int y = tid == 0 ? 1 : 4;

  buf = 0;
  for(int i = 0; i < x; i++) {
    for(int j = 0; j < y; j++) {
      barrier(CLK_LOCAL_MEM_FENCE);
      A[1-buf][tid] = A[buf][(tid+1)%4];
      buf = 1 - buf;
    }
  }
}
```

Running straight-line code example (http://rise4fun.com/GPUVerify-OpenCL/NC9**)**

```
__kernel void foo(__local int *A, int idx) {
  int x;
  int y;

  x = A[tid + idx];
  y = A[tid];
  A[tid] = x + y;
}
```

Running straight-line code example in Boogie (http://rise4fun.com/Boogie/akha**)**

```
// Introduce 2 arbitrary threads
var tid$1 : int;
var tid$2 : int;

// Assume there are N threads in total
const N : int;
axiom N == 1024;

// Race instrumentation
var READ_HAS_OCCURRED_A : bool;
var READ_OFFSET_A : int;
var WRITE_HAS_OCCURRED_A : bool;
var WRITE_OFFSET_A : int;

// Remove declaration of array [A] and dualise [idx] parameter
procedure foo(idx$1: int, idx$2: int)
  // Choose two arbitrary threads
  requires 0 <= tid$1 && tid$1 < N;
  requires 0 <= tid$2 && tid$2 < N;
  requires tid$1 != tid$2;
  // Non-array formal initially equal
  requires idx$1 == idx$2;
  // Clear read/write sets
  requires !READ_HAS_OCCURRED_A;
  requires !WRITE_HAS_OCCURRED_A;
  modifies READ_HAS_OCCURRED_A, READ_OFFSET_A, WRITE_HAS_OCCURRED_A, WRITE_OFFSET_A;
{
  // Dualised local variables
  var x$1: int; var x$2 : int;
  var y$1: int; var y$2 : int;

  // Translation of "x = A[tid+idx]"
  call LOG_READ_A(tid$1 + idx$1);
  call CHECK_READ_A(tid$2 + idx$2);
  havoc x$1, x$2;

  // Translation of "y = A[tid]"
  call LOG_READ_A(tid$1);
  call CHECK_READ_A(tid$2);
  havoc y$1, y$2;

  // Uncommenting this barrier fixes the race
  // call barrier();

  // Translation of "A[tid] = x + y"
  call LOG_WRITE_A(tid$1);
  call CHECK_WRITE_A(tid$2);
```

```
}

// Race logging and checking procedures
procedure {:inline 1} LOG_READ_A(offset : int)
  modifies READ_HAS_OCCURRED_A, READ_OFFSET_A;
{
  if (*) { // non-deterministically choose to log or not
    READ_HAS_OCCURRED_A := true;
    READ_OFFSET_A := offset;
  }
}

procedure {:inline 1} CHECK_READ_A(offset: int) {
  assert (WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_A != offset);
}

procedure {:inline 1} LOG_WRITE_A(offset : int)
  modifies WRITE_HAS_OCCURRED_A, WRITE_OFFSET_A;
{
  if (*) {
    WRITE_HAS_OCCURRED_A := true;
    WRITE_OFFSET_A := offset;
  }
}

procedure {:inline 1} CHECK_WRITE_A(offset : int)
{
  assert (WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_A != offset);
  assert (READ_HAS_OCCURRED_A ==> READ_OFFSET_A != offset);
}

procedure {:inline 1} barrier()
{
  assume(!READ_HAS_OCCURRED_A);
  assume(!WRITE_HAS_OCCURRED_A);
}
```