Checking Equivalence of SPMD Programs Using Non-Interference

Stavros Tripakis
Christos Stergiou
Roberto Lublinerman

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-11
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-11.html

January 29, 2010

Checking Equivalence of SPMD Programs Using Non-Interference*

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

Roberto Lublinerman
Pennsylvania State University
rluble@psu.edu

January 29, 2010

This work is dedicated to the memory of Amir Pnueli.

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 us to develop methods to check determinism and equivalence. A key property in achieving this is non-interference, formulated as validity of logical formulas automatically derived from the program, that imply determinism. Automatically derived post-conditions can be used to check equivalence of non-interfering programs. We report on a prototype that can prove non-interference of NVIDIA CUDA programs.

1 Introduction

Writing correct programs has always been difficult, and a large part of computer science research is devoted in developing methods to assist programmers in this task. Recently, the surge of parallel computing architectures such as *multicores* has brought with it hopes to go beyond the limits of Moore's law, but also worries that programming will become harder [5].

One of the reasons why parallel programming is difficult, is that parallel architectures often use a multi-threaded, shared-memory, interleaving-based programming model. This results in inherently non-deterministic behavior, which is hard to understand and debug. This has led some researchers to claim that threads should be avoided [23, 19]. Other concurrency models, such as Kahn Process Networks [16], ensure deterministic results despite process interleaving. Unfortunately, most multiprocessor architectures widely used today do not follow such models, and use threads instead. What is worse, the semantics of these architectures are often ambiguous and not well-documented, and execution sometimes yields unexpected results [29].

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), such as the NVIDIA

*This report is an updated version of [20], with major addition Section 7, reporting on a prototype implementation and providing preliminary ideas on how to handle loops. Thanks to Carlos Coelho for useful discussions. Part of this work was done at Cadence Research Labs. This work is supported by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U.S. Army Research Office (ARO #W911NF-07-2-0019), the U.S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, Lockheed-Martin, National Instruments, Thales and Toyota.

GeForce 8 Series. GPUs enjoy great popularity today, as a result of offering great computing power at relatively low cost [28]. Motivated by this, we consider the CUDA programming model [2], used in NVIDIA's GPUs.

CUDA is based on the Single Program, Multiple Data (SPMD) parallel computation model, 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*). On the other hand, GPU programming is 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., runs too slow. Thus, a significant effort is spent trying to optimize the program to achieve better performance [28]. 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 [28, 2].

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 [28, 2]), 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).

In this paper we propose methods 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. This does not guarantee correctness of the programs *per se*. However, the original program is usually a straightforward parallel implementation of the algorithm, thus, it is easier to check that this original program is correct. Then, checking equivalence is enough to guarantee correctness of the optimized versions as well.[1]

After studying publicly available CUDA programs [2], 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*, in terms of the variables they read and write. Non-interference turns out to be a key property in our study, the main contributions of which are the following:

First, we introduce a simple formal model for SPMD programs. Second, we formally define determinism and equivalence of such programs. Third, we propose a formal notion of non-interference and show that it is a sufficient condition for determinism. Fourth, we propose a method to check equivalence of non-interfering programs. Our definition of non-interference, as well as the method to check equivalence, rely on checking validity of logical formulas that can be automatically derived from the program. Finally, we illustrate our methods throughout the paper using a parallel program performing array inversion, coming from the CUDA benchmark suite.

2 Related work

Checking program equivalence for sequential programs has been studied in [31, 12]. There is a large body of research on checking correctness of parallel programs (e.g., see [14, 21, 24, 17, 18, 22] and [27] for a survey of more recent work). In fact, much of the work in formal verification techniques such as *model checking* [26, 9] has been in part motivated by the additional complexity due to concurrency.

Most of this research, however, deals with quite general versions of the verification problem, in terms of either the model used (for instance, general threads synchronizing with *locks* or similar mechanisms), or the properties that need to be checked (which can be specified, for instance, using some general formalism such as

[1] Similar equivalence checking methods are part of the standard flow in circuit design, as well as in more recent methodologies such as *model-based design* (e.g., refining a Simulink floating-point model to a fixed-point model).

temporal logic [21]). In contrast, the SPMD model we use in this paper is restricted (for instance, there are no locks, only barrier synchronization), and we focus on specific properties: non-interference, determinism and equivalence.

The *interference-free property* used in the proof framework of [24] 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 [3, 15]) and dynamic (e.g., see [30]). [11] 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 [3]. The fact that many parallel programs are written to be deterministic has been observed by other researchers as well (e.g., see [27]). Currently, attempts are being made to bring determinism to mainstream object-oriented languages (e.g., Deterministic Parallel Java [6]).

[32, 25, 34] study verification of MPI programs [1]. MPI is based on message-passing communication and is thus different from the SPMD model that we study in this paper, which uses shared memory. [32] are interested in checking equivalence of a parallel MPI program with a sequential program. Here we are interested in checking equivalence of parallel SPMD programs.

Non-interference is a prominent notion in computer security (e.g., see [13]), but the interpretation there is usually that information does not flow from confidential data to public data.

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 [35]). 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.

In model-checking, there is a large body of work on how to alleviate *state-explosion*, by eliminating redundant interleavings using *partial-order reduction* (e.g., see [33, 10]), or by exploiting *symmetries* (e.g., see [8]). However, the goal there is not to use non-interference to statically ensure determinism and equivalence.

[4] 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.[2]

3 Background: the CUDA programming model

There are obviously many different types of concurrent programs, depending on the parallel architectures that these programs are meant to run upon, and the programming model that they use. In order to facilitate understanding of the formal model we present in Section 4, we provide here a short description, with examples, of the CUDA model, which has motivated this work.

Parallel architecture:

CUDA programs are meant to run on a GPU, which typically consists of a *host*, which is a traditional CPU, and one or more compute *devices*, which are massively data-parallel co-processors. Each device consists of a set of *cores* plus some *global memory*, which can be accessed by all cores. Each core consists of a processing element (a processor) plus some *local memory*.

CUDA programs:

We consider in this paper a simple class of CUDA, where a program consists of three parts:

```
global array declarations;
thread function declaration;
thread spawning;
```

[2] [2] states that "`__syncthreads()` is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code 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.

The first part consists of a list of declarations of arrays. Memory for these arrays is to be allocated into the global memory space of the multi-core device. The second part declares a thread function, to be executed by each thread that will be spawned on the device. Each thread function is a piece of sequential code similar to a C function. The third part consists of a command specifying how many threads to spawn. CUDA programs are more general, in the sense that they are general sequential programs (e.g., written in C) where thread spawning commands can appear anywhere in the code. For simplicity, in this paper we consider the restricted class above.

An example of a CUDA program is given below.

```
// global array declaration
float A[1024], B[1024];

// thread function declaration
void reverse1(float* Out, float* In, int M)
{
  int i = noThreads * coreId + threadId;
  Out[M-1-i] = In[i];
}

// thread spawning
reverse1 <<<1024>>> (B, A, 1024);
```

The program declares two arrays `A` and `B`: they are to be stored in the global memory of the device. Then the thread body is declared as function `reverse1`, where: argument `M` is the size of the arrays; `noThreads`, `coreId` and `threadId` are parameters (number of threads per core, core id, and thread id, respectively), to be instantiated upon execution, for each thread.[3] The third part of the program specifies that $K = 1024$ threads must be spawned. Conceptually, a programming model such as CUDA gives the programmer the impression that the number of available cores is unbounded, thus, K can be arbitrarily large. If the number of threads per core (`noThreads`) is T, then conceptually $C = \lceil \frac{K}{T} \rceil =$ `noCores` cores are required to run the program.[4]

Transforming the program to optimize performance:

The array reversal application can be rewritten as follows (for simplicity, we assume $K = C \cdot T$):

```
void reverse2(float* Out, float* In)
{
  float Loc[noThreads];
  int i = coreId * noThreads + threadId;
  int j = (noCores-1-coreId) * noThreads + threadId;
  int k = noThreads - 1 - threadId;
  Loc[threadId] = In[i];
  __syncthreads();
  Out[j] = Loc[k];
}
```

The main idea is to split the tasks performed by the threads in two phases. In Phase 1, threads read from the input global array and store values in a local-memory array `Loc`. A separate instance of `Loc` is allocated at

[3] In CUDA, a core is called a *block* and the set of cores is called a *grid*. Blocks are conceptually arranged in the grid as a one-, two-, or three-dimensional array. Thus, the index of a block can be up to 3-dimensional: parameters `blockIdx.x`, `blockIdx.y`, etc., are used for this purpose. Similarly, threads in a block are also conceptually arranged in 1D, 2D, or 3D arrays. This facilitates programming with 2D and 3D objects that are frequently used in computer graphics. For simplicity, we consider single-dimensional indices in this paper. However, our approach directly extends to multi-dimensional indices as well.

[4] In practice, the number of cores in a given device may be smaller than $\lceil \frac{K}{T} \rceil$. Different policies could be used in such a case. One such policy is to partition the set of threads into groups, such that each group is enough to run on the available set of cores. Then the groups are executed in sequence.

each core. In Phase 2, threads copy from `Loc` to the output global array, and in the process of doing so also reverse the order of the values. The `syncthreads` (barrier synchronization) command ensures that Phase 1 is complete when Phase 2 starts. The new program achieves better performance than the first version, because threads access global memory (array `B`) in a so-called *coalesced* manner: see [28, 2] for details.

Looking at `reverse2`, it is not immediately obvious that it correctly implements array-reversal, or in other words, that it is equivalent to the "naive" version `reverse1`. In fact, even in such a simple application, the indices `i,j,k` used by `reverse2` are sufficiently complex to require time to understand the logic behind the rewriting. This process is tedious and error-prone. The goal of this paper is to provide tools to ensure that nothing goes wrong, that is, that `reverse2` is equivalent to `reverse1`.

4 A formal model of SPMD programs

In this section we provide a formal model for SPMD programs. This model, although inspired by the CUDA programming model, is independent and can be used in other similar contexts as well. For reasons of simplicity in exposition, our formal model makes a number of assumptions, such as acyclicity of programs (no loops). Loops are handled by our implementation as discussed in Section 7.

A SPMD program is defined to be a tuple

$$P = (G, L, F)$$

where 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, as described below. A type is a basic type such as boolean, integer, real. The size of an array A, denoted $\mathsf{sz}(A)$, is an arithmetic expression involving constants or special pre-defined *parameters* C (number of processing *cores*) and T (number of threads per core). (In the CUDA code shown in Section 3, C and T are represented by `noCores` and `noThreads`, respectively.) Given an array symbol A, the size of A is denoted $\mathsf{sz}(A) \in \mathbb{N}$.

The automaton F modeling a thread function is a tuple

$$F = (Q, q_0, R)$$

where 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*, or 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.

Note that although our model does not contain explicit local (i.e., per thread) variables, these can be easily modeled using local arrays.

An *expression* can be of the following forms: a *constant*, such as 0, 1.5, *true*, and so on; one of the pre-defined parameters C, T, b (representing the index of the core that a given thread is running on, and ranging from 0 to $\mathsf{C}-1$) and t (representing the local index of a thread in its core, and ranging from 0 to $\mathsf{T}-1$); an *arithmetic expression* of the form $e+e'$, $e-e'$, etc.; a *boolean expression* of the form $e > e'$, $e \wedge e'$, etc.; or an *array expression* of the form $A[e]$, where A is an array name in G or L, and e is an arithmetic expression of integer type. In the CUDA code shown in Section 3, b and t are represented by `coreId` and `threadId`, respectively.

A *condition statement* is a boolean expression. An *assignment statement* has the form $l := e$, where e is an expression and l is an array expression.

Let us provide an example of an SPMD program. This example models an array reversal program. We first model the "naive" version of the program (with function "reverse1", see Section 3) 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

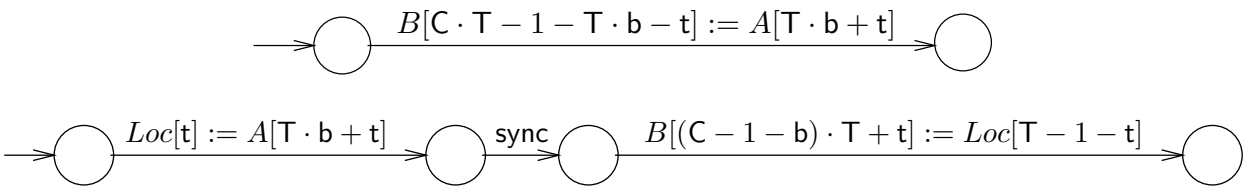

Figure 1: Thread automata F^1 (top) and F^2 (bottom).

unidimensional). The 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 $\mathsf{T}\cdot\mathsf{b}+\mathsf{t}$.

A second, optimized version of the program (using function "reverse2", see Section 3) 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 goal in this paper is to devise methods to check that the two SPMD programs are indeed equivalent.

Structural Assumptions:

Let $P = (G, L, F)$ be a SPMD program. We assume that F is *deterministic*, that is, there is no location $q \in Q$ such that q has more than one outgoing assignment transitions, or both assignment transitions and condition transitions.

We also assume that F is *structurally deadlock-free*, i.e., for every location q, if all outgoing program transitions from q are labeled with conditions, then the union of these conditions is equivalent to *true*.

We also assume that F is *acyclic*, i.e., there is no sequence of program transitions leading from a given location q to itself. This and the fact that Q is finite implies that some locations will have no outgoing program transitions. We call these locations *final*. We can assume, without loss of generality, that there is a single final location.

We finally assume that the structure of F is as illustrated in Figure 2, namely, F is a *chain of k sub-automata*, linked with sync transitions. We denote this as $F = F_1 \rightarrow F_2 \rightarrow \cdots \rightarrow F_k$. Each sub-automaton F_i has no sync transition. Also, each F_i where $i < k$ has a unique location q_s and a unique sync transition $(q_s, q'_s, \mathsf{sync})$, such that q'_s is the "initial" location of F_{i+1}. We call each F_i a *sync-segment*. 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.

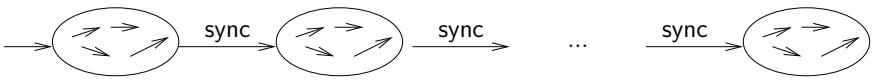

Figure 2: Structure of a thread automaton.

We classify global and local array symbols as *input* or *output*. A global array A is an *input array* if it is not written at all in F, that is, there is no assignment transition in F of the form $A[e] := e'$. A is an *output array* if it is not read at all in F, i.e., there is no assignment transition in F of the form $e := e'$ such that A appears in e'. We assume that all global arrays can be classified either as input arrays, or as output arrays, but not both. A local array B is classified as input or output with respect to a given sync-segment F_i: B is an input array in F_i if B is not written at all in F_i; B is an output array in F_i if it is not read at all in F_i. For checking equivalence, we will assume that a local array B can be either an input array or an output array in F_i, but not both. Note that B can still be an input array in F_i and an output array in F_j if $j \neq i$. Also note that we do not need the above assumption for checking non-interference (Section 6). For example, in both F^1 and F^2 (Figure 1) global array A is an input array and global array B is an output array. Local

array *Loc* is an input array in the first sync-segment of F^2 and an output array in the second sync-segment of F^2.

Instantiation and Semantics:

A SPMD program, although it refers to parameters C and T, does not instantiate these parameters. Indeed, in principle, a SPMD program should be written independently of the actual values of these parameters, and should work correctly in any instantiation. An *instance* of a SPMD program $P = (G, L, F)$ is represented as a tuple (P, C, T), where $C, T \in \mathbb{N}$ are two positive integers, representing the instantiation of parameters C and T, respectively. In a SPMD program instance (P, C, T), there are in total $C \cdot T$ threads running concurrently, each executing the sequential program described in F. The set $\{0, 1, ..., C \cdot T - 1\}$ is called the set of *global thread indices* for (P, C, T), denoted $\mathcal{I}(C, T)$.

In a SPMD program instance (P, C, T), there is a single copy of every global array $A \in G$. On the other hand, each local array $B \in L$ is instantiated C times, representing the fact that there is one copy of B allocated at each core. Elements of an array A are indexed from 0 to $\mathsf{sz}(A, C, T) - 1$, where $\mathsf{sz}(A, C, T)$ is the integer number obtained by replacing, in $\mathsf{sz}(A)$, C by C and T by T, and evaluating the resulting expression.

Consider an expression e: it generally involves global or local memory array symbols, constants, and the parameters $\mathsf{C}, \mathsf{T}, \mathsf{b}, \mathsf{t}$. By fixing these parameters to concrete positive integer values C, T, b, t, we get a *parameter-free* expression, that is, an expression involving only array symbols and constants. We denote this parameter-free expression, obtained by substituting concrete values to the parameters, by $e(\mathsf{C}/C, \mathsf{T}/T, \mathsf{b}/b, \mathsf{t}/t)$, or $e(C, T, b, t)$ in short.

The semantics of a SPMD program instance (P, C, T), denoted $[\![P, C, T]\!]$, is defined to be a *labeled transition system* (LTS) $[\![P, C, T]\!] = (S, S_0, \rightarrow)$ where:

S is the set of *states*. Each state $s \in S$ is a partial function that assigns a value to each element of every global array $A \in G$, to each element of every instance B^k of every local array $B \in L$, where $k \in \{0, 1, ..., C-1\}$, and to every instance of a *program counter* variable $\mathsf{pc}^t \in Q$, which records the location of thread n, where $n \in \mathcal{I}(C, T)$. States are partial functions because some arrays may not be initialized (however, we will enforce initialization below). We will denote by $s(v), s(\mathsf{pc}^n), s(A[i])$, etc., the values of v, pc, $A[i]$, etc., in state s. If $s(\mathsf{pc}^n)$ is the final location for all $n \in \mathcal{I}(C, T)$, then s is called a *final* state.

$S_0 \subseteq S$ is the set of *initial* states. For each $s \in S_0$, and for every $n \in \mathcal{I}(C, T)$, we have: $s(\mathsf{pc}^n) = q_0$. Also, every array in G (resp., L) assumes one of the possible values in G_0 (resp., L_0). Array elements can have arbitrary initial values, however, we will assume that local arrays are guaranteed to be initialized during execution (see below).

$\rightarrow$ is a set of labeled transitions. Each transition is a triplet (s_1, β, s_2), also denoted $s_1 \xrightarrow{\beta} s_2$, where s_1, s_2 are states and β is either sync, or a pair (n, α), where $n \in \mathcal{I}(C, T)$ and α is a condition or an assignment.

Given a state s and a parameter-free expression e, $s(e)$ denotes the value of e at state s: this is the value obtained by replacing all sub-expressions $A[j]$ of e by $s(A[j])$ and performing any arithmetic or logical operations in e. We say that a boolean expression e is satisfied at s if $s(e)$ evaluates to *true*.

When e is not parameter-free, its evaluation generally depends not only on the state s, but also on the global thread index $n \in \mathcal{I}(C, T)$ (as well as the values C and T, of course). Given $n \in \mathcal{I}(C, T)$, define $\mathsf{b}(n)$ and $\mathsf{t}(n)$ to be the quotient and the remainder of the division $\frac{n}{T}$, respectively:

$$n = \mathsf{b}(n) \cdot T + \mathsf{t}(n). \tag{1}$$

Let us now define the rules for the transitions of the LTS $[\![P, C, T]\!]$. First, consider the case $s_1 \xrightarrow{n,\alpha} s_2$, where $n \in \mathcal{I}(C, T)$. In this case we adopt the usual *interleaving* semantics, where only thread n moves and all other threads remain at the same location. Let $\mathsf{pc}_i = s_i(\mathsf{pc}^n)$, for $i = 1, 2$. Then, F must have a program transition $(\mathsf{pc}_1, \mathsf{pc}_2, \alpha)$ and one of the following must hold:

- Either α is a condition e and the parameter-free expression $e(C, T, \mathsf{b}(n), \mathsf{t}(n))$ is satisfied at state s_1. In this case, the values of all variables, except pc^n, remain the same in s_2 as in s_1.
- Or α is the assignment $l := e$ where l is some array expression $A[e']$. Let $v = s_1(e(C, T, \mathsf{b}(n), \mathsf{t}(n)))$, i.e., v is value that e assumes in s_1 when evaluated by thread n. Let $j = s_1(e'(C, T, \mathsf{b}(n), \mathsf{t}(n)))$. Then s_2 is identical to s_1, except that the value of array element $A[j]$ in s_2 is set to v, and $s_2(\mathsf{pc}^n) = \mathsf{pc}_2$.

Second, consider the case $s_1 \xrightarrow{\mathsf{sync}} s_2$. In this case all threads synchronize and move simultaneously. Then, F must have a transition $(\mathsf{pc}_1, \mathsf{pc}_2, \mathsf{sync})$, such that $\forall n \in \mathcal{I}(C,T) : s_1(\mathsf{pc}^n) = \mathsf{pc}_1 \wedge s_2(\mathsf{pc}^n) = \mathsf{pc}_2$. (Notice that, because we assume that the thread automaton is a "chain" of sub-automata linked by sync transitions, it is not possible for different threads to synchronize while being at different locations.) The value of all other variables except program counters remains unchanged.

Our semantics assumes that assignments are *atomic*, that is, they cannot be interrupted by other threads. This assumption may seem unrealistic, especially in cases where the expressions involved in the assignment are long (i.e., require many computation steps), involve accesses to global memory, etc. It is true that in such cases execution of these assignments may not be atomic. This problem can be overcome, however, during the modeling phase: thread automata are only a modeling formalism, not a programming language. When translating from a programming language (e.g., such as CUDA) to thread automata, care can be taken to "split" non-atomic statements into sequences of atomic ones.

A *run* in the LTS $[\![P,C,T]\!]$ is a sequence of $k \geq 0$ transitions, starting at an initial state:

$$\rho = s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_k} s_k,$$

where $s_0 \in S_0$, and $s_i \xrightarrow{\beta_i} s_{i+1}$ is a transition of $[\![P,C,T]\!]$, for $i = 0, ..., k-1$. We say that the run ρ *reaches* state s_k and s_k is called a *reachable* state. The set of all reachable states of P with respect to C,T is denoted $\mathcal{R}(P,C,T)$. The run ρ is called *maximal* if s_k is a final state. The set of all reachable final states of P with respect to C,T is denoted $\mathcal{R}_f(P,C,T)$. Note that every run in $[\![P,C,T]\!]$ is finite: this follows from the assumption that F is acyclic. On the other hand, the sets $\mathcal{R}(P,C,T)$ and $\mathcal{R}_f(P,C,T)$ may be infinite, because the domains of state variables (arrays) may be infinite.

Assignment Assumptions:

Let $\rho = s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_k} s_k$ be a run in the LTS $[\![P,C,T]\!]$ and let A be a global or local array of P. Let $i \in \{1, ..., k\}$ and let $j \in \{0, ..., \mathsf{sz}(A,C,T) - 1\}$. We say that the j-th element of A is *written* in the i-th transition of ρ, if $\beta_i = (n, A[e] := e')$ and $s_{i-1}(e(C,T,\mathsf{b}(n),\mathsf{t}(n))) = j$. We say that the j-th element of A is *read* in the i-th transition of ρ, if $\beta_i = (n, l := e)$ and e contains a sub-expression $A[e']$ such that $s_{i-1}(e'(C,T,\mathsf{b}(n),\mathsf{t}(n))) = j$.

We assume that $[\![P,C,T]\!]$ satisfies the *local array initialization* (LAI) property. Intuitively, LAI states that every local array element is initialized before used. This means that every element of the array is written (by some thread) before the same element is read (by the same or possibly some other thread). This assumption is semantical: it must hold in every possible execution of the program. Formally, this is expressed as follows. Let $\rho = s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_k} s_k$ be a run in the LTS $[\![P,C,T]\!]$ and let A be a local array of P. Then, if there exists $j \in \{0, ..., \mathsf{sz}(A,C,T) - 1\}$ and $i \in \{1, ..., k\}$ such that $A[j]$ is read in the i-th transition of ρ, then there exists some $\ell \in \{1, ..., i-1\}$ such that $A[j]$ is written in the ℓ-th transition of ρ.

We also assume that $[\![P,C,T]\!]$ satisfies the *single array assignment* (SAA) property. Intuitively, SAA states that every element of a global or local output array is assigned exactly once in every execution of the system. Formally, this is expressed as follows. Let $\rho = s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_k} s_k$ be a run in the LTS $[\![P,C,T]\!]$ and let A be a local or global array of P. If A is not a global input array, then for all $j \in \{0, ..., \mathsf{sz}(A,C,T) - 1\}$, there must be exactly one $i \in \{1, ..., k\}$ such that $A[j]$ is written in the i-th transition of ρ.

The above assumptions are not generally guaranteed by our SPMD model. They can be enforced, however, by conditions similar to the non-interference condition presented in Section 6. The details are omitted due to lack of space.

5 Properties of interest

Our ultimate goal is to provide a method for proving 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 the particular order of thread interleavings. We are thus motivated to define *determinism* first, and then define equivalence for deterministic programs.

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 introduce the following equivalences. Consider a SPMD program $P = (G, L, F)$. Let $[\![P, C, T]\!] = (S, S_0, \rightarrow)$, for given $C, T \in \mathbb{N}$. Two states $s, s' \in S$ are said to be *equivalent*, denoted $s \approx s'$, if for each $A \in G$, for any $i \in \{0, ..., \mathsf{sz}(A, C, T) - 1\}$, $s(A[i]) = s'(A[i])$. Let $\rho_1 = s_0^1 \xrightarrow{\beta_1^1} s_1^1 \xrightarrow{\beta_2^1} \cdots \xrightarrow{\beta_{k_1}^1} s_{k_1}^1$ and $\rho_2 = s_0^2 \xrightarrow{\beta_1^2} s_1^2 \xrightarrow{\beta_2^2} \cdots \xrightarrow{\beta_{k_2}^2} s_{k_2}^2$ be two runs in $[\![P, C, T]\!]$. The two runs are said to be *equivalent*, denoted $\rho_1 \approx \rho_2$, if $s_0^1 \approx s_0^2 \Rightarrow s_{k_1}^1 \approx s_{k_2}^2$, that is, 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 $s_0^1 = s_0^2 \Rightarrow s_{k_1}^1 = s_{k_2}^2$.

The above definitions extend to state of two LTSs coming from different SPMD programs P_1 and P_2, with potentially different instantiations of parameters C, T, as long as P_1 and P_2 have the same set of global arrays. We will use this to define equivalence between SPMD programs below. For simplicity, we will assume that P_1 and P_2 have identical sets of input and output global arrays: that is, array A is an input (resp., output) array in P_1 iff it is an input (resp., output) array in P_2. We will also assume that parameters C, T are instantiated identically in the two programs. Both assumptions can be lifted without compromising the results of our framework, however, this would make the presentation heavier, and we opt for simplicity.

Determinism:

Let P be a SPMD program and let $C, T \in \mathbb{N}$. P is said to be *deterministic with respect to* C, T if for any two maximal 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 $C, T \in \mathbb{N}$.

Program Equivalence:

Let $P_1 = (G, L_1, F_1)$ and $P_2 = (G, L_2, F_2)$ be two SPMD programs with identical sets of global arrays. Let $C, T \in \mathbb{N}$. P_1 and P_2 are said to be *equivalent with respect to* C, T, denoted $P_1 \approx_{C,T} P_2$, if (1) P_1 is deterministic with respect to C, T, (2) P_2 is deterministic with respect to C, T, and (3) for all maximal runs ρ_1 in $[\![P_1, C, T]\!]$ and ρ_2 in $[\![P_2, C, T]\!]$, we have $\rho_1 \approx \rho_2$. Let Γ be a subset of $\mathbb{N}$, representing a set of conditions on parameters C, T. We say that P_1 and P_2 are *equivalent with respect to* Γ, denoted $P_1 \approx_\Gamma P_2$, if for all $(C, T) \in \Gamma$, we have $P_1 \approx_{C,T} P_2$.

6 Analysis

Our goals are the following: (1) to check whether a given SPMD program is deterministic, and (2) to check whether two deterministic SPMD programs are equivalent. A key property in achieving these goals is *non-interference*, which roughly states that different threads access different array elements, or the same element but at different times.

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 like the above. Because of the chain-of-sub-automata assumption (Figure 2), it suffices to ensure the absence of race conditions separately on each sync-segment F_i of F. Indeed, from the fact that threads must synchronize on

sync transitions, 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 it cannot interfere with itself. In other words, that we cannot have two threads executing statements of F_i that interfere with each other. Notice that F_i is a special case of a thread automaton, without sync transitions, except for the transition from F_i to the next sync-segment F_{i+1}. Then, let F_i be the thread automaton (Q, q_0, R).

We define the following two 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 the definitions of LHS and RHS on our running example. First, consider thread automaton F^1 (Figure 1, top). F^1 has no sync transitions, therefore, it consists of a single sync-segment: F^1 itself. We have:

$$\mathsf{LHS}(F^1) = \{B[\mathsf{C} \cdot \mathsf{T} - 1 - \mathsf{T} \cdot \mathsf{b} - \mathsf{t}]\} \quad \text{and} \quad \mathsf{RHS}(F^1) = \{A[\mathsf{T} \cdot \mathsf{b} + \mathsf{t}]\}.$$

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

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

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.

Let F be a thread automaton such that $F = F_1 \to \cdots \to F_k$. Fix $C, T \in \mathbb{N}$. We say that *a sync-segment F_i is non-interfering with respect to C, T* if

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

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

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

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

The above formulas are formulas of first-order logic with equality, with array symbols considered to be unary function symbols.

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.*

Proofs can be found in Appendix A.

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^2_1 and F^2_2. Following the definitions, we get: $\mathcal{E}_g(F^2_1) = \emptyset$, $\mathcal{E}_l(F^2_1) = \{(\mathsf{t}, \mathsf{t})\}$, $\mathcal{E}_g(F^2_2) = \{(e, e)\}$, 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

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

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:

$$\mathsf{LHS}(F^3) = \{Loc[\mathsf{t}], B[(\mathsf{C} - 1 - \mathsf{b}) \cdot \mathsf{T} + \mathsf{t}]\}, \qquad \mathsf{RHS}(F^3) = \{A[\mathsf{b} \cdot \mathsf{T} + \mathsf{t}], Loc[\mathsf{T} - 1 - \mathsf{t}]\}.$$

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, it can be seen that this implementation is non-deterministic, and incorrect.

Checking Equivalence:

Let P_1 and P_2 be two deterministic SPMD programs with identical sets of global arrays. Let Γ be a subset of $\mathbb{N}^2$, representing a set of conditions on parameters C, T. We represent the set $\Gamma \subseteq \mathbb{N}^2$ by its characteristic formula ϕ_Γ: the latter is a boolean expression on parameters C, T, such that a tuple $(C, T) \in \mathbb{N}^2$ is in Γ iff it satisfies ϕ_Γ. We want to check whether $P_1 \approx_\Gamma P_2$. We do this in two steps: (1) For each P_i, $i = 1, 2$, we compute a *post-condition* Φ_{P_i}. The latter is a formula that relates global and local array values at the end of program execution. (2) We check whether the post-conditions imply equality of global output arrays. We next make these steps precise and illustrate them on our running example.

Let $P = (G, L, F)$ be a SPMD program. Let Π denote the set of all control-flow paths in F, that is, all paths from the initial location q_0 of F to some final location (recall that F is acyclic, therefore Π is a finite set). For each $\pi \in \Pi$ we will compute a boolean expression ϕ_π. Let $\pi = q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} q_m$. Each α_i in π is either a condition statement, that is, a boolean expression e, or an assignment $l := e$. Define ψ_i to be: e, if α_i is the condition e, and $l = e$ if α_i is the assignment $l := e$. Then, we define ϕ_π to be the conjunction of all ψ_i, and the *post-condition* of P, denoted Φ_P, to be the disjunction of all ϕ_π:

$$\phi_\pi := \bigwedge_{i=1,...,m} \psi_i \qquad \text{and} \qquad \Phi_P := \bigvee_{\pi \in \Pi} \phi_\pi \tag{2}$$

In all ψ_i, local array symbols are superscripted by b (e.g., Loc will appear as Loc^{b}). This is because there is a separate copy of every local array at each core, and we need to refer to each copy individually.

Let us return to our running example of array reversal programs P_1 and P_2, with thread automata F^1 and F^2, respectively (Figure 1). The post-conditions for P_1 and P_2 are:

$$\begin{aligned}\Phi_{P_1} &:= B[\mathsf{C} \cdot \mathsf{T} - 1 - (\mathsf{b} \cdot \mathsf{T} + \mathsf{t})] = A[\mathsf{b} \cdot \mathsf{T} + \mathsf{t}] \\ \Phi_{P_2} &:= Loc^{\mathsf{b}}[\mathsf{t}] = A[\mathsf{b} \cdot \mathsf{T} + \mathsf{t}] \ \wedge \ B[(\mathsf{C} - 1 - \mathsf{b}) \cdot \mathsf{T} + \mathsf{t}] = Loc^{\mathsf{b}}[\mathsf{T} - 1 - \mathsf{t}]\end{aligned}$$

Theorem 2 *Let P be a non-interfering SPMD program w.r.t. $C, T \in \mathbb{N}$. For all $s \in \mathcal{R}_f(P, C, T)$, the following formula is satisfied at state s:*

$$\forall \mathsf{b} \in \{0, ..., C-1\}, \forall \mathsf{t} \in \{0, ..., T-1\} : \Phi_P(C, T) \tag{3}$$

In the above theorem, $\Phi_P(C, T)$ denotes the formula obtained by replacing variables C, T in Φ_P by concrete values C, T.

Let P_1 and P_2 be two SPMD programs having the same set G of global arrays. Recall that P_1 and P_2 must have the same classification of global arrays into input and output arrays. Let $G_{out} \subseteq G$ be the set of output global arrays. For $i = 1, 2$, let ϕ_i be the post-condition formula Φ_{P_i}, with the addition that array symbols in G_{out} are labeled by superscript ^{i}. That is, $B \in G_{out}$ will appear as B^1 in ϕ_1 and as B^2 in ϕ_2. This is done to distinguish the outputs of the two programs. All local array symbols that are common to both programs are also superscripted similarly. Input global array symbols do not need to be distinguished: in fact, by having the same input symbols in both formulas, we implicitly encode the assumption that input array values are the same for both programs.

We define formulas ϕ_{post} and ϕ_{out} as follows:

$$\begin{aligned} \phi_{post} &:= \forall \mathsf{b} \in \{0, ..., \mathsf{C}-1\}, \forall \mathsf{t} \in \{0, ..., \mathsf{T}-1\} : \phi_1 \wedge \phi_2 \\ \phi_{out} &:= \bigwedge_{B \in G_{out}} \forall j \in \{0, ..., \mathsf{sz}(B)-1\} : B^1[j] = B^2[j] \end{aligned}$$

We represent the set $\Gamma \subseteq \mathbb{N}^2$ by its characteristic formula ϕ_Γ: the latter is a boolean expression on parameters C, T, such that a tuple $(C, T) \in \mathbb{N}^2$ is in Γ iff it satisfies ϕ_Γ.

Theorem 3 *Suppose P_1 and P_2 are non-interfering SPMD programs, with respect to any C, T such that $(C, T) \in \Gamma$. Then, $P_1 \approx_\Gamma P_2$ if the following formula is valid:*

$$\forall \mathsf{C}, \mathsf{T} \in \mathbb{N} : \big(\phi_\Gamma \wedge \phi_{post}\big) \Rightarrow \phi_{out} \tag{4}$$

Formula (4) instantiates on our running example as follows:

$$\begin{aligned} \forall \mathsf{C}, \mathsf{T} \in \mathbb{N} : \quad & \Big(\forall \mathsf{b} \in \{0, ..., \mathsf{C}-1\}, \forall \mathsf{t} \in \{0, ..., \mathsf{T}-1\} : \\ & B^1[\mathsf{C} \cdot \mathsf{T} - 1 - (\mathsf{b} \cdot \mathsf{T} + \mathsf{t})] = A[\mathsf{b} \cdot \mathsf{T} + \mathsf{t}] \ \wedge \ Loc^{\mathsf{b}}[\mathsf{t}] = A[\mathsf{b} \cdot \mathsf{T} + \mathsf{t}] \wedge \\ & B^2[(\mathsf{C} - 1 - \mathsf{b}) \cdot \mathsf{T} + \mathsf{t}] = Loc^{\mathsf{b}}[\mathsf{T} - 1 - \mathsf{t}]\Big) \\ & \quad \Rightarrow \forall j \in \{0, ..., \mathsf{C} \cdot \mathsf{T} - 1\} : B^1[j] = B^2[j] \end{aligned} \tag{5}$$

A proof that Formula (5) is valid can be found in Appendix B.

7 Implementation and experiments

We have built a prototype tool that can automatically check non-interference of CUDA programs. Equivalence checking has not been implemented yet in the tool, but the non-interference checking functionality is useful independently, and not available in other tools, as far as we know at the time of writing.

Our 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 $b \cdot T + t$, where b and t are the core and thread ID variables, respectively, and T is the number of threads per core, 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 [2]: `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. This works for the programs above but does not work for all programs. For example, the `BlackScholes` CUDA application contains the following thread function pattern:

```
const int tid = noThreads * coreId + threadId;
const int  TN = noThreads * noCores;
for(int i = tid; i < N; i += TN)
  BlackScholesBodyGPU(A[i], B[i], ...);
```

where `tid` is computed as the global thread index $T \cdot \mathtt{b} + \mathtt{t}$, $\mathtt{TN} = C \cdot T$ is the total number of threads, `A`, `B`, ..., are arrays, `N` is the size of these arrays, and `BlackScholesBodyGPU` is the function the performs the computation.

Such cases can be handled by adapting the non-interference conditions that need to be checked. In particular, we can generate non-interference conditions of the form:

$$\begin{aligned}&\forall b_1, b_2 \in \{0, ..., C-1\}, \forall t_1, t_2 \in \{0, ..., T-1\}, \forall i_1, i_2 \in \mathbb{N} : \\ &\quad \Big(\big(b_1 \neq b_2 \vee t_1 \neq t_2\big) \wedge (\exists k_1, k_2 \in \mathbb{N} : i_1 = k_1 \cdot \mathtt{TN} + t_1 \wedge i_2 = k_2 \cdot \mathtt{TN} + t_2 \wedge t_1 \leq i_1 < \mathtt{N} \wedge t_2 \leq i_2 < \mathtt{N}) \Big) \\ &\qquad \Rightarrow e_1(C, T, b_1, t_1, i_1) \neq e_2(C, T, b_2, t_2, i_2)\end{aligned}$$

where i_1, i_2 are variables corresponding to the instantiation of the loop index `i` for the two threads, and variables k_1, k_2 represent loop iterations. e_1, e_2 are left-hand or right-hand side expressions potentially using variables i_1, i_2, in addition to variables $\mathtt{C}, \mathtt{T}$, and so on. Then, $e_j(C, T, b_j, t_j, i_j)$, for $j = 1, 2$, denotes the expression obtained by substituting the values of these variables, as described in Section 6.

It is worth noting that, in order to prove that the above non-interference condition is valid, it suffices to prove that the following, *quantifier-free* formula, corresponding to its negation, is unsatisfiable:

$$\begin{aligned}0 \leq b_1, b_2 < C \wedge 0 \leq t_1, t_2 < T \wedge \big(b_1 \neq b_2 \vee t_1 \neq t_2\big) \wedge i_1 = k_1 \cdot \mathtt{TN} + t_1 \wedge i_2 = k_2 \cdot \mathtt{TN} + t_2 \wedge \\ t_1 \leq i_1 < \mathtt{N} \wedge t_2 \leq i_2 < \mathtt{N} \wedge e_1(C, T, b_1, t_1, i_1) = e_2(C, T, b_2, t_2, i_2)\end{aligned}$$

Because such formulas are quantifier-free, they can be directly handled by SMT solvers. For instance, to ensure that different threads don't write to the same `A[i]` element in the `BlackScholes` example, it suffices to prove unsatisfiability of the above formula, where $e_1(C, T, b_1, t_1, i_1) = e_2(C, T, b_2, t_2, i_2)$ instantiates to $i_1 = i_2$. Yices takes < 1 sec to prove the formula unsatisfiable for $C = T = 256$ and $\mathtt{N} = 256^3$.

8 Conclusions and ongoing work

We have proposed a novel framework for proving determinism and equivalence of SPMD programs. Our framework relies on a notion of non-interference requiring that different threads access different array elements, or the same element but at different times (thanks to barrier synchronization).

We are currently working toward strengthening our tool so that it can handle a larger set of CUDA programs. Another promising direction is to cast the framework in a theory of arrays. Even though features such as array nesting ($A[B[e]]$) generally result in undecidability [7], we may be able to exploit the restricted form of formulas used in our framework to obtain more positive results.

References

[1] Message-Passing Interface (MPI). See http://www.mcs.anl.gov/research/projects/mpi/.

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

[3] 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.

[4] A. Aiken and D. Gay. Barrier inference. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 342–354, 1998.

[5] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, E. Lee, N. Morgan, G. Necula, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View. Technical Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, Mar 2008.

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

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

[8] E. Clarke, E. Emerson, S. Jha, and A. Sistla. Symmetry reductions in model checking. In *CAV'98*, pages 147–158. Springer, 1998.

[9] E.A. Emerson and E. Clarke. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logic of Programs*. LNCS 131, 1981.

[10] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not. (POPL'05)*, 40(1):110–121, 2005.

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

[12] B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Inf.*, 45(6):403–439, 2008.

[13] J.A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[14] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[15] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, pages 226–239, 2007.

[16] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, Proceedings of IFIP Congress 74*. North-Holland, 1974.

[17] R.M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.

[18] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[19] E.A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[20] R. Lublinerman and S. Tripakis. Checking Equivalence of SPMD Programs Using Non-Interference. Technical Report UCB/EECS-2009-42, EECS Department, University of California, Berkeley, Mar 2009.

[21] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[22] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[23] J. Ousterhout. Why threads are a bad idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference. Available online.

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

[25] R. Palmer, G. Gopalakrishnan, and R. Kirby. Semantics driven dynamic partial-order reduction of MPI-based parallel programs. In *Parallel and Distributed Systems - Testing and Debugging (PADTAD-V)*, July 2007.

[26] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th Intl. Sym. on Programming*, volume 137 of *LNCS*, 1981.

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

[28] 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 *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, February 2008.

[29] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-cc multiprocessor machine code. In *POPL 2009*. See talk slides available from the first author's web site.

[30] 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.

[31] K.C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Verification of source code transformations by program equivalence checking. In *In Compiler Construction, 14th International Conference, CC 2005, Proceedings, volume 3443 of LNCS*, pages 221–236. Springer, 2005.

[32] S. Siegel, A. Mironova, G. Avrunin, and L. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Trans. on Software Engineering and Methodology*, 17(2):1–34, 2008.

[33] A. Valmari. Eliminating redundant interleavings during concurrent program verification. In *PARLE'89*, volume 366 of *LNCS*, pages 89–103. Springer, 1989.

[34] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. Kirby, and R. Thakur. Formal verification of practical mpi programs. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 261–270, New York, NY, USA, 2009. ACM.

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

A Proofs

A.1 Proof of Theorem 1

Define the run ρ^* of $[\![P, C, T]\!]$ to be the run where the order of thread interleaving is fixed and given by the global thread index, such that for every $n, n' \in \mathcal{I}(C, T)$, if $n < n'$ then thread n must execute before thread n' in ρ^*. In other words, for every sequence of transitions $s_1 \overset{n,\alpha}{\rightarrow} s_2 \overset{n',\alpha'}{\rightarrow} s_3$ in ρ^*, we have $n \leq n'$. Notice that ρ^* is uniquely defined.

We will show that for any other run ρ, we have $\rho^* \simeq \rho$. We will do this by transforming ρ to a run ρ', such that $\rho' \simeq \rho$, and ρ and ρ' have only one difference: a pair of successive transitions $s_1 \overset{n,\alpha}{\rightarrow} s_2 \overset{n',\alpha'}{\rightarrow} s_3$ in ρ such that $n > n'$ is swapped to give $s_1 \overset{n',\alpha'}{\rightarrow} s_2' \overset{n,\alpha}{\rightarrow} s_3'$ in ρ'. We will show that it is always possible to do this swapping and moreover that $s_3' = s_3$. Then, it should be clear that $\rho \simeq \rho'$. By repeatedly applying swapping, we can transform ρ to ρ^*. Since all intermediate runs are strongly equivalent, it follows that $\rho \simeq \rho^*$.

To show that swapping is always possible, we distinguish the following cases.

Case (A): α' is a condition statement, i.e., a boolean expression e. In this case, s_2 and s_3 are identical except for the program counter of thread n': this is because condition statements do not modify arrays.

We need to show that e is satisfied at state s_1. Then, the transition $s_1 \overset{n',e}{\rightarrow} s_2'$ exists, and s_2' is identical to s_1 except for the program counter of thread n'. Thus, the transition $s_2' \overset{n,\alpha}{\rightarrow} s_3'$ also exists, and s_3' must be identical to s_3.

Suppose s_1 does not satisfy e. On the other hand, we know that s_2 satisfies e. Then, there must exist some array element whose value changes during transition $s_1 \overset{n,\alpha}{\rightarrow} s_2$. This means that α must be an assignment of the form $A[e_1] := e_3$. Moreover, e must have an array sub-expression $A[e_2]$. Finally, it must be that

$$s_1(e_1[C, T, \mathsf{b}(n), \mathsf{t}(n)]) = s_2(e_2[C, T, \mathsf{b}(n'), \mathsf{t}(n')]) \tag{6}$$

that is, if e_1 evaluates to some index j at thread n then e_2 evaluates to the same index at thread n'. Let b_1, b_2, t_1, t_2 be such that $n = b_1 \cdot T + t_1$ and $n' = b_2 \cdot T + t_2$. Also note that both the assignment $A[e_1] := e_3$ and the condition statement e must be statements of the same sync-segment, say F_i. This is because there is no sync-transition (in fact, there is no transition at all) between transitions $s_1 \overset{n,\alpha}{\rightarrow} s_2$ and $s_2 \overset{n',\alpha'}{\rightarrow} s_3$. We distinguish two further cases.

Case (A.1): A is a global array. Then $A[e_1] \in \mathsf{LHS}(F_i)$ and $A[e_2] \in \mathsf{RHS}(F_i)$. Thus, $(e_1, e_2) \in \mathcal{E}_g(F_i)$. Then Formula (2) is not valid. Indeed, $n \neq n'$ implies $b_1 \neq b_2 \vee t_1 \neq t_2$, and (6) implies that $e_1[C, T, b_1, t_1] = e_2[C, T, b_2, t_2]$ holds in the logic of uninterpreted functions. This contradicts the assumption that F is non-interfering w.r.t. C, T.

Case (A.2): A is a local array. Then again $A[e_1] \in \mathsf{LHS}(F_i)$ and $A[e_2] \in \mathsf{RHS}(F_i)$. In this case, $(e_1, e_2) \in \mathcal{E}_l(F_i)$. Then Formula (2) is not valid. Indeed, A is a local array, thus there is a separate instance of A at each core $k \in \{0, ..., C-1\}$. Thus, n and n' must be threads running at the same core, that is, $b_1 = b_2$. This and $n \neq n'$ imply $t_1 \neq t_2$, and (6) implies that $e_1[C, T, b_1, t_1] = e_2[C, T, b_1, t_2]$ holds in the logic of uninterpreted functions. Again this contradicts the assumption that F is non-interfering w.r.t. C, T. This completes Case (A).

Case (B): α' is an assignment statement $A[e] := e'$. In this case, both transitions $s_1 \overset{n',\alpha'}{\rightarrow} s_2'$ and $s_2' \overset{n,\alpha}{\rightarrow} s_3'$ exist. We need to show that $s_3' = s_3$. Suppose $s_3' \neq s_3$. This means that there exists some array A and element $A[j]$ such that $s_3'(A[j]) \neq s_3(A[j])$. There are two cases: either $A[j]$ is set in both α and α', or it is only set in one of them, and the other modifies a value used in the first. In both cases, using reasoning similar to the above, we can show that one of non-interference formulas (2) or (2) is invalid, which contradicts the assumption that F is non-interfering. The details are omitted.

A.2 Proof of Theorem 2

Suppose $s \in \mathcal{R}_f(P, C, T)$. Suppose (3) is not satisfied at s. Then there exist $b \in \{0, ..., C-1\}$ and $t \in \{0, ..., T-1\}$ such that for any control-flow path π, $\phi_\pi[C, T, b, t]$ is not satisfied at s. Let $n = b \cdot T + t$. Let ρ be a maximal run ρ starting at some initial state s_0 and reaching s. P is non-interfering w.r.t. C, T, therefore, by Theorem 1, P is strongly deterministic w.r.t. C, T. This means that we can assume that ρ is such that thread n is the last thread to execute, after all other threads have executed: by strong determinism, ρ will still reach the same state s.

Let $\pi = q_0 \overset{\alpha_1}{\rightarrow} q_1 \overset{\alpha_2}{\rightarrow} \cdots \overset{\alpha_m}{\rightarrow} q_m$ be the control-flow path that thread n follows in ρ. There must be some $i \in \{1, ..., m\}$ such that $\psi_i[C, T, b, t])$ is not satisfied at s, where ψ_i is the boolean expression obtained from α_i. We distinguish two cases.

Case (A): α_i is an assignment statement $A[e] := e'$. Then, ψ_i is the equality $A[e] = e'$. Let $j = s(e[C, T, b, t])$ and $v = s(e'[C, T, b, t])$. $\psi_i[C, T, b, t]$ not satisfied at s means $s(A[j]) \neq v$. Since π is the control-flow path that thread n follows in ρ, thread n must execute the assignment $A[e] := e'$. Therefore, ρ must have a transition $s_1 \overset{n,\alpha_i}{\rightarrow} s_2$. We claim that: (1) $s_1(e[C, T, b, t]) = j$ (which means that $A[j]$ is written in this transition) and (2) $s_1(e'[C, T, b, t]) = v$. From the semantics of assignments, (1) and (2) imply $s_2(A[j]) = v$. By the SAA assumption, $A[j]$ is written exactly once in ρ, therefore, its value at s must be the same as its value at s_2. Thus, $s(A[j]) = v$: contradiction.

We proceed to prove claims (1) and (2) above. We will show that every sub-expression of e or e' has the same value at s_1 as it has at s. Such a sub-expression can be of the following type:

- A constant: obviously it always has the same value.
- A parameter among $\mathsf{C}, \mathsf{T}, \mathsf{b}, \mathsf{t}$: because these parameters are substituted by the same values C, T, b, t, respectively, they are the same in $s_1(e[C,T,b,t])$ and $s(e[C,T,b,t])$.
- A global input array element: input arrays are never written, thus they maintain a constant value throughout a run.
- An element of a writable array, say, $B[k]$. By the LAI assumption, $B[k]$ must be written before it is read, therefore, before the transition $s_1 \overset{n,\alpha_i}{\rightarrow} s_2$. By the SAA assumption, $B[k]$ is only written once. Therefore, $B[k]$ has the same value at s_1 and at s.

Thus, claims (1) and (2) hold, which completes the proof for case (A).

Case (B): α_i is a condition statement, i.e., a boolean expression e. Then, ψ_i is e. $\psi_i[C,T,b,t]$ not satisfied at s means $s(e[C,T,b,t]) = \mathit{false}$. Since π is the control-flow path that thread n follows in ρ, thread n must execute the condition statement e. Therefore, ρ must have a transition $s_1 \overset{n,e}{\rightarrow} s_2$, which implies that $s_1(e[C,T,b,t]) = \mathit{true}$. Following a reasoning similar to the above, we can show that $e[C,T,b,t]$ takes the same value at s_1 and at s: contradiction. This completes the proof.

A.3 Proof of Theorem 3

Suppose $P_1 \not\approx_\Gamma P_2$. Then there exist $\mathsf{C}, \mathsf{T} \in \mathbb{N}$ such that $(\mathsf{C}, \mathsf{T}) \in \Gamma$ and $P_1 \not\approx_{\mathsf{C},\mathsf{T}} P_2$. This in turn means that there exist maximal runs $\rho_1 \in [\![P_1, \mathsf{C}, \mathsf{T}]\!]$ and $\rho_2 \in [\![P_2, \mathsf{C}, \mathsf{T}]\!]$ such that $\rho_1 \not\approx \rho_2$. That is, ρ_1 and ρ_2 start at equivalent initial states $s_0^1 \approx s_0^2$ but end at non-equivalent final states $s_1 \not\approx s_2$.

$(\mathsf{C}, \mathsf{T}) \in \Gamma$ implies that ϕ_Γ is satisfied by C, T. We will show that ϕ_{post} also holds, but ϕ_{out} does not hold. This means (4) is invalid.

$s_1 \not\approx s_2$ implies that there exist $B \in G_{out}$ and j such that $s_1(B[j]) \neq s_2(B[j])$. We will show that $\phi_{post} \wedge B^1[j] \neq B^2[j]$ is a satisfiable formula. Suppose it is not. Then, ϕ_{post} implies $B^1[j] = B^2[j]$. In the theory of uninterpreted functions this means that if states s and s' satisfy ϕ_{post} then $s(B[j]) = s'(B[j])$. By Theorem 2, s_1 and s_2 satisfy ϕ_{post}, thus, $s_1(B[j]) = s_2(B[j])$: contradiction.

B Proving equivalence for the array reversal example

As shown in Section 6, Formula (4) instantiates on our running example as Formula (5). To prove that the two array reversal programs are equivalent, we need to show that Formula (5) is valid. Suppose this is not the case. Then there exist $\mathsf{C}, \mathsf{T} \in \mathbb{N}$ such that

$$\begin{aligned}&\Big(\forall \mathsf{b} \in \{0,...,\mathsf{C}-1\}, \forall \mathsf{t} \in \{0,...,\mathsf{T}-1\} : \\ &B^1[\mathsf{C}\cdot\mathsf{T}-1-(\mathsf{b}\cdot\mathsf{T}+\mathsf{t})] = A[\mathsf{b}\cdot\mathsf{T}+\mathsf{t}] \wedge \\ &Loc^{\mathsf{b}}[\mathsf{t}] = A[\mathsf{b}\cdot\mathsf{T}+\mathsf{t}] \wedge \\ &B^2[(\mathsf{C}-1-\mathsf{b})\cdot\mathsf{T}+\mathsf{t}] = Loc^{\mathsf{b}}[\mathsf{T}-1-\mathsf{t}]\Big)\end{aligned} \tag{7}$$

holds and

$$\forall j \in \{0,...,\mathsf{C}\cdot\mathsf{T}-1\} : B^1[j] = B^2[j] \tag{8}$$

does not hold. The latter implies there exists $j \in \{0,...,\mathsf{C}\cdot\mathsf{T}-1\}$ such that $B^1[j] \neq B^2[j]$. We can find unique $b_0 \in \{0,...,\mathsf{C}-1\}$ and $t_0 \in \{0,...,\mathsf{T}-1\}$ such that $j = b_0\cdot\mathsf{T}+t_0$. Then, $B^1[b_0\cdot\mathsf{T}+t_0] \neq B^2[b_0\cdot\mathsf{T}+t_0]$.

Let $b = \mathsf{C}-1-b_0$ and $t = \mathsf{T}-1-t_0$. Then:

$$\mathsf{C}\cdot\mathsf{T}-1-(b\cdot\mathsf{T}+t) = b_0\cdot\mathsf{T}+t_0 \tag{9}$$

From (7) and the facts $b_0, b \in \{0, ..., \mathsf{C}-1\}$ and $t_0, t \in \{0, ..., \mathsf{T}-1\}$, we get the following equalities:

$$B^1[\mathsf{C}\cdot\mathsf{T}-1-(b\cdot\mathsf{T}+t)] = A[b\cdot\mathsf{T}+t] \quad (10)$$

$$Loc^b[t] = A[b\cdot\mathsf{T}+t] \quad (11)$$

$$B^2[(\mathsf{C}-1-b)\cdot\mathsf{T}+t_0] = Loc^b[\mathsf{T}-1-t_0] \quad (12)$$

From (9) and (10) we get

$$B^1[b_0\cdot\mathsf{T}+t_0] = A[b\cdot\mathsf{T}+t] \quad (13)$$

From (13), (11) and the fact $t = \mathsf{T}-1-t_0$, we get

$$B^1[b_0\cdot\mathsf{T}+t_0] = Loc^b[t] = Loc^b[\mathsf{T}-1-t_0] \quad (14)$$

From (14), (12) and the fact $b = \mathsf{C}-1-b_0$, we get

$$B^1[b_0\cdot\mathsf{T}+t_0] = B^2[(\mathsf{C}-1-b)\cdot\mathsf{T}+t_0] = B^2[b_0\cdot\mathsf{T}+t_0]$$

which contradicts our assumption $B^1[b_0\cdot\mathsf{T}+t_0] \neq B^2[b_0\cdot\mathsf{T}+t_0]$. Thus, (5) must be valid.