GPU Concurrency: Weak Behaviours and Programming Assumptions
– TECHNICAL APPENDICES –

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan,
Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson

Abstract

This document contains technical appendices to the paper GPU Concurrency: Weak Behaviours and Programming Assumptions *[1].*

1. Additional litmus tests

1.1. exch-sl

In Fig. 1, we distil the Stuart and Owens lock implementation [5] into a test called **exch-sl** (for "spin lock using exchange"), again with additional barriers that we ignore for now.

On AMD TeraScale 2 and GCN 1.0 we observe weak behaviour for the OpenCL version of **exch-sl** presented in App. 2.3. On insertion of global memory fences we no longer observe this weak behaviour.

init: `x=0 ∧ m=1` **final:** `r0=0 ∧ r2=0` **threads:** inter-CTA

0.1	`st.cg [x],1`	1.1	`atom.exch r0,[m],1`
0.2(+)	`membar.gl`	1.2	`setp.eq r1,r0,0`
0.3	`atom.exch r0,[m],0`	1.3(+)	`@r1 membar.gl`
		1.4	`@r1 ld.cg r2,[x]`

obs/100k	GTX5 0	TesC 59	GTX6 59	Titan 336	GTX7 0	HD6570 742	HD7970 956

Figure 1: Spin lock using exchange (exch-sl)

2. ISA mappings and OpenCL litmus tests for testing AMD chips

2.1. Mappings from OpenCL to Evergreen and Southern Islands ISA instructions

Tables 1 and 2 indicate how the AMD OpenCL compiler used in our experiments maps OpenCL statements to Evergreen [2] and Southern Islands [3] ISA instructions, respectively. The tables correspond to the PTX-to-OpenCL mapping of [1, Tab. 5], except that we omit details of atomic addition and volatile accesses which are not used in our OpenCL litmus tests. Note that OpenCL atomics are named slightly differently to CUDA atomics: the CUDA `atomicCAS` and `atomicExch` operations are named `atomic_cmpxchg` and `atomic_xchg` in OpenCL 1.2 [4].

2.2. OpenCL version of the spin lock example

Fig. 2 presents an OpenCL analogue of the CUDA spin lock implementation shown in [1, Fig. 2]. The methods of [1, Fig. 2] are members of a struct representing a lock. In the OpenCL version the lock is instead passed to the `lock` and `unlock` routines as a parameter, since OpenCL does not allow structs to be equipped with member functions . Otherwise the translation from Fig. [1, Fig. 2] to Fig. 2 is direct: the CUDA `__device__` keyword is removed, and CUDA atomics are replaced with their OpenCL counterparts.

```
1   void lock(global int *mutex) {
2     while( atomic_cmpxchg( mutex, 0, 1 ) != 0 );
3(+)  mem_fence(CLK_GLOBAL_MEM_FENCE);}
4   void unlock(global int *mutex) {
5(+)  mem_fence(CLK_GLOBAL_MEM_FENCE);
6     atomic_xchg( mutex, 0 );}
```

Figure 2: OpenCL analogue of the CUDA spin lock of [1, Fig. 2]. OpenCL versions of the additional fences are highlighted

2.3. OpenCL litmus tests

We now present OpenCL versions of the litmus tests discussed in Sections 3 and 4 of [1].

Tracking load-load and store-store pairs As discussed in [1, Sec 4.4] we do not yet have a version of optcheck for our AMD testing setup, and checked the compiled versions of our OpenCL litmus tests by hand to guard against compiler optimisations. During manual checking it was difficult to distinguish between load operations in the case where a thread issues multiple loads; similarly, pairs of store operations were difficult to distinguish between. To make manual checking more reliable we edited our tests as follows (assuming loads and stores of 32-bit values). We modified store-store pairs within a thread by setting an additional significant bit in the literal value associated with the second store. For instance, if a thread would issue two stores of the literal value `1` we edited the test so that the thread would store the literal values `0x00000001` and `0x01000001`. We also edited all tests so that within a thread the top 30 bits of the result of the first load issued by the thread are masked out, the top 29 bits of the result of the second load issued by the thread are masked out, and so on.

As an example, in the **mp** test of Fig. 5, T_0 stores the values `0x00000001` and `0x01000001` to `x` and `y`, and T_1 masks the result of its first load by `0x00000011` and the result of its second load by `0x00000111`.

This combination of bit-setting for stores and bit-masking for loads ensures that (a) the possible final values in registers always correspond to the possible final values before modification, and (b) it is straightforward to recognise which ISA

OpenCL	Evergreen
`atomic_cmpxchg`	`MEM_RAT_CMPXCHG`
`atomic_xchg`	`MEM_RAT_XCHG`
`mem_fence(CLK_GLOBAL_MEM_FENCE)`	`WAIT_ACK: Outstanding_acks <= 0`
store to global int	`MEM_RAT_CACHELESS_STORE_RAW`
load from global int	`VFETCH`
control flow (`while`, `if`)	loops, jumps and predicated instructions

Table 1: OpenCL to Evergreen ISA mapping

OpenCL	Southern Islands
`atomic_cmpxchg`	`buffer_atomic_cmpswap`
`atomic_xchg`	`buffer_atomic_swap`
`mem_fence(CLK_GLOBAL_MEM_FENCE)`	`s_waitcnt vmcnt(0)`
store to global int	`tbuffer_store_format_x`
load from global int	`tbuffer_load_format_x`
control flow (`while`, `if`)	jumps and predicated instructions

Table 2: OpenCL to Southern Islands ISA mapping

instructions correspond to which loads and stores in load-load and store-store pairs during manual inspection.

To aid readability we only apply bit-setting and bit-masking in the tests presented in this section in cases where at least one load-load or store-store pair is present.

Avoiding alias-based optimisations In some tests, notably **coRR**, we found that the AMD OpenCL compiler optimise two loads from the same location into a single load. This optimisation is legitimate, but yields ISA that does not encode the litmus test of interest. To make the optimisation impossible we equip each OpenCL kernel with an additional parameter named `zero`, and replace successive reads from a location `x` by a read from `x`, a read from `x + zero`, a read from `x + zero + zero`, and so on. At compile-time the compiler knows nothing about the values that `zero` may take, and is thus forced to generate separate load instructions. At runtime we pass 0 as the value for `zero` to ensure that each load is from the same location.

Avoiding predicated instructions Some of the PTX litmus tests presented in [1, Sec. 3.3] use predicated instructions; for instance the **cas-sl** test of[1, Fig. 9]. We attempted to represent these tests faithfully in OpenCL using conditional statements but found that the compiler would invariantly issue a memory fence before a conditional test. For the **dlb-mp** and **cas-sl** examples of [1, Sec. 3.3] and the **exch-sl** example, predication is not necessary for the purposes of litmus testing: the truth of the final condition can be seen to be independent of the values taken by predicate registers. For these examples we thus created OpenCL variants that do not use predication. As mentioned in [1, Sec. 3.3.3] this was not possible for the **sl-future** test, which uses both predicated load and predicated stores.

- Fig. 3 presents the OpenCL **sb** test used for evaluation of incantations in [1, Sec. 4].
- Fig. 4 presents the OpenCL **lb** test used for evaluation of incantations in [1, Sec. 4].
- Fig. 5 presents the OpenCL **mp** test discussed in [1, Sec. 3.1.2] and used for evaluation of incantations in [1, Sec. 4]. This test uses bit-setting and bit-masking to enable manual checking for store-store and load-load reorderings.
- Fig. 6 presents the OpenCL **coRR** test discussed in [1, Sec. 3.1.1] and used for evaluation of incantations in [1, Sec. 4]. This test uses the special `zero` parameter to avoid alias-based optimisations. Bit-masking is used to enable manual checking for load-load reorderings.
- Fig. 7 presents the OpenCL **dlb-mp** test discussed in [1, Sec. 3.3.1]. This test uses bit-masking to enable manual checking for load-load reorderings. The predicates present in the PTX version of [1, Fig. 7] are omitted as discussed above.
- Fig. 8 presents the OpenCL **dlb-lb** test discussed in [1, Sec. 3.3.1].
- Fig. 9 presents the OpenCL **cas-sl** test discussed in [1, Sec. 3.3.2]. The predicates present in the PTX version of [1, Fig. 9] are omitted as discussed above.
- Fig. 10 presents the OpenCL **exch-sl** test discussed in Sec. 1.1. The predicates present in the PTX version of Fig. 1 are omitted as discussed above.

init: `*x=0 ∧ *y=0` **final:** `r0=0 ∧ r1=0` **threads:** inter-CTA

```
0.1   *x = 1;          1.1   *y = 1;
0.2   r0 = *y;         1.2   r1 = *x;
```

Figure 3: OpenCL store buffering litmus test, sb

init: `*x=0 ∧ *y=0` **final:** `r0=1 ∧ r1=1` **threads:** inter-CTA

```
0.1   r0 = *x;         1.1   r1 = *y;
0.2   *y = 1;          1.2   *x = 1;
```

Figure 4: OpenCL load buffering litmus test, lb

init: `*x = 0 ∧ *y = 0` **final:** `r1 = 1 ∧ r2 = 0` **threads:** inter-CTA

```
0.1   *x = 0x00000001;                          1.1   r1 = *y;
0.2(+) mem_fence(CLK_GLOBAL_MEM_FENCE);         1.2(+) mem_fence(CLK_GLOBAL_MEM_FENCE);
0.3   *y = 0x01000001;                          1.3   r2 = *x;
                                                1.4   r1 = r1 & 0x00000011;
                                                1.5   r2 = r2 & 0x00000111;
```

Figure 5: Message passing OpenCL litmus test, mp; fences are added to obtain mp+fence

init: `*x = 0 ∧ zero = 0` **final:** `r1 = 1 ∧ r2 = 0` **threads:** inter-CTA

```
0.1   *x = 1;                                   1.1   r1 = *x;
                                                1.2(+) mem_fence(CLK_GLOBAL_MEM_FENCE);
                                                1.3   r2 = *(x + zero);
                                                1.4   r1 = r1 & 0x00000011;
                                                1.5   r2 = r2 & 0x00000111;
```

Figure 6: coRR OpenCL litmus test; fence is added to obtain coRR+fence. The dummy variable `zero` is used to prevent a compiler optimisation that optimises the two loads from `x` into a single load

init: `*t = 0 ∧ *d = 0` **final:** `r0 = 1 ∧ r1 = 0` **threads:** inter-CTA

```
0.1   *d = 1;                                   1.1   r0 = *t;
0.2   mem_fence(CLK_GLOBAL_MEM_FENCE);          1.2(+) mem_fence(CLK_GLOBAL_MEM_FENCE);
0.3   r2 = *t;                                  1.3   r1 = *d;
0.4   r2 = r2 + 1;                              1.4   r0 = r0 & 0x00000011;
0.5   *t = r2;                                  1.5   r1 = r1 & 0x00000111;
```

Figure 7: OpenCL litmus test for message passing w/ dynamic load-balancing, dlb-mb; fence is added to obtain dlb-mb+fence

init: `*t = 0 ∧ *h = 0` **final:** `r0 = 1 ∧ r1 = 1` **threads:** inter-CTA

```
0.1   r0 = atomic_cmpxchg(h, 0, 1);             1.1   r1 = *t;
0.2(+) mem_fence(CLK_GLOBAL_MEM_FENCE);         1.2(+) mem_fence(CLK_GLOBAL_MEM_FENCE);
0.3   *t = 1;                                   1.3   r3 = atomic_cmpxchg(h, 0, 1);
```

Figure 8: OpenCL litmus test for load buffering with dynamic load-balancing, dlb-lb; fences are added to obtain dlb-lb+fence

init: `*x = 0 ∧ *m = 1` **final:** `r0 = 0 ∧ r2 = 0` **threads:** inter-CTA

```
0.1   *x = 1;                                   1.1   r0 = atomic_cmpxchg(m, 0, 1);
0.2(+) mem_fence(CLK_GLOBAL_MEM_FENCE);         1.2(+) mem_fence(CLK_GLOBAL_MEM_FENCE);
0.3   r0 = atomic_xchg(m, 0);                   1.3   r2 = *x;
```

Figure 9: OpenCL litmus test for spin lock using compare-and-swap, cas-sl; fences are added to obtain cas-sl+fence

init: `*x = 0 ∧ *y = 1` **final:** `r1 = 1 ∧ r2 = 0` **threads:** inter-CTA

```
0.1   *x = 1;                                   1.1   r0 = atomic_xchg(y, 1);
0.2(+) mem_fence(CLK_GLOBAL_MEM_FENCE);         1.2(+) mem_fence(CLK_GLOBAL_MEM_FENCE);
0.3   r0 = atomic_xchg(y, 0);                   1.3   r2 = *x;
```

Figure 10: OpenCL litmus test for spin lock using exchange, exch-sl; fences are added to obtain exch-sl+fence

References

[1] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS '15*, 2015. To appear.

[2] AMD. Evergreen family instruction set architecture: Instructions and microcode, revision 1.1a, Nov. 2011.

[3] AMD. Southern Islands series instruction set architecture, revision 1.1, Dec. 2012.

[4] Khronos OpenCL Working Group. The OpenCL specification (version 1.2, revision 19), Nov. 2012.

[5] Jeff A. Stuart and John D. Owens. Efficient synchronization primitives for GPUs. *Computing Research Repository (CoRR)*, abs/1110.4623, 2011. http://arxiv.org/pdf/1110.4623.pdf.