Automatic Termination Analysis for GPU Kernels*

Jeroen Ketema and Alastair F. Donaldson

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

Abstract

We describe a method for proving termination of massively parallel GPU kernels. An implementation in KITTeL is able to show termination of 94% of the 598 kernels in our benchmark suite.

1 Introduction

Graphics processing units (GPUs) are highly parallel shared-memory processors that can accelerate compute intensive applications. To leverage the power of a GPU, a programmer identifies a part of an application that exhibits parallelism; this part can then be extracted into computational *kernel* and *offloaded* to a GPU.

Kernel programming languages such as CUDA [14] and OpenCL [11] are data-parallel languages that use *barriers* for synchronisation. Roughly, when a thread reaches a barrier, it waits until all other threads have also reached a barrier. Once a barrier has been reached by every thread, execution stalls until all outstanding writes to shared memory have been committed. Committing all writes ensures that any write to shared memory that occurs before a barrier is visible to any thread after the barrier; this enables threads to communicate.

As GPUs are separate devices to which kernels are offloaded, it is generally difficult to perform live debugging. Hence, different means are needed to identify bugs. In previous work [3, 6], we have looked at uncovering data races. Here we consider termination.

Unlike CPU applications, which may be reactive, GPU kernels are *required* to terminate: any data computed by a kernel is inaccessible from the CPU as long as the kernel has not terminated. Besides this practical consideration, termination is also important from a theoretical perspective: the data race detection method described in [3], which underpins our data race detection tool GPUVerify, is only sound for terminating kernels.

We describe and evaluate a method for proving termination of kernels. Termination analysis is complicated by concurrency, but there is no need to reason about recursive calls or dynamically changing data structures since recursion and dynamic memory allocation are not generally supported by kernel programming languages.

The contributions of this paper are two-fold:

1. We leverage termination analysis for sequential programs to obtain an analysis technique for GPU kernels; the technique abstracts from all threads except an arbitrary one.
2. We adapt an existing termination analysis tool—KITTeL [8, 9]—and show that it can be successfully applied to a large set of real-world kernels.

2 Reduction to Sequential Termination Analysis

We present the kernel programming language from [5], which has the following grammar:

$$\begin{array}{lll} \text{expr} & e & ::= c \mid v \mid A[e] \mid e_1 \mathbin{op} e_2 \\ \text{stmt} & s & ::= v := e \mid A[e_1] := e_2 \mid \text{if } (e)\,\{ss_1\}\,\text{else}\,\{ss_2\} \mid \text{while } (e)\,\{ss\} \mid \text{barrier} \\ \text{stmts} & ss & ::= \varepsilon \mid s; ss \end{array}$$

Here, c and v represent constants and *thread-private* variables; A and op represent *shared* arrays and arbitrary binary operators. As explained in the introduction, the barrier statement

* This work was supported by the EU FP7 STREP project CARP (project number 287767).

$$\frac{[\![e]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}}}{(\sigma_\mathrm{v}, \sigma_\mathrm{A}, \text{while}\,(e)\,\{ss\}; ss') \to_\mathrm{s} (\sigma_\mathrm{v}, \sigma_\mathrm{A}, ss \cdot \text{while}\,(e)\,\{ss\}; ss')}\ (\textsc{Loop-T}) \qquad \frac{\neg[\![e]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}}}{(\sigma_\mathrm{v}, \sigma_\mathrm{A}, \text{while}\,(e)\,\{ss\}; ss') \to_\mathrm{s} (\sigma_\mathrm{v}, \sigma_\mathrm{A}, ss')}\ (\textsc{Loop-F})$$

(a) The thread-level rules (operating over thread states (σ_v, ss) and shared memory σ_A)

$$\frac{K(t) = (\sigma_\mathrm{v}, ss) \qquad (\sigma_\mathrm{v}, \sigma_\mathrm{A}, ss) \to_\mathrm{s} (\sigma'_\mathrm{v}, \sigma'_\mathrm{A}, ss') \qquad K' = K[t \mapsto (\sigma'_\mathrm{v}, ss')]}{(\sigma_\mathrm{A}, K) \to_\mathrm{k} (\sigma'_\mathrm{A}, K')}\ (\textsc{Step})$$

$$\frac{\left(\forall t : \exists \sigma_\mathrm{v} : \bigvee \begin{array}{c} (\exists ss : K(t) = (\sigma_\mathrm{v}, \text{barrier}; ss) \wedge K'(t) = (\sigma_\mathrm{v}, ss)) \\ (K(t) = (\sigma_\mathrm{v}, \varepsilon) \qquad \wedge K'(t) = (\sigma_\mathrm{v}, \varepsilon)) \end{array}\right) \qquad \exists t, \sigma_\mathrm{v}, ss : K(t) = (\sigma_\mathrm{v}, \text{barrier}; ss)}{(\sigma_\mathrm{A}, K) \to_\mathrm{k} (\sigma_\mathrm{A}, K')}\ (\textsc{Barrier})$$

(b) The Kernel-level rules (operating over kernel states (σ_A, K))

Figure 1 Operational semantics of our kernel programming language

allows for synchronisation between threads; ε represents the empty sequence of statements. A *kernel program* P is a sequence of statements ss; all threads execute the same sequence ss. For technical reasons we assume that an expression e has at most one sub-expression of the form $A[e']$, so that evaluating an expression involves reading at most once from the shared state; a kernel can be trivially preprocessed to satisfy this restriction.

A *thread state* is a pair (σ_v, ss), where σ_v represents the private memory of a thread—mapping private variables to values—and where ss is the sequence of the statements the thread needs to execute. A *kernel state* is a pair (σ_A, K), where σ_A represents the shared memory of the kernel—mapping shared arrays to sequences of values—and where K is a map from a *finite* set of thread identifiers t to thread states. The *initial kernel state* of a kernel program P is any state such that the second component of each $K(t)$ is P.

Figure 1 gives the operational semantics of the language. For brevity, we omit the rules for the assignments and if-statement, which are straightforward, and refer the reader to [5]. The rules for the while-statement evaluate the guard e under σ_v and σ_A, denoted $[\![e]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}}$, and proceed accordingly. As can be seen from rule Step, the language has an interleaving semantics. Rule Barrier is used for synchronisation between threads: no thread can proceed beyond a barrier unless all threads have either reached a barrier or have terminated. The rule requires that at least one thread is actually at a barrier; this ensures that the rule no longer fires once all threads have terminated (i.e., once all have reached a state $(\sigma_\mathrm{v}, \varepsilon)$).

We next reduce the termination problem for kernel programs to a sequential termination problem. The reduction makes termination analysis for kernel programs thread-modular by checking termination of a single, arbitrary thread under an environmental abstraction that over-approximates the effects of the other threads.

To obtain the abstraction, existentially quantify the premise of each thread-level rule over all array stores σ_A and replace rule Barrier by the thread-level rule $(\sigma_\mathrm{v}, \sigma_\mathrm{A}, \text{barrier}; ss) \to (\sigma_\mathrm{v}, \sigma_\mathrm{A}, ss)$. Denote by $\to_{\mathrm{s},\star}$ the over-approximating thread-level reduction relation such obtained. The relation ensures that the contents of σ_A is irrelevant and that a thread no longer has to wait for any other thread once it reaches a barrier. We have the following.

▶ **Theorem 2.1.** *Let P be a kernel program. If for each σ_v and σ_A it holds that all reductions $(\sigma_\mathrm{v}, \sigma_\mathrm{A}, P) \to_{\mathrm{s},\star} \cdots \to_{\mathrm{s},\star} \cdots$ are finite, then P terminates under the semantics of Figure 1.*

Proof. Suppose the contrary, then there exists an infinite reduction ρ of P. As the number of threads is finite, there is a thread t that is selected an infinite number of times by rule Step. We construct an infinite reduction for t under $\to_{\mathrm{s},\star}$: (i) for each application of Step

selecting t apply the over-approximating version of the thread-level rule employed and (ii) for each application of rule BARRIER employ the over-approximating barrier rule. The over-approximating rules fire, as (i) the existential quantification over all array stores σ_A ensures that e can be evaluated precisely as in ρ and as (ii) the thread-level barrier rule essentially skips a barrier. Hence, we have an infinite reduction for t under $\rightarrow_{s,\star}$, contradiction. ◀

A theorem related to the one above underpins the soundness of GPUVerify, where shared state abstraction allows race-freedom to be verified by considering just *two* arbitrary threads [3]. Observe that the theorem only modifies the operational semantics; kernel programs are left unchanged. Furthermore, the reverse of the theorem does not hold: termination might depend on shared memory sub-expressions evaluating to specific values.

3 Experimental Evaluation

To evaluate the effectiveness of Theorem 2.1, we adapted the `KITTeL` termination analysis tool [8, 9] and applied it to a suite of 598 kernels, 381 of which have loops. To demonstrate that our approach works out-of-the-box, we included the loop-free kernels in our evaluation. The kernels have on average 86 lines of code and originate from nine sources:

- *AMD Accelerated Parallel Processing SDK* v2.6 [1] (78 kernels, 54 of which have loops).
- *NVIDIA GPU Computing SDK* v5.0 [13] (183 kernels, 109 of which have loops); we also include 8 kernels from v2.0 of the SDK, 7 of which have loops.
- *C++ AMP Sample Projects* [12] (20 kernels, 16 of which have loops)
- The *gpgpu-sim* benchmarks [2] (33 kernels, 24 of which have loops)
- The *Parboil* benchmarks v2.5 [16] (25 kernels, 19 of which have loops)
- The *Rodinia* benchmark suite v2.4 [4] (36 kernels, 24 of which have loops)
- The *SHOC* benchmark suite [7] (87 kernels, 53 of which have loops)
- The *PolyBench/GPU* benchmark suite [10] (64 kernels, 49 of which have loops)
- Rightware *Basemark CL* v1.1 [15] (64 kernels, 26 of which have loops)

Each suite is publicly available except for *Basemark CL* which was provided to us under an academic license. The collection covers all the publicly available GPU benchmark suites we are aware of. The kernel counts above do not include 5 kernels that we manually removed because they use CUDA surfaces or thread fences [14], which we currently do not support.

KITTeL The `KITTeL` termination analysis tool [8, 9] consists of a front-end, `llvm2KITTeL`, which takes llvm bitcode[1] and translates this into an integer-based rewrite system. The back-end automatically tries to prove termination of the generated rewrite system.

We adapted `llvm2KITTeL` to handle kernels (compiled to bitcode by Clang[2]); we did not make any changes to the termination analysis back-end. As `llvm2KITTeL` models only a single thread and already abstracts from most memory operations (yielding nondeterministic values for loads from memory), the changes we needed to make were minimal. To summarise: (i) we ensured that `llvm2KITTeL` abstracts loads even in cases where it usually does not (e.g., when a pointer points to a unique global variable representing a single integer), (ii) we disabled the hoisting of loop-invariant loads from loops (due to concurrency the loaded value might differ between loop iterations), and (iii) we made `llvm2KITTeL` aware of the fact that

[1] `http://llvm.org/`
[2] `http://clang.llvm.org/`

the number of threads executing a kernel is constant for the duration of an execution (the number of threads is often referred to in loop guards; hence, awareness that this number is constant—or at least bounded—is often critical for showing termination).

Loop Invariants Currently, `KITTeL` does not infer loop invariants that may be needed for proving termination. We provided these invariants by hand and proved them correct with GPUVerify prior to running our experiments. In principle we could extend GPUVerify's invariant inference engine [3] to generate the needed invariants; this would require infrastructure to feed the generated invariants into `KITTeL`.

We required loop invariants stating: (i) the loop counter must be positive (31 kernels), (ii) the step value for the loop counter is positive (18 kernels), (iii) the loop counter is always smaller than or equal to a value which is subtracted from it in the loop guard (2 kernels).

Experimental Setup All experiments were conducted on a Mid 2009 MacBook Pro with a 2.53GHz Intel Core 2 Duo and 4GB RAM running OS X 10.9.2 and Clang/llvm 3.4. The reported times are averages over three runs and include the time needed to compile a kernel into bitcode. The timeout used was 10 minutes. We adapted `llvm2KITTeL` as described above and always invoked the tool with its `-increase-strength` option—turning left and right shifts into multiplications and divisions, respectively. The latter facilitates termination analysis of kernels where the loop counter is being shifted. The SMT solver used with `KITTeL` was Z3 v4.3.1. Both `llvm2KITTeL` and `KITTeL` were downloaded on 21-04-2014.[3]

Results Unsurprisingly, `KITTeL` managed to prove termination of all 217 loop-free kernels. On average termination of these kernels was shown in 0.63s and the maximum time required was 6.15s. Six of the kernels needed over 1s; this was either due to a long compilation time or the kernel consisting of a large number of subroutines.

Of the 381 kernels with loops, 346 could be shown terminating. On average termination was shown in 7.23s and the maximum time needed was 254.17s (see also Figure 2). Of the 35 kernels for which termination could not be shown, 31 reached the timeout of 10 minutes. In 4 cases `KITTeL` indicated that the constructed rewrite system was nonterminating (this does not imply that the original kernels are nonterminating, as the constructed rewrite system in general over-approximates the behaviour of a thread).

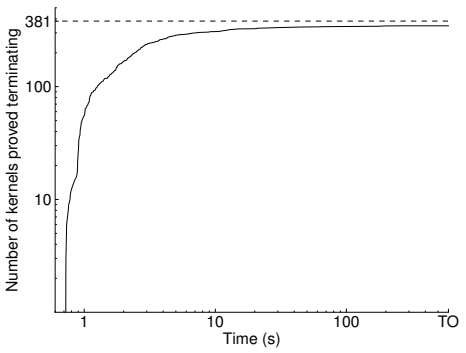

Figure 2 Cumulative histogram showing the time taken to prove termination of the kernels with loops

We manually inspected the 35 kernels to see why termination could not be shown. All 4 cases where `KITTeL` indicated nontermination would require reasoning over floating point numbers. In 4 other cases built-in atomic increment operations would need to be modelled as returning monotonically increasing values—instead of arbitrary ones, as is currently the case. In 19 cases termination would require reasoning about shared memory and, hence, the over-approximation from Theorem 2.1 is too coarse.

The above leaves 8 kernels, all of which timed out. Of these, 2 could be shown terminating using a very coarse over-approximation of division—yielding unconstrained nondeterministic

[3] `https://github.com/s-falke/{llvm2kittel,kittel-koat}`

values. One kernel could be shown terminating with `llvm2KITTeL`'s `-only-loop-conditions` option, which abstracts all basic blocks except those from which loops can be exited.

In the case of 2 kernels the function bodies were very large which resulted in a timeout in `llvm2KITTeL` (these were the only timeouts in `llvm2KITTeL`). In the 3 remaining cases a timeout occurred in `KITTeL`, although the generated rewrite system was terminating.

4 Conclusion

We have described an approach for proving termination of massively parallel GPU kernels by reducing the termination problem for these kernels to a sequential termination problem. With the help of an adapted version of `KITTeL` the reduction allowed us to prove termination of 94% of the kernels in our benchmark set and of 91% of the kernels with loops.

As part of future work we would like to automatically infer loop invariants that are required for proving termination. Moreover, we would like to investigate whether performance can be improved by outlining—as opposed to inlining—loops into separate procedures.

Acknowledgements The authors wish to thank Adam Betts, Nathan Chong and Stephan Falke for feedback on the paper. The authors also wish to thank Stephan Falke for making `KITTeL`'s source code publicly available and for answering questions regarding the tool.

References

1 AMD. AMD Accelerated Parallel Processing (APP) SDK. `http://developer.amd.com/sdks/amdappsdk`.
2 A. Bakhoda et al. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, pages 163–174, 2009.
3 A. Betts et al. GPUVerify: a verifier for GPU kernels. In *OOPSLA*, pages 113–132, 2012.
4 S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization*, pages 44–54, 2009.
5 N. Chong, A. F. Donaldson, and J. Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *POPL*, pages 397–410, 2014.
6 P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *ESOP*, pages 270–289, 2013.
7 A. Danalis et al. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU*, pages 63–74, 2010.
8 S. Falke, D. Kapur, and C. Sinz. Termination analysis of C programs using compiler intermediate languages. In *RTA*, pages 41–50, 2011.
9 S. Falke, D. Kapur, and C. Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *VSTTE*, pages 261–277, 2012.
10 S. Grauer-Gray et al. Auto-tuning a high-level language targeted to GPU codes. In *InPar*, 2012.
11 Khronos OpenCL Working Group. The OpenCL specification, version 1.2, 2012.
12 Microsoft Corporation. C++ AMP sample projects for download (MSDN blog). `http://goo.gl/eb8ob`.
13 NVIDIA. GPU Computing SDK. `https://developer.nvidia.com/gpu-computing-sdk`.
14 NVIDIA. CUDA C programming guide, version 5.0, 2012.
15 Rightware Oy. Basemark CL. `http://www.rightware.com/benchmarking-software/basemark-cl/`.
16 J. Stratton et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, UIUC, 2012.