Portable Inter-Workgroup Barrier Synchronisation for GPUs

Supplementary Material

Tyler Sorensen
Imperial College London, UK
t.sorensen15@imperial.ac.uk

Alastair F. Donaldson
Imperial College London, UK
alastair.donaldson@imperial.ac.uk

Mark Batty
University of Kent, UK
m.j.batty@kent.ac.uk

Ganesh Gopalakrishnan
University of Utah, USA
ganesh@cs.utah.edu

Zvonimir Rakamarić
University of Utah, USA
zvonimir@cs.utah.edu

chip	vendor	#CUs	type	short name
GTX 980	Nvidia	16	discrete	980
Quadro K5200	Nvidia	12	discrete	K5200
Iris 6100	Intel	47	integrated	6100
HD 5500	Intel	24	integrated	5500
Radeon R9	AMD	28	discrete	R9
Radeon R7	AMD	8	integrated	R7
Mali-T628	ARM	4	integrated	T628-4
Mali-T628	ARM	2	integrated	T628-2

Table 1: The GPUs we used for empirical evaluation

1. Introduction

This is supplementary material as referenced in the paper *Portable Inter-Workgroup Barrier Synchronisation for GPUs*.

2. Custom OpenCL 2.0 Atomic Implementations

The GPUs in Table 1 all support the OpenCL 2.0 memory model except for Nvidia and ARM chips. For these chips, we provide custom implementations of the OpenCL 2.0 atomic operations. Our Nvidia implementation is based on previous empirical testing of these chips [1]. We use inline PTX (Nvidia's low level intermediate language [2]) to provide Nvidia specific memory fences. Our ARM implementation uses OpenCL 1.1 memory fence instructions. Both use OpenCL 1.1 read-modify-write instructions. While our implementations come with no proof of correctness, we are conservative with our fence placement and encounter no issues in our experiments (Section 5). These implementations should be seen as temporary until OpenCL 2.0 is more widely supported.

We only provide the atomic operations used in the inter-workgroup barrier (Section 4) and mutex implementations (Appendix 3). These mappings may not be correct when a larger portion of the memory model is considered (e.g. SC and relaxed memory orders). Our Nvidia mapping is given in Table 2 and our ARM mapping is given in Table 3. Each atomic instruction takes a location `l` and optionally an operand `v`. The memory order and scope annations are given in square brackets. We list abbreviated names (e.g. dropping `atomic` and `explicit`) for conciseness.

Atomic operation	mapping
`atomic_int`	`int`
`fetch_add[acq_rel, device](l,v))`	`membar.gl;` `atomic_add(l,v);` `membar.gl;`
`load[acq, device](l)`	`ret = *l;` `membar.gl;`
`store[rel, device](l,v)`	`membar.gl;` `*l = v;`
`exchange[acq_rel, device](l,v)`	`membar.gl;` `atomic_xchg(l,v);` `membar.gl;`

Table 2: Nvidia custom OpenCL 2.0 atomic mapping

Atomic operation	mapping
`atomic_int`	`int`
`fetch_add[acq_rel, device](l,v))`	`mem_fence(GLOBAL_FENCE);` `atomic_add(l,v);` `mem_fence(GLOBAL_FENCE);`
`load[acq, device](l)`	`ret = *l;` `mem_fence(GLOBAL_FENCE);`
`store[rel, device](l,v)`	`mem_fence(GLOBAL_FENCE);` `*l = v;`
`exchange[acq_rel, device](l,v)`	`mem_fence(GLOBAL_FENCE);` `atomic_xchg(l,v);` `mem_fence(GLOBAL_FENCE);`

Table 3: ARM custom OpenCL 2.0 atomic mapping

```
struct mutex {
  atomic_int l;
};

void lock(mutex *m) {
   while( atomic_exchange_explicit(&(m->l),
          1, memory_order_acq_rel, memory_scope_device)
          == 1);
}

void unlock(mutex *m) {
   atomic_store_explicit(&(m->l), 0,
   memory_order_release, memory_scope_device);
}
```

Figure 1: Inter-workgroup OpenCL spin-lock

```
struct mutex {
  atomic_int counter;
  atomic_int now_serving;
};

void lock(mutex *m) {
  int ticket = atomic_fetch_add_explicit(&(m->counter),
               1, memory_order_acq_rel,
               memory_scope_device);

  while( atomic_load_explicit(&(m->now_serving),
         memory_order_acq, memory_scope_device)
          != ticket);


}

void unlock(mutex *m) {
  int tmp = atomic_load_explicit(&(m->now_serving),
            memory_order_acquire, memory_scope_device);
  tmp+=1;
  atomic_store_explicit(&(m->now_serving),
    tmp, memory_order_release, memory_scope_device);
}
```

Figure 2: Inter-workgroup OpenCL ticket-lock

3. Mutex Implementations

Here, we provide the implementation of the the two mutex approaches used in this work. The spin-lock is shown in Figure 1 and the ticket-lock is shown in Figure 2. See Section 5.1 for explanation of these locks. These lock implementations may not be optimal in terms of memory order annotations (e.g. some accesses may be able to be demoted to relaxed). However, we prefer to be conservative in our implementations, especially given that the protocol (which uses these locks) is only executed once per kernel.

4. Language and execution calculation

We alter the language covered by library abstraction: we exclude atomic sections and rewrite the top-level parallel composition of threads with a parallel composition of workgroups: The language becomes:

$$\begin{aligned}
C &::= \mathsf{skip} \mid c \mid m \mid C;C \mid \\
&\qquad \mathsf{if}(x)\ \{C\}\ \mathsf{else}\ \{C\} \mid \mathsf{while}(x)\ \{C\} \\
G &::= C_1 \parallel \ldots \parallel C_k \\
\mathcal{L} &::= \{m = C_m \mid m \in M\} \\
\mathcal{C}(\mathcal{L}) &::= \mathsf{let}\ \mathcal{L}\ \mathsf{in} \\
&\qquad G_1 ||| \ldots ||| G_n
\end{aligned}$$

The program is the combination of library $\mathcal{L}$ with methods $m \in M$, and a client $\mathcal{C}$. The composition of threads is now structured into a parallel composition of workgroups, where each workgroup is itself a parallel composition of threads. The commands then follow library abstraction: they include skip, a set of base commands $c \in \mathsf{BComm}$ (which we shall alter to include scoped accesses and barriers), method calls $m \in \mathsf{Method}$, sequential composition, branching and loops.

The global barrier takes no arguments and provides no return value, so although library abstraction provides arguments and return values, we do not use them here.

Memory accesses. We support a very restricted subset of OpenCL memory accesses as base commands:

- non-atomic stores, $\mathsf{store}(x, a)$, and loads, $\mathsf{load}(x)$,
- scoped release stores, $\mathsf{store}_{\mathsf{REL},\sigma}(x, a)$, and scoped acquire loads, $\mathsf{load}_{\mathsf{ACQ},\sigma}(x)$, for some scope $\sigma \in \{\mathsf{wg}, \mathsf{dv}\}$, and
- workgroup and global barriers, $\mathsf{barrier}_{\mathsf{wg}}$ and $\mathsf{barrier}_{\mathsf{dv}}$.

Each of these base commands generates a corresponding event in an execution according to the following set of rules.

Execution calculation. We augment the thread-local semantics of library abstraction in the following ways:

- We introduce scopes to the store and load events of atomics.
- We introduce barrier entry and exit events. In the thread-local semantics, we make a set of the two event identifiers, and keep them in an auxiliary.
- We have sequential composition build up a sequence of these barrier event sets, one at the workgroup level and one at the device level.
- We define a nested parallel composition of work-items, in place of the flat parallel thread composition of library abstraction, and we have this composition build a barrier instance relation from the sequences of barrier identifiers.

Figure 3 presents these changes to the thread local semantics.

5. Soundness theorem.

The soundness of the abstraction relation is captured by the following theorem statement:

THEOREM 1 (Abstraction). *Assume that* $(\mathcal{L}_1, \mathcal{I}_1)$, $(\mathcal{L}_2, \mathcal{I}_2)$, $(\mathcal{C}(\mathcal{L}_2), \mathcal{I} \uplus \mathcal{I}_2)$ *are safe,* $(\mathcal{C}(\mathcal{L}_1), \mathcal{I} \uplus \mathcal{I}_1)$ *is non-interfering, and* $(\mathcal{L}_1, \mathcal{I}_1) \sqsubseteq (\mathcal{L}_2, \mathcal{I}_2)$. *Then* $(\mathcal{C}(\mathcal{L}_1), \mathcal{I} \uplus \mathcal{I}_1)$ *is safe and*

$$\mathsf{client}([\![\mathcal{C}(\mathcal{L}_1)]\!](\mathcal{I} \uplus \mathcal{I}_1)) \subseteq \mathsf{client}([\![\mathcal{C}(\mathcal{L}_2)]\!](\mathcal{I} \uplus \mathcal{I}_2)).$$

This statement has only changed from that in library abstraction insofar as the underlying definitions of the memory model and NONINTERF, and the proof that it holds need only change in its supporting lemmas and propositions. The proof of soundness of the abstraction relation relies on four other properties:

1. The *decomposition lemma*, that provides us with the semantics of the library and client parts of the program separately.
2. The *composition lemma*, that states that the behaviour of the composition of the library and client parts of the program can be defined by combining the execution sets of the separate parts.
3. Per-execution safety under arbitrary extension of happens-before.
4. Total ordering of interface accesses by sequenced-before.

The last two properties are straightforward to establish. For the third note that the definition of heterogeneous races, hr, implies that adding happens-before edges can only remove faults, so we have:

PROPOSITION 2. *If an execution* X *is safe, then so is its extension with any* R.

The fourth property follows from the fact that the thread-local semantics has not been extended in a way that introduces partial ordering into the generation of the sequenced-before, sb, relation.

PROPOSITION 3. *For any consistent execution* X,

$$\mathsf{interf}(\mathsf{hb}(X)) \cap \mathsf{thd} = \mathsf{interf}(\mathsf{sb}(X)).$$

Decomposition lemma

LEMMA 4 (Decomposition). *For any* $X \in [\![\mathcal{C}(\mathcal{L}_1)]\!](I \uplus I_1)$ *satisfying* NONINTERF,

$$\mathsf{client}(X) \in [\![\mathcal{C},\, \mathsf{hbL}(\mathsf{core}(\mathsf{lib}(X)))]\!] I; \qquad (1)$$
$$\mathsf{lib}(X) \in [\![\mathcal{L}_1,\, \mathsf{hbC}(\mathsf{core}(\mathsf{client}(X)))]\!] I_1. \qquad (2)$$

Furthermore,

- $\mathsf{client}(X)$ *and* $\mathsf{lib}(X)$ *satisfy* NONINTERF*; and*
- *if* X *is unsafe, then so is either* $\mathsf{client}(X)$ *or* $\mathsf{lib}(X)$.

The proof of decomposition follows that of library abstraction closely, and relies on several other properties. The first is a direct consequence of the structure of the axioms of the model.

PROPOSITION 5. *Consider an execution* X *and a set of actions* $B \subseteq A(X)$. *Let* $X' = X|_B$. *For every one of the following axioms, if* X *satisfies the axiom, then* X' *does:* ACYCLICITY, COHERENCE, *and* ATOMICRF.

Now we consider the proof of Lemma 4. First note that because X satisfies NONINTERF, it is clear that client and library projections of X are reproducible by the thread-local semantics:

$$(A(\mathsf{client}(X)), \mathsf{sb}(\mathsf{client}(X)), \mathsf{bi}(\mathsf{client}(X)) \in \langle \mathcal{C}(\cdot) \rangle I;$$
$$(A(\mathsf{lib}(X)), \mathsf{sb}(\mathsf{lib}(X)), \mathsf{bi}(\mathsf{lib}(X))) \in \langle \mathsf{MGC}_s(\mathcal{L}_1) \rangle I_1,$$

where s represents the thread and workgroup structure of $\mathcal{C}(\mathcal{L}_1)$. It is also clear that NONINTERF holds for $\mathsf{client}(X)$ and $\mathsf{lib}(X)$. Now we establish the consistency of $\mathsf{client}(X)$ and $\mathsf{lib}(X)$. Proposition 5, leaves us only NONATOMICRF remaining to establish under the extended happens-before relation.

Because NONINTERF holds over X, we know that mo, rf and bi do not cross between the library and the client, and therefore the projection of sw from X matches its derivation in the extended execution. Now we must show that NONATOMICRF holds under each extension in the client and the library case. They are analogous, so we work on the client case. We let $R = \mathsf{hbL}(\mathsf{core}(\mathsf{lib}(X)))$, and we show that:

$$\mathsf{hb}(\mathsf{client}(X)) = (\mathsf{sb}(\mathsf{client}(X)) \cup \mathsf{sw}(\mathsf{client}(X)) \cup R)^+.$$

We have:

$$(\mathsf{sb}(\mathsf{client}(X)) \cup \mathsf{sw}(\mathsf{client}(X)) \cup R)^+ \subseteq (\mathsf{sb}(X) \cup \mathsf{sw}(X) \cup \mathsf{hb}(X)^+ \subseteq \mathsf{hb}(X).$$

Since $\mathsf{sb}(\mathsf{client}(X))$, $\mathsf{sw}(\mathsf{client}(X))$ and R relate client actions only, this implies

$$(\mathsf{sb}(\mathsf{client}(X)) \cup \mathsf{sw}(\mathsf{client}(X)) \cup R)^+ \subseteq \mathsf{hb}(\mathsf{client}(X)).$$

We now show that:

$$\mathsf{hb}(\mathsf{client}(X)) \subseteq (\mathsf{sb}(\mathsf{client}(X)) \cup \mathsf{sw}(\mathsf{client}(X)) \cup R)^+.$$

Take an arbitrary edge $(u, v) \in \mathsf{hb}(\mathsf{client}(X))$. There is a chain of edges from u to v in $\mathsf{sb}(X)$ and $\mathsf{sw}(X)$. By inspecting the thread-local semantics, we see that we can split any $\mathsf{sb}(X)$ edge in the chain to make all call and return actions explicit. NONINTERF guarantees that $\mathsf{sw}(X)$ does not cross between the library and the client. Hence, any sequence of library actions in the chain is bounded by a call and a return action, and those are covered by an R edge. Following this observation further, we can replace every sequence of library edges with the corresponding R edge, and we have:

$$(u, v) \in (\mathsf{sb}(\mathsf{client}(X)) \cup \mathsf{sw}(\mathsf{client}(X)) \cup R)^+,$$

as required.

Then, because NONINTERF guarantees that all accesses to a non-atomic location reside in only a single component, we have NONATOMICRF and we have shown that:

$$\begin{aligned}\mathsf{client}(X) &\in [\![\mathcal{C},\ \mathsf{hbC}(\mathsf{core}(\mathsf{lib}(X)))]\!]I;\\ \mathsf{lib}(X) &\in [\![\mathcal{L}_1,\ \mathsf{hbL}(\mathsf{core}(\mathsf{client}(X)))]\!]I_1.\end{aligned}$$

Note that any heterogeneous race must be between two accesses to the same component, by NONINTERF. Therefore, any such race will also be a race in $\mathsf{client}(X)$ or $\mathsf{lib}(X)$, as required.

Composition lemma

LEMMA 6 (Composition). *Consider*

$$\begin{aligned}X &\in [\![\mathcal{C},\ \mathsf{hbL}(\mathsf{core}(Y))]\!]I;\\ Y &\in [\![\mathcal{L}_2,\ \mathsf{hbC}(\mathsf{core}(X))]\!]I_2,\end{aligned}$$

such that

- $A(X) \cap A(Y) = \mathsf{interf}(X) = \mathsf{interf}(Y)$ *and otherwise action identifiers in* $A(X)$ *and* $A(Y)$ *are different;*
- $\mathsf{interf}(\mathsf{sb}(X)) = \mathsf{interf}(\mathsf{sb}(Y))$*; and*
- X *and* Y *satisfy* NONINTERF.

Then for some $Z \in [\![\mathcal{C}(\mathcal{L}_2)]\!](I \uplus I_2)$ *we have* $X = \mathsf{client}(Z)$ *and* $Y = \mathsf{lib}(Z)$. *Furthermore, if* X *is unsafe, then so is* Z.

Again, the proof follows library abstraction closely.

PROPOSITION 7.

cross between components. We can identify parts of the cycle in one component, say the client, and then, using the call and return events, identify an edge in the other, say the library, that is present in the happens-before extended executions in the assumptions of the lemma. We have found a cycle in this execution, a contradiction of our assumptions.

Finally note that if there is a heterogeneous race in X, by NONINTERF, and because we have established equality over the happens-before projection, we know that there is a heterogeneous race in Z as well, as required.

5.0.1 Soundness theorem: proof sketch

Given an execution X of $\mathcal{C}(\mathcal{L}_1)$, we decompose it into $\mathsf{client}(X)$ and $\mathsf{lib}(X)$. We then apply abstraction under extension with the hb edges induced by $\mathsf{client}(X)$ to get a library-local execution Y of $\mathcal{L}_2$ with a history matching $\mathsf{lib}(X)$.We then compose Y with $\mathsf{client}(X)$ to produce the required execution of $\mathcal{C}(\mathcal{L}_2)$.

References

[1] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS*, pages 577–591. ACM, 2015.

[2] Nvidia. Parallel thread execution ISA: Version 4.2, March 2015. http://docs.nvidia.com/cuda/pdf/ptx_isa_4.2.pdf.

$$\begin{aligned}
\langle \mathsf{store}(x, v, \mathsf{release}, \mathsf{device})\rangle_t &= \{(\{w\}, \{x\}, (\mathsf{rel}, = \{w\}, \mathsf{W} = \{w\}, \mathsf{loc}_x = \{w\}, \mathsf{val}_v = \{w\}), \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)\} \\
\langle \mathsf{load}(x, \mathsf{acquire}, \mathsf{device})\rangle_t &= \{(\{r\}, \{x\}, (\mathsf{acq}, = \{r\}, \mathsf{R} = \{r\}, \mathsf{loc}_x = \{r\}), \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)\} \\
\langle \mathsf{barrier}()\rangle_t &= \{(\{b_1, b_2\}, \emptyset, (\mathsf{entry}_{\mathsf{wg}} = \{b_1\}, \mathsf{exit}_{\mathsf{wg}} = \{b_2\}), \{b_1, b_2\}, \{b_1, b_2\}, \{(b_1, b_2)\}, \emptyset, [(b_1, b_2)])\} \\
\langle \mathsf{global\_barrier}\rangle_{\mathsf{t}} &= \{(\{b_1, b_2\}, \emptyset, (\mathsf{entry}_{\mathsf{dv}} = \{b_1\}, \mathsf{exit}_{\mathsf{dv}} = \{b_2\}), \{b_1, b_2\}, \{b_1, b_2\}, \emptyset, [(b_1, b_2)])\} \\
\langle C_1; C_2\rangle_t &= \{(E_1 \cup E_2, I_1 \cup I_2, lbl_1 \cup lbl_2, \mathsf{thd}_1 \cup \mathsf{thd}_2 \cup (E_1 \times E_2), \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \mathsf{sb}_1 ; \mathsf{sb}_2, \emptyset, \mathsf{bi}_1 \cdot \mathsf{bi}_2) | \\
&\qquad (E_1, I_1, lbl_1, \mathsf{thd}_1, \mathsf{sb}_1, \mathsf{bi}_1) \in \langle C_1\rangle_t \wedge (E_2, I_2, lbl_2, \mathsf{thd}_2, \mathsf{sb}_2, \mathsf{bi}_2) \in \langle C_2\rangle_t\}
\end{aligned}$$

$$\begin{aligned}
&\langle C_1 \parallel \ldots \parallel C_n\rangle = \{(\textstyle\bigcup_{t=1}^n E_t, \bigcup_{t=1}^n I_t, \bigcup_{t=1}^n lbl_t, \bigcup_{t=1}^n \mathsf{thd}_t, \bigcup_{t=1}^n \mathsf{sb}_t, \mathsf{bi}_{\mathsf{wg}}, \bigcup_{t=1}^n \mathsf{bi}_t) \mid \\
&\qquad (\forall t = 1..n.\ (E_t, I_t, lbl_t, \mathsf{thd}_t, \mathsf{sb}_t, \mathsf{bi}_t) \in \langle C_t\rangle)\}\wedge \\
&\qquad \mathsf{bi}_{\mathsf{wg}} = \{(e_1, e_2) \mid e_1, e_2 \in (\mathsf{entry}_{\mathsf{wg}} \cup \mathsf{exit}_{\mathsf{wg}}) \wedge \exists\, n\, t_1\, t_2 . e_1 \in b_{t_1,n} \wedge e_2 \in b_{t_2,n}\}\} \\
&\langle \mathsf{let}\ \{m = C_m \mid m \in M\}\ \mathsf{in}\ C_1 ||| \ldots ||| C_n\rangle I \ = \ \{(\textstyle\bigcup_{t=1}^n E_t, \bigcup_{t=1}^n I_t, \bigcup_{t=1}^n lbl_t, \bigcup_{t=1}^n \mathsf{thd}_t, \bigcup_{t=1}^n \mathsf{sb}_t, \mathsf{bi}) \mid \\
&\qquad ((\forall t = 1..n. (E_t, I_t, lbl_t, \mathsf{thd}_t, \mathsf{sb}_t, \mathsf{bi}_{\mathsf{wg},t}, \mathsf{bi}_t) \in \langle C_t\rangle) \wedge \\
&\qquad (\forall t = 1..n. \forall u. \exists \text{ finitely many } v. (v, u) \in \mathsf{sb}_t)\wedge \\
&\qquad \mathsf{bi} = (\textstyle\bigcup_{t=1}^n \mathsf{bi}_{\mathsf{wg},t}) \cup \{(e_1, e_2) \mid e_1, e_2 \in (\mathsf{entry}_{\mathsf{dv}} \cup \mathsf{exit}_{\mathsf{dv}}) \wedge \exists\, n\, t_1\, t_2 . e_1 \in b_{t_1,n} \wedge e_2 \in b_{t_2,n}\}
\end{aligned}$$

Figure 3: Thread-local semantics. $lbl_1 \cup lbl_2$ is union over each label lifted to the list of all labels.