Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels

Ethel Bardsley and Alastair F. Donaldson

Imperial College London
{emb2009,afd}@imperial.ac.uk

Abstract. We describe the design and implementation of methods to support reasoning about data races in GPU kernels where constructs other than the standard barrier primitive are used for synchronization. At one extreme we consider kernels that exploit implicit, coarse-grained synchronization between threads in the same *warp*, a feature provided by many architectures. At the other extreme we consider kernels that reduce or avoid barrier synchronization through the use of *atomic* operations. We discuss design decisions associated with providing support for warps and atomics in GPUVerify, a formal verification tool for OpenCL and CUDA kernels. We evaluate the practical impact of these design decisions using a large set of benchmarks, showing that warps can be supported in a scalable manner, that a coarse abstraction suffices for efficient reasoning about most practical uses of atomic operations, and that a novel, refined abstraction captures an important design pattern where atomic operations are used to compute unique array indices. Our evaluation revealed two previously unknown bugs in publicly available benchmark suites.

1 Introduction

The rise of the use of graphics processing units (GPUs) for general purpose programming allows for high-throughput massively parallel problems to be accelerated on relatively cheap commodity hardware. This throughput is achieved on GPUs by running thousands of threads in parallel. GPUs are thus suited to a variety of parallel tasks ranging from graphics and imaging to simulation, medical imaging, and computational finance.

The massively parallel nature of graphics cards gives rise to concurrency bugs, such as data races and deadlocks. Data races lead to non-determinism, incorrect computation and undefined behavior. There has been recent interest in the program analysis community on methods for formal or semi-formal analysis of GPU kernels, leading to methods for finding bugs in [14,5] or proving correctness properties of [13,3,8,12] GPU kernels, principally focused on data races.

The main GPU programming models, OpenCL [10] and CUDA [16] organize threads into multiple, independent work groups, and provide a *barrier* operation for synchronizing threads within the same work group. When a thread reaches a barrier it must wait for every thread in its work group to arrive at the barrier. The barrier ensures that all memory accesses issued before the barrier have completed on barrier exit. The threads in the work group then continue execution beyond the barrier. From the perspective

of race analysis tools, barriers allow analysis to be restricted to separate *barrier intervals* [14], and each barrier interval can be checked for data races with respect to a *single* thread schedule [13,14,3,5]. However, the runtime overhead of barrier synchronization is high [16, §5.4.3] and there are instances where ensuring race-freedom using barriers is cumbersome or impossible without destroying parallelism. Two features of modern GPU designs allow these problems to be reduced to some extent: *warps*, where implicit synchronization is guaranteed due to lock-step execution of threads, and *atomic* read-modify-write operations, which enable memory locations to be updated asynchronously and lock-free synchronization to be implemented. Because concurrent atomic operations on a memory location are not considered racy, atomics allow acceptable non-determinism to arise from the order of thread interleavings within a barrier interval, thus it is no longer sound to consider a single thread schedule during race analysis.

In this paper, we discuss design decisions associated with providing support for warps and atomics in GPUVerify, an existing verification technique and tool for OpenCL and CUDA kernels [3]. For warps we present a *two-pass* approach where intra- and inter-warp analyses are performed separately, and a *re-sync* approach where intra-warp synchronization at the instruction level is accounted for in a general analysis. In contrast to a recent method for bug-finding in the presence of atomics which heuristically explores thread interleavings [4], we employ abstraction to enable verification of data race-freedom. For kernels that use atomics merely for asynchronous shared state updates we show that a coarse abstraction, where shared memory reads yield arbitrary values, suffices for analysis. This coarse abstraction yields false positives when atomics are used to ensure non-interference between threads. We have identified an important use case where threads atomically increment a counter to compute a successive series of unique indices, and present a novel refined abstraction to efficiently capture this use case.

We evaluate the precision and performance of our methods using a set of 199 CUDA and 190 OpenCL kernels. Warp-aware analysis allows verification of 7 kernels whose race-freedom depends on inter-warp synchronization; GPUVerify previously reported false positives for these examples. Atomics are used by 22 kernels, making verification tools inapplicable to these examples prior to this work. We discovered two previously unknown bugs in these kernels, in the ParBoil [18] and CUDA 5.0 SDK [16] suites, one directly related to use of atomics, which we have reported to the developers concerned. After fixing these bugs, we were able to verify 15 of the kernels that used atomics.

In summary, our main contributions are:

- Two methods for supporting warps when reasoning about races in GPU kernels;
- A coarse abstraction for accommodating atomic operations and a novel refined abstraction to capture an important atomic-based synchronization pattern;
- An implementation of our methods in the open source GPUVerify tool, and an experimental evaluation over a large set of publicly available kernels.

2 Background

We briefly review important aspects of the GPU kernel programming model (Section 2.1), discuss warps and atomics in more detail (Section 2.2), and summarize the GPUVerify verification method on which we build (Section 2.3).

```
kernel void add(local float *A)
{
  A[tid] = A[tid] +
         A[(tid + 1)%N];
}
```

(a) OpenCL kernel with data race

```
kernel void add(local float *A)
{ float temp = A[(tid + 1)%N];
  barrier();
  A[tid] = A[tid] + temp;
}
```

(b) Data race eliminated via barrier

Fig. 1: OpenCL kernels illustrating data races and the use of barriers

2.1 GPU kernel programming model

A conventional modern GPU (e.g. a design from NVIDIA or AMD) consists of many *processing elements* (PEs) organized into *compute units*. Each PE is equipped with a portion of private memory, each compute unit includes a portion of shared memory accessible to the PEs of the compute unit, and there is a global memory available to all PEs on the GPU. The OpenCL [10] and CUDA [16] programming models roughly mirror this structure; we discuss the OpenCL case. On OpenCL, a *kernel* is executed in parallel by a number of *work groups*, each of which runs on a compute unit. A work group consists of a number of *work items* (often, and in this paper, referred to as threads), each of which executes on a PE. Thread-private variables are stored in PE private memory, and threads in a work group share data stored in the memory space of the compute unit. Data in GPU global memory is shared among all threads executing a kernel.

Behavior of the kernel is specified by a single kernel function, a template describing the behavior of each thread. A thread has access to a thread id which it can use to behave in an individual manner. Threads in the same work group synchronize via the *barrier* primitive. When a thread reaches a barrier the thread stalls until all threads in its work group have reached the same barrier. The barrier enforces memory ordering, guaranteeing that memory accesses issued before the barrier will have completed before threads commence execution beyond the barrier. Barriers allow synchronization only between threads in the same work group.

The GPU kernel programmer must carefully place barriers to avoid *data races*:

Definition 2.1 (Warp- and atomic-oblivious data race). *An execution of a GPU kernel has a data race if two distinct threads access a common memory location, at least one of the accesses modifies the location, and no barrier synchronization between the threads separates these accesses.*

The behavior of a kernel with a data race is undefined according to the OpenCL specification. In practice data races lead to non-determinism, and expose re-orderings of loads and stores due to relaxed underlying memory models.

Figure 1a shows a simple OpenCL kernel[1] that exhibits data races between adjacent threads. There is a race, for example, between threads 0 and 1 because thread 0 reads

[1] OpenCL supports multi-dimensional arrangements of work groups and threads. For ease of presentation all our example kernels are one-dimensional, and we use `tid` and `N` to abbreviate the OpenCL syntax for the id of a thread and the total number of threads, respectively.

from $A[1]$ (via `A[(tid + 1)% N]`), thread 1 writes to $A[1]$ (via `A[tid]`) and there is no guarantee on the order in which these accesses will occur. Figure 1b shows how a barrier can be used to eliminate this race: all threads must reach the barrier until any can proceed past the barrier, thus the conflicting accesses allowed by the kernel of Figure 1a cannot be simultaneous in the kernel of Figure 1b.

2.2 Warps and atomics

Warps and implicit synchronization GPU architectures from NVIDIA and AMD provide a degree of implicit synchronization between threads. On NVIDIA hardware, threads are divided into power-of-two-sized subgroups of at least size 32, known as *warps* [16, §4.1]. AMD designs provide a similar notion of a *wavefront* [1] of threads. We use the term *warp* to denote this feature in general. Threads in the same warp execute in lock-step, sharing a program counter. Threads in the warp cannot simultaneously execute distinct instructions (*predicated execution* [16, §5.4.2] is used to handle non-uniform execution of conditional code by a warp), thus the scope for data races and non-determinism within a warp is reduced. This mode of execution is termed *SIMT* (Single Instruction, Multiple Thread) by NVIDIA, and is analogous to SIMD (Single Instruction Multiple Data).

Warp-level synchronization guarantees can allow expensive barrier synchronizations to be omitted. If the kernel of Figure 1a is executed by 32 threads on an NVIDIA GPU, these threads will be scheduled as a single warp. Every thread will read from `A[(tid + 1)%N]` before *any* thread writes to `A[tid]`, making a data race impossible. Intra-warp races can only occur when two threads in a warp attempt to simultaneously update the same location, for example via a statement such as `A[0] = tid`.

Exploitation of warps is recommended in the CUDA programming guide [16], and efficient algorithms have been developed that depend on this feature: Sengupta et al show the number of barrier synchronization operations required during a parallel scan can be reduced from $\log_2(N)$ to $\log_{32}(N)$, where N is the number of threads, by first scanning within warps, using implicit synchronization, and then aggregating across warps [17]. The OpenCL programming model aims to be general purpose and thus does not acknowledge the existence of warps, so relying on platform-specific warp behavior leads to non-portable code. However, the new OpenCL 2.0 extension specification [9, §9.17, p133] contains an optional extension for *subgroups*, which allow the behavior of warps to be captured. Furthermore, since many OpenCL kernels are ported from CUDA versions, a warp-sensitive analysis for OpenCL can aid in distinguishing between data races preserved by the porting process, and data races introduced by porting due to assumptions about warps which are not valid in OpenCL.

Atomic operations OpenCL and CUDA are equipped with a set of atomic *read-modify-write* intrinsics. Concurrent atomic operations on the same memory location are *not* considered racy, thus atomics allow a memory location to be updated asynchronously by multiple threads in a manner that is considered race-free. Such updates can lead to non-determinism due to the order in which threads are scheduled. The example kernel of Figure 2a uses the OpenCL `atomic_inc` intrinsic to implement a histogram: `A` is an array of data values, and `B` is an array of buckets; on finding that value `t` is present in `A`, a thread increments the bucket at offset `t` from `B`. Using an atomic operation ensures that buckets are incremented consistently, and because increment operations are commutative,

```
kernel void histo(local int* A,
                  local int* B) {

  int t = A[tid];
  atomic_inc(&B[t]);

}
```

(a) Efficient histogram implementation using an atomic operation

```
kernel void histo(local int* A,
                  local int* B) {
  int t = A[tid];
  for (int j = 0; j < N; j++) {
    if (tid == j)
      B[t]++;
    barrier();
} }
```

(b) Without atomics, a race-free histogram is not efficient

Fig. 2: An illustration of the advantages brought by atomic operations

the order in which threads interleave is not important. If `atomic_inc(&B[t])` in Figure 2a was changed to a non-atomic increment, `B[t]++`, there could be data races on buckets, leading to an insufficient number of increments at best, and memory corruption at worst. It is not feasible to safely implement this kind of kernel without atomics; the kernel of Figure 2b shows how barrier synchronization can be used to serialize bucket updates, but this destroys parallelism by effectively serializing the kernel as a whole. Atomics can also be used for communication between threads in distinct work groups, to ensure race-freedom. In Section 4.2 we show how atomics can be used to compute disjoint array indices across multiple work groups.

Data races in the presence of warps and atomics We refine Definition 2.1 to take account of warps and atomics. If two threads are in the same warp then a *warp synchronization* occurs between the threads on execution of every instruction. The new parts of the definition are emphasized:

Definition 2.2 (Warp- and atomic-aware data race). *An execution of a GPU kernel has a data race if two distinct threads access a common memory location, at least one of the accesses modifies the location,* **at least one of the accesses is non-atomic***, and no barrier* **or warp** *synchronization between the threads separates these accesses.*

2.3 Race analysis using GPUVerify

The GPUVerify tool [3] takes as input an OpenCL or CUDA kernel, optionally annotated with loop invariants and procedure specifications. GPUVerify uses the Clang/LLVM framework to process the kernel, translating it into a sequential program expressed in the Boogie verification language [11]. This transformation encodes race checks using assertions such that if the sequential program can be proven correct[2] (i.e. free from assertion failures) then the kernel is guaranteed to be free from data races. The sequential program is checked using the Boogie verifier [2].

GPUVerify scales to large thread counts by encoding in the sequential program the execution of the kernel by an arbitrary distinct *pair* of threads [3]. This pair of threads

[2] We use *correct* to mean *partially correct*; GPUVerify does not perform termination analysis.

are considered to execute in lock-step, so that they execute exactly the same sequence of instructions. Uniform execution of conditionals and loops is enforced in the sequential program via *predicated execution* [3]. This fixed schedule eliminates thread interleavings. However, data race analysis with respect to arbitrary thread interleavings is possible by maintaining read and write sets for shared arrays. Let (s, t) denote the pair of threads under consideration, and associate with each array A a set $\mathcal{R}_A$ of read offsets and $\mathcal{W}_A$ of written offsets. Execution of a write instruction where s and t write to A at offsets o_s and o_t, respectively, is modelled by adding o_s to $\mathcal{W}_A$ and then checking that o_t does not belong to $\mathcal{R}_A \cup \mathcal{W}_A$. Read operations are handled similarly, with the check relaxed to allow read sharing. At a barrier, $\mathcal{R}_A$ and $\mathcal{W}_A$ are set to be empty for every array A. This transformation is valid in the context of race checking, as a correct kernel is deterministic for a given input, and threads cannot communicate aside from barriers, between which there is no guaranteed schedule. The effects of the other threads are thus abstracted.

Consider again the example of Figure 1a. GPUVerify reasons that this kernel is racy by selecting an arbitrary pair of threads s and t, and introducing read and write sets, $\mathcal{R}_A$ and $\mathcal{W}_A$, for the array A, which are initially empty. The reads from `A[tid]` and `A[(tid + 1)%N]` are first checked by adding s and $(s + 1)\%N$ to $\mathcal{R}_A$ and checking that t and $(t+1)\%N$ do not belong to $\mathcal{W}_A$; this holds trivially because $\mathcal{W}_A$ is empty. The write to `A[tid]` is then checked by adding s to $\mathcal{W}_A$ and checking that $t \notin \mathcal{R}_A \cup \mathcal{W}_A$. This logging and checking is encoded using a set of constraints, and races between specific threads are detected by solving for s and t. In the case $t = (s + 1)\%N$, we have $t \in R_A \cup W_A$, so a race is reported.

For the two-thread reduction used by GPUVerify to be sound it is necessary to over-approximate the effects of additional threads. The simplest solution is to make *no* assumptions about the behavior of additional threads, assuming that these threads may update the shared state arbitrarily. This can be achieved in two ways [3]:

- **Adversarial abstraction**: shared arrays are removed altogether, and every read from a shared array instead returns a non-deterministic value
- **Equality abstraction**: shared arrays are updated non-deterministically (havocked) each time a barrier is reached

Adversarial abstraction is sufficient for checking race-freedom of many kernels and avoids the need to reason about arrays. Equality abstraction (so called because both threads have an equal but arbitrary view of the shared state) is more refined, and is necessary when race-freedom of a kernel requires agreement between threads on the contents of a shared memory location, such as a flag.

The soundness of the two-thread abstraction is argued in [3], and of race analysis via a single schedule in [14,19].

3 Warp-Aware Race Analysis

We considered two approaches to supporting intra-warp synchronization during race analysis, which we call the *re-sync* method and the *two-pass* method.

3.1 Re-sync method

In the re-sync method (so called because threads synchronize at barriers, and analogously threads in the same warp *re*-synchronize after each instruction), intra- and inter-warp

races are checked simultaneously. Race analysis works as described in Section 2.3, but after each uniform read and write instruction with associated array A, the sets $\mathcal{R}_A$ and $\mathcal{W}_A$ are set to be empty if the threads under consideration belong to the same warp.

Consider the example of Figure 1a with 64 threads, i.e. $N = 64$, and suppose that these threads are organized into two warps, each of size 32. No races will be detected between threads s and t in the same warp, i.e. if $s, t \in \{0, \ldots, 31\}$ or $s, t \in \{32, \ldots, 63\}$: the read set $\mathcal{R}_A$ is cleared immediately before the write to `A[tid]` is analyzed. On the other hand, races will be detected for the cases $s = 31, t = 32$ and $s = 63, t = 0$; we explain the $s = 31, t = 32$ case. After the read operations we have $\mathcal{R}_A = \{s, s + 1\} = \{31, 32\}$; because s and t are in different warps $\mathcal{R}_A$ is *not* made empty; the write is then analyzed by adding s, i.e. 31, to $\mathcal{W}_A$ and checking whether t, i.e. 32, belongs to $\mathcal{R}_A$. This is the case, so a data race is reported.

This is sufficient to maintain soundness in the uniform case, where the threads follow the same path, as for some racy code `A[o] = ...`, o_s will still be in W_A when $o_t \notin W_A$ is checked, and thus the assertion failure will still be reported. For the divergent case (referred to by [14] as a "porting race"), this reset is predicated, such that, for threads s, t, with enabled predicates p_s, p_t, the reset is predicated by $p_s \wedge p_t$. For example, in the racy code `if (tid < 16) {A[o] = 1} else {A[o] = 2}`, if s follows the *then* branch and t takes the *else*, the reset won't occur until the threads re-converge, and so the case $o_s = o_t$ will report assertion failure as per the regular GPUVerify method.

3.2 Two-pass method

The two-pass method involves two independent analyses that can run in parallel, one checking exclusively for inter-warp data races, the other exclusively for intra-warp data races. Inter-warp data race analysis proceeds according to the method outlined in Section 2.3, except that the arbitrary threads s and t are constrained to reside in different warps. For intra-warp race analysis, s and t are constrained to reside in the same warp, and for each write instruction we check that the offsets o_s and o_t being written to are different; there is no need to maintain read and write sets or analyze read instructions.

With respect to the running example of Figure 1a, with 64 threads organized as two warps of size 32, the intra-warp case of the two-pass method determines that the write `A[tid]` leads to disjoint accesses for any distinct threads s, t, thus there are no intra-warp races. The inter-warp case detects the races between threads 31 and 32 and threads 0 and 63 in the manner described for the re-sync method, except that there is no need to consider setting the read/write sets for A to be empty between instructions.

This is implemented as, when thread paths are uniform, altering the log mechanism such that, for writes, $W_A := \{o_s\}$ instead of $W_A := W_A \cup \{o_s\}$, and making it the empty set otherwise. This maintains soundness, as the write set will contain the current instruction's offset for the unified case, and in the non-unified case the technique behaves as without this modification.

It is clear that the re-sync and two-pass methods achieve the same goal. Our hypothesis was that the two-pass method might lead to faster verification by decomposing analysis into two simpler cases that can be checked in parallel. Our experiments in Section 5 validate this hypothesis with respect to a 215 example kernels: the two-pass method outperforms the re-sync method in many cases.

3.3 Inter-warp synchronization and shared state abstraction

Recall from Section 2.3 that the two-thread reduction used by GPUVerify depends on an accompanying abstraction of the shared state. Adversarial abstraction provides no guarantees about the contents of the shared state and thus combines directly with our approaches to warp-based synchronization. Combining warp-level synchronization with equality abstraction requires some care. With equality abstraction, shared arrays are havocked at every barrier. Consider the following code snippet, which is incorrect when executed by a single warp of at least three threads:

```
if(tid == 0) {
  A[0] = 1; A[1] = 1; A[2] = 1;
}
// At this point, A = { 1, 1, 1, ... }
A[tid] = 0;
// Now A = { 0, 0, 0, ... }
if(tid == 0) {
  // The assertion should thus fail
  assert(A[0] == 1 || A[1] == 1 || A[2] == 1);
}
```

Suppose we analyze this example using the two-thread reduction with straightforward equality abstraction. Consider the pair of threads 0, 1. After execution of the first conditional there are no data races and the threads' view of A is $\{1, 1, 1, 1, \dots\}$. The assignment `A[tid] = 0` by threads 0 and 1 leads to a state where $A = \{0, 0, 1, 1, \dots\}$. This is incomplete: it does not take into account the actions of additional threads. Hence the pair 0, 1 erroneously conclude, at the assertion, that at least one of `A[0]`, `A[1]` and `A[2]` is equal to 1, namely `A[2]`.

To rectify equality abstraction in the presence of warps it is necessary to perform additional havocking: after a write instruction to array A, the array A must be havocked to reflect the fact that *other* unmodelled threads in the warp have also modified A. With respect to the above example this means that the threads' view of `A` is arbitrary after each instruction, leading (as desired) to states in which the assertion fails.

4 Race Analysis and Abstraction for Atomic Operations

As discussed in Section 2.2, atomic operations relax the definition of what constitutes a data race, reflected in Definition 2.2. This allows designated memory locations to be updated concurrently in manner that is considered non-racy. Such concurrent updates are a valid source of non-determinism, violating the assumption on which race analysis in GPUVerify and other methods rests: that a race-free kernel behaves deterministically. As a result, it is not sound in general to restrict analysis to a single thread schedule in the presence of atomic operations.

For a precise analysis geared towards bug-finding this is problematic: to accurately find bugs arising from atomic manipulation it is necessary to resort to exploring thread interleavings. This has been investigated in the context of the GKLEE bug-finding tool for CUDA [4], where delay bounding [7] is used to limit schedule explosion.

We have observed that in practice most GPU kernels that use atomics do so for simple purposes, such as updating shared data asynchronously or computing unique

array indices. We focus here on using abstraction to prove race-freedom for these sorts of kernels, without resorting to exploration of thread interleavings.

4.1 Over-approximating atomics with adversarial abstraction

Suppose we wish to analyze a kernel that updates elements of an array A atomically.[3] If we handle A using adversarial abstraction, so that every read from A yields a non-deterministic result, then there is no need to explicitly consider thread interleavings arising from non-determinism introduced by atomic updates to A: adversarial abstraction encodes *at least* the non-determinism that could arise from such updates.

Under adversarial abstraction we can adapt the race analysis procedure described in Section 2.3 as follows. For a shared array A, in addition to read and write sets $\mathcal{R}_A$ and $\mathcal{W}_A$ we introduce an *atomic* set $\mathcal{A}_A$ recording offsets from A that have been accessed atomically. Suppose the threads under consideration are s and t, and that an instruction ι causes s and t to access offsets o_s and o_t of a shared array A, respectively. We log the access made by s by adding o_s to $\mathcal{R}_A$, $\mathcal{W}_A$, or $\mathcal{A}_A$ depending on whether ι is a read, write or atomic operation. We then check the access made by t, reporting a data race if:

- $o_t \in \mathcal{W}_A \cup \mathcal{A}_A$ in the case where ι is a non-atomic read
- $o_t \in \mathcal{R}_A \cup \mathcal{W}_A \cup \mathcal{A}_A$ in the case where ι is a non-atomic write
- $o_t \in \mathcal{R}_A \cup \mathcal{W}_A$ in the case where ι is an atomic operation

This extension of our method is sufficient for analysis of kernels where the return values of atomic operations do not influence whether or not data races occur. An example is the histogram kernel of Figure 2a: array B is updated atomically, thus B must be adversarially abstracted. However, because no data is subsequently read from B, this coarse abstraction of B cannot lead to false positive data race reports. Our approach thus allows for sound race analysis of this simple example. In Section 5 we report on a data race we detected in one of the ParBoil benchmarks [18], where both atomic and non-atomic operations are used to manipulate the same array without adequate synchronization.

It is *not* sound in general to use equality abstraction for an array that is atomically updated: atomics allow non-determinism between barriers, so multiple reads from an atomically-manipulated memory may yield different results.

4.2 A refined abstraction for repetition-free atomic operations

The example of Figure 3 demonstrates how atomic operations can be used to compute disjoint indices for array accesses. In the figure, `in` and `out` are distinct shared arrays of length `MAX`, and `c` is a pointer to a shared counter, initialized to zero. The unspecified `compute` procedure performs some computation on the `i`-th element of `in`, returning a value. The `atomic_inc` operation atomically increments the shared memory location pointed to by its argument and returns the previous value of this location.

This design pattern is useful in parallel processing of data where the computation time per data element may vary in an unpredictable manner. Such variance means that it

[3] In practice atomics are often used to update single memory locations, such as counters; we can regard these as single-element arrays.

```
private int i = atomic_inc(c);
while(i < MAX) {
  out[i] = compute(in, i);
  i = atomic_inc(&c);
}
```

Fig. 3: Using atomic increment to compute disjoint array indices.

is not possible to achieve high performance by statically allocating a fixed chunk of data elements to each thread. A classic example of this is fractal image computation, where time to convergence for a pixel varies dramatically across the image, and we have seen the above design pattern used (in a more sophisticated form) for lock-free division of work in optimized Mandelbrot fractal kernels that ship with the CUDA SDK.

The basic atomic support described in Section 4.1 would report a false positive data race for the above example. This is due to adversarial abstraction of the counter, which allows two distinct threads to see common values returned by `atomic_inc`, leading to write-write data races on `A`. The example is in fact race-free when executed by multiple threads. This is because, although the sequence of values a thread obtains by calling `atomic_inc` is dependent on the thread schedule, the sequences of values obtained by two distinct threads must be disjoint—the counter only ever increases and thus (assuming the counter does not overflow) it will never contain the same value twice.

If we can identify that a location l is accessed exclusively via `atomic_inc` operations then we can refine adversarial abstraction to take advantage of the "repetition-free" nature of this operation. Suppose we have a set $\mathsf{used}(l)$ recording all the values that have been read from l so far during the program. Initially $\mathsf{used}(l)$ is empty. We can model an application of `atomic_inc` to location l by returning a non-deterministically chosen value that does *not* belong to $\mathsf{used}(l)$, and then adding this value to $\mathsf{used}(l)$ so that it is not returned again in future. This refined abstraction thus knows nothing about the location l *except* that its current value is different from any other value previously returned by `atomic_inc`. This additional knowledge is sufficient to capture the case where `atomic_inc` is used to derive a unique array index.

More generally, we can compute this refined abstraction for a location l if we can determine that l is manipulated exclusively using a single, *repetition-free* function.

Definition 4.1 (Repetition-free function). *Let S be a set and $f : S \to S$ a function, with $f^k : S \to S$ denoting f applied k times. We say that f is* repetition-free *if for every $x \in S$ and $m, n \geq 0$ with $m \neq n$, $f^m(x) \neq f^n(x)$. That is, f has no* periodic points*.*

The `atomic_inc` operation can be viewed as updating a location storing value v to store $f(v)$, where f is the repetition-free function defined by $f(x) = x + 1$. We can consider the `atomic_add` operation, which takes a location and a non-negative integer argument n, similarly in the case where n is positive: applying `atomic_add` to a location holding value v updates the location to store $f^n(v)$, where $f(x) = x + 1$. The operations `atomic_dec` and `atomic_sub` can be treated analogously using the repetition-free function g defined by $g(x) = x - 1$.

This abstraction is technically unsound because it does not take into account the possibility of overflow, which may cause a location to yield the same value twice if an operation such as increment is called an extremely large number of times. Our aim in this work is to provide pragmatic support for reasoning about kernels that use atomics, thus we use the abstraction without regard for overflow. If overflow is a concern then soundness can be restored through the addition of overflow checks (with a corresponding increase in verification burden).

4.3 Implementation issues for atomics

Supporting atomic operations using adversarial abstraction (Section 4.1) is straightforward: we adapted GPUVerify to determine statically those arrays that may be manipulated atomically and force adversarial abstraction of these arrays. We used the existing encoding of read and write sets, described in [3], to add sets recording atomic accesses, and implemented atomic-aware race checks as described in Section 4.1.

To support the refined atomic abstraction of Section 4.2 we made GPUVerify aware of the repetition-free atomic operations `atomic_inc` and `atomic_dec`, and implemented an analysis that determines whether an array is only ever accessed using a single repetition-free atomic operation; we say that such an array is *repetition-free*. A call to `atomic_add` or `atomic_sub` with a positive numeric argument is regarded as consisting of a series of increments or decrements respectively.

For each repetition-free array A we introduce in the Boogie program generated by GPUVerify a map $\mathsf{used}_A : \mathsf{Int} \times \mathsf{Int} \rightarrow \mathsf{Bool}$. If $\mathsf{used}_A(x, v)$ holds, this indicates that offset x of A has previously yielded the value v, and thus will not yield v when accessed again using the repetition-free operation. When translating an atomic operation on repetition-free array A in the context of threads s and t, suppose that the threads access array offsets o_s and o_t and store the operation results into private variables z_s and z_t, respectively. We generate the following sequence of Boogie statements (presented here using mathematical syntax) to model the atomic operation:

$\mathcal{A}_A := \mathcal{A}_A \cup \{o_s\}$;	Log the atomic access made by thread s
assert $o_t \notin \mathcal{R}_A \cup \mathcal{W}_A$;	Ensure the atomic access made by thread t does not race
havoc z_s, z_t;	The threads receive values that are arbitrary, except:
assume $\neg\mathsf{used}_A(o_s, z_s)$;	neither value has been used
assume $\neg\mathsf{used}_A(o_t, z_t)$;	previously at this offset, and
assume $z_s \neq z_t$	the threads receive different values
$\mathsf{used}_A(o_s, z_s) := \mathsf{true}$;	The values are now marked as used up
$\mathsf{used}_A(o_t, z_t) := \mathsf{true}$;	

Thus, o_s and o_t are guaranteed unique, and subsequent use of them to index into some array will be correctly found race-free.

In Section 5 we evaluate the overhead in terms of verification time of using this refined abstraction over regular adversarial abstraction.

5 Experimental Evaluation

To evaluate our implementation of warp and atomic support in the GPUVerify tool [3] we considered the following benchmark suites:

- CUDA 5.0 SDK benchmarks (171 CUDA kernels)
- CUDA 2.0 SDK benchmarks (8 CUDA kernels do not appear in the 5.0 SDK)
- C++ AMP samples, translated into CUDA, from [3] (20 CUDA kernels)
- AMD APP SDK (78 OpenCL kernels)
- ParBoil benchmarks (25 OpenCL kernels)
- SHOC benchmarks (87 OpenCL kernels)

Of the 199 CUDA and 190 OpenCL kernels, 6 and 16 use atomic operations, respectively. The benchmarks and our tool chain, with instructions on how to re-run our experiments, are available online.[4]

Experiments were performed on a PC with a 3.4GHz Intel i7-2600 and 16GB RAM running Ubuntu 13.04, using GPUVerify revision 988 (2013-11-25), and Z3 4.3.1. A time limit of 900 seconds (15 minutes) per kernel was used for analysis.

Impact of warp-level synchronization We ran GPUVerify with warp-level synchronization enabled (warp size 32) across the 199 CUDA kernels. We found 7 cases where verification succeeded with warp-level synchronization enabled but failed without. GPUVerify is thus able to provide precise results for these kernels where before it would report false positives. We were surprised to find one case (`dwtHaar1D`) where verification succeeded with the two-pass method but failed with re-sync. In this case re-sync requires a loop invariant that makes reference to whether the threads under consideration are in the same warp, which GPUVerify does not infer. With the two-pass method the intra-warp case is trivial to verify, and a simpler loop invariant which *is* inferred suffices for the inter-warp case.

Figure 4 compares verification times across the CUDA benchmarks with respect to the re-sync and two-pass methods. A point at coordinates (x, y) represents a benchmark for which analysis (successful verification, or the report of a failed proof attempt) took x seconds using the re-sync method and y seconds using the two-pass method. The figure shows that the two-pass method is faster in many cases, sometimes dramatically. We attribute this to the fact that the two-pass method involves solving two simpler verification problems which are solved in parallel. We also compared verification time using the re-sync method to verification time without warp-level synchronization, for the CUDA kernels where the verification result was not affected by warp-awareness. We observed some fluctuation in verification times between examples, but overall the performance difference was negligible: verification using the re-sync method was 1.043 times slower than with verification without support for warps. Thus warp-awareness does *not* compromise verification speed.

Impact of support for atomic operations Prior to this work, GPUVerify (nor any other *verification* tool for GPU kernels) was applicable to the 22 kernels in our suite that use atomic operations. Using GPUVerify we found two bugs in these kernels.

In a CUDA 5.0 SDK Mandelbrot kernel, where atomic operations are used for work distribution in a manner similar to the example of Figure 3, we found a read-write data race arising due to a missing barrier. The race was not due to misuse of atomics, but the kernel was not amenable to analysis prior atomic support. We reported this race to engineers at NVIDIA who confirmed and subsequently fixed the issue.

[4] `http://multicore.doc.ic.ac.uk/tools/GPUVerify/NFM2014`

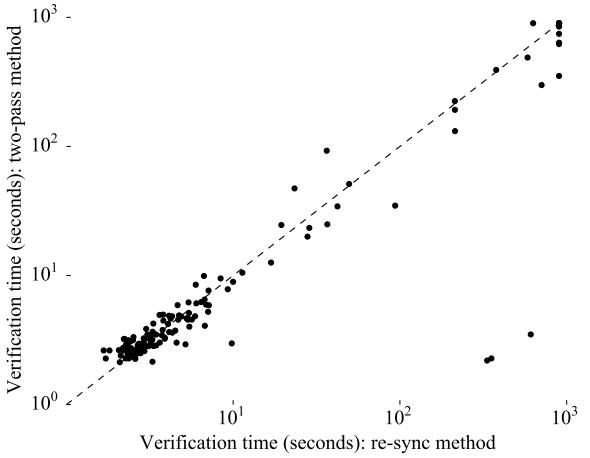

Fig. 4: Verification times for two-pass vs. re-sync methods over 199 CUDA kernels

We discovered an atomic/non-atomic race in a sophisticated histogram implementation kernel in the ParBoil suite (`tpacf/gen_hists`). In this example, work groups share histogram buckets in group-shared memory. Threads first initialize this memory to zero, then repeatedly update histogram buckets atomically. No barrier was issued between bucket initialization and bucket update, leading to races between these phases. This race was confirmed by the maintainers of the Parboil suite.

These bugs *cannot* be found directly using GKLEE, a bug-finding tool for CUDA kernels [14] that has been extended with support for atomics [4]. This is because the kernels manipulate floating point data which GKLEE does not support. Floating point operators are approximated by GPUVerify through the use of uninterpreted functions [3].

We also found what is strictly a read/atomic race in the `histo/histo_main` ParBoil example. A non-atomic read is used to retrieve the value of a histogram bucket before an atomic update is applied. We do not regard this as a programmer error: we believe the intention is that the read should be an atomic read operation, which OpenCL 1.2 does not directly provide. However, atomic read *is* provided by the recently announced OpenCL 2.0, so the kernel should be re-written accordingly in due course.

After fixing these bugs, we were able to verify race-freedom for 15 of the 22 kernel that use atomics. In 13 cases verification was fully automatic: GPUVerify was able to automatically generate loop invariants required to prove race-freedom. In 2 cases it was necessary to provide loop invariant annotations for verification to succeed. These invariants were unrelated to the use of atomics – they were necessary to capture disjointness of the data access patterns associated with non-atomic arrays. The invariants are available in our online set of benchmarks.

Of the 7 kernels for which verification failed, 4 were kernels used in the implementation of breadth-first-search graph algorithms in the ParBoil and SHOC suites. These kernels are only correct with respect to non-trivial, quantified preconditions on input

arrays, beyond the limited support for precondition annotations currently provided by GPUVerify, thus the tool reports a write-atomic race for each of these kernels.

Two of the CUDA 5.0 Mandelbrot kernels use an atomic counter to compute disjoint indices into shared arrays as discussed in Section 4.2. However, in each thread block only the "master" thread, thread 0, is responsible for updating the global index counter, obtaining a base index used by all threads in the block. In this setting the two-thread reduction does not allow a proof of race-freedom for a pair of non-master threads s and t in different thread blocks. Even with our refined atomic abstraction, in the absence of concrete knowledge about master thread behavior, s and t cannot deduce that their base indices are distinct. In future work we plan to solve this issue by extending the two-thread reduction to allow specific threads, such as master threads, to be concretely represented. To evaluate our refined atomic abstraction we created a simplified Mandelbrot fractal generator, capturing all the behavior of the more complex of the two Mandelbrot examples, but simplified so that each thread directly computes its array indices from a global counter, eliminating the role of a master thread. After this simplification, we were able to verify the example using the refined abstraction of Section 4.2.

The final kernel using atomics that we could not verify is the `histo/histo_main` kernel discussed above: after we fixed the read/atomic race, GPUVerify reported possible races on other, non-atomic arrays; we have yet to find strong enough loop invariants to eliminate these false positives.

6 Related Work

Several recent works have focused on GPU kernel verification using SMT solving [13,3], combined static and dynamic analysis [12] and separation logic with permissions [8]. The closest work to GPUVerify is the PUG technique and tool [13], and the methods have been compared qualitatively and experimentally [3]. To our knowledge, ours is the first work to present support for either warps or atomics in a verification technique.

Dynamic symbolic execution is used by the GKLEE [14] and KLEE-CL [5] tools to find bugs in CUDA and OpenCL kernels, respectively. The GKLEE tool accurately models warp-based execution and thus can find bugs in CUDA kernels precisely, without reporting false positive data races that are impossible due to warp scheduling constraints. An extension to the GKLEE tool considers analysis of CUDA kernels that use atomic operations [4]. On discovering a potential conflict involving atomic accesses, thread schedules are enumerated to try to find a concrete counterexample to correctness. Delay bounding [7] is used to limit schedule explosion. This method has proven effective in finding bugs, but cannot be used to verify *absence* of defects. As noted in Section 5, application of GKLEE is limited due to lack of support for floating point operations. A proposal for extending the KLEE-CL method with support for atomics, via a symbolic encoding of schedules, is proposed as future work in [6], but has not been implemented.

The two-thread reduction employed by GPUVerify is also used in other methods for GPU kernel analysis [13,15], and that several methods exploit the fact that race analysis can be performed with respect to a single thread schedule [13,14,5].

7 Conclusions and Future Work

We have presented methods for extending a GPU kernel verification technique with support for two additional inter-thread communication mechanisms: warps and atomics. Our experimental evaluation shows that these extensions, implemented in the GPUVerify tool, allow a larger set of kernels to be successfully analyzed.

Our main direction for future work will be the investigation of more sophisticated abstractions for reasoning about atomic operations: extending the two-thread reduction so that manipulation of atomic variables by master threads can be precisely handled, as discussed in Section 5, and designing custom abstractions to capture further design patterns associated with the use of atomics. We also plan to investigate the use of verification methods for kernel optimization. For example, warp divergence [14], where threads in the same warp simulate different control flow paths through predicated execution, is often regarded as a performance bug. By combining support for reasoning about warps with prior work on barrier divergence [3], we can investigate the use of verification to prove absence of warp divergence in complex kernels.

References

1. AMD, Inc.: AMD graphics cores next (GCN) architecture, white paper (2012)
2. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO (2005)
3. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: OOPSLA (2012)
4. Chiang, W.F., Gopalakrishnan, G., Li, G., Rakamarić, Z.: Formal analysis of GPU programs with atomics via conflict-directed delay-bounding. In: NFM (2013)
5. Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic testing of OpenCL code. In: HVC (2012)
6. Collingbourne, P.C.: Symbolic Crosschecking of Data-Parallel Floating Point Code. Ph.D. thesis, Imperial College London (2012)
7. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: POPL (2011)
8. Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs using permission-based separation logic. In: BYTECODE (2013)
9. Khronos Group: The OpenCL extension specification, version 2.0 (2013)
10. Khronos Group: The OpenCL specification, version 2.0 (2013)
11. Leino, K., Rustan, M.: This is Boogie 2 (2008), manuscript KRML 178 (2008)
12. Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., Lerner, S.: Verifying GPU kernels by test amplification. In: PLDI (2012)
13. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: FSE (2010)
14. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: PPoPP. ACM (2012)
15. Li, P., Li, G., Gopalakrishnan, G.: Parametric flows: automated behavior equivalencing for symbolic analysis of races in CUDA programs. In: SC (2012)
16. NVIDIA Corporation: CUDA C programming guide (2013), version 5.5
17. Sengupta, S., Harris, M., Garland, M.: Efficient parallel scan algorithms for GPUs. Tech. Rep. NVR-2008-003, NVIDIA (2008)
18. Stratton, J.A. *et al.*: Parboil: A revised benchmark suite for scientific and commercial throughput computing. Tech. Rep. IMPACT-12-01, UIUC (2012)
19. Collingbourne, P., Donaldson, A.F., Ketema, J., Qadeer, S.: Interleaving and lock-step semantics for analysis and verification of GPU kernels. In: ESOP (2013)