Forward Progress on GPU Concurrency

Alastair F. Donaldson[1], Jeroen Ketema[2], Tyler Sorensen[3], and John Wickerson[4]

1 **Department of Computing, Imperial College London, UK**
alastair.donaldson@imperial.ac.uk
2 **Embedded Systems Innovation by TNO, Eindhoven, the Netherlands**
jeroen.ketema@tno.nl
3 **Department of Computing, Imperial College London, UK**
tyler.sorensen15@imperial.ac.uk
4 **Department of Electrical and Electronic Engineering, Imperial College London, UK**
j.wickerson@imperial.ac.uk

Abstract

The tutorial at CONCUR will provide a practical overview of work undertaken over the last six years in the Multicore Programming Group at Imperial College London, and with collaborators internationally, related to understanding and reasoning about concurrency in software designed for acceleration on GPUs. In this article we provide an overview of this work, which includes contributions to data race analysis, compiler testing, memory model understanding and formalisation, and most recently efforts to enable portable GPU implementations of algorithms that require forward progress guarantees.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases GPUs, concurrency, formal verification, memory models, data races

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2017.1

1 Introduction

Graphics processing units (GPUs) offer a large degree of parallelism at a relatively low cost, and are now routinely applied to the acceleration of a wide variety of computational tasks that go well beyond the domain of graphics (see e.g. [39] for details of many application areas and developments).

It is invariably harder to design a software application that takes advantage of GPU parallelism than it is to write a sequential version of the application that runs only on the CPU. Furthermore, parallel programming for GPUs is in many ways more complicated than parallel programming for multi-core CPUs. This is because achieving high performance requires working in low level languages, such as CUDA [33] and OpenCL [24], which provide close-to-the-metal language features to enable mapping an algorithm to the architectural capabilities of a device. Numerous high level programming models have been proposed to ease the burden of GPU programming, via automatic generation of low level code, but as yet are not widely adopted.

Low level programming of GPUs is made challenging by traditional concurrency bugs such as data races, by issues related to relaxed memory, and by the constrained hardware execution model on which software executes. Furthermore, reliable production compilers for GPUs are hard to maintain due to the challenge of keeping pace with changing architectures and evolving source languages, and because GPU-specific optimisations—e.g. ones that

28th International Conference on Concurrency Theory (CONCUR 2017).
Editors: Roland Meyer and Uwe Nestmann; Article No. 1; pp. 1:1–1:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

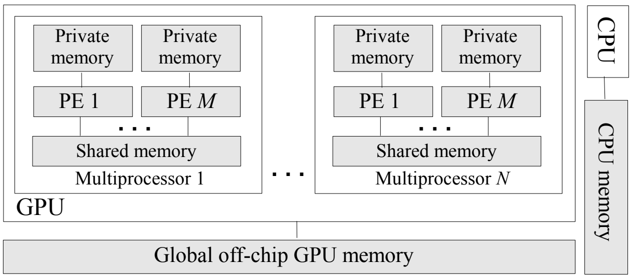

Figure 1 Overview of a typical GPU architecture

aim to reduce control-flow divergence between threads—are important for performance but error-prone to implement.

After giving a brief overview of the GPU programming model (Section 2), we present an overview of a line of work pursued by the Multicore Programming Group at Imperial College London, with various collaborators, which has contributed to: automated data race analysis (Section 3), compiler testing (Section 4), memory models (Section 5), and forward progress guarantees to support blocking algorithms (Section 6). While we principally focus on the approach taken by our line of work, we briefly discuss related approaches. In each area, we also touch on open problems that will require future research to solve.

2 Overview of the GPU Programming Model

GPUs are programmed using an SPMD (single program, multiple data) model, in which a large number of *processing elements* (PEs) all execute the same program, with each PE operating on a different subset of some shared data. At the hardware level, PEs are typically grouped into *multiprocessors* (see Figure 1). Each PE generally has access to a few hundred bytes of *private memory*; all the PEs in the same multiprocessor have a few kilobytes of *shared memory* available; and a large amount of *global memory* is shared between all multiprocessors.

The two main programming languages for writing SPMD programs are OpenCL [24] and CUDA [33], both of which derive from the C programming language. The languages follow the above sketched hardware hierarchy quite closely by subdividing the execution of a program among a number of *work-items* (or *threads*)—mapping to PEs. These work-items are a grouped into (multi-dimensional) *work-groups* (or *thread-blocks*)—mapping to multiprocessors—which in turn are grouped into (multi-dimensional) *NDRanges* (or *grids*). A programmer specifies the program to be executed by a single work-item—called a *GPU kernel*—and defines the work-group and NDRange sizes that should be used for execution. The OpenCL programming model uses the terms *work-items*, *work-groups*, and *NDRanges*, while CUDA uses *threads*, *thread-blocks*, and *grids*. For the remainder of the article we use the OpenCL terminology, except that we find it more natural to use *thread* rather than *work-item*.

An OpenCL kernel computing a prefix-sum (or scan) [25] is presented in Figure 2. The kernel is intended to be executed by a single 1-dimensional work-group. The keyword **`global`** indicates that the arrays `in` and `out` reside in global memory. Similarly, the keyword **`local`** (not used in the example) indicates that an array resides in shared memory. Any variable declared without the **`global`** or **`local`** keyword is thread-private. To enable each thread to operate on different data, several functions are provided that allow a thread to access its unique id. One of these functions is `get_local_id`, which yields the unique id of a thread

```
kernel void KoggeStone(global int *in, global int *out) {
    unsigned tid = get_local_id(0);
    out[tid] = in[tid];
    barrier(CLK_GLOBAL_MEM_FENCE);
    for (unsigned offset = 1; offset < tid; offset *= 2) {
        int temp;
        if (tid >= offset)
          { temp = out[tid - offset]; }
        barrier(CLK_GLOBAL_MEM_FENCE);
        if (get_local_id(0) >= offset)
          { out[tid] = temp + out[tid]; }
        barrier(CLK_GLOBAL_MEM_FENCE);
    }
}
```

Figure 2 A Kogge-Stone prefix-sum GPU kernel in OpenCL

within a work-group.

Except when performing the most basic computations, threads typically need to share intermediate results. In OpenCL and CUDA, these intermediate results are traditionally shared with the help of *synchronisation barriers*: each thread writes its intermediate results to local or global memory and then calls the **barrier** function. The **barrier** function stalls the thread until all other threads *within the same work-group* have also called the function and all outstanding memory operations have been committed. The argument of the **barrier** function indicates whether operations on local or global memory need to be committed. In the case of the GPU kernel of Figure 2, only operations on global memory are committed.

As stressed above, barriers can *only* be used for communication within work-groups; they cannot be used for communication between work-groups. The same holds for early implementations of atomic operations on GPUs. This was rectified with OpenCL 2.0, which defines atomic operations that can operate between work-groups. The semantics of OpenCL 2.0 atomics are subtle, and expressing them precisely is a research endeavour we are pursuing, thus we postpone a more detailed discussion of them to Section 5. Finally, it is worth mentioning that the new NVIDIA CUDA 9.0 provides primitives for barrier synchronisation across work-groups [34].

3 Static Data Race Analysis

Data races are an important kind of defect that affect shared memory concurrent programs, and GPU kernels in particular. Informally, a GPU kernel exhibits a *data race* if it is possible for two distinct threads, t_1 and t_2, to issue memory operations accessing a common location m such that: at least one of the operations is non-atomic, at least one of the operations modifies m, and t_1 and t_2 do not synchronise between the operations. The semantics of data races are not well defined in CUDA nor in older versions of OpenCL, and data races are a form of undefined behaviour in OpenCL 2.0 [24] and later. As well as being defects in their own right, data races can lead to unwanted nondeterminism, which can cause other bugs that are hard to reproduce and fix.

To aid programmers in reasoning about data races in GPU kernels we spent several years designing GPUVerify, a static data race analysis tool [11, 10, 4]. GPUVerify caters primarily for the *traditional* style of GPU programming where threads in distinct work-groups do not communicate during the lifetime of a kernel, and where intra-work-group communication is exclusively via synchronisation barriers.

The traditional setting affords a key property: when a GPU kernel is executed on a given input, if *any* thread schedule can exhibit a data race, then *all* thread schedules exhibit a data race. We briefly sketch the argument for this. Let us call a data race occurring in a given thread schedule a *principal race* if the associated memory accesses a_1 and a_2 are guaranteed to occur, regardless of the instructions executed by other threads. That is, the conditions for the associated threads to issue a_1 and a_2 arise directly from the input on which the GPU kernel is executed, and do not depend on the thread schedule. Clearly, if a thread schedule exhibits a principal race involving accesses a_1 and a_2, then *every* thread schedule must exhibit this principal race. Furthermore, in the traditional GPU programming model, every thread schedule that exhibits a race is guaranteed to exhibit a principal race. This is because a traditional GPU kernel behaves *entirely deterministically* in the absence of a data race [16].[1] Threads in different work-groups have no means of synchronising, and thus can only influence one another by racing. Threads in the same work-group can only synchronise via barriers. Hence, in the absence of data races, the execution of each thread is independent of the actions of all other threads until barrier synchronisation occurs, and the state of each thread upon reaching a barrier is independent of the chosen thread interleaving. Repeating this argument for each following barrier, we see that traditional GPU kernel execution is deterministic. Because a race-free GPU kernel is deterministic, a kernel exhibiting a race is guaranteed to have a fixed set of principal races and, hence, if any schedule exhibits a race then all do.

GPUVerify leverages the above result to reduce the problem of reasoning about data races in highly parallel GPU kernels to reasoning about assertions in a sequential program. This is achieved by translating a parallel kernel into a sequential program that models one particular thread schedule between each pair of barrier synchronisation points in the kernel, tracking the reads and writes of threads, and using assertions to determine the conditions under which data races occur. In practice, GPUVerify uses a schedule in which threads execute in lock-step [18].

Because GPU kernels are often executed by hundreds or even thousands of threads, it would not be practical to model all these threads explicitly. Instead, GPUVerify exploits the fact that data races occur *pairwise*, and only models execution of a kernel by a pair of threads. The identities of these threads are made symbolic in the sequential program that GPUVerify generates, so that reasoning is in fact performed over all possible pairs. To over-approximate the effects of additional threads, the shared state of the GPU kernel is made abstract. By default GPUVerify uses an extremely coarse abstraction, where other threads are assumed to have arbitrary effects on the shared state [10].

The sequential program generated by GPUVerify is encoded in the Boogie intermediate verification language [5], whose supporting tool—also called Boogie—generates verification conditions that are discharged by SMT solvers such as Z3 [19] and CVC4 [6]. To handle loops, Boogie requires invariants to be supplied. For this purpose we have equipped GPUVerify with a tailored engine for loop invariant inference, which is by now relatively mature. The engine works by speculating candidate loop invariants via a set of rules, which we devised through careful study of a large set of GPU kernels [9]. The candidate invariants speculated for a kernel are fed to the Houdini algorithm [20], which computes the largest conjunction of candidates that form an inductive invariant for the kernel. The computed invariant is then used in an attempt to prove freedom from data races.

[1] We assume here that concurrency is the only possible cause of nondeterminism; sources of nondeterminism due to e.g. accessing invalid memory locations are an orthogonal concern.

Several other methods for testing and verifying properties of GPU kernels have been proposed. These include approaches based on dynamic analysis [35, 48, 28], verification via SMT solving [29, 38, 44], symbolic execution [30, 17], and program logic [12, 26].

Open problems

Despite operating reasonably automatically on a relatively large set of GPU kernels [4, 9], the GPUVerify technique has three main limitations.

First, the loop invariant inference method used by the tool can be inefficient, due to the sheer number of candidate invariants that are speculated. Many of the candidates turn out to be incorrect or non-inductive, so that many iterations of the Houdini algorithm are required to refute them, each iteration requiring one or more expensive calls to an SMT solver. We believe that a significantly more efficient mechanism for computing relevant loop invariants could be built by performing abstract interpretation over carefully-designed abstract domains that capture the memory access patterns of GPU kernels.

Second, the shared state abstraction employed by GPUVerify leads to false alarms for kernels whose race-freedom depends on richer properties of the shared state. To aid in the verification of such kernels, GPUVerify supports *barrier invariants*—properties of the shared state that are proven to be satisfied by threads individually on entry to a barrier synchronisation operation, and which can thus be assumed to hold for all threads on exit from the barrier [15]. However, barrier invariants must currently be specified manually, and have only been investigated practically in one domain so far: reasoning about parallel prefix sums. Relatedly, our recent work on termination analysis for GPU kernels shows that the coarse shared state abstraction used by GPUVerify often suffices for proving kernel termination in a thread-modular manner, but identifies a number of cases where richer shared state abstractions are required [23].

Third, GPUVerify fundamentally exploits the traditional GPU programming model where barriers are the only means for synchronisation. As discussed further in Sections 5 and 6, modern GPU kernels increasingly go beyond the boundaries of this simple computational model, using atomic operations to implement fine-grained concurrent algorithms. In general, reasoning about data race-freedom of GPU kernels that use atomic operations is just as hard as reasoning about data race-freedom for arbitrary concurrent programs. In particular, a race-free GPU kernel need not behave deterministically if it uses atomic operations. This breaks the property that allows GPUVerify to reason about a kernel via translation to a sequential program. Although GPUVerify can handle some limited use cases for atomic operations [7], the tool over-approximates more general uses of atomics, so that it will report false alarm data races for algorithms that, for instance, protect shared data structures using mutexes built via atomic operations. Nevertheless, we conjecture that the contexts in which GPU programmers use atomic operations in practice are likely to be limited and idiomatic, and that there may be scope for targeted analyses that exploit this idiomatic nature.

In related work, we note that GKLEE, another GPU race detection tool [30], has also been extended to reason about atomics in some scenarios [14], and that reasoning about atomic operations in GPU kernels via *resource invariants* in separation logic has recently been proposed [3].

4 Many-Core Compiler Fuzzing

The race analysis provided by GPUVerify (see Section 3) operates on the intermediate representation of the LLVM compiler framework, generated by Clang's OpenCL and CUDA front-ends. More generally, it is common for static race analysis tools to operate at the level of program source code, or at some intermediate representation associated with a particular tool chain. Either way, an analysis that establishes properties of a program at some higher-than-binary level of abstraction relies on correct downstream compilation.

Aware that guarantees provided by GPUVerify are conditional on the reliability of CUDA and OpenCL compilers, we undertook research into testing such compilers. We focused on testing OpenCL compilers using two methods: *random differential testing* [32], where compilers are cross-checked against each other using randomly-generated programs, and *equivalence modulo inputs testing* [27], where a single compiler is tested via a family of programs that, for a given input, ought to yield equivalent results. In both cases, mismatches are indicative of compiler bugs. Both methods are examples of *fuzzing*, where a system is tested against randomly-generated inputs.

We built a tool, CLsmith [31], which extends the Csmith generator for C programs [47] to the domain of OpenCL. It was relatively simple to extend Csmith to generate "embarrassingly parallel" OpenCL kernels in which threads do not communicate at all. More challenging was to devise methods for generating OpenCL kernels in which threads *do* communicate—either using barriers or atomic operations—but in a manner such that the generated kernels are guaranteed to be free from data races and to compute deterministic results.[2]

We armed CLsmith with three modes for generating such kernels: *barrier* mode, where generated kernels are equipped with shared arrays, indexed in a manner that avoids data races; *atomic-sections* mode, where atomic operations are used to build sections of code that can only be executed by a single thread, structured such that the particular thread that executes a section depends on the thread schedule, but such that the side-effects associated with executing the section are independent of the specific thread that executes the section; and *atomic-reductions* mode, where associative, commutative atomic operations (such as `atomic_add` and `atomic_min`) are used to perform reductions on data values computed locally by individual threads, ultimately leading to deterministic results.

Via an experimental campaign applying CLsmith to 21 OpenCL (device, compiler)-configurations, covering a range of GPU, CPU, FPGA, and emulator implementations [31], we discovered more than 50 OpenCL compiler bugs, most affecting commercial implementations. Surprisingly, and disappointingly from an academic perspective, these bugs were almost exclusively *sequential*: basic compilation bugs that could manifest in a single-threaded kernel. We found a large number of defects affecting programs that manipulate data via user-defined structures. While we found a small number of bugs that required barrier synchronisation operations to be present in order to manifest, even these did not appear to be directly concurrency-related: they could affect kernels that, despite featuring barriers, did not depend on inter-thread communication for result computation.

Open problems

The fact that our testing methods did not identify any compiler bugs that were directly concurrency-related could be because (a) OpenCL compilers do not yet perform sophisticated concurrency-related optimisations, (b) our method did in fact trigger concurrency-related compiler bugs, but in addition triggered so many basic sequential compiler bugs that the

[2] As argued in Section 3, kernels that only use barrier operations for inter-thread communication are guaranteed to be deterministic if they are race-free. However, atomic operations open the possibility for race-free kernels to be nondeterministic.

effects of the concurrency-related bugs were masked, or (c) the methods we investigated so far for detecting concurrency-related bugs are too simple.

Possibility (a) could be investigated with reference to open source compilers, such as the Intel back-end for the Beignet open source implementation of OpenCL.[3] Running a similar testing campaign against more mature OpenCL compilers in the future might shed light on possibility (b).

As we discuss further in Section 5, OpenCL 2.0 includes a full suite of well-defined atomic operations, together with a memory model specification. With respect to possibility (c), this raises the problem of how to generate random OpenCL kernels that exercise these concurrency-related features in non-trivial yet predictable ways, to test more thoroughly the compilation of concurrency primitives.

5 GPU Memory Models

As mentioned in Section 3, many GPU programmers have been going beyond the traditional barrier synchronous model of computation, writing fine-grained concurrent algorithms that manipulate irregular data structures, e.g. graphs. This raises the question of what sort of *memory models* GPU architectures present, and in particular, what values a thread might observe when reading from a shared memory location.

Empirically, we used *litmus tests* to investigate the memory models of several NVIDIA and AMD GPUs [1], showing (a) that they *do* exhibit relaxed (i.e. non-sequentially consistent) behaviour, and (b) that several programming idioms relied on by high level algorithms assume that such relaxed behaviour *cannot* occur, and thus are not guaranteed to work correctly on these platforms. In follow-up work we substantiated this possibility by designing a testing framework geared towards automatically provoking relaxed memory bugs in GPU-accelerated applications [40].

In large part, the issues that we discovered in these works stem from the lack of a memory model specification in CUDA, and in OpenCL prior to OpenCL 2.0. The OpenCL 2.0 specification [24] provides a memory model specification based on the C11 memory model [22]. However, the memory model is complicated by the fact that OpenCL features a hierarchical organisation of threads and memory spaces, as discussed in Section 2 and illustrated by Figure 1. For example, threads in the same work-group can use fast shared memory to communicate, while threads in different work-groups must communicate via slower global memory. OpenCL also features *shared virtual memory* (not depicted in the overview of Figure 1), through which device threads can communicate with threads running on the host processor. The situation is further complicated by the fact that memory accesses are *scoped*. For example, a write to global memory can have *device* scope, so that its effects are visible to all threads running on a device, or the more restricted *work-group* scope, so that although the write will eventually propagate to all threads, the write can only contribute to reliable synchronisation with threads in the same work-group. These subtleties are intended to provide optimisation opportunities for OpenCL programmers and language implementers. Our attempt to formalise the memory model [8] led to the identification of some fundamental flaws in the way the model handles the class of *sequentially-consistent* (SC) memory accesses. SC is the default mode for accessing memory, and is intended to be the easiest for programmers to reason about (but not the most efficient), because all SC accesses in a program are guaranteed to be executed in some total order on which all

[3] `https://www.freedesktop.org/wiki/Software/Beignet/`

threads agree. Our formalisation exposed a tension between this total order (which is visible to *all* threads), and OpenCL's scoping mechanism (which is intended to restrict visibility to just *some* threads). We were able to construct a small OpenCL program that involves SC accesses on two different devices, and show that even if these accesses are scoped to be visible only within the device that performs them, the compiler is still obliged to enforce an inter-device order between them [8, Example 10]. As such, the existing OpenCL memory model provides guarantees that are too strong to be efficiently implemented by a compiler. In our work, we suggest how the situation could be resolved in a future revision of the OpenCL memory model by enforcing a total order only between SC memory accesses that have the same scope [8, §5].

Although OpenCL's scoping mechanism allows some synchronisation patterns to be optimised, its usefulness is hampered by its requirement that two memory accesses can only synchronise if both use a scope that is wide enough to encompass the other. This symmetry inhibits the popular work-stealing pattern [13], which is often used to implement algorithms with irregular workloads. In order for a work-group to allow its work to be stolen by another work-group, it must scope all of its memory accesses widely enough to encompass all the work-groups that might wish to steal. Using such a wide scope is unnecessary in the common case when no stealing occurs and visibility is only needed within the current work-group. To rectify this shortcoming, Orr et al. have proposed *remote-scope promotion* (RSP) [36]. In their proposal, memory accesses can default to the intra-work-group scope (for good performance in the common case), and when a thread wishes to steal from another work-group, it invokes RSP to temporarily widen the scope of the victim's memory accesses, so that it can perform the necessary synchronisation to steal work.

Using the Isabelle proof assistant and the HERD memory model simulator [2], we formalised Orr et al.'s RSP extension and their proposed mapping to AMD's prototype GPU architecture [45]. These efforts brought to light several corner cases in the design, which we discussed in detail with Orr et al. We also discovered two serious flaws in the proposed mapping, both of which could lead to incorrect results being calculated. We were able to propose fixes in both cases. This work emphasises the value of applying formal methods to the early-stage design of hardware support for concurrent programming language features.

In more recent work, we have gone on to develop techniques for making the bug-finding process applied to Orr et al.'s work more automatic. Our MEMALLOY tool [46] takes any source language memory model, any target architecture memory model, and a description of a compilation mapping from source language to target architecture, and uses automatic constraint-solving technology to seek programs with behaviours that are forbidden at the source language level but are nonetheless observable on the target architecture when the compilation mapping is applied. For instance, the tool is able to reproduce (and minimise) our two manually-found RSP-related bugs in under two hours. We also used MEMALLOY to explore the compilation process from OpenCL to NVIDIA GPUs, and thereby discovered that the memory model we had deduced for NVIDIA GPUs through litmus testing (see the beginning of this section) was too weak to support the natural compilation scheme for OpenCL. After strengthening (and re-validating) the NVIDIA model, MEMALLOY was able to prove (up to a bound) the absence of memory-related bugs in the compilation scheme.

Open problems

There remains work to be done to investigate how the OpenCL memory model is implemented on the other kinds of devices that are OpenCL targets, such as FPGAs and digital signal processors (DSPs). For instance, Intel's OpenCL compiler for its FPGAs supports some of OpenCL's atomic operations, and it may be possible to employ formal methods to help ascertain whether they are implemented correctly and as efficiently as possible.

Also, existing studies of GPU memory models (both at the language level and the architecture level) have focused primarily on the single-device setting. However, OpenCL is designed to allow multiple devices to communicate with each other and with the host processor (via shared virtual memory). There is a need for techniques to help ascertain whether the OpenCL memory model has been implemented correctly and efficiently in this case. Furthermore, when shared virtual memory is used in 'fine-grained system' mode (whereby devices have simultaneous access to the entirety of the host's memory), there is a need to understand how the OpenCL memory model (as implemented on those devices) interacts with the memory model of the host processor. A general theory of how to *compose* memory models may be called for.

6 Blocking Algorithms and Forward Progress

Many parallel programs on traditional multicore systems (e.g. CPUs) have blocking behaviours. That is, one thread in the system will spin, waiting for another thread in the system to reach a certain point in the program. These interactions are typically orchestrated using flag values in a shared memory region. Common examples of concurrent idioms with blocking behaviours are mutexes and synchronisation barriers. In order to execute correctly (e.g. terminate or make progress), blocking behaviours require certain properties from the thread scheduler. Namely, if a thread t_1 is waiting for a thread t_2, the scheduler must ensure that t_2 is allowed to execute, thus freeing t_1 from waiting. If the system, for whatever reason, cannot guarantee that t_2 will eventually execute, then t_1 might wait indefinitely. A scheduler which guarantees the relative execution of threads is said have the *forward progress* property.

Currently, GPU specifications do not provide forward progress properties between threads in different work-groups, effectively disallowing popular existing idioms (and thus, programs) from multicore CPU systems to be ported to GPUs in a reliable manner. However, GPU programmers have discovered that by exploiting quirks in today's GPUs, certain blocking idioms can be made to execute as expected. This can be achieved pragmatically by determining (via trial and error) the number of work-groups for which forward progress appears to be guaranteed, and hard coding this number into the GPU kernel. This can be fiddly, since in practice the number is influenced by the program (e.g. due to register usage), the target GPU (e.g. due to hardware resource limits), and the GPU driver (e.g. due to its policy on how resources are allocated). Moreover, this approach to writing applications suffers from two drawbacks. First, applications that exploit such quirks are *not portable* even across GPUs from the same designer: undesirable behaviour (e.g. indefinite spinning) may occur when executing the program on a different GPU model, applying otherwise semantics-preserving changes to the program, or updating the GPU driver. Second, because the behaviour is not guaranteed by GPU programming language specifications, and applications written in this manner are generally not tested on a GPUs from a wide range of vendors [41], an application tested on multiple GPU models from one vendor may nevertheless fail when executed on a GPU from another vendor. Indeed, we observed that an inter-work-group synchronisation barrier written for one GPU (e.g. the NVIDIA GTX Titan) might deadlock when run on a different GPU (e.g. the Intel HD5500).

To address the above drawbacks, we performed a large empirical study (across eight GPUs spanning four vendors) to search for a forward progress abstraction that was (a) able to account soundly for the behaviour across the GPUs that we tested, and (b) useful enough to allow the implementation of popular blocking idioms [42]. After probing our zoo of GPUs with litmus tests, we determined that all GPUs had the following property: once a work-group begins executing a kernel (i.e. the work-group becomes *occupant* on a hardware resource), it will continue executing until it reaches the end of the kernel. We call this execution model the *occupancy bound* execution model, because the number of work-groups for which relative forward progress is guaranteed is bound by the hardware resources available for executing work-groups; i.e. the hardware resources determine how many work-groups can be occupant.

The occupancy bound execution model naturally allows mutex synchronisation, as a thread that acquires a lock is guaranteed to continue executing, thus eventually releasing the lock.[4] A portable synchronisation barrier across work-groups is, however, much more difficult to achieve. To address this, we developed a *discovery protocol*, which dynamically, at kernel launch, determines a safe, lower-bound estimate of the number of occupant work-groups. Work-groups that are not part of the initial wave of occupant work-groups discover that this is the case, and immediately exit the kernel. As a result, they do not participate in any blocking synchronisation. Kernels must be designed to be agnostic to the number of executing work-groups, to account for the fact that the number of occupant work-groups may vary across GPUs and between driver versions, and due to the fact that the discovery protocol does not guarantee discovering all work-groups that are initially occupant, and may discover a different number of work-groups during different executions of the same kernel. Previous work shows that it is usually possible to design kernels to satisfy this constraint, and programs with this property are said to use the *persistent thread* programming model [21].

The discovery protocol uses a polling mechanism, protected by a mutex, to allow work-groups to mark themselves as executing. Because the poll is only open for a finite amount of time, it is possible that the occupant work-groups may not be able to mark themselves, and contribute to the computation, thus under utilising the GPU. We show through a wide range of experiments that on current GPUs, we are able to define a discovery protocol that has a recall of nearly 100%. Our main insight is that using a *fair* mutex, such as a ticket lock, to guard the polling data structure leads to better recall than is achievable using a more traditional compare-and-swap-based spin-lock.

Open problems

As part of our work, we benchmarked a set of applications that use our discovery protocol and inter-work-group synchronisation barrier against versions of the same applications that use a *multi-kernel* approach to achieve global synchronisation (by explicitly ending and re-launching the kernel). We observed a wide range of performance profiles across the eight GPUs that we tested. For example, on Intel GPUs, our inter-work-group barrier provides a median 28% performance improvement compared with the multi-kernel approach, while on ARM GPUs we observed a median 50% *slowdown* using the inter-work-group barrier. An interesting open problem is to investigate whether a performance model can be developed that can predict when it is beneficial to use an inter-work-group barrier. Recent work has started to develop such a model for NVIDIA GPUs, but is yet to be extended to GPUs of other vendors [37].

In our work so far we have only examined the relative forward progress properties associated with threads in *different* work-groups. Forward progress properties between threads at different levels of the GPU execution hierarchy may not be the same. As with

[4] Of course, it is possible as usual for deadlock to occur due to mismanagement of multiple mutexes.

inter-work-group forward progress, GPU specifications do not provide much guidance with respect to intra-work-group forward progress. In this context we think the following are interesting research topics: (a) an empirical investigation of the forward progress properties that GPUs appear to provide at each level of the execution hierarchy, and (b) the identification of applications that might benefit from blocking idioms at additional levels of this hierarchy.

Finally, because GPU specifications *do not* provide forward progress guarantees at present, we believe that working with standards committees in this area is vital. Our understanding from talking with various contacts in the GPU industry is that one reason vendors are reluctant to provide forward progress guarantees is so that they can reserve the right to dynamically change the computational resources assigned to a given GPU kernel, e.g. to make computational resources available to other tasks, or to reduce energy consumption. In our most recent work we have proposed an extension to the GPU programming model called *cooperative kernels*, specifically designed with blocking algorithms in mind, which we hope may allow the forward progress requirements of blocking algorithms to co-exist with the need for dynamic changes in the hardware resources that are assigned to kernels [43].

Acknowledgements

The line of work on which we report was contributed to by many co-authors: Jade Alglave, Ethel Bardsley, Mark Batty, Bradford Beckmann, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Hugues Evrard, Ganesh Gopalakrishnan, Daniel Liew, Daniel Poetzl, Shaz Qadeer, Zvonimir Rakamarić, and Paul Thomson.

References

1 Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS*, pages 577–591, 2015.

2 Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

3 Afshin Amighi, Saeed Darabi, Stefan Blom, and Marieke Huisman. Specification and verification of atomic operations in GPGPU programs. In *SEFM*, pages 69–83, 2015.

4 Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. Engineering a static verification tool for GPU kernels. In *CAV*, pages 226–242, 2014.

5 Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

6 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

7 Ethel Barsdley and Alastair F. Donaldson. Warps and atomics: Beyond barrier synchronisation in the verification of GPU kernels. In *NFM*, 2014.

8 Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL*, pages 634–648. ACM, 2016.

9 Adam Betts, Nathan Chong, Pantazis Deligiannis, Alastair F. Donaldson, and Jeroen Ketema. Implementing and evaluating candidate-based invariant generation. *IEEE Trans. Software Eng.*, 2017. To appear.

10 Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Trans. Program. Lang. Syst.*, 37(3):10:1–10:49, 2015.

11 Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: A verifier for GPU kernels. In *OOPSLA*, pages 113–132, 2012.

12 Stefan Blom, Marieke Huisman, and Matej Mihelčić. Specification and verification of GPGPU programs. *Science of Computer Programming*, 95(3):376–388, 2014.

13 Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

14 Wei-Fan Chiang, Ganesh Gopalakrishnan, Guodong Li, and Zvonimir Rakamarić. Formal analysis of GPU programs with atomics via conflict-directed delay-bounding. In *NFM*, pages 213–228, 2013.

15 Nathan Chong, Alastair F. Donaldson, Paul Kelly, Jeroen Ketema, and Shaz Qadeer. Barrier invariants: A shared state abstraction for the analysis of data-dependent GPU kernels. In *OOPSLA*, pages 605–622, 2013.

16 Nathan Chong, Alastair F. Donaldson, and Jeroen Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *POPL*, pages 397–410, 2014.

17 Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic crosschecking of data-parallel floating-point code. *IEEE Trans. Software Eng.*, 40(7):710–737, 2014.

18 Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *ESOP*, pages 270–289, 2013.

19 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

20 Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, pages 500–517, 2001.

21 Kshitij Gupta, Jeff Stuart, and John D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *InPar*, pages 1–14, 2012.

22 ISO/IEC. *Programming languages – C*. International standard 9899:2011, 2011.

23 Jeroen Ketema and Alastair F. Donaldson. Termination analysis for GPU kernels. *Science of Computer Programming*, 2017. To appear.

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

25 Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comp.*, C-22(8):786—793, 1973.

26 Kensuke Kojima and Atsushi Igarashi. A Hoare logic for SIMT programs. In *APLAS*, pages 58–73, 2013.

27 Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *PLDI*, pages 216–226, 2014.

28 Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying GPU kernels by test amplification. In *PLDI*, pages 383–394, 2012.

29 Guodong Li and Ganesh Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *FSE*, pages 187–196, 2010.

30 Guodong Li, Peng Li, Geoffrey Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. GKLEE: Concolic verification and test generation for GPUs. In *PPoPP*, pages 215–224, 2012.

31 Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *PLDI*, pages 65–76, 2015.

32 William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

33 NVIDIA. CUDA C programming guide, version 5.5, 2013.

34 NVIDIA. CUDA 9 features revealed: Volta, cooperative groups and more, accessed 2017. `https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/`.

35 NVIDIA. CUDA-MEMCHECK, accessed 2017. `http://docs.nvidia.com/cuda/cuda-memcheck/`.

36 Marc S. Orr, Shuai Che, Ayse Yilmazer, Bradford M. Beckmann, Mark D. Hill, and David A. Wood. Synchronization using remote-scope promotion. In *ASPLOS*, pages 73–86, 2015.

37 Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on GPUs. In *OOPSLA*, pages 1–19, 2016.

38 Phillipe A. Pereira, Higo F. Albuquerque, Hendrio Marques, Isabela Silva, Celso Carvalho, Lucas C. Cordeiro, Vanessa Santos, and Ricardo Ferreira. Verifying CUDA programs using SMT-based context-bounded model checking. In *SAC*, pages 1648–1653, 2016.

39 Hamid Sarbazi-Azad, editor. *Advances in GPU Research and Practice*. Morgan Kaufmann, 2017.

40 Tyler Sorensen and Alastair F. Donaldson. Exposing errors related to weak memory models in GPU applications. In *PLDI*, pages 100–113, 2016.

41 Tyler Sorensen and Alastair F. Donaldson. The hitchhiker's guide to cross-platform OpenCL application development. In *IWOCL*, pages 2:1–2:12, 2016.

42 Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. Portable inter-workgroup barrier synchronisation for gpus. In *OOPSLA*, pages 39–58, 2016.

43 Tyler Sorensen, Hugues Evrard, and Alastair F. Donaldson. Cooperative kernels: GPU multitasking for blocking algorithms. In *ESEC/FSE*, 2017. To appear.

44 Stavros Tripakis, Christos Stergiou, and Roberto Lublinerman. Checking equivalence of SPMD programs using non-interference. In *HotPar*, pages 1–5, 2010.

45 John Wickerson, Mark Batty, Bradford M. Beckmann, and Alastair F. Donaldson. Remote-scope promotion: clarified, rectified, and verified. In *OOPSLA*, pages 731–747, 2015.

46 John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *POPL*, pages 190–204, 2017.

47 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294, 2011.

48 Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. GRace: A low-overhead mechanism for detecting data races in GPU programs. In *PPoPP*, pages 135–146, 2011.