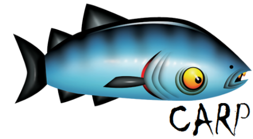
CARP



D2.2.B: Defect Survey

Grant Agreement:	287767
Project Acronym:	CARP
Project Name:	Correct and Efficient Accelerator Programming
Instrument:	Small or medium scale focused research project (STREP)
Thematic Priority:	Alternative Paths to Components and Systems
Start Date:	1 December 2011
Duration:	36 months
Document Type[1]:	D (Deliverable)
Document Distribution[2]:	PU (Public)
Document Code[3]:	CARP-ICL-DD-005
Version:	v1.4
Editor (Partner):	A. Betts (ICL)
Contributors:	ICL, REAL, UT, RIGHT, ARM
Workpackage(s):	WP2
Reviewer(s):	ARM
Due Date:	30 September 2012
Submission Date:	10 November 2012
Number of Pages:	23

[1]MD = management document; TR = technical report; D = deliverable; P = published paper; CD = communication/dissemination.
[2]PU = Public; PP = Restricted to other programme participants (including the Commission Services); RE = Restricted to a group specified by the consortium (including the Commission Services); CO = Confidential, only for members of the consortium (including the Commission Services).
[3]This code is constructed as described in the Project Handbook.

Copyright © 2014 by the CARP Consortium.

D2.2.B: Defect Survey

A. Betts[1], A. Donaldson[1], E. Hajiyev[3], M. Huisman[2], G. Kouveli[5], A. Kravets[5], A. Lokhmotov[5], M. Mihelčić[2], D. Takacs[3], T. Virolainen[4]

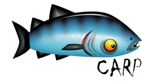
[1]ICL, [2]UT, [3]REAL, [4]RIGHT, [5]ARM

REVISION HISTORY

Date	**Version**	**Author**	**Modification**
2012-08-13	0.1	M. Mihelčić (UT)	Details of OpenCL defects
2012-08-17	0.2	A. Betts (ICL)	Details of CUDA defects
2012-10-15	0.3	G. Kouveli (ARM)	Provided text on SHOC defects
2012-10-19	0.4	A. Betts (ICL)	Split defect survey part of deliverable into separate document
2012-10-23	0.5	A. Kravets (ARM)	Provided text on Rodinia defects
2012-10-30	1.0	A. Betts (ICL)	Completed draft of deliverable for review by ARM
2012-11-02	1.1	A. Lokhmotov (ARM)	Some fixes following review
2012-11-05	1.2	A. Betts (ICL)	Changes to defect survey according to review comments from ARM
2012-11-08	1.3	A. Donaldson (ICL)	Added details of defects discovered in AMD and CUDA SDKs
2014-01-20	1.4	A. Donaldson (ICL)	Added citation

APPROVALS

Role	**Name**	**Partner**	**Date**
Workpackage and Deliverable Leader	A. Lokhmotov	ARM	2012-11-09
Project Coordinator	A.F. Donaldson	ICL	2012-11-09



Contents

1 Executive Summary **4**

2 Background to GPU programming **5**

2.1 Programming model 5

2.2 Execution model 5

3 General GPU kernel defects **7**

3.1 Data races 7

3.2 Barrier divergence 9

4 OpenCL-specific defects **12**

4.1 Memory defects 12

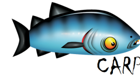
4.2 Miscellaneous defects 13

5 CUDA-specific defects **17**

5.1 Atomics 17

5.2 Warp shuffling 17

6 Examples of defects in open source samples **19**



1 Executive Summary

In recent years, massively parallel *accelerator* processors, primarily General Purpose Graphics Processing Units (GPGPUs) from companies such as AMD and NVIDIA, have become widely available to end-users. Accelerators offer tremendous compute power at a low cost, and tasks such as media processing, medical imaging and eye-tracking can be accelerated to beat CPU performance by orders of magnitude.

Parallel programming present a serious challenge for software developers. A system may contain one or more of the plethora of devices on the market, with many more products anticipated in the immediate future. Applications must exhibit *portable correctness*, operating correctly on *any* accelerator. Software bugs in media processing domains can have serious financial implications, and GPUs are being used increasingly in domains such as medical image processing [11] where safety is critical. Thus there is an urgent need for verification techniques to aid construction of correct GPU software.

In this document, we describe a number of programming idioms specific to GPU programming that can lead to undefined or non-portable behaviour. To that end, we have analysed the OpenCL and CUDA specifications (since these are the dominant programming languages for GPUs) and picked out potential software *defects*, giving examples where appropriate. Note that defects concerned with floating-point arithmetic and alignment are not documented, since they are more general issues not tied exclusively to GPU programming.

We begin with a review of concepts and terminology relevant to GPUs and GPGPU programming (§2). Then we document general defects common to both OpenCL and CUDA programs (§3), before analysing those specific to OpenCL (§4) and CUDA (§5). Finally, we present examples of defects in OpenCL and CUDA kernels which we have encountered during our analysis of the Rodinia and SHOC benchmark suites (see D2.2A), and in the CUDA and AMD SDK samples (§6).

2 Background to GPU programming

To understand the nature of defects in GPU programs, it is first necessary to provide an overview of GPU programming, for which we adopt the nomenclature of the OpenCL programming model. The only exception to this rule is when we outline defects concerning **warps**, which are a CUDA-specific notion that have no equivalent in OpenCL.

Note that, when presenting example code, we adhere to the current OpenCL standard [3] and, where applicable, CUDA syntax as outlined in the CUDA C programming guide [6].

2.1 Programming model

OpenCL provides a high-level abstraction for mapping computation across GPU hardware (also called a **device**), centred around the notion of a **kernel** program being executed by many parallel **work items** (threads), together with a specification of how these work items should be partitioned into **work groups** (group of threads). The kernel is a template specifying the behaviour of an arbitrary work item, parameterised by work item and work group id variables. Expressions over these ids allow distinct work items to operate on separate data and follow differing execution paths through the kernel. Work items in the same work group can synchronise during kernel execution, while work items in distinct work groups execute completely independently.

Each work item has access to four different types of memory:

- **Private memory:** only visible to a single work item.
- **Local memory:** visible to all work items in a single work group.
- **Global memory:** visible to all work items across all work groups.
- **Constant memory:** visible to all work items across all work groups.

Kernels are attached to **command queues**, which allows dependencies between kernels to be enforced.

2.2 Execution model

A GPU typically consists of a large number of simple **functional units**. Subsets of functional units are grouped together into **cores** (or **streaming multiprocessors** in CUDA parlance), which can work independently.

The runtime environment associated with a GPU programming model must interface with the GPU driver to schedule execution of work items across functional units. Each work group is typically assigned to a single core. Thus, distinct work groups can execute in parallel on different cores.

A core executes a work-group by scheduling fixed-sized **sub-groups** of work-items in a round-robin fashion. On some architectures, the sub-group size is one, meaning that a core switches between individual work-items on each cycle (*e.g.* ARM Mali-T600). On other architectures, the sub-group size is a multiple of two (*e.g.* 32 on NVIDIA GPUs) and a core switches between sub-groups executing in lock-step.

Program fragment	Predicated form

`if(lid > N)`
`  x = 0;`
`else`
`  x = 1;`
	` p = (lid > N);`
	` p => x = 0;`
	`!p => x = 1;`

`while(i < x) {`
`  i++;`
`}`
	`p = (i < x);`
	`while(exists t :: t.p) {`
	`  p => i++;`
	`  p => p = (i < x);`
	`}`

Figure 2.1: Predicated forms for conditionals and loops.

Predicated execution

Functional units in a GPU core execute in lock-step, in SIMD fashion. Work items within a sub-group occupy a core's functional units, and thus must also execute in lock-step. Conditional statements and loops through which distinct work items in the same sub-group should take different paths must therefore be simulated, and this is achieved using *predicated execution*.

Consider the conditional statement in the top-left of Figure 2.1, where `lid` denotes the local id of a work item within its group and `x` is a local variable stored in private memory. This conditional can be transformed into the straight-line code shown in the top-right of the figure, which can be executed by a sub-group in lock-step. The meaning of a statement *predicate*=>*command* is that a work item should execute *command* if *predicate* holds for that work item, otherwise the work item should execute a no-op. All work items evaluate the condition `lid > N` into a local boolean variable `p`, then execute both the *then* and *else* branches of the conditional, predicated by `p` and `!p` respectively.

Loops are turned into predicated form by dictating that all work items in a sub-group continue to execute the loop body until the loop condition is false for *all* work items in the sub-group, with work items for whom the condition does not hold becoming disabled. This is illustrated for the loop in the bottom-left of Figure 2.1 (where `i` and `x` are local variables) by the code fragment shown in the bottom-right of the figure. First, the condition `i < x` is evaluated into local variable `p`. Then the sub-group loops while `p` remains true for some work item in the sub-group, indicated by `exists t :: t.p`. The loop body is predicated by `p`, and thus has an effect only for enabled work items.

Barrier synchronisation

When a work item t_1 writes to an address in local or global memory, the result of this write is not guaranteed to become visible to another work item t_2 unless t_1 and t_2 *synchronise*. As noted above, there is *no* mechanism for work items in distinct work groups to synchronise during kernel execution.[1] Work items in the same group can synchronise via *barriers*. Intuitively, a work item belonging to work group *g* waits at a *barrier* statement until every work item in *g* has reached the barrier. Passing the barrier guarantees that all writes to local and global memory by work items in *g* occurring before execution of the barrier have been committed.

[1] Atomic operations on global memory are available in some GPU architectures, but cannot reliably implement inter-group synchronisation due to lack of progress guarantees between groups.

3 General GPU kernel defects

3.1 Data races

A data race is a classic flaw in concurrent software which arises when two work items access the same memory location, at least one of those work items is performing a write operation, and no synchronisation mechanism or atomic operation is used to access into that location. There are various flavours of this problem in GPGPU programming, as described in the following.

Intra-kernel races

Local memory is a resource that is visible to all work items within a work group. However, there is no pre-defined execution order on work items and, as a result, races occur when there are at least two work items that access the same location. Figure 3.1 illustrates the problem. Here, all even-numbered (respectively odd-numbered) work items write their work item identifier to local memory location A (respectively B). Thus the final values written to these locations depend on the order of execution of work items.

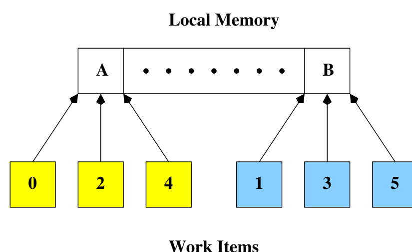


Figure 3.1: Example of intra-kernel data race to local memory.

An example OpenCL kernel exhibiting this sort of this data race appears below:

```
__kernel void racy (__local int* A, __local int* B) {
  const size_t tid = get_local_id(0);
  if (tid % 2 == 0)
    A[0] = tid;
  else
    B[0] = tid;
}
```

The same reasoning applies to global memory, except that the issue is no longer restricted to work items. That is, because work groups also execute asynchronously on a device, different work groups can race on global memory, as the following kernel exemplifies:

```
__kernel void racy (__global int* A, __global int* B) {
  const size_t tid = get_global_id(0);
  if (tid % 2 == 0)
    A[0] = tid;
  else
    B[0] = tid;
}
```

Inter-kernel races

Racing on global memory is also possible across kernel boundaries. There are two general flavours of this problem.

First, as depicted in Figure 3.2, the order in which kernels are queued and executed are not necessarily the same. In the example given, there are no data races when kernels 1 and 2 execute in the order queued. However, specifying out-of-order execution on the command queue, as permitted by OpenCL, allows the run time to order the kernels in an arbitrary fashion; clearly, the results depend on that ordering if there are shared objects between kernels 1 and 2.

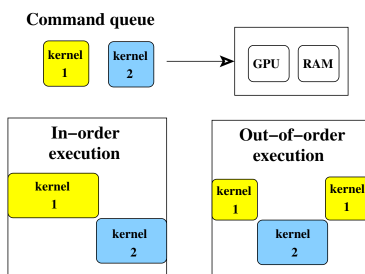


Figure 3.2: Example of inter-kernel data race when the command queue is executed out of order.

However, executing command queues in order does not guarantee an absence of data races, as illustrated in Figure 3.3. Here two in-order command queues are dispatched to a device with two execution resources, namely a GPU and a CPU, but which share memory. Suppose that kernels 3 and 4 are attached to the CPU and kernel 5 is attached to the GPU. Because command queues execute asynchronously to each other and, in this example, kernel 5 can proceed in parallel with kernels 3 and 4, these kernels race on the shared memory.

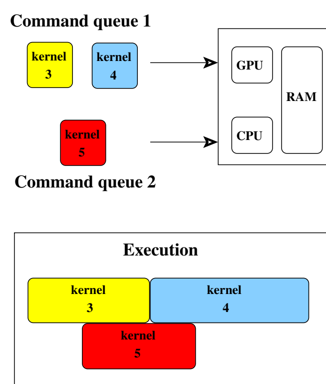


Figure 3.3: Example of inter-kernel data race when multiple command queues are dispatched to a shared-memory device.

Host thread and kernel race

Setting up and launching a kernel occurs on the host side through a series of calls to the OpenCL or CUDA API. However, host execution does not stall until the kernel completes *unless* there are explicit synchronisation points on the host side. Therefore, if the host attempts to copy data

back from a kernel, not ensuring that the kernel has terminated leads to undefined results. This defect appears pictorially in Figure 3.4.

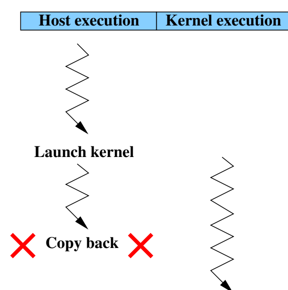


Figure 3.4: Example of host copying data back before kernel finishes.

3.2 Barrier divergence

Although work items in the same work group execute asynchronously, it is possible to synchronise their activities using *barriers*. Intuitively, a work item belonging to a work group waits at a barrier until *every* work item in that work group has reached the barrier. However, if work items can reach different barriers then behaviour is not defined and work items can deadlock.

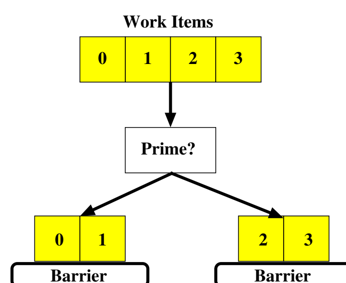


Figure 3.5: Example of barrier divergence.

Figure 3.5 gives an example of the problem. Suppose that work items 0 through 3 execute the same code but then split on whether their work item identifier is prime or not. Furthermore, after the split, there is an immediate barrier. In this case, work items $\{0,1\}$ and $\{2,3\}$ will each wait indefinitely at different barriers and execution cannot continue.

While there is clarity across all programming models for what barrier divergence means in loop-free code, the situation is far from clear for code with loops. Consider the example kernel shown on the left of Figure 3.6, taken from [**?**].

This kernel is intended to be executed by a work group of four work items, and declares an array `A` of two shared buffers, each of size four. Local variable `buf` is an index into `A`, representing the *current* buffer.

The work items execute a nest of loops. On each inner loop iteration a work item reads the value of the current buffer at index `lid`+1 modulo 4 and writes the result into the non-current buffer at index `lid`. A barrier is used to avoid data races on `A`. Notice that local variables `x` and `y` are set to 4 and 1 respectively for work item 0, and to 1 and 4 respectively for all other work items. As a result, we expect work item 0 to perform four outer loop iterations, each

```
shared int A[2][4];

void kernel() {
  int buf, x, y, i, j;
  x = (lid == 0 ? 4 : 1);
  y = (lid == 0 ? 1 : 4);
  buf = i = 0;
  while(i < x) {
    j = 0;
    while(j < y) {
      barrier();
      A[1-buf][lid] =
        A[buf][(lid+1)%4];
      buf = 1 - buf;
      j++;
    }
    i++;
  }
}
```

```
p = (i < x);
while(exists t :: t.p) {
  p => j = 0;
  q = p && (j < y);
  while(exists t :: t.q) {
    q => barrier();
    q => A[1-buf][lid] =
           A[buf][(lid+1)%4];
    q => buf = 1 - buf;
    q => j++;
    q => q = p && (j < y);
  }
  p => i++;
  p => p = (i < x);
}
```

Figure 3.6: Illustration of the subtleties of barriers in nested loops.

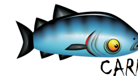
involving one inner loop iteration, while other work items will perform a single outer loop iteration, consisting of four inner loop iterations.

According to the guidance in the CUDA documentation such a kernel appears to be valid: all work items will hit the barrier statement four times. Taking a snapshot of the array A at each barrier and at the end of the kernel, we might expect to see the following:
$A = \{\{0,1,2,3\},\{-,-,-,-\}\} \rightarrow \{\{0,1,2,3\},\{1,2,3,0\}\}$
$\rightarrow \{\{2,3,0,1\},\{1,2,3,0\}\} \rightarrow \{\{2,3,0,1\},\{3,0,1,2\}\}$
$\rightarrow \{\{0,1,2,3\},\{3,0,1,2\}\}$

However, consider the predicated version of the kernel shown in part on the right of Figure 3.6. This is the form in which the kernel executes on an NVIDIA GPU. The four work items comprise a single sub-group. All work items will enter the outer loop and execute the first inner loop iteration. Then work item 0 will become disabled (`q` becomes *false*) for the inner loop. Thus the barrier will be executed with some, but not all, work items in the sub-group enabled. On NVIDIA hardware, a barrier is compiled to a `bar.sync` instruction in the PTX (Parallel Thread Execution) assembly language. According to the PTX documentation [7], *"if any thread in a* [*sub-group*] *executes a* `bar` *instruction, it is as if all the threads in the* [*sub-group*] *have executed the* `bar` *instruction"*. Thus work items 1, 2 and 3 will *not* wait at the barrier until work item 0 returns to the inner loop: they will simply continue to execute past the barrier, performing three more inner loop iterations. This yields the following sequence of state-changes to A:
$A = \{\{0,1,2,3\},\{-,-,-,-\}\} \rightarrow \{\{0,1,2,3\},\{1,2,3,0\}\}$
$\rightarrow \{\{0,3,0,1\},\{1,2,3,0\}\} \rightarrow \{\{0,3,0,1\},\{1,0,1,0\}\}$
$\rightarrow \{\{0,1,0,1\},\{1,0,1,0\}\}$

After the inner loop exits, work item 0 becomes enabled, but all other work items become disabled, for a further three outer loop iterations, during each of which work item 0 executes a single inner loop iteration. The state of A thus remains $\{\{0,1,0,1\},\{1,0,1,0\}\}$.

The OpenCL standard [3] gives a better, though still informal definition, stating: *"If a barrier is inside a loop, all* [*threads*] *must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier"*, which at least can be interpreted as



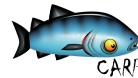
Architecture	Final state of A
NVIDIA Tesla C2050	$\{\{0,1,0,1\},\{1,0,1,0\}\}$
AMD Tahiti	$\{\{0,1,2,3\},\{1,2,3,0\}\}$
ARM Mali-T600	$\{\{0,1,2,3\},\{3,0,1,2\}\}$
Intel Xeon X5650	$\{\{*,*,*,1\},\{3,0,1,2\}\}$

Figure 3.7: The litmus test of Figure 3.6 yields a range of results across varying platforms

rejecting the example of Figure 3.6.

To investigate this issue in practice, we implemented the litmus test of Figure 3.6 in both CUDA and OpenCL and ran the test on GPU architectures from NVIDIA, AMD and ARM, and on an Intel Xeon CPU (for which there is an OpenCL implementation). Our findings are reported in Figure 3.7. Observe that the test result does not agree between any two vendors. The NVIDIA results match our above prediction. The AMD result also appears to stem from predicated execution. ARM's Mali architecture does *not* work using predicated execution [4], so perhaps unsurprisingly gives the "intuitive" result we might expect. For Intel Xeon, we found that different work items reported different values for certain array elements in the final shared state, indicated by asterisks in Figure 3.7, which we attribute to cache effects.

The example of Figure 3.6 is contrived in order to be small enough to explain concisely and examine exhaustively. It does, however, illustrate that barrier divergence is a subtle issue, and that non-obvious misuse of barriers can compromise correctness and lead to implementation-dependent results.



4 OpenCL-specific defects

4.1 Memory defects

To understand the nature of these defects, we first need to review some OpenCL terminology. In particular, OpenCL differentiates between two categories of data types:

1. A **buffer object** can store a collection of a scalar data type (e.g. an integer) or a user-defined structure. Elements in a buffer are stored in a sequential fashion and can be accessed via a kernel pointer executing on the GPU. Typically a buffer object in GPU memory is initialised by copying from a region of host memory. This copy can be avoided, however, through a **host pointer**. In these cases, the GPU may cache that memory.

2. An **image object** is used to represent two- or three-dimensional data such as a frame buffer or image. How elements of an image object are stored is opaque and they cannot be directly accessed using a pointer. Care has to be taken with these data types to avoid undefined behaviour.

The following describes issues with buffer and image objects.

Aliasing host memory

It is possible to initialise two or more buffer objects using the same host pointer, basically creating *aliases* to an area of host memory. Consider the following snippet of OpenCL code:

```
cl_mem input1 =
    clCreateBuffer(..., CL_MEM_USE_HOST_PTR,..., data,...);
cl_mem input2 =
    clCreateBuffer(..., CL_MEM_USE_HOST_PTR,..., data,...);
```

Here `data`, which is a pointer to data residing in host memory, is now effectively aliased by buffer objects `input1` and `input2`. This becomes problematic if the buffer objects are then written to by different kernels that execute asynchronously (e.g. they are in different command queues or the command queue can execute out of order). In effect, this creates a data race on the underlying area of host memory.

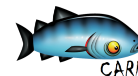
Violating memory access declarations

When creating a buffer or image object, it is possible to override its default read-write attribute by specifying it is read-only (`CL_MEM_READ_ONLY`) or write-only (`CL_MEM_WRITE_ONLY`). However, any subsequent read (respectively write) to a write-only (respectively read-only) object is undefined.

Concurrent access to sub-buffers

In OpenCL it is possible to divide a buffer object or image into a number of chunks called **sub-buffers**. This allows multiple kernels to operate on sub-buffers at the same time.

However, reading, writing, or copying between a buffer object and its sub-buffer(s) in a concurrent fashion is undefined. In a similar vein, performing any of those operations between



overlapping sub-buffers is undefined. The only valid concurrent activity allowed on a buffer and its sub-buffers, or on overlapping sub-buffers, is a sequence of reads.

Mapping buffer objects

Buffer objects residing on a device can be mapped into the host address space through `clEnqueueMapBuffer`, which returns a pointer to the mapped region. Any memory access outside of the range of the mapped region, however, is undefined.

Manipulating images on the host

Images residing in device memory can be read into host memory using `clEnqueueReadImage`, or conversely written to from host memory using `clEnqueueWriteImage`.

Correct use of these functions has several stipulations. First of all, any pending operations on the image object must complete before the read or write initiates. Secondly, the image object cannot be mapped into host memory. Finally, any subsequent operation on the image object through the device cannot commence until the read or write has finished.

Incomplete transfer

A buffer object can be read into host memory or written by the host with `clEnqueueReadBuffer` or `clEnqueueWriteBuffer`, respectively. When these operations are non-blocking, the host code must wait for an event signalling completion. Consider the following snippet of code:

```
input  = clCreateBuffer(context, CL_MEM_COPY_HOST_PTR,
                        sizeof(float) * count, data, NULL);

cl_event eventObject;
err = clEnqueueWriteBuffer(commands, input, CL_FALSE,
                           0, sizeof(float) * count,
                           data, 0, NULL, &eventObject);

// Wait for the event.
// Removing this line of code is erroneous.
clWaitForEvents(1,&eventObject);
data[0] = -100;
```

Here a memory copy is initiated on `input` through `clEnqueueWriteBuffer`. It is non-blocking as the third parameter is `CL_FALSE`, but there is an event attached to the operation through the last parameter `eventObject`. The call to `clWaitForEvents` forces the host to suspend until that operation is complete. Removing that call, on the other hand, would mean that the assignment to `data` on the subsequent line occurs asynchronously.

4.2 Miscellaneous defects

Callback on OpenCL build function

If a callback function is defined then the OpenCL program can be built asynchronously. In this case, the callback function will be called when the program executable has been built. It is the responsibility of the application to ensure that this function is thread safe otherwise unexpected behaviour arises.

Work item divergence

Specific OpenCL functions must be performed by all work items in a work group and, furthermore, with the same argument values, otherwise the results are undefined. Examples of functions under this restriction are: `async_work_group_copy`, `async_work_group_strided_copy` and `wait_group_events`.

Race setting kernel argument

Setting up the parameters of an OpenCL kernel is achieved through the `clSetKernelArg` function call. However, when multiple host threads are executing and attempting to set the arguments of the kernel to different values, the behaviour of the kernel is undefined. The following example demonstrates this defect using OpenMP threads:

```
int tid, nthreads, err;
int* container;

#pragma omp parallel private(nthreads, tid)
{
  tid = omp_get_thread_num();
  err = clSetKernelArg(theKernel,
                       0,
                       sizeof(cl_mem),
                       container[tid%2]);
}
```

Here a parallel region of host code is declared through an OpenMP pragma with `nthreads` threads. The host thread gets its thread identifier from the OpenMP run-time system using `omp_get_thread_num`. Then, if this is value is even, it tries to set the first kernel argument to `container[0]`, otherwise it tries to set it to `container[1]`. The value of the kernel argument is therefore dependent on the thread identifier, which is erroneous.

Incorrect initialisation of work items or work groups

It is possible that the number of work items and/or work-groups defined in the host code is not initialised correctly. As a result some values may not be computed or we may have index out-of-bound errors and the kernel crashes. The following example demonstrates this defect:

```
__kernel void square(__global float* input,
                     __global float* output)
{
    const size_t tid = get_global_id(0);
    output[tid]= input[tid] * input[tid];
}
```

If the number of work items exceeds the size of the arrays `input` and `output`, then there is an index out-of-bound error. On the other hand, if the number of work items is less than these array dimensions, some values will not be computed.

Launching kernel with variable in local memory

Kernels can be launched from within other kernels. However, when the callee kernel expects a formal parameter in local memory space, and the actual argument is declared in the caller

kernel, the behaviour is implementation defined. Consider the following example:

```
__kernel void kernelOne(__local float* input);

__kernel void kernelTwo(__global float* input,
                        __global float* output)
{
    __local float temp [1024];
    kernelOne(temp);
}
```

The problem is that `temp` is declared locally in `kernelTwo` within device local memory and then passed to `kernelOne`.

Host and device endianness

The endianness of a device and the host are key concerns in the portability of OpenCL programs since OpenCL strives to support heterogeneous platforms whose endianness differ.

Variables residing in global or constant memory on the device might have a different endianness to the host. To allow the OpenCL compiler to do endian conversion on load or store operations from or to these variables, the developer can mark them with an attribute indicating the storage choice: `__attribute__ ((endian(host))` specifies the variable uses the endianness of the host, whereas `__attribute__ ((endian(device))` indicates the variable uses the endianness of the device on which the program will be executed. The default endianness is the device type.

However, when an OpenCL program relies on the endianness of a particular device, it clearly becomes incompatible with devices whose endianness differ. As stated in the OpenCL specification:

> ... developers need to make sure that their kernels are tested on both big-endian and little-endian devices to ensure source compatibility with OpenCL devices now and in the future.

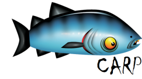
Endianness can also unintentionally become part of the kernel implementation when casting between types. The following code snippet, taken directly from the OpenCL specification, demonstrates the problem:

```
// An array of floats
float x[4] = {0.0f, 1.0f, 2.0f, 3.0f};
// Create a floating-point vector of width 4
float4 v = vload4(0, x);

// Cast the floating-point vector into an unsigned integer vector.
// Portable
uint4 y = (uint4) v;

// Cast the floating-point vector into an unsigned short vector.
// Not portable
ushort8 z = (ushort8) v;
```

The first type cast is portable because the the source vector `v` and the destination vector `y` consist of four elements. This means that the byte ordering between elements in the vector and the ordering between elements in the vector remain the same.

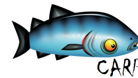


However, the vector `z` contains eight elements and the byte ordering of its elements depends on the endianness of the device. On a little-endian machine `z` is stored as:

0x4040, 0x0000, 0x4000, 0x0000, 0x3f80, 0x0000, 0x0000, 0x0000

whereas on a big-endian machine it is:

0x0000, 0x0000, 0x3f80, 0x0000, 0x4000, 0x0000, 0x4040, 0x0000



5 CUDA-specific defects

5.1 Atomics

One way to avoid data races on a shared memory location is to use *atomic* functions, which implement a read-write-modify operation without interference from other work items. CUDA currently supports atomic functions on 32-bit or 64-bit locations residing in local or global memory, e.g. `atomicDec` atomically decrements an integer value by one. However, it is the responsibility of the programmer to ensure that any location updated by an atomic function is not concurrently updated by a *non-atomic* statement.

To exemplify this defect, first consider a CUDA kernel with only atomic updates:

```
__global__ void atomics (int* n)
{
  atomicAdd(n, 100);
}
```

Here, the formal parameter `n` resides in global memory and is incremented atomically by 100. Assuming `n` is initially 0, and that there are 32 work items, the final value of `n` is 3200.

The problem arises when the kernel is modified as follows:

```
__global__ void atomics (int* n)
{
  // Non-atomic reset of n
  *n = 0;
  atomicAdd(n, 100);
}
```

Now there are no guarantees on when `n` will be reset to 0, thus there is a data race on `n` between the two instructions.

5.2 Warp shuffling

As noted in Chapter 2, the CUDA execution model concentrates around a sub-group of 32 work items, a so-called **warp**. The concept of a warp is supposed to be hidden from the programmer, i.e. it is a run-time detail, although it is almost inescapable if a programmer wants to increase performance. CUDA therefore provides several programming tricks and offers several intrinsics to manipulate warps with the sole aim of increasing performance.

One such intrinsic is `__shfl` which facilitates the exchange of data between work items in a warp without passing through the memory hierarchy, which is much more costly. But care must be taken with these warp shuffling intrinsics, for two reasons.

First, when two work items in a warp wish to exchange a value, both must be *active*. In this sense, active means that the work item executes the instructions: at a branch, it is possible for a work item to become disabled because it does not satisfy a data-dependent decision, and it will only become enabled again at a particular merge point. Thus, if one of the work items is in a disabled state, the value retrieved during the warp shuffle is undefined.

Warp shuffling intrinsics also take an optional width parameter that divides the warp into *sub-warps*. This allows data to be exchanged *within* sub-warps rather than across the entire

warp. However, the width of each sub-warp must be a power of two, to enable equal division; again, failure to comply with this property produces undefined behaviour.

6 Examples of defects in open source samples

In analysing kernels from the AMD Accelerated Parallel Processing SDK [1], the CUDA SDK [5], and in the analysis of the Rodinia [2] and SHOC [10] benchmark suites we have encountered a number of defects, which we now describe.

AMD SDK: LU decomposition

We found an example of an intra-group data race in an LU decomposition kernel shipped with the AMD APP SDK [1]. Part of the code for this kernel is shown in Figure 6.1. Threads write into global memory array `inplaceMatrix` at lines 9-12. The threads then synchronise at a barrier at line 15. This barrier takes the `CLK_LOCAL_MEM_FENCE` flag, specifying that threads synchronise with respect to *local* memory, but *not necessarily* with respect to global memory. As a result this barrier does not guarantee that the writes to `inplaceMatrix` will not race with subsequent reads from `inplaceMatrix` at lines 27-30.

The problem can be fixed by strengthening the barrier to also synchronise on global memory, replacing the barrier statement with:

```
barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
```

CUDA *N*-body simulation

We discovered a write-write data race in the *N*-body example that shipped with the CUDA SDK v2.3 [5]. This example uses multiple CUDA kernels to numerically approximate a system of N interacting bodies [9]. This is an ideal problem for parallelisation since interactions between each pair of bodies can be calculated independently. The CUDA implementation of this example decomposes the N^2 pair-interactions into smaller $k \times k$ tiles, each of which is assigned to a one-dimensional group of k threads. Within each group, every thread is assigned to a distinct body (a row of the tile) and sequentially considers the interactions associated with this body to compute an updated state for the body.

The kernel implements an optimisation for small values of N where threads are arranged in two-dimensional groups, and multiple threads within a group are assigned to the same body. Consequently, the interactions calculated by threads assigned to the same body must be summed. A barrier ensures that each thread has completed its sub-calculation, and then a conditional is used to ensure that a single "master" thread performs the summation. However, a data race could occur because a similar condition was not in place to ensure that only this master thread would perform a final update to the position and velocity of the body. As a result, it was possible for the master thread's final update, using the full summation, to be overwritten by partial results computed by other threads.

We reported this data race to Lars Nyland at NVIDIA who confirmed that "It was a real bug, and it caused real issues in the results. It took significant debugging time to find the problem." [8]. NVIDIA had subsequently fixed this bug in v3.0 of the CUDA SDK.

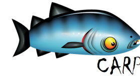
```
__kernel void kernelLUDecompose(__global double4* LMatrix,
                              __global double4* inplaceMatrix,
                               int d,
                               __local double* ratio)
{
  ...
  if(get_local_id(0) == 0)
  {
    (D == 0) ? (ratio[lidy] = inplaceMatrix[ y * xdimension + d / VECTOR_SIZE].s0 / ...):1;
    (D == 1) ? (ratio[lidy] = inplaceMatrix[ y * xdimension + d / VECTOR_SIZE].s1 / ...):1;
    (D == 2) ? (ratio[lidy] = inplaceMatrix[ y * xdimension + d / VECTOR_SIZE].s2 / ...):1;
    (D == 3) ? (ratio[lidy] = inplaceMatrix[ y * xdimension + d / VECTOR_SIZE].s3 / ...):1;
  }

  barrier(CLK_LOCAL_MEM_FENCE);

  if(y >= d + 1 && ((x + 1) * VECTOR_SIZE) > d)
  {
    ...

    if(x == d / VECTOR_SIZE)
    {
      ...
    }
    else
    {
      inplaceMatrix[y * xdimension + x].s0 = result.s0;
      inplaceMatrix[y * xdimension + x].s1 = result.s1;
      inplaceMatrix[y * xdimension + x].s2 = result.s2;
      inplaceMatrix[y * xdimension + x].s3 = result.s3;
    }
  }
}
```

Figure 6.1: Intra-group data race on global memory in LU decomposition kernel. The writes to `inplaceMatrix` at lines 9-12 are separated from the reads at lines 27-30 by a *local* memory barrier, but this is not sufficient to guarantee data race freedom because `inplaceMatrix` is an array in *global* memory.



Defects in the Rodinia benchmark suite

Our analysis of the Rodinia benchmarks [2] (see D2.2A) revealed three defects:

- The *HotSpot* benchmark uses a buffer that is defined with the `CL_MEM_USE_HOST_PTR` flag, indicating that this buffer should reside in host memory. However, the benchmark code used OpenCL API calls to copy the contents of this buffer to/from device memory, violating the intention specified by the flag and leading to undefined behaviour.
- The *Pathfinder* benchmark exhibited a potential data race due to a missing barrier. The nature of this data race is similar to those of the data races in the AMD and CUDA SDKs, discussed above.
- The *BFS* benchmark contained a functional defect where a 2D array index was incorrectly calculated. This led to the kernel computing incorrect results when invoked with a non-square array of work items.

We have reported these bugs to the Rodinia developers, who have confirmed them.

Defects in the SHOC benchmark suite

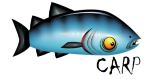
Our analysis of the SHOC benchmarks [10] (see D2.2A) revealed two defects:

- The *Sort* benchmark performs a radix sort using a radix value of 4. This means that a 32-bit integer is split into 8 4-bit digits. The kernel is invoked multiple times, with each invocation sorting the array based on the values of a specific digit of the elements (starting with the least significant one and moving one digit to the left at every iteration). For every iteration, the input and output buffers are switched, except in the final iteration where the buffers were not switched. This could lead to the computation of incorrect results. Interestingly this defect did not manifest in the benchmark's self-validation code because the test inputs lie in the range 0..15, and so fit into one 4-bit digit. As a result sorting is complete by the end of the first kernel invocation.
- In one version of the *BFS* benchmark, synchronisation across distinct work-groups is attempted based on the assumption that if the number of work groups does not exceed the number of compute units the work groups are guaranteed to execute in parallel. This is not valid OpenCL code, as there is no mechanism for synchronisation between work-groups in OpenCL [3].

The SHOC developers have confirmed the first bug and fixed it in their repository.

Regarding the second bug, the SHOC developers commented that this invalid OpenCL code behaved correctly when executed on NVIDIA GPUs and offered a significant performance improvement over alternative implementations that lie within the defined semantics of OpenCL. This illustrates the fact that the portability aims of OpenCL come at a price: developers may be forced to write code that works in general but performs sub-optimally on particular architectures, or may instead opt to write code which, strictly, has undefined semantics, but which (for implementation-defined reasons) behaves sensibly and efficiently on a particular architecture.

We encountered several inconsistencies in the SHOC benchmark with respect to the synchronisation of OpenCL API calls. Within individual benchmarks we found that event lists



were used for some but not all commands that were enqueued to a command queue. Because the benchmarks use in-order command queues these inconsistencies have no semantic effects. However, if the benchmark implementations switched in the future to use out-of-order command queues these inconsistencies could lead to buggy behaviour.

Bibliography

[1] AMD. AMD Accelerated Parallel Processing (APP) SDK. http://developer.amd.com/sdks/amdappsdk/pages/default.aspx.

[2] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a verifier for GPU kernels. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*, pages 113–132. ACM, 2012.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Khronos OpenCL Working Group. *The OpenCL specification, version 1.1*, October 2012.

[5] A. Lokhmotov. Mobile and embedded computing on Mali GPUs. In *Second UK GPU Computing Conference*, December 2010.

[6] NVIDIA. CUDA toolkit release archive. developer.nvidia.com/cuda-toolkit-archive.

[7] NVIDIA. *CUDA C Programming Guide Version 4.2*, October 2012.

[8] NVIDIA. *Parallel Thread Execution (PTX) ISA v2.3*, October 2012.

[9] L. Nyland, April 2012. Personal communication.

[10] L. Nyland, M. Harris, and J. Prins. Fast *n*-body simulation with cuda. In *GPU Gems 3*. Addison Wesley, 2007. Chapter 31.

[11] K. Spafford. The Scalable HeterOgeneous Computing (SHOC) benchmark suite. https://github.com/spaffy/shoc/wiki.

[12] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-M. W. Hwu, B. P. Sutton, and Z.-P. Liang. Accelerating advanced MRI reconstructions on GPUs. *J. Parallel Distrib. Comput.*, 68(10):1307–1318, 2008.