Engineering a Static Verification Tool for GPU Kernels

Ethel Bardsley[1], Adam Betts[1], Nathan Chong[1], Peter Collingbourne[2], Pantazis Deligiannis[1], Alastair F. Donaldson[1], Jeroen Ketema[1], Daniel Liew[1], Shaz Qadeer[3]

[1]Imperial College London, [2]Google[⋆], [3]Microsoft Research

Abstract. We report on practical experiences over the last 2.5 years related to the engineering of GPUVerify, a static verification tool for OpenCL and CUDA GPU kernels, plotting the progress of GPUVerify from a prototype to a fully functional and relatively efficient analysis tool. Our hope is that this experience report will serve the verification community by helping to inform future tooling efforts.

1 Introduction

Graphics processing units (GPUs) are now a mainstay technology with which to accelerate computationally intensive applications. The OpenCL [25] and CUDA [33] programming models allow general-purpose computations to be offloaded to run on a variety of GPU platforms. In these programming models, a computation to run on the GPU is described using a *kernel* function, a template describing the behaviour of a single thread. Threads are organized into a set of groups, threads in the same group can synchronize with each other using *barriers*, and each thread has a unique id which it can use to access distinct data and follow distinct control paths from other threads.

A challenge in GPU programming is to avoid *data races*, where distinct threads access a common memory location, at least one access is a write, and there is no intervening barrier synchronization. Data races tend to arise due to a combination of intricate data access patterns necessary to achieve high memory performance, which can be hard to write correctly, and the desire to minimize expensive barrier synchronization operations, also to maximize performance. Data races lead to nondeterministically occurring bugs that can be hard to diagnose and fix, and since performance is the *sole* motivation for GPU offloading, race-prone programming styles are not likely to go away.

In response to the GPU programming paradigm and the problem of data races, a variety of formal and semi-formal methods for finding, or proving absence, of defects in GPU kernels have been proposed [27,8,26,23,28,14,29,10,4,12,5]. Over the last 2.5 years, our contribution to this area has been GPUVerify,[1] an open source tool for static verification of race-freedom for OpenCL and CUDA kernels.

Our research papers [8,15,11] have presented the top-level ideas that underpin the GPUVerify approach, and focus on arguing soundness of the approach with respect to an operational semantics for a core GPU kernel programming language. Embedding these ideas in a tool that can be applied directly to the source code of real-world examples with a reasonable degree of efficiency and automation has required a significant

[⋆] Peter Collingbourne was at Imperial College London when he contributed to this work.

[1] http://multicore.doc.ic.ac.uk/tools/GPUVerify

optimization effort and a number of important engineering decisions. This has been guided by a growing set of GPU kernel benchmarks which now counts 564 examples. In this tool paper we aim to communicate this engineering experience and insight, not reflected in the aforementioned research papers, to the verification community in the hope that it will be of general interest and may help inform future tooling efforts.

We provide an overview of GPUVerify (Sect. 2) and describe how we have evolved the front-end capabilities of the tool in order to handle a large set of benchmarks (Sect. 3). We then describe and evaluate several methods for improving the verification performance of the tool (Sect. 4). Our aim is for GPUVerify to be useful to industry, motivating steps for minimizing false positives and presenting clear error messages, which we describe (Sect. 5); we also discuss steps taken to ease uptake of the tool by industrial partners (Sect. 6). We conclude with a summary of lessons learned and the identification of *invariant generation* as a key challenge for future work (Sect. 7).

Related work A number of other works on GPU kernel analysis have appeared recently and can be categorized into methods for verification [27,26,23,12,4] and bug-finding via symbolic execution [28,14,29,10]. Our research papers provide a detailed discussion of how these works relate to GPUVerify, including experimental comparisons [8,11]. We do *not* compare GPUVerify with related tools here: the aim of this work is not to promote GPUVerify as "the colonel of kernel verification tools," but rather to communicate the insights into verification tool development that have emerged from the project.

2 Overview of the GPUVerify Technique

The key idea behind GPUVerify is that a massively parallel GPU kernel can be proven free from data races by deriving a *sequential* program from the kernel and verifying that this program is free from assertion failures. This avoids reasoning about thread interleavings and allows existing verification techniques for sequential programs to be reused. If verification of the sequential program fails, the proof failure may shed light on a defect in the original kernel, if one exists. Alternatively, the failure may be a false positive arising due to the abstractions employed while constructing the sequential program, or due to limitations of the method used to verify the sequential program; in practice, the main limitation relates to loop invariant generation.

The method GPUVerify uses to transform a kernel into a sequential program is presented in detail in [8,15]. The transformation proceeds in three steps. First, thread id-sensitive control flow is eliminated by *predicating* each statement [1]. For example, the if-statement shown to the right will be turned into the code on the far right, where a statement of the form $p \Rightarrow s$ is a predicated statement which behaves as a no-op if p is *false*, and has the same effect as s if p is *true*. Predication is semantics-preserving and ensures that every thread follows the same control path through the kernel.

```
if (tid > 0) s1;  |   (tid > 0) ⇒ s1;
else s2;          |  !(tid > 0) ⇒ s2;
```

Second, each access to a shared memory array is instrumented to allow data races to be detected. For each thread t and each array A two sets are introduced: one to track reads from and one to track writes to A by t. Upon an access, the offset at which the access occurs is recorded in the appropriate set, and occurrence of a data race is checked by considering overlap between relevant sets.

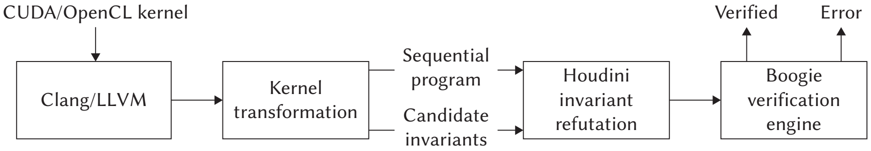


Fig. 1. The GPUVerify architecture, which draws on the Clang/LLVM and Boogie frameworks

The third and final step applies a *two-thread reduction*. This is an abstraction that removes all but two *arbitrary* threads and then combines the two threads into a single sequential program by applying a round-robin schedule. The effects of additional unmodeled threads on the shared state are over-approximated using abstraction. That the reduction is sound is explained in detail in [8,15], and hinges on the observation that if barriers are the only mechanism for synchronization then a race-free kernel behaves *deterministically* when applied to a given input; thus, as long as data races are detected, analysis can focus on the single round-robin schedule. The method is incomplete due to the shared state abstraction which over-approximates the effects of additional threads, and may include error-inducing behaviours that are infeasible during concrete execution. The two-thread reduction has been used in other works on GPU kernel analysis [27], and the idea of reducing verification complexity through pairwise reasoning is well-known and has been employed, for example, in model checking of cache coherence protocols [13,30,39].

Architecture The architecture of GPUVerify is depicted in Fig. 1, and leverages the mature and widely used Clang/LLVM[2] and Boogie [6] tool chains. Clang/LLVM is used to parse CUDA and OpenCL kernels and lower them into LLVM intermediate representation (IR). This removes all complex syntactic features (including C++ templates), yielding a simple representation of a kernel. The kernel transformation process first invokes Bugle, our custom-built LLVM IR-to-Boogie translator, to translate the obtained LLVM IR into a Boogie program, giving a Boogie representation of the kernel. Predication, race instrumentation and two-thread reduction are then applied, as outlined above, to yield the sequential Boogie program to be verified. Kernel transformation also speculates candidate loop invariants based on a number of custom-designed templates, which attempt to capture data access idioms we observed in many kernels.

After kernel transformation, GPUVerify uses the Houdini algorithm [20] (implemented as part of the Boogie framework) to compute the largest conjunctive invariant over the set of speculated loop invariants, discarding any candidate invariants that cannot be proven.[3] The sequential program and synthesized invariant are then passed to the Boogie verification engine for the actual verification. Boogie in turn invokes an SMT solver: the Z3 solver is the default [32], and we have added support for the CVC4 solver [7], as discussed further in Sect. 4. The result of this stage is either successful verification of the sequential program, which implies race-freedom of the original kernel, or an error indicating that the original kernel may exhibit a defect.

[2] http://llvm.org/

[3] That we obtain the *largest* conjunction is a property of the Houdini algorithm [20].

3 Applying GPUVerify to a Large Set of Benchmarks

We have applied GPUVerify to 564 kernels gathered from nine sources:

- *AMD Accelerated Parallel Processing SDK* v2.6 [2] (78 OpenCL kernels)
- *NVIDIA GPU Computing SDK* v5.0 [34] (166 CUDA kernels); we also include a further 8 CUDA kernels from a previous version of the SDK (v2.0)
- Microsoft *C++ AMP Sample Projects* [31] (20 kernels, hand translated to CUDA)
- The *gpgpu-sim* benchmarks [3] (33 CUDA kernels)
- The *Parboil* benchmarks v2.5 [38] (25 OpenCL kernels)
- The *Rodinia* benchmark suite v2.4 [9] (36 OpenCL kernels)
- The *SHOC* benchmark suite [16] (87 OpenCL kernels)
- The *PolyBench/GPU* benchmark suite [21] (49 OpenCL kernels)
- Rightware *Basemark CL* v1.1 [37] (62 OpenCL kernels)

Each suite is publicly available except for *Basemark CL* which was provided to us under an academic license. This collection covers all the publicly available GPU benchmark suites that we are aware of. The kernel counts above do not include 41 kernels that we manually removed from our study: (i) 16 kernels are trivially race-free as they are run by a single thread, (ii) 8 kernels use features that are currently unsupported by GPUVerify, such as CUDA *surfaces*, and (iii) 17 kernels require refinements of the GPUVerify verification method that cannot currently be applied automatically [11].

Our default assumption is that these benchmarks are free from defects, thus our aim is verification. However, in the process of applying GPUVerify we have identified, reported and fixed several data race bugs. At the time of writing, running with full optimizations on our experimental platform (described in Sect. 4), and with a timeout of 10 minutes per benchmark, GPUVerify can verify 422 kernels and reports possible defects for 115. We know that some of these failures are (and expect most to be) false positives that demand improved invariant inference, but some may correspond to further bugs that we have not yet identified. The timeout is reached in 27 cases.

We now explain how we have managed the evolution of GPUVerify's front-end capabilities from a simple prototype applied to hand-crafted examples to a tool with wide applicability to GPU kernel benchmarks. In Sect. 4 we discuss engineering decisions related to the performance of verification.

Incremental front-end support The starting point for GPUVerify was the idea of using sequential program verification technology to analyse GPU kernels, but it took several iterations to arrive at the method described in Sect. 2. To allow us to experiment with example kernels while our ideas were in flux, we first devised a manual process for translating GPU kernels into Boogie. Starting with Boogie mitigated the risk of investing in a CUDA or OpenCL front-end and subsequently discovering that our ideas would not be practical. Using our Boogie-based GPU kernel language we implemented a prototype of the kernel transformation step from Fig. 1 and manually encoded a number of kernels into Boogie to evaluate the prototype. After encoding around 20 examples it became clear that our technique had promise, but that we would need to invest in a front-end for OpenCL and/or CUDA to study a larger set of examples.

We first designed a translator that mapped OpenCL and CUDA kernels (subject to various restrictions) into our Boogie kernel language. This translator used Clang to parse kernels, and performed translation at the level of the Clang abstract syntax tree (AST). The structured nature of the Clang AST allowed for a relatively simple transformation into structured Boogie. This was essential as we did not know how to apply predication (a key part of our method, see Sect. 2) to unstructured programs. We were able to process a fairly large set of benchmarks using this front-end, facilitating our first publication on GPUVerify [8] which presents an evaluation using 163 kernels. However, the "structured" limitation eliminated kernels exhibiting unstructured control-flow (arising e.g. from switch statements and short-circuit evaluation of Boolean operators). Working at the Clang AST level also meant that we had to directly deal with syntactic features ranging from the difference between `while` and `for` loops (easy but annoying), through details of struct accesses (medium difficulty), to handling of C++ templates arising in CUDA code (fiendishly difficult, and not attempted).

We realized that to apply the tool widely it would be beneficial to work at the level of LLVM intermediate representation (IR), by which point complex syntactic features have been desugared. Because LLVM IR is unstructured, we focused on solving the problem of applying predication to unstructured control-flow-graphs [15] and implemented Bugle, our custom LLVM-to-Boogie translator (see Sect. 2), which produces unstructured Boogie code to which the new predication method can be applied. We considered leveraging an existing LLVM-to-Boogie translator, SMACK [35], but opted to build a custom translator that could take direct advantage of the relatively simple nature of the GPU programming model.

Environment modeling for OpenCL and CUDA Significant further engineering effort was required to model built-in functions provided by OpenCL and CUDA. For OpenCL and CUDA, respectively, this included 164 and 231 built-in math functions and 136 and 30 atomic operations. For OpenCL we benefited from `libclc`,[4] an open source OpenCL library implementation. In addition, we have equipped GPUVerify with support for OpenCL image types, CUDA textures and an abstraction of a widely used CUDA random number generation library. Our environment modeling is not complete (e.g. we do not yet support CUDA *surfaces*) and it is a moving target as OpenCL and CUDA continue to evolve. Nevertheless, our modeling effort so far allows GPUVerify to process many practical examples.

4 Engineering Issues for Efficient Verification

Our first implementation of GPUVerify worked for small examples, but did not perform well on more complex kernels involving multiple loops and many shared memory accesses. We now describe the steps we have taken to improve verification performance through efficient memory modeling, uniformity analysis to reduce the need for predication, supporting multiple SMT solvers, and optimizations to produce formulas that can be efficiently processed by SMT solvers.

Experimental setup Throughout this section we report experimental results over the 564 kernels in our benchmark collection (see Sect. 3). All experiments were conducted

[4] http://libclc.llvm.org/

on a compute cluster using nodes with Intel Xeon EP-2620 cores at 2GHz with 16GB RAM running RedHat Linux 6.3, using Z3 v4.3.1, CVC4 v1.4-prerelease from 29-01-2014 and Clang/LLVM v3.4. Times reported are averages over three runs.

We use *baseline* to refer to GPUVerify equipped with the efficient memory model and uniformity analysis described below, but without any of the additional optimizations we go on to discuss. We use the *responsive* set to refer to the 492 kernels for which verification completes (in 391 with "success", in 101 with "possible defect") for *baseline* and all more highly optimized configurations. We report speedup results with respect to the *responsive* set. We do not further discuss 12 kernels for which GPUVerify reached the timeout with every optimization configuration. In 60 cases, GPUVerify reached the timeout for some optimization configurations but not others. We do not include these cases when discussing speedups afforded by optimizations.

Our tool chain, non-commercial benchmarks and experimental data (in the form of interactive graphs) are available online.[5]

Modeling memory The C language rules for pointer casting apply to CUDA and OpenCL, meaning that it is legitimate to cast an expression e of type $\mathsf{T}*$, where T is some type, to an expression of type $\mathsf{char}*$, after which offsets from e can be addressed with byte-level granularity. An example of casting in practical GPU code appears in a histogram kernel shipped with the CUDA SDK. The kernel works on an array of char data and starts by initializing the array to be uniformly zero. During initialization the array is cast from $\mathsf{char}*$ to $\mathsf{int}*$ to allow zero-initialization to be performed word-by-word, which is more efficient than working byte-by-byte.

For GPUVerify to work "out-of-the-box" we must handle this kind of pointer usage even though it is uncommon. We initially let our Bugle front-end model memory at byte-level granularity. To illustrate this, consider an array A with elements of type short, and a write instruction $A[i] = x$. In the Boogie code generated by Bugle with byte-level memory modeling, A is declared as a map from 32-bit offsets to bytes, and the single write is translated into two byte-level writes ('bv' stands for bitvector):

```
var A : [bv32]bv8;      // Map from addresses to bytes
A[i*2+0] := x[8:0];     // The write to A is modeled by two byte-level writes.
A[i*2+1] := x[16:8];    // We use *, + and integer literals for brevity.
```

This representation is problematic, especially for data types with large widths such as double: it leads to many loads and stores that need to be instrumented when performing race analysis and complicates the loop invariants necessary to prove race-freedom. Both issues place significant demands on the underlying SMT solver. In practice we found that verification with this simple memory model was unacceptably slow.

To overcome this problem we have developed a unification algorithm that conservatively determines whether an array A with element type T may ever be accessed at a granularity that is not a multiple of $\mathsf{sizeof}(\mathsf{T})$. If such an access may be possible, A is modeled with byte-level granularity as described above. Otherwise A is modeled with "type-level" granularity as a map from addresses to bitvectors of size $8 * \mathsf{sizeof}(\mathsf{T})$: accessing an element of A leads to a single read or write in the generated Boogie code.

With this analysis, byte-level modeling is avoided in all but 40 of our 564 kernels and in 39 of these cases at least one array is still modeled with type-level granularity.

[5] http://multicore.doc.ic.ac.uk/tools/GPUVerify/CAV2014

We evaluated the impact of byte- vs. type-level modeling using the 365 kernels in our collection for which *baseline* GPUVerify responds within 60 seconds. Turning off the memory analysis described above, forcing byte-level memory modeling everywhere, we find that 31 kernels reach the 10 minute timeout and 12 kernels flip from verifying to failing (due to more complex invariants that can be necessary when reasoning at the byte level). Overall, analysis took 6.6× longer, but this is not a fully fair comparison because (a) the 31 kernels that timeout might in practice take much longer to verify, and (b) comparing times for a kernel where the verification result differs has limited meaning. Nevertheless, the slow-down associated with byte-level modeling indicates that our memory analysis is necessary in making GPUVerify practically useful.

Our experience supports existing evidence that, in the context of verification, modeling memory at the lowest common denominator level of bytes does not scale [36].

Uniformity analysis The kernel transformation performed by GPUVerify involves *predicating* a kernel and applying the *two-thread reduction* (see Sect. 2). Predication is essential for handling fragments of a kernel where threads might take different control flow paths, and because distinct threads may operate on distinct data the two-thread reduction must in general introduce a pair of variables for each private variable appearing in a kernel, one copy for each thread being modeled.

We have observed that in practical kernels, some or all control flow is often *uniform* across threads: the guards of conditional and loop statements do not depend (directly or indirectly) on thread ids. In fact, to achieve high performance when writing code for mainstream GPUs it is important to *minimize* thread divergence, and have threads follow the same control flow path whenever possible [22]. We found it often necessary to provide loop invariants to *recover* uniformity between the two threads under consideration, by asserting equality between predicates guarding execution and between id-insensitive private variables.

To avoid the overhead of generating such invariants and the duplication of private variables that are guaranteed to be uniform, we have designed and implemented a *uniformity analysis*. This is a taint analysis working at the control-flow graph level that uses the program dependence graph [18] to determine which variables and basic blocks are *non-uniform* because they are (transitively) control- or data-dependent on the thread ids. The analysis initially sets every variable and block to be *uniform*, except the `tid` variable which is *non-uniform*. Uniformity information is then updated repeatedly until a fixpoint is reached: a variable becomes *non-uniform* if it is assigned an expression that contains a non-uniform variable, or if it is updated inside a non-uniform block; a block becomes non-uniform if it is found to be (transitively) control-dependent on a condition that contains a non-uniform variable. Predication need only be applied to non-uniform blocks, and only non-uniform private variables need to be duplicated when the two-thread reduction is applied. This reduces the burden of loop invariant generation and leads to smaller SMT formulas due to the reduction in private variables.

To illustrate uniformity analysis, consider the example code snippet of Fig. 2(a), contrived for purposes of illustration, where private variables `x`, `y` and `z` are assumed to be initially uniform between threads. Fig. 2(b) shows the result of applying predication to the kernel according to the scheme discussed in Sect. 2, and then duplicating the statements according to the two-thread reduction so that each of the two threads has

```
if (x > 0) {
  if (tid < x)
    y++;
  z += y;
  x /= 2;
}
```

(a)

```
(x1 > 0 ∧ tid1 < x1) ⇒ y1++;
(x2 > 0 ∧ tid2 < x2) ⇒ y2++;
(x1 > 0) ⇒ z1 += y1;
(x2 > 0) ⇒ z2 += y2;
(x1 > 0) ⇒ x1 /= 2;
(x2 > 0) ⇒ x2 /= 2;
```

(b)

```
if (x > 0) {
 (tid1 < x) ⇒ y1++;
 (tid2 < x) ⇒ y2++;
 z1 += y1;
 z2 += y2;
 x /= 2;
}
```

(c)

Fig. 2. Uniformity analysis: (a) original code with the outer conditional uniform and the inner conditional non-uniform; (b) after predication and two-threaded duplication, without uniformity analysis; (c) after predication and two-threaded duplication, with uniformity analysis.

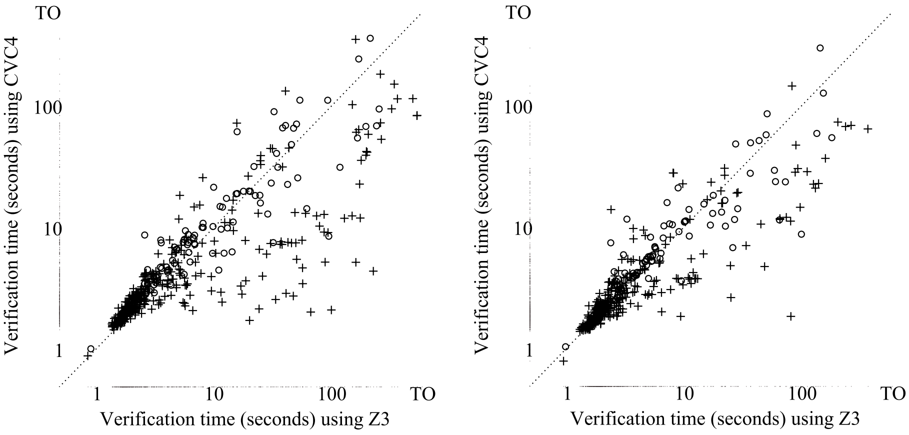
its own copy v_i of each private variable v ($i \in \{1, 2\}$) and every statement is executed separately by each thread. Uniformity analysis determines that the condition `tid < x` is non-uniform, because it refers to `tid`. As a result, `y` is non-uniform because the statement `y++` is control-dependent on the condition `tid < x`. Because `y` is non-uniform, `z` is also deemed non-uniform because it is updated by the statement `z += y` which involves `y` on the right-hand side. The variable `x` and thus the condition `x < 0` remain uniform. With the results of uniformity analysis, GPUVerify is free to perform less aggressive predication and duplication of the kernel, illustrated by Fig. 2(c). Only the inner conditional is predicated, the private variable `x` is not duplicated, and there is thus only one assignment to `x`.

As in the byte-level modeling experiment described above, we evaluated the impact of uniformity analysis using the 365 kernels in the *responsive* set for which *baseline* GPUVerify responds within 60 seconds. Turning off uniformity analysis, 6 kernels reach the 10 minute timeout and 33 flip from verifying to failing; the latter is due to the lack of loop invariants required to recover uniformity when the analysis is disabled. Overall, analysis took 1.9× longer with uniformity analysis disabled, but this is not a fully fair comparison for the same reasons as in the byte-level modeling experiment.

A similar analysis has been proposed for optimizing OpenCL kernels for CPU (rather than GPU) performance [24]. The analyses were developed independently.

Support for CVC4 Boogie uses the Z3 solver [32] by default. We added support to Boogie for CVC4 [7], which also provides the theories used by GPUVerify (bitvectors, arrays, and uninterpreted functions). Our main motivation here was CVC4's permissive license: shipping GPUVerify with CVC4 in place of Z3 would make it easier for industrial users to try the tool (see Sect. 6). Two by-products of this effort are that we found and reported several bugs in CVC4 which were promptly fixed, and that our CVC4 support has been committed to Boogie, making CVC4 available to other Boogie users. It is also useful for us to have two solvers available for evaluation, to help determine when poor performance on a kernel is due to a solver quirk vs. a fundamental issue.

Fig. 3 presents log scale scatter plots comparing the performance of GPUVerify using CVC4 vs. Z3 with (a) baseline (no optimizations) and (b) all optimizations described below enabled, over the *responsive* benchmarks. A point (x, y) corresponds to a kernel for which end-to-end verification took x and y seconds, using Z3 and CVC4 respectively. Points above/below the diagonal correspond to kernels where Z3 performed better/worse than CVC4. We distinguish between kernels for which verification suc-



(a) Baseline

(b) All optimizations

Fig. 3. Scatter plots comparing the performance of Z3 and CVC4 over the *responsive* benchmarks. A symbol $+$, respectively $\circ$, represents a kernel for which verification succeeds, respectively fails.

ceeds ($+$) and fails ($\circ$). The total time for analysis using Z3 and CVC4 with all optimizations was 5774 seconds and 3448 seconds respectively, indicating that CVC4 was 1.7× faster than Z3 over the *responsive* set. Going beyond this set and with optimizations enabled, verification for 15 kernels timed out using Z3 but completed using CVC4, and the converse was true for 8 kernels.

Optimizing for verification performance We now describe four methods for optimizing the Boogie programs generated by GPUVerify so that they lead to SMT formulas that are easier to decide. The optimizations preserve the result of verification (precisely the same assertions can fail).[6] Each optimization was motivated by one or more challenging examples, for which the optimization led to an encouraging speedup. We evaluate the optimizations experimentally across our benchmark suite using Z3 and CVC4, and comment on the performance that could be gained through portfolio verification.

Eliminating redundant read instrumentation The first optimization is extremely simple: when we can deduce statically that an array is never written to, we do not perform race analysis for the array. This is a common situation, as GPU kernels often read data from one or more input arrays, and write results to separate output arrays. The optimization may seem trivial, but we mention it because we only considered it after two years' work on GPUVerify. Our efforts and attention were focused on more sophisticated challenges, and this "low hanging fruit" escaped our attention. Our results below show that the optimization is effective overall, serving as a reminder that, when optimizing a program analysis method, it is worth exploring easy optimization avenues first.

Optimizing within barrier intervals The idea of redundant read instrumentation led us to devise a refinement of this optimization. Define a *barrier interval* to be a call-free,

[6] An exception to this is that in some cases the "redundant reads" optimization actually aids in verification, because shared state abstraction is unnecessary for read-only arrays.

single-entry single-exit region of a control flow graph which starts and ends with a barrier [28]. If shared array A is never written to in a barrier interval $\mathcal{I}$ then there is no need to check for races on A during $\mathcal{I}$: in the absence of writes, there is no possibility for races between instructions in $\mathcal{I}$, and the barriers guarding entry to and exit from $\mathcal{I}$ eliminate the possibility of races between reads inside and writes outside $\mathcal{I}$.[7]

Private array removal Vector data types and operations are widely used in GPU code. In LLVM IR, thread-private vector data is represented as residing in *memory*, rather than in virtual registers. Our Bugle front-end translates each private memory region in LLVM IR into a separate Boogie map, and vector element access are represented at the Boogie level via indexing operations into these maps. Because a vector has a fixed number of elements (e.g. x, y, z and w for a 4D vector), the map indexing expressions are always taken from a small set of literal values. We implemented a pass in GPUVerify which identifies when a map is indexed exclusively using a set of k distinct literals. In such cases, the map and associated indexing expressions are replaced by k distinct scalar variables, each representing an element of the original map. This reduces the extent to which array reasoning is required; our hypothesis was that this would improve solver performance.

Watchdog race checking Recall from Sect. 2 that data race detection is performed by introducing sets containing the offsets of array accesses. In practice, such sets can be modeled via their characteristic functions using maps. However, this requires quantifiers to express invariants relating to the contents of sets, such as emptiness.

To avoid quantifiers and the associated theorem proving burden, we originally devised a non-deterministic representation of sets [8], based on [17]. Let s and t denote the arbitrary threads considered by the two-thread reduction. For an array A, we introduce variables allowing at most one read from and at most one write to A to be tracked. We then instrument each read operation issued by thread s with a nondeterministic choice between updating the instrumentation variables to record the offset that was read from, or leaving the instrumentation variables untouched. Write operations are instrumented similarly. On kernel entry, and at each barrier, the instrumentation variables are set to indicate that no accesses are tracked. Races between s and t are detected by checking whether offsets accessed by thread t conflict with the offsets tracked by the instrumentation variables. The nondeterministic encoding is sound for race detection because proving correctness of the sequential program generated by GPUVerify involves showing that a conflict between threads on an array is impossible for *all* resolutions of nondeterminism [8]. Treating the two threads under consideration asymmetrically is also sound because verification involves considering *all* possible ordered pairs of distinct threads, so for any pair of distinct threads s and t, analysis will be performed with respect to the ordered pair (s, t) as well as the ordered pair (t, s).

The above encoding avoids quantifiers, but each array access leads to a nondeterministic choice so that the number of paths through the instrumented program grows exponentially with the number of accesses. For kernels that exhibit hundreds of syntactically distinct reads and writes, this leads to prohibitively slow verification.

[7] For brevity, this description of the optimization focusses on the situation where all threads executing a kernel are in a single work group; GPUVerify is sensitive to the multi-group case.

Solver	Configuration	Total Time (secs)			Total Speedup			Aggregate Speedups			
		All	Pass	Fail	All	Pass	Fail	Min	Max	Med	Avg
z3	baseline	11882	9070	2812							
	rr	10464	8074	2389	1.1	1.1	1.2	0.5	15.7	1.0	1.2
	rr+bi	10016	7629	2387	1.2	1.2	1.2	0.6	18.5	1.0	1.3
	rr+bi+pa	8206	5973	2232	1.4	1.5	1.3	0.6	77.1	1.1	1.8
	rr+bi+pa+wd	5774	3966	1807	2.1	2.3	1.6	0.5	86.9	1.1	2.5
cvc4	baseline	6080	3616	2464							
	rr	5000	3116	1884	1.2	1.2	1.3	0.1	4.4	1.2	1.3
	rr+bi	5002	3094	1907	1.2	1.2	1.3	0.1	4.3	1.1	1.3
	rr+bi+pa	4450	2611	1838	1.4	1.4	1.3	0.1	16.6	1.1	1.4
	rr+bi+pa+wd	3448	1921	1526	1.8	1.9	1.6	0.3	16.5	1.2	1.5

Table 1. Summary of optimization results for different solvers. Each speedup is reported with respect to the baseline results for the relevant solver

This led us to devise an alternative race detection method which we call *watchdog* race checking. Watchdog race checking uses a single, unconstrained constant representing an offset with respect to which races should be checked: the "watched offset". Verification involves proving for every array that a data race at the watched offset is impossible. Because the watched offset is arbitrary, this implies that every offset of each array is race-free. For each array, two Booleans are introduced to record whether a read from or write to the watched offset has occurred. Initially these Booleans are false, and they are reset at each barrier. Thread s sets the "read" Boolean to true whenever it reads from the watched offset, and similarly for the "write" Boolean. A race between s and t is reported if thread t reads from the watched offset and the "write" Boolean is *true*, or if thread t writes to the watched offset and either the "read" or "write" Boolean is *true*. The non-deterministic choice per array access is eliminated.

In practice we have adapted the watchdog method so that at each barrier we nondeterministically choose whether to check for data races until the next barrier. This allows the Boolean variables to be set to *false* at barriers by simply *assuming* that they are false. This removes these variables from the modifies sets (modsets) of loops, simplifying invariant generation. We thus reduce blow-up from being exponential in the number of array accesses to exponential in the number of barriers, typically a much smaller number.

Effect of optimizations on the benchmarks Table 1 shows the effects of our optimizations over the *responsive* set of kernels. We show results with Z3 and CVC4 being used for SMT solving. Recall that *baseline* refers to GPUVerify with type-level memory modeling and uniformity analysis but without further optimizations. We use **rr**, **bi**, **pa** and **wd** to refer to the redundant read, barrier interval, private array and watchdog race checking optimizations, respectively. We consider applying these optimizations on top of *baseline* in this order; this was the order in which we added the optimizations to

Portfolio Configuration	**Total Time (secs)**		
	All	Pass	Fail
Z3 and CVC4, baseline	4875	3031	1843
Z3 and CVC4, all optimizations	2900	1696	1203
All solver and optimization configurations	2825	1659	1166

Table 2. Summary of theoretical optimization results using portfolio solving

GPUVerify, so it illustrates the evolution of the tool. The *Total Time* columns show the total time for analysis, summed over all benchmarks (*All*), and also restricted to benchmarks for which verification passes (*Pass*) and fails (*Fail*). For each configuration except *baseline*, the *Speedup* columns show the speedup of an optimization configuration over baseline, for each solver. The *Aggregate Speedups* columns show the minimum (*Min*), maximum (*Max*), median (*Med*) and mean (*Avg*) speedups over *baseline*.

The key message from Table 1 is that our optimizations are increasingly effective, and effective overall, but that the overall speedups afforded by our efforts are modest: 2.1× with Z3 and a 1.8× with CVC4. The maximum and minimum speedups per benchmark show that the effects of an "optimization" can be dramatic, both positively and negatively: an 86.9× speedup is observed for one benchmark with Z3 and full optimizations (543 seconds to 6 seconds); with CVC the worst speedup is 0.3× (a 3.3× *slowdown*) with full optimizations. The median and mean results suggest that our optimizations have little impact on a significant number of the benchmarks. With the exception of watchdog race checking (a change in race instrumentation is relevant to *all* benchmarks) this is not surprising: many kernels do not exhibit read-only arrays (thus **rr** cannot help), many do not use vectors (thus **pa** cannot help) and we have already argued that the **bi** optimization is rather specialized. We find it counter-intuitive that the **rr** optimization, which simplifies the Boogie program generated by GPUVerify, has such a negative impact in the worst case with CVC4. The associated kernel has a single read-only array and went from 4 seconds using *baseline* to 49 seconds using **rr**. The unpredictable nature of SMT solvers motivates using multiple solvers during analysis.

The potential for portfolio verification Our combination of solvers and optimizations opens the door for "portfolio verification": running multiple instances of GPUVerify completely independently using different solver and optimization configurations, reporting the first analysis result yielded by a configuration. Table 2 shows the lowest total time for analysis over the *responsive* benchmarks that could be expected using portfolio verification with multiple solver and optimization configurations. Comparing the best total time in Table 1 (3448 seconds for CVC4) with the total time for full portfolio verification in Table 2 (2825 seconds) the best further speedup portfolio verification could give is a modest 1.2× overall. We see the main potential of portfolio verification to be minimizing the response time of GPUVerify.

5 False Positives and Error Reporting

Although GPUVerify performs sound verification, we envisage the tool being useful in practice for bug-finding, where failed proof attempts shed light on genuine defects. Feedback from GPU programmers elicited through talks and tutorials at industry-focused events appear to support this usage mode. We have taken several steps to reduce false positives and improve the quality of error messages reported by the tool.

Aliasing assumptions on kernel entry A GPU kernel operates on a number of shared arrays, provided as pointer arguments. According to the OpenCL and CUDA documentation, there is nothing to stop these pointers from aliasing one another. In practice we have not encountered a single case of such aliasing across our set of 564 benchmarks. To be truly sound, GPUVerify should assume that pointers could overlap arbitrarily. This would lead to false positive race reports for practically *all* array accesses, rendering the tool unusable. To avoid this, we took the pragmatic decision to have GPUVerify silently assume that distinct pointer parameters to a kernel refer to disjoint arrays.

While validating GPUVerify, an engineer at our industrial partner Rightware identified this source of unsoundness: *"I have probably uncovered a minor bug in GPUVerify . . . if we have a kernel like* [the slightly simplified example on the right] *GPUVerify happily says it's all right. However the user can ... set the same memory object as an argument for both a and b ...* [w]*hich has a clear race condition"*. When asked whether this scenario is likely in practice, the engineer confirmed: *"We don't have any kernels where it would be wise to pass the same pointer value in multiple arguments"*, but suggested that GPUVerify could emit a warning about its aliasing assumptions if the developer has not used the C99 `restrict` qualifier to indicate explicitly that pointer arguments refer to disjoint data: *"I'd recommend a warning when not using restrict, because in probably all the practical cases the kernel arguments are separate"*.

```
__kernel void
aliasing(__global int* a,
         __global int* b) {
  b[get_global_id(0)+1]
    = a[get_global_id(0)];
}
```

In response to this advice we added a pass to GPUVerify which emits a warning if a kernel has multiple `__global` pointer arguments that are not restrict-annotated.

Auto-inlining Although GPUVerify supports a modular analysis mode, automatic generation of procedure specifications is challenging, and imprecise specifications lead to false positives. Typical GPU kernels are free from recursion and function pointers (both are forbidden in OpenCL and only recently allowed in CUDA as part of CUDA 3.1) and are presented in whole-program form. To avoid false positives we automatically apply aggressive inlining to kernels that do not use recursion or function pointers, repeatedly inlining calls until no calls remain. All non-kernel functions are then discarded, and verification is attempted on the now monolithic kernel functions. One downside to this pragmatic approach is that if a source file contains a function that is never invoked, but which would exhibit a data race if it were to be invoked, GPUVerify will not analyse the function and thus will not report a possible defect. Across the *responsive* set, in *baseline* mode, we find that disabling auto-inlining leads to false positives being reported for 31 kernels where verification succeeds with auto-inlining enabled.

KernelInterceptor: verifying dynamically collected kernel instances A GPU kernel is typically designed to work correctly only for certain thread counts and input values, and GPUVerify requires preconditions specifying constraints on these parameters to avoid false positive error reports. Providing these preconditions can be a barrier to using the tool: preliminary experience with engineers at Rightware and ARM suggest that even the developer of a kernel may not be able to immediately identify suitable constraints on kernel parameters. To help overcome this we have designed KernelInterceptor [5], a tool to accompany GPUVerify for analysis of OpenCL kernels. KernelInterceptor is a shim which intercepts calls to the OpenCL host API used to specify thread counts and input parameters, gathering this data for all kernel instances launched during the running of an application. GPUVerify can then be automatically invoked using preconditions corresponding to each kernel instance that was observed, eliminating false positives arising due to unrealistic parameter values.

Error reporting Our initial GPUVerify prototype reported verification failures by printing a trace for the Boogie program generated by the tool; this information was of limited use even to us as the tool's developers. To allow clear error reports referring to the original source code of a kernel we have extended our Bugle front-end so that source information, available from LLVM IR if Clang is invoked appropriately, is embedded in the generated Boogie code via Boogie attributes.

Meaningfully reporting data races proved more difficult than we anticipated, because a data race involves *two* access operations: a *first* access is logged, and subsequently a race is detected due to a conflicting *second* access. At the Boogie level, an assertion corresponding to the *second* access fails. Source information for this access is available from attributes attached to the assertion, but source information for the *first* access is *not* directly available. Furthermore, the race report may stem from an abstract trace that jumps over a loop by replacing the loop with a summary computed using an invariant. In this case it may be that no specific *first* access is responsible for the race report; instead, it may be that GPUVerify could not find a strong enough invariant to prove race-freedom between the loop and the *second* access. To overcome this reporting problem we use Boogie's *state capture* facility to ask the SMT solver to provide a valuation of all program variables after each logging operation and at the head of each loop. This allows us to walk an abstract counterexample trace and determine whether the possible race is due to a specific *first* access or instead stems from the abstraction of a loop. In the former case we can provide a specific race report, otherwise we report all relevant array accesses in the loop as possibly racing with the *second* access.

6 Engagement With Industry

We briefly summarize our efforts to make it easy for industrial users to access and learn about GPUVerify, and discuss preliminary feedback from two industrial partners.

Industry-friendly licenses The licenses associated with the Clang/LLVM and Boogie frameworks mean that they can be used freely in a commercial context. This is not true of the Z3 solver. Providing support for CVC4, with a license attractive to industry, has been vital in allowing industrial partners to try out GPUVerify.

Web access We have made GPUVerify available as a web service through Microsoft's `rise4fun` site,[8] allowing interested users to try the tool with no installation overhead.

Tutorial videos To give potential users in industry a practical overview of GPUVerify we have recorded a series of tutorial videos, available on YouTube. We have also given in-person tutorials at industry-focussed conferences and OpenCL vendor sites.

[8] http://rise4fun.com/GPUVerify-OpenCL and http://rise4fun.com/GPUVerify-CUDA

Preliminary industrial feedback Feedback from our industrial contacts at Rightware and ARM on GPUVerify has been encouraging. Rightware have used GPUVerify to verify race-freedom across their Basemark CL suite, discovering one defect in the process. We are collaborating with engineers at ARM on adapting GPUVerify to provide tailored analysis support for OpenCL kernels targeting ARM's Mali GPU series.

7 Lessons Learned and Future Problems

We summarize the principal take-aways from our experience building GPUVerify, and pose what in our view is the main challenge associated with future work in this area.

Lessons learned We hope the following may be informative for future projects.

Target a tractable problem GPUVerify's goals are modest: attempt to verify race-freedom (not full functional correctness) for GPU kernels (not arbitrary C programs). With this tight scope we have been able to exploit the relative simplicity of GPU kernels to achieve a fairly high degree of automation and efficiency.

Re-use infrastructure We cannot overstate how much we have gained by exploiting Clang/LLVM, Boogie, Z3 and CVC4. When considering infrastructure re-use, it is worth paying attention to licensing issues if the ultimate goal is industrial uptake.

Evolve front-end capabilities All software verification tools have to face the "front-end" problem. Restricting to a toy language simplifies front-end development but dooms a tool to only academic use; working with a full-fledged compiler infrastructure can blur implementation difficulty with the essence of a new idea. We advocate a staged approach to this problem as outlined in Sect. 3, which is aided by intermediate verification languages such as Boogie [6] and Why3 [19].

Beware of outliers We optimized GPUVerify in response to kernels for which performance was particularly bad. We achieved massive speedups for some outliers, but were brought down to earth by the modest overall speedups and new *negative* outliers resulting from our optimizations (see Table 1). Evaluating the general effectiveness of verification optimizations requires a large set of benchmarks.

Challenge: flexible invariant generation The main weakness of GPUVerify is its invariant generation capabilities. We use Houdini for invariant generation in GPUVerify because, though somewhat brute force, it is *flexible* and applicable to arbitrary programs. At present we are unable to exploit advanced invariant generation techniques because they restrict the form of programs to which they can be applied. We offer our large set of publicly available benchmarks as a challenge for invariant generation researchers interested in lifting restrictions on program form.

Acknowledgments This work was supported by the EU FP7 project CARP, EPSRC grant EP/K011499/1, a PhD studentship partly sponsored by ARM Ltd., a gift from Intel Corporation, and the Imperial College London UROP scheme. The project benefited from discussions at Dagstuhl seminar 13142. We are grateful to Teemu Virolainen at Rightware and Anton Lokhmotov at ARM for their feedback on GPUVerify.

References

1. Allen, J., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: POPL. pp. 177–189 (1983)
2. AMD: AMD Accelerated Parallel Processing (APP) SDK, http://developer.amd.com/sdks/amdappsdk
3. Bakhoda, A., Yuan, G.L., Fung, W.W., Wong, H., Aamodt, T.M.: Analyzing CUDA workloads using a detailed gpu simulator. In: ISPASS. pp. 163–174 (2009)
4. Bardsley, E., Donaldson, A.F.: Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels. In: NFM. pp. 230–245 (2014)
5. Bardsley, E., Donaldson, A.F., Wickerson, J.: KernelInterceptor: Automating gpu kernel verification by intercepting kernels and their parameters. In: IWOCL (2014)
6. Barnett, M., et al.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. pp. 364–387 (2005)
7. Barrett, C., et al.: CVC4. In: CAV. pp. 171–177 (2011)
8. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: OOPSLA. pp. 113–132 (2012)
9. Che, S., et al.: Rodinia: A benchmark suite for heterogeneous computing. In: Workload Characterization. pp. 44–54 (2009)
10. Chiang, W.F., Gopalakrishnan, G., Li, G., Rakamaric, Z.: Formal analysis of GPU programs with atomics via conflict-directed delay-bounding. In: NFM. pp. 213–228 (2013)
11. Chong, N., Donaldson, A.F., Kelly, P., Ketema, J., Qadeer, S.: Barrier invariants: a shared state abstraction for the analysis of data-dependent GPU kernels. In: OOPSLA (2013)
12. Chong, N., Donaldson, A.F., Ketema, J.: A sound and complete abstraction for reasoning about parallel prefix sums. In: POPL. pp. 397–410 (2014)
13. Chou, C.T., Mannava, P.K., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: FMCAD. pp. 382–398 (2004)
14. Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic crosschecking of data-parallel floating-point code. IEEE Trans. Software Eng. (2014), to appear
15. Collingbourne, P., Donaldson, A.F., Ketema, J., Qadeer, S.: Interleaving and lock-step semantics for analysis and verification of GPU kernels. In: ESOP. pp. 270–289 (2013)
16. Danalis, A., et al.: The scalable heterogeneous computing (SHOC) benchmark suite. In: GPGPU 2010. pp. 63–74 (2010)
17. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of DMA races using model checking and *k*-induction. Formal Methods in System Design 39(1), 83–113 (2011)
18. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)
19. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: ESOP. pp. 125–128 (2013)
20. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: FME. pp. 500–517 (2001)
21. Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a high-level language targeted to GPU codes. In: InPar (2012)
22. Harris, M., Buck, I.: GPU flow-control idioms. In: GPU Gems 2. Addison-Wesley (2005)
23. Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs using permission-based separation logic. In: BYTECODE (2013)
24. Karrenberg, R., Hack, S.: Improving performance of OpenCL on CPUs. In: CC. pp. 1–20 (2012)
25. Khronos OpenCL Working Group: The OpenCL specification, version 2.0 (2013)

26. Leung, A., Gupta, M., Agarwal, Y., et al.: Verifying GPU kernels by test amplification. In: PLDI. pp. 383–394 (2012)
27. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: FSE. pp. 187–196 (2010)
28. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: Concolic verification and test generation for GPUs. In: PPoPP. pp. 215–224 (2012)
29. Li, P., Li, G., Gopalakrishnan, G.: Parametric flows: Automated behavior equivalencing for symbolic analysis of races in CUDA programs. In: SC. pp. 29:1–29:10 (2012)
30. McMillan, K.: Verification of infinite state systems by compositional model checking. In: CHARME. pp. 219–234 (1999)
31. Microsoft Corporation: C++ AMP sample projects for download (MSDN blog), http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx
32. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340 (2008)
33. NVIDIA: CUDA C programming guide, version 5.5 (2013)
34. NVIDIA: GPU Computing SDK (accessed 2013), https://developer.nvidia.com/gpu-computing-sdk
35. Rakamaric, Z., Hu, A.J.: Automatic inference of frame axioms using static analysis. In: ASE. pp. 89–98 (2008)
36. Rakamaric, Z., Hu, A.J.: A scalable memory model for low-level code. In: VMCAI. pp. 290–304 (2009)
37. Rightware Oy: Basemark CL, http://www.rightware.com/benchmarking-software/basemark-cl/
38. Stratton, J., et al.: Parboil: A revised benchmark suite for scientific and commercial throughput computing. Tech. Rep. IMPACT-12-01, UIUC (2012)
39. Talupur, M., Tuttle, M.R.: Going with the flow: Parameterized verification using message flows. In: FMCAD. pp. 1–8 (2008)