Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

# Termination analysis for GPU kernels ☆

Jeroen Ketema *, Alastair F. Donaldson *

*Department of Computing, Imperial College London, London, United Kingdom*

ARTICLE INFO

*Article history:*
Received 22 May 2016
Received in revised form 19 January 2017
Accepted 27 April 2017
Available online xxxx

*Keywords:*
Termination
Abstraction
GPUs
Concurrency

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.



## 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.

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 com-

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

* Corresponding authors.
*E-mail addresses:* jketema@imperial.ac.uk (J. Ketema), afd@imperial.ac.uk (A.F. Donaldson).

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

http://dx.doi.org/10.1016/j.scico.2017.04.009
0167-6423/

puted 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 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 Fig. 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.

```
__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();
    }
}
```

**Fig. 1.** The Kogge–Stone prefix-sum in CUDA.

Please cite this article in press as: J. Ketema, A.F. Donaldson, Termination analysis for GPU kernels, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.04.009