Parameterized Verification of GPU Kernel Programs

Guodong Li*
Fujitsu Labs of America
Sunnyvale, CA,
Email: ligd@cs.utah.edu

Ganesh Gopalakrishnan
School of Computing, University of Utah
Salt Lake City, UT,
Email: ganesh@cs.utah.edu

***Abstract*—We present an automated symbolic verifier for checking the functional correctness of GPGPU kernels parametrically, for an arbitrary number of threads. Our tool PUG$_{para}$ checks the functional equivalence of a kernel and its optimized versions, helping debug errors introduced during memory coalescing and bank conflict elimination related optimizations. Key features of our work include: (1) a symbolic method to encode a comparative assertion across two kernel versions, and (2) techniques to overcome SMT solver input-language restrictions through overapproximations, yielding an efficient bug-hunting method.**

***Keywords*-GPU programming; Formal verification; Parameterized reasoning; Satisfiability Modulo Theories (SMT); Symbolic analysis; Correctness of Optimizations.**

I. Introduction

There is an explosive growth of interest in *Graphical Processing Units (GPU)* for speeding up computations occurring at all application scales~[11]. *When properly programmed*, GPUs can yield 20x to 100x performance compared to traditional CPUs. Often this requires heroic acts of programming: (i)~keep the GPU threads busy; (ii)~ensure coalesced data transfers from the GPU global memory to the GPU shared memory; and (iii)~minimize bank conflicts during shared memory accesses. Unfortunately, bugs are frequently introduced during CUDA programming and optimization; and few tools are available to verify CUDA programs. In this paper, we present the first (to the best of our knowledge) parameterized reasoning method for GPU kernels.

GPU kernels are comprised of extremely light-weight Single Instruction Multiple Data (SIMD) threads that synchronize sparingly using barriers. These little resemble threads of C/Java that are heavy-weight, and synchronize using locks/monitors. In [13], we introduce an SMT~[22] based approach for analyzing GPU kernels through a new tool PUG that can handle kernels of thousands of lines of code – but for a fixed number (*e.g.*, two or three) threads. It builds a symbolic model (as transition relation) according to the operational semantics of a kernel. PUG's main drawback is that it explodes in complexity when confronted with a growing number of threads during functional correctness checking (PUG often times out on four threads). This makes it very difficult to downscale a kernel and check it. While modeling a small number (e.g., two) threads often suffices for race checking, it is almost always impossible to express functional correctness over such small thread populations.

In [14] we present a tool called GKLEE and a checking methodology that dramatically improves the capabilities in this area. GKLEE is the first concolic verifier and test generator for CUDA GPU programs. GKLEE detects several forms of data races, bank conflicts, non-coalesced memory accesses, deadlocks, and also reports thread/warp divergences accurately. GKLEE employs a new schedule generation method consisting of a canonical sequential schedule interlaced with SIMD execution within each thread warp. This avoids the exponentially of general schedule generation, and detects bugs without omissions or false alarms. GKLEE generates tests that guarantee code coverage, assisted by many new GPU-specific test minimization heuristics. GKLEE has found bugs and issues in CUDA SDK kernels, and can handle multi-kernel examples. *However, GKLEE still does not offer parameterized verification capabilities,* exceeding normally allocated amounts of computational resources on many small to medium examples at about 2K threads.

We show in this paper that taking a different approach to SMT-encoding than PUG or GKLEE can result in a practically feasible parameterized verification approach. We show that for many kernels, this method (called PUG$_{para}$) vastly outperforms our previous methods. *In our new parameterized approach, only one (parameterized) thread is modeled*. Our tool PUG$_{para}$ based on this approach tracks *how data flows through the threads in consecutive computational rounds*. Over these rounds, it symbolically reasons about the possible values of shared variables contributed by all threads. From one perspective, *it implicitly implements the Omega Test~[20] using SMT techniques*. When this checking approach applies to a kernel, it is sound (no false alarms will be reported). We also propose an over-approximation approach to combat the capacity limits of SMT solvers, in order to locate bugs quickly.

One of the main applications of our method is to check the equivalence of a kernel and its optimized version. This parameterized method is particularly suitable for handling typical optimizations for CUDA kernels such as memory coalescing and bank conflict elimination (which often preserve the loop structures).

* Work done as part of the author's PhD dissertation at Utah

In essence, our parameterized method makes full use of the symmetric nature of SIMD computations. In the SIMD model, within each (synchronized) step, each thread performs similar computations on different data. Thus, the behavior of all the threads can be inferred by investigating one arbitrary thread. After one execution round, the threads exchange data and go the next round. Modeling such exchanges by an arbitrary number of threads is challenging; we rely on symbolic matching and SMT solving to address this problem in this paper.

We organize the paper by first presenting the generic, non-parameterized approach extended from [13], then present the parameterized approach. We then compare the performance of these approaches on realistic CUDA programs.

II. Background and Motivating Examples

A CUDA kernel is launched as an 1D or 2D *grid* of *thread blocks*. The total size of a 2D grid is `gridDim.x` × `gridDim.y`. The coordinates of a (thread) block are ⟨`blockIdx.x`, `blockIdx.y`⟩. The dimensions of each thread block are `blockDim.x` and `blockDim.y`. Each block contains `blockDim.x` × `blockDim.y` threads, each with coordinates ⟨`threadIdx.x`, `threadIdx.y`⟩. These threads can share information via *shared memory*, and synchronize via *barriers* (__syncthreads()). Threads belonging to distinct blocks must use the much slower *global memory* to communicate, and may not synchronize using barriers.

The values of `gridDim` and `blockDim` determines the *configuration* of the system, *e.g.* the sizes of the grid and each block. For a thread, `blockIdx` and `threadIdx` give its block index in the grid and its thread index in the block respectively. For brevity, we use *gdim*, *bid*, *bdim*, and *tid* for *gridDim*, *blockIdx*, *blockDim* and *threadIdx* respectively. Clearly, constraints $bid.* < gdim.*$ for $* \in \{x, y\}$ and $tid.* < bdim.*$ for $* \in \{x, y, z\}$ always hold.

Consider the following simple example with 2D blocks, which is simplified from the "transpose" kernel in CUDA SDK 2.0 [7]. Note that a variable with modifier `shared` is "global" for all threads within a block; and private variables have no modifiers.

```
void naiveTranspose (int *odata, int* idata,
                     int width, int height) {
  int xIndex = bid.x * bdim.x + tid.x;
  int yIndex = bid.y * bdim.y + tid.y;
  if (xIndex < width && yIndex < height) {
    int index_in  = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;
    odata[index_out] = idata[index_in];
  }
  int i, j;       // for the post-condition
  postcond(i < width && j < height =>
    odata[i * height + j] == idata[j * width + i]);
}
```

The threads transpose the array in parallel: each thread reads $idata$ at location $(bid.x * bdim.x + tid.x) + width * (bid.y * bdim.y + tid.y)$ and writes it to $odata$ at location $(bid.y * bdim.y + tid.y) + height * (bid.x * bdim.x + tid.x)$. The functional correctness of this kernel is specified in the post-condition: the element at location $j * width + i$ in the input array $idata$ must be deposited into location $i * height + j$ in the output array $odata$. This property should hold *for all valid configurations* as well as *all possible input values*.

This naïve kernel suffers from non-coalesced writes, and can be more than 10x slower than the following optimized kernel for large matrices. The kernel below is optimized to ensure that all global reads and writes are coalesced, and avoids bank conflicts in the shared memory. The computations between two consecutive barriers constitute a *barrier interval (BI)* or *round*. This example contains two rounds of computation.

```
void OptimizedTranpose (int *odata, int *idata,
                        int width, int height) {
  __shared__ float block[bdim.x][bdim.x+1];
  // read the matrix tile into shared memory
  int xIndex = bid.x * bdim.x + tid.x;
  int yIndex = bid.y * bdim.y + tid.y;
  if((xIndex < width) && (yIndex < height)) {
    int index_in = yIndex * width + xIndex;
    block[tid.y][tid.x] = idata[index_in];
  }
  __syncthreads();
  // write the transposed tile to global memory
  xIndex = bid.y * bdim.y + tid.x;
  yIndex = bid.x * bdim.x + tid.y;
  if ((xIndex < height) && (yIndex < width)) {
    int index_out = yIndex * height + xIndex;
    odata[index_out] = block[tid.x][tid.y];
  }
}
```

We may use the same post-condition as before to specify the functional correctness of this optimized kernel. Moreover, the equivalence of these two kernels can be specified as: suppose the two kernels take the same inputs, *i.e.* the same $idata$, $width$ and $height$, then after execution they produce the same outputs (in $odata$) *for all possible configurations*. The main challenge here is to show this for any number of threads and any input value.

A. Related Work

Verification of CUDA Kernels. Traditional testing methods are ineffective at producing guarantees because they assume concrete input values as well as a fixed numbers of threads, and examine only those concurrency schedules produced by the execution environment. Past efforts in thread verification have focused on multi-threaded programs synchronizing using locks and semaphores [9]. These methods are inapplicable for GPU kernels. Our work is tailored for CUDA which is very widely used; it will easily apply to emerging standards (*e.g.*, OpenCL~[16]).

There are only few GPU-specific checkers reported in the past. Table I gives a comparison of these tools. An instrumentation based technique is reported [4] to find races and shared memory bank conflicts. This is a dynamic

Comparison Categories	PUG$_{para}$ (extend from [13])	GKLEE [14]	[4] (GRace [27])
Methodology	Symbolic Analysis	Concolic Exec. in virtual machine	Dyn. Check (+ Static Analysis)
Level of Analysis	Source Code	LLVM Bytecode	Source Code Instrument.
Bugs Targeted	Race, Func. Corrct., Equiv. Check	Corrct. & Perf. Bugs	Race, Bank Conflict
Program Inputs	Fully Symbolic	Symbolic + Concrete	No Symbolic
Parameterized?	Yes (for both Race and Equiv. Check)	No	No

Table˜I
COMPARISON OF FORMAL VERIFIERS OF GPU PROGRAMS

testing approach in which the program is instrumented with checking code, and only those executions occurring on a specific platform are considered. A similar method [27] is used to find races assisted by static analysis. Static analysis is performed first to locate possible candidates so as to reduce the runtime overheads caused by instrumented code. These runtime methods cannot accept symbolic inputs and verify function correctness on open inputs, not to mention handling an arbitrary number of threads.

GKLEE [14] builds a virtual machine (VM) modeling the thread computations on the GPU. When a GPU program is executed in the VM, the tool checks deadlocks, several forms of data races, and performance bugs (*e.g.* bank conflicts, non-coalesced memory accesses, thread/warp divergences). The execution in the VM considers only a fixed numbers of threads; hence only a portion of valid configurations are examined. Moreover, GKLEE executions often exceed memory/time limits on many small to medium examples containing non-trivial branches. For example, the Bitonic-Sort kernel (of about 50 lines of code) will cause blow-up when the thread number is greater than 8.

As far as race checking and bank conflict detection goes, the techniques used in PUG can easily accommodate the use of symbolic thread identifiers. However, these straight-forward extensions do now work for functional equivalence checking.

The KLEE-FP tool [6] handles OpenCL code. Its main use is in crosschecking OpenCL code against an initial scalar sequential version, and also for race detection in such code. Its approach to floating-point equivalence is based on expression normalization. KLEE-FP is not a parameterized checker, however. Its floating-point reasoning methods can be incorporated into PUG$_{para}$ which currently lacks the ability to handle float numbers.

Parameterized Verification. There are abstraction based techniques [19], [5] that help reduce the problem of verifying parameterized systems with infinite states to that of checking corresponding finite-state abstractions. The abstraction methods employed include counter abstraction [19] which helps abstract process identities, or environmental abstraction [5], which provides an abstract counting method for the number of processes satisfying a given predicate. These techniques require manual effort to obtain the abstractions. They also do not directly apply to GPUs.

There are efforts [1], [18] that apply automatic induction to generate and verify invariants pertaining to parameterized systems. In most cases, manual effort is required to obtain the invariants, although Pnueli *et al.* [18] presented a way to automatically compute invariants given an appropriate abstraction relation. None of these methods consider CUDA-style computations.

The reduction from infinite states to equivalent finite states in these works is based on finding an appropriate cut-off k of the parameter of the system. The goal is to establish that a property is satisfied by k processes if and only if it is satisfied by any number ($> k$) of processes. For example, [10] proposes tighter bounds of cut-off for parameterized systems, independent of the communication topology. In contrast, our technique considers only one parameterized thread and does not require symmetry reduction.

Equivalence Checking. Many approaches have been proposed for checking the equivalence of two sequential programs. For instance, equivalence checkers [21], [24] perform a dependence graph abstraction of programs containing only affine loops. The basic idea is to use the Omega test to check whether the relations depicting the dependence graphs are equal. Unfortunately, the Omega test approach supports only linear arithmetic. Since these works do not exploit decision procedures, they are unable to handle many arithmetic transformations. They can only deal with programs with high similarities.

TVOC [2] first verifies loop transformations using a specific proof rule called Permute, and then verifies structure-preserving optimizations. It relies on extra information supplied by the compiler to generate verification conditions which are fed to an SMT solver for satisfiability checking. Zaks and Pnueli [26] also used SMT solving to verify structure-preserving optimizations. Their verifier attempts to find invariants connecting the models of the two programs. However, it is difficult to identify sufficiently precise invariants for non-trivial optimizations. Also, these checkers can handle only sequential programs.

An equivalence checking method for CUDA kernels is discussed in [23]. It makes many assumptions and restrictions on the input programs and is not parameterized. No implementation of this method is reported.

III. SMT Encoding and Non-parameterized Checking

Although CUDA kernels are concurrent programs executing in parallel, CUDA programmers often intend to write *deterministic* programs whose final results are independent of the concurrent schedule. We have presented a static checker [13] and a symbolic executor [14] to determine whether a program is deterministic; and also proved that a deterministic program could be serialized such that the accesses on shared variables are executed in a sequential order (*i.e.* the canonical schedule). In this section we give a different (and slightly better) order to sequentialize the shared variable accesses. Unlike the one in [13], this order does not use symbolic arrays to support schedule ids and requires only local information of the accesses. This encoding serves as the basis for comparing the parameterized method and the non-parameterized one.

A. Encoding Sequential Structures

Basic Statements. Our encoding assigns SSA indices to variables. Specifically, the following translation function 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, and $v \uplus ([i] \mapsto x)$ denotes the update of array v by setting the element at i to x. Note that a write to an array variable is actually modeled as an array update. We also give below a simple example of applying Γ.

$$\begin{array}{lcl}
\Gamma(e_1 \;\texttt{op}\; e_2) & \doteq & \Gamma(e_1) \;\texttt{op}\; \Gamma(e_2) \\
\Gamma(v := e) & \doteq & v_{\texttt{next}(v)} = \Gamma(e) \\
\Gamma(v[e_1] := e_2) & \doteq & v_{\texttt{next}(v)} = \\
 & & v_{\texttt{cur}(v)}([\Gamma(e_1)] \mapsto \Gamma(e_2)) \\
\Gamma(v) & \doteq & v_{\texttt{cur}(v)}
\end{array}$$

$$\begin{array}{lcl}
\text{int k = 0;} & & k_1 = 0 \\
\text{int a[3];} & & i_1 = a_0[1] + k_1 \\
\text{int i = a[1] + k;} & \xrightarrow{\Gamma} & a_1 = a_0([0] \mapsto i_1 * k_0) \\
\text{a[0] = i * k;} & & i_2 = i_1 + 1 \\
\text{i++;} & &
\end{array}$$

Branches. The SSA indices of the variables updated in the two clauses of a conditional statement "if $c\ blk_1$ else blk_2" should be synchronized so that subsequent statements have a consistent view of their values. The following example gives an illustration: $i_1 = i_0$ is added into the first clause and only variable i_1 will be referenced later. Here notation `ite` stands for "`if then else`".

$$\begin{array}{l}
\texttt{if}\ i > 0 \\
\quad \{\ j = i * 10;\ \ k = j - i;\ \}\quad \xrightarrow{\Gamma} \\
\texttt{else} \\
\quad i = j + k; \\
\texttt{ite}\quad (i_0 > 0, \\
\qquad j_1 = i_0 * 10 \ \wedge\ k_1 = j_1 - i_0 \ \wedge\ i_1 = i_0, \\
\qquad i_1 = j_0 + k_0 \ \wedge\ j_1 = j_0 \ \wedge\ k_1 = k_0 \\
\qquad)
\end{array}$$

Our checker handles other structures including variable aliasing, static scopes and function calls. More details are available at [13].

Serializing Concurrent Executions We now illustrate the translation of `shared` variable updates in concurrent executions. Suppose we have to translate a global assignment `v[tid.x] = tid.x + 1` where v is a shared array. Note that n threads are being allowed to concurrently perform this assignment. On the other hand, since no data race exists and the program is deterministic, we can specify an order in which the assignments are executed by assigning SSA indexes to v. A typical order is to have the threads execute the assignments with respect to their thread ids: thread 0 executes first, then thread 1 executes, ..., finally thread $n-1$ executes. Such order is called the *natural order*.

$$v_1[0] = 0 + 1 \ \wedge\ v_2[1] = 1 + 1 \ \wedge\ \ldots\ \wedge\ v_n[n-1] = n - 1 + 1$$

Now consider a more complicated example where v is the only shared variable. As usual we assume that no data races occur on v. In the first round, all threads execute `v[i] = v[j] + tid.x`. After all threads finish this assignment, the second round containing `v[k]++` starts execution.

```
v[i] = v[j] + tid.x;   __syncthreads();   v[k]++;
```

The natural order generates the following constraint.

$$\begin{array}{lcl}
\text{Thread } t_0 & \ldots & \text{Thread } t_{n-1} \\
v_1[i] = v_0[j] + 1 & \ldots & v_n[i] = v_{n-1}[j] + n \\
v_{n+1}[k] = v_n[k] + 1 & \ldots & v_{2n}[k] = v_{2n-1}[k] + 1
\end{array}$$

Formally, the combined transition system for n threads is

$$\begin{array}{l}
\texttt{trans}(t_x, n) \equiv v_{x+1}[i] = v_x[j] + 1 \ \wedge\ v_{n+x+1}[k] = v_{n+x}[k] + 1 \\
\texttt{TRANS}(t, n) \equiv \bigwedge_{x \in [0,n-1]} \texttt{trans}(t_x, n)\ .
\end{array}$$

We give below the model of the `naiveTranpose` kernel. Each thread has a *private* copy of local variables such as $xIndex$. They are referred to by $xIndex^{s_i}$ in each thread s_i. Similarly we can obtain the model $\texttt{TRANS}^t(t, n)$ for the optimized kernel (we use s and t to refer to the source (naive) and target (optimized) kernel respectively). The encoding of the post-condition is trivial and not shown here.

$$\begin{array}{l}
\texttt{trans}(s_i, n) \equiv \\
\quad \text{xIndex}_1^{s_i} = bid^{s_i}.x * bdim.x + s_i.x \ \wedge \\
\quad \text{yIndex}_1^{s_i} = bid^{s_i}.y * bdim.y + s_i.y \ \wedge \\
\quad \texttt{ite}(\text{xIndex}_1^{s_i} < \text{width}_0 \ \wedge\ \text{yIndex}_1^{s_i} < \text{height}_0, \\
\qquad\quad \text{index\_in}_1^{s_i} = \text{xIndex}_1^{s_i} + \text{width}_0 * \text{yIndex}_1^{s_i} \ \wedge \\
\qquad\quad \text{index\_out}_1^{s_i} = \text{yIndex}_1^{s_i} + \text{height}_0 * \text{xIndex}_1^{s_i} \ \wedge \\
\qquad\quad \text{odata}_{i+1}^{s} = \text{odata}_i^{s} \uplus ([\text{index\_out}_1^{s_i}] \mapsto \text{idata}_0^{s}[\text{index\_in}_1^{s_i}]), \\
\qquad\quad \text{odata}_{i+1}^{s} = \text{odata}_i^{s}) \\
\texttt{TRANS}^s(s, n) \equiv \bigwedge_{i \in [0,n-1]} \texttt{trans}(s_i, n)
\end{array}$$

Equivalence Checking Given the models $\texttt{TRANS}^s$ and $\texttt{TRANS}^t$ for two kernels with inputs $\vec{i}$ and output $\vec{o}$, the kernels are equivalent if and only if the following constraint holds. We subscript the variables in the source and target kernel with s or t respectively.

$$\forall n.\, \texttt{TRANS}^s(s,n) \wedge \texttt{TRANS}^t(t,n) \wedge (\vec{i}^s = \vec{i}^t) \Rightarrow (\vec{o}^s = \vec{o}^t)\ .$$

Unfortunately, an SMT solver is unable to handle this quantified formula since the definition of `TRANS` is recursive over the number of threads and the solver requires a concrete n to unroll the recursion. This also forbids using induction (*e.g.* k-induction [17]) to to perform the proof. Moreover, the fact that our translation conjoins the models of n threads will make SMT solving quite complex (and of course it would not lead to a parametric approach).

The Assertion Language (for Property Checking) In addition to equivalence checking, PUG$_{para}$ also checks properties specified as assertions (*e.g.* in the postconditions). Our assertion language supports the definition of Boolean formulas using C syntax. One of the main features of this assertion language is that it allows the definition of loops, handling recursive properties and variables with symbolic values. For instance, consider a reduction kernel which computes the sum of the elements in the input array $idata$ and stores this sum in $odata$. A suitable post-condition specifying functional correctness is the following, where n is the number of elements in $idata$.

```
for (i = 1; i ≤ n; i++) {odata += idata[i];}
```

In some cases, functional correctness can be specified recursively. Consider a `scan` kernel which computes the parallel prefix sum of the input elements. We show below a suitable postcondition.

$$g\_odata[0] = 0 \wedge$$
$$(0 < i < n-1 \Rightarrow g\_odata[i+1] = g\_odata[i] + g\_idata[i])$$

IV. Parameterized Checking

This section describes how to perform parameterized encoding. The key is to calculate the value of an output element regardless of the number of threads.

A. Single Conditional Assignment

Our method builds a symbolic model according to the accesses on shared arrays. We first present a method which eliminates all the intermediate variables so that only the accesses on shared arrays are left (an optimization is presented in Section IV-C). For example, the body of the `naiveTranspose` contains a conditional assignment (CA) to $odata$ as follows.

```
if (bid.x * bdim.x + tid.x < width &&
    bid.y * bdim.y + tid.y < height)
  odata[(bid.y * bdim.y + tid.y) +
        height * (bid.x * bdim.x + tid.x)] =
  idata[(bid.x * bdim.x + tid.x) +
        width * (bid.y * bdim.y + tid.y)];
```

This can be interpreted by an mapping from $odata$ to $idata$. Let $c(tid)$, $addr_d(tid)$ and $addr_s(tid)$ denote the condition $bid.x * bdim.x + tid.x < width \ \wedge \ bid.y * bdim.y + tid.y < height$, the destination address $(bid.y * bdim.y + tid.y) + height * (bid.x * bdim.x + tid.x)$ and the source address $(bid.x * bdim.x + tid.x) + width * (bid.y * bdim.y + tid.y)$ respectively. This CA can be denoted as $c\ ?\ odata[addr_d] := idata[addr_s]$, where $odata[addr_d]$ and $idata[addr_s]$ are called the *range* and *domain* of the CA respectively. Now consider the k^{th} element in the output array, $odata[k]$. Its value comes from either (1) $idata[addr_s(s_i)]$ for some s_i provided that $k = addr_d(s_i)$ and the guard holds (there is only one such s_i since no race occurs on $idata$); or (2) the old value of $odata[k]$ if $\nexists s_i : k = addr_d(s_i) \ \wedge \ c(s_i)$. For brevity we write $p(s_i)$ for the predicate $(k = addr_d(s_i)) \ \wedge \ c(s_i)$. The diagram in Figure 1 indicates how $odata[k]$ is computed: if p holds for thread s_1, then $odata[i] = idata[addr_s(s_1)]$, otherwise thread s_2 is investigated, and so on. Here we use the "xor" operator $\oplus$ to emphasize that at most one thread satisfies p. If no thread satisfies p, then the old value of $odata[k]$ is used. As before we will use SSA indices to subscript the accesses, *e.g.* $odata_1$ denote the first write to $odata$.

Key Observation: The approach so far seems to suggest the enumeration of n threads. However, observe that *since there exists no conflict, at most one thread will satisfy p.* Therefore, *we can build an SMT constraint considering only one thread* (with symbolic ID s_i).

$$(\exists s_i : p(s_i)) \Rightarrow odata_1[i] = idata_0[addr_s(s_i)] \text{ for that } s_i$$
$$(\forall s_i : \neg p(s_i)) \Rightarrow odata_1[k] = odata_0[k]$$

Note that, for a given s_i, $\neg p(s_i)$ does not necessarily indicates that $odata[k]$ takes its old value — only there exists no such s_i will the value of $odata[k]$ be unchanged. Thus we cannot conclude that $odata_1[k] = \texttt{ite}(p(s_i),\ idata_0[addr_s(s_i)],\ odata_0[k])$. Instead, $odata_1[k] = odata_0[k]$ only if p doesn't hold for all s_i.

Unfortunately, existing SMT solvers often fail to handle quantified formulas (they return an inconclusive answer "unknown"). To overcome this limitation and make sure that our verifier gives conclusive answers, we derive unquantified formulas from the quantified ones and use them as the constraints. From the first formula we can derive $p(s_i) \Rightarrow odata_1[k] = idata_0[addr_s(s_i)]$ for a fresh variable s_i, which indicates that, for any s_i, if $p(s_i)$ is true then $odata[k]$'s value comes from $idata_0[addr_s(s_i)]$. The absence of conflicts enables us to eliminate the $\exists$ quantifier by introducing the fresh variable s_i. For the second formula we apply the approach detailed in section IV-D.

Formal Status: It should be noted that unsolved quantified formulas may lead to under-approximations but over-approximations: if PUG$_{para}$ reports a bug, then this bug is real; if a kernel is correct, then PUG$_{para}$ won't report a bug. However PUG$_{para}$ may fail to reveal some bugs in a kernel.

$$odata[k] = \begin{array}{c} p(s_1) \\ \circ \\ idata[addr_s(s_1)] \end{array} \oplus \begin{array}{c} p(s_2) \\ \circ \\ idata[addr_s(s_2)] \end{array} \begin{array}{c} \ldots \\ \ldots \\ \ldots \end{array} \begin{array}{c} p(s_n) \\ \circ \\ idata[addr_s(s_n)] \end{array} \oplus \begin{array}{c} \text{else} \\ \circ \\ odata_{old}[k] \end{array}$$

Figure˜1.˜ Calculating CAs over multiple threads.

We call the derived formulas *Verification Conditions*. Section IV-D presents additional details of our technique.

B. Instantiation of Conditional Assignments (CA)

Now consider a more complicated case where an expression contains multiple instances of a shared variable. For example, $v[a_1]$ op $v[a_2]$, where op is a binary operator, reads variable v twice—at addresses a_1 and a_2 respectively. Let us assume that immediately preceding these reads, there exists a conditional assignment (CA) $p\ ?\ v[e] := w$. The question is: what is the value of $v[a_1]$ op $v[a_2]$ in terms of w? Or more specifically, suppose p is a predicate on $v[a_1]$ op $v[a_2]$; then what is the value of $v[a_1]$ op $v[a_2]$ in terms of w?

As indicated in Figure 2, for the first read $v[a_1]$, we introduce a fresh variable s_1 to denote the ID of the thread writing the value to $v[a_1]$. In other words, $(p(s_1)\ \wedge\ a_1 = e(s_1)) \Rightarrow v[a_1] = w(s_1)$. For the second read $v[a_2]$, note that we cannot use the same s_1 because the write may come from another thread. Thus we introduce another fresh variable s_2 for the thread writing to $v[a_2]$ such that $(p(s_2)\ \wedge\ a_2 = e(s_2)) \Rightarrow v[a_2] = w(s_2)$. These two formulas connect expression $v[a_1]$ op $v[a_2]$ with w such that the value of this expression can be obtained from two instantiations (one for s_1 and the other for s_2) of w. In general, if an expression contains n reads from variable v, then n fresh variables and n formulas are created.

Considering only these two formulas, one verification condition for $P(v[a_1]\ \text{op}\ v[a_2])$ is shown below. It reduces the checking on $v[a_1]$ and $v[a_2]$ to that on $w_1(s_1)$ and $w_2(s_2)$.

$$\begin{array}{c} p(s_1)\ \wedge\ a_1 = e(s_1)\ \wedge\ p(s_2)\ \wedge\ a_2 = e(s_2)\ \Rightarrow \\ P(w(s_1)\ \text{op}\ w(s_2)) \end{array}$$

For instance, consider the optimized `Transpose` kernel. Let $X(i)$ and $Y(i)$ be $bid.x{*}bdim.x{+}i$ and $bid.y{*}bdim.y{+}i$ respectively. This kernel contains two CAs:

```
if (X(tid.x) < width && Y(tid.y) < height)
  block[tid.y][tid.x] =
  idata[Y(tid.y) * width + X(tid.x)];

if (Y(tid.x) < height && X(tid.y) < width)
  odata[X(tid.y) * height + Y(tid.x)] =
  block[tid.x][tid.y];
```

The value of the i^{th} output element $odata[i]$ may be tracked back to an element in the $block$ first, then to an element in the $idata$. That is, it can be obtained by the sequential composition of the two CAs. We instantiate the tids in the first and second assignments to be t_1 and t_2 respectively (recall that we use t rather than s for this optimized kernel). An important point here is to match the first CA's range $block_1^t[t_2.x][t_2.y]$ and the CA's domain $block_1^t[t_1.y][t_1.x]$ using constraint $t_2.x = t_1.y\ \wedge\ t_2.y = t_1.x$.

$$\begin{array}{l} i = X(t_2.y) * height + Y(t_2.x)\ \wedge \\ (Y(t_2.x) < height \wedge X(t_2.y) < width)\ \Rightarrow \\ \quad odata_1^t[i] = block_1^t[t_2.x][t_2.y] \\ (t_2.x = t_1.y\ \wedge\ t_2.y = t_1.x)\ \wedge\ (X(t_1.x) < width \wedge \\ Y(t_1.y) < height)\ \Rightarrow \\ \quad block_1^t[t_2.x][t_2.y] = idata_0^t[Y(t_1.y) * width + X(t_1.x)] \end{array}$$

We may dig deeper into these formulas. Suppose $X(t.x), Y(t.y) < \min(width, height)$ holds for any t, *i.e.* thread t accesses data within the bounds of the 2-D input array with height $height$ and $width$, then the above two formulas become

$$\begin{array}{l} i = X(t_2.y) * height + Y(t_2.x)\ \Rightarrow odata_1^t[i] = block_1^t[t_2.x][t_2.y] \\ (t_2.x = t_1.y\ \wedge\ t_2.y = t_1.x)\ \Rightarrow \\ block_1^t[t_2.x][t_2.y] = idata_0^t[Y(t_1.y) * width + X(t_1.x)]\ . \end{array}$$

If each block of threads is a square such that $bdim.x = bdim.y$, then we can derive the following formula justifying the correctness of the optimized kernel – the input array is correctly transposed *no matter how many threads are considered.* Note that this kernel is designed with implicit assumptions that (1) each block is square; and (2) only those threads with tid t satisfying $X(t.x), Y(t.y) < \min(width, height)$ should participate in the computation. In fact, *our encoding models exactly this design and also helps reveal hidden assumptions.* For example, PUG$_{para}$ reports a bug when the block is not square by relying on the following check for valid configurations:

$$\begin{array}{c} odata_1^t[X(t_1.x) * height + Y(t_1.y)] = \\ idata_0^t[Y(t_1.y) * width + X(t_1.x)] \end{array}$$

Equivalence Checking. The equivalence of the two example kernels requires $odata_1^s[i] = odata_1^t[i]$ provided that all above constraints hold and $idata_0^s = idata_0^t$. Note that only $odata[i]$ is instantiated only once for each kernel. We need to reducing the checking on $odata_1^s[i]$ and $odata_1^t[i]$ to that on the elements in the input array $idata$.

C. Barrier Interval and Control Flow

The statements between two consecutive barriers are within a *Barrier Interval* (BI). Since there are no conflicts within a BI, writes to the same shared variable will not fall on the same address. We may use this fact to simplify the generated constraints. Consider the following diagram where BI 1 contains multiple writes to v and in BI 2 property P reads v.

$$\begin{tikzcd}
p(s_1) \mathbin{?} v[e(s_1)] := w(s_1) & & p(s_2) \mathbin{?} v[e(s_2)] := w(s_2) \\
& P(v[a_1] \text{ op } v[a_2]) \arrow[ul, "p(s_1) \wedge a_1{=}e(s_1)"'] \arrow[ur, "p(s_2) \wedge a_2{=}e(s_2)"] &
\end{tikzcd}$$

Figure˜2.˜ Instantiation of conditional assignments.

	$p_1 \mathbin{?} v[e_1] := w_1$
BI 1	$p_2 \mathbin{?} v[e_2] := w_2$
	$\dots$
	$p_n \mathbin{?} v[e_2] := w_n$
BI 2	$P(v[a])$

The non-conflicting assumption indicates that at most one $v[e_i]$ would match $v[a]$. Thus instead of writing a pair of constraints for each CA, we can combine all the CA constraints to be an embedded `ite` expression (here v_1 and v_0 represent v's value right before BI 1 and BI 2 respectively). The main benefit is now we have only one quantified formula rather than n formulae.

In some cases (*e.g.* the two `Transpose` kernels) the quantified formula is not needed at all because $v[a]$'s value comes from one of the writes in BI 1.

$$\begin{array}{l}
\text{ite}(a = e_1 \ \wedge \ p_1, \ P(w_1), \\
\quad \text{ite}(a = e_2 \ \wedge \ p_2, \ P(w_2), \\
\qquad \dots)) \quad \text{and} \\
(\nexists i \in [1, n] : a = e_i \ \wedge \ p_i) \Rightarrow P(v_0[a])
\end{array}$$

A further optimization we employed is to keep the control flow of the BI and not eliminate all intermediate variables. The program below (the left column) contains two conditional jumps. Instead of flattening this program to generate three CAs: $c_1 \wedge c_2 \ ? \ v[e_1] = w_1$, $c_1 \wedge \neg c_2 \ ? \ v[e_2] = w_2$ and $\neg c_2 \ ? \ v[e_3] = w_3$, we keep this control flow structures and generate the constraint as shown on the right. This representation, which mimics those in Section III, reduces substantially the size of the constraints and make them much more readable.

```
if ($c_1$) {                          ite($c_1$,
   if ($c_2$) $v[e_1] = w_1$;           ite($c_2$,
   else    $v[e_2] = w_2$;                  $a = e_1 \Rightarrow P(w_1)$,
}                                           $a = e_2 \Rightarrow P(w_2)$,
else $v[e_3] = w_3$;                     $a = e_3 \Rightarrow P(w_3)$))
```

D. Quantified Formulas

We attempt to convert a quantified formula into an equivalent quantifier-free formula whenever possible. One quantified formulas we encountered so far is of the following format, where t is the thread id with domain $[1..n]$, f is a function of t, c is the guard on t, a is an expression not involving t, and P is a predicate indicating the value of a variable is unchanged:

$$(\forall t \in [1..n] : \neg(a = f(t) \ \wedge \ c(t))) \ \Rightarrow \ P$$

We introduce a function $g : \text{int} \to \text{int}$ by defining $g(t) = a$ if $(a = f(t)) \ \wedge \ c(t)$ and $g(t) = \text{undefined}$ otherwise. That is, $g(t)$ returns the address a satisfying $(a = f(t)) \ \wedge \ c(t)$. Let the integer space $\mathbb{S}$ be $\{g(t) \mid t \in [1..n]\}$, *i.e.* the set of all addresses obtained by applying g on the thread IDs. In a typical CUDA kernel, function g is an increasing or decreasing function. Without loss of generality we assume g is increasing. Usually the space $\mathbb{S}$ is discrete such that $\forall t \in [1..n) : \exists v : a(i) < v < a(i+1)$. The fact that there exists no t satisfying $a = g(t)$ is equivalent to there exists a t such that a falls between $g(t)$ and $g(t+1)$ (note that we need to extend g's definition to $t = 0$ and $t = n + 1$ here).

$$\begin{array}{l}
(\forall t \in [1..n] : \neg(a = g(t))) \Longleftrightarrow \\
(\exists t \in [0\,..\,n] : g(t) < a < g(t+1)) \\
\text{where } g \text{ is an increasing function}
\end{array}$$

Moreover, there exists at most one such t since g is an increasing function. In order to obtain an un-quantified verification condition, we can introduce a fresh variable t to eliminate the $\exists$ quantifier to obtain the final verification condition.

$$t \in [0\,..\,n] : g(t) < a < g(t+1) \ \Rightarrow \ P$$

It is not hard to see that the g functions for the two `Transpose` kernels are increasing and their quantified formulas can be converted in this manner. In fact, under valid configurations (*e.g.* the block is of square size), their spaces $\mathbb{S}$ are continuous over the thread IDs, thus the quantified formulas will never be used and can be safely removed.

Fast Bug Hunting. If quantifier elimination is impossible, then we can further loosen the requirement of *proving* the properties. Our goal then would be to locate property violations quickly by ignoring the quantified formula.

Consider the following sequence. Even the quantified formulas are nonconvertible, we know conclusively that $P(f(w))$ should be true if both $e_3 = e_4 \ \wedge \ p_2$ and $e_2 = e_1 \ \wedge \ p_1$ hold. Thus any violation of the predicate $(e_3 = e_4 \ \wedge \ p_2 \ \wedge \ e_2 = e_1 \ \wedge \ p_1) \ \Rightarrow \ P(f(w))$ reveals a real bug. PUG$_{para}$ is able to find such bugs fast.

$$p_1 \mathbin{?} v[e_1] := w; \qquad p_2 \mathbin{?} v[e_3] := f(v[e_2]); \qquad \text{assert } P(v[e_4])$$

Coverage. Our approach may be criticized on many grounds. For example, our parameterized method employs under-approximation because of the inability of solvers to handle quantified formulas. Yet our encoding ensures that all (conditional) assignments are covered. With our proposed quantifier elimination technique and the techniques described in Section IV-C, all combinations (conjunctions) of the CAs in different BIs are encoded. We believe that in practice PUG$_{para}$ will miss few bugs.

Contrast with Omega Tests. An Omega Test [20] based approach may be used to match the address of a read and the range of a CA by building a relation (over the thread IDs) from the address to the range $\{address \rightarrow range \mid cond\}$. The main advantage of this approach is that it won't generate quantified formulas. However, Omega Tests only support linear expressions while non-linear expressions are prevalent in CUDA kernels (*e.g.* in the two `Transpose` kernels). Our SMT-based method can be regarded as an alternative to Omega Tests to handle non-linear expressions (of particular types).

E. Loops

Our method works better for kernels containing no loops. For kernels with loops, a naïve solution is to fully unroll the loop. However, loop unrolling may not scale, especially with nested loops. Also, loop bounds may involve symbolic values, making it impossible to perform loop unrolling without assigning concrete values to relevant inputs. Our solution is to align the loops or down-size the iteration space.

The loop problem becomes much less severe in equivalence checking. Typical CUDA optimizations often preserve the loop structures of the source kernel such that we may just need to compare the bodies of the loops. A similar assumption is made in [26]. For example, we can optimize the following loop where $sdata$ is a shared array

```
for(unsigned int k = bdim.x / 2; k > 0; k >>= 2) {
  if ((tid.x % (2*k)) == 0)
    sdata[tid.x] += sdata[tid.x + k];
  __syncthreads();
}
```

to the one below by eliminating the slow modulo arithmetic:

```
for(unsigned int k = 1; k < bdim.x; k *= 2) {
  int index = 2 * k * tid.x;
  if (index < bdim.x)
    sdata[index] += sdata[index + k];
  __syncthreads();
}
```

Since the operator $+$ in the body is commutative and associative, the two loop headers can be normalized to be the same. Then the two respective CAs are as follows; on them, the equivalence checking approach discussed in previous sections can be used:

$$s.x \,\%\, (2*k) = 0 \quad ? \;\; s.x := s.x + k$$
$$2*k*t.x < bdim.x \;\; ? \;\; 2*k*t.x := 2*k*t.x + k$$

When the loop alignment fails, we unroll the loops fully (to the given bounds). This happens when optimizations other than memory coalescing and bank conflict elimination are applied. We plan to port the method in [2] to deal with other typical loop transformations.

A notion of program products, cross-products, is proposed in [26]. The compiler correctness verification is reduced to checking a single program which synchronizes the original and transformed programs. However, the restriction of this cross-product based approach to structurally similar programs limits its applicability to structure-preserving transformations. Another line of work [3] extended this method to handle more asynchronous programs by (manually) setting up the synchronous points and inserting invariants into the product program. Our method is similar to these two works in the sense that we also need to align the loops to obtain structurally similar programs.

It is still a non-trivial challenge to parametrically verify structurally dissimilar sequential programs, and optimizations not conforming to specific patterns. We believe that we can overcome some of the limitations of our approach by introducing a richer set of inference rules (*e.g.* for complicated loop optimizations). Typical transformation rules can be verified once and for all, or over each execution [15], [12].

Symmetry Reduction. In many cases, loop bounds in a CUDA kernel depends on the size of a block. As many CUDA kernels are designed to run on arbitrarily-sized blocks~[8], one can expect to be able to reduce block sizes to a reasonable value before running PUG$_{para}$. Currently, such downscaling is done manually. We plan to develop an automatic symmetry reduction approach to identify, for a property p, the minimum number of threads n for which p should be checked. We believe that parameterized analysis and SMT-based analysis can help determine n in practice.

V. Experimental Results

PUG$_{para}$ uses Z3 [25] as the SMT solver. Z3's expressions are based on bit vectors (bounded integers); thus the solving time depends on the number of bits.

We performed experiments on a laptop with an Intel Core(TM)2 Duo 1.60GHz processor and 2GB memory to check some representative kernels in CUDA SDK 2.0 Suite [7], each of which contains both unoptimized and optimized kernels. Table II shows the SMT solving time in seconds. Here n denotes the number of GPU threads. The Transpose kernels are not equivalent when n is not a perfect square; we mark these cases by a '$*'$. The reduction kernels contains loops whose upper bounds depend on n, making the generic method blow up on n. Notation $16b$ indicates that 16-bit bit-vectors are used; T.O denotes Time Out ($>$ 5 minutes). These benchmarks employ the multiplication operation extensively, and hence, the analysis time of PUG$_{para}$ is quite

Kernel	**Non-parameterized**				**Parameterized**	
	n = 4	8	16(+C.)	32(+C.)	-C.	+C.
Transpose (8b)	<1	$<1^*$	7.3	15.4^*	T.O	<0.1
Transpose (16b)	28	$<1^*$	T.O(1.2)	$37(14.3)^*$	T.O	<0.1
Transpose (32b)	T.O	1.5^*	T.O(4.3)	T.O(31)	T.O	0.16
Reduction (8b)	1	41	T.O(T.O)	T.O(T.O)	0.2	0.2
Reduction (12b)	21	T.O	T.O	T.O	15	11

Table~II
EVALUATION RESULTS FOR EQUIVALENCE CHECKING OF SOME SDK KERNELS (BUG FREE VERSIONS).

sensitive to the sizes of the bit-vectors. This may cause even the parameterized method to time-out. In this case, we must concretize some of the symbolic variables (*i.e.* give them concrete values, indicated by the "+C." flag) and then compare the results.

Our testing addresses two kinds of bugs. The first kind is due to incorrect configurations for running kernels: for example, using a non-square block (for the Tranpose kernel); or, using a value of ACCN that is not a power of 2 (in the Scalar Product kernel). PUG$_{para}$ is able to reveal violations of these assumptions. The second class of bugs is those intentionally introduced within correct kernels, *e.g.* by modifying the addresses of accesses on shared variables or the guards of conditional statements.

Table III compares our parameterized and non-parameterized approaches. Not surprisingly, the parameterized method is much faster.

Kernel	**Non-param.**			**Param.**
	n = 4	8	16	
Transpose (16b)	0.16	0.53	2.7	<0.1
Transpose (32b)	0.54	1.8	7.9	0.26
Reduction (8b)	0.2	3.4	T.O	<0.1
Reduction (16b)	0.3	7	T.O	<0.1
Reduction (32b)	0.8	9.2	T.O	0.1

Table~III
EVALUATION RESULTS FOR EQUIVALENCE CHECKING OF SOME SDK KERNELS (BUGGY VERSIONS).

PUG$_{para}$ has checked more kernels than shown in above tables, some of which come from GPGPU programming classes. Although they are small-medium sized programs (typically 50-200 lines of code), it still is non-trivial to verify some of these highly optimized parallel programs. Furthermore, even for a small kernel, loop unrolling often results in many CAs to be checked. It is encouraging that PUG$_{para}$ was able to identify bugs within few seconds on some of these kernels.

VI. CONCLUDING REMARKS

In this paper, we detailed several directions for developing parameterized equivalence verification methods for GPU programs. We then presented specific details pertaining to PUG$_{para}$, the first such parameterized checker (so far as we know). Using PUG$_{para}$, we have obtained many encouraging preliminary results on several small-to-medium real-world kernels. As to our future work, many extensions are planned. In addition to finding better ways to handle quantified formulas and non-trivial loop transformations, we plan to extend PUG$_{para}$ to deal with more complicated programs, and incorporate it into our symbolic executor GKLEE. We also plan to extend PUG$_{para}$ to support floating-point numbers.

REFERENCES

[1] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore~D. Zuck, *Parameterized verification with automatically computed inductive assertions*, International Conference on Computer Aided Verification (CAV), 2001.

[2] Clark~W. Barrett, Yi~Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore~D. Zuck, *TVOC: A translation validator for optimizing compilers*, International Conference on Computer Aided Verification (CAV), 2005.

[3] Gilles Barthe, Juan~Manuel Crespo, and César Kunz, *Relational verification using product programs*, 17th International Symposium on Formal Methods (FM), 2011.

[4] Michael Boyer, Kevin Skadron, and Westley Weimer, *Automated dynamic analysis of CUDA programs*, Third Workshop on Software Tools for MultiCore Systems, 2008.

[5] Edmund~M. Clarke, Muralidhar Talupur, and Helmut Veith, *Proving ptolemy right: The environment abstraction framework for model checking concurrent systems*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008.

[6] Peter Collingbourne, Cristian Cadar, and Paul Kelly, *Symbolic testing of OpenCL code*, Haifa Verification Conference (HVC), 2011.

[7] *CUDA zone. www.nvidia.com/object/cuda_home.html.*

[8] *Cuda programming guide version 1.1*, Chapter 6, Section 6.2 on Matrix Multiplication.

[9] Cormac Flanagan and Stephen~N. Freund, *Type-based race detection for Java*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2000.

[10] Youssef Hanna, Samik Basu, and Hridesh Rajan, *Behavioral automata composition for automatic topology independent verification of parameterized systems*, 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE), 2009.

[11] David˜B. Kirk and Wen mei W.˜Hwu, *Programming massively parallel processors*, Morgan Kauffman, 2010.

[12] Guodong Li, *Validated compilation through logic*, 17th International Symposium on Formal Methods (FM), 2011, www.cs.utah.edu/~ligd/VCL.

[13] Guodong Li and Ganesh Gopalakrishnan, *Scalable SMT-based verification of GPU kernel functions*, ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT FSE), 2010, www.cs.utah.edu/fv/PUG.

[14] Guodong Li, Peng Li, Geof Sawaga, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga˜P. Rajan, *GKLEE: Concolic verification and test generation for GPUs*, 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2012, www.cs.utah.edu/fv/GKLEE.

[15] Guodong Li and Konrad Slind, *Trusted source translation of a total function language*, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008.

[16] OpenCL. http://www.khronos.org/opencl.

[17] Lee Pike, *Real-time system verification by k-induction*, Tech. Report TM-2005-213751, NASA Langley Research Center, May 2005, Available at http://www.cs.indiana.edu/~lepike/pub_pages/reint.html.

[18] Amir Pnueli, Sitvanit Ruah, and Lenore˜D. Zuck, *Automatic deductive verification with invisible invariants*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2001.

[19] Amir Pnueli, Jessie Xu, and Lenore˜D. Zuck, *Liveness with (0, 1, infty)-counter abstraction*, International Conference on Computer Aided Verification (CAV), 2002.

[20] William Pugh, *The omega test: a fast and practical integer programming algorithm for dependence analysis*, ACM/IEEE conference on Supercomputing (SC), 1991.

[21] K.˜C. Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens, *Verification of source code transformations by program equivalence checking*, 14th Conference on Compiler Construction (CC), 2005.

[22] *Satisfiability Modulo Theories Competition (SMT-COMP). http://www.smtcomp.org/2009.*

[23] Stavros Tripakis, Christos Stergiou, and Roberto Lublinerman, *Checking non-interference in SPMD programs*, 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar), 2010.

[24] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe, *Equivalence checking of static affine programs using widening to handle recurrences*, International Conference on Computer Aided Verification (CAV), 2009.

[25] *Z3: An SMT solver. research.microsoft.com/en-us/um/redmond/projects/z3.*

[26] Anna Zaks and Amir Pnueli, *CoVaC: Compiler validation by program analysis of the cross-product*, 15th International Symposium on Formal Methods (FM), 2008.

[27] Mai Zheng, Vignesh˜T. Ravi, Feng Qin, and Gagan Agrawal, *GRace: A low-overhead mechanism for detecting data races in GPU programs*, 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2011.