Portable Inter-workgroup Barrier Synchronisation for GPUs

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

Abstract

Despite the growing popularity of GPGPU programming, there is not yet a portable and formally-specified barrier that one can use to synchronise across workgroups. Moreover, the occupancy-bound execution model of GPUs breaks assumptions inherent in traditional software execution barriers, exposing them to deadlock. We present an *occupancy discovery* protocol that dynamically discovers a safe estimate of the occupancy for a given GPU and kernel, allowing for a starvation-free (and hence, deadlock-free) inter-workgroup barrier by restricting the number of workgroups according to this estimate. We implement this idea by adapting an existing, previously non-portable, GPU inter-workgroup barrier to use OpenCL 2.0 atomic operations, and prove that the barrier meets its natural specification in terms of synchronisation.

We assess the portability of our approach over eight GPUs spanning four vendors, comparing the performance of our method against alternative methods. Our key findings include: (1) the recall of our discovery protocol is nearly 100%; (2) runtime comparisons vary substantially across GPUs and applications; and (3) our method provides portable and safe inter-workgroup synchronisation across the applications we study.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords GPU, OpenCL, barrier, synchronisation, portability

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984032

```
kernel unsafe(global int *flag, int A, int B) {
 if (get_group_id(0) == A)
   while(*flag != 1);

 if (get_group_id(0) == B)
   *flag = 1;
}
```

Figure 1: A non-portable kernel that potentially deadlocks

1. Introduction

With the growing use of GPUs in dynamic parallel applications, methods that fully exploit existing GPU hardware in the face of varying amounts of available parallelism have become crucially important. Typical large-scale applications in this area—graph analysis [7], particle partitioning [6], and software rendering pipelines [30]—share their computation across multiple *workgroups* (i.e. groups of threads, called *blocks* in CUDA [23]).

Execution barrier synchronisation, where a thread waits at a barrier until all threads reach the barrier, is a popular method for inter-thread communication because (1) it efficiently and simultaneously aligns the computations of threads, and (2) it makes pre-barrier memory updates available to all threads, implementing a well-specified and simple memory consistency model.

Unfortunately, existing GPUs and their associated programming languages do not support *inter-workgroup* barriers. This is because a GPU program (known as a *kernel*) can be configured with many more workgroups than the underlying hardware can execute concurrently. Execution of large numbers of workgroups is achieved in any "occupancy-bound" fashion, by delaying the scheduling of some workgroups until others have executed to completion. As a consequence, traditional fair scheduling guarantees associated with CPU threads are *not* provided between workgroups executing a kernel [11].

As an example, consider the kernel in Figure 1. This kernel accepts a pointer to a global variable, `flag` (initialised to 0), and two integers, `A` and `B`. The workgroup with id `B` (obtained through the OpenCL primitive `get_group_id(0)`) writes `1` to `flag`, while the workgroup with id `A` spins waiting for workgroup `B` to write to `flag`. However, the GPU execution model is licensed to postpone execution of workgroup `B` until execution of workgroup `A` has completed, which will lead the kernel to deadlock due to starvation.

This idiom of one workgroup waiting on another is at the heart of an inter-workgroup barrier. When executing a barrier instance, each workgroup will wait on all other workgroups to reach the same barrier instance. If a single workgroup is blocked by the occupancy-bound execution model, then the barrier execution will deadlock due to starvation. This behaviour is easy to observe on current GPUs, e.g. a traditional CPU software barrier [12, ch. 17], ported to synchronise across workgroups in OpenCL, leads to deadlock on an AMD Radeon R7 GPU when used in a GPU kernel configured with more than 4096 threads (with 256 threads per workgroup), as this exceeds the occupancy of the GPU for a kernel realisation, i.e. the number of workgroups that can execute the binary version of the kernel concurrently on the GPU. We abbreviate this concept as *occupancy of the GPU* for the remainder of this paper.

Current approaches Existing GPU applications that require inter-workgroup barrier synchronisation rely on either the *multi-kernel* or the *occupancy assumption* method.

In the multi-kernel method, an application is manually split into multiple kernels, with a transition from one kernel to another each time an inter-workgroup barrier is required. The transfer of control from the GPU to the CPU host between kernels provides the barrier semantics implicitly. This method is portable, making no assumptions about concurrent execution of workgroups. A drawback is that interaction between the GPU and CPU can be expensive, due to the overhead associated with kernel launch and because the contents of registers and local memory (*shared memory* in CUDA) do not persist across kernel calls, and thus cannot be reused.

The alternative *occupancy assumption* approach (studied in related work [11, 32]), employs a traditional software execution barrier. Starvation (and hence deadlock) is avoided through *a priori* knowledge about the number of workgroups that can be scheduled concurrently for a particular kernel and GPU. This avoids the efficiency problems of the multi-kernel approach, but is *non-portable*, since workgroup occupancy varies dramatically between GPUs depending on hardware resources (e.g. the number of compute units), and on a per-kernel basis according to the resources (such as registers and local memory) that a workgroup requires to execute the kernel. These resources are in part determined by the compiler and thus may vary between compiler versions.

In a related vein, OpenCL also provides *nested parallelism* [17, pp. 32–33] (or *dynamic parallelism* in CUDA [23, app. C]) where GPU threads can themselves launch a new kernel and then wait until the new kernel is finished. While this feature may be useful for applications examined in this work (e.g. applications with irregular parallelism), the high level idiom of nested parallelism (i.e. fork-and-join) is different enough from barrier synchronisation (i.e. synchronise existing threads) that we do not consider it in this paper.

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

Memory consistency An inter-workgroup barrier must also provide memory ordering properties: threads must observe up-to-date memory values during post barrier execution, and data-races between accesses separated by the barrier must be forbidden. The barrier must be implemented using sufficient synchronisation constructs, such as atomic operations and memory fences, to ensure these properties.

Our contributions Our overall contribution is to show that portable inter-workgroup barrier synchronisation can be successfully implemented, if both the memory consistency model *and* the execution model are considered.

Underpinning our contribution is a novel *occupancy discovery* protocol that provides a (safe) estimate on the number of occupant workgroups dynamically is, at the beginning of a kernel execution (a precise definition of *occupancy* is given in Section 2.3). The protocol can then be used to set up an execution environment such that the remainder of the kernel is only executed by the workgroups found to be occupant. Because the number of workgroups that execute the kernel is dynamic, depending on the discovered occupancy, this execution environment requires that kernels are agnostic to the number of executing workgroups (discussed in Section 3).

Because occupant workgroups exhibit traditional fair scheduling guarantees, they can reliably participate in an inter-workgroup barrier. In this context, we extend an existing barrier implementation to use the atomic operations of OpenCL 2.0. The memory ordering properties of these instructions are well-defined and allow for our barrier implementation to be formally reasoned about. In particular, we (1) provide a formal specification of the memory ordering properties to be provided by an abstract inter-workgroup barrier, and (2) prove that our concrete implementation honours the specification.

To assess portability, we evaluate our method across eight GPUs spanning four vendors (shown in Table 1). We first assess the recall, i.e. the percentage of relevant instances retrieved, of the estimate returned by our discovery protocol. We show that using heuristics, the recall is almost 100% (i.e. the occupancy estimate almost perfectly matches the occupancy bound). We then examine the Pannotia [7] and Lonestar-GPU [5] benchmarks, which currently achieve inter-workgroup synchronisation using the multi-kernel and occupancy assumption methods, respectively. We adapt all relevant applications from each suite to use our protocol/barrier combination and compare runtimes with the original implementations. In all cases, our approach enables portable execution as expected, and although performance benefits vary between GPUs, in many cases we observe good speedups by modifying applications that originally used the multi-kernel approach to use our inter-workgroup barrier.

To summarise, our main contributions are:

- We develop an *occupancy discovery* protocol, which dynamically computes a safe estimate of the workgroup occupancy for a given GPU and kernel (Section 3).
- We augment an inter-workgroup barrier (given in [32]) to exploit the dynamic workgroup occupancy discovered by our protocol. By using OpenCL 2.0 atomic operations, we are able to prove intuitive memory ordering guarantees (Section 4).
- We evaluate our occupancy discovery protocol, showing that it is portable across eight GPUs spanning four vendors (Table 1), achieving near-perfect occupancy estimates (Section 5.1).
- We evaluate the performance benefits of using our protocol/barrier combination against the multi-kernel and occupancy assumption methods for inter-workgroup synchronisation. Results vary across GPUs and applications, but in some cases our method provides a noticeable runtime speedup, up to 2.34× (Sections 5.2 and 5.3).

A web page containing supplementary material (i.e. experimental results, implementation details and additional formalisation for the barrier correctness) is located at: `http://multicore.doc.ic.ac.uk/projects/gpubarrier/`.

2. Background

We provide an overview of OpenCL and its memory model (Section 2.1) and describe the key prior work on which our contribution rests: the inter-workgroup barrier of Xiao and Feng [32] (Section 2.2), and the occupancy-bound execution model (Section 2.3).

2.1 OpenCL

An OpenCL program comprises *host* code, executed on the CPU, and *device* code, executed on a device (in our case a GPU). OpenCL supports a hierarchical execution model that mimics some of the specialised hardware it is intended to run on (GPU hardware in particular). Threads[1] are partitioned into disjoint, equally sized sets called *workgroups*.

An OpenCL *kernel* runs on a device, and is launched by the host code via API calls that allow the number of workgroups and threads per workgroup to be specified. Kernels are written in OpenCL C, which is based on C99. A kernel is expressed in SIMT (single instruction multiple thread) form, whereby all threads execute the same program but may query their unique thread identifiers to access distinct data and follow varying control paths.

An intra-workgroup execution barrier can be used for deterministic and consistent communication within a workgroup. To aid in finer-grained communication, including communication between workgroups, OpenCL provides a set of atomic read-modify-write instructions that allow a thread to atomically access and update a memory location. OpenCL provides two shared memory regions: local memory, which is shared only between threads in the same workgroup (called *shared memory* in CUDA); and global memory, which is shared between all threads on the device.

Execution environment The *execution environment* provides information about the ids and number of threads executing a kernel, accessed through the following functions (called workitem built-in functions [16, pp. 69-70]): `get_local_id`, a workgroup local id (unique and contiguous within workgroup); `get_group_id`, a workgroup id (the same value for all threads in a workgroup); and `get_global_id`, a global id (unique and contiguous for all threads); `get_local_size`, the number of threads per workgroup; `get_num_groups`, the number of workgroups; and `get_global_size`, the number of threads (across all workgroups). The execution environment is *static*: the number of threads and workgroups is fixed on kernel launch. Threads can query the execution environment to access contiguous unique data in a data-parallel program. In CUDA, the `threadIdx`, `blockIdx` and `blockDim` structs implement analogous functionality.

Memory model The OpenCL 2.0 memory model, based on that of C++11 [15], employs a *catch-fire* semantics, where races on regular variables lead to undefined behaviour. Atomic variables are provided to give semantics to code that would otherwise be racy. Synchronisation between threads can be achieved by associating a *memory order* with each atomic variable access. The memory orders relevant to this work are *release* (applied to store operations) and *acquire* (applied to load operations). An acquire load that reads a value written by a release store in a different thread creates a *happens-before* edge from the store operation to the load operation, i.e. the operations *synchronise*.

In OpenCL (and unlike C++11), an atomic access has an associated *memory scope* annotation, specifying a level of

[1] In strict OpenCL terminology, threads are called *workitems*; we opt to use *thread* due to its pervasiveness.

```
1  if (get_local_id() + 1 < get_num_groups()) {
2    while (flag[get_local_id() + 1] == 0);
3  }
4
5  barrier();
6
7  if (get_local_id() + 1 < get_num_groups()) {
8    flag[get_local_id() + 1] = 0;
9  }
```

(a) Master workgroup code

```
1  barrier();
2  if (get_local_id() == 0) {
3    flag[get_group_id()] = 1;
4    while (flag[get_group_id()] == 1);
5  }
6  barrier();
```

(b) Slave workgroup code

Figure 2: XF inter-workgroup execution barrier

the OpenCL execution hierarchy. This declares the intent to concurrently access the variable only within this level of the hierarchy, so that synchronisation is only provided within the given scope. The scope annotations relevant to this work are *workgroup* and *device*, allowing synchronisation between threads only in the same workgroup, and between arbitrary threads executing a kernel, respectively.

2.2 The XF Software Execution Barrier

Figure 2 illustrates a variant of the XF (Xiao/Feng) software execution barrier [32], a GPU inter-workgroup barrier provided in the CUB CUDA library [22]. Originally implemented in Nvidia's CUDA language, the XF barrier has been shown to offer high performance, exhibiting a low level of memory contention and avoiding read-modify-write instructions. The variant we show is ported to OpenCL and removes an intra-workgroup barrier we determined to be redundant.

The XF barrier uses a *master/slave* model, where one workgroup is selected to be the master (Figure 2(a)) and the remaining workgroups are slaves (Figure 2(b)). Function `barrier()` denotes an intra-workgroup barrier operation.

The slave workgroups start with an intra-workgroup barrier, to ensure that all threads in the local workgroup have arrived at the inter-workgroup barrier (slave line 1). A representative thread in the workgroup (with local id `0`) writes 1 to the workgroup's index in `flag`, an array of flags at least as large as `get_num_groups()`, to indicate that the workgroup has arrived at the inter-workgroup barrier (slave line 3). The representative thread then spins (slave line 4), waiting for the master workgroup to release the barrier. The remaining threads in the workgroup wait at the final workgroup barrier instruction (slave line 6) for the representative.

Each thread in the master workgroup takes responsibility for managing one slave workgroup: the workgroup with group id equal to the thread's local id plus one; one is added to the local id because the group with id 0 is the master workgroup and does not need to be managed. Each master thread spins until the workgroup it is managing has arrived at the barrier (master line 2). The master workgroup then performs a workgroup barrier (master line 5). Given that master threads manage all other (slave) workgroups, the completion of this workgroup barrier denotes that all threads across all workgroups have arrived at the barrier. Each master thread now releases the workgroup it is managing by setting that workgroup's flag to 0 (line 8).

The code of Figure 2 assumes that there are at least as many threads per workgroup as there are workgroups, but is easily adapted to cater for a larger number of workgroups by having each thread in the master workgroup manipulate the flags of more than one workgroup.

The XF barrier fails to directly provide portable inter-workgroup synchronisation for two reasons. First, because progress between workgroups is not guaranteed, the barrier is prone to deadlock due to starvation. Running the XF barrier on the chips of Table 1 with 1024 workgroups (each with 256 threads) causes deadlock for *every* GPU; reducing the number of workgroups to 128 results in deadlock for all chips except 980 and R9; while reducing the workgroup count to 2 avoids deadlock in all cases. The XF barrier was originally evaluated on GPUs where the number of concurrently executing workgroups was known *a priori* so that deadlock could be avoided [32]. Secondly, some GPUs have been shown to have relaxed memory models, where memory operations may appear to execute out of order [1]. The original XF barrier implementation, presented in CUDA (which lacks a rigorous memory model) does not formally take account of memory ordering properties.

2.3 Occupancy-Bound Execution Model

OpenCL does not currently specify a formal execution model for inter-workgroup interactions, and the behaviour of programs with such interactions is cautioned against in the standard [17, p. 31]: *"A conforming implementation may choose to serialize the workgroups so a correct algorithm cannot assume that workgroups will execute in parallel. There is no safe and portable way to synchronize across the independent execution of workgroups since once in the work-pool, they can execute in any order."*

In previous work [11, 31], it is suggested (and supported by empirical evidence) that GPUs provide an *occupancy-bound* execution model for inter-workgroup interactions. Here we more formally define the occupancy-bound execution model. Our definition is limited to the execution of a single kernel. We also assume that non-compute functionality (e.g. graphics) is disabled by kernel execution. This final assumption is consistent with our empirical observations, e.g. the OS graphics layer becomes unresponsive when our kernels are executed.

A workgroup is *occupant* if it has executed at least one instruction using the GPU's hardware resources, and has not yet completed kernel execution. The occupancy-bound execution model requires the following conditions, relating to occupant workgroups, to hold:

- **No indefinite preemption**: An occupant workgroup is guaranteed to eventually be scheduled for further execution on the GPU, regardless of the behaviour of other workgroups. Consequently, two workgroups that are simultaneously occupant can participate in blocking communications with each other that require fair scheduling (e.g. spin-locks).
- **Utilisation**: There is a constant $N > 0$, the *occupancy bound*, such that if m workgroups are occupant, with $m < N$, if there exist $k > 0$ workgroups that have not yet commenced execution, one of these workgroups will eventually become occupant. Additionally, no more than N workgroups can occupant. Consequently, a blocking communication that requires up to N workgroups to be simultaneously occupant (but makes no assumptions about the order in which they become occupant) can be used without fear of starvation-induced deadlock.

The occupancy bound N depends on the amount of hardware resources available. This clearly depends on the GPU, but also the amount of resources required by the target kernel. A kernel that uses a large amount of local memory or registers will require more resources, which may lower N. Because the resources used by a kernel may depend on details of compilation (e.g. register allocation), N also depends on the OpenCL framework.

The persistent thread model The persistent thread model is a GPU programming model that allows kernels to exploit fair scheduling guarantees provided by the occupancy-bound execution model. To use the persistent thread model, programmers must ensure that they launch kernels with at most Q workgroups, where Q is less than or equal to the occupancy bound N (sometimes referred to as the *maximal launch* [11]). Under this restriction, all workgroups will eventually be occupant, so idioms that require fair scheduling across workgroups, e.g. inter-workgroup barriers, can be used. Applications that use the persistent thread programming model currently use occupancy assumption methods to determine N. A main contribution of our paper is a method for programmatically determining a safe estimate for N.

One empirical validation of the occupancy-bound execution model is the existence of an occupancy bound N such that an inter-workgroup barrier succeeds when executed with N workgroups but deadlocks when executed with $N + 1$ workgroups. Our work explicitly finds occupancy bounds across a range of GPUs in Section 5.1. Many instances of previous work (which use the persistent thread model) also acknowledge this bound [5, 6, 11, 13, 19, 21, 25, 30–32], adding to the empirical evidence for this execution model.

3. Occupancy Discovery

We now detail our *occupancy discovery* protocol for dynamically determining a safe estimate of the occupancy bound for a given GPU and kernel. Recall that the occupancy-bound execution model guarantees existence of an occupancy bound N for a given GPU and kernel such that N workgroups can be simultaneously occupant during execution, and are guaranteed to be fairly scheduled. Given that N is unknown, our aim is to dynamically discover an estimate n with $0 < n \leq N$, i.e. a lower bound for N. The n discovered workgroups may then proceed to successfully complete computation that requires forward progress between workgroups using a persistent thread style programming model.

```
lock(mutex);
if (poll_open){
   M[get_group_id()] = count;
   count++;
   unlock(mutex);
} else {
   unlock(mutex);
   return NON_PARTICIPATING;
}
```
polling phase

```
lock(mutex);
if (poll_open) {
   poll_open = false;
}
unlock(mutex);
return PARTICIPATING;
```
closing phase

Figure 3: Occupancy discovery protocol executed by a single representative thread per workgroup

To achieve this, all workgroups execute the discovery protocol at the start of the kernel, prior to any other computation. In the protocol, each workgroup executes a routine that returns a value indicating whether the calling workgroup is *participating*. A workgroup is participating if and only if the workgroup commences execution of the protocol before any workgroup has finished executing the protocol. By definition, participating workgroups are simultaneously occupant. The occupancy-bound execution model thus guarantees that participating workgroups are fairly scheduled, and the total number of participating workgroups is a lower bound for N.

Workgroups found to be non-participating immediately exit; workgroups found to be participating continue the kernel computation. Thus computation occurring after the discovery protocol (what we call the *main* kernel computation) can assume fair scheduling of workgroups.

Under this scheme, the native workitem built-in functions (see Section 2.1) may no longer provide contiguous unique values for threads executing the main kernel computation. For example, if the discovered participating workgroups do not have contiguous native ids then they cannot use these ids to access contiguous unique data. To overcome this problem, the occupancy discovery protocol constructs a new, dynamically determined, execution environment with replacements for the workitem built-in functions that do satisfy the contiguous unique property for participating workgroups.

3.1 Implementation

Figure 3 shows the discover protocol implementation. There are four variables located in the global memory region (recall from Section 2.1 the global memory region is shared across workgroups) : `count`, an integer to record the number of participating groups (initially 0); `poll_open`, a boolean recording whether the poll is open (initially `true`); `mutex`, an object that provides mutual exclusion through `lock` and `unlock` functions (initially `unlocked`); and `M`, an array that records intermediate values of `count`. All protocol variables are required to be initialised prior to the protocol execution (e.g. by a separate kernel or via OpenCL host-to-device memory copies).

The protocol is split into two phases, a *polling* phase and *closing* phase. Both phases are protected using `mutex`. For now we leave the implementation of `mutex` along with the corresponding locking functions abstract, here we only require mutual exclusion in locked regions; concrete mutex implementations are discussed in Section 5.1. The protocol is executed by one representative thread per workgroup.

In the polling phase, a thread checks whether the poll is open (line 2). If so, the thread marks its workgroup as *participating* by recording the current value of `count` in array `M`, at an index according to the thread's workgroup id, and incrementing `count` (lines 3-4); the thread then moves on to the closing phase. On the other hand, if the poll is closed, the thread exits the protocol, returning a non-participating flag (line 8).

A thread that successfully completes the polling phase (observing an open poll) then enters the closing phase (line 11). If the poll is still open, the thread closes the poll (lines 12-13). Only the first thread to enter the closing phase performs this action; future threads observe a closed poll. Either way, the thread returns a participating flag (line 16).

Because thread scheduling is non-deterministic, a single thread might execute the polling phase *and* closing phase prior to any other thread commencing execution of the protocol. This would lead to an estimated occupancy bound of 1. Clearly we would prefer to discover a tighter lower bound, especially if the true occupancy bound N is large. We show experimentally that a suitable choice of mutex implementation leads to tight estimates of N in practice (see Section 5).

It may be possible to add heuristics to this protocol to improve the recall of discovered workgroups. For example, it seems natural that a pause inserted between the polling and closing phase may increase the recall (this would give occupant threads more time to poll before the poll is closed). In this work we do not consider such a heuristic for two reasons: (1) we show that with the right mutex implementation we can achieve a very high recall without any additional heuristics (see Section 5) and (2) because OpenCL does not provide any portable pause or sleep function, the pause would have to be implemented as some kind of busy spin. The busy spin would have to be tuned per GPU in order to minimise overhead while maximising recall. In Section 5.4, we briefly discuss pilot experiments applying our occupancy discovery protocol to CPU implementations of OpenCL, in which we employ this spinning heuristic.

Function	Derivation
`p_get_num_groups()`	`count`
`p_get_group_id()`	`M[get_group_id()]`
`p_get_global_id()`	`p_get_group_id() * get_local_size() + get_local_id()`
`p_get_global_size()`	`count * get_local_size()`

Table 2: Occupancy discovery execution environment functions

```
__kernel native_environment(...) {
  int id = get_global_id();
  if (id < data_size) {
     //do one element of work based on id%
  }
}
```

(a) Native execution environment paradigm

```
__kernel participating_environment(...) {
  for (id = p_get_global_id();
       id < data_size;
       id += p_get_global_size()) {
          // do multiple elements of work based on the
          // the number of participating groups
  }
}
```

(b) Participating group execution environment paradigm

Figure 4: Illustration of code transformation required when going from the native execution environment to the participating group execution environment

Execution environment construction The variables `M` and `count` are used to construct a new execution environment in which only participating workgroups take part in computation. Because `count` is initialised to zero and incremented once by each participating workgroup, after the protocol execution `count` contains the number of participating workgroups. Additionally, the value of `count` observed by a participating workgroup prior to incrementing (line 3) is recorded in array `M` at an index corresponding to the id of the workgroup, providing a unique *participating workgroup id*, such that the sequence of participating workgroup ids is contiguous. that is contiguous and unique.

Table 2 defines the new execution environment functions. A thread can query: the number of participating workgroups (`p_get_num_groups`), its participating workgroup id (`p_get_group_id`), a contiguous unique id across all participating threads (`p_get_global_id`), and the total number of participating threads (`p_get_global_size`). Each function is analogous to a native OpenCL function (the function without the `p_` prefix), but the new functions consider only participating workgroups.

Execution environment constraints As participating workgroups are determined dynamically, the main kernel computation must be agnostic to the number of executing workgroups at launch time. This is in contrast to the native OpenCL execution environment, where the number of executing workgroups is fixed on kernel launch.

In the native OpenCL execution environment a kernel can be launched with enough workgroups such that each thread computes at most one piece of data. A kernel that uses the occupancy discovery execution environment must be able to dynamically adapt the per-thread computation based on the number of participating groups, which can be queried during kernel execution. The kernels we adapted to use occupancy discovery in our experiments (see Section 5) required only simple transformations to satisfy this constraint, which is similar to the program transformations required for the persistent thread model [11]. The code of Figure 4(a) illustrates the OpenCL existing execution environment paradigm; threads compute at most one item of work depending on their id. The code of Figure 4(b) shows how the code of Figure 4(a) is adapted to use the participating group execution environment. That is, each thread performs a dynamic amount of computation based on the number of participating groups.

3.2 Properties and Correctness

Here we argue that our occupancy discovery protocol satisfies certain key properties required by client applications. Line numbers refer to Figure 3.

At least one participating workgroup To ensure that main kernel computation occurs, at least one workgroup must be identified as participating. Our protocol satisfies this requirement because the poll is initialised to open. This means that at least the first thread to enter the polling phase will mark itself as participating and go on to finish the protocol, returning `PARTICIPATING` at line 16.

Consistent participating count To provide participating threads with valid execution environment values, the protocol must ensure that, upon completion, all threads view the same value in `count` and that this value is equal to the number of workgroups that return `PARTICIPATING`.

Every thread returning `PARTICIPATING` increments `count` exactly once, because there are no loops in the protocol and the only path to returning `PARTICIPATING` (line 16) requires incrementing `count` (line 4). Furthermore, the value in `count` does not change once any thread returns from the protocol. There are two possible return points: participating (line 16) and non-participating (line 8). At both return points, the poll must be closed. In the case of a participating return, the poll has either been closed by the returning thread or by an earlier thread (lines 12 and 13). In the non-participating case, the poll was observed to be closed (line 2). Once the poll is closed, it remains closed (there is no place in the protocol that re-opens the poll). If the poll is closed, `count` cannot be modified.

Because all threads that return `PARTICIPATING` increment `count` once, and `count` does not change after a thread returns, all threads that return `PARTICIPATING` must observe `count` to contain the total number of threads that ultimately return `PARTICIPATING`.

Participating workgroups are simultaneously occupant The main purpose of the protocol is to create an execution environment that guarantees fair scheduling between workgroups, under the assumption of the occupancy-bound execution model. We now argue that workgroups identified as participating are indeed simultaneously occupant.

We show this property by counterexample. Let $\mathbb{P}$ be the set of workgroups that return `PARTICIPATING`. Because simultaneous occupancy concerns multiple workgroups, $\mathbb{P}$ must contain at least two workgroups. Assume that there exists a workgroup $w \in \mathbb{P}$ that is *not* simultaneously occupant with all of the other workgroups in $\mathbb{P}$ (that is, there is at least one workgroup in $\mathbb{P}$ that is not simultaneously occupant with w) and let $\mathbb{P}' = \mathbb{P} \setminus \{w\}$. There are now two possible cases to consider (under the occupancy bound execution model). Either w finishes execution before at least one workgroup in $\mathbb{P}'$ starts execution, or at least one workgroup in $\mathbb{P}'$ finishes execution before w begins execution.

First, we consider the case where w finishes execution before at least one workgroup $w' \in \mathbb{P}'$ starts execution. In order for w to finish execution, w must have executed the discovery protocol, returning `PARTICIPATING` (line 16). In order for w to have reached this line, the poll must be closed, either by w or an different participating workgroup (lines 12 and 13). Now w' eventually starts execution and begins executing the discovery protocol. However, because w has finished execution (and consequently the poll is closed), w' must observe a closed poll at line 2. Thus, w' must return `NON_PARTICIPATING` (line 8). Therefore w' is not a participating group, a contradiction.

The second case is where at least one workgroup $w' \in \mathbb{P}'$ finishes execution before w begins execution. The argument is exactly the same as the first case with w and w' being swapped; that is, w cannot return `PARTICIPATING` because w' (having returned `PARTICIPATING`) has closed the poll before w started execution. Thus w cannot be a participating group, a contradiction.

4. Inter-workgroup Barrier

We now present our inter-workgroup barrier, which augments the XF barrier with two missing features: (1) atomic instructions necessary for proper memory ordering properties and data-race freedom, and (2) an execution environment that ensures fair scheduling between workgroups. We describe its implementation, and give an overview of a hand proof of its memory-ordering properties.

4.1 OpenCL 2.0 Memory Model Primer

The OpenCL memory model defines the behaviour of concurrent memory accesses, including the atomic accesses and workgroup barriers used in our inter-workgroup barrier. The memory model is *axiomatic*: program behaviours are represented as sets of complete executions, each a graph of the memory events of one path of control flow with relations representing ordering in the execution.

The memory model has two phases. The first finds *consistent executions* by filtering prospective program executions: imposing a set of constraints on *happens-before*, hb, a relation on events that collects together thread-local program order and inter-thread synchronisation. The second looks for *data-races* in the consistent executions, defined as an absence of happens-before between conflicting non-atomic accesses to a single variable. If even a single race exists in a single consistent execution, the entire program is given undefined behaviour. Otherwise, the consistent executions represent the program's behaviour.

Our implementation relies on happens-before edges created between threads as demonstrated by the dashed arrows in the two consistent execution shapes of Figure 5.

Figure 5(a) presents an execution with intra-workgroup barrier synchronisation. Intuitively, a given barrier call gives rise to barrier entry and exit events that span the threads of a workgroup. A barrier entry and exit event on the same thread are ordered by *program order*. Program order induces happens-before between events on the same thread, drawn as solid vertical arrows. For each intra-workgroup barrier call, its associated events are collected into a barrier instance, signified by the surrounding dashed box. Each barrier entry synchronises with every exit in the same instance. We call these synchronisation edges a *barrier web*.

In the example of Figure 5(b), called *message passing*, one thread writes to x and then y, and another reads from y and then x. In the absence of synchronisation, the non-atomic accesses to x would form a race. However, for threads in either the same or different workgroups, a device-scoped acquire read such as c that reads from a device-scoped release write such as b, results in the creation of a happens-before edge, drawn as a dashed arrow in Figure 5(b). Happens-before is transitive, and the a-to-d edge avoids a data race on x, and forces d to read from a.

4.2 Implementation

The original XF barrier implementation is given in Figure 2. As discussed in Section 2.2, it has an *arrival* phase, where the master waits for every slave to announce its presence at the barrier, and a *departure* phase where the master releases the slaves from the barrier. Concurrent non-atomic accesses to the flag array would lead to data races in OpenCL. As a consequence, the XF barrier has undefined behaviour. Moreover, the purpose of each phase is to *synchronise*, first from the slave threads to the master, and then from the master to the slave threads. Non-atomic accesses do not provide this sort of synchronisation.

To realise the intended behaviour of the XF barrier in OpenCL 2.0, we augment the original implementation with atomic instructions. We declare `flag` as an array of atomic integers and we make all accesses release and acquire device-scoped atomics, avoiding races and inducing synchronisation. More precisely, slave line 3 and master line 8 become device-scoped release stores, and master line 2 and slave line 4 become device-scoped acquire loads.

Because the barrier covers only participating groups, we replace its thread id functions to use the execution environment provided by our discovery protocol (see Table 2): `get_num_groups` becomes `p_get_disc_groups` (master lines 1 and 7), returning the number of participating groups, and `get_group_id` becomes `p_get_group_id` (slave lines 3 and 4).

4.3 Barrier Specification

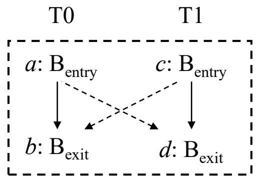
We provide a generic barrier specification, parameterised by an OpenCL scope and integrate it with the formal OpenCL memory model of Batty et al. [3]. The OpenCL workgroup barrier primitive behaves according to our specification instantiated with workgroup scope, closely following OpenCL [17, p. 53], whereas our inter-workgroup barrier behaves according to the specification instantiated with device scope. This symmetry suggests that the device-scoped barrier specification should be natural to OpenCL programmers familiar with the workgroup barrier.

OpenCL programs are required to be free from *barrier divergence* [16, p. 97]. Barrier divergence occurs when either (a) two workitems in the same workgroup reach syntactically distinct intra-workgroup barrier statements, or (b) an intra-workgroup barrier statement appears in a nest of loops, and two workitems reach the barrier statement having executed different numbers of iterations for at least one enclosing loop. See Collingbourne et al. for a formal definition [8].

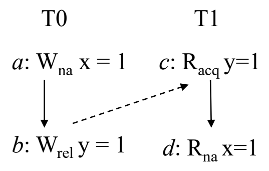
Kernels that exhibit barrier divergence have undefined behaviour, so we restrict our approach to kernels that are divergence free. We assume that two properties follow from barrier divergence-freedom: all barrier instances cover all threads at the workgroup scope and all participating threads at device scope, and no barrier instance links barrier events from within the XF barrier to those outside. We define the synchronisation that the barrier provides, and we declare the definition of faulting behaviour to be out of scope.

The specification requires two additions to the formal model of Batty et al.: new machinery to generate prospective executions with barrier events from programs, and new happens-before edges in the memory model.

Memory-model barrier specification A thread executing a dynamic instance of a barrier gives rise to two program-ordered events on each thread in its domain (workgroup or device): a barrier entry event (B_{entry}) followed by a barrier exit event (B_{exit}). The events of each barrier instance are all related by the barrier instance relation, bi, which covers the threads of the workgroup or device, respectively. We capture the new synchronisation with a relation called barrier



(a) Barrier web



(b) Release acquire message passing

Figure 5: Two OpenCL synchronisation patterns used by our inter-workgroup barrier

synchronises-with, $\mathsf{sw_B}$, made up of edges from each barrier entry event to each exit within a given barrier instance:

$$\mathsf{sw_B} = (B_{entry} \times B_{exit}) \cap \mathsf{bi}$$

The $\mathsf{sw_B}$ edges arising from a single barrier instance are precisely the edges in the barrier web of Figure 5(a). We call the dashed box surrounding the web an *instance-box* — in future we elide the web, drawing only the box. Intuitively, the web ensures that any access preceding a barrier happens-before any access following any barrier in the same instance, avoiding races between prior and following accesses.

4.4 Correctness of the Inter-workgroup Barrier

Our correctness proof reuses the *library abstraction* method of Batty et al. [2], and consists of two parts: the extension of library abstraction to work with the OpenCL memory model, adding scoped atomics and barriers, and the application of extended library abstraction to show that our implementation behaves according to our specification. We leave to the supplementary material the technical details of the extension of library abstraction to OpenCL, and focus on the properties that must be established for the soundness of our implementation, with a more detailed treatment in Appendix A.

Progress guarantee We rely on a form of progress: a property we assume from the occupancy-bound execution model and therefore our participating group environment. Our implementation uses spin loops that wait for a write on another thread. If they were to repeatedly read from older writes, the barrier would hang. To avoid this, we prohibit an infinite sequence of happens-before-ordered reads from failing to see a write from another thread.

Abstraction of the inter-workgroup barrier We take the augmented XF barrier as our implementation, although library abstraction would apply equally well to other implementations of the barrier. We take the abstract device-level barrier described in Section 4.3 as our specification. Neither our implementation nor our specification can exhibit races because each only uses barriers and device-level atomic accesses, and a race requires at least one non-atomic access.

Library abstraction requires us to establish equivalence of the inter-thread synchronisation generated by the implementation and the specification, and guarantees that the behaviour of the implementation will match the behaviour of the specification in any program that uses the barrier.

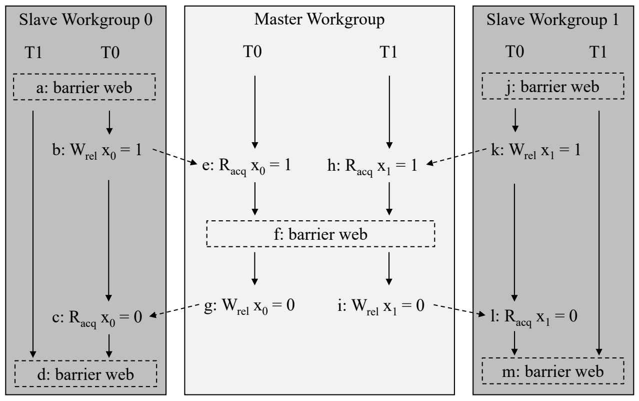


Figure 6: Idiomatic execution of XF barrier

The crux of the proof involves looking at a single barrier call across all threads. There are four cases to be considered for call-return pairs: same-workgroup, master-slave, slave-master, and slave-slave, where in the latter three cases the master and slave are in different workgroups. In the specification, the barrier web covers all of the threads belonging to participating workgroups identified by the discovery protocol, and creates synchronisation between every pair of threads. For each of the four cases, we must show that the XF barrier replicates this synchronisation.

We write down an *axiomatisation* of the events that make up an execution of the barrier: these event graphs coincide with the various paths of control flow through the XF barrier.

We consider the synchronisation made in a consistent execution, starting by ruling out the execution of the master whose events read repeatedly from the same write by noting that the first thread of each workgroup unconditionally writes, violating our progress assumption. Then we exclude any slave from repeatedly reading from the same write in its loop by noting that some master thread writes to the location. The remaining axiomatisations match the events listed vertically in Figure 6.

It is left to show that the implementation produces synchronisation in each of the four cases. We argue that it does so by identifying synchronisation of the two varieties presented in Figure 5 in the execution of the augmented XF barrier, and recalling that happens-before is transitive.

same-workgroup Whether the call and return reside on the master workgroup or a slave, in Figure 6 there is an instance box across the workgroup. The happens-before edge is ensured by the barrier web of a workgroup-scoped barrier as in Figure 5(a).

slave-master As the master breaks out of its loop, it must read the slave's write of the flag array. The device-level release and acquire synchronise as in Figure 5(b), creating a happens-before edge from one thread on the slave to one on the master. In Figure 6, the preceding barrier on the slave, and the following one on the master then complete the happens-before edge between the slave's call and the master's return.

master-slave Similarly the slave breaks out of its loop by the reading of the master's write creating synchronisation that is extended by the barriers on the master and slave.

slave-slave In this case, a slave synchronises with the master, and then the master synchronises with a slave. The prior, intervening, and following barriers complete the call-to-return happens-before edge.

The proof concludes by applying the abstraction theorem, extended to OpenCL to establish that the memory behaviour of the augmented XF barrier matches our natural device-scoped barrier specification.

5. Empirical Evaluation

We use microbenchmarks to measure the recall of the occupant workgroup estimate provided by the discovery protocol with respect to the occupancy bound (Section 5.1). We then examine several benchmarks that use the *multi-kernel* method for inter-workgroup synchronisation, adapting the benchmarks to use the discovery protocol/barrier combination and comparing the runtime of the two approaches (Section 5.2). We also consider several applications written using the non-portable *occupancy assumption* approach for inter-workgroup synchronisation. We modify these applications to be portable using our discovery protocol and measure the runtime overhead caused by the discovery protocol and associated execution environment (Section 5.3). Finally, we report on pilot experiments where we test our protocol on CPU implementations of OpenCL (Section 5.4).

The chips we consider (Table 1) all support the OpenCL 2.0 memory model except for the 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) to provide Nvidia specific memory fences. Our ARM implementation uses OpenCL 1.1 memory fence instructions. While our implementations come with no proof of correctness, we are conservative with our fence placement and encounter no issues in our experiments. Our custom implementations can be found in the supplementary material. These implementations should be seen as temporary until OpenCL 2.0 is more widely supported.

To preface these experimental results, we note that even though the occupancy-bound execution model is not officially endorsed by OpenCL, our experiments indicate that the model is supported by all the GPUs we considered (Table 1). Namely, the use of our inter-workgroup barrier never led to a deadlock and we were always able to find an occupancy bound. It appears that the occupancy-bound execution model is a useful *de facto* property of devices that support OpenCL, so we believe there is a case for incorporating the model officially in OpenCL, perhaps as an extension.

5.1 Recall of Discovery Protocol

As discussed in Section 3, the occupancy discovery protocol identifies a *subset* of occupant workgroups for a given GPU and kernel. It is desirable for the discovered occupancy to be as close as possible, and ideally equal, to the occupancy bound for the GPU and kernel. We use microbenchmarks to experimentally assess the recall of the occupancy discovered by our protocol, experimenting with two concrete mutex implementations and four kernel configurations.

Microbenchmarks The occupancy bound for a given GPU and kernel depends on the resources used by the kernel. To thoroughly evaluate the recall of our discovery protocol, we require a set of kernels with a variety of resource-usage characteristics. We consider two resources here: *local memory*, and *threads per workgroup*. Register pressure can also affect the occupancy bound of a kernel [26], but register allocation is a function of the compiler, and as such we do not have fine-grained control over this resource.

The microbenchmarks are based on a kernel that calls the discovery protocol to identify participating groups, which then execute one inter-workgroup barrier. The single barrier synchronisation in the microbenchmark can be configured to use participating workgroups, or all workgroups. This allows us to perform an unsafe search for the occupancy bound. This kernel is parameterised by (a) the amount of local memory that is allocated, and (b) the number of threads per workgroup. A single microbenchmark is an instance of the kernel with a particular choice for these parameters.

Mutex implementations In Section 3 we left details of how the mutex used by the discovery protocol is implemented abstract. We experiment with two mutex implementations, a *spin-lock* and a *ticket-lock*, which we implement using OpenCL 2.0 atomic operations. The mutex implementations are given in the supplementary material.

The *spin-lock* [27, p. 269] is implemented via a flag variable that can be *locked* or *unlocked*. To lock the mutex, a thread enters a spin loop that uses an atomic test-and-set operation to write *locked* to the flag and obtain the previous flag value. The thread exits the loop when the previous flag value is *unlocked*, indicating that the thread has successfully acquired the lock. A thread unlocks the mutex by writing *unlocked* to the flag.

The *ticket-lock* [27, p. 276] mutex uses two counters: a *ticket value* and a *servicing value*. To lock the mutex, a thread first atomically increments the ticket value, obtaining the old value as a *ticket*. The thread then polls the servicing value until it matches the thread's ticket, in which case the thread has acquired the lock. A thread unlocks the mutex

chip	#CUs	OB or mutex	11	L1	1W	LW
		OB	512.0	32.0	32.0	32.0
980	16	TL	512.0	32.0	32.0	32.0
		SL	24.9	4.0	3.3	3.1
		OB	192.0	12.0	24.0	12.0
K5200	12	TL	192.0	12.0	24.0	12.0
		SL	10.1	1.1	3.7	2.3
		OB	96.0	6.0	41.0	6.0
6100	47	TL	94.9	6.0	41.0	6.0
		SL	12.4	3.8	8.2	3.5
		OB	48.0	3.0	21.0	3.0
5500	24	TL	48.0	3.0	21.0	3.0
		SL	8.1	2.9	7.2	2.7
		OB	896.0	48.0	224.0	48.0
R9	28	TL	896.0	48.0	224.0	48.0
		SL	39.9	7.3	19.4	7.7
		OB	256.0	16.0	64.0	16.0
R7	8	TL	250.3	16.0	64.0	16.0
		SL	20.0	3.1	9.2	4.6
		OB	256.0	256.0	4.0	4.0
T628-4	4	TL	256.0	256.0	4.0	4.0
		SL	19.6	18.3	3.2	3.1
		OB	128.0	128.0	2.0	2.0
T628-2	2	TL	128.0	128.0	2.0	2.0
		SL	11.2	9.5	2.0	2.0

Table 3: Occupancy for each chip and microbenchmark for occupancy bound (OB), and the average discovered occupancy using the ticket lock (TL) and spin lock (SL)

by incrementing the servicing value. Unlike the spin-lock, where contending threads may obtain the mutex in any order, the ticket lock is *fair*: threads obtain the mutex in the order in which they request the mutex.

Experimental setup For a given GPU, we use the OpenCL framework to identify the maximum number of bytes of local memory that can be allocated (L), and the maximum number of threads per workgroup (W); these values vary between devices. We then consider four microbenchmark instances for the GPU, with allocated local memory set to either 1 or L bytes, and either 1 or W threads per work group. These instances capture extreme points of the resource parameter space. We execute each microbenchmark 50 times per chip and mutex implementation, recording the mean and standard deviation of the number of discovered workgroups.

To determine the occupancy bound of a microbenchmark, we disable the discovery protocol and attempt to execute the inter-workgroup barrier (unsafely) across all workgroups, searching for a value N such that the microbenchmark with the unsafe inter-workgroup barrier succeeds for N workgroups, but deadlocks with $N + 1$ workgroups.

Recall results Figure 7 shows the recall of occupancy discovery for both mutex implementations, as a percentage of the occupancy bound (y-axis, plotted using a log scale to account for the low recall of the spin-lock). For each GPU we show results for four microbenchmarks; label xy (with $x \in \{1, L\}$ and $y \in \{1, W\}$) indicates the resource parameters associated with a microbenchmark. Light and dark grey bars show recall for the spin-lock and ticket-lock mutexes, respectively. Standard deviation whiskers are shown for the spin-lock results, and omitted for the ticket-lock results, which exhibited negligible deviation. The black horizontal bars show the number of compute units (CUs) as reported by the OpenCL framework per chip (as a percentage of the occupancy bound). Table 3 shows the average concrete occupancy numbers for each chip, benchmark and mutex.

Our results show that with the ticket-lock mutex, our protocol almost always provides 100% recall (discovering the occupancy bound), showing suboptimal recall of still more than 95% in two cases: R7 and 6100 for the 11 microbenchmark. In contrast, the spin-lock performs poorly, both in mean discovered occupancy (often less than 50% of the occupancy bound) and consistency across runs, as shown by the high standard deviation.

We attribute the high accuracy of the ticket-lock-based discovery protocol to the fairness provided by this mutex implementation, which provides a high likelihood that many workgroups will enter the poll before any workgroup closes the poll (see Figure 3). In contrast, with the unfair spin-lock, there is a higher likelihood that a workgroup will execute the polling and closing phases in quick succession, closing the poll before many other workgroups enter the polling phase.

The black horizontal bars show that the CUs (reported by OpenCL) provide neither an accurate nor safe estimate of occupancy. For example, on 980, R9 and R7, the number of CUs is always lower than the occupancy bound, even for the extreme LW case where the maximum amount of local memory is allocated and the maximum number of threads per workgroup are requested. Thus, using CUs to estimate occupancy would under-utilise GPU resources. For T628-4, T628-2, and K5200, the number of CUs corresponds to the occupancy bound only with high resource parameters.

It is surprising to see that on 6100 and 5500 (Intel), the number of CUs can be *higher* than the occupancy bound. In these cases, using the number of CUs as an occupancy estimate would cause an inter-workgroup barrier to deadlock. The reason behind this, as reported by Mrozek and Zdanowicz [21], is that Intel reports the number of *execution units* (EU) as the number of CUs and it may require more than one execution unit to run a workgroup.

Mrozek and Zdanowicz also provide an Intel-specific formula for computing the thread occupancy (as opposed to workgroup occupancy) for kernels that (a) do not use any local memory and (b) are launched with large workgroup sizes. These constraints correspond to our microbenchmark 1W. The occupancy formula states that the number of occupant threads is found by multiplying three values together: the number of CUs (i.e. EUs in this specific Intel case), the threads per EU (obtained through device documentation [14]), and the SIMD size (queried through the OpenCL API). For example, on 5500 this gives $24 \cdot 7 \cdot 32 = 5376$ as the number of occupant threads. This is exactly the number of occupant threads we find for 5500 on microbenchmark

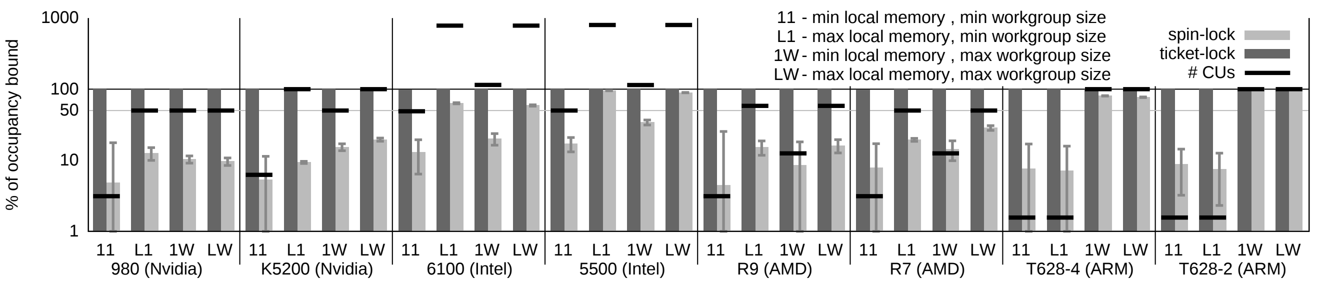


Figure 7: Discovered occupancy and number of compute units compared to the occupancy bound

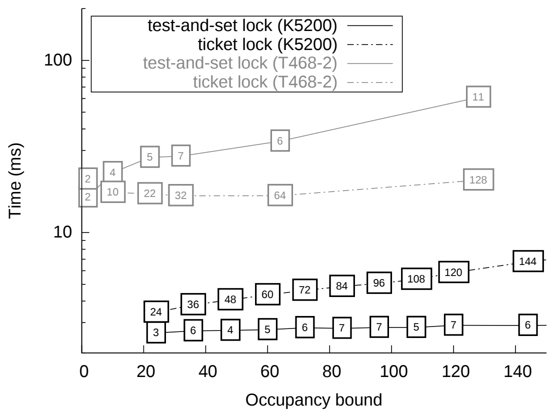


Figure 8: Protocol timing for K5200 and T628-2

1W; recall that to get the number of threads we multiply the number of workgroups with the number of threads per workgroup, in this case $256 \cdot 21 = 5376$.

No formula is given for when a kernel uses local memory or small workgroup sizes, although a reading of the documentation suggests some formula may exist based on the amount of local memory allocated per group of EUs. However, such a formula would be (a) difficult to deduce from the documentation, (b) have no guarantees of safety, and (c) only be valid until a new graphics architecture is released.

Timing results To measure the runtime cost of the discovery protocol, we also performed timing measurements. We benchmark the protocol timings per chip as a function of the occupancy bound. To vary the occupancy bound, we consider a series of microbenchmarks, instantiated with increasing resource utilisation that leads to a decreasing occupancy bound. We use the workgroup size as our resource value and consider multiples of eight up to the maximum workgroup size. Each microbenchmark is run 20 times and both the average discovered occupancy and the average time to run the discovery protocol is recorded. These microbenchmarks do not contain an execution of the inter-workgroup barrier as we are only interested in the protocol time.

Discovery protocol timing results for two representative chips, K5200 and T628-2, are shown in Figure 8. For these two chips, K5200 outperforms T628-2 in all cases; however, we are more interested comparing the different mutex strategies. The data labels are the discovered occupancy for each point. There are more data points for the K5200 as it supports a larger maximum workgroup size and hence has more values for resource parameters.

For chips K5200 (shown in Figure 8), 980, R9, and R7 the protocol using the ticket-lock is less performant than using spin-lock. This is consistent with what is generally reported for fair vs. unfair mutexes [27]. Conversely, for chips T628-2 (shown in Figure 8), T628-4, 6100 and 5500, the protocol using the ticket-lock is more performant than using spin-lock. This may be due to inefficient RMW operations inside the spin-lock loop. For all chips, we observe the protocol runtime increases with the occupancy bound, although the extent varies across chips.

Regardless of performance, the low recall of the spin-lock disqualifies it for use in our protocol (Section 5.1). Future work might consider augmenting the spin-lock, e.g. through busy waiting, to achieve a higher recall. For most chips, the protocol executed in 5–10ms depending on the occupancy bound. The exceptions are the embedded ARM chips, which took 20–100ms to execute the protocol (for either mutex).

5.2 Comparison With the Multi-kernel Approach

We now consider applications that originally used the multi-kernel paradigm, but which can be naturally expressed as a single kernel with inter-workgroup barriers. In these applications, several kernels are called from a host-side loop that exits when an application-specific *stopping criterion* is met. The kernels compute data that must be copied back to the host at the end of each loop iteration in order for the stopping criterion to be evaluated. An inter-workgroup barrier allows the computation to be expressed as one large kernel, with the host-side loop migrated to run on the GPU. This requires fewer kernel launches and less host/device data movement.

For such applications, we compare the performance of the application expressed in its original multi-kernel form, vs. as a single kernel using an inter-workgroup barrier.

Multi-kernel applications The Pannotia OpenCL applications benchmark the performance of graph algorithms containing irregular parallelism patterns [7]. We consider four of the applications that utilise the multi-kernel idiom: sssp (single source shortest path), mis (maximal independent set), color (graph colouring), and bc (betweenness centrality). The remaining applications in the Pannotia either do not use the multi-kernel idiom, or use multi-dimensional execution environments (where thread ids are assigned in multi-dimensional structure rather than a flat, linear structure), which our discovery protocol does not currently handle. We used our discovery protocol and inter-workgroup barrier to write a single-kernel version of each application.

The Pannotia applications are reported to have different performance characteristics depending on the input data sets to which they are applied [7]. We benchmark each application with all provided data sets. Each application comes with two data sets, with the exception of sssp, which has only one.

Experimental setup Because workgroup size can have a substantial impact on runtime and maximum occupancy [29], we first run a tuning phase to determine a good number of threads per workgroup. For each GPU, application, and input data-set, we run the application repeatedly with power-of-two workgroup sizes, ranging from 32 to the maximum workgroup size supported by the GPU (preliminary testing suggested that workgroup sizes below 32 did not perform well). We execute each combination 10 times and record the workgroup size that provides the fastest average runtime. We tune the original multi-kernel application and our adapted discovery protocol applications independently.

For both the multi-kernel and discovery protocol implementations, the application/data-set combinations are then executed 20 times with the workgroup size found during the tuning phase, and the average runtime (excluding file IO for data-sets) is recorded. Application runtime is longer for the ARM chips, which are designed to maximise energy-efficiency; we halve the number of iterations for these chips.

Results For each GPU, application and input, Figure 9 contains a bar plotting the average speedup associated with executing the single-kernel version of the application enabled by our inter-workgroup barrier compared with the original multi-kernel version. The shade and border of a bar indicate the associated application and input. For each GPU, the figure also shows the geometric mean (GM), median, maximum, and minimum speedups taken over all applications and inputs. All speedup values are given in Table 4.

We observe varied results. For Nvidia GPUs (980 and K5200), the inter-workgroup barrier and multi-kernel variants have similar runtimes. For Intel chips (6100 and 5500), the inter-workgroup barrier always provides a non-negligible speedup, with mean speedups of 1.36 and 1.28, respectively. The AMD results differ between chips: the inter-workgroup barrier always improves runtime on R9, while on R7 we observe three speedups, three slowdowns, and one case where performance is unaffected. On ARM, most results show that using our inter-workgroup barrier worsens runtime substantially, except for the bc application on the 128k data-set, which shows a large improvement.

The performance of the barrier is sensitive to the input for an application. For example, the inter-workgroup barrier on R7 accelerates the color application for the eco data-set, but slows this application down for the circ data-set.

5.3 Portability vs. Specialisation

We now turn our attention to the use of our inter-workgroup barrier to create *portable* versions of applications that previously relied on *a priori* knowledge related to occupancy, and the price one pays for deploying a portable version of an application vs. relying on assumptions about occupancy.

For this purpose, we study applications from the CUDA Lonestar-GPU benchmark suite, written in CUDA and thus originally targeting only Nvidia GPUs. Like Pannotia, the Lonestar-GPU applications operate on graph algorithms that exhibit irregular parallelism patterns [5]. Four Lonestar-GPU applications use a non-portable XF inter-workgroup barrier that (a) relies on assumptions about occupancy, and (b) does not consider formal memory model issues (in part due to the lack of an agreed formal memory model for CUDA). The Lonestar-GPU suite provides an occupancy estimation method that uses a CUDA-specific query function to assess the resources used by a kernel, but this estimate is not guaranteed to be safe, and is not portable.

The relevant Lonestar-GPU applications are: mst (minimum spanning tree), dmr (delaunay mesh refinement), sssp (single source shortest path) and bfs (breadth first search). The Lonestar-GPU and Pannotia sssp applications are fundamentally different, the former using task queues to manage the workload, and the latter using using linear algebra methods. Much like Pannotia, we use the multiple data sets provided for each application. The sssp and bfs applications have three input data-sets, mst and dmr have two each.

Portable and specialised OpenCL applications We have ported these four applications to OpenCL, replacing the barrier implementation with our memory model-aware barrier, and the non-portable occupancy estimation function with our portable occupancy discovery protocol. These versions of the applications should be portable (at least in terms of their barrier behaviour) across OpenCL-conformant platforms that exhibit the occupancy-bound execution model.

However, we speculate that there may be specific OpenCL platforms for which a developer might wish to trade portability for performance, using our memory model-aware barrier, but exploiting *a priori* knowledge about occupancy to avoid the overhead of running our discovery protocol. To investigate this trade-off, we created non-portable specialised variants of the four applications for each of our GPU platforms. These variants store pre-computed occupancy bound data, and launch kernels with maximum thread counts derived

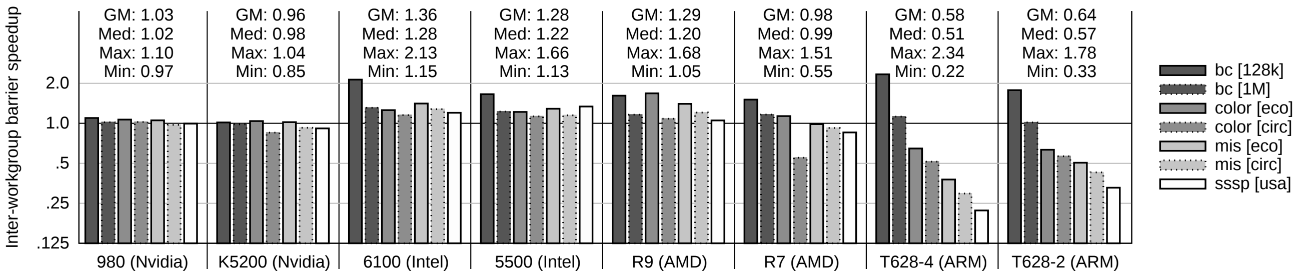


Figure 9: Runtime comparison of multi-kernel paradigm vs. inter-workgroup barrier

from this occupancy data. As well as avoiding running our discovery protocol, these specialised variants can use the native OpenCL thread id functions (see Section 3), which may allow compilers to make more optimisations.

Portability constraints and issues We encountered numerous hurdles during the process of porting the Lonestar-GPU applications to OpenCL, which we have written up as a separate experience report [28]. These issues include OpenCL compiler bugs, OpenCL driver issues, and program bugs that only manifested on certain chips. We were able to report and confirm many of these issues with industry representatives. In particular: we were unable to run dmr on non-Nvidia chips due to floating point issues, we were unable to run sssp with the usa dataset on Intel due to memory requirements, and we did not run sssp with the usa dataset on ARM or R7 due to prohibitively long runtimes (over 45 mins per execution). These issues are all *independent* of the inter-workgroup barrier and stem from portability issues in GPU programming languages.

Additionally some porting judgement was required in cases where CUDA constructs have no direct OpenCL analogue; for instance 1D texture memory is not supported for OpenCL 1.1 (the OpenCL version Nvidia supports) and warp aware primitives (e.g. warp shuffle) are not provided in OpenCL. For these issues, we used global memory instead of texture memory and re-wrote warp aware idioms to use intra-workgroup barriers instead.

Experimental setup We used the same tuning process described for the Pannotia benchmarks (Section 5.2) to find a suitable workgroup size per GPU, application and input data-set combination. To determine the occupancy bound for the specialised applications, we create specialised variants for each chip, application and input combination; these applications have the occupancy bound hard coded inside the application. The occupancy bound, N, is validated if the application terminates with N workgroups but not $N+1$. Much like in our microbenchmark experiments (Section 5.1), we find N through a trial-and-error binary search.

We run each chip, application, input combination for 20 iterations using both the portable and specialised variants. We use the workgroup size found in the tuning phase for both

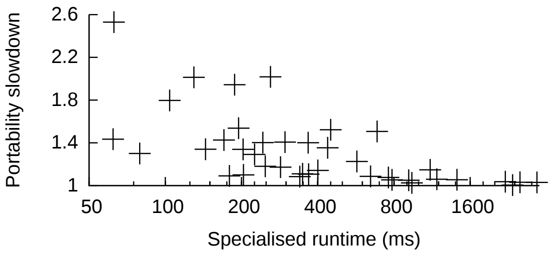


Figure 11: Portability overhead as runtime increases

variants; the specialised variants use the validated occupancy bound. The average runtime (excluding file IO for data-sets) is recorded. The runtime of these applications is such that we were able to run all iterations on the ARM chips.

Results Results showing the slowdown caused by portability are shown in Figure 10 (concrete slowdown numbers per chip and application are given in Table 4). For consistency across chips, we show only two data-sets on the bfs, mst and sssp applications. Because this graph measures slowdown, bars that are taller than the 1.0 mark indicate that the performance was degraded by using our portable constructs. We see that portability on Nvidia chips (for our OpenCL application variants) costs a mean of 1.3x slowdown compared with relying on occupancy assumptions. For Intel chips, the cost of portability is low, with a slowdown of 1.11x at worst. The results for AMD are split: R9 suffers the worst slowdowns across all the GPUs we consider, with a mean slowdown of 1.71x, while slowdowns associated with R7 are more modest, with a mean of 1.17x. Interestingly, we find that portability provides a *speedup* for the bfs and sssp applications on the ARM GPUs. In principle, we could investigate whether performance differences are due to the dynamic execution environment or the execution of the protocol; however we leave these experiments to future work.

The scatter plot of Figure 11 provides an overview over how the slow-down associated with portability relates to the overall runtime of the specialised application. Each cross refers to a particular application, input data set and GPU. The x-coordinate of the cross indicates the runtime of the

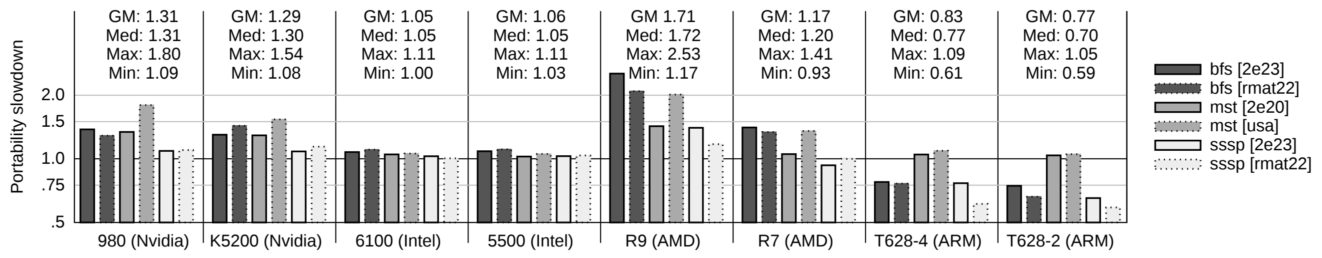


Figure 10: Portability slowdown for inter-workgroup barriers

specialised version of the application (averaged over 20 runs), and the y-coordinate shows the slow-down resulting from switching to a portable version of the application. The results suggest that portability has a constant cost, rather than a scaling cost: the longer the runtime of the specialised application, generally the smaller the overhead associated with portability. For example, across all GPUs, applications that took longer than 400 and 800 ms for computation in their original form showed a maximum 1.6× and 1.2× portability slowdown, respectively.

Overall, our results show that using portable constructs over specialisation can cause a slowdown, varied between applications and chips. However, understanding these results should consider that specialised applications may be *fragile* not only when ported, but also for the same GPU under compiler upgrades or code modifications. Anything that affects the amount of resources required by the kernel could potentially change the occupancy of the kernel and introduce deadlocks. Given these considerations, we believe that going forward, developers should look into optimising the portable constructs rather than taking the specialised approach.

It may be possible for vendors to modify their OpenCL frameworks to provide native support for the occupancy-bound execution model. For example, vendors could provide a way to launch a kernel without specifying the number of workgroups, and instead specify that only occupant workgroups should execute the kernel. It is possible that the runtime could determine occupancy bounds efficiently using low-level proprietary knowledge. It is likely then that the native execution environment functions could be used (allowing for any compiler or hardware optimisations around these native functions). We imagine that such support would reduce the portability slowdown we observe using our method.

5.4 OpenCL on CPUs

This work exclusively focuses on GPU platforms because today's implementation of OpenCL appear to implement a non-traditional execution model, the occupancy-bound execution model, which makes implementing a portable execution barrier non-trivial. On the other hand, execution barriers for CPU systems do not account for such execution models (e.g. see [12, ch. 17]). This is because many CPU concurrency frameworks allow for preemption and have a scheduler that, in practice, provides fair scheduling across all threads.

			occupancy after delay			
CPU chip	CUs	OB	0	10	100	1000
Intel i7-5600U	4	4	1.0	1.3	3.3	4.0
AMD A10-7850K	4	4	1.0	1.0	1.5	3.8

Table 5: Occupancy for two CPU chips; we show the occupancy bound (OB), and the average discovered occupancy (out of 50 runs) using the ticket lock with different amounts of delay between the polling and closing phases

Because OpenCL can be run on some CPU systems, we ran some pilot experiments using the occupancy benchmarks of Section 5.1 to investigate the execution models associated with CPU implementations. Our hypothesis was that we would not be able to find an occupancy bound; that is, an inter-workgroup barrier would be deadlock-free on CPUs for any (reasonable) number of workgroups.

We were thus surprised to observe an occupancy bound of 4 for the two CPU frameworks we tested: the Intel SDK for OpenCL applications (build 10094, and driver version 5.2.0.10094) and AMD APP SDK (version 2004.6), running on the Intel and AMD CPUs (respectively) detailed in Table 5. This means OpenCL implementations of inter-workgroup barriers can deadlock even on CPU systems if executed with too many workgroups. Because CPUs do not natively provide this behaviour, we hypothesise that the runtime must implement the occupancy-bound execution model on top of the native CPU scheduler; this would be straightforward, e.g. by storing workgroups in a queue.

We also found that the recall of our discovery protocol, even using the ticket lock, was poor on CPUs. To increase the recall we inserted a delay between the polling and closing phase of the protocol (see Section 3). The delay consists of a loop where the thread performs a lock and unlock function on the mutex. For the ticket lock, this allows other threads to queue in front of the spinning thread to poll. Our results (Table 5) show that a delay can increase the recall of the protocol on CPUs. In these experiments we did not observe any difference in the occupancy bound when varying the amount of local memory allocated (unsurprising, since

chip	barrier speedup for Pannotia							portability slowdown for Lonestar-GPU					
	bc		color		mis		sssp	bfs		mst		sssp	
	128k	1M	eco	circ	eco	circ	usa	2e23	rmat22	2e20	usa	2e23	rmat22
980	1.10	1.02	1.07	1.02	1.05	0.97	0.99	1.38	1.29	1.34	1.80	1.09	1.10
K5200	1.02	0.98	1.04	0.85	1.02	0.93	0.92	1.30	1.43	1.29	1.54	1.08	1.14
6100	2.13	1.31	1.26	1.15	1.41	1.28	1.20	1.08	1.11	1.05	1.06	1.03	1.00
5500	1.66	1.22	1.22	1.13	1.29	1.15	1.34	1.09	1.11	1.03	1.05	1.03	1.04
R7	1.62	1.16	1.68	1.08	1.40	1.20	1.05	2.53	2.09	1.43	2.01	1.40	1.17
R9	1.51	1.16	1.13	0.55	0.99	0.92	0.85	1.41	1.34	1.05	1.35	0.93	1.00
T628-4	2.34	1.12	0.65	0.51	0.38	0.30	0.22	0.78	0.76	1.05	1.09	0.77	0.61
T628-2	1.78	1.02	0.63	0.57	0.51	0.43	0.33	0.75	0.66	1.04	1.05	0.65	0.59

Table 4: Pannotia application speedups using the inter-workgroup barrier (higher is better), and Lonestar-GPU application slowdowns for using a portable vs. non-portable barrier (lower is better)

CPU platforms do not exhibit software-managed memory) nor the number of threads per workgroup.

These experiments show that synchronisation constructs (in our case an execution barrier) must account for the execution model of the framework in which they operate, not just the hardware on which they will be executed: CPU devices are adept at managing preemption between threads, yet the CPU OpenCL implementations we considered still exhibit an occupancy-bound execution model. Defining and reasoning about execution models and the synchronisation constructs that they allow promises to be a fruitful area of research for frameworks with native support for concurrency, e.g. OpenCL [17], C++ [15], and OpenMP [24].

6. Related Work

The basis for our work is the persistent thread model [11] and the XF software execution barrier [32]; we reason formally about our barrier implementation using techniques for C++11 concurrency abstraction [2]. Our empirical evaluation uses the Pannotia [7] and Lonestar-GPU benchmarks [5], which were originally designed to examine performance characteristics of irregular GPU workloads.

Irregular parallelism on GPUs The need for an inter-workgroup barrier arises due to irregular parallelism on GPUs; this has been examined with various approaches in several related works. Hower et al. propose a work-stealing programming idiom for irregular computations using shared concurrent queue data structures [13]. Orr et al. extend this work by proposing new primitive GPU synchronisation operations, allowing for more efficient work stealing [25]. Other approaches (e.g. [20]) use the multi-kernel method, but focus on data representations, and computations exploit low-level GPU-specifics (e.g. warp-level instructions).

GPU models Our library abstraction proof is based on a formalisation of the OpenCL 2.0 memory model [3]. Other notable models developed in prior work include several variants of the hierarchical-race-free model [10], and a formal model of a fragment of PTX (the compiler intermediate representation for Nvidia GPUs) [1].

Our occupancy discovery protocol is based on enabling the persistent thread model for GPUs [11]. Related works studying execution models in the context of the GPUVerify [4] and GKLEE [18] tools do not account for scheduling properties of workgroups. Gaster provides an execution model for intra-workgroup interactions, and describes how barriers can be implemented at this level [9].

Wu et al. present a persistent thread CU-centric programming model that exploits CU locality across workgroups [31]. To prevent over-saturating the hardware, they present a protocol in which some of the persistent threads immediately exit the kernel without performing any computation. This is similar to our discovery protocol where non-occupant workgroups immediately exit. Essentially both protocols circumvent the proprietary GPU scheduler to enforce a certain mapping between workgroups and CUs.

7. Conclusions

As GPUs and GPU applications grow in variety, so will the need for robust and portable synchronisation and programming idioms. Building on non-portable previous work, we have developed, analysed, and evaluated a GPU inter-workgroup barrier in a portable context for the first time. To achieve this, we use an occupancy discovery protocol to estimate the number of persistent threads, i.e. threads which have traditional fair scheduling guarantees. We then use the new OpenCL 2.0 memory model to create an inter-workgroup barrier with intuitive memory model properties.

We evaluated our discovery protocol through microbenchmarks, and benchmarked the runtime effects of the portable inter-workgroup barrier when compared to existing methods for inter-workgroup synchronisation on GPUs. While our experimental results show that the runtime behaviour of our barrier varies between chips, applications and even inputs, we have shown that our inter-workgroup barrier is semantically portable across a wide range of GPUs. In future work, we aim to merge our methods with work that explores GPU performance portability. We hope to identify ways for our methods to be portable both semantically and in terms of performance.

Acknowledgments

We are grateful to: Brad Beckman, Marco Cornero, Hugues Evrard, Hedley Francis, Thibaut Lutz, Marc Orr, Sven Van Haastregt, and John Wickerson for feedback and insightful discussions around this work, and the OOPLSA reviewers (paper and artifact) for their thorough evaluations and feedback which greatly improved this paper. This work was supported in part by an equipment grant from GCHQ, a gift from Intel Corporation, an EPSRC Impact Acceleration Award, the Royal Academy of Engineering, the Lloyds Register Foundation, NSF CCF 1346756 and ACI 1535032.

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] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, pages 235–248. ACM, 2013.

[3] M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL*, pages 634–648. ACM, 2016.

[4] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Trans. Program. Lang. Syst.*, 37(3):10, 2015.

[5] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *IISWC*, pages 141–151. IEEE, 2012.

[6] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *SIGGRAPH*, pages 57–64. Eurographics Association, 2008.

[7] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular GPGPU graph applications. In *IISWC*, pages 185–195. IEEE, 2013.

[8] P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *ESOP*, pages 270–289. Springer, 2013.

[9] B. Gaster. A look at the OpenCL 2.0 execution model. In *IWOCL*, pages 2:1–2:1. ACM, 2015.

[10] B. R. Gaster, D. Hower, and L. Howes. HRF-relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models. *Trans. Archit. Code Optim.*, 2015.

[11] K. Gupta, J. Stuart, and J. D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Proceedings of Innovative Parallel Computing*, InPar, pages 1–14. IEEE, 2012.

[12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[13] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous-race-free memory models. In *ASPLOS*, pages 427–440. ACM, 2014.

[14] Intel. The compute architecture of Intel processor graphics gen9, version 1.0, Aug. 2015.

[15] ISO/IEC. Standard for programming language C++, 2012.

[16] Khronos Group. The OpenCL C specification version: 2.0. `https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf`.

[17] Khronos Group. The OpenCL specification version: 2.0 (rev. 29), July 2015. `https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`.

[18] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *PPoPP*, pages 215–224. ACM, 2012.

[19] S. Maleki, A. Yang, and M. Burtscher. Higher-order and tuple-based massively-parallel prefix sums. In *PLDI*, pages 539–552. ACM, 2016.

[20] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *PPoPP*, pages 117–128. ACM, 2012.

[21] M. Mrozek and Z. Zdanowicz. GPU daemon: Road to zero cost submission. In *IWOCL*, pages 11:1–11:4. ACM, 2016.

[22] Nvidia. CUB, April 2015. `http://nvlabs.github.io/cub/`.

[23] Nvidia. CUDA C programming guide, version 7, March 2015. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[24] OpenMP Architecture Review Board. OpenMP application programming interface version 4.5, November 2015.

[25] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood. Synchronization using remote-scope promotion. In *ASPLOS*, pages 73–86. ACM, 2015.

[26] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *ASPLOS*, pages 407–418. ACM, 2013.

[27] Y. Solihin. *Fundamentals of Parallel Computer Architecture: Multichip and Multicore Systems*. Solihin Publishing, 2009.

[28] T. Sorensen and A. F. Donaldson. The hitchhiker's guide to cross-platform OpenCL application development. IWOCL, pages 2:1–2:12. ACM, 2016.

[29] Y. Torres, A. Gonzalez-Escribano, and D. Llanos. Understanding the impact of CUDA tuning techniques for Fermi. In *High Performance Computing and Simulation (HPCS)*, pages 631–639, 2011.

[30] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the GPU. In *HPG*, pages 29–37, 2010.

[31] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *ICS*, pages 119–130. ACM, 2015.

[32] S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In *IPDPS*, pages 1–12. IEEE, 2010.

A. Inter-workgroup Barrier: Verification

A.1 OpenCL Memory Model

We assume familiarity with the memory model of Batty et al. [3], which we simplify and restrict to programs using only device-scoped, release and acquire atomic stores and loads, together with non-atomics and workgroup barriers.

Program executions are graphs of memory-access events, captured by the tuple $(E, I, \mathsf{lbl}, \mathsf{thd}, \mathsf{wg}, \mathsf{sb})$, where: E is a set of event identifiers, lbl maps event ids to attributes (e.g., event scope or value), I is a set of non-atomic initialisation writes of 0, thd/wg relate events in the same thread/workgroup, sb captures program order over events, and bi is the barrier-instance relation of Section 4.3.

A tuple $(\mathsf{rf}, \mathsf{mo})$, called the *witness*, records where reads *read from* in rf, and *modification order*, mo, a total order of the writes at each atomic location. Together, an execution, X, and a witness, w, form a *candidate execution*, (X, w).

Base sets are derived from the attributes of lbl: call/return events (call/ret); reads/writes (R/W); releases/acquires (rel/acq); barrier entry/exit events (entry/exit); the location of reads/writes (loc); and events at non-atomic locations (nal). The loc relation, relating same-location events, is derived from lbl.

Happens-before and visibility are derived relations, simplified from previous work [3], and extended with the barrier semantics of Section 4.3. Here, $^+$ is transitive closure, ; is sequential composition, and $^?$ is reflexive closure. Happens-before, $\mathsf{hb} = (\mathsf{sb} \cup \mathsf{sw})^+$, per-location hb, $\mathsf{hbl} = \mathsf{hb} \cap \mathsf{loc}$, and visibility, $\mathsf{vis} = (\mathsf{W} \times \mathsf{R}) \cap \mathsf{hbl} \setminus (\mathsf{hbl}; [\mathsf{W}]; \mathsf{hb})$, are unchanged, but *synchronises-with* includes synchronisation across bi at wg and dv level:

$$\mathsf{sw} = \big(\mathsf{rf} \cap (\mathsf{rel} \times \mathsf{acq})\big) \cup \big(\mathsf{bi} \cap (\mathsf{entry} \times \mathsf{exit})\big)$$

The model is reduced: we have only global atomics, and no SC accesses or read-modify-writes.

$$\begin{aligned}
\textsc{Acyclicity} &= \mathsf{irreflexive}(\mathsf{hb}) \\
\textsc{Coherence} &= \mathsf{irreflexive}((\mathsf{rf}^{-1})^?; \mathsf{mo}; \mathsf{rf}^?; \mathsf{hb}) \\
\textsc{AtomicRF} &= \mathsf{irreflexive}(\mathsf{rf}; \mathsf{hb}) \\
\textsc{NonatomicRF} &= \mathsf{empty}((\mathsf{rf}; \mathsf{nal}) \setminus \mathsf{vis})
\end{aligned}$$

The top-level semantics of OpenCL has a *catch fire* design: a program that exhibits a consistent faulty execution is given undefined behaviour, $\mathbb{X}$. Fault-free programs behave according to the set of their consistent executions. Heterogeneous races, $\mathsf{f}_{\mathsf{hr}} = \mathsf{cnf} \setminus (\mathsf{hb} \cup \mathsf{hb}^{-1}) \setminus \mathsf{incl} \setminus \mathsf{thd}$, defined in terms of conflicts, $\mathsf{cnf} = (\mathsf{W} \times \mathsf{W}) \cup (\mathsf{W} \times \mathsf{R}) \cup (\mathsf{R} \times \mathsf{W})$, are faults. For $\mathcal{X}$, the pre-executions of a given program, we have:

$$\begin{aligned}
\text{allowed}(\mathcal{X}) \;=\; & \textbf{if } \exists X \in \mathcal{X}.\, \exists w.\, \text{faulty}(X, w) \textbf{ then } \mathbb{X} \\
& \textbf{else } \{X \in \mathcal{X} \mid \exists w.\, \text{consistent}(X, w)\}
\end{aligned}$$

A.2 Progress Guarantee

The inter-workgroup barrier verification relies on an assumption enforced by the occupancy-bound execution environment:

DEFINITION 1 (PROGRESS). *Given an execution X, and writes $w_1 \xrightarrow{\mathsf{mo}} w_2$ in X, any happens-before chain of reads, $(r_1 \xrightarrow{\mathsf{hb}} r_2 \xrightarrow{\mathsf{hb}} \ldots)$, where each r_i reads from w_1, i.e. $r_i \xrightarrow{\mathsf{rf}} w_1$, must be finite.*

A.3 Abstraction Theorem

We rework C/C++ *library abstraction* of Batty et al. [2] for OpenCL. An abstraction relation, $\sqsubseteq$, over library calls is shown to be *adequate*, i.e. to establish observational refinement under an arbitrary client. The approach relies on a *library-local semantics*, $[\![\mathcal{L}]\!]$, producing executions of library $\mathcal{L}$ with threads bound by call and ret events.

Non-interference We require *non-interference*, NONINTERF: library reads and writes, selected by lib, access a disjoint set of locations, LLoc, from those of the client. We extend NONINTERF to barriers: bi may not cross between the client and the library.

$$\begin{aligned}
&\forall a \in R \cup W.\, \mathsf{lib}(a) \iff \mathsf{loc}(a) \in \mathsf{LLoc} \,\wedge \\
&\forall (b_1, b_2) \in \mathsf{bi}.\, \mathsf{lib}(b_1) \iff \mathsf{lib}(b_2)
\end{aligned}$$

Library-local semantics The library-local semantics of the inter-workgroup barrier obeys a protocol: all threads must call the inter-workgroup barrier the same number of times. Together with NONINTERF, this implies a tightly constrained pattern for use of the barrier, captured by the *most-general client*, a special client that, when composed with the library to form a program, exercises all possible executions of the barrier. Here $|||^+$ is the parallel composition of any number of workgroups, $||^+$ is the composition of a number of threads to form a workgroup, and ^{n} denotes sequential repetition n times (we elide initialisation of LLoc to 0):

$$\text{MGC}(\mathcal{L}) = |||^+(||^+(\mathsf{barrier};)^n) \text{ for } n \in \mathbb{N}$$

We define the library-local semantics: $[\![\mathcal{L}]\!] = [\![\text{MGC}(\mathcal{L})]\!]$, with call and ret events bracketing each barrier call. We define $[\![\mathcal{L}, R]\!]$ as the execution of $\mathcal{L}$ under the MGC with arbitrary extension of hb with ret to call edges R.

Histories and the abstraction relation The history of execution X is the call and ret events of X, written $\mathsf{interf}(X)$, together with the projection of hb from call to ret, written $\mathsf{hbL}(X)$: $\mathsf{history}(X) = (\mathsf{interf}(X), \mathsf{hbL}(X))$.

We define *abstraction*, $\sqsubseteq$, over histories: $(A_1, G_1) \sqsubseteq (A_2, G_2) = (A_1 = A_2) \wedge (G_1 = G_2)$. We raise the definition of abstraction to libraries by employing the library-local semantics. For safe $\mathcal{L}_1$ and $\mathcal{L}_2$:

$$\begin{aligned}
&\mathcal{L}_1 \sqsubseteq \mathcal{L}_2 = \\
&\quad \forall R.\, \forall H_1 \in \mathsf{history}([\![\mathcal{L}_1, R]\!]).\, \exists H_2 \in \mathsf{history}([\![\mathcal{L}_2, R]\!]). \\
&\qquad H_1 \sqsubseteq H_2
\end{aligned}$$

A.4 Soundness Theorem

The soundness of the abstraction relation is captured by the following theorem statement, with $\mathsf{client}(\mathcal{X})$ representing the projection to the client events for each execution in $\mathcal{X}$:

THEOREM 2 (Abstraction). *Assume that $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{C}(\mathcal{L}_2)$ are safe, $\mathcal{C}(\mathcal{L}_1)$ is non-interfering, and $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$. Then $\mathcal{C}(\mathcal{L}_1)$ is safe and* $\mathsf{client}(\llbracket\mathcal{C}(\mathcal{L}_1)\rrbracket) \subseteq \mathsf{client}(\llbracket\mathcal{C}(\mathcal{L}_2)\rrbracket)$.

Proof sketch The proof of soundness follows Batty et al. [2] directly. The *decomposition lemma*, takes an execution of $\mathcal{C}(\mathcal{L})$ and provides corresponding executions of the library, $\mathcal{L}$, and client, $\mathcal{C}$, parts of the program separately. The *composition lemma*, takes executions of the library, $\mathcal{L}$, and client, $\mathcal{C}$, parts of the program and provides a corresponding execution of $\mathcal{C}(\mathcal{L})$. The proof of each lemma follows the original.

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 $\mathcal{C}(\mathcal{L}_2)$.

A.5 Inter-workgroup Barrier Specification and Implementation

The specification, $\mathcal{L}_{\mathsf{spec}}$, is a simple piece of code that incontrovertibly captures the intended semantics of the barrier. In our case the specification is simply a call to the inter-workgroup barrier:

```
inter_workgroup_barrier()
```

The inter-workgroup barrier implementation, $\mathcal{L}_{\mathsf{imp}}$, is that described in Section 4.2.

A.6 Specification and Implementation Axiomatisation

We characterise the pre-executions of the barrier in generalised execution shapes we call *axiomatisations*.

Specification An execution of the barrier specification has the following shape for a:call, c:ret, and b:barrier$_{\mathsf{dv}}$, representing the entry and exit event pair in sequence:

$$S^{\text{spec}} \;=\; (\{a,b,c\},\; a \xrightarrow{\mathsf{sb}} b \xrightarrow{\mathsf{sb}} c)$$

Implementation In the execution shapes below, load and store events take two arguments representing the location that they access, and the value that is read or written, respectively, the barrier$_{\mathsf{wg}}$ event represents barrier entry and exit events, in sequence, and x_t represents the distinguished location allocated to thread t to use for its synchronisation with a slave workgroup. We use the superscript in e^k to indicate a sequence of k distinct events of the sort specified by e, and we write e^+ for an infinite sequence of events.

There are two possible shapes on the master workgroup. The first is the shape $S^{\text{master}}_{t,k}$, made up of the following events in sequence:

$$\begin{aligned}
a &: \mathsf{call}\\
&: (\mathsf{load}_{\mathsf{ACQ}}(x_t, i) \text{ such that } i = 0)^k\\
b &: \mathsf{load}_{\mathsf{ACQ}}(x_t, i) \text{ such that } i = 1\\
c &: \mathsf{barrier}_{\mathsf{wg}}\\
d &: \mathsf{store}_{\mathsf{REL}}(x_t, 0)\\
e &: \mathsf{ret}
\end{aligned}$$

The second is the result of repeatedly reading a value other than 1, S^{master}_t, with events:

$$\begin{aligned}
a &: \mathsf{call}\\
&: (\mathsf{load}_{\mathsf{ACQ}}(x_t, i) \text{ such that } i = 0)^+
\end{aligned}$$

There are two possible shapes on the first thread of the slave workgroup. The first is the shape, $S^{\text{slave}}_{t,k}$, with events:

$$\begin{aligned}
a &: \mathsf{call}\\
b &: \mathsf{barrier}_{\mathsf{wg}}\\
c &: \mathsf{store}_{\mathsf{REL}}(x_t, 1)\\
&: (\mathsf{load}_{\mathsf{ACQ}}(x_t, i) \text{ such that } i \neq 1)^k\\
d &: \mathsf{load}_{\mathsf{ACQ}}(x_t, i) \text{ such that } i = 1\\
e &: \mathsf{barrier}_{\mathsf{wg}}\\
f &: \mathsf{ret}
\end{aligned}$$

The second is the result of repeatedly reading a value other than 0, S^{slave}_t, with events:

$$\begin{aligned}
a &: \mathsf{call}\\
b &: \mathsf{barrier}_{\mathsf{wg}}\\
c &: \mathsf{store}_{\mathsf{REL}}(x_t, 1)\\
&: (\mathsf{load}_{\mathsf{ACQ}}(x_t, i) \text{ such that } i \neq 0)^+
\end{aligned}$$

There is only a single shape possible for threads in the slave workgroup, other than the first, S^{slave}, with events:

$$\begin{aligned}
a &: \mathsf{call}\\
b &: \mathsf{barrier}_{\mathsf{wg}}\\
c &: \mathsf{barrier}_{\mathsf{wg}}\\
d &: \mathsf{ret}
\end{aligned}$$

A.7 Satisfying the Abstraction Relation

Safety of $\mathcal{L}_{\mathsf{imp}}$ and $\mathcal{L}_{\mathsf{spec}}$. The specification performs no memory accesses and is therefore free of heterogeneous races. The implementation performs all of its accesses with atomics at the widest scope, device, and is therefore free from heterogeneous races.

Abstraction: We now show that $\mathcal{L}_{\mathsf{imp}} \sqsubseteq \mathcal{L}_{\mathsf{spec}}$. We have that $\mathcal{L}_{\mathsf{imp}}$ and $\mathcal{L}_{\mathsf{spec}}$ are safe, so their executions have defined behaviour and both $\llbracket\mathcal{L}_{\mathsf{imp}}, R\rrbracket$ and $\llbracket\mathcal{L}_{\mathsf{spec}}, R\rrbracket$ are their respective sets of library-local consistent executions. Choose an arbitrary R and execution $X_1 \in \llbracket\mathcal{L}_{\mathsf{imp}}, R\rrbracket$ with history (A_1, G_1). We must show that there is an execution of $\llbracket\mathcal{L}_{\mathsf{spec}}, R\rrbracket$ with the same history.

The MGC restricts the program shape: on every thread in every workgroup there are the same number of calls to the inter-workgroup barrier. Take the thread in X_1 with the largest number of barrier calls, and let this number be n. The inter-workgroup barrier specification does not block, so we can choose an execution, X_2 of the specification under the MGC, with n barrier calls on each thread. We will show that the history of X_1, (A_1, G_1) matches the history of X_2, (A_2, G_2).

We proceed by finite induction over sb-prefixes of the executions X_1 and X_2. Start with prefixes containing only the initialisation events of each, and then at each step add the sb-next call/ret events, and all events sb-between, on each thread. We will establish that for every step, the histories of the two prefixes match, all calls in the prefix of X_1 terminate, and for any event added to the end of the prefix of X_1, the only visible write at every synchronisation location is a write of 0. Note that if this holds for every finite prefix, by the finite structure of the MGC, this holds for X_1 and X_2, as required.

The base case is provided by the fact that all locations are zero initialised. For an arbitrary step, all preceding calls to the barrier have a matching history, no thread in the specification is blocked, and the preceding writes visible to any reads in the newly added events are 0.

The axiomatisations above give us the execution shapes of the barrier calls on each thread of the implementation and specification under the MGC. For the prefix of X_1, all prior calls to the barrier terminate, so the number of workgroup barriers used between the call and return events is the same across all threads on the master (just one) and slave (two). This means that the barrier instances identified by the thread-local semantics will link all of the barrier events in the first call together, then the second and so on, up to the newly added events. The newly added events must also match the axiomatisations.

We can discard nonterminating shapes Suppose there were a non-terminating master thread with newly added events of shape S_t^{master}, repeatedly reading from x_t. The slave thread t must be of shape $S_{t,k}^{\text{slave}}$ or S_t^{slave}, and in either case, there is an unconditional write of x_t. The non-terminating master thread would have to perform an infinite number of reads of the visible write of 0, ignoring the mo-later write of 1, violating the progress axiom, a contradiction. Without the shape S_t^{master}, the newly added events must match $S_{t,k}^{\text{master}}$. Now suppose there is a non-terminating slave thread with newly added events of shape S_t^{slave}. The only shape allowed for the corresponding master thread, $S_{t,k}^{\text{master}}$, has an unconditional write of x_t so we can discard both nonterminating shapes, and we have established that newly added implementation barrier calls terminate.

This gives us: a master workgroup, where each call to the barrier on each thread is of the shape $S_{t,k}^{\text{master}}$, and a set of slave workgroups where on the first thread each call to the barrier is of the shape $S_{t,k}^{\text{slave}}$, and on all of the others each call to the barrier is of the shape S^{slave}. The terminating shapes feature both call and ret events, so the interface events of every consistent execution of the new prefix of X_1 match those of X_2.

Guarantees match, and all locations set to 0 There are four cases to consider: same-workgroup, slave-master, master-slave and slave-slave. For the same-workgroup cases, be they master or slave, workgroup barrier synchronisation gives us the required happens before edges.

For slave-master synchronisation, note that on every slave thread, the call event is followed by a barrier, synchronising with all other thread in the workgroup. The first thread of the workgroup has a following write release on a location only used by this thread and the master, event c in the shape $S_{t,k}^{\text{slave}}$. One of the master threads accesses the same location, and this thread has the shape $S_{t,k}^{\text{master}}$. Event b on this thread corresponds to a successful read of the value 1 at the location. This read can only be from the write on the slave, because all other writes of the value 1 are hidden by more recent writes in happens-before, by the inductive hypothesis. Reading from the slave synchronises. Finally event c causes all master threads to synchronise, event d sets the value of the location back to 0, and we have replicated the call-to-return synchronisation of the specification from slave to master, and we have the visible write of 0, as required.

For master-slave synchronisation, the argument is similar, but we rely on the release write d from the master shape $S_{t,k}^{\text{master}}$ synchronising with event d from $S_{t,k}^{\text{slave}}$.

The slave-slave case is similar again relying on both synchronisation from the slave event c to the master event b and from the master event d to the slave event d.

This covers all of the cases necessary to show that the call-return happens before edges match the specification the inductive case. The fact that the MGC only generates finite numbers of calls to the barrier completes the proof.