Termination Analysis for GPU Kernels[☆]

Jeroen Ketema, Alastair F. Donaldson

Department of Computing, Imperial College London
London, United Kingdom

Abstract

We describe a thread-modular technique for proving termination of massively parallel GPU kernels. The technique reduces the termination problem for these kernels to a sequential termination problem by abstracting the shared state, and as such allows us to leverage termination analysis techniques for sequential programs. An implementation in `KITTeL` is able to show termination of 94% of 604 kernels collected from various sources.

Keywords: termination, abstraction, GPUs, concurrency
2010 MSC: 68Q60

1. Introduction

Termination analysis for sequential programs has made significant progress in the last two decades, owing to the discovery of transition invariants [1], forming the basis for tools like Terminator [2], and advances in termination analysis techniques for term rewriting systems, as implemented by tools like AProVE [3] and `KITTeL` [4, 5]. Researchers are now turning their attention to termination analysis for concurrent programs, which can be difficult due to the need for inter-thread reasoning to establish that computational progress is not unbounded.

The main contribution of this paper is to show that, despite the general difficulty of termination analysis in the presence of concurrency, in the domain of graphics processing unit (GPU) programming, *existing* methods for establishing termination of sequential programs can be successfully re-used to enable termination arguments for GPU programs to be established.

GPUs are highly parallel shared-memory processors that can accelerate computationally intensive applications such as medical imaging [6] and computational fluid dynamics [7]. 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 execute on a GPU.

[☆]This work was supported by the EU FP7 STREP project CARP (project number 287767).
Email addresses: `jketema@imperial.ac.uk` (Jeroen Ketema), `afd@imperial.ac.uk` (Alastair F. Donaldson)

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. For this reason, many researchers (including us [8, 9]) have looked at proving safety properties of kernels, in particular ones related to *data races* (see [8] for a recent overview of the work in this area). The current paper is the first to consider *termination*.[1]

Termination is important from a theoretical perspective, e.g., because the data race detection method described in [8], which underpins our GPUVerify tool, is only sound for terminating kernels. However, it is even more important from a practical perspective. 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 the data being inaccessible, kernels with accidental infinite loops can have a severe impact on the systems on which they run: while working on the experiments from [11], we accidentally introduced infinite loops on numerous occasions; this often made our systems unresponsive, and sometimes caused transient hardware failures and spontaneous reboots.

The termination technique we describe below is thread-modular. It operates by abstracting the state shared between the threads of a kernel and by considering each thread in isolation. As such, we reduce a concurrent termination problem to a sequential one, and are able to build on and re-use existing techniques and tools for proving termination. In fact, the sequential termination problem we end up with is somewhat easier than usual in that there is no reason to consider recursive calls and dynamically changing data structures; these features are generally not supported by kernel programming languages.

The contributions of this paper are as follows:

1. We leverage termination analysis techniques for sequential programs to obtain an analysis technique for GPU kernels. The analysis technique considers the execution of a kernel for a single arbitrary thread, using abstraction to over-approximate the possible effects of other threads; we show that if the arbitrary thread terminates in this abstract setting, then the GPU kernel is also guaranteed to terminate.
2. We adapt an existing termination analysis tool—`KITTeL` [4, 5]—and leverage the Clang/LLVM compiler to obtain a largely automatic source code-level termination analysis tool for CUDA [12] and OpenCL [13], the most widely used GPU programming languages.
3. We present an evaluation of our method on a set of 604 CUDA and OpenCL kernels, of which 386 have loops. Termination analysis is naturally fully automatic for the loop-free kernels, as well as for 90% of the kernels with loops, backing up our claim that methods for sequential termination analysis are effective when applied in the domain of GPU programming. We note that the success is in large part due to the fact that

[1] An outline of our approach was presented at the International Workshop on Termination in 2014 [10].

termination of GPU kernels rarely depends on values in shared memory.
4. We consider various features of `KITTeL` and evaluate their effectiveness over our set of 604 kernels. The evaluation highlights that more research into bitvector modelling and invariant inference seems appropriate in the context of sequential termination analysis.

In our view, the fact that sequential termination analysis techniques can be pushed towards providing automated termination analysis for GPU kernels is an encouraging result that shows how far the termination analysis field has come.

2. Anatomy of a GPU Kernel

Kernel programming languages such as CUDA [12] and OpenCL [13] are data-parallel languages that use *barriers* for synchronisation. When a thread reaches a barrier, it waits until all other threads have also reached the barrier. Once the barrier has been reached by all threads, execution stalls until all outstanding writes to shared memory have been committed. Committing the writes ensures that any write to shared memory that occurs before the barrier is visible to all threads after the barrier; this enables the threads to communicate.

As a running example throughout this paper we use the kernel depicted in Figure 1. This kernel, written in the CUDA kernel programming language [12], implements a Kogge-Stone prefix-sum [14]. Given an array `in` with values n_0, n_1, ..., n_i, ..., n_m, the kernel computes an array `out` with values

$$n_0,\ n_0 + n_1,\ \ldots,\ \sum_{0 \le k \le i} n_k,\ \ldots,\ \sum_{0 \le k \le m} n_k\,.$$

Computation of these values proceeds by having a *thread block* consisting of `blockDim.x` threads execute the prefix-sum algorithm. The array parameters `in` and `out` are *shared* between all threads, i.e., they are *global* arrays in CUDA terminology. The variable `temp` is *local*, meaning that every thread has a *private* copy not accessible to any other thread. The execution of a thread may depend on its unique identifier `threadIdx.x`, and threads may synchronise by calling the **`__syncthreads`** function, which represents a barrier in CUDA.

Although the kernel in Figure 1 is intended to be executed by a single 1-dimensional thread block, thread blocks may be multi-dimensional, and multiple blocks may be grouped into a *grid* to perform a computation. Moreover, there is an additional layer of memory, called *shared* memory, which sits in between global and local memory and which is shared solely between the threads in a single block.

We ignore multi-dimensional thread blocks in the remainder of this paper, as they only differ from 1-dimensional blocks in the availability of multi-dimensional thread identifiers, and these can easily be encoded as 1-dimensional identifiers. We also ignore grids, because their behaviour is identical to that of individual blocks, except that *no* synchronisation is possible between the threads in different blocks. As such, termination at the level of grids follows trivially once we are able to prove termination at the level of blocks.

```
__global__ void KoggeStone(int *in, int *out) {
    out[threadIdx.x] = in[threadIdx.x];
    __syncthreads();
    for (unsigned offset = 1; offset < blockDim.x; offset *= 2) {
        int temp;
        if (threadIdx.x >= offset)
          {temp = out[threadIdx.x - offset];}
        __syncthreads();
        if (threadIdx.x >= offset)
          {out[threadIdx.x] = temp + out[threadIdx.x];}
        __syncthreads();
    }
}
```

Figure 1: The Kogge-Stone prefix-sum in CUDA

Given that we only consider a single thread block, we bunch together global and shared memory, and simply call this memory *shared* memory from here onwards. Our implementation, discussed in Section 5, supports multi-dimensional thread blocks, grids, and global and shared memory in full, and is also able to handle kernels written in OpenCL [13].

3. Kernel Programming Language

We define a simple kernel programming language, with an interleaving operational semantics, with respect to which we will describe our thread-modular termination analysis in the next section. The language is a slight variation on the language from an earlier paper in which we investigate the correctness of prefix-sums [15]: the grammar of the language is taken as-is, but instead of considering a type system with multiple types, we consider only a single type Word, the type of all memory words. A single type suffices for our exposition, and adapting the analysis to handle multiple types is trivial.

The grammar of our language is as follows:

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

Here, c ranges over literal values, v and A range, respectively, over scalar and array variable names, and op ranges over an unspecified set of binary operators. The language is easily extended to cater for operators of other arities. Literals, variables, and array elements are all of type Word; each binary operator has type $\mathsf{Word} \times \mathsf{Word} \to \mathsf{Word}$. We also assume the existence of two designated literal values $\text{true}, \text{false} \in \mathsf{Word}$, representing the Booleans.

The statements $v := e$ and $A[e_1] := e_2$ denote assignment to variables and array elements, respectively. The statement $\text{if}\,(e)\,\{ss_1\}\,\text{else}\,\{ss_2\}$ represents conditional execution, and $\text{while}\,(e)\,\{ss\}$ allows for iteration. The barrier statement enables synchronisation between threads. By ε we denote an empty sequence of statements, and $s; ss$ prefixes a sequence of statements ss with a statement s.

```
out[tid] := in[tid];
barrier;
offset := 1;
while (offset < N) {
  if (tid ≥ offset)
    { temp := out[tid − offset]; }
  barrier;
  if (tid ≥ offset)
    { out[tid] := temp + out[tid]; }
  barrier;
  offset := offset ∗ 2;
}
```

Figure 2: The kernel from Figure 1 represented in our simple kernel programming language

A *kernel program* P is a sequence of statements ss; all threads executing the kernel program execute ss. Figure 2 shows a translation of the Kogge-Stone kernel of Figure 1 into our simple kernel programming language, where we omit empty else-branches for brevity. Each call to **__syncthreads** is turned into a barrier statement. The CUDA built-in variables `threadIdx.x` and `blockDim.x` map to the variables *tid* and N; these variables represent the unique identity of a thread and the total thread-count, respectively.

Operational Semantics. Let Var be a set of variables and Arr be a set of arrays. Our semantics is defined over *variable stores* σ_{v}, mapping variables $v \in \mathsf{Var}$ to elements of type Word, *array stores* σ_{A}, mapping arrays $A \in \mathsf{Arr}$ to maps M of type $\mathsf{Word} \to \mathsf{Word}$, and a finite set $D \subseteq \mathsf{Word}$ of *thread identifiers*. We assume D to be arbitrary but fixed in the remainder.

Each expression is evaluated under a variable store σ_{v} and an array store σ_{A}. Denoting the evaluation of an expression e by $[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} \in \mathsf{Word}$, we define:

$$[\![c]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} = c \qquad [\![A[e]]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} = \sigma_{\mathrm{A}}(A)([\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}})$$
$$[\![v]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} = \sigma_{\mathrm{v}}(v) \qquad [\![e_1 \mathbin{op} e_2]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} = [\![e_1]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} \mathbin{op} [\![e_2]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}}$$

A variable store is paired with a sequence of statements to form a *thread state* $(\sigma_{\mathrm{v,t}}, ss_t)$, where $\sigma_{\mathrm{v,t}}$ represents the *private memory* of a thread $t \in D$, mapping private variables to values, and where ss_t is the sequence of statements that remains to be executed by t. An array store is combined with multiple thread states to form a *kernel state* (σ_{A}, K). The array store σ_{A} represents the *shared memory* of a kernel, mapping shared array elements to values, and K is a map from D to thread states, specifying the current state of each thread executing the kernel.

Our operational semantics, as presented in Figure 3, is an interleaving semantics defined over kernel states. The semantics consists of two parts:

- a *thread-level semantics* (Figure 3a) describing the execution of a single statement by a single thread given a triple $(\sigma_{\mathrm{v}}, \sigma_{\mathrm{A}}, ss)$, where σ_{v} is the

$$\frac{x = [\![e]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}}}{(\sigma_\mathrm{v}, \sigma_\mathrm{A}, v := e; ss') \to_\mathrm{t} (\sigma_\mathrm{v}[v \mapsto x], \sigma_\mathrm{A}, ss')} \text{ (T-Assign)}$$

$$\frac{x_1 = [\![e_1]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}} \qquad x_2 = [\![e_2]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}} \qquad M = \sigma_\mathrm{A}(A)[x_1 \mapsto x_2]}{(\sigma_\mathrm{v}, \sigma_\mathrm{A}, A[e_1] := e_2; ss') \to_\mathrm{t} (\sigma_\mathrm{v}, \sigma_\mathrm{A}[A \mapsto M], ss')} \text{ (T-Array)}$$

$$\frac{[\![e]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}}}{(\sigma_\mathrm{v}, \sigma_\mathrm{A}, \text{if } (e) \{ss_1\} \text{ else } \{ss_2\}; ss') \to_\mathrm{t} (\sigma_\mathrm{v}, \sigma_\mathrm{A}, ss_1 \cdot ss')} \text{ (T-Ite-True)}$$

$$\frac{\neg[\![e]\!]^{\sigma_\mathrm{v}}_{\sigma_\mathrm{A}}}{(\sigma_\mathrm{v}, \sigma_\mathrm{A}, \text{if } (e) \{ss_1\} \text{ else } \{ss_2\}; ss') \to_\mathrm{t} (\sigma_\mathrm{v}, \sigma_\mathrm{A}, ss_2 \cdot ss')} \text{ (T-Ite-False)}$$

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

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

(a) The thread-level rules (specified over triples $(\sigma_\mathrm{v}, \sigma_\mathrm{A}, ss)$)

$$\frac{\begin{array}{c} K(t) = (\sigma_{\mathrm{v,t}}, ss_t) \\ (\sigma_{\mathrm{v,t}}, \sigma_\mathrm{A}, ss_t) \to_\mathrm{t} (\sigma'_{\mathrm{v,t}}, \sigma'_\mathrm{A}, ss'_t) \qquad K' = K[t \mapsto (\sigma'_{\mathrm{v,t}}, ss'_t)] \end{array}}{(\sigma_\mathrm{A}, K) \to_\mathrm{k} (\sigma'_\mathrm{A}, K')} \text{ (K-Step)}$$

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

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

Figure 3: Operational semantics of our kernel programming language

current (private) variable store of the thread, σ_A is the current (shared) array store of the kernel, and ss is the sequence of statements that remains to be executed by the thread;

- a *kernel-level semantics* (Figure 3b) describing the interleaving and synchronisation of the threads executing a kernel given a kernel state (σ_A, K) representing the current execution state of the kernel.

The thread-level rules of Figure 3a are standard for an imperative language, except that, instead of a single store, we have both a variable and an array store to account for private and shared memory. The assignment statements evaluate the relevant expressions and update the relevant store using the resulting values; the rules for the if- and while-statements evaluate the guard e under σ_v and σ_A

and proceed accordingly. In the figure, $\sigma[x \mapsto y]$ denotes a map identical to σ except that $\sigma[x \mapsto y](x) = y$. Furthermore, $ss \cdot ss'$ denotes the concatenation of the sequences ss and ss'.

The kernel-level rule K-Step of Figure 3b facilitates the interleaving of threads by selecting a thread state $K(t) = (\sigma_{\mathrm{v,t}}, ss_t)$ and performing a single thread-level step with respect to $(\sigma_{\mathrm{v,t}}, \sigma_{\mathrm{A}}, ss_t)$. Observe that no thread-level rule applies when $ss_t = \varepsilon$ or $ss_t = \text{barrier}; ss'_t$, i.e., when thread t has terminated or is at a barrier. Thus, rule K-Step can only fire when there is at least one active thread that is able to execute a non-barrier statement.

Rule K-Barrier enables synchronisation between threads: no thread can proceed beyond a barrier until each thread $t \in D$ either

- has reached a barrier, i.e., $K(t) = (\sigma_{\mathrm{v,t}}, \text{barrier}; ss_t)$, or
- has terminated, i.e., $K(t) = (\sigma_{\mathrm{v,t}}, \varepsilon)$.

In the first case, the barrier statement is removed from the sequence of statements that remains to be executed by t. In the second case, the state of t remains unchanged. The second premise of rule K-Barrier ensures that at least one thread is at the barrier. This guarantees that the rule does not fire once all threads have terminated. If we omit this condition, termination of all threads would not imply kernel termination (as defined below).

Remark 1 (Barrier Semantics). As in [15], the kernel-level rule K-Barrier follows the MPI programming model [16] in which threads may synchronise at syntactically distinct barriers. In OpenCL and CUDA the rules for barrier synchronisation are stricter [13, 12], requiring that (i) all threads synchronise at the same barrier, and that (ii) all threads have executed the same number of iterations of a loop l in case the barrier occurs inside l. A precise semantics for barrier synchronisation in GPU programming is presented in [8, 9].

The less restrictive MPI-style synchronisation model makes our termination analysis and associated main theorem (Theorem 3) more widely applicable. Adding stricter conditions to rule K-Barrier to capture the specific synchronisation requirements for GPU kernels does not affect our approach to proving termination. In other words, if a separate technique is able to ascertain that barriers are used correctly under the stricter rules for GPU kernels, then our termination analysis method can be applied unmodified.

Remark 2 (Atomicity of Statements). The interleaving semantics of Figure 3 assumes that statements are executed atomically with respect to sequentially consistent memory [17]. This does not accurately reflect the execution and memory models of GPUs [11]. However, this only affects kernels with data-races.

We believe that racy kernels should be avoided, as it is not clear from the OpenCL and CUDA specifications [13, 12] whether such kernels are well-defined. A separate data race analysis, e.g., the one implemented by GPUVerify [8, 18], can be used to assess race-freedom.

Execution and Termination. Let P be a kernel program. An *initial state of* P is any kernel state (σ_{A}, K) such that for each thread identifier $t \in D$ we have $K(t) = (\sigma_{\mathrm{v}}, P)$ with $\sigma_{\mathrm{v}}(tid) = t$ and $\sigma_{\mathrm{v}}(N) = |D|$. Thus, each thread starts executing from the same sequence of statements P, and the variables *tid* and N represent, respectively, the unique identity of a thread and the total thread-count. Our main result in the next section does not depend on *tid* and N, and, hence, our approach facilitates parameterised termination analysis for GPU kernels with respect to arbitrary thread counts. Nevertheless, we prefer to make *tid* and N explicit, because concrete kernels often rely on certain predicates to hold relating the number of threads and the values of certain kernel parameters. For example, in order to correctly compute a prefix-sum over an array of length n, the kernel of Figure 2 requires $|D| = n$.

Given an initial state $\mathcal{K}_0$ of a kernel program P, an *execution of* P *starting from* $\mathcal{K}_0$ is any finite or infinite sequence

$$\mathcal{K}_0 \to_{\mathrm{k}} \mathcal{K}_1 \to_{\mathrm{k}} \cdots \to_{\mathrm{k}} \mathcal{K}_i \to_{\mathrm{k}} \mathcal{K}_{i+1} \to_{\mathrm{k}} \cdots$$

with each $\mathcal{K}_i$ $(i \geq 0)$ a kernel state, and where the consecutive states are related by the $\to_{\mathrm{k}}$-relation of Figure 3b. A program P is said to *terminate for an initial kernel state* $\mathcal{K}_0$ *of* P, if all executions starting from $\mathcal{K}_0$ are finite. A kernel program P *terminates* if it terminates for each of its initial kernel states.

4. Proving Termination by Thread-Modular Analysis

We now reduce the termination problem for kernel programs to a sequential termination problem. The reduction makes the termination analysis of kernel programs thread-modular by checking termination of a single, arbitrary thread under an environmental abstraction that over-approximates the effects of all other threads. This allows the termination analysis for GPU kernels to re-use the wealth of existing research into termination analysis for sequential programs, including the rewriting-based techniques that we re-use in our tool implementation (see Section 5).

Our environmental abstraction ignores the contents of the (shared) array store and instead assumes that accessing an array element yields a nondeterministically selected value from Word. We achieve this by replacing the array-aware expression evaluation function $[\![\cdot]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} : \mathrm{expr} \to \mathsf{Word}$ by an array-*oblivious* function $[\![\cdot]\!]^{\sigma_{\mathrm{v}}}_{\star} : \mathrm{expr} \to 2^{\mathsf{Word}}$ defined as:

$$[\![c]\!]^{\sigma_{\mathrm{v}}}_{\star} = \{c\} \qquad [\![A[e]]\!]^{\sigma_{\mathrm{v}}}_{\star} = \mathsf{Word}$$
$$[\![v]\!]^{\sigma_{\mathrm{v}}}_{\star} = \{\sigma_{\mathrm{v}}(v)\} \qquad [\![e_1 \mathbin{op} e_2]\!]^{\sigma_{\mathrm{v}}}_{\star} = \{x_1 \mathbin{op} x_2 \mid x_1 \in [\![e_1]\!]^{\sigma_{\mathrm{v}}}_{\star} \wedge x_2 \in [\![e_2]\!]^{\sigma_{\mathrm{v}}}_{\star}\}$$

Thus, $[\![e]\!]^{\sigma_{\mathrm{v}}}_{\star}$ yields the set of all possible values an expression e may evaluate to given a variable store σ_{v}, and assuming that any array access may yield any value from Word.

$$\frac{x \in [\![e]\!]^{\sigma_\mathrm{v}}_\star}{(\sigma_\mathrm{v}, v := e; ss') \to_\mathrm{a} (\sigma_\mathrm{v}[v \mapsto x], ss')} \text{ (A-Assign)}$$

$$\frac{}{(\sigma_\mathrm{v}, A[e_1] := e_2; ss') \to_\mathrm{a} (\sigma_\mathrm{v}, ss')} \text{ (A-Array)}$$

$$\frac{\mathrm{true} \in [\![e]\!]^{\sigma_\mathrm{v}}_\star}{(\sigma_\mathrm{v}, \mathrm{if}\,(e)\,\{ss_1\}\,\mathrm{else}\,\{ss_2\}; ss') \to_\mathrm{t} (\sigma_\mathrm{v}, ss_1 \cdot ss')} \text{ (A-Ite-True)}$$

$$\frac{\mathrm{false} \in [\![e]\!]^{\sigma_\mathrm{v}}_\star}{(\sigma_\mathrm{v}, \mathrm{if}\,(e)\,\{ss_1\}\,\mathrm{else}\,\{ss_2\}; ss') \to_\mathrm{a} (\sigma_\mathrm{v}, ss_2 \cdot ss')} \text{ (A-Ite-False)}$$

$$\frac{\mathrm{true} \in [\![e]\!]^{\sigma_\mathrm{v}}_\star}{(\sigma_\mathrm{v}, \mathrm{while}\,(e)\,\{ss\}; ss') \to_\mathrm{a} (\sigma_\mathrm{v}, ss \cdot \mathrm{while}\,(e)\,\{ss\}; ss')} \text{ (A-Loop-True)}$$

$$\frac{\mathrm{false} \in [\![e]\!]^{\sigma_\mathrm{v}}_\star}{(\sigma_\mathrm{v}, \mathrm{while}\,(e)\,\{ss\}; ss') \to_\mathrm{a} (\sigma_\mathrm{v}, ss')} \text{ (A-Loop-False)}$$

$$\frac{}{(\sigma_\mathrm{v}, \mathrm{barrier}; ss) \to_\mathrm{a} (\sigma_\mathrm{v}, ss)} \text{ (A-Barrier)}$$

Figure 4: Abstract operational semantics of our kernel programming language

Abstract Operational Semantics. Employing our updated definition of expression evaluation, we specify an abstract operational semantics in Figure 4. The semantics is defined over *abstract states* (σ_v, ss), with σ_v a variable store and ss a sequence of statements. Hence, the semantics can be thought of as executing a single thread while ignoring the array store.

We discuss the abstract rules and relate them to the concrete rules of Figure 3. Rule A-Assign is the corresponding abstract version of rule T-Assign. The rule evaluates e under our abstract expression evaluation function before nondeterministically assigning one of the values from $[\![e]\!]^{\sigma_\mathrm{v}}_\star$ to v. As we abstract from the array store, rule A-Array is a simple no-op, unlike rule T-Array, which updates the array store. The rules for the if- and while-statements generalise the corresponding rules from Figure 3a by checking whether the appropriate truth value occurs in $[\![e]\!]^{\sigma_\mathrm{v}}_\star$. As the abstract semantics models the sequential semantics of a single thread, rule K-Step is disposed of, and K-Barrier is lowered to rule A-Barrier, which is a no-op. To summarise: our abstract semantics ensures that the contents of σ_A becomes irrelevant and that a thread no longer has to wait for any other thread once it reaches a barrier.

Abstract Execution and Termination. Recall that D is a finite set of thread identifiers. If P is a kernel program, then an *abstract initial state of* P is defined as any pair (σ_v, P) such that $\sigma_\mathrm{v}(tid) \in D$ and $\sigma_\mathrm{v}(N) = |D|$. Given an abstract

initial state $\mathcal{A}_0$ of P, an *abstract execution of P starting from $\mathcal{A}_0$* is any finite or infinite sequence

$$\mathcal{A}_0 \to_{\mathrm{a}} \mathcal{A}_1 \to_{\mathrm{a}} \cdots \to_{\mathrm{a}} \mathcal{A}_i \to_{\mathrm{a}} \mathcal{A}_{i+1} \to_{\mathrm{a}} \cdots$$

with each $\mathcal{A}_i$ ($i \geq 0$) an abstract state, and where the consecutive states are related by the $\to_{\mathrm{a}}$-relation of Figure 4. A program P is said to *terminate for an initial abstract state $\mathcal{A}_0$ of P*, if all abstract executions starting from $\mathcal{A}_0$ are finite. A kernel program P *terminates under the abstract semantics* if it terminates for each of its initial abstract states.

Thread-Modular Analysis. We now present our main result, which shows that termination analysis can be performed by considering the termination of each thread in isolation with respect to our abstract semantics.

Theorem 3. *Let P be a kernel program. If P terminates under the abstract semantics of Figure 4, then P terminates under the interleaving semantics of Figure 3.*

Proof. Given a thread t and a kernel state $\mathcal{K} = (\sigma_{\mathrm{A}}, K)$, define $\mathrm{abs}_t(\mathcal{K}) = K(t)$. Trivially, we have that $\mathrm{abs}_t(\mathcal{K}) = \mathrm{abs}_t(\mathcal{K}')$ if $\mathcal{K} \to_{\mathrm{k}} \mathcal{K}'$ by rule K-Step and when the rule *is not* instantiated by thread t. Moreover:

1. If $\mathcal{K} \to_{\mathrm{k}} \mathcal{K}'$ by rule K-Step and the rule *is* instantiated by thread t, then we have that $\mathrm{abs}_t(\mathcal{K}) \to_{\mathrm{a}} \mathrm{abs}_t(\mathcal{K}')$ by the abstract rule corresponding to thread-level rule employed in the application of K-Step. This follows once we observe that for any expression e, variable store σ_{v}, and array store σ_{A}, we have $[\![e]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} \in [\![e]\!]^{\sigma_{\mathrm{v}}}_{\star}$ (as $[\![A[e]]\!]^{\sigma_{\mathrm{v}}}_{\sigma_{\mathrm{A}}} \in \mathsf{Word}$).
2. If rule K-Barrier is applied, then there are two cases to consider, as t may either be at a barrier or may have terminated. If t is at a barrier, we have that $\mathrm{abs}_t(\mathcal{K}) \to_{\mathrm{a}} \mathrm{abs}_t(\mathcal{K}')$ by rule A-Barrier. This follows as $K(t) = (\sigma_{\mathrm{v,t}}, \mathrm{barrier}; ss_t)$ and $K'(t) = (\sigma_{\mathrm{v,t}}, ss_t)$. If t has terminated, we have that $\mathrm{abs}_t(\mathcal{K}) = \mathrm{abs}_t(\mathcal{K}')$, as $K(t) = K'(t)$.

We now prove the contrapositive of our theorem. Thus, suppose there exists a kernel program P which has an infinite execution under our interleaving semantics. Rule K-Step must fire infinitely often in the course of this execution, because if only rule K-Barrier would fire from some $\mathcal{K} = (\sigma_{\mathrm{A}}, K)$ onwards, then we have for at least one thread t that the sequence of statements ss_t in $K(t) = (\sigma_{\mathrm{v,t}}, ss_t)$ equals an infinite sequence of barrier statements, which is impossible as ss_t is finite.

Suppose that the infinite execution of P starts from $\mathcal{K}_0$. As the number of threads is finite, we have by the pigeon-hole principle that there exists at least one thread $t \in D$ for which rule K-Step fires an infinite number of times as part of the infinite execution. By the observations at the beginning of this proof, it is now immediate that there also exists an infinite abstract execution starting from $\mathrm{abs}_t(\mathcal{K}_0)$, and the theorem follows. □

```
if (tid = 0)
  { A[0] := 1; }
temp := 0;
while (temp < N) {
  if (tid = 0)
    { A[0] := A[0] + 1; }
  barrier;
  temp := A[0];
  barrier;
}
```

Figure 5: A kernel that cannot be shown terminating using Theorem 3

With the above theorem in hand, we can show that the kernel of Figure 2 terminates. As the variable *offset* is thread-private and as its value does not depend on any shared array access, its value is uniquely determined in the abstract semantics by the abstract expression evaluation function. More specifically, assuming no overflow occurs, the value of *offset* strictly increases during each loop-iteration, as the variable is initially set to a positive value and multiplied by 2 at the end of each iteration. Hence, as N is a private variable with a constant value, termination of the program under the abstract semantics follows, and by Theorem 3 the same holds true under the concrete semantics.

In general, the reverse of the above theorem does not hold: termination might depend on shared memory sub-expressions evaluating to specific values. Consider, e.g., the kernel of Figure 5, where we omit empty else-branches. Termination of this kernel depends on thread 0 assigning increasingly larger values to $A[0]$. However, in our abstract semantics, the value assigned to *temp* during each loop-iteration will be chosen nondeterministically and, hence, may always be chosen to be smaller than N. Although this is the case, we show experimentally in Section 6 that for the vast majority of kernels in open source benchmark suites, termination does not depend on the values of shared memory sub-expressions.

5. Implementation in KITTeL

The `KITTeL` termination analysis tool [4, 5] consists of a front-end, named `llvm2KITTeL`, which takes LLVM bitcode[2] and translates this into an *integer-based rewrite system*, and a back-end which tries to automatically prove termination of the generated rewrite system.

In *integer-based rewriting*, rules are of the following form:

$$f(x_1, \ldots, x_m) \to g(e_1, \ldots, e_n)\ [C]\,.$$

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

$$
\begin{array}{rll}
\text{start} & \to \text{loop\_head}(\mathit{tid}, 1) & [0 \leq \mathit{tid} < N] \\
\text{loop\_head}(\mathit{tid}, \mathit{offset}) & \to \text{loop\_body}_1(\mathit{tid}, \mathit{offset}) & [\mathit{offset} < N] \\
\text{loop\_head}(\mathit{tid}, \mathit{offset}) & \to \text{loop\_tail} & [\mathit{offset} \geq N] \\
\text{loop\_body}_1(\mathit{tid}, \mathit{offset}) & \to \text{if}_1(\mathit{tid}, \mathit{offset}) & [\mathit{tid} \geq \mathit{offset}] \\
\text{loop\_body}_1(\mathit{tid}, \mathit{offset}) & \to \text{loop\_body}_2(\mathit{tid}, \mathit{offset}, \mathit{temp}) & [\mathit{tid} < \mathit{offset}] \\
\text{if}_1(\mathit{offset}) & \to \text{loop\_body}_2(\mathit{tid}, \mathit{offset}, \mathit{temp}) & \\
\text{loop\_body}_2(\mathit{tid}, \mathit{offset}, \mathit{temp}) & \to \text{if}_2(\mathit{tid}, \mathit{offset}, \mathit{temp}) & [\mathit{tid} \geq \mathit{offset}] \\
\text{loop\_body}_2(\mathit{tid}, \mathit{offset}, \mathit{temp}) & \to \text{loop\_body}_3(\mathit{tid}, \mathit{offset}) & [\mathit{tid} < \mathit{offset}] \\
\text{if}_2(\mathit{tid}, \mathit{offset}, \mathit{temp}) & \to \text{loop\_body}_3(\mathit{tid}, \mathit{offset}) & \\
\text{loop\_body}_3(\mathit{tid}, \mathit{offset}) & \to \text{loop\_head}(\mathit{tid}, \mathit{offset} * 2) & \\
\text{loop\_tail} & \to \text{end} &
\end{array}
$$

Figure 6: The kernel from Figure 1 represented as an integer-based rewrite system

Here, f and g are function symbols, $x_1, \ldots, x_m$ are variables of type Int (the type of mathematical integers), $e_1, \ldots, e_n$ are expressions of type Int, and C is a Boolean expression. The expressions $e_1, \ldots e_n$, and C may contain variables of type Int; these variables may be fresh and need not come from $x_1, \ldots, x_m$. *Rewriting* $f(i_1, \ldots, i_m)$ to $g(j_1, \ldots, j_n)$ with $i_1, \ldots i_m, j_1, \ldots j_n \in \mathsf{Int}$ requires C to be satisfied. Specifically, there must exist a substitution σ mapping variables to mathematical integers such that $\sigma(f(x_1, \ldots, x_m)) = f(i_1, \ldots, i_m)$ and $\sigma(g(e_1, \ldots, e_n)) = g(j_1, \ldots, j_n)$, and such that $\sigma(C)$ holds. We refer the reader to [4] for further details; an example of an integer-based rewrite system is presented in Figure 6, and discussed below.

Changes to KITTeL. We adapted `llvm2KITTeL` to handle GPU kernels (compiled to LLVM bitcode by Clang[3]); 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: (i) we ensured that `llvm2KITTeL` abstracts loads even in the cases where it usually would 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 values may differ between loop iterations), and (iii) we made `llvm2KITTeL` aware of the fact that 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).

Given the above changes, `llvm2KITTeL` transforms the kernel from Figure 1 into the rewrite system of Figure 6, where we omit empty conditions. As in Section 3, *tid* and N correspond, respectively, to the CUDA built-in variables `threadIdx.x` and `blockDim.x`. Observe that the shared arrays `in` and

[3] `http://clang.llvm.org/`

`out` have been removed in accordance with Theorem 3. Barriers have also been removed, as they correspond to no-ops in our abstract semantics. The translation is otherwise straightforward. The top-most rule showcases the fact that not all variables need to occur on the left-hand side of a rule: *tid* only occurs on the right-hand side and in the condition $0 \leq tid < N$.

Although the changes made to `llvm2KITTeL` are straightforward, using the tool requires awareness of loop invariants and the limitations of mathematical integers.

Loop Invariants. To prove termination, it may be necessary to specify certain loop invariants—ideally automatically inferred by the employed termination tool [19]. For example, in the case of the kernel from Figure 1, proving termination requires one to observe that `offset` is always greater than 0.

In `KITTeL`, loop invariants can either be specified by hand, or inferred with the help of the Apron numerical abstract domain library [20]. We experimented with both strategies, as detailed in the next section. In the case of manually provided loop invariants, we ensured correctness by proving the validity of the invariants with the help of GPUVerify prior to running our experiments. In principle, we could extend GPUVerify's invariant inference engine to generate the required invariants automatically. However, this would require infrastructure to feed the generated invariants into `KITTeL`, which is currently lacking.

GPUVerify uses the Houdini algorithm [21] to infer invariants using a template-based approach [8, 22]. Candidate invariants are first guessed using a set of predefined templates, and an iterative process is then used to eliminate candidates that do not hold, converging on a set of mutually inductive invariants. As discussed in the next section, we have identified several classes of invariants that work well for termination analysis. From these classes, we could in principle derive a set of templates for GPUVerify's invariant inference engine, so as to automatically infer many of the loop invariants required for proving termination.

Bitvectors. Integer-based rewrite systems are defined over mathematical integers. This offsets them from the hardware that executes GPU kernels, which represents integers as finite bitvectors. As a result, we can construct kernels that can be shown to be terminating with `KITTeL`, but which will fail to terminate when executed on actual hardware, and vice versa. To guard against this, `llvm2KITTeL` offers the possibility to enrich the rewrite systems it generates with additional constraints that mimic bitvector behaviour [5]. We experimented with this option as detailed in the next section, although in line with most termination research we defaulted to using mathematical integers.

6. Evaluation

To evaluate the effectiveness of Theorem 3, we applied `KITTeL` to a suite of 604 OpenCL and CUDA kernels, 386 of which have loops. To demonstrate that our approach works out-of-the-box (bar the manual specification of loop invariants), with the kernel code itself being the only source of information about

Table 1: Observed thread counts for the kernels in our benchmark set

Thread count	1	2-100	101-10,000	10,001-1,000,000	> 1,000,000
Number of kernels	17	37	149	258	143

the presence of (potentially non-terminating) loops, we included the loop-free kernels in our evaluation. The kernels have on average 86 lines of code[4] and originate from nine sources:

- The *AMD Accelerated Parallel Processing SDK* v2.6 [23] (79 kernels, 55 of which have loops).
- The *NVIDIA GPU Computing SDK* v5.0 [24] (184 kernels, 110 of which have loops); we also included 8 kernels from v2.0 of the SDK, 7 of which have loops.
- The *C++ AMP Sample Projects* [25] hand-translated to CUDA (20 kernels, 16 of which have loops)
- The *gpgpu-sim* benchmarks [26] (33 kernels, 24 of which have loops)
- The *Parboil* benchmarks v2.5 [27] (25 kernels, 19 of which have loops)
- The *Rodinia* benchmark suite v2.4 [28] (40 kernels, 25 of which have loops)
- The *SHOC* benchmark suite [29] (87 kernels, 55 of which have loops)
- A set of kernels generated from the *PolyBench/C* benchmarks v3.2 [30] by the `PPCG` [31] parallel code generator (64 kernels, 49 of which have loops)
- Basemark's (previously Rightware's) *Basemark CL* v1.1 [32] (64 kernels, 26 of which have loops)

The kernel counts above do not include 4 kernels that we manually removed because they use CUDA surfaces [12], which we currently do not support. 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 except for LonestarGPU [33].

Thread Counts. Recall from Section 3 that the correctness of a kernel often depends on the number of threads that execute the kernel. In principle, this also applies to termination. Hence, stopping short of manually inspecting each kernel to see whether the thread count impacts termination, we took a pragmatic approach and ran every benchmark on a GPU to determine the number of threads each kernel is usually executed with. The obtained numbers were then used to fix the thread counts for the duration of our experiments.

[4]Counted using cloc 1.60 (`http://cloc.sourceforge.net`)

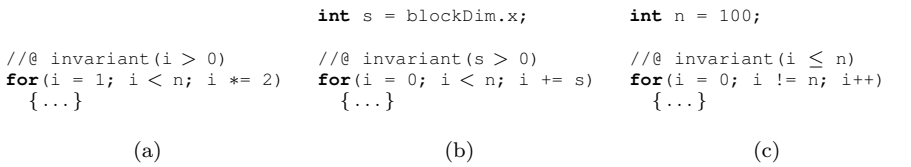

Figure 7: Code fragments exemplifying the forms of the invariants we specified manually

As can be seen in Table 1, the concrete thread counts for the kernels varied widely, with the majority of the kernels having a thread count of over 10,000. This underlines the need for a scalable thread-modular analysis: the state spaces that would need to be considered when modelling all threads explicitly would be exceedingly large.

Experimental Setup. All experiments were performed on a workstation with a 4-core Intel Xeon E5-2609 2.4GHz and 16GB of RAM running Ubuntu 14.04 and Clang/LLVM 3.5.1. The reported times are averages over five runs and include the time needed to compile a kernel to bitcode. The timeout used was 5 minutes.

We adapted `llvm2KITTeL` as described in the previous section and invoked the tool with the `-solver` option, which eliminates rewrite rules with unsatisfiable constraints as early as possible during the generation of a rewrite system, generally resulting in smaller systems. As default we used manually specified loop invariants and passed the `-increase-strength` option to `llvm2KITTeL`. The `-increase-strength` option turns left and right shifts into, respectively, multiplications and divisions by 2. This transformation facilitates termination analysis of GPU kernels where the loop counter is being shifted, which is common in kernels implementing reductions and prefix-sums. The underlying SMT solver used was Yices 2.3.0. Both `llvm2KITTeL`[5] and `KITTeL`[6] were downloaded on the 12th of February 2015.

Below, we first discuss the performance of the default setup over our benchmark set, and analyse the kernels for which termination could not be shown. Thereafter, we consider the impact of the `-increase-strength` option on the termination analysis, and the effectiveness of `KITTeL`'s automatic invariant inference capabilities based on Apron 0.9.10, which are not used by default. Finally, we consider `llvm2KITTeL`'s bitvector modelling capabilities.

Manually Specified Loop Invariants. We manually specified loop invariants for 53 kernels, as `KITTeL` would otherwise fail to prove termination of these kernels. The required invariants stated that (i) the loop counter is positive (32 kernels), that (ii) the variables defining the step value of the loop counter are positive

[5] `https://github.com/s-falke/llvm2kittel`
[6] `https://github.com/s-falke/kittel-koat`

(19 kernels), and that (iii) the loop counter is less than or equal to the value it is tested against for inequality in the loop guard (2 kernels). As mentioned in the previous section, we proved all of these invariants correct with the help of GPUVerify prior to running our experiments.

Figure 7 gives an example of each kind of manually specified invariant, using JML-style notation [34]. We note that the specified invariants provide sufficient information for `KITTeL` to be able to prove termination. To understand the examples, it is important to be aware that `KITTeL`'s termination analysis works on a loop-by-loop basis while *ignoring* variable initialisations that occur outside these loops. In the case of the loop of Figure 7a, this means that absence of the invariant would imply that `KITTeL` would analyse the loop assuming both $\texttt{i} > 0$ and $\texttt{i} \leq 0$ (for `i` is initialised outside the loop). However, in the latter case the loop counter would either decrease or remain zero, leading to nontermination. Similarly, for the loop of in Figure 7b, omitting the loop invariant would imply that `KITTeL` would assume `s` to have a completely arbitrary value. Hence, `i` would not necessarily increase with each iteration, again leading to nontermination. Finally, in the case of Figure 7c, absence of the invariant would imply that `KITTeL` could assume that $\texttt{i} > \texttt{n}$ on entry to the loop. As such, incrementing `i` would never yield the loop exit condition $\texttt{i} \mathrel{!}= \texttt{n}$.

6.1. Results for the Default Setup

Unsurprisingly, `KITTeL` managed to prove termination of each of the 218 loop-free kernels: for these kernels `llvm2KITTeL` always outputs the single-rule integer-based rewrite system

$$\text{start} \to \text{end}\,,$$

where loop-freeness of a kernel is determined by querying the control-flow graph of the kernel for cycles. On average termination of the loop-free kernels was shown in 0.40s, with the maximum time required being 0.56s.

The cumulative histogram of Figure 8 plots the analysis time for the 386 kernels *with* loops on a log-log scale. A point at (x, y) indicates that for x kernels with loops, termination could be shown within y seconds. Of the 386 kernels with loops, 348 could be shown to be terminating, with the analysis taking less than 1s in the case of 266 kernels. On average, termination was shown in 1.69s and the maximum time needed was 65.88s. Of the 38 kernels for which termination could not be shown, 34 reached the timeout of 5 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).

We manually inspected the 38 kernels to see why termination could not be shown. All 4 cases where `KITTeL` indicated nontermination would require reasoning over floating point numbers, which `KITTeL` does not support (see Figure 9a for an example, where termination follows as $\texttt{i} + \texttt{1.0} > \texttt{i}$ for small floating point numbers, assuming `i` is not updated in the loop body).

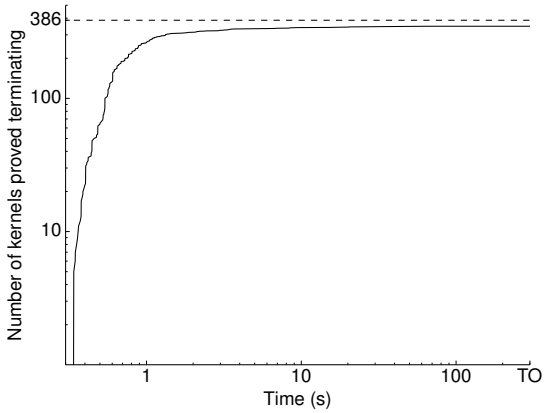

Figure 8: Cumulative histogram showing the time taken to prove termination of the 386 kernels with loops in the default setup

```
for(float i = 1.0; i < n; i += 1.0)
  {...}
```

(a)

```
while(atomicInc(p) < n)
  {...}
```

(b)

Figure 9: Loops that could not be shown terminating by `KITTeL`

In 4 cases, kernel termination depends on the behaviour of built-in atomic increment operations (see Figure 9b for an example, where `p` points to a shared memory word, and where termination follows as `atomicInc` returns monotonically increasing values, assuming the shared memory word pointed to by `p` is not updated in the loop body). Currently, `llvm2KITTeL` models atomic increment operations as returning unconstrained, arbitrary values and, hence, termination cannot be shown.

In 20 cases, termination would require reasoning about shared memory and, hence, the over-approximation from Theorem 3 is too coarse (see Figure 5 for an example). Of these 20 cases, there are 5 where the value tested against in the loop guard lives in shared memory, although this value is never changed during execution. In 11 cases, termination depends on some constant data in shared memory being of a specific form, e.g., a graph without cycles or a null-terminated string. In the 4 remaining cases, a thread may only terminate once it knows that all threads have finished performing a certain computation; this information is communicated through shared memory.

The above leaves 10 kernels, all timing out, with 5 timeouts in `llvm2KITTeL` and 5 in the `KITTeL` back-end. Of the 5 kernels timing out in `llvm2KITTeL`, 4 could be shown terminating by turning off the `-increase-strength` option (see below). For the remaining kernel timing out in `llvm2KITTeL`, no combination of options could be found that let `llvm2KITTeL` produce an integer-based rewrite system within the 5 minute timeout period. For the 5 kernels timing out in

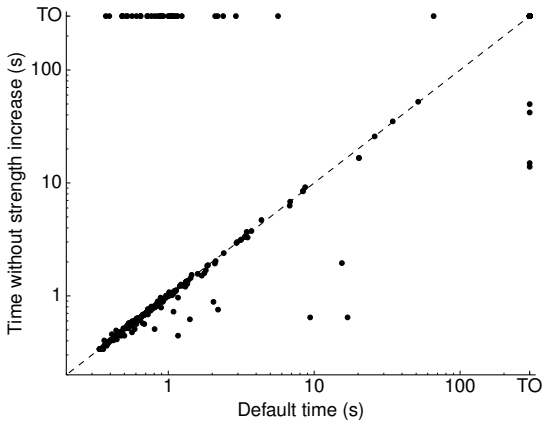

Figure 10: Scatter plot comparing the time taken to prove termination of kernels with loops in the default setup and with the `-increase-strength` option turned off

`KITTeL`, the generated rewrite systems could be shown terminating by hand, indicating that the analysis techniques used by `KITTeL` could be improved.

6.2. Impact of the Increase Strength Option

As the `-increase-strength` option does not change the shape of the control-flow graph of a kernel, turning off the option did not affect `KITTeL`'s ability to prove termination of all 218 loop-free kernels. Each loop-free kernel could still be shown terminating within 0.56s. For this reason we focus on kernels with loops in the remainder.

By default, `llvm2KITTeL`'s modelling of left and right shifts is imprecise, with the operations yielding unconstrained nondeterministic values. This is improved upon by the `-increase-strength` option by lifting left and right shifts to multiplication and division by 2, which `llvm2KITTeL` models more accurately. Hence, we hypothesised that turning off the `-increase-strength` option would (i) cause `KITTeL` to no longer be able to show termination of kernels that use shifts to update loop counters, leading instead to timeouts due to the use of unconstrained nondeterministic values, and would (ii) reduce analysis time for kernels that do not use shifts to update loop counters, as the less accurate modelling generally yields integer-based rewrite systems that still terminate, but which have fewer constraints.

The scatter plot of Figure 10 compares the default setup with the setup that has the `-increase-strength` option turned off. A point at (x, y) indicates that the analysis took x seconds in the default setup vs. y seconds with the option turned off. Points lying below the diagonal thus indicate cases where it was beneficial to disable the `-increase-strength` option. The axes are plotted using log scales.

In the case of 41 kernels where termination could be shown in the default setup, timeouts occurred once the `-increase-strength` option was turned off.

These 41 kernels are *precisely* those kernels in our benchmark set that use shifts to update loop counters, confirming our hypothesis that termination of these kernels can no longer be shown.

In the case of 4 kernels, the opposite happened: although termination could not be shown in the default setup, turning the `-increase-strength` option off allowed `KITTeL` to prove termination. Closer inspection revealed that the 4 kernels implement cryptographic hash functions (variants of the MD5 and SHA-256 algorithms), which extensively use shifts, but not to update loop counters. The less precise modelling caused `llvm2KITTeL` to generate fewer constraints and allowed it to produce integer-based rewrite systems, where it was not able to do so before (see above).

Disregarding the 41 kernels for which termination could no longer be shown once the `-increase-strength` option was turned off, the Wilcoxon signed ranks test [35] ($T = 19250$, $n = 345$, $p < 0.05$) indicated that there is a significant difference in running time with and without the option. The sum of the negative difference ranks ($\sum R_- = 22366$) was larger than the sum of positive difference ranks ($\sum R_+ = 19250$), indicating that less running time was required when the `-increase-strength` option was turned off. Moreover, the effect size for the matched-pair samples was 0.31.[7] Hence, our analysis provides some evidence that running time improves for kernels that do not use shifts to update loop counters when `-increase-strength` is turned off, confirming our second hypothesis.

The above suggests it would be beneficial to refine the `-increase-strength` option by introducing a static analysis that determines whether a shift is used to update a loop counter. This would enable `llvm2KITTeL` to selectively transform shifts based on their use.

6.3. Automatic Invariant Inference

To experiment with the automatic inference of invariants, we removed all manually specified invariants from our kernels and turned on invariant inference in `KITTeL`. As in the case of the previous two experiments, `KITTeL` managed to prove termination of all 218 loop-free kernels, where the maximum time required was again 0.56s. Hence, we again ignore all loop-free kernels in the remainder.

As `KITTeL` does not know a priori which kernels require inference of loop invariants, it indiscriminately applies Apron to each kernel it is asked to prove termination of. Hence, as invariant inference takes time and as each inferred invariant increases the size of constraints in the integer-base rewrite system, we hypothesised that inferring invariants would significantly increase the overall running time. Moreover, Apron is only able to infer loop invariants of the first two forms we discussed above (positive loop counter and positive step value), and we thus also hypothesised that `KITTeL` would not be able to prove termination

[7] The effect size [35] determines the degree of association, similar to a correlation coefficient, and ranges between 0 and 1. Conventions define the effect size to be small = 0.10, medium = 0.30, or large = 0.50.

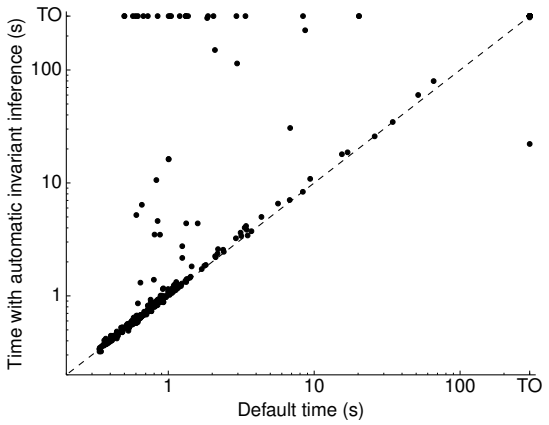

Figure 11: Scatter plot comparing the time taken to prove termination of kernels with loops in the default setup and with automatic invariant inference

of the 2 kernels requiring an invariant of the third form (loop counter less than or equal to the value it is tested against for inequality).

The scatter plot of Figure 11 compares the default setup (with manually supplied invariants) and the setup with automatic invariant inference. A point at (x, y) indicates that the analysis took x seconds in the default setup vs. y seconds with automatic invariant inference. Thus, a point lying above the diagonal denotes a benchmark for which the default options led to faster analysis than when automatic invariant inference was turned on. The axes are plotted using log scales.

In the case of 15 kernels that did not require manually supplied invariants, `KITTeL` failed to prove termination within the 5 minute timeout period once automatic invariant inference was turned on, while it was able to show termination in the default setup. In 45 of the 53 cases where manual invariants were required, `KITTeL` managed to prove termination once automatic inference was turned on. In 2 of the remaining 8 cases, `KITTeL` managed to infer the correct invariant but timed out nevertheless. The remaining 6 kernels included the 2 kernels requiring an invariant stating that the loop counter is less than or equal to the value it is tested against for inequality, confirming our second hypothesis. In the case of the other 4 kernels, it is not clear to us why `KITTeL` failed to infer the correct invariants. This might possibly be due to limitations of Apron, but giving a definitive answer to this question requires a more thorough understanding of Apron than we currently have.

In one case termination could be shown once automatic invariant inference was turned on, while `KITTeL` was not able to show termination of the kernel in either the default setup or in the `-increase-strength` experiment, while the invariants we added manually to this kernel should have sufficed. Hence, the additional inferred invariants simplified the task of proving termination.

Disregarding the 2 kernels that required invariants Apron is not capable

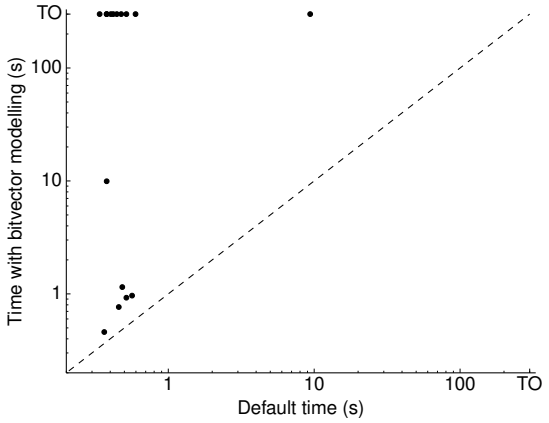

Figure 12: Scatter plot comparing the time taken to prove termination of kernels with loops in the default setup and with bitvector modelling (C++ AMP Sample Projects only)

of generating, the Wilcoxon signed ranks test ($T = 8365$, $n = 384$, $p < 0.05$) indicated that there is a significant difference in running time with and without automatic inference. The sum of the positive difference ranks ($\sum R_+ = 47915$) was larger than the sum of negative difference ranks ($\sum R_- = 8365$), indicating that more running time was required once automatic inference was turned on. Moreover, the effect size for the matched-pair samples was 0.67. Hence, our analysis provides evidence that running time suffers when automatic invariant inference is used, confirming our first hypothesis. Note, however, that the times reported for the default setup did not include the time an expert needed to manually provide the necessary invariants. On the other hand, the invariants inferred by Apron were not checked for correctness, while the manually provided invariants were.

6.4. Impact of Bitvector Modelling

For our final experiment we considered the bitvector modelling mode of `llvm2KITTeL` [5]. In this mode `llvm2KITTeL` generates additional constraints mimicking bitvector behaviour. As we expected the additional constraints to cause significant slowdown, we only considered the 20 kernels from the C++ AMP Sample Projects collection (16 of which have loops), which in the default setup could all be shown to be terminating *without* the help of manually supplied invariants.

As in our previous experiments, `KITTeL` was able to prove termination of all loop-free kernels within 0.56s. Hence, we focus on the 16 kernels with loops in the remainder.

The scatter plot of Figure 12 compares the default setup and the setup with bitvector modelling. A point at (x, y) indicates that the analysis took x seconds in the default setup vs. y seconds with bitvector modelling. A point lying above

the diagonal thus indicates that the analysis is faster in the default setup. The axes are plotted using a log scale.

Of the 16 kernels with loops, `KITTeL` only managed to prove termination of 6 within the 5 minute timeout period, and in *each case* proving termination was *slower* than in the default setup, as we expected.

We manually inspected the remaining 10 kernels to see whether `KITTeL` should in principle be able to prove termination, or whether this is hindered by the bitvector modelling (e.g., due to the occurrence of bitvector overflows). In 9 cases we did not find any reason why termination should not be provable, and we attribute the occurrence of timeouts to the increased number of constraints in the generated rewrite system. In the remaining case, a loop bound was an input parameter to the kernel and the loop counter would always be incremented by 2. This means that, if the loop bound would be chosen equal to the maximum integer value, termination would not occur. Hence, the difference between bitvectors and mathematical integers is not just a theoretical oddity, but shows up in concrete edge cases.

7. Related Work

Although the literature on termination analysis is vast, only limited effort has been directed towards proving termination of multi-threaded programs. A technique for proving termination of individual threads is described in [36]. Whole program termination of multi-threaded programs using atomic operations is considered in [37, 38], and termination of programs consisting of "concurrent objects" (an instance of the actor model) is discussed in [39, 40]. The current paper is the first to consider whole program termination for multi-threaded programs (kernels) where the main synchronisation mechanism is barrier synchronisation.[8]

The technique described in [38] assumes that a counterexample to termination exists and does case splitting over the counterexample using a set of proof rules. A case is eliminated when it is found to be contradictory, and termination is shown when no cases remain. In contrast, each of the other aforementioned techniques, including the one described in this paper, can be seen as an instance of *rely/guarantee reasoning* [41]. In [36] threads are abstracted in an environment that can be relied upon to behave in a certain way by the thread whose termination one wants to prove. A proof rule that facilitates proving termination in a rely/guarantee setting is discussed in [37]. In our case, the rely/guarantee conditions are extremely weak: a read from shared memory cannot be relied upon by a thread to yield any specific value, and no thread guarantees anything about the values it writes to shared memory. This is similar to the technique described in [39], which loses all the information about the shared

[8] A model of a single CUDA kernel is mentioned in [38]. Unfortunately, the model does not take into account thread identifiers and allows for threads to ignore certain barriers. Hence, termination of the model does not reflect termination of the original kernel.

state at each scheduling point. The technique described in [40] improves upon [39] by introducing rely/guarantee conditions that establish that shared data is only changed a finite number of times during program execution.

Besides being an instance of rely/guarantee reasoning, our method for proving termination is also related to the technique that underpins the soundness of GPUVerify. Similar to our method, the technique implemented in GPUVerify depends on a shared state abstraction. The abstraction allows race-freedom to be proved by considering *two* arbitrary threads [8, 9].

Our choice for `KITTeL` was motivated by the fact that Clang/LLVM is currently the only freely available compiler that is able to handle CUDA and OpenCL kernels. Hence, not wanting to spend significant effort developing a new compiler font-end, this limited us to termination analysis tools that accept LLVM bitcode as input. Of these there are currently two: `KITTeL` and AProVE [3]. The latter is not open source and, hence, could not easily be adapted along the lines described in Section 5.

AProVE is also able to handle the integer-based rewrite systems generated by `llvm2KITTeL`. However, experimenting with this combination revealed that the systems generated by `llvm2KITTeL` are in general not of the kind AProVE is tuned towards, resulting in poor performance. Tailoring `llvm2KITTeL` more towards AProVE is virtually impossible, as there is no publicly available information detailing the kinds of systems for which AProVE does have satisfactory performance. For these reasons, we did not consider the combination of `llvm2KITTeL` and AProVE in our experimental evaluation. Other tools that are able to prove termination of integer-based rewrite systems, such as Ctlr [42], generally do not allow for fresh variables to occur on the right-hand sides of rewrite rules—which we use to model our shared state abstraction—and, hence, are not suitable for our purposes.

8. Conclusion and Future Work

We have presented a thread-modular technique for proving termination of massively parallel GPU kernels. The technique reduces the termination problem for these kernels to a sequential termination problem by abstracting the shared state. Implementing the technique in `KITTeL`, we were able to prove termination of 94% of the kernels in our benchmark set, and of 90% of the kernels with loops. Our experience shows that the excellent progress the rewriting community has made in relation to sequential termination analysis can be directly leveraged to enable efficient termination analysis in the emerging domain of GPU programming—an encouraging result.

Experimental Conclusions. The effectiveness of `KITTeL` [4, 5] is partially due to the replacement of shifts by multiplications and divisions. Without these replacements the percentage of successful termination proofs drops to 88%, or to 81% for the kernels with loops. Our experiments suggest that we can obtain a higher success rate by making the replacements more selective, only replacing those shifts that affect loop counters.

All above percentages are based on kernels for which we manually specified loop invariants. Usability-wise it would be much preferred to automatically infer invariants. `KITTeL` supports this approach through its use of the Apron numerical abstract domain library [20]. However, as our experiments show, the use of this library increases running time, and it is not always the case that all necessary invariants are inferred. To limit these effects, `KITTeL` could potentially benefit from a bespoke invariant inference engine that solely generates invariants facilitating termination proofs (compared to Apron which is fairly generic in its aims). A bespoke engine should infer at least the invariants of Figure 7. A selected number of other invariants should possibly also be inferred, because these may help improve the chances of proving termination, as one kernel in our inference experiment indicates.

Although `KITTeL` is highly effective, the tool defaults to modelling integers as mathematical integers. Hence, there is a mismatch with actual hardware, which represents integers as bitvectors. Enabling bitvector modelling in `KITTeL` is possible, and this shows that the difference between integers and bitvectors is not just theoretical, but that concrete kernels exist for which the behaviour is different. Unfortunately, performance of bitvector modelling is currently poor, and further research is needed to make it perform as well as the approach using mathematical integers.

Future Work. The current main shortcoming of our approach is the very coarse shared state abstraction where accessing the shared state yields completely non-deterministically selected values. Hence, besides the future work suggested by our experiments, and as discussed above, we would like to investigate more precise shared state abstractions. In particular, we would like to investigate abstractions which model certain properties of constant data living in shared memory, e.g., by automatically identifying the constant data and making it private. This should facilitate termination proofs for kernels where either the constant value tested against in a loop guard lives in shared memory, or where the constant data is required to be of a specific form. Relatedly, we would like to apply our prior work on *barrier invariants* [43] to capture invariants of the shared state that are key to establishing termination; this will potentially enable proving termination of kernels where values communicated between threads during kernel execution affect termination.

We would also like to investigate abstractions that are able to handle the atomic increment operations supported by CUDA and OpenCL (see Figure 9b). Termination of kernels using these operations could potentially be established by modelling these operations as returning monotonically increasing values; an idea also applied to the reasoning about data race-freedom of such kernels [44].

Finally, we would like to implement the described approach to proving of GPU kernels in other termination analysis tools to see how these compare with `KITTeL`. A prime candidate here is T2 [45], as `llvm2KITTeL` has been extended with the ability to generate input for this tool [46]. A comparison with the Arctor tool from [38] would also be interesting, as this tool currently seems to be the only tool that offers truly scalable termination analysis for concurrent

programs. A significant difficulty with this latter comparison will be that Arctor accepts neither kernel sources nor LLVM bitcode as input and, hence, a front-end would need to be developed to be able to experiment with a benchmark suite of the size we considered in the current paper.

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 both him and Marc Brockschmidt for answering questions regarding the tool.

References

[1] A. Podelski, A. Rybalchenko, Transition invariants, in: Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004), 2004, pp. 32–41.

[2] B. Cook, A. Podelski, A. Rybalchenko, Termination proofs for systems code, in: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, 2006, pp. 415–426.

[3] T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, Proving termination and memory safety for programs with pointer arithmetic, in: Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR 2014), Vol. 8562 of Lecture Notes in Computer Science, 2014, pp. 208–223.

[4] S. Falke, D. Kapur, C. Sinz, Termination analysis of C programs using compiler intermediate languages, in: Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA 2011), 2011, pp. 41–50.

[5] S. Falke, D. Kapur, C. Sinz, Termination analysis of imperative programs using bitvector arithmetic, in: Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2012), Vol. 7152 of Lecture Notes in Computer Science, 2012, pp. 261–277.

[6] J. E. Cates, A. E. Lefohn, R. T. Whitaker, GIST: An interactive, GPU-based level set segmentation tool for 3D medical images, Medical Image Analysis 8 (2004) 217–231.

[7] M. J. Harris, Fast fluid dynamics simulation on the GPU, in: GPU Gems, Addison-Wesley, 2004, Ch. 38, pp. 637–665.

[8] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, J. Wickerson, The design and implementation of a verification technique for GPU kernels, ACM Transactions on Programming Languages and Systems 37 (3) (2015) 10:1–10:49.

[9] P. Collingbourne, A. F. Donaldson, J. Ketema, S. Qadeer, Interleaving and lock-step semantics for analysis and verification of GPU kernels, in: Proceedings of the 22nd European Symposium on Programming (ESOP 2013), Vol. 7792 of Lecture Notes in Computer Science, 2013, pp. 270–289.

[10] J. Ketema, A. F. Donaldson, Automatic termination analysis for GPU kernels, in: Proceedings of the 14th International Workshop on Termination (WST 2014), 2014, pp. 1–5.

[11] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, J. Wickerson, GPU concurrency: Weak behaviours and programming assumptions, in: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015), 20015, pp. 577–591.

[12] Nvidia, CUDA C programming guide, version 5.0 (2012).

[13] Khronos OpenCL Working Group, The OpenCL specification, version 1.2 (2012).

[14] P. M. Kogge, H. S. Stone, A parallel algorithm for the efficient solution of a general class of recurrence equations, IEEE Transactions on Computers C-22 (8) (1973) 786–793.

[15] N. Chong, A. F. Donaldson, J. Ketema, A sound and complete abstraction for reasoning about parallel prefix sums, in: Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014), 2014, pp. 397–410.

[16] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd Edition, The MIT Press, 1999.

[17] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Transactions on Computers C-28 (9) (1979) 690–691.

[18] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, S. Qadeer, Engineering a static verification tool for GPU kernels, in: Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014), Vol. 8559 of Lecture Notes in Computer Science, 2014, pp. 226–242.

[19] M. Heizmann, J. Hoenicke, J. Leike, A. Podelski, Linear ranking for linear lasso programs, in: Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis, Vol. 8172 of Lecture Notes in Computer Science, 2013, pp. 365–380.

[20] B. Jeannet, A. Miné, Apron: A library of numerical abstract domains for static analysis, in: Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009), Vol. 5643 of Lecture Notes in Computer Science, 2009, pp. 661–667.

[21] C. Flanagan, K. R. M. Leino, Houdini, an annotation assistant for ESC/Java, in: Proceedings of the 2001 International Symposium of Formal Methods Europe (FME 2001), Vol. 2021 of Lecture Notes in Computer Science, 2001, pp. 500–517.

[22] A. Betts, N. Chong, P. Deligiannis, A. F. Donaldson, J. Ketema, Implementing and evaluating candidate-based invariant generation, CoRR abs/1612.01198, http://arxiv.org/abs/1612.01198.

[23] AMD, AMD Accelerated Parallel Processing (APP) SDK, http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/.

[24] Nvidia, CUDA code samples, https://developer.nvidia.com/cuda-code-samples.

[25] 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.

[26] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, T. M. Aamodt, Analyzing CUDA workloads using a detailed GPU simulator, in: Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009), 2009, pp. 163–174.

[27] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, W.-M. W. Hwu, Parboil: A revised benchmark suite for scientific and commercial throughput computing, Tech. Rep. IMPACT-12-01, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign (2012).

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC 2009), 2009, pp. 44–54.

[29] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, J. S. Vetter, The scalable heterogeneous computing (SHOC) benchmark suite, in: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU 2010), 2010, pp. 63–74.

[30] L.-N. Pouchet, PolyBench/C: The polyhedral benchmark suite, http://www.cs.ucla.edu/~pouchet/software/polybench/.

[31] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, F. Catthoor, Polyhedral parallel code generation for CUDA, ACM Transactions on Architecture and Code Optimization 9 (4) (2013) 54:1–54:23.

[32] Basemark, Basemark CL, `http://www.basemark.com/product-catalog/basemark-cl/`.

[33] M. Burtscher, R. Nasre, K. Pingali, A quantitative study of irregular programs on GPUs, in: Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC 2012), 2012, pp. 141–151.

[34] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, W. Dietl, JML reference manual (draft) (2013).

[35] G. W. Corder, D. I. Foreman, Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach, Wiley, 2009.

[36] B. Cook, A. Podelski, A. Rybalchenko, Proving thread termination, in: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), 2007, pp. 320–330.

[37] C. Popeea, A. Rybalchenko, Compositional termination proofs for multi-threaded programs, in: Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012), Vol. 7214 of Lecture Notes in Computer Science, 2012, pp. 237–251.

[38] A. Kupriyanov, B. Finkbeiner, Causal termination of multi-threaded programs, in: Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014), Vol. 8559 of Lecture Notes in Computer Science, 2014, pp. 814–830.

[39] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, Cost analysis of concurrent OO programs, in: Proceedings of the 9th Asian Symposium on Programming Languages and Systems (APLAS 2011), Vol. 7078 of Lecture Notes in Computer Science, 2011, pp. 238–254.

[40] E. Albert, A. Flores-Montoya, S. Genaim, E. Martin-Martin, Termination and cost analysis of loops with concurrent interleavings, in: Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA 2013), Vol. 8172 of Lecture Notes in Computer Science, 2013, pp. 349–364.

[41] C. B. Jones, Tentative steps toward a development method for interfering programs, ACM Transactions on Programming Languages and Systems 5 (4) (1983) 596–619.

[42] C. Kop, N. Nishida, Constrained term rewriting tool, in: Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-20), Vol. 9450 of Lecture Notes in Computer Science, 2015, pp. 549–557.

[43] N. Chong, A. F. Donaldson, P. H. J. Kelly, J. Ketema, S. Qadeer, Barrier invariants: A shared state abstraction for the analysis of data-dependent GPU kernels, in: Proceedings of the 28th ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA 2013), 2013, pp. 605–622.

[44] E. Bardsley, A. F. Donaldson, Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels, in: Proceedings of the 6th International NASA Formal Methods Symposium (NFM 2014), Vol. 8430 of Lecture Notes in Computer Science, 2014, pp. 230–245.

[45] B. Cook, A. See, F. Zuleger, Ramsey vs. lexicographic termination proving, in: Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013), Vol. 7795 of Lecture Notes in Computer Science, 2013, pp. 47–61.

[46] M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, N. Piterman, T2: temporal property verification, in: Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016), Vol. 9636 of Lecture Notes in Computer Science, 2016, pp. 387–393.