Formal Analysis Techniques for Reliable GPU Programming: Current Solutions and Call to Action

Alastair F. Donaldson, Imperial College London
Ganesh Gopalakrishnan, University of Utah
Nathan Chong, University College London
Jeroen Ketema, Imperial College London
Guodong Li, University of Utah
Peng Li, University of Utah
Anton Lokhmotov, dividiti
Shaz Qadeer, Microsoft Research

Abstract

GPU-accelerated computing is being adopted increasingly in a number of areas, ranging from high-end scientific computing to mobile and embedded computing. While GPU programs routinely provide high computational throughput in a number of areas, they also prove to be notoriously difficult to write and optimize correctly, largely due to the subtleties of GPU concurrency. This chapter discusses several issues that make GPU programming hard, and examines recent progress on rigorous methods for formal analysis of GPU software. Our key observation is that given the fast-paced advances in GPU programming, the use of rigorous specification and verification methods must be an integral part of the culture of programming and training, and not an afterthought.

1. GPUs in Support of Parallel Computing

When the stakes are high, money is no object in a nation's quest for computing power. For instance, the SAGE air-defense network [41] was built using a computing system with 60,000 vacuum tubes, weighing 250 tons, consuming 3MW of power, and at a cost of over $8 billion, in 1964. Now (five decades later), a door-lock computer often exceeds SAGE in computing power, and more than ever, we are critically dependent on computers becoming faster as well as more energy efficient. This is not only true for defense, but for every walk of science and engineering, and societal sector.

Graphics processing units (GPUs) are a natural outgrowth of the computational demands placed by today's applications on such stringently power-capped and wire-delay bounded hardware [12]. GPUs achieve higher computational efficiency than CPUs by employing simpler cores, and hide memory latency by switching away from stalled threads. Overall, their throughput-oriented execution model is a good match for many data-parallel applications.

The rate of evolution of GPUs is spectacular: from the 3M transistor Nvidia NV3 in 1997, their trajectory is marked by the 7B transistor Nvidia GK110 (Kepler) in 2012 [42]. Nvidia's CUDA programming model, introduced in 2007, provided a dramatic step up from the baroque notation associated with writing pixel and vertex shaders, and the recent CUDA 7.5 [34] offers versatile concurrency primitives. OpenCL, an industry standard programming model [25] supported by all major device vendors, including Nvidia, AMD, ARM and Imagination Technologies, provides a straightforward, portable mapping of computational intent to tens of thousands of cores.

GPUs now go far beyond graphics applications, forming an essential component in our quest for parallelism, in diverse areas such as gaming, web search, genome sequencing and high-performance supercomputing.

Bugs in Parallel and GPU Code. Correctness of programs has been a fundamental challenge of computer science even from the time of Turing [33]. Parallel programs written using CPU threads or message passing (e.g., MPI) are more error prone than sequential programs, as the programmer has to encode the logic of synchronization and communication among the threads, and arrange for resources (primarily the memory) to be shared. This situation leads to bugs that are very difficult to locate and rectify.

In contrast to general concurrent programs, GPU programs are "embarrassingly parallel", with threads being subject to structured rules for control and synchronization. Nevertheless, GPU programs pose certain unique debugging challenges which have not received sufficient attention, as will be clear in the sequel when we discuss GPU bugs in detail. Left unchecked, GPU bugs can become show-stoppers, rendering simulation results produced by multi-million dollar scientific projects utterly useless. Sudden inexplicable crashes, non-reproducible executions, as well as non-reproducible scientific results are all the dirty laundry that often goes unnoticed amidst the din of Exascale computing.

Yet, these bugs do occur, and seriously worry experts who often struggle to bring up newly commissioned machines or are pulled away from doing useful science. The purpose of this chapter is to describe some of these challenges, provide an understanding of the solutions being developed, and characterize what remains to be accomplished.

After introducing GPUs at a high level (Section 2), we survey some key GPU correctness issues in a manner that the program analysis and verification community can benefit from (Section 3). The key question addressed is: *What are the correctness challenges, and how can we benefit from joint efforts to address scalability and reliability?* We then address the question: *How can we build rigorous correctness checking tools that handle today's as well as upcoming forms of heterogeneous concurrency?* To this end we discuss existing tools that help establish correctness, providing a high level description of how they operate and summarizing their limitations (Section 4). We conclude with our perspective on the imperatives for further work in this field through a call to action for (a) research-driven advances in correctness checking methods for GPU-accelerated software, motivated by open problems, and (b) dissemination and technology transfer activities to increase uptake of analysis tools by industry (Section 5).

```
1     #define ACCUM_N 1024
2     __global__ void dotProduct(float *d_A, float *d_B,
3                                float *d_c, int n) {
4       __shared__ float accumResult[ACCUM_N];
5       int tid = threadIdx.x;
6       int bdim = blockDim.x;
7
8       // (a) Compute partial sums
9       for (int i = tid; i < ACCUM_N; i += bdim) {
10        float sum = 0;
```

```
11          for (int j = i; j < n; j += ACCUM_N) {
12            sum += d_A[j] * d_B[j];
13          }
14          accumResult[i] = sum;
15        }
16
17        // (b) Reduce
18        for (int stride = ACCUM_N / 2; stride > 0;
19             stride >>= 1) {
20          __syncthreads();
21          for (int i = tid; i < stride; i += bdim) {
22            accumResult[i] += accumResult[stride + i];
23          }
24        }
25
26        if (tid == 0) *d_c = accumResult[0];
27      }
```

Figure 1: Dot product kernel adapted from the CUDA 5.0 SDK

2. A Quick Introduction to GPUs

GPUs are commonly used as parallel co-processors under the control of a host CPU in a heterogeneous system. In this setting, a task with abundant parallelism can be offloaded to the GPU as a kernel: a template specifying the behavior of an arbitrary thread. Figure 1 presents a CUDA kernel for performing a parallel dot product of two vectors, adapted from the CUDA 5.0 SDK [35], which we use as a running example.

We provide a brief overview of the GPU programming model. As we introduce each concept and component, we give the CUDA term for the concept or component, followed by the OpenCL term in parentheses if it is different, and use the CUDA term thereafter.

Organization of threads. Kernels are executed on a GPU by many lightweight *threads* (*work-items*) organised hierarchically as a *grid* (*NDRange*) of *thread-blocks* (*work-groups*), as illustrated by Figure 2. For example, a grid of 4 thread-blocks each of 256 threads results in 1024 threads each running a copy of a kernel. The `__global__` annotation in Figure 1 indicates that `dotProduct` is a kernel function.

Within the kernel, a thread can query its position within the grid hierarchy (as well as the grid and thread-block dimensions) using built-ins such as `threadIdx` (`get_local_id`) and `blockDim` (`get_local_size`). Grids and blocks can be multi-dimensional, and e.g. `blockDim.x` (`get_local_size(0)`) and `threadIdx.x` (`get_local_id(0)`) provide, respectively, the size of a thread-block and the id of a thread within its block in the first dimension. This allows threads to operate on distinct data and to follow distinct execution paths through the kernel.

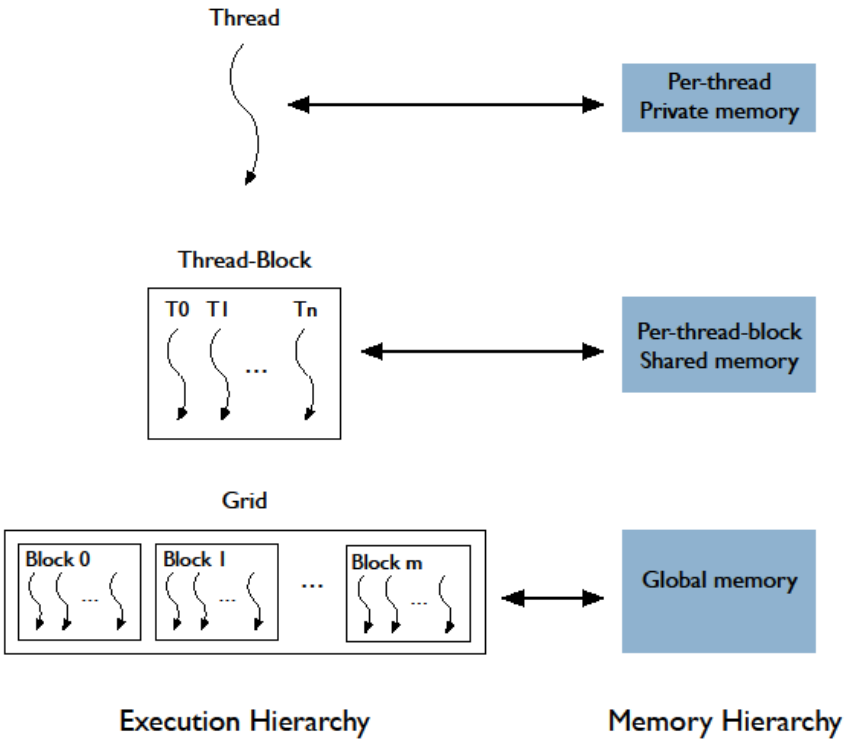

Figure 2: GPU thread and memory hierarchy, adapted from [34]

Memory spaces. Threads on a GPU can access data from multiple memory spaces arranged in a hierarchy that reflects thread organisation, as shown in Figure 2. In descending order of size, scope-visibility and latency:

• A large global memory visible to all threads in a grid. In CUDA, pointer parameters to a kernel refer to global memory arrays, thus `d_a`, `d_b` and `d_c` in Figure 1 are global arrays. The kernel computes the dot product of the arrays `d_A` and `d_B` and stores the result in `d_c` (a one-element array).

• A per-thread-block *shared* (*local*) memory visible to all threads within the same thread-block. The `__shared__` annotation in Figure 1 specifies that the `accumResult` array resides in shared memory. Each thread-block has a distinct copy of this array, which is used for accumulating partial dot product values.

• A small per-thread *private* memory. Loop variables `i`, `j` and `stride` in Figure 1 reside in private memory. Each thread has a separate copy of these variables.

It is the programmer's responsibility to orchestrate data movement between the global and shared memory spaces. Memory *coalescing* is an important property for performance reasons. When adjacent threads access contiguous memory locations, the hardware can coalesce these accesses into fewer memory transactions and therefore increase bandwidth.

While threads share certain memory spaces, memory writes do not become instantaneously visible to all threads (as this would severely impede performance). In computer architecture, the concept of memory consistency models is used to clearly explain when threads may observe the writes of other threads. GPU programming models specify a *weak memory consistency model* [34, 25]. That is, the updates and order of accesses of a given thread are not guaranteed to be observable by other threads.

Barrier synchronization. Barriers facilitate safe communication between threads of the same thread-block. A barrier operation causes a thread to stall until all threads in the thread-block have reached the same barrier. In fact, the barrier must be reached under the same control-flow by all threads in order to avoid the problem of *barrier divergence*, which results in undefined behavior. The dot product kernel uses barrier synchronization, indicated by `__syncthreads()`, to ensure proper ordering of updates to the shared array `accumResult`.

Barriers cannot be used for synchronization between threads in distinct thread-blocks. Instead, this requires the programmer to split the workload into multiple kernels that are invoked in sequence. Threads in different thread-blocks can use atomic operations to communicate via global memory.

Warps and lock step execution. On Nvidia GPUs, the hardware executes a thread-block by dynamically partitioning it into sets of *warps*; AMD GPUs have a similar notion of a *wave-front*. Currently, Nvidia specifies that a warp is a set of 32 adjacent threads of the thread-block. Threads within the same warp execute in *lock step* and so are implicitly synchronized; we comment on the opportunities and dangers presented by this phenomenon in Section 3.

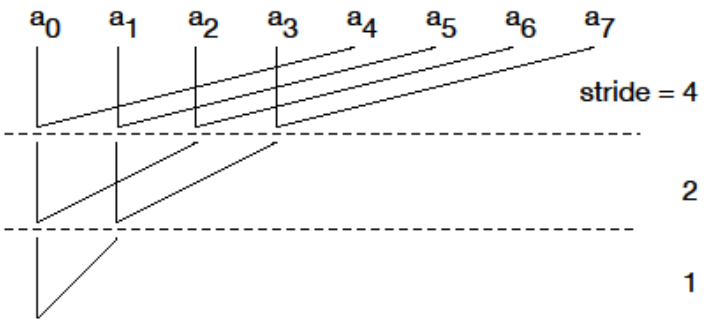

Figure 3: A tree reduction as used in the kernel of Figure 1

Dot product example. We are now suitably equipped to discuss the kernel in Figure 1. Consider the kernel when it is invoked with a single thread-block of 128 threads (i.e. `blockDim.x` = 128) where the length *n* of the input arrays equals 4096.

The kernel has two parallel phases. In the first phase, partial products are accumulated into the shared array `accumResult`. The outer loop assigns a distinct thread to each element of the `accumResult` array. Because there are more elements (`ACCUM_N`) than threads the outer loop assigns `ACCUM_N/blockDim.x` = 8 elements per thread. For example, thread 0 is assigned elements 0, 128, 256, . . . , 896. The result for each element *i* of the output is accumulated in the thread-private variable `sum` and accumulated by the inner loop. The inner loop performs *n*/`ACCUM_N` = 4 partial products at `ACCUM_N` intervals. For example, the inner loop for thread 0 when *i* = 0 will compute $\sum a_j b_j$ for *j* in {0, 1024, 2048, 3072}. The stride of accesses ensures coalesced memory accesses to global memory.

In the second phase, the partial products are reduced into the final expected result. Instead of summing `ACCUM_N` elements serially, the kernel uses a parallel tree reduction. The reduction is performed using a logical tree. A simplified reduction tree for 8 elements is given in Figure 3. Each iteration of the loop – using descending power-of-two `stride` values – corresponds to a different

level of the tree. The barrier ensures that the updates of a given level are ordered before any access at the next level and is thus a form of inter-thread communication.

3. Correctness Issues in GPU Programming

We now outline four key classes of correctness issues that can make the transition from CPUs to GPUs hard.

Data races. Insufficient or misplaced barriers can lead to *data races*. A data race occurs if two threads access the same memory location (in global or shared memory) where at least one access is non-atomic, at least one access modifies the location, and there is no intervening barrier synchronization that involves both threads. (In OpenCL 2.0, synchronization between certain atomic operations can also be used to avoid races; we do not delve into the details of this here.) For example, consider the in-place tree reduction of the dot product kernel in Figure 1. The barrier at line 20 ensures that all accesses at a given iteration (when `stride` = *k*, say) are ordered before any accesses at the next iteration (when `stride` = *k*/2). If the barrier is omitted then thread 0 with `stride` 2 and thread 2 with `stride` 4 will race on `accumResult[2]`.

In most concurrent programs, races are tell-tale indicators that something is seriously wrong with the code, including the potential for non-deterministic results or semantically incoherent updates due to insufficient atomicity. GPU programming models (e.g. OpenCL) require the programmer to write data-race-free code. Programs containing data races have undefined semantics. Consequently, many compiler optimizations assume race-freedom; in the presence of races they may produce unexpected results.

One can guard against data races through conservative barrier placement, but excessive use of barriers can be problematic for two reasons. First, barriers are expensive to execute, so that superfluous synchronization introduces an unnecessary performance overhead. Second, placing barriers in conditional code can be dangerous due to the problem of barrier divergence discussed in Section 2.

On Nvidia GPUs, many CUDA programmers choose to elide barriers in the case where synchronization is required only between threads in the same warp. For instance, the last 6 iterations of the tree reduction loop in phase two of the dot product kernel (Figure 1) might be replaced with the following sequence of statements, to avoid explicit synchronization when the stride is less than or equal to 32:

```
if(tid < 32) accumResult[tid] += accumResult[tid + 32];
if(tid < 16) accumResult[tid] += accumResult[tid + 16];
if(tid <  8) accumResult[tid] += accumResult[tid +  8];
if(tid <  4) accumResult[tid] += accumResult[tid +  4];
if(tid <  2) accumResult[tid] += accumResult[tid +  2];
if(tid <  1) accumResult[tid] += accumResult[tid +  1];
```

This practice relies on the compiler preserving implicit intra-warp synchronization during optimization. It is not clear whether this is the case. Advice related to intra-warp synchronization was removed in version 5.0 of the CUDA Programming Guide, and there is disagreement between

practitioners about the guarantees provided by current platforms in this regard (see, e.g., an Nvidia forum discussion on the topic [37]).

Nevertheless, some samples that ship with the CUDA 5.0 SDK rely on implicit intra-warp synchronization. More surprisingly, OpenCL kernels in open source benchmark suites such as Parboil [47] and SHOC [22] elide barriers in the above manner. This is clearly erroneous since the notion of lock step warps is not part of the OpenCL specification.

Recent work has shown that Nvidia and AMD GPUs exhibit weak memory behaviors (as discussed in Section 2), whereby non-sequentially consistent executions can be observed, and that this is a source of subtle software defects [1]. Bugs arising due to weak memory effects will become increasingly relevant as GPU applications move towards exploiting fine-grained concurrency in place of barrier synchronization.

```
1    // Pre: 'count' initialized to #thread-blocks
2    __threadfence();
3    __syncthreads();
4    if(threadIdx.x == 0) {
5      atomicDec(count);
6      while(atomicAdd(count, 0) > 0) { }
7    }
8    __syncthreads();
9    __threadfence();
```

Figure 4: An attempt to implement an inter-block barrier in CUDA is foiled by lack of progress guarantees between thread-blocks

Lack of forward progress guarantees. The lack of fair scheduling of threads on GPU architectures is another source of defects.

Figure 4 shows an attempt to implement an inter-block barrier in CUDA. The idea behind this well-known strategy is as follows. Each thread synchronizes within its block (line 3) after which the *leader* of each thread-block (the thread with `threadIdx.x` = 0) atomically decrements a counter (line 5). Assuming the counter was initialized to the total number of thread-blocks (comment on line 1), it might appear that spinning until the counter reaches zero (line 6) would suffice to ensure that every leader has passed the decrement (line 5). (Note that the counter value is retrieved by atomically adding zero to the counter; `atomicAdd` returns the old value of the location operated upon. Using an atomic operation, rather than a plain load operation, avoids data races between accesses to the counter.) If spinning until the counter reaches zero would indeed ensure that every leader has passed the decrement then a global synchronization would have been achieved, and the leader could re-synchronize with the rest of its block (line 8), allowing execution to recommence. The purpose of the fences (lines 2 and 9) is to ensure that memory access operations before the global synchronization point take effect before memory access operations after the global synchronization point. For a community discussion of the strategy and its problems, see e.g. [45].

The problem with this barrier implementation is that it assumes forward progress between thread-blocks. However, if sufficiently many CUDA blocks are requested then they will be scheduled in *waves*: a number of blocks will be scheduled and must run to completion before compute units are

freed for further blocks to be scheduled. This scenario leads to deadlock: leaders of the blocks scheduled in the first wave get stuck spinning at line 6, waiting for the counter to be decremented by leaders of additional blocks; these blocks in turn cannot be scheduled until the initial blocks complete execution. An analogous attempt to implement global synchronization in OpenCL does not work either, for the same reason.

This example illustrates unfair scheduling across thread-blocks. Similar lack of progress issues arise between threads *within* a CUDA thread-block because, as discussed in Section 2, a thread-block is subdivided into warps such that threads in the same warp execute in lock step, executing identical instruction sequences. This leads to deadlock if a programmer attempts to force a thread to busy-wait for another thread in the same warp; progress would require the threads to genuinely execute distinct instructions at runtime. This defeats the naïve implementation of an intra-block critical section based on busy waiting. For an example of this issue, with a community discussion, see [46].

Floating point accuracy. Achieving equivalence to a reference implementation can be especially challenging for kernels that compute on floating point data, which are commonplace in high performance computing.

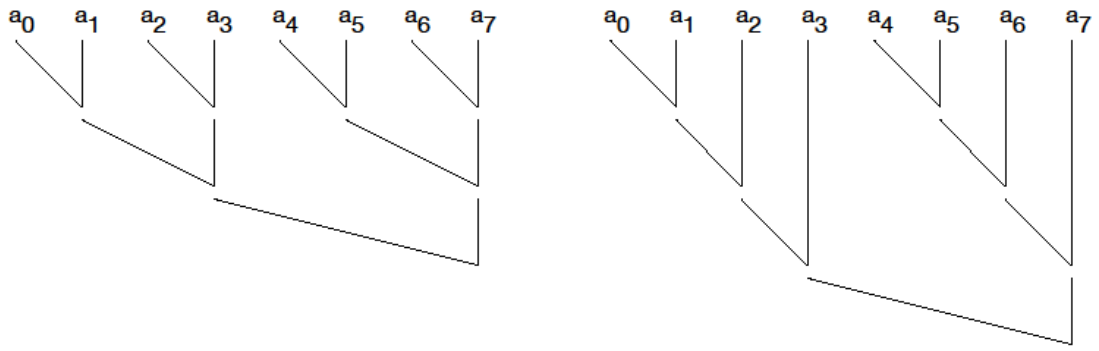

Figure 5: Both reduction trees compute a sum over a_i but can yield different results because floating point addition is not associative

It can be convenient to assume that floating point operators have the algebraic properties of real numbers when designing parallel algorithms. For instance, the sum reduction operation, in which all elements of an array are added, can be parallelized through a computation tree illustrated by the diagrams of Figure 5. A tree-based approach allows reduction in a logarithmic number of steps if the number of parallel processing elements exceeds the size of the array. The figure shows two possible computation trees for an eight-element array. If addition were *associative*, that is, if the law $(x + y) + z = x + (y + z)$ held for all floating point values x, y and z, with + denoting floating point addition, the computation trees would yield identical results, and the result would be the same as that of the left-associative sum obtained from a straightforward sequential implementation. However, it is well known that floating point addition is not associative so that results are expected to differ depending on how the parallel algorithm is structured.

While these issues are shared by CPUs, what perhaps makes these concerns particularly important for GPUs is the fact that GPUs are throughput oriented, and have much smaller caches. Thus, the added penalty of using double-precision arithmetic everywhere (instead of single-precision almost everywhere) tends to be higher with GPUs, especially since the number of double-precision units on

a GPU is typically small. There are also language-specific issues related to floating point accuracy in the context of GPUs. For example, whether *denormal* numbers are accurately represented in OpenCL is implementation-defined, and OpenCL provides a *half precision* data type, specifying only *minimal* (not exact) accuracy requirements for associated operators. Implementation differences make it hard to write high-performance floating point code that behaves with an acceptable degree of precision on multiple GPU platforms.

4. The Need for Effective Tools

The issues discussed in Section 3 show that practitioners working at the level of OpenCL and CUDA cannot avoid being exposed to the "sharp edges" of concurrency. In some application areas it may be possible for non-expert programmers to bypass these issues through the use of domain specific languages with associated compilers that generate low-level code automatically (e.g. [48]), at the price of flexibility and generality. For programmers who require the flexibility afforded by writing lower-level code directly, help is needed in easing the GPU programming task in various forms. These include clearer formal documentation and "best practices" guides, as well as *tool support* for correctness checking.

We now detail state-of-the-art research in methods for data race analysis of GPU kernels, for which custom checkers have been designed. Effective methods for checking forward progress properties have not yet been studied, and only initial steps have been taken in relation to addressing floating point concerns [20].

	Dynamic bug-finders		**Symbolic bug-finders**		**Verifiers**
	CMC	Oclgrind	GKLEE	GKLEE$_p$	GPUVerify
OpenCL/CUDA support	CUDA	OpenCL	CUDA	CUDA	CUDA + OpenCL
Analysis coverage	Low		Medium		High
Precise bug reports	✓		✓	X	X
Correctness guarantees	X		X		✓
Scalability to large thread counts	High	Medium	Low	Medium	High
Degree of automation	High		Medium		Low

Table 1: A summary of the features of several dynamic and symbolic bug-finders and verifiers for CUDA and OpenCL

4.1. A Taxonomy of Current Tools

Most research and tooling efforts on GPU correctness issues have focused detecting or proving absence of data races. The techniques can broadly be categorized into dynamic analysis methods [13, 49, 27, 30] (though the method of [27] combines dynamic and static analysis), symbolic execution-based approaches [29, 20, 31], and methods that use static, logic-based verification [28, 9, 10, 11, 43].

Table 1 presents a top-level summary of the capabilities of a selection of publicly available tools for analysing CUDA and OpenCL kernels. The table focuses on tools that are actively maintained and is not meant to be exhaustive; indeed, we discuss some further tools in what follows.

Dynamic bug-finding tools, including Oclgrind [39, 38] and CUDA-MEMCHECK (CMC) [36], detect defects during dynamic analysis, either while running a kernel on hardware (CMC), or in emulation mode (Oclgrind). These tools are easy to apply automatically and are precise in the bugs they report because defects are directly observed. They provide low analysis coverage (and no correctness guarantees) because they check a kernel with respect to a single input. Scalability to large thread counts is limited: Oclgrind serializes execution; the overhead of dynamic race instrumentation with CMC is modest, at the price that only races on shared memory, not global memory, can be detected.

Symbolic bug-finding tools, including GKLEE [29] and KLEE-CL [20], retain bug-finding precision but improve on analysis coverage compared with purely dynamic techniques. This is achieved through *symbolic execution*, whereby all possible kernel a constraint solver is used to check for defects along a path for input values, providing correctness guarantees on a per-path basis. State-of-the-art tools build on the KLEE symbolic execution engine [14]. These tools tend to scale poorly to large thread counts: the bottleneck for symbolic execution is constraint solving overhead, and analysis of a large number of threads requires very large sets of constraints to be solved. (In Section 4.3 we discuss GKLEE$_p$, an extension of GKLEE that uses *parametric flows* to scale to larger thread counts.) Automation is basically as high as with dynamic approaches; a little more work may be required to decide which inputs to a kernel should be treated as symbolic, though dependence analysis can partially automate this process [32].

Verification tools, including GPUVerify [9, 10] and PUG [28], provide the highest degree of analysis coverage and are able to guarantee absence of certain types of defects under well-defined conditions. (The words "verification" and "guarantee" are arguably contentious since both tools have known limitations and sources of unsoundness. We use these terms because a key aim of these tools is to guarantee full behavioral coverage, in contrast to bug-finding methods that do not aim to verify correctness and instead prioritize defect detection.) The price for high coverage is a lack of precision in bug reporting: both tools suffer from a high false positive rate, and it may be necessary for developers to manually analyse bug reports and write annotations, in the form of preconditions, loop invariants and barrier invariants [17], to eliminate false alarms.

We now explain how state-of-the-art tools exploit the traditional GPU programming model to achieve scalable analyses (Section 4.2), and describe two such tools, GKLEE and GPUVerify, in more detail (Sections 4.3 and 4.4, respectively).

4.2. Canonical schedules and the two thread reduction

Most existing program analysis tools for GPU kernels are restricted to "traditional" GPU programs, where barrier synchronization is the only means for communication between threads. The tools leverage the simplicity of this setting to achieve scalable analyses, as we now explain. For ease of presentation, we assume in the following that all threads are in a single thread-block, so that all threads synchronize at a barrier.

Race freedom implies determinism. Suppose that a traditional kernel is free from data races. From the initial state of the kernel, a thread executes until it reaches its first barrier without interacting with other threads: any such interaction would constitute a data race. The threads either reach different barriers, in which case the kernel suffers from barrier divergence, or all threads reach the *same* barrier. Thus the initial state of the kernel uniquely determines the state of the kernel when all threads reach their first barrier. If the threads all reach the same barrier then, by a similar argument, the threads execute deterministically until they reach another barrier (or the end of the kernel), and so on. The series of barriers at which the threads will synchronize, and the state of the kernel at each of these barriers, and at kernel exit, is independent of the order in which threads are

scheduled. The argument is presented more formally in [18], and requires that the internal actions of threads are deterministic (e.g., threads should not employ randomization).

Detecting races: "all for one and one for all". Suppose a kernel exhibits a data race with respect to some input and thread schedule. Then *every thread schedule exhibits a data race with respect to this input*. A formal argument for this result is presented in [29]. Informally, if a kernel can exhibit a race for a given input then there exists a non-empty set of *earliest* races for the input: races that are guaranteed to occur, independent of whether any race has already occurred. Every schedule is guaranteed to expose an earliest race.

Restricting to a canonical schedule. The "race freedom implies determinism" and "all for one and one for all" properties combine to yield a powerful schedule reduction method: when analysing a traditional GPU kernel it suffices to consider a *single* schedule, as long as data races are detected for this schedule and reported as errors. As a result, the astronomical schedule space of a GPU kernel executed by thousands of threads can be collapsed so that just one *canonical schedule* is considered. Reduction to a canonical schedule is an example of *partial order reduction* [24], a key state space reduction technique used by many model checking tools.

Reduction to a pair of threads. Because a data race always involves a pair of threads, further scalability can be achieved by restricting race analysis to consider the execution of a single, arbitrary thread pair, using abstraction to over-approximate the behavior of other threads. If pairwise race-freedom can be demonstrated then, because the thread pair under consideration was arbitrary, the kernel is race-free for all threads. If on the other hand a race is detected, the race may be genuine or may be a false alarm induced by the abstraction process. As part of the case studies in Sections 4.3 and 4.4, we discuss below how this *two thread reduction* is employed by the GKLEE$_p$ and GPUVerify tools.

4.3. Symbolic bug finding case study: GKLEE

GKLEE is based on *symbolic execution* [15], and builds on the execution mechanisms provided by the KLEE tool [14].

Consider a program fragment for which control flow depends on the evaluation of a predicate `f` applied to a program input:

```
if (f(input)) {
  /* branch 1 */
} else {
  /* branch 2 */
}
```

Traditional testing requires explicitly supplied concrete inputs that cover both branches. The power of *symbolic* execution is that the variable `input` can be marked as symbolic, and thus treated initially as unconstrained. On reaching the conditional, a constraint solver determines whether there exist concrete values for `input` such that `f` evaluates to *true*, and such that `f` evaluates to *false*. If there exist inputs such that both branches are feasible then "branch 1" is explored under the constraint `f(input)` and "branch 2" is explored under the constraint ¬`f(input)`, ensuring that both branches are covered. The KLEE tool implements symbolic execution of LLVM bit-code and has shown to be very effective in detecting subtle bugs in systems code that evaded manual testing for several years [14].

We use the following example to illustrate how GKLEE [29] evolves the sequential analysis capabilities of KLEE to achieve race checking for CUDA (following a first step in which a CUDA kernel is compiled into LLVM bit-code). In what follows we use `tid` and `bdim` as shorthand for `threadIdx.x` and `blockDim.x`, respectively, and we do not specify the behavior of function `f`.

```
__global__ void race(int *v, int x) {
  v[tid] = v[(tid + x) % bdim];
  __syncthreads();
  if (tid % 2 == 0) {
    ... = v[tid];
  } else {
    v[f(tid)] = ...;
  }
}
```

GKLEE adapts KLEE's memory organization to realize the hierarchical organization shown in Figure 2. Based on the idea of canonical scheduling (Section 4.2), GKLEE symbolically executes the given GPU threads *sequentially* between two barriers, recording potentially racing accesses. A constraint solver is then queried to detect scenarios in which races actually manifest. In the above example, the following instruction precedes the first barrier:

```
v[tid] = v[(tid + x) % bdim];
```

If this instruction is sequentially executed from the point of view of two threads with ids 0 and 1 (for example), we have:

```
v[0] = v[(0 + x) % bdim]; v[1] = v[(1 + x) % bdim];
```

In this execution, a potential racing pair of accesses consists of a read from `v[(0 + x) % bdim]` by thread 0 and a write to `v[1]` by thread 1. A constraint solver easily determines that this pair indeed races for `x` = 1.

By default, GKLEE explicitly models all threads that execute a kernel; this approach does not scale to large thread counts. An extension to GKLEE, named GKLEE$_p$ [31], employs the two thread reduction (Section 4.2) to perform *parameterized* race analysis between an arbitrary pair of threads: regardless of the number of threads that actually execute a kernel, data race detection between barriers can be performed with respect to two threads with symbolic ids, representative of the full thread population.

In the above example, the code after the barrier exhibits *thread divergence*: the even-numbered threads read from `v[tid]` while the odd-numbered threads write to `v[f(tid, p)]`. GKLEE$_p$ handles thread divergence as follows: (1) it creates a separate *flow group* for each branch target, in this case one for all even-numbered threads and one for all odd-numbered threads; (2) within each flow group, GKLEE$_p$ conducts pairwise symbolic execution; (3) the tool checks whether a race is possible *within* each flow group, as well as *across* flow groups. In our example, if `f` is injective (one-to-one), then there are no races within flow groups, while if `f` yields the same value for two odd-numbered threads then there is a write-write race on `v` in the second flow group. There is a potential race *across* flow groups if it is possible for `f(tid)` under condition `tid % 2 ≠ 0` to be equal to `tid` under condition `tid % 2 = 0`. The power of a symbolic solver lies in the fact that it can, very often, find solutions for such equalities by analyzing the function body of functions such as `f` and generating the necessary set of constraints.

A drawback of parameterized checking is that using the two thread reduction necessitates abstraction of additional threads. This means that, unlike GKLEE, GKLEE$_p$ can yield false alarms. Another potential drawback is that the number of flow groups can grow exponentially. Recent work has shown that static analysis can be used to overcome this problem in several practical examples [32].

4.4. Verification case study: GPUVerify

The GPUVerify tool [9, 10, 4] aims for full verification of race-freedom for OpenCL and CUDA kernels. As with GKLEE, a kernel is first translated to LLVM bit-code. The canonical scheduling idea (Section 4.2) is exploited to transform the kernel into a *sequential* program in the Boogie intermediate verification language [7]. During this transformation, memory access instructions are instrumented so that the conditions for checking race-freedom are encoded as assertions to be checked in the Boogie program. Thus proving race-freedom of a kernel amounts to verifying the sequential Boogie program generated by this process. This allows for re-use of the Boogie verification framework (widely used by other verification tools, including VCC [19] and Spec# [8]), where a program is checked by generating a verification condition that can be discharged to an SMT solver. Verification of programs with loops requires loop invariants, and GPUVerify uses the Houdini candidate-based invariant generation algorithm [23] for this purpose.

To achieve scalability to large numbers of threads, GPUVerify employs a two thread reduction (see Section 4.2). Unlike GKLEE$_p$, which handles thread-divergent control flow by considering multiple pairs of threads, GPUVerify symbolically encodes all control flow scenarios with respect to a single thread pair via *predicated execution* [21, 9, 10]. Because additional threads are modeled abstractly when the two thread reduction is employed, analysis using GPUVerify can yield false alarms. Another source of false alarms stems from a lack of preconditions describing constraints on kernel execution; to some extent this can be overcome through interception of kernels at runtime to capture details of their execution environment [6]. A large experimental evaluation with respect to publicly available benchmark suites has demonstrated that GPUVerify is able to handle many practical GPU kernels [4], and the tool has been used to find numerous defects in these benchmark suites (see e.g., a confirmed and fixed SHOC bug report [44] and acknowledgments to Jeroen Ketema in the Rodinia 3.1 release notes [40]). For kernels whose correctness depends on richer shared state properties, GPUVerify supports *barrier invariant* annotations [17]. These allow the precision of the shared state abstraction to be increased, at the expense of manual effort.

5. Call to Action

We have discussed a number of issues that affect GPU program correctness and the capabilities of current solutions. We now consider the future role GPUs will play in software systems and the extent to which the current analysis techniques can be effective in improving the quality of this software.

GPUs will become more pervasive. The high development cost associated with application-specific processors makes the idea of removing such processors from an embedded chip, and implementing its functionality as software running on a GPU, an appealing one. The idea that GPUs may gain traction in, for example, the automotive domain further motivates the need to consider safe GPU programming. We may be comfortable with GPU-powered infotainment systems of cars being developed without rigorous correctness analysis, but we are likely less comfortable with this situation if next-generation GPUs play a more fundamental role.

As James Larus points out in a recent CACM letter to the editor [26], for reasons of liability manufacturers will be increasingly required to demonstrate that software has been rigorously developed, and will thus turn their attention to the best available tools and practices. As GPUs become more pervasive there will be an even greater need for the sorts of tools surveyed in this article.

Current tools show promise. Existing tools for GPU program analysis have demonstrated that they can help improve the quality of GPU software. For example, GKLEE and GPUVerify have found numerous bugs in kernels shipped with the Lonestar, Parboil, Rodinia and SHOC suites.

However, many challenges lie ahead. We conclude with a call to action on what we believe to be some of the most pressing issues for researchers and industry.

Solving basic correctness issues. While today's tools show that analysis for GPU correctness issues related to mis-synchronization is tractable, Table 1 shows that no single method provides high analysis coverage and precise bug reporting while scaling to large thread counts. We call to researchers for innovation in this regard, through clever combinations of methods or via new approaches to the analysis problem. A particular challenge is reasoning about fine-grained concurrency; support for reasoning about atomic operations in mature tools is limited [16, 5], though recent methods for reasoning about atomics via concurrent separation logic show promise [2]. Fully supporting fine-grained concurrency will require accounting for the relaxed memory behaviour of GPU architectures [1].

Equivalence checking. A GPU program is often the evolution of a CPU program, via several optimizing transformations. For floating-point code these transformations may be “fuzzy”, preserving functionality within an acceptable margin of error and not providing bit-level equivalent results. A fascinating open research challenge is to provide assistance in this transformation process via formal equivalence checking methods that tolerate an acceptable degree of fuzziness; this will require coping with the combined challenges of reasoning about floating-point and concurrency.

Clarity from vendors and standards bodies. Recent research into GPU memory consistency [1] indicates that vendor documentation and open standards are not clear on subtle details related to the concurrency semantics of GPU kernels. These subtleties are of essential importance to compiler-writers, expert programmers, and developers of formal analysis techniques. We call to vendors and standards bodies to engage further with the programming languages research community to achieve better clarity.

User validation of tools. Despite their promise, the current analysis tools for GPU programming have only a modest following of professional developers. Higher user engagement is needed to better understand the most pressing problems that developers face. One preliminary success story here is a recent integration of GPUVerify into the Graphics Debugger 2.0 for the ARM Mali GPU series [3]. We call on other platform vendors to similarly promote uptake and continued development of promising research tools.

Concerted community action coupled with dialog with industry is a sure-footed way to attain these objectives. After all, future Exascale machines and intelligent navigation systems must trust GPUs to safely accelerate computing, and this safety requires tools to aid in systematic software development – something that this chapter scratches the surface of.

Acknowledgments

We are grateful to Geof Sawaya, Tyler Sorensen and John Wickerson for feedback on a draft of this chapter, and to Daniel Poetzl for guidance regarding the placement of memory fences in Fig. 4.

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 557–591 , 2015.

[2] A. Amighi, S. Darabi, S. Blom, and M. Huisman. Specification and verification of atomic operations in GPGPU programs. In SEFM, pages 69–83, 2015.

[3] ARM. Debugging OpenCL Applications with Mali Graphics Debugger V2.1 and GPUVerify, https://community.arm.com/groups/arm-mali-graphics/blog/2015/04/14/debugging-opencl-applications-with-mali-graphics-debugger-v21-and-gpuverify. Retrieved Jan 19, 2016.

[4] 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, 2014.

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

[6] E. Bardsley, A. F. Donaldson, and J. Wickerson. KernelInterceptor: automating GPU kernel verification by intercepting kernels and their parameters. In IWOCL, pages 7:1–7:5, 2014.

[7] M. Barnett et al. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[8] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[9] 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.

[10] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a verifier for GPU kernels. In *OOPSLA*, pages 113–132, 2012.

[11] S. Blom, M. Huisman, and M. Mihelcic. Specification and verification of GPGPU programs. *Sci. Comput. Program.*, 95:376–388, 2014.

[12] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.

[13] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic analysis of CUDA programs. In *Proceedings of the Third Workshop on Software Tools for MultiCore Systems (STMCS)*, 2008.

[14] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[15] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[16] W.-F. Chiang, G. Gopalakrishnan, G. Li, and Z. Rakamaric. Formal analysis of GPU programs with atomics via conflict-directed delay-bounding. In *NFM*, pages 213–228, 2013.

[17] N. Chong, A. F. Donaldson, P. Kelly, J. Ketema, and S. Qadeer. Barrier invariants: a shared state abstraction for the analysis of data-dependent GPU kernels. In *OOPSLA*, 2013.

[18] N. Chong, A. F. Donaldson, and J. Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *POPL*, pages 397–410, 2014.

[19] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, pages 23–42, 2009.

[20] 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.

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

[22] A. Danalis et al. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU* 2010, pages 63–74, 2010.

[23] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, pages 500–517, 2001.

[24] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of LNCS. Springer, 1996.

[25] Khronos. The OpenCL C specification, November 2013.

[26] J. Larus. Responsible programming not a technical issue. In *Commun. ACM*, page 8, Oct. 2014. Letter to the Editor.

[27] A. Leung, M. Gupta, Y. Agarwal, et al. Verifying GPU kernels by test amplification. In *PLDI*, pages 383–394, 2012.

[28] G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *FSE*, pages 187–196, 2010.

[29] 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, 2012.

[30] P. Li, C. Ding, X. Hu, and T. Soyata. LDetector: A low overhead race detector for GPU programs. In *Proc. 5th Workshop on Determinism and Correctness in Parallel Programming (WODET)*, 2014.

[31] P. Li, G. Li, and G. Gopalakrishnan. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *SC*, pages 29:1–29:10, 2012.

[32] P. Li, G. Li, and G. Gopalakrishnan. Practical symbolic race checking of GPU programs. In *SC*, pages 179–190, 2014.

[33] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, Apr. 1984.

[34] Nvidia. CUDA C programming guide, version 7.5, September 2015.

[35] Nvidia. CUDA code samples, https://developer.nvidia.com/gpu-computing-sdk. Retrieved Jan 19, 2016.

[36] Nvidia. CUDA-MEMCHECK, https://developer.nvidia.com/CUDA-MEMCHECK. Retrieved Jan 19, 2016.

[37] Nvidia CUDA ZONE forum, https://devtalk.nvidia.com/default/topic/632471/is-syncthreads-required-within-a-warp-. Retrieved Jan 19, 2016.

[38] Oclgrind, https://github.com/jrprice/Oclgrind. Retrieved Jan 19, 2016.

[39] J. Price and S. McIntosh-Smith. Oclgrind: An extensible OpenCL device simulator. In *IWOCL*, 2015.

[40] Rodinia version history, https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/TechnicalDoc. Retrieved Jan 19, 2016.

[41] SAGE: The First National Air Defense Network, http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/sage/. Retrieved Jan 19, 2016.

[42] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010.

[43] R. Sharma, M. Bauer, and A. Aiken. Verification of producer-consumer synchronization in GPU programs. In *PLDI*, pages 88–98, 2015.

[44] SHOC project bug report, https://github.com/vetter/shoc/issues/30. Retrieved Jan 19, 2016.

[45] Stack Overflow: Inter-Block Barrier in CUDA, http://stackoverflow.com/questions/7703443/inter-block-barrier-on-cuda. Retrieved Jan 19, 2016.

[46] Stack Overflow: How to implement a critical section in CUDA? http://stackoverflow.com/questions/2021019/how-to-implement-a-critical-section-in-cuda. Retrieved Jan 19, 2016.

[47] J. Stratton et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, UIUC, 2012.

[48] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *TACO*, 9(4):54, 2013.

[49] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. GMRace: Detecting data races in GPU programs via a low-overhead scheme. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):104–115, 2014.