A Symbolic Verifier for CUDA Programs

Guodong Li[1], Ganesh Gopalakrishnan[1], Robert M. Kirby[1], Dan Quinlan[2]

[1] School of Computing, University of Utah
[2] Lawrence Livermore National Laboratory, Livermore, CA, USA

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods

General Terms Verification

Keywords CUDA, Formal Verification, Symbolic Analysis, SPMD Program

1. Introduction

There is increasing interest in performing general-purpose computations on GPUs such as the CUDA architecture [2]. Unfortunately, programs on parallel architectures are prone to subtle correctness bugs caused by data races and incorrect exploration of parallelism. Traditional testing methods tend to miss bugs as they cannot guarantee examining all relevant thread interleavings. Explicit state model checking assumes concrete input values, again limiting coverage, as it is expensive to obtain a sufficient set of concrete program input values.

We present a preliminary automated verifier based on mechanical decision procedures which is able to prove functional correctness of CUDA programs and guarantee (modulo reasonable assumptions) to detect bugs such races. We first encode the concurrent behavior of multiple threads to a constraint formula containing symbolic variables. We then send this formula to the Yices [3] solver for satisfiability check to determine whether the correctness is ensured or a bug is found. Our encoding ensures that *all possible input values and thread interleaving* are considered, *albeit in a symbolic manner*. To mitigate interleaving explosion, we employ a symbolic partial order reduction (POR) technique.

1.1 Related Work

An instrumentation based technique [1] is reported to find races and shared memory bank conflicts. However, they assume concrete inputs values, and only explore one thread interleaving. A determinism (*i.e.* no races) checking tool [5] constructs constraints from an automaton without considering the communication (*e.g.* value passing) among threads. In contrast, our tool works on control flow graphs; and models communicating (and even interfering) programs while deduce non-interference statically. It is trivial to apply our method to check equivalence of two Cuda programs as it suffices to conjunct the two models and compare the outputs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00

1.2 CUDA Programming Model

CUDA extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads. Each of the threads that execute a kernel is given a unique thread ID that is accessible within the kernel through the built-in $threadIdx$ variable — a 3-dimensional vector indexing a *thread block*. A block contains multiple concurrent threads cooperating through shared memory and synchronizating by calling the barrier function $__syncthreads()$. Threads in different blocks will not synchronize by the barrier.

2. Symbolic Verifier

SAT-based Bounded Model Checking (BMC) is one of the leading techniques for model checking systems. It constructs a propositional formula enumerating all possible executions of length k. That is, the system behavior is modeled by a propositional formula. This formula along with the negation of a property to be checked is dumped to a SAT solver for satisfiability check. If the solver says yes, then the property is violated as witnessed by the counter example returned by the solver. A representative BMC approach, C-Bounded Model Checking (CBMC) [7], translates a C program with no loops or function calls into single assignment form (SSA form). It is extended to TCBMC [8] that handles multi-threaded C programs by bounding the number of context switches.

Our method also translates the program into a bounded program by unrolling the loops to the given bound and inlining functions. However, our method differs from TCBMC and other traditional methods in:

- We target CUDA programs which feature different synchronization mechanisms such as using barriers instead of locks and mutexes.
- We use an SMT solver instead of SAT solvers so as to handle arrays, uninterpreted functions, etc. In contrast, TCBMC relies on SAT solvers and assumes no arrays and other compound data structures, thus it needs to preprocess the source program by instantiating each possible value for an array index.
- We keep the program's control flow structure rather than flattening the program and guarding each statement with its path condition. We also generate much simpler constraints. For instance, for each shared variable read, TCBMC builds a constraint of size $O(n_v \times n_t)$, where n_v is the number of blocks writing this variable and n_t the number of threads; while our method generates only a simple array assignment.
- We apply a POR technique to reduce redundant interleavings based on the fact that Cuda programs are well synchronized.

2.1 Generic Modeling

The execution of a concurrent program is scheduler dependent. Suppose threads t_1 and t_2 perform the following accesses on a

shared variable k, where each access is superscripted by the thread id and subscripted by its position (e.g. line number) in the control flow. Depending on the execution order of the three writes, k's value read by t_1 could be x, y or z, which result from schedules $\{k_1^{t_1}; k_2^{t_1}; \ldots\}$, $\{k_1^{t_1}; k_1^{t_2}; k_2^{t_1}; k_2^{t_2}\}$ and $\{k_1^{t_1}; k_2^{t_1}; k_2^{t_2}; k_1^{t_2}\}$ respectively. A comprehensive model should enumerate all these three possibilities.

thread 1	thread 2
$k_1^{t_1} := x;$	$k_1^{t_2} = y;$
read $k_2^{t_1};$	$k_2^{t_2} = z;$

The main idea of modeling concurrent executions is to associate with each shared variable access a schedule id (SId), which indicates its timestamp (step) in a schedule. The SId set in the above example is $\{k_1^{t_1}, k_1^{t_2}, k_2^{t_1}, k_2^{t_2}\}$; expression $k[k_1^t]$ gives k's value at step k_1^t. An ordering of SIds constitutes a schedule. The following ordering gives a possible schedule, where the read gets value y. Note that this read at $k_1^{t_2}$ get the previous (written) value from $k[k_1^{t_2}-1]$. A set of schedules can be specified by a constraint on the orders. Clearly permuting these SIds on $\{1, \ldots, 4\}$ will generate all possible schedules; and constraint $k_2^{t_1} < k_1^{t_2}$ allows only one order.

$$k_1^{t_1} = 1 \ \wedge\ k_1^{t_2} = 2 \ \wedge\ k_2^{t_1} = 3 \ \wedge\ k_2^{t_2} = 4$$

Now we show how to generate models for CUDA programs. We show below a simple CUDA kernel and the generated constraint. The variables in a thread t are superscripted by t; expression $v \uplus (i \mapsto x)$ denotes the update of array v by setting the element at i to x; and `ite` stands for "`if then else`".

```
__global__ kernel (unsigned int* k) {
  unsigned int s[2][3] = {{0,1,2},{3,4,5}};
  unsigned int i = threadIdx.x;
  unsigned int j = k[i] - i;
  if (j < 3)
    { k[i] = s[j][0];  j = i + j; }
  else
    s[1][j && 0x11] = k[i] * j;
  __syncthreads();
  k[j] = s[2][1] + j;
}
```

$$
\begin{array}{l}
\texttt{TRANS(t)} \equiv \\
s_1^t[0] = \lambda i \in \{0,1,2\}.i \ \wedge\ s_1^t[1] = \lambda i \in \{0,1,2\}.i+3) \wedge \\
i_1^t = t \ \wedge\ j_1^t = k[k_0^t][i_1^t] - i_1^t \ \wedge \\
\text{ite}(j_1^t < 3,\ k[k_1^t] = k[k_1^t - 1] \uplus ([i_1^t] \mapsto s_1^t[j_1^t][0]) \ \wedge\ j_2^t = i_1^t + j_1^t \\
\qquad\qquad \wedge\ s_2^t = s_1^t, \\
\qquad\qquad s_2^t = s_1^t \uplus ([1][j_1^t \# 0x11] \mapsto k[k_2^t][i_1^t] \times j_1) \ \wedge \\
\qquad\qquad j_2^t = j_1^t \ \wedge\ k[k_1^t] = k[k_1^t - 1]) \\
k[bar_0] = k[bar_0 - 1] \ \wedge \\
k[k_3^t] = k[k_3^t] \uplus ([j_2^t] \mapsto s_2^t[2][1] + j_2^t)
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\texttt{TRANS}(t_1, \cdots, t_n) \equiv \bigwedge_{i \in [1,n]} \texttt{TRANS}(t_i)} \\
\multicolumn{2}{l}{\texttt{ORDER}(t_1, \cdots, t_n) \equiv} \\
(1) & \bigwedge_{i \in [1,n]} (k_0^{t_i} < \{k_1^{t_i}, k_2^{t_i}\} < bar_0 < k_3^{t_i}) \\
(2) & bar_0 < l \ \wedge\ \bigwedge_{i \in [1,n],\, j \in [0,3]} (k_j^{t_i} < l) \text{ where } l = 4n+1. \\
(3) & \texttt{rank}(bar_0) = 0 \ \wedge\ \bigwedge_{i \in [1,n],\, j \in [0,3]} (\texttt{rank}(k_j^{t_i}) = 4i + j)
\end{array}
$$

Logical formula are built according to a topological order of the nodes in the CFG. During this process, SSA indexes are assigned to variables; SIds are created for shared variable accesses.

Converting Basic Statements. The following Γ constructs a logical formula from single statements and expressions, where `next` and `cur` return the next and the current SSA indices of a variable respectively. Note that each shared variable access is associated with its SId. A write to an array variable is actually modeled as an array update.

$$
\begin{array}{lcl}
\Gamma(e_1 \ \texttt{op}\ e_2) & = & \Gamma(e_1) \ \texttt{op}\ \Gamma(e_2) \\
\Gamma(v := e) & = & \left\{ \begin{array}{ll} v_{\texttt{next}(v)} = \Gamma(e) & \text{if } v \text{ is a local variable} \\ v[v^t_{\texttt{next}(v)}] = \Gamma(e) & \text{otherwise} \end{array} \right. \\
\Gamma(v) & = & \left\{ \begin{array}{ll} v_{\texttt{cur}(v)} & \text{if } v \text{ is a local variable} \\ v[v^t_{\texttt{next}(v)}] & \text{otherwise} \end{array} \right.
\end{array}
$$

Handling Control Flow. The SSA indices of the variables updated in the two clauses of a branch statement "`if` c `then` blk_1 `else` blk_2" should be synchronized so that subsequent statements have a consistent view of their values. The following example gives an illustration: $i_3 = i_2$ is added into the first clause so that later on i_2 is invisible and only variable i_3 will be referred.

$$
\begin{array}{ll}
\texttt{ite}(c,\ i_2 = e_1,\ i_3 = e_2) & \text{becomes} \\
\texttt{ite}(c,\ i_2 = e_1 \ \wedge\ i_3 = i_2,\ i_3 = e_2) &
\end{array}
$$

Such synchronization is done at the join node by inserting the following formula into $\Gamma(blk_1)$ (and similarly to $\Gamma(blk_2)$), where $\texttt{cur}(blk, v)$ returns v's last SSA index in blk. For instance, in the given example $s_2^t = s_1^t$ is added into the first clause of the "if" statement, and $j_2^t = j_1^t \wedge k[k_1^t] = k[k_1^t - 1]$ the second clause. Consequently, only s_2^t will be referred in subsequent statements.

$$v_j = v_i \quad \text{for} \quad i = \texttt{cur}(blk_1, v),\ j = \texttt{cur}(blk_2, v) \text{ such that } i < j$$

Constraining Schedules. An assignment to the SIds of a shared variable determines a concurrent execution over the threads with respect to this variable. Supposed there are m accesses, we assign $0, \cdots, m-1$ to their SIds with respect to the program order in each thread. A shared barrier has only one SId among all threads. Given threads $t_1, \cdots, t_n$, predicate $\texttt{ORDER}(t_1, \cdots, t_n)$ constrains the assignments by requiring:

- The program order must be respected in each thread.
- All SIds are natural numbers less than $n_t \times n_v + n_b$, where n_t, n_v and n_b are the numbers of threads, accesses in each thread and barriers respectively.
- All SIds have distinct values. This is enforced by introduced a function `rank` that maps SId_1 and SId_2 to different values for $SId_1 \neq SId_2$. Note that, for any f, $f(i) \neq f(j)$ implies $i \neq j$.

A valid schedule of the given example for two threads is:

$$
\begin{array}{l}
k_0^{t_1} = 0 \ \wedge\ k_1^{t_1} = 1 \ \wedge\ k_0^{t_2} = 2 \ \wedge\ k_1^{t_2} = 3 \ \wedge\ k_2^{t_2} = 4 \ \wedge \\
k_2^{t_1} = 5 \ \wedge\ bar_0 = 6 \ \wedge\ k_3^{t_2} = 7 \ \wedge\ k_3^{t_1} = 8
\end{array}
$$

Checking Properties. In order to detect races we record in aid the thread id of an access. The aid is set to $\perp$ at barriers. At each write access with SId i we check whether the previous access operates on the same address as well as is from other threads: $aid[i-1] \notin \{t, \perp\}$. If yes then a race is found. For example, we give below the access ids and access types (R for read and W for write) of the above schedule. The first two writes are preceded by accesses from the same thread; the third by a barrier; and the fourth by a write to a different address ($j^{t_1} \neq j^{t_2}$). Hence no race is found for this schedule. As our encoding guarantees that all valid schedules are investigated, a race exhibiting in any particular schedule will not be missed.

SId :	$k_0^{t_1}$	$k_1^{t_1}$	$k_0^{t_2}$	$k_1^{t_2}$	$k_2^{t_2}$	$k_2^{t_1}$	bar_0	$k_3^{t_2}$	$k_3^{t_1}$
type :	$R\ i$	$W\ i$	$R\ i$	$W\ i$	$R\ i$	$R\ i$		$W\ j$	$W\ j$
aid :	t_1	t_1	t_2	t_2	t_2	t_1	$\perp$	t_2	t_1

Users can specify the properties to be checked using our `assume` and `guarantee` directives. If a precondition $assume(P)$ and a postcondition $guarantee(Q)$ are specified, formula $P \wedge \neg Q$ is added into the constraint. For example, we can specify the correctness of the bitonic sort program over four threads as follows:

```
__global__ bitonic (int vals[]) {...}
void guarantee ()
  {assert(vals[t]≤vals[t+1]≤vals[t+2]≤vals[t+3]);}
```

In practice, when a correctness property is to be checked, the race detection can be disabled even for race sensitive programs since it is examined implicitly, *e.g.* races may lead to the violation of the property.

2.2 Incremental Modeling with Partial Order Reduction

The method presented in section 2.1 is designed to be generic, *e.g.* even unstructed and interfering programs can be modeled and checked. For example, each thread may execute statement $\texttt{atomicAdd}(k, 1)$ for counting, the order of executions doesn't matter; and races on shared variable k will not affect the correctness of the program.

On the other hand, this method may suffer from the performance problem due to the possibility of exploring too many schedules. In practice users may assume certain patterns on CUDA programs such as shared variable accesses are non-interfering and threads synchronize in a specific manner. This allows room for performance improvement using reduction techniques.

Partial order reduction (POR) exploits the commutativity of concurrent transitions to prune redundant interleavings. Specifically, if $i_1, \cdots, i_n$ are evaluated to distinct memory addresses, then accesses $v^{t_1}[i_1], \cdots, v^{t_n}[i_n]$ are *non-conflicting* and it suffices to examine one arbitrary interleaving of them. The key of our POR method is to identify and sequentialize non-conflicting accesses. so that only one interleaving is investigated. Given an access at $k_i^{t_1}$ writing address $a_1^{t_1}$, for another access at $k_j^{t_2}$ that writes or reads address $a_2^{t_2}$, if $t_1 \neq t_2 \wedge a_1^{t_1} = a_2^{t_2}$ is unsatisfiable then these addresses do not overlap and the accesses are non-conflicting. In this case we may specify an order in which $k_i^{t_1}$ happens before $k_j^{t_2}$; or, as a simplification, remove $k[k_j^{t_2}]$ and have the second access reuse $k[k_i^{t_1}]$ by setting $k_j^{t_2} = k_i^{t_1}$. In the above example, the first three accesses to k are non-conflicting because their addresses are the thread id t. For more complicated programs the challenge is to determine the values of $a_1^{t_1}$ and $a_2^{t_2}$.

2.3 Barrier Interval

CUDA intra-block thread executions exhibit a regular pattern: $\{t_1, \cdots, t_n\}$ execute $\rightarrow$ barrier $\rightarrow$ $\{t_1, \cdots, t_n\}$ execute $\rightarrow \cdots$. Since an access before a barrier will never conflict with an access after this barrier, we may focus on the accesses between two consecutive barriers (so called a *barrier interval* or *BI*). If the accesses in a BI are non-conflicting, we build a transition constraint by sequentializing them; then we move to the next BI. To improve performance, we utilize Yices's *incremental SMT solving* facility that reuses existing conflict clauses in the context when checking new expressions. As an illustration we consider the following program where shared variables are marked with a hat for readability.

$$1: \ j^t := \widehat{i}^t + t + 1; \quad 2: \ \text{__synthreads}; \quad 3: \ e_1 = \widehat{k}^t[\widehat{i}^t]; \\ 4: \ \widehat{k}^t[j^t] = e_2; \quad 5: \ \text{__synthreads}; \quad 6: \ \text{write } \widehat{i}^t$$

Let's consider the case of two threads t_1 and t_2. The first BI consists of statement 1. Since there is no write to i, accesses $i[i_0^{t_1}]$ and $i[i_0^{t_2}]$ are non-conflicting and can be set to $i[0]$, *i.e.* both of their SIds are 0. The transition upto statement 2 is:

$$\texttt{TRANS}(t_1, t_2)_2 \equiv j_1^{t_1} = i[0] + t_1 + 1 \ \wedge \ j_1^{t_2} = i[0] + t_2 + 1$$

The second BI consists of a read and a write to variable k. We need to determine whether their addresses may overlap for different threads. Given $\texttt{TRANS}(t_1, t_2)_2 \wedge t_1 \neq t_2$, expression $j_1^t = i[0]$ is unsatisfiable for $t \in \{t_1, t_2\}$, so does $j_1^{t_1} = j_1^{t_2}$, hence the accesses to k are non-conflicting. In this case we set their SIds to 0 to obtain

htp2.5cm

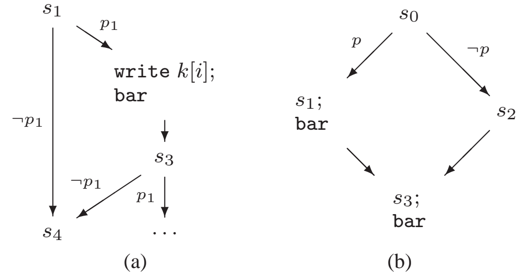

Figure 1. Example CFGs.

the transition constraint upto statement 4:

$$\texttt{TRANS}(t_1, t_2)_4 \equiv \texttt{TRANS}(t_1, t_2)_2 \ \wedge \ \bigwedge_{t \in \{t_1, t_2\}} (\Gamma(e_1^t) = k[0][j_1^t]) \\ \wedge \ k[1] = k[0] \uplus (j_1^{t_1} \mapsto \Gamma(e_2^{t_1})) \uplus (j_1^{t_2} \mapsto \Gamma(e_2^{t_2}))$$

If we remove the barrier at statement 5, then the second BI includes statement write $\widehat{i}^t$. Expression $j_1^t = \widehat{i}$ is satisfiable since $\widehat{i}$ may be written a value equal to $j_1^{t_1}$ or $j_1^{t_2}$. In fact this conflict is reduced to the conflict on i at statements 3 and 6, which will be caught in the next step that performs check on i.

The key point here is to build the constraints following the program order without considering interleavings. A race depending on schedules will be reduced to another race that can be found without thread interleavings. In the above example, although $k[j^{t_1}]$ and $k[i^{t_2}]$ may conflict in a schedule where $\texttt{write} \ i^{t_2}$ occurs before the write $k[j^{t_1}]$, this conflict is actually reduced to the conflict on i that can be identified in the program order.

If all the shared variable accesses within a BI are proven to be non-conflicting, then, as indicated by Theorem 2.1, the conflict graph (obtained by inserting conflicting edge into the CFG) is acyclic and the entire BI is race free.

Theorem 2.1. (Sequentializibility) *Within a barrier interval, if each pair of accesses, at least one of which is a write, is proven to be non-conflicting with respect to the program order, then the entire interval is race free and can be sequentialized.*

Proof Sketch. We say that an expression is *schedule independent* if its value is irrelevant to the schedules. Consider two accesses with addresses i and j respectively on the same shared variable. If neither i nor j is dependent (control-dependent and data-dependent) on a shared variable, then obviously expression $i = j$ is schedule independent. Otherwise, suppose i or j depends on a shared variable k, if the accesses to k is non-conflicting, *i.e.* k is schedule independent, then clearly $i = j$ is schedule independent as well. Thus the checking is reduced to examining k. If all accesses are shown to be non-conflicting, then they are schedule independent.

2.4 Conditional Barrier

To facilitate symbolic analysis we maintain path conditions on the edges. Path conditions are taken into consideration when addresses are compared. Consider for example the CFG in Figure 1.a, formula $i^{t_1} = i^{t_2} \ \wedge \ p_1^{t_1} \ \wedge \ p_1^{t_2}$ is established when checking whether accesses $k[i]$ conflict.

Many CUDA programs are well synchronized such that BIs are easy to identify. What if a barrier is within a conditional branch? This requires us to continue exploring the other branch and building constraints until encountering another barrier. For the CFG in Figure 1.a we also consider whether $\texttt{write} \ k[i]$ will conflict with the accesses in s_4.

Figure 1.b gives another illustration where the left branch contains a barrier. Suppose s_0 is a statement containing no shared variable access, the conflict check includes the following expressions (here $\not\sim$ denotes non conflicting). It may be noted that the comparison between s_1 and s_3 is guarded by path condition $\neg p$ which indicates that s_3 cannot be reached by thread t_2 through path p^{t_2} when thread t_1 is at s_1.

$$p^{t_1} \wedge p^{t_2} \Rightarrow s_1^{t_1} \not\sim s_1^{t_2} \qquad p^{t_1} \wedge \neg p^{t_2} \Rightarrow s_1^{t_1} \not\sim s_2^{t_2}$$
$$p^{t_1} \wedge \neg p^{t_2} \Rightarrow s_1^{t_1} \not\sim s_3^{t_2} \qquad \neg p^{t_1} \Rightarrow s_2^{t_1} \not\sim s_3^{t_2}$$

which can be simplified by reusing the path conditions:

$$\texttt{ite}\,(p^{t_1},\ \texttt{ite}(p^{t_2},\ s_1^{t_1} \not\sim s_1^{t_2},\ s_1^{t_1} \not\sim s_2^{t_2} \wedge s_1^{t_1} \not\sim s_3^{t_2}),\ s_2^{t_1} \not\sim s_3^{t_2})$$

This method works particularly well for structural correct programs [6] where threads make the same branch decisions. On the other hand, since the generic method presented in Section 2.1 is able to handle arbitrary barrier structures, it can be applied to those tricky cases as well.

Generic Method vs. Incremental Method on Conflict Detection. The incremental method establishes pairwise constraints on shared variable accesses, thus the number of constraints is $n \times m$ where n and m is the number of writes and reads respectively. It is suitable for BIs containing a small number of accesses. The generic method, in contrast, relies on the SId assigned to each access, thus the number of constraints is linear to the number of accesses. However it may explore too many schedules than necessary. Hence our final method is the combination of these methods as presented above.

2.5 Other Issues

Our method translates the program into a bounded program by unrolling the loops to a certain bound. The unrolling can be done in the incremental modeling phase. For example, given a loop guarded by condition p_1, we first unroll it once and check the satisfiability of p_1 on the computed transition, if yes then we continue unrolling until p_1 doesn't hold. Finally we get an acyclic CFG as illustrated in Figure 1.a. To ensure preciseness we check and make sure that the loop condition becomes false after the unrolling.

To model aliasing induced by pointers we may use a global array to represent the shared memory. Since typical CUDA programs exhibit very limited pointer functionality, esp. no pointer arithmetic operations, this will not incur problems in practice.

CUDA programs are highly symmetric such that all threads execute the same kernel parameterized by its thread id. In general we only need to consider two threads for conflict check. In some cases, the address of an access may depend on a value contributed by multiple threads as illustrated below. This doesn't bring us any problem as we just need to add more threads when building the model incrementally.

$$\texttt{if}\ p\ \{i = a[t_1] + a[t_2];\}\ \texttt{else}\ \{i = a[t_1] + a[t_3];\}$$
$$\texttt{write}\ a[i];$$

3. Experimental Results and Future Work

We performed preliminary experiments on a machine with an Intel Pentium4 3.60GHz processor to check the reduction and the bitonic sort program in CUDA SDK 2.0 Suite [2]. We use the ROSE compiler [4] to parse CUDA programs; the constraint generation time is negligible. The following table shows the SMT solving time in seconds. Here n denotes the number of threads; T.O denotes Time Out ($>$ 10 minutes). Correctness is proven for bug-free programs, and Bug is for bugged programs obtained by removing barriers or modifying the statements intentionally to introduce bugs. Correctness check takes longer time since the solver needs to prove unsatisfiability (*i.e.* absence of bugs) for all cases.

Property	**Reduction**		**Bitonic Sort**	
	n = 2	n = 4	n = 2	n = 4
Correctness	0.46	T.O	560	T.O
Bug	0.35	240	1.29	T.O
Correctness (with POR)	<0.1	2.84	<0.1	3.7
Bug (with POR)	<0.1	0.16	<0.1	0.45

We plan to address more synchronization schemes (*e.g.* inter-block parallelism and CPU–GPU communication); and apply it to verify the compilation and optimization of CUDA programs.

References

[1] M. Boyer, K. Skadron and W. Weimer. Automated Dynamic Analysis of CUDA Programs. http://www.cs.virginia.edu/∼mwb7w/cuda/.

[2] CUDA Zone. http://www.nvidia.com/object/cuda_home.html.

[3] Yices: An SMT Solver. http://yices.csl.sri.com.

[4] The ROSE Compiler. http://www.rosecompiler.org/.

[5] R. Lublinerman and S. Tripakis, Checking Equivalence of SPMD Programs Using Non-Interference. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-42.html.

[6] A. Aiken and D. Gay, Barrier Inference. 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), 1998.

[7] Bounded Model Checking for ANSI-C. http://www.cprover.org/cbmc/.

[8] I. Rabinovitz and O. Grumberg. Bounded Model Checking of Concurrent Programs. CAV, pp. 82-97, 2005.