Checking Non-Interference in SPMD Programs

Stavros Tripakis Christos Stergiou
University of California, Berkeley
chster,stavros@eecs.berkeley.edu

Roberto Lublinerman
Pennsylvania State University
rluble@psu.edu

Abstract

We study one of the basic multicore and GPU programming models, namely, SPMD (Single-Program Multiple-Data) programs. We define a formal model of SPMD programs based on interleaving threads that manipulate global and local arrays, and synchronize via barriers. SPMD programs are written with the intention to be deterministic, although programming errors may result in this not being true. SPMD programs are also frequently modified toward optimal performance. These facts motivate the need for methods to check determinism and program equivalence. A key property in achieving this is non-interference. We formulate non-interference as validity of logical formulas automatically derived from the program, we show that non-interference implies determinism, and we report on a prototype that can prove non-interference of NVIDIA CUDA programs.

1 Introduction

The goal of this paper is to develop methods that help programmers build correct multi-threaded programs, and in particular programs running on modern *graphics processing units* (GPUs). GPUs enjoy great popularity today, as a result of offering great computing power at relatively low cost [11]. Motivated by this, we consider the CUDA programming model [1], used in NVIDIA's GPUs.

CUDA is based on the Single Program, Multiple Data (SPMD) parallel computation paradigm, where concurrent threads execute the same code, although they may not follow exactly the same execution path. CUDA is free from some of the plagues of parallel programming: for instance, it does not provide *locks* explicitly (although it does provide *barrier synchronization*). However, CUDA programming is still difficult because of another reason: a "naive" parallel implementation of a given algorithm is in most cases non-optimal in terms of run-time, i.e., the program runs too slow. Because of this, a significant effort is spent trying to optimize the program to achieve better performance [11]. This is done by exploiting the particularities of the architecture. Although no general rule exists, it is often the case that global-memory accesses are very expensive and thus need to be reduced to a minimum so that they do not create a bottleneck. Moreover, memory bandwidth often depends on how memory is accessed, that is, on the memory access *patterns*. Subtle modifications in such patterns can result in orders-of-magnitude performance improvements [11, 1].

Optimizing the program is done by transforming it so that it uses the specifics of the underlying platform optimally. Currently, these transformations are done "manually", since automating them is beyond the reach of state-of-the-art compilers. Although methodologies and guidelines exist to help programmers (e.g., *coalesced* global memory access [11, 1]), these are fairly general and leave a large gap which must be filled by the programmer's creativity and care. This is a difficult and error-prone task (a simple example is provided in this paper).

Our ultimate goal is to develop methods and tools to make this task error-free. In particular, methods that allow the programmer to check *equivalence* of two programs: the program before the transformation and the one after the transformation.

After studying publicly available CUDA programs [1], it has come to our attention that these programs are written to be *deterministic*, in the sense that their final result does not depend on the interleaving order. It is not surprising for programmers to want to write deterministic programs. However, determinism by no means comes for free in CUDA. It is achieved by ensuring that concurrent threads are *non-interfering*. Non-interference roughly states that different threads access different array elements, or the same element but at different times.

The main contribution of this paper is a method to automatically check non-interference in SPMD programs. The method relies on checking validity of logical formulas that can be automatically derived from the program. Non-interference is shown to be a sufficient condition for determinism. We also report on a prototype tool that checks non-interference of CUDA programs. [13] is an extended version of this report that includes ideas on checking equivalence.

2 Related work

There is a large body of research on checking correctness of parallel programs (see [13] for references). Most of this research, however, deals with quite general versions of the verification problem, in terms of either the model used (e.g., threads synchronizing with *locks*), or the properties to be checked. In contrast, the SPMD model we use in this paper is restricted, e.g., there are no locks, only barrier synchronization, and we focus on a specific property: non-interference.

The *interference-free property* used in the proof framework of [9] is weaker than ours. Ours essentially guarantees absence of *races*, where two or more threads access the same memory location and at least one access is a write. Races have been heavily studied in the context of programs with synchronization mechanisms such as locks. Many techniques to detect races that are not "protected" by locks have been proposed, both static (e.g., see [2, 8]) and dynamic (e.g., see [12]). [7] observes that this notion of races does not capture all problematic interactions among threads, and proposes the stronger non-interference property of *atomicity*, in the context of ConcurrentJava [2]. The fact that many parallel programs are written to be deterministic has been observed by other researchers as well (e.g., see [10]). Currently, attempts are being made to bring determinism to mainstream object-oriented languages (e.g., Deterministic Parallel Java [4]).

Checking nondeterminacy has been considered in [6], but the model and method used there are different from ours. Their method is based on building ordering graphs from the synchronization primitives.

Non-interference has received a lot of attention in the parallel compilation community, in particular under the general problem of data dependency analysis for arrays (e.g., see [14]). The major difference of this body of work with ours is that, in parallel compilation, the problem is how to *extract* parallelism from a sequential piece of code (with loops manipulating arrays, etc.), whereas here, the parallelization has been performed by the programmer, and our objective is to prove that the parallel code is non-interfering.

[3] proposes a method to check the barrier-based synchronization patterns of SPMD programs. Incorrect barrier synchronization may occur when barriers are executed conditionally. This problem does not arise in our model where barriers are assumed to be unconditional.[1]

3 SPMD programs

Our model of SPMD programs is inspired by CUDA [1]. A SPMD program is defined to be a tuple $P = (G, L, F)$. G is a list of *global array names*, each with a type and size. L is a list of *local array names*, each with a type and size. F is an automaton formalizing the thread function of the program: $F = (Q, q_0, R)$. Q is a finite set of *locations* (the "control states" of the automaton). $q_0 \in Q$ is the *initial* location. R is a set of *program transitions*. A program transition is a tuple (q, q', α), also denoted $q \xrightarrow{\alpha} q'$, where $q, q' \in Q$ are the source and destination locations, respectively, and α is either a *condition statement*, an *assignment statement*, or the special `sync` *statement*, as described below. A program transition labeled with a condition (resp., assignment) statement is called a condition (resp., assignment) transition. A program transition labeled with `sync` is called a `sync` transition. Our model does not contain explicit local (i.e., per thread) variables, but these can be easily modeled using local arrays.

Let us provide an example of a SPMD program. This example models an array reversal program. We first model a naïve version of the program as a tuple $P_1 = (G, L^1, F^1)$, with $G = \{A[\mathsf{C}\cdot\mathsf{T}], B[\mathsf{C}\cdot\mathsf{T}]\}$, $L^1 = \emptyset$ (no local arrays), and F^1 being the automaton shown in Figure 1 (top). $A[\mathsf{C}\cdot\mathsf{T}]$ denotes an array of length $\mathsf{C}\cdot\mathsf{T}$ (in this case both arrays A and B are unidimensional). C and T are parameters, representing the number of processing cores and number of threads per core, respectively. This program implements the parallel assignment $B[i] := A[M-1-i]$, for $i = 0$ to $M-1$, where $M = \mathsf{C}\cdot\mathsf{T}$. Index i is implemented by the expression $\mathsf{T}\cdot\mathsf{b}+\mathsf{t}$. b is a parameter representing the index of the core that a given thread is running on: it ranges from 0 to $\mathsf{C}-1$. t represents the local index of a thread in its core: it ranges from 0 to $\mathsf{T}-1$.

[1] [1] states that "`__syncthreads()` is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise execution is likely to hang or produce unintended side effects." Conditional barriers appear in only 3 out of 57 examples included in the CUDA SDK.

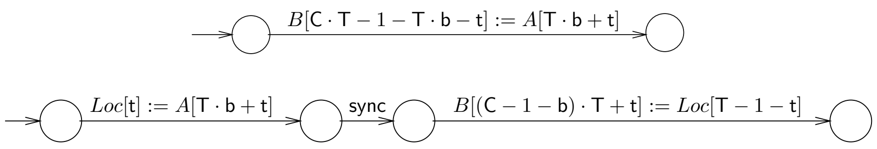

$B[\mathsf{C}\cdot\mathsf{T}-1-\mathsf{T}\cdot\mathsf{b}-\mathsf{t}] := A[\mathsf{T}\cdot\mathsf{b}+\mathsf{t}]$
$Loc[\mathsf{t}] := A[\mathsf{T}\cdot\mathsf{b}+\mathsf{t}]$
sync
$B[(\mathsf{C}-1-\mathsf{b})\cdot\mathsf{T}+\mathsf{t}] := Loc[\mathsf{T}-1-\mathsf{t}]$

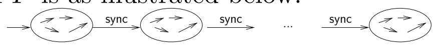
Figure 1: Thread automata F^1 (top) and F^2 (bottom).

A second, optimized version of the program can be modeled as a tuple $P_2 = (G, L^2, F^2)$, with G same as for P_1, $L^2 = \{Loc[\mathsf{T}]\}$, and F^2 being the automaton shown in Figure 1 (bottom).

It is not at all trivial to see that the alternative implementation is equivalent to the original implementation of array reversal, that is, produces the same output array B for any input array A. Our ultimate goal is to devise methods to check that the two SPMD programs are indeed equivalent.

SPMD programs are given interleaving semantics of threads communicating through global and local arrays, and synchronizing via barriers. The semantics are defined for fixed positive integer values C and T of parameters C and T, respectively. Given a SPMD program P, and given C and T, the semantics, denoted $[\![P, C, T]\!]$, is a labeled transition system (LTS) $(S, S_0, \rightarrow)$ where: S is the set of *states*, recording the values of all elements in the local and global arrays, plus the program counters (locations) of all threads; S_0 the set of *initial states*, and $\rightarrow$ is the *transition relation* (generally non-deterministic because of interleaving). Due to lack of space, we omit the details and refer the reader to [13].

We assume that F is deterministic, deadlock-free, and acyclic (loops are handled by our tool as discussed in Section 6). We also assume that the structure of F is as illustrated below:



That is, F is a *chain of k sub-automata*, linked with sync transitions. Each sub-automaton, called a *sync-segment*, has no sync transition. In the examples of Figure 1, F^1 consists of a single sync-segment since it contains no sync statement. F^2 consists of two sync-segments.

4 Determinism

Our ultimate goal is to prove equivalence of SPMD programs. But what does equivalence exactly mean? For sequential programs, which are deterministic, it is reasonable to define equivalence as follows: programs P_1 and P_2 are equivalent if, given the same inputs, they produce the same outputs. This definition does not directly apply to SPMD programs, because the latter are inherently non-deterministic: the outputs of a SPMD program may be different depending on thread interleavings. We are thus motivated to define determinism.

Before doing so, however, we must also define precisely what we mean by "inputs" and "outputs". Usually, in GPU applications, one is not interested in the values of local arrays or other local variables, but only in the values of global arrays. Motivated by this, we say that two runs ρ_1 and ρ_2 in $[\![P, C, T]\!]$ are *equivalent*, denoted $\rho_1 \approx \rho_2$, if, assuming all global arrays have the same value when the programs begin, they will have the same value when the programs end. The two runs are said to be *strongly equivalent*, denoted $\rho_1 \simeq \rho_2$, if, assuming they start at the same state, they end up at the same state.

P is said to be *deterministic with respect to* C, T if for any two runs ρ and ρ' in $[\![P, C, T]\!]$, we have $\rho \approx \rho'$. If $\rho \simeq \rho'$ then P is said to be *strongly deterministic with respect to* C, T. P is said to be *deterministic* (respectively, *strongly deterministic*) if it is deterministic (respectively, strongly deterministic) with respect to C, T, for any $C, T \in \mathbb{N}$.

5 Non-Interference

In the system $[\![P, C, T]\!]$, there are $C \cdot T$ threads running, where C is the number of cores and T the number of threads per core. All these threads may access the same locations of global memory. Moreover, for each core, the T threads running on that core may access the same location of local memory of this core. To ensure determinism, we need to ensure that no *race conditions* occur in these global or local memory accesses. Race conditions can occur when two threads access the same memory location, at least one access is a write, and the two accesses may happen in any order. Non-interference ensures that race conditions do not occur.

Let F be the thread automaton on which we wish to ensure absence of race conditions. Because F is structured in segments, it suffices to ensure absence of races separately on each sync-segment F_i of F. Indeed, threads must synchronize on sync transitions, thus it is impossible for two sync-segments F_i, F_j with $i \neq j$ to interfere: if $i < j$ then, in any execution, all transitions of F_i are guaranteed to take place before any transition of F_j.

Thus, it suffices to check, for each sync-segment F_i of F, that we cannot have two threads executing statements of F_i that interfere with each other. F_i is a special thread automaton (without sync), so let $F_i = (Q, q_0, R)$. We define the following sets of expressions: $\mathsf{LHS}(F_i)$, called the set of all *left-hand side* expressions of F_i, is defined to be the set of all expressions l such that $l := e$ is some assignment statement of F_i. $\mathsf{RHS}(F_i)$, called the set of all *right-hand side* expressions of F_i, is defined to be the set of all array sub-expressions of an expression e, such that either $l := e$ is some assignment statement of F_i or e is some condition statement of F_i. An *array sub-expression* of e is a sub-expression of e which is also an array expression. For example, if $e = A[3 + B[\mathsf{t}]]$ then e has two array sub-expressions: e itself and $B[\mathsf{t}]$.

LHS only contains array expressions, since, by definition, in every assignment $l := e$, l is an array expression. The reason we include only array expressions in RHS is because only array expressions can be assigned to, thus, only such expressions can interfere with each other. Although we could have included all sub-expressions in RHS without affecting the results given below, this would result in redundant expressions in RHS. Note that LHS and RHS are finite sets.

Let us illustrate LHS and RHS on our running example. First, consider F^1 (Figure 1, top). F^1 has a single sync-segment: F^1 itself. We have:

$$\begin{aligned} \mathsf{LHS}(F^1) &= \{B[\mathsf{C}\cdot\mathsf{T} - 1 - \mathsf{T}\cdot\mathsf{b} - \mathsf{t}]\} \\ \mathsf{RHS}(F^1) &= \{A[\mathsf{T}\cdot\mathsf{b} + \mathsf{t}]\}. \end{aligned}$$

Next, consider F^2 (Figure 1, bottom). F^2 consists of two sync-segments: $F^2 = F^2_1 \to F^2_2$. We have:

$$\begin{aligned} \mathsf{LHS}(F^2_1) &= \{Loc[\mathsf{t}]\}, \\ \mathsf{RHS}(F^2_1) &= \{A[\mathsf{T}\cdot\mathsf{b} + \mathsf{t}]\}, \\ \mathsf{LHS}(F^2_2) &= \{B[(\mathsf{C} - 1 - \mathsf{b})\cdot\mathsf{T} + \mathsf{t}]\}, \\ \mathsf{RHS}(F^2_2) &= \{Loc[\mathsf{T} - 1 - \mathsf{t}]\}. \end{aligned}$$

We next define two set of *potentially interfering expression pairs* of F_i. The set $\mathcal{E}_g(F_i)$ is defined to be the set of all (e_1, e_2) such that there exists global array symbol $A \in G$ such that $A[e_1] \in \mathsf{LHS}(F_i)$ and $A[e_2] \in \mathsf{LHS}(F_i) \cup \mathsf{RHS}(F_i)$. The set $\mathcal{E}_l(F_i)$ is defined to be the set of all (e_1, e_2) such that there exists local array symbol $B \in L$ such that $B[e_1] \in \mathsf{LHS}(F_i)$ and $B[e_2] \in \mathsf{LHS}(F_i) \cup \mathsf{RHS}(F_i)$. The intuition is that two threads interfere iff there exists a pair of potentially interfering expressions (e_1, e_2) such that e_1 and e_2 evaluate to the same value in the two threads. Notice that we need not worry about expressions of the form $A[e_1] \in \mathsf{LHS}(F_i)$ and $B[e_2] \in \mathsf{LHS}(F_i) \cup \mathsf{RHS}(F_i)$, where A and B are different array symbols. This is because, even if e_1 and e_2 can be made equal, A and B refer to different locations in memory, thus, there is no possibility for races.

Fix $C, T \in \mathbb{N}$. We say that *sync-segment F_i is non-interfering with respect to C, T* if

1. for every $(e_1, e_2) \in \mathcal{E}_g(F_i)$, the following formula is valid:

$$\begin{aligned} &\forall b_1, b_2 \in \{0, ..., C-1\}, \forall t_1, t_2 \in \{0, ..., T-1\} : \\ &(b_1 \neq b_2 \vee t_1 \neq t_2) \Rightarrow e_1(C, T, b_1, t_1) \neq e_2(C, T, b_2, t_2) \end{aligned}$$

2. for every $(e_1, e_2) \in \mathcal{E}_l(F_i)$, the following formula is valid:

$$\begin{aligned} &\forall b \in \{0, ..., C-1\}, \forall t_1, t_2 \in \{0, ..., T-1\} : \\ &\quad t_1 \neq t_2 \Rightarrow e_1(C, T, b, t_1) \neq e_2(C, T, b, t_2) \end{aligned}$$

The above formulas are formulas of first-order logic with equality, with array symbols considered to be unary function symbols. For an expression e, $e(C, T, ...)$ denotes the expression obtained by replacing variables $\mathsf{C}, \mathsf{T}, ...$ in e by concrete values $C, T, ...$

We say that *F is non-interfering with respect to* C, T if for all $i \in \{1, ..., k\}$, F_i is non-interfering with respect to C, T. We say that F_i *is non-interfering* if it is non-interfering with respect to C, T for all $C, T \in \mathbb{N}$. We say that F *is non-interfering* if for all $i \in \{1, ..., k\}$, F_i is non-interfering.

Theorem 1 *Let $P = (G, L, F)$ be a SPMD program and let $C, T \in \mathbb{N}$. If F is non-interfering w.r.t. C, T then P is strongly deterministic with respect to C, T.*

Let us apply Theorem 1 to show that the SPMD program of Figure 1 (top) is deterministic. The sets $\mathsf{LHS}(F^1)$ and $\mathsf{RHS}(F^1)$ have been given above. According to the definition above, $\mathcal{E}_g(F^1) = \{(e, e)\}$, where e is $\mathsf{C}\cdot\mathsf{T} - 1 - \mathsf{b}\cdot\mathsf{T} - \mathsf{t}$, and $\mathcal{E}_l(F^1) = \emptyset$. To show non-interference, we must prove that for all $C, T \in \mathbb{N}$, for all $b_1, b_2 \in \{0, ..., C-1\}$ and for all $t_1, t_2 \in \{0, ..., T-1\}$ such that $b_1 \neq b_2$ or $t_1 \neq t_2$, the following inequality holds: $C\cdot T - 1 - (b_1\cdot T + t_1) \neq C\cdot T - 1 - (b_2\cdot T + t_2)$. This follows directly from the assumptions. Similarly, we can show that the alternative array-reversal program P_2 with thread automaton F^2 is also non-interfering. F^2 consists of two sync-segments, F_1^2 and F_2^2. Following the definitions, we get: $\mathcal{E}_g(F_1^2) = \emptyset$, $\mathcal{E}_l(F_1^2) = \{(\mathsf{t},\mathsf{t})\}$, $\mathcal{E}_g(F_2^2) = \{(e,e)\}$, where e is $(\mathsf{C} - 1 - \mathsf{b}) \cdot \mathsf{T} + \mathsf{t}$, and $\mathcal{E}_l(F_2^2) = \emptyset$. Then, to prove that F^2 is non-interfering, we show the two facts: $t_1 \neq t_2 \Rightarrow t_1 \neq t_2$, and $(b_1 \neq b_2 \vee t_1 \neq t_2) \Rightarrow (C-1-b_1) \cdot T + t_1 \neq (C-1-b_2) \cdot T + t_2$.

It is instructive to consider a third implementation of array reversal, which does not satisfy the non-interference property. This happens if we remove the sync statement from thread automaton F^2: call the resulting thread automaton F^3. F^3 has a single sync-segment (itself) and we have: $\mathsf{LHS}(F^3) = \{Loc[\mathsf{t}], B[(\mathsf{C}-1-\mathsf{b}) \cdot \mathsf{T} + \mathsf{t}]\}, \mathsf{RHS}(F^3) = \{A[\mathsf{b} \cdot \mathsf{T} + \mathsf{t}], Loc[\mathsf{T} - 1 - \mathsf{t}]\}$. Then, $\mathcal{E}_l(F^3)$ includes the pair $(\mathsf{t}, \mathsf{T}-1-\mathsf{t})$ and we can no longer prove the implication $t_1 \neq t_2 \Rightarrow t_1 \neq T-1-t_2$. In fact, the implication can be shown to be false simply by setting $t_1 = 0$ and $t_2 = T-1$. Thus, F^3 is interfering. In fact, this implementation is non-deterministic and incorrect.

6 Tool and experiments

We have built a prototype tool that can automatically check non-interference in CUDA programs. This functionality is not available in other tools, as far as we know. The tool uses CIL (http://hal.cs.berkeley.edu/cil/) to parse and analyze CUDA programs. The tool then generates non-interference conditions that are submitted to the Yices SMT solver (http://yices.csl.sri.com/). Yices cannot handle non-linear constraints, therefore, in expressions such as $b \cdot T + t$, we instantiate T to a constant. Our tool can handle multidimensional arrays.

At present our tool can run on the `reverse1`, `reverse2` programs presented in this paper and on the following programs from the CUDA SDK suite [1]: `clock`, `nbody`, `simpleZeroCopy` and `transpose`. All these programs are proved non-interfering completely automatically in < 1 sec. Our tool currently handles loops with statically known bounds by *unrolling* the loop, an approach that does not extend to programs with unknown loop bounds. We are currently extending our method to handle more loops, by generating a variant of non-interference condition that guarantees absence of interference at *any* iteration of a loop. It is worth noting that such formulas are guaranteed to be quantifier-free [13].

7 Conclusions, future work

We described a method and tool for checking non-interference and proving determinism in SPMD/CUDA programs. Future work includes: strengthening our tool so that it can handle a larger set of CUDA programs; implementing methods to check program equivalence [13]; enhancing the method with elements from the theory of arrays [5].

References

[1] NVIDIA CUDA Programming Guide Version 2.0, 6/7/2008. At http://www.nvidia.com/cuda.

[2] M. Abadi, C. Flanagan, and S. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[3] A. Aiken and D. Gay. Barrier inference. In *POPL'98*.

[4] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *HotPar'09*.

[5] A. Bradley, Z. Manna, and H. Sipma. What's decidable about arrays. In *VMCAI, LNCS 3855*, pages 427–442. Springer, 2006.

[6] P. Emrath and D. Padua. Automatic detection of nondeterminacy in parallel programs. *SIGPLAN Not.*, 24(1):89–99, 1989.

[7] C. Flanagan and S. Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, 2003.

[8] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV* 2007.

[9] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[10] M. Rinard. Analysis of multithreaded programs. In *SAS*, volume 2126 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[11] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *PPOPP 2008*.

[12] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[13] S. Tripakis, C. Stergiou, and R. Lublinerman. Checking Equivalence of SPMD Programs Using Non-Interference. Tech. Rep. UCB/EECS-2010-11, UC Berkeley, Jan 2010. At http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-11.html.

[14] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley, 1996.