Towards a formalisation of the OpenCL memory model

Mark Batty

University of Cambridge
GPU relaxed memory models

Collaboration with:

J. Alglave
UCL

A. Donaldson
Imperial

D. Poetzl
Oxford

T. Sorensen
Utah

J. Wickerson
Imperial
OpenCL 2.0 specification

Defined by Khronos Group: an extension of the C11 MM

Use atomics to write racy concurrent code

— But —

Scopes allow atomics to target a particular cache depth

Address space divided: local, global, private, constant

Cannot assume forward progress

The language provides barriers for SIMD concurrency
OpenCL address spaces

Global memory
OpenCL address spaces

Device

Global memory
OpenCL address spaces

Device

Work group

…

Work group

Global memory
OpenCL address spaces

Device

Work group

WI ... WI

... Work group

WI ... WI

Global memory
OpenCL address spaces

Device

Work group

WI ... WI

Local memory

Work group

WI ... WI

Local memory

Global memory
OpenCL address spaces

Device

Work group

Local memory

Global memory
Axiomatic semantics

The semantics of a program is a set of execution graphs; here is one:

```
int r1, r2;
global int x=0;
global atomic_int y=0;
x=1;               \parallel r1=load_{RLX}(&y);
store_{RLX}(&y,1); \parallel r2=x;
```

```
W x=0
```

```
W y=0
```

```
R_{RLX} y=1
```

```
W x=1
```

```
W_{RLX} y=1
```

```
W x=1
```

```
R x=0
```

```
ghb
```
Data races are faults

Racy, non-atomic accesses cause the program to have undefined behaviour.

```c
int r1, r2;
global int x=0;
global atomic_int y=0;
x=1; r1=load_RLX(&y);
store_RLX(&y,1); r2=x;
```

```
W x=0
W y=0
ghb
W x=1
W_RLX y=1
dr
RRLX y=1
R x=0
```
Data races are faults

Racy, non-atomic accesses cause the **program** to have undefined behaviour.

```c
int r1, r2;
global int x=0;
global atomic_int y=0;
x=1; r1=load_RLX(&y); store_RLX(y, 1); r2=x;
```

**Racy, non-atomic accesses cause the program to have undefined behaviour.**

Data races are faults
Choose stronger memory orders to synchronise.

```c
int r1, r2;
global int x=0;
global atomic_int y=0;
x=1;
```

Choose stronger memory orders to synchronise.

```c
r1=load_{RLX}(&y);
store_{RLX}(&y,1);
r2=x;
```

**Memory orders:** RLX, ACQ, REL, SC
Global happens before

Choose stronger memory orders to synchronise.

```c
int r_1, r_2;
global int x=0;
global atomic_int y=0;
x=1;  // | r_1=load_{ACQ}(&y);
store_{REL}(&y,1); // r_2=x;
```

rf from REL to ACQ becomes an ghb edge
Local happens before

Local/global memory spaces each have own hb.

```c
int r1, r2;
local int x=0;
local atomic_int y=0;
x=1;  
|| r1=load_{ACQ}(&y);
store_{REL}(&y,1); || r2=x;
```
Local and global hb are (mostly) separate

They can form a cycle with program order!

```
local atomic_int l=0;
global atomic_int g=0;
r1 = load_{ACQ}(&l);
store_{REL}(&g,1);
r2 = load_{ACQ}(&g);
store_{REL}(&l,1);
```
Several opportunities for caching

Device

Work group

WI ... WI

Local memory

Work group

WI ... WI

Local memory

Global memory
Memory scopes

Each atomic access takes a **memory scope** parameter:

- work-item
- work-group
- device
- all-devices

Two actions have **inclusive scope** if their scope params match, and their LCA is at or above that scope.
Racy, non-inclusive atomics are faults

...and non-inclusive atomics do not synchronise.

```c
int r1, r2;
global atomic_int x=0;
global atomic_int y=0;

\ work-group 1 \ work-group 2
store_{RLX-WG}(&x, 1);  r1=load_{ACQ-WG}(&y);
store_{REL-WG}(&y, 1);  r2=load_{RLX-WG}(&x);
```

and non-inclusive atomics do not synchronise.
Racy, non-inclusive atomics are faults

...and non-inclusive atomics do not synchronise.

```c
int r1, r2;
global atomic int x=0;
global atomic int y=0;
\ work-group 1
store_{RLX-WG}(&x, 1);
store_{REL-WG}(&y, 1);
\ work-group 2
r1 = load_{ACQ-WG}(&y);
r2 = load_{RLX-WG}(&x);
```

...and non-inclusive atomics do not synchronise.
Inclusively-scoped atomics synchronise

There is no data race in the execution.

```c
int r1, r2;
global atomic_int x=0;
global atomic_int y=0;
\ work-group 1 \ work-group 2
store_{RLX-WG}(&x,1);
store_{REL-DEV}(&y,1);
\ r1=load_{ACQ-DEV}(&y);
\ r2=load_{RLX-WG}(&x);
```

```
W x=0
W y=0

\ ghb \\
W_{RLX-WG} x=1
W_{REL-DEV} y=1
R_{ACQ-DEV} y=1
R_{RLX-WG} x=1
```
No assumption of forward progress

This breaks waiting for another work-item:

\[
x = 1; \quad \text{while}(\text{load}(&y) \not\equiv 1); \quad r = x;
\]

store(&y,1); \quad \text{while}(\text{load}(&y) \not\equiv 1);

But we can use the atomics to manage contention:

\[
\text{do } \{ \\
\quad t1 = \text{load}(&\text{top}); \\
\quad t2 = \text{mutate}(t1); \\
\} \text{ while } \\
\quad \text{CAS}(&\text{top},t1,t2)
\]
Current state

Preliminary Lem and Herd models

Preliminary compiler mapping to Nvidia PTX

Model seems weak: several ideas for a stronger sound model