The Hitchhiker's Guide to Cross-Platform OpenCL Application Development

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

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

ABSTRACT

One of the benefits to programming of OpenCL is platform portability. That is, an OpenCL program that follows the OpenCL specification should, in principle, execute reliably on any platform that supports OpenCL. To assess the current state of OpenCL portability, we provide an experience report examining two sets of open source benchmarks that we attempted to execute across a variety of GPU platforms, via OpenCL. We report on the portability issues we encountered, where applications would execute successfully on one platform but fail on another. We classify issues into three groups: (1) framework bugs, where the vendor-provided OpenCL framework fails; (2) specification limitations, where the OpenCL specification is unclear and where different GPU platforms exhibit different behaviours; and (3) programming bugs, where non-portability arises due to the program exercising behaviours that are incorrect or undefined according to the OpenCL specification. The issues we encountered slowed the development process associated with our sets of applications, but we view the issues as providing exciting motivation for future testing and verification efforts to improve the state of OpenCL portability; we conclude with a discussion of these.

1. INTRODUCTION

Open Computing Language (OpenCL) is a general-purpose parallel programming model, designed to be implementable on a range of devices including CPUs, GPUs, and FPGAs [17]. Much like mainstream programming languages (e.g. C and Java), the OpenCL specification describes abstract semantics. Concrete platforms that support OpenCL are then responsible for providing a framework that successfully executes applications according to the abstract specification. This contract between programming model and platform enables *portability*; that is, a programmer can develop programs based on the specification and then execute the program on any platform that supports the programming model.

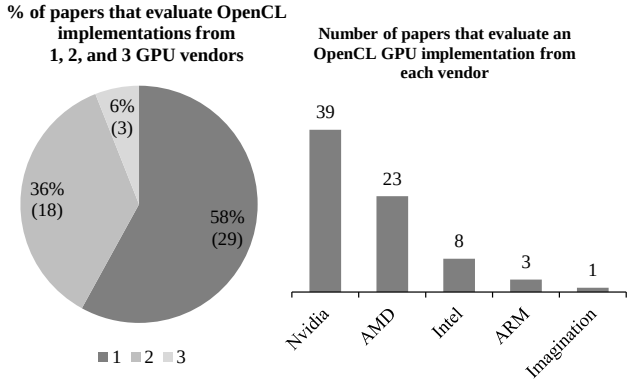

Figure 1: The number of vendors whose OpenCL GPU implementations are evaluated in 50 recent papers listed at http://hgpu.org

As discussed in Sec. 3, we focus on GPU platforms in this study. Many GPU vendors provide implementations of OpenCL for their respective platforms. In principle, this means that programs adhering to the OpenCL specification should be executable across these platforms. However, in our experiences many GPU applications (especially in the research literature) target platforms from a *single* vendor. To quantify this claim, we manually examined the 50 most recent papers listed on the GPU aggregate website http://hgpu.org (retrieved 25 Jan. 2016) that feature evaluation of OpenCL applications on GPU platforms (we exclude papers that exclusively report results for CPUs and/or FPGAs). Our findings are summarised in Fig. 1. The pie chart shows that over half (58%) of the papers evaluated GPUs from one vendor only. Only three papers (6%) evaluated

chip	vendor	CUs	type	abbr.	OCL
GTX 980	Nvidia	16	discrete	980	1.1
Quadro K500	Nvidia	12	discrete	K5200	1.1
Iris 6100	Intel	47	integrated	6100	2.0
HD 5500	Intel	24	integrated	5500	2.0
Radeon R9	AMD	28	discrete	R9	2.0
Radeon R7	AMD	8	integrated	R7	2.0
Mali-T628	ARM	4	integrated	T628-4	1.2
Mali-T628	ARM	2	integrated	T628-2	1.2

Table 1: The GPUs we consider, spanning designs from four vendors

benchmark	app. name	description	GPU architecture	source language
Pannotia	*p-sssp*	single source shortest path	AMD Radeon HD 7000	OpenCL 1.0
Pannotia	*p-mis*	maximal independent set	AMD Radeon HD 7000	OpenCL 1.0
Pannotia	*p-colour*	graph colouring	AMD Radeon HD 7000	OpenCL 1.0
Pannotia	*p-bc*	betweenness centrality	AMD Radeon HD 7000	OpenCL 1.0
Lonestar	*ls-mst*	minimum spanning tree	Nvidia Kepler and Fermi	CUDA 7
Lonestar	*ls-dmr*	delaunay mesh refinement	Nvidia Kepler and Fermi	CUDA 7
Lonestar	*ls-bfs*	breadth first search	Nvidia Kepler and Fermi	CUDA 7
Lonestar	*ls-sssp*	single source shortest path	Nvidia Kepler and Fermi	CUDA 7

Table 2: The applications we consider

on GPUs from three vendors, and no paper presented experiments from more than three vendors. The figure also shows a histogram counting the number of papers that conducted evaluation on a GPU from each vendor. Nvidia and AMD are by far the most popular, even though other major vendors (e.g. ARM, Imagination, Qualcomm) all provide OpenCL support for their GPUs. Our investigation suggests that insufficient effort has been put into assessing the guarantees of portability that OpenCL aims to provide.

In this paper, we discuss our experiences with porting and running several open source applications across eight GPUs spanning four vendors, detailed in Tab. 1. For each chip we give the full GPU name, vendor, number of compute units (CUs), specify whether the GPU is discrete or integrated, provide a short name that we use throughout the paper for brevity, and indicate which version of OpenCL the GPU supports (OCL). As Tab. 1 shows, we consider GPUs of different sizes (based on number of compute units), and consider both integrated and discrete chips. We also attempt to diversify the intra-vendor chips. For Nvidia the 980 and K5200 are from different Nvidia architectures (Maxwell and Kepler, respectively). For Intel the 6100 is part of the higher end Iris product line, while the 5500 is part of the consumer HD series. The applications we consider (which are summarised in Tab. 2) are taken from two benchmark suites, Pannotia [9] and Lonestar [8]. For each application we give the benchmark suite it is associated with, a short description, the GPU architecture family the application was evaluated on, and the original source language of the application. We describe the benchmark suites and our motivation for choosing these applications in more detail in Sec. 3.

This report serves to assess the current state of portability for OpenCL applications across a range of GPUs, by detailing the issues that blocked portability of the applications we studied. In this work, we consider *semantic* portability rather than *performance* portability; that is, the issues we document deal with the functional behaviour of applications rather than runtime performance. Prior work has examined and addressed the issue of performance portability for OpenCL programs on CPUs and GPUs (for example [25, 26, 2]); however, we encountered these issues when simply attempting to run the applications across GPUs, without any attempt to optimise runtime per platform. We report on these semantic portability issues in detail, classifying them into three main categories:

- **Framework bugs**, where a vendor-provided OpenCL implementation behaves incorrectly according to the OpenCL specification.
- **Specification limitations**, where the OpenCL specification is unclear and where different GPU implementations exhibit different behaviours.
- **Program bugs**, where the original program contains a bug that we observe to be dormant when the program is executed on the originally-targeted platform, but which appears when the program is executed on different platforms.

Several recent works have raised reliability concerns in relation to GPU programming. Compiler fuzzing has revealed many bugs in OpenCL compilers [19], targeted litmus tests have shown surprising hardware behaviours with respect to relaxed memory [1], and program analysis tools for OpenCL have revealed correctness issues, such as data races, when used to scrutinise open source benchmark suites [3, 10]. In contrast to this prior work, which specifically set out to expose bugs, either through engineered synthetic programs [19, 1], or by searching for defects that might arise under rare conditions [3, 10], we report here on portability issues that we encountered "in the wild". These issues arose without provocation when attempting to run open source applications. In fact, as discussed further in Section 3, the porting effort that led to this study was undertaken as part of a separate, ongoing research project; to make progress on that research project we were hoping that we would *not* encounter such issues. We believe that the "real-world" nature of the issues experienced may be closer to what GPU application developers encounter day-to-day, compared with the issues exposed by targeted testing and formal verification.

Our hope is that this report will make the following contributions to the OpenCL community. For software engineers endeavouring to develop portable OpenCL applications, it can serve as hazard map for issues to be aware of, and suggestions for working around such issues. For vendors, it can serve to identify areas in OpenCL frameworks that would benefit from more robust examination and testing. For researchers, the issues we report on may serve as motivational case-studies for new verification and testing methods.

Despite the challenges we faced, in most cases we were able to find a work-around, and overall we consider our experience a success: OpenCL application portability can be achieved with effort, and this effort will diminish as vendor implementations improve, aspects of the specification are clarified, and better analysis tools become available.

The structure of the paper is as follows: Sec. 2 contains an overview of OpenCL and common elements of a GPU OpenCL framework. The applications we ported are described in Sec. 3. Section 4 documents the issues we classified as framework bugs. Section 5 documents the issues we classified as specification limitations. Section 6 documents the issues we classified as programming bugs. We then suggest ways that we believe the state of portability of OpenCL GPU programs could be improved in Sec. 7. Finally, we conclude in Sec. 8.

2. BACKGROUND ON OPENCL

OpenCL Programming. An OpenCL application consists of two parts: *host* code, usually executed on a CPU, and *device* code, which is executed on an accelerator device; in this paper we consider GPU accelerators. The host code is usually written in C or C++ (although wrappers for other languages now exist) and is compiled using a standard C/C++ compiler (e.g. GCC or MSVC). The OpenCL framework is accessed through library calls that allow for the set-up and execution of a supported device. The API for the OpenCL library is documented in the OpenCL specification [17], and it is up to the vendor to provide a conforming implementation that the host code can link to.

The device code is written in OpenCL C [14] (similar to C99). The code is written in an SIMT (single instruction multiple thread) manner, such that all threads execute the same code, but have access to unique thread identifiers. The device code must contain one or more entry functions where execution begins; these functions are called *kernels*.

OpenCL supports a hierarchical execution model that mirrors features common to some of the specialised hardware that OpenCL kernels are expected to execute on, in particular features common to many GPU architectures. Threads are partitioned into disjoint, equally-sized sets called *workgroups*. Threads within the same workgroup can use OpenCL primitives for efficient communication. For example, each workgroup has a disjoint region of *local memory*; only threads in the same workgroup can communicate using local memory. OpenCL also provides an intra-workgroup execution barrier. On reaching a barrier a thread waits until all threads in its workgroup have reached the barrier. Barriers can be used for deterministic communication. To aid in finer-grained and intra-device communication, OpenCL provides a set of atomic read-modify-write instructions where threads can atomically access, modify and store a value to memory. All device threads have access to a region of *global memory*.

Newer GPUs provide support for the OpenCL 2.0 memory model [17, pp. 35-53], which is similar to the C++11 memory model [13, pp. 1112-1129]. In this model, synchronisation memory locations must be declared with special *atomic types* (e.g. `atomic_int`). Accesses to these memory locations can be annotated with a *memory order* indicating the extent to which the access will synchronise with other accesses (e.g. *release*, *acquire*), and a *scope* in the OpenCL hierarchy to indicate with which other threads in the concurrency hierarchy the access should communicate (e.g. a scope can be intra-workgroup or inter-workgroup). If no memory order is provided, a default memory order of sequentially consistent is used [14, p. 103]. Rules on the orderings provided by these annotations are given both in the standard and (more formally) in recent academic work [5].

While support in OpenCL 2.0 facilitates finer-grained interactions between the host and device, traditionally the host and device interact at a course level of granularity, and this is the case for the applications we consider in this paper. The host and device do not share a memory region, thus the host must explicitly transfer any input data the kernel needs to the device through the OpenCL API. The host is responsible for then setting the kernel arguments and finally launching the kernel, again all using the OpenCL API.

A similar language for programming GPUs is CUDA [21]. This language is Nvidia-specific and thus not portable across GPU vendors.

Components of an OpenCL Environment. To enable OpenCL support for a given device, a vendor must provide a *compiler* for OpenCL C that targets the instruction set of the device, and a *runtime* capable of coordinating interaction between the host and the specific device. It is the role of the OpenCL specification to define requirements that the compiler and runtime must adhere to in order to successfully execute valid applications. It is the vendor's job to ensure that these requirements are met in practice, and clarity in the OpenCL specification is essential to achieving this.

The device, compiler and runtime comprise a complete OpenCL environment. Issues in any one of these components can cause the contract between the OpenCL specification and the vendor-provided environment to be violated.

3. EVALUATED APPLICATIONS

This experience report is a by-product of an ongoing project that explores using the OpenCL 2.0 relaxed memory model [17, pp. 35-53] to design custom synchronisation constructs for GPUs. For that project, we sought benchmarks that might benefit from the use of fine-grained communication idioms. We discovered that applications containing *irregular* parallelism over dynamic workloads provided a good fit for our goals. With this in mind, we found two suites of open source benchmarks to experiment with: Pannotia [9] and Lonestar [8]. The applications are summarised in Tab. 2. The short names of Pannotia and Lonestar applications are prefixed with "p" and "l", respectively.

The Pannotia benchmarks were originally developed to examine fine-grained performance characteristics of irregular parallelism on GPUs, such as cache hit rate and data transfer time. The benchmarks were written in OpenCL 1.0, and evaluated using AMD GPUs. There are six applications in the benchmark suite in total, of which we consider four. The two applications we did not consider were structured in a way such that we could not easily see how to apply our experimental custom synchronisation constructs (recall that applying these constructs was what motivated us to evaluate these benchmarks across GPUs from a range of vendors).

The Lonestar applications were originally written in CUDA and evaluated using Nvidia GPUs; we ported these applications to OpenCL. Like the Pannotia applications, the Lonestar applications measure various performance characteristics of irregular applications, including control flow divergence between threads.

The Lonestar applications use non-portable, Nvidia-specific constructs, including single dimensional texture memory, warp-aware operations (e.g. warp shuffle commands), and a device-level barrier. For each, we attempted to provide portable OpenCL alternatives, changing texture memory to global memory, rewriting warp-aware idioms to use workgroup synchronisation, and using the OpenCL 2.0 memory model to write a device-level barrier. There are seven applications in the Lonestar benchmark suite, of which we consider four. Similar to the Pannotia benchmarks, the three applications we did not consider were structured in a way that we could not easily see how to apply our custom synchronisation constructs.

Both benchmark suites contain an *sssp* application, however they are fundamentally different. The Lonestar version (*ls-sssp*) uses shared task queues to manage the dynamic workload. The Pannotia version (*p-sssp*) is implemented

by iterating over common linear algebra methods. We thus consider them as two distinct applications.

4. FRAMEWORK BUGS

Here we outline three issues that we believe, to the best of our knowledge and debugging efforts, to be framework bugs. We experienced these issues when experimenting with custom synchronisation constructs in the applications of Tab. 2 across the chips of Tab. 1.

For each bug, we give a brief summary that includes a short description of the bug, the platforms on which we observed the bug, the status of the bug (indicating whether we have reported the issue and if so whether it is under investigation) and, if applicable, a work-around. We additionally give each issue a label for ease of reference in the text.

After the summary, we elaborate more about how we came across the issue and our debugging attempts. Where we have not reported the issues, this is due to exposure of the issue requiring use of our custom synchronisation constructs, the fruits of an ongoing and as-yet-unpublished project. Once we publish these constructs, we will report the issues.

Framework bug 1: compiler crash

Summary: The OpenCL kernel compiler crashes nondeterministically.
Platforms: 5500 and 6100 (Intel)
Status: Unreported
Workaround: Add preprocessor directives to reduce the number of kernels passed to the compiler
Label: FB-CC

We encountered this error when experimenting with custom synchronisation constructs in the *p-sssp* application. The original application contained four kernel functions. Using our synchronisation construct, we implemented three new kernel functions, each of which performed some or all of the original computation using different approaches (e.g. by varying the number and location of synchronisation operations). For convenience, we located all seven kernel functions in a single source file.

We noticed that when we executed scripts to benchmark the different kernels, the application would crash roughly one in ten times with an unknown error, producing an output that looks like a memory dump. Our debugging efforts showed that the application was crashing when the OpenCL C compiler was invoked via the OpenCL API function `clBuildProgram`.

In an attempt to find the root cause of this issue, we tried to reduce the size of the OpenCL source file. We were able to reduce the problem to a kernel file that contained only two large kernel functions. At this point, when *either* of the kernel functions were removed, the error disappeared. Our hypothesis is that the error is due to the OpenCL kernel file containing multiple large kernel functions. We were able to work around this issue by surrounding the kernel functions in the kernel file with preprocessor conditionals. We then used the `-D` compiler flag to exclude all kernels except the one we were currently benchmarking.

We have not yet reported this issue as the kernels which cause the compiler to crash contain our custom synchronisation constructs.

Framework bug 2: deadlock with break-terminating loops

Summary: Loops without bounds (using `break` statements to exit) lead to kernel deadlock
Platforms: K5200 (Nvidia), R7, R9 (AMD)
Status: Unreported
Workaround: Re-write loop as a `for` loop with an over-approximated iteration bound
Label: FB-BTL

When experimenting with the Pannotia benchmarks, we found it natural to write the applications using an unbounded loop which breaks when a terminating condition is met (e.g. when there is no more work to process). The following code snippet illustrates this idiom:

```
1  while (1) {
2    terminating_condition = true;
3
4    // do computation, setting terminating_condition
5    // to false if there is more work to do
6
7    if (terminating_condition) {
8       break;
9    }
10 }
```

On K5200, R7 and R9, we discovered that this idiom can deterministically cause non-termination of the kernel. Our debugging attempts led us to substitute the infinite loop with a finite loop with large bounds (keeping the `break` statements). We began with a loop bound of `INT_MAX`. After this change, the applications correctly terminated. To determine if threads were actually executing the loop `INT_MAX` times, we tracked how many times each of the threads executed the loop. We observed that no thread actually executed the loop for `INT_MAX` iterations. That is, each thread terminated early through the break statement.

Given this, we believe that the non-termination in the original code with the infinite loop is due to a framework bug (e.g. a compiler bug). The work-around is to replace `while(1)` loop header with a `for` loop header that uses a large over-approximation of the number of iterations of the loop that will actually be executed.

As with FB-CC, we did not report the issue yet because this example uses our currently unpublished synchronisation constructs. While we do not believe that the issue is related specifically to the new synchronisation constructs, it does seem that a suitably complex kernel is required to cause this behaviour; our attempts to reduce the issue to a significantly smaller example caused the problem to disappear.

Framework bug 3: defunct processes

Summary: GPU applications become defunct and unresponsive when run with a Linux host
Platforms: R7 and R9 (AMD)
Status: Known
Workaround: Change host OS to Windows
Label: FB-DP

In experimenting with new synchronisation constructs in the Pannotia applications we generated kernels that could potentially have high runtimes (around 30 seconds). Most systems we experimented with employed a GPU watchdog daemon (see Sec. 5) which catches and terminates kernels

that have executed for longer than a certain time limit (usually 1–10 seconds).

On most systems, we were able to disable the watchdog and successfully execute our long running kernels. However, for AMD GPUs under a Linux OS, the watchdog could not be disabled and, in fact, was quite disruptive in its behaviour. Specifically, long running kernels were not only terminated, but they became *defunct*. These defunct processes were unable to be killed by any command we were able to find. A defunct GPU application blocks the GPU from executing any new GPU application. The only means to remove the defunct process and allow the GPU to execute applications again was a power cycle on the machine.

Unlike the previous issues, this is a known issue and documented on the vendor's OpenCL distribution webpage.[1] This issue is documented as only affecting GPU applications with a Linux OS host. Thus, to attempt to work around this issue, we switched the OS on the machine to Windows; we did not observe the error after this switch. Due to the severity of this issue (i.e. applications simply not being runnable and requiring a reboot to restore the GPU), we hope the issue is fixed soon.

5. SPECIFICATION LIMITATIONS

In this section we document issues we encountered that we believe to be genuine specification limitations. That is, these issues arise from constructs whose semantics is not precisely dictated by the OpenCL specification, leading to different platforms exhibiting different behaviours. While these do not necessarily correspond to application or specification bugs, they are behaviours we found interesting and, in some cases, surprising. Each issue required investigating erroneous application behaviours on certain platforms when the same application ran as expected on other platforms.

Specification limitation 1: device memory allocation failures

Summary: Allocating device memory fails nondeterministically
Platforms: 5500 and 6100 (Intel)
Workaround: Self-manage large chunks of memory
Label: SL-DMAF

For each GPU application (and input data-set) we experimented with, we identified the memory requirements for both the host and device. We then only ran applications and input data-sets on hosts (and GPUs) with sufficient resources to satisfy the application requirements. One such application requirement is the maximum amount of memory that can be allocated at a time. A platform can check if it satisfies this requirement by querying the device attribute named `CL_DEVICE_MAX_MEM_ALLOC_SIZE` through the OpenCL API. Because for Intel GPUs global device memory is mapped to system DRAM, the total application memory (host and device) needs to be less than amount of system DRAM [12, p. 15].

When running experiments, we noticed that occasionally GPU memory allocations would fail on Intel platforms, returning the error `CL_MEM_OBJECT_ALLOCATION_FAILURE`. This error occurs despite the fact that the total available RAM and global memory on the host and GPU exceeds the memory requirements of the application. The available memory on the host can be obtained by examining the Windows performance monitor utility and the total GPU global memory can be determined through the `CL_DEVICE_GLOBAL_MEM_SIZE` value. In an effort to reduce the application to a smaller test case, we found that this behaviour arises when allocating large chunks of host memory (e.g. through `malloc`) before the failing device memory allocations. The total amount of host memory allocated was ~1.2GB while the total amount of device memory allocated was ~400MB. The host machine runs Windows 10 with 8GB total memory; when idling ~4GB of memory is free.

We were able to ask an Intel representative about this behaviour. The response was that memory allocation was failing due to the fragmentation of previously allocated memory. That is, the previously allocated memory was positioned in such a way that there did not exist a contiguous piece of memory large enough to satisfy the memory allocation request. The solution proposed to us was to modify our code to perform a small number of large allocations, rather than many smaller allocations, and to manually write memory management code to divide the large allocated chunks of memory into the smaller chunks used by the program.

This issue shows that even though a machine has the resources required to run a particular OpenCL application, the application may still nondeterministically fail to allocate memory unless the memory is explicitly user-managed. Luckily this issue is not fatal and can be caught by checking the result of every device memory allocation.

Because of the potential for introducing bugs using low-level user managed data, we did not add this fix to our application and instead opted to simply detect and re-run executions that failed due to this issue until the memory allocation succeeded. Applications that require a higher degree of robustness (e.g. applications that cannot tolerate resets) will likely not be able to take this approach.

Interestingly, while conceptually it would seem that this issue could appear on any system, we observed it only for 5500 and Iris. The machine containing 5500 also contains a small Nvidia discrete GPU (GeForce 940m). Even though the Nvidia chip has less device memory than the 5500, the Nvidia chip reliably runs these applications with no issues.

Specification limitation 2: GPU watchdogs

Summary: Long running GPU kernels are killed by a watchdog
Platforms: All platforms running a display server (e.g. X11 on Linux or any Windows OS)
Workaround: Run on platform with no graphics layer or modify the watchdog
Label: SL-GW

Rendering of the OS GUI (e.g. Windows explorer or X11 on Linux) is frequently achieved using a GPU. Thus, if a GPU application does not terminate, the OS GUI can become unresponsive resulting in a "freeze". In our experience, on Windows operating systems this requires a reboot to terminate the GPU application; on Linux it can usually be resolved by opening a virtual console and manually killing the GPU application.

To avoid having to use these disruptive measures to kill non-terminating GPU applications, operating systems with

[1] see: http://developer.amd.com/download/AMD_APP_SDK_Release_Notes_Developer.pdf

a GPU-driven GUI often have a GPU *watchdog* daemon, which kills GPU activities that run for longer than a certain time limit. The granularity at which the GPU activity is killed varies across platforms. For example, the GPU watchdog in Windows kills the entire process that launched the long running kernel (i.e. the entire OpenCL application) while the X11 watchdog only kills the kernel, and an error code is returned to the OpenCL host. By default, the time limit threshold is relatively short, usually only several seconds. The pragmatic advantage of a watchdog is obvious: it protects against system freezes. The price for this benefit is that the watchdog prohibits execution of long-running, but terminating, GPU kernels.

In our experiments with the Pannotia and Lonestar applications, we developed kernels that executed for far longer than allowed by each of the watchdogs of our test systems, leading to these applications (or kernels in the applications) being killed. The method for "taming" the watchdog was different for each OS. On Windows, the watchdog can be straightforwardly configured through the Windows registry TDR keys[2] (requiring a reboot to register a new configuration). On Linux (using X11), the watchdog can be modified through the Xorg configuration file[3] and requires only the X11 service to be restarted.

We encountered two cases where watchdog taming was not straightforward. First, the AMD OpenCL framework appeared to contain a watchdog bug under Linux that prevented GPU applications from terminating correctly; we document this Sec. 4 as issue FB-DP. The workaround for this was to load a Windows OS on the machine containing AMD GPUs. Second, the machines containing the ARM GPUs used a custom Linux-based OS with a custom GUI. For this OS, we ascertained that the watchdog logic was hard-coded into the GPU driver and there existed no way to disable (or modify the time threshold of) the watchdog. To work around this issue, we loaded a traditional Linux OS on the machine; this allowed manual watchdog control.

Variations between operating systems related to default timing thresholds and mechanisms for configuring and disabling GPU watchdogs make it difficult to produce portable GPU applications that invoke long-running kernels. This is especially difficult given the different granularities at which GPU activities are killed. Because watchdog properties are not specified by the OpenCL SDK, there is no portable method, to our knowledge, by which an application can be programmed defensively to guard against watchdog effects.

Specification limitation 3: Occupancy vs. compute units

Summary: The number of reported compute units cannot safely be used as a lower bound on workgroup occupancy when using a persistent thread model
Platforms: 5500 and 6100 (Intel)
Workaround: Use trial and error to determine number of persistent threads
Label: SL-OCU

OpenCL does not provide traditional forward progress guarantees between workgroups [17, p. 31]:

> A conforming implementation may choose to serialize the work-groups 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 work-groups since once in the work-pool, they can execute in any order.

Because workgroups may be serialised, an OpenCL application can deadlock if a scheduled workgroup spins waiting for a workgroup that is not scheduled. A proposed workaround is known as the *persistent thread model*, which assumes that GPU kernels launched with at most as many workgroups as can be occupant on the GPU will have traditional forward progress guarantees [11]. Using the experimentation described in [11], we found that the GPUs in our study appear to satisfy the persistent thread model. Therefore, by determining the workgroup occupancy of the GPU and kernel (as the occupancy depends on the resources used by the kernel, e.g. local memory), we can successfully implement applications that rely on forward progress between occupant workgroups.

The Lonestar applications contain a custom device-level execution barrier that assumes kernels are launched with at most the number of persistent workgroups, otherwise the execution barrier will deadlock. When experimenting with these applications across vendors, we assumed that the number of reported GPU compute units would provide a safe lower bound on the number of the number of persistent workgroups. Our assumption was based on documentation from various sources. For example, the OpenCL optimisation guide from Intel states:[4]

> An OpenCL device has one or more compute units. A workgroup executes on a single compute unit.

Our interpretation of this quote led us to believe that (at least in the case of Intel) the GPU can be occupied by at least as many workgroups as there are compute units.

For most platforms we experimented with, we found that the number of compute units was a safe (although in some cases not tight) estimate of the persistent workgroups. That is, a device level barrier could safely be executed with kernels running as many workgroups as compute units. The one exception was in the case of GPUs from Intel (i.e. 5500 and Iris). For these GPUs we found that the reported number of compute units was much higher than the number of workgroups that could safely execute a device-level execution barrier.

In particular, 5500 and Iris report 24 and 47 compute units respectively from the device query OpenCL API (Querying the `CL_DEVICE_MAX_COMPUTE_UNITS` attribute). The *ls-bfs* application, which contains a device-level barrier, will deadlock on these chips if run with the number of workgroups equal to the number of compute units (and with 256 threads per workgroup). However, the same application will run without error if executed with 9 and 18 workgroups for 5500 and Iris respectively. We determined these numbers through trial and error; that is, launching the application with successively fewer workgroups until we did not observe deadlock. This method is arduous as deadlocks

[2] see: https://msdn.microsoft.com/en-us/library/windows/hardware/ff569918(v=vs.85).aspx

[3] see: http://nvidia.custhelp.com/app/answers/detail/a_id/3029/~/using-cuda-and-x

[4] from https://software.intel.com/sites/landingpage/opencl/optimization-guide/Basic_Concepts.htm

on these machines required a reboot to restore the GPU. Additionally this method is not robust. Kernels that use more resources (e.g. more local memory) have fewer persistent workgroups. For example, a test application that uses the maximum amount of available local memory is only able to run with at most 3 and 6 workgroups for 5500 and 6100 respectively without deadlock.

This experience shows that applications relying on the persistent thread model cannot use the number of compute units as a safe estimate for occupancy.

Specification limitation 4: memory consistency

Summary: Pre OpenCL 2.0 GPUs do not support some inter-workgroup communication.
Platforms: Pre OpenCL 2.0 GPUs
Workaround: None
Label: SL-MC

The memory model of a device determines which values load operations are allowed to return, e.g. stored values may be temporarily held in a private store buffer or cache and not visible to all other threads. Typically, operations known as *fences* can be used to enforce ordering between memory accesses, e.g. by flushing the local cache. Because synchronisation constructs often require fine-grained interactions between threads, the memory model must be considered when implementing such constructs. Because our aim was to experiment with custom synchronisation constructs in the Pannotia and Lonestar applications, we required a portable and well-documented memory model to reason about these applications using our constructs.

While the OpenCL 1.x (1.0, 1.1, and 1.2) specification includes memory fence functions, their behaviour is not well documented. For example, the specification for OpenCL 1.0 contains only three memory fence instructions and one page of documentation [15, p. 200]. The other OpenCL 1.x versions do not change in with respect to memory fences from 1.0. In contrast, the C++11 memory model has 17 pages of documentation and several examples [13, pp. 1112-1129]. The OpenCL 2.0 specification describes a memory model similar to that of C++11, which allows for well-defined fine-grained communication between threads. Unfortunately, OpenCL 2.0 is not widely supported currently. To experiment on a wide range of chips, we attempted to manually implement a subset of the OpenCL 2.0 atomic operations and fences, which we use as building blocks for our custom synchronisation constructs.

We produced two custom OpenCL 2.0 memory model implementations. The first implementation uses fences available in OpenCL 1.x for synchronisation; therefore this implementation can be executed by any chip supporting OpenCL. Our second implementation uses Nvidia-specific constructs, namely inline PTX [22] (the low-level intermediate language for Nvidia chips). This implementation can only be executed on Nvidia chips; however, the memory model for Nvidia PTX has been empirically investigated (see [1]), the results of which we used to inform our implementation.

Given that the OpenCL 2.0 memory model is similar to C++11, we based our OpenCL 2.0 implementation on an existing compilation mapping of C++11 atomic operations to PowerPC assembly instructions given in [6]. Namely, our implementation provides the same placement of fences as the C++11 to PowerPC compiler mapping, except using OpenCL 1.0 fences (or PTX fences [22, p. 187] for Nvidia chips) instead of PowerPC fences. In cases when we were unsure, we erred on the side of safety, e.g. we added the strongest available fences to attempt to guarantee release/acquire semantics for atomic operations equipped with these memory order annotations.

chip	vendor	CL version	passed tests
GTX 480	Nvidia	OpenCL 1.1	No
Tesla C2075	Nvidia	OpenCL 1.1	No
GTX 980	Nvidia	OpenCL 1.1	Yes
Quadro K500	Nvidia	OpenCL 1.1	Yes
HD 4400	Intel	OpenCL 1.2	No
Radeon HD 7970	AMD	OpenCL 1.2	No
Radeon HD 6570	AMD	OpenCL 1.2	No
Mali-T628	ARM	OpenCL 1.2	Yes
Mali-T628	ARM	OpenCL 1.2	Yes

Table 3: Pre OpenCL 2.0 chips we tested for consistent inter-workgroup communication

To test our custom OpenCL 2.0 memory model implementation, we wrote some custom synchronisation constructs (e.g. mutexes and concurrent data-structures) with accompanying unit tests. Chips that supported OpenCL 2.0 passed the unit tests (as expected). We then ran the tests on a variety of pre OpenCL 2.0 chips via our custom memory model implementation. The chips we tested and the outcomes are listed in Tab. 3. The chips that passed these tests were considered in our overall study and are also given in Tab. 1. The chips that failed were not considered any further in our study as we found no way to enforce reliable inter-workgroup synchronisation. The chips common to Tab. 1 and Tab. 3 are the chips that do not support OpenCL 2.0, but passed our memory model unit tests. The chips in Tab. 1 but not in Tab. 3 are chips that already support OpenCL 2.0.

The two Nvidia chips that failed (Tesla C2075 and GTX 480) were shown in previous work to not support inter-workgroup communication regardless of synchronisation (fences) [1]. This was hypothesised to be due to incoherent caches; that is, the L1 cache (shared within a workgroup) and the L2 cache (shared over all workgroups) did not maintain a consistent state. We speculate that the other chips that failed may have similar characteristics. The chips that passed the unit tests under our custom OpenCL 2.0 implementation have continued to reliably execute all our other experiments.

This experience shows that custom implementations of OpenCL 2.0 primitives can behave substantially differently across different platforms. Developers who are programming for platforms that do not support OpenCL 2.0 but use some OpenCL 2.0 constructs should tread carefully, even if they have a well-informed custom implementations.

Specification limitation: floating-point accuracy

Summary: Platforms may compute with differing floating-point accuracy
Platforms: Intel GPUs compared to Nvidia GPUs
Workaround: None
Label: SL-FPA

The *ls-dmr* application was originally written to use double-precision floating-point values; however only four of the GPUs we considered support double-precision: 980, K5200, R7, and R9 (this can be checked through the `cl_khr_fp64`

chip	Linux	Windows
R9	Limited	Yes
R7	Limited	Yes
T628-2	Yes	No
T628-4	Yes	No

Table 4: GPU Operating system portability

extension). To try to obtain preliminary results for the other GPUs, we experimented with changing the double-precision values to single-precision values. Our hope was *ls-dmr* would be safe with respect to this this change and we could test the feasibility of custom synchronisation constructs in this application. The application contains a validation check upon completion, which we use to determine success or failure.

The first platform we attempted to run the single-precision variant of *ls-dmr* on was 5500. On this platform, the single-precision variant executed without issue, terminating successfully and passing the validation check. We were thus hopeful that the single-precision variant would be robust enough to run on other platforms.

We next tried the single-precision variant on K5200; although this device supports double-precision, we wanted to access the behaviour (e.g. performance and correctness) of the single-precision variant. We observed the application deterministically returned error codes (e.g. due to out of bounds memory accesses) or did not terminate. We conclude that on this platform, single-precision is not sufficient for this application. Thus, even though single-precision appeared to work on 5500, it reliably failed on K5200.

This experience shows that the behaviour of applications can change across platforms, even when the same floating-point precision is used. Floating-point representation and accuracy for GPUs have been documented and discussed in previous work (for example, Nvidia floating-point properties are discussed in [27]). Additionally, the OpenCL C specification provides several avenues for implementation defined behaviour, as illustrated by the following quote [14, p. 176]

> Support for denormalized numbers with single-precision floating-point is optional. Denormalized single-precision floating-point numbers passed as input or produced as the output of single-precision floating-point operations . . . may be flushed to zero.

These implementation defined behaviours can lead to differences in application behaviours with respect to floating-point computations. That is, the deliberate flexibility allowed for floating-point precision in the specification can provide portability difficulties in GPU applications.

Due to the complex nature of this application (over 700 lines of code, including complex numerical routines and synchronisation primitives) we were unable to pinpoint the exact location where inaccuracies arise or how to work around them. Thus we were only able to run this application on GPUs which supported double-precision computations.

Specification limitation 6: operating system portability

Summary: Applications written to target a specific OS (e.g. Linux) are generally not immediately portable to another OS (e.g. Windows). Because of OS constraints on GPUs, this hinders application portability.
Platforms: R7, R9 (AMD), T628-4, T628-2 (ARM)
Workaround: extend code to use OS portable constructs
Label: SL-OSP

In our experiences attempting to run applications across a variety of GPUs, we found that some GPUs were only accessible (or reliable) through a certain operating system. This means that for applications to run on these "OS-constrained" GPUs, the host code must be portable across multiple operating systems. This is in contrast to our experiences with CPUs, which are typically compatible with multiple mainstream operating systems.

We summarise operating system compatibility for the GPUs used in this study in Tab. 4. This table lists the chips of Tab. 1 for which we experienced issues with OS support. For the other chips, while support for both Linux and Windows is available, we only experimented only with a single operating system as the available OS didn't cause issues. For R7 and R9, we began our experiments on Linux, but due to the defunct process framework bug (see Sec. 4 issue FB-DP), we had to change the host operating system to Windows. For ARM GPUs, there is no option to run Windows on the host machine. This shows that at least half of our GPUs required a particular (and distinct) OS to successfully run OpenCL GPU applications. The OS constraints could possibly apply to more of our chips as we did not test (a) Intel chips on Linux or (b) Nvidia chips on Windows.

The Pannotia and Lonestar applications were both developed targeting the Linux operating system; however, we required that the applications run under Windows (for R7 and R9). We found that none of the applications were portable to Windows as provided and we had to add macros for OS portability. The issues we encountered were largely in file I/O and timing functionality. We additionally found several more portability issues:

- The `endian.h` header is not standard on Windows. We had to provide a custom implementation of several functions.
- Windows provides a non-portable definition of `max` and `min` that clash with the `std`-provided `max` and `min`

After identifying these issues, we were able to successfully extend the code to be portable across operating systems.

We found that the Pannotia applications contained a bug in file I/O which was silent on Linux but caused a fatal runtime error for Windows. We have contacted the authors who are looking into these issues.

6. PROGRAMMING BUGS

Here we describe three issues that we class as *programming bugs*. That is, the application code contained bugs that could arise from (a) valid (and well-specified) OpenCL executions that the application did not account for, or (b) executions leading to undefined behaviour (and hence no

guaranteed semantics). However, these bugs did not manifest themselves frequently (or at all) on the original target platform. When ported to another platform these bugs manifested more frequently, causing incorrect results or crashes. Thus, although program bugs on one platform may be dormant, the same bugs can have serious consequences when executed on a different platform.

Programming bug 1: data races

Summary: Applications contain harmful data races that do not appear on the originally-targeted platform, but do appear after porting to another platform
Applications: *ls-bfs* and *ls-sssp*
Status: Reported and acknowledged
Workaround: Add additional synchronisation constructs
Label: PB-DR

This issue was encountered when porting the *ls-bfs* and *ls-sssp* applications to OpenCL, as they were originally written in CUDA. These applications use a custom device execution barrier construct that allows deterministic communication across workgroups if memory accesses are separated (i.e. synchronised) with this barrier. If these accesses are not synchronised, communication may be nondeterministic. We observed these applications contain data races that could lead to computation of incorrect results.

The data races we discovered are present in the original CUDA versions of the *ls-bfs* and *ls-sssp* applications, but we only observed errors when running the ported OpenCL variant. Specifically, the CUDA variant executed reliably on the 980 but the OpenCL variant exhibited errors roughly one in twenty executions when executed on the same 980.

To explain the data races, we show the high level steps of the two applications, which follow the same pattern. Each application uses two global task lists, one for input tasks from which threads take work, and one for output tasks to which threads push new work. The steps the application takes are then as follows:

1. The threads cooperate to process work from the input task list and push new work to the output task list;
2. The threads synchronise using an execution barrier;
3. Local pointers to input and output task lists are swapped, so that the input task list becomes the output and vice versa;
4. The output task list is reset; this was previously the input task list whose work was computed in step 1;
5. If input task list is empty the application terminates, otherwise the process repeats starting at step 1.

The data race we discovered occurs between steps 5 and 1 (when the steps repeat). Specifically, all workgroups synchronise at step 2. At this point one workgroup (a say) may execute steps 3–5. Workgroup a then observes that the input task list is not empty and continues to step 1 in which it pushes some work to the output task list. Another workgroup (b say), which may have been stalled at step 2 during this time, then begins to execute. Workgroup b executes steps 3 and 4. While executing step 4, b resets the output work list that a has previously pushed work to. Thus the work pushed by a is lost. This leads to incorrect results in the application. The solution is to add another execution barrier after step 4.

We have reported these data races to the developer who has acknowledged the issue. The developer mentioned that rather than placing another (expensive) barrier inside the loop, they are devising a new way to manage the task queues, which neither contains the race nor requires the barrier.

Programming bug 2: stability

Summary: Applications that fail rarely (deemed acceptable by the developer) on one platform, fail frequently on a different platform
Applications: *ls-dmr*
Status: Acknowledged
Workaround: None

This issue was encountered when running experiments on *ls-dmr*. We observed that even the original CUDA application failed rarely (roughly one in a hundred runs) when executed on Nvidia chips. These failures were in the form of kernel failures that can arise e.g. due to out of bounds memory accesses, or deadlocks. We spoke to a developer who confirmed that these occasional failures were known but deemed acceptable due to the nature of the computation.

In our OpenCL port, we observed roughly the same failure rate on Nvidia chips. However, when we attempted to run the application on AMD chips (R7 and R9) we observed failures on *every* execution. Thus *ls-dmr* is completely unusable on these platforms. We were unable to run this application on other GPUs as it requires double-precision floating-point arithmetic, which is only supported on the Nvidia and AMD chips we consider.

We see both a positive and negative aspect to this result. On the negative side, applications that were deemed stable enough on one platform may become unusable when run on a different platform. On the positive side, if the common failures on AMD are the cause of the uncommon failures on Nvidia, then the application would be much easier to debug on an AMD platform where the errors are frequent.

We did not attempt to debug this issue as the application is complex and modifying it would require domain knowledge outside the range of our expertise. The kernel contains over 700 lines of code, including complex numerical routines and custom synchronisation constructs. We have reported our experiences to the developer, hoping that our insights could aid in producing a fix.

Programming bug 3: structs containing pointers as kernel arguments

Summary: Struct kernel arguments containing global memory pointers behave as expected on some platforms and fail on others
Applications: All Lonestar OpenCL ports
Status: Acknowledged
Workaround: Pass all global memory buffers to kernel individually
Label: PB-KA

The Lonestar benchmarks are originally provided in CUDA; we then created OpenCL ports to investigate these applications across multiple vendors. The original CUDA applications utilised structs to encapsulate logically related data.

For example, a *graph* struct contains several arrays describing nodes and edges. The *graph* struct is initialised on the host and then copied to the GPU global memory, still in the *graph* struct form.

In porting the Lonestar applications to OpenCL, we attempted to use the same struct-based approach. The analogous OpenCL structs contained pointers to global memory that were set to particular global memory addresses (previously allocated on the host) by a distinct initialisation kernel. We then passed the (initialised) struct to the kernel containing the main computation of the application.

In our experience, all of our OpenCL Lonestar applications ports behaved as expected across all chips using this data-structure idiom *except* for *ls-mst* on ARM chips. For these GPUs we observed that executions of this application would frequently produce results that failed the application post-condition (roughly 75% of the runs).

The main computation in the *ls-mst* application happens in a host-side loop that enqueues several kernels. We attempted to debug further by recording intermediate values at each iteration of this host-side loop. However, when we instrumented the code to record intermediate values, we observed that the error disappeared and the application would deterministically return the expected result. In fact, any delay placed inside the computation loop (e.g. a print statement) tended to cause the application to behave as expected.

We attempted several other debugging methods. First, we attempted *sequentialising* the program (i.e. reducing the number of GPU threads to one). A well-defined sequential program with static inputs should produce deterministic outputs; however the application continued to provide non-deterministic outputs. We then used the OCLGRIND tool [24] to check for errors (e.g. memory safety violations). The tool reported no errors with the application.

Give that (1) this issue was only observed on ARM GPUs and (2) our debugging attempts did not reveal any programming bugs, we believed that the issue was a framework bug (see Sec. 4) and we contacted ARM representatives. After some discussion, we discovered that the Lonestar data-structure idiom (described in the previous paragraphs) appeared to be undefined in OpenCL. Specifically, the OpenCL spec states [16, p. 235]:

> Arguments to kernel functions that are declared to be a struct or union do not allow OpenCL objects to be passed as elements of the struct or union

However, despite this quote disallowing the Lonestar data-structure idiom, our Lonestar OpenCL ported applications behaved exactly as expected on all platforms except ARM. To the best of our knowledge, CUDA does not have the same restrictions on data-structure kernel arguments. Thus the original CUDA Lonestar applications would not have this issue. We are now aware of this issue in our OpenCL ports of the Lonestar applications and will address them by removing struct kernel arguments and replacing them with multiple non-struct arguments containing the fields of the original structs. Because this specification rule is quite constraining (requiring verbose changes to an otherwise natural solution), we believe that this issue could also be seen as specification limitation. If the OpenCL specification relaxed this constraint to allow struct arguments containing global memory pointers, then OpenCL applications could be better engineered with more data encapsulation and less calls to the verbose OpenCL API.

7. LOOKING FORWARD

Despite the issues we encountered, we consider our portability experiences successful. That is, we were able to experiment with a variety of custom synchronisation constructs across a wide range of different GPUs (as this was our initial aim). However, we believe that we as a community of academics, vendors, and application developers can work together to improve the current state of portability of OpenCL GPU applications.

In this section we identify several areas in which we believe innovation could help to to address some of the specification limitations we encountered, in some cases making concrete proposals for action. We discuss the current work that we are aware of in each area, and hope that our discussion will motivate future work.

7.1 Conformance Tests

In order to use the OpenCL or logo on a product, a vendor implementation must execute and pass the OpenCL conformance tests (available to OpenCL adopters). These tests serve to ensure that potential platforms correctly implement the OpenCL API. However given our experiences and results in previous work (e.g. [19]), we believe that the existing conformance tests could be extended to catch some of the framework bugs we experienced.

As a concrete example, two of the framework bugs we encountered (FB-CC and FB-BLT in Sec. 4) are likely compiler bugs. Recent work on OpenCL compiler fuzzing has produced tests which revealed a surprisingly large number of OpenCL compiler bugs [19]. These types of tests could be integrated into the conformance test suite.

Another example is the memory model tests recently developed for Nvidia chips [1]; similar tests could be generated for the OpenCL 2.0 memory model. While we did not directly observe errors related to OpenCL 2.0 memory model operations, we still believe such tests could be useful, e.g. allowing developers to check custom implementations of the OpenCL 2.0 memory model. In PD-MC (Sec. 5) we noticed that for some chips we were able to successfully provide a custom implementation and for others we were not. It would be useful to test our custom implementations in depth.

Our proposal: public conformance tests. The OpenCL conformance tests are not publicly available; obtaining them requires membership of the Khronos Group, which is not free. Furthermore, we believe that for legal reasons and various technical reasons (e.g. legacy device support), it may be difficult and time consuming to influence the official conformance tests. As an alternative, we suggest that the community develops a public and open source set of *supplementary conformance tests*, to which anyone can contribute tests through a process whereby candidate tests are reviewed by members of the community for technical suitability.

7.2 Specification Details

We found the most difficult aspect of experimenting with applications across a variety of platforms relates to areas where the OpenCL specification is unclear; in particular the issues discussed in Sec. 5. A lack of clarity inevitably leads to differences in the way multiple vendor implementations behave. We believe the specification should allow for clarity

in some of these areas without necessarily becoming more restrictive. This could be realised, for example, through optional extensions that can be queried at runtime.

One of the more disruptive aspects we had to deal with was the GPU watchdog (see PD-GW in Sec. 5). We found no mention of the watchdog in the OpenCL specification, yet every system with a graphical operating system contained a watchdog. Dealing with the watchdog also varied between systems, and for half of the GPUs we experimented with (R7, R9, T628-4, T628-2) required a completely new OS to be loaded in order to circumvent the watchdog.

Our proposal: API support for GPU watchdogs. Our experience would have been much simpler if the watchdog could be controlled (or at least queried) through the OpenCL API. An OpenCL extension could enabled this.

A concern we have in relation to the custom synchronisation primitives we are developing is whether our assumptions about the OpenCL execution model are valid. Specifically, when experimenting with a device-wide execution barrier, we assume the persistent thread model [11]. While the OpenCL specification now gives memory ordering rules for inter-workgroup communication through the memory model, it gives no execution model for inter-workgroup interactions. Empirically, it appears all GPUs we experimented with implement the persistent thread execution model, but we do not know whether this is certainly the case, nor whether it is the case by design. Additionally, as seen in PD-OCU (Sec. 5), there are seemingly no guarantees on how workgroups are mapped to compute units. Defining an inter-workgroup execution model would (a) help application developers to know if the assumptions they are making are valid, (b) allow vendors to expose features of their devices that developers can exploit to achieve high performance, and (c) provide authors of testing and verification tools with details of execution model constraints with respect to which programs must be checked.

Our proposal: API support for querying execution model properties. We propose that an inter-workgroup execution model should be incorporated into OpenCL, through API functions that allow an application to query whether a device offers a persistent thread model, and to elicit details related to the mapping of work groups to compute units for a given kernel. Our proposal could initially be addressed via an OpenCL extension.

7.3 Verification and Testing Tools

We believe GPU verification and testing tools can provide a way to detect and aid in repairing programming bugs. Specifically, such tools can aid in detecting and fixing the sorts of bugs we report on in Sec. 6. Our experiences with defects that manifest on some platforms but not others suggest that analysis tools should aim to detect executions that are buggy according to the OpenCL specification, whether or not the executions are observed on a particular device. This, combined with robust conformance tests, will help ensure that applications are portable.

For OpenCL we are aware of several current tools, each with their own strengths and shortcomings.

The Oclgrind tool is a dynamic analyser that checks for various issues including out of bounds memory accesses and data races [24]. It additionally can check validity of OpenCL API calls on the host. The tool has been designed so that it can be extended with further analyses, a recent example being a state-of-the-art analysis that detects reads from uninitialised variables [23]. Because Oclgrind is a dynamic analysis tool, the errors it reports are always genuine bugs. A downside of this is that Oclgrind can only detect errors that are exposed by a concrete execution on a given program input, with respect to a particular choice of thread interleavings. Thus Oclgrind cannot guarantee the absence of bugs.

A more formal tool, GPUVerify, uses verification condition generation to attempt to formally prove properties of OpenCL kernels [7]. It checks for safe memory accesses and data races. The engine assumes a data race free model of OpenCL (different from the more recent OpenCL 2.0 memory model) which guarantees deterministic runs. Under this assumption, GPUVerify is able to efficiently check all interleavings of threads executing a kernel. Modulo some practical caveats, GPUVerify is a sound analyser—the tool will catch all possible data race bugs. On the downside, GPUVerify may also report false alarms, that is, report errors that are not concretely bugs. Another consideration is that many of the custom synchronisation primitives we experiment with violate GPUVerify's data race-free assumption, and GPUVerify would need to be extended to reason about these programs; currently the tool has only limited support for reasoning about atomic operations [4].

We also mention related tools for CUDA. Similar to Oclgrind, cuda-memcheck (provided by Nvidia) dynamically checks for out of bounds memory accesses and has an option to check for data races [20]. The GKLEE tool provides similar checks but has the ability to model symbolic inputs and explores all possible interleavings [18]. GKLEE assumes the same data race-free model as GPUVerify, thus would need to be adapted to handle custom synchronisation constructs.

Status and future of tools. In this experience report, the only tool we were able to leverage was Oclgrind to check for memory safety. We were unable to use GPUVerify because of the lack of support for atomic operations, which our custom synchronisation idioms relied on. Our hope is that these tools will continue to be developed and adapted to the new features and programming styles of OpenCL. It may encourage developers to use such tools more frequently if they are supported by, and easily integrated into, vendor developer tools. One such example is the integration of GPUVerify into the ARM Mali Graphics Debugger.[5]

8. CONCLUSION

We have shared our experiences arising from porting several open source applications to run across a variety of OpenCL GPU platforms. In our efforts we encountered several issues, which we divided into three categories: framework bugs, specification limitations, and programming bugs. In most cases, we were able to work around the issues and largely consider our experience a success. However, we do believe that the current state of portability in OpenCL programs could be improved. We hope that our suggestions for improvements in relation to OpenCL portability issues will

[5]See: https://community.arm.com/groups/arm-mali-graphics/blog/2015/04/14/debugging-opencl-applications-with-mali-graphics-debugger-v21-and-gpuverify

aid the OpenCL community in realising the full potential offered by the programming model.

9. ACKNOWLEDGEMENTS

Thanks to Daniel Liew (Imperial College London) for comments and feedback, Sven van Haastregt and Marco Cornero (ARM) for discussions and comments around our ARM results, Sreepathi Pai for discussions around the Lonestar benchmarks, and Shuai Che for discussions around the Pannotia benchmarks. Some of the equipment used in this study was supported by a grant from GCHQ.

10. 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] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. van Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. PENCIL: A platform-neutral compute intermediate language for accelerator programming. In *PACT*, pages 138–149, 2015.

[3] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, and S. Qadeer. Engineering a static verification tool for GPU kernels. In *CAV*, pages 226–242. Springer, 2014.

[4] E. Bardsley and A. F. Donaldson. Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels. In *NFM*, pages 230–245. Springer, 2014.

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

[6] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *POPL*, pages 509–520. ACM, 2012.

[7] 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.

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

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

[10] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic crosschecking of data-parallel floating-point code. *IEEE Trans. Software Eng.*, 40(7):710–737, 2014.

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

[12] Intel. Opencl optimization guide, 2014. https://software.intel.com/sites/default/files/managed/72/2c/gfxOptimizationGuide.pdf.

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

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

[15] Khronos Group. The OpenCL specification, version 1.0. https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf.

[16] Khronos Group. The OpenCL specification, version 1.2. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[17] Khronos Group. The OpenCL specification, version 2.0. 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] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *PLDI*, pages 65–76. ACM, 2015.

[20] Nvidia. CUDA-memcheck. https://developer.nvidia.com/CUDA-MEMCHECK.

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

[22] Nvidia. Parallel thread execution ISA: Version 4.3, March 2015. http://docs.nvidia.com/cuda/pdf/ptx_isa_4.3.pdf.

[23] M. Pflanzer, A. F. Donaldson, and A. Lascu. Automatic test case reduction for OpenCL. In *IWOCL*. ACM, 2016.

[24] J. Price and S. McIntosh-Smith. Oclgrind: An extensible opencl device simulator. In *IWOCL*, pages 12:1–12:7. ACM, 2015.

[25] J. Shen, J. Fang, H. J. Sips, and A. L. Varbanescu. Performance gaps between OpenMP and OpenCL for multi-core CPUs. In *ICPP Workshops*, pages 116–125. IEEE Computer Society, 2012.

[26] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *ICFP*, pages 205–217. ACM, 2015.

[27] N. Whitehead and A. Fit-florea. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs, 2015. http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf.