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 parameterically, for an arbitrary number of threads. Our tool PUGpara 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 PUGpara) vastly outperforms our previous methods. In our new parameterized approach, only one (parameterized) thread is modeled. Our tool PUGpara 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 \( \text{gridDim.x} \times \text{gridDim.y} \). The coordinates of a (thread) block are \((\text{blockIdx.x}, \text{blockIdx.y})\). The dimensions of each thread block are \(\text{blockDim.x} \times \text{blockDim.y}\). Each block contains \(\text{blockDim.x} \times \text{blockDim.y}\) threads, each with coordinates \((\text{threadIdx.x}, \text{threadIdx.y})\). 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 \(\text{gridDim}\) and \(\text{blockDim}\) determines the configuration of the system, e.g. the sizes of the grid and each block. For a thread, \(\text{blockIdx.x}\) and \(\text{threadIdx.x}\) give its block index in the grid and its thread index in the block respectively. For brevity, we use \(\text{gdim}\), \(\text{bid}\), \(\text{bdim}\), and \(\text{tid}\) for \(\text{gridDim}, \text{blockIdx}, \text{blockDim}\) and \(\text{threadIdx}\) respectively. Clearly, constraints \(\text{bid}.x < \text{gdim}.x\) for \(* \in \{x, y\}\) and \(\text{tid}.x < \text{bdim}.x\) 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.

```c
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 + yIndex;
        odata[index_out] = 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 = xIndex + yIndex;
        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 \(\text{idata}, \text{width}\) and \(\text{height}\), then after execution they produce the same outputs (in \(\text{odata}\)) 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
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 straightforward
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 PUGpara 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
the corresponding finite-state abstractions. The abstraction
methods employed include counter abstraction [19] which
helps abstract process identities, or environmental abstrac-
tion [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 pro-
posed for checking the equivalence of two sequential pro-
grams. 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 spe-
cific proof rule called Permute, and then verifies structure-
preserving optimizations. It relies on extra information sup-
plied 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 invari-
ants 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 restric-
tions 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 \( \Gamma \) 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 \( v \psi ([i] \mapsto x) \) 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 \( \Gamma \).

\[
\begin{align*}
\Gamma(e_1 \ op \ e_2) & \equiv \Gamma(e_1) \ op \ \Gamma(e_2) \\
\Gamma(v := e) & \equiv v_{\text{next}(v)} = \Gamma(e) \\
\Gamma(v[i] := e_2) & \equiv v_{\text{next}(v)} = v_{\text{cur}(v)}(\Gamma(e_2)) \\
\Gamma(v) & \equiv v_{\text{cur}(v)} \\
\end{align*}
\]

Branches. The SSA indices of the variables updated in the two clauses of a conditional statement \( \text{if } c \ blk_1 \ 	ext{else } blk_2 \) 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 \( \text{ite} \) stands for \( \text{"if then else"} \).

\[
\begin{align*}
\text{ite} \ (i_0 > 0, \ j_1 = i_0 * 10 \ \& \ k_1 = j - i; \ j = j + i, \ k = k + 1) & \rightarrow \nabla \\
\end{align*}
\]

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, etc. Finally thread \( n-1 \) executes. Such order is called the natural order.

\[
v_1[0] = 0 + 1 \ \& \ v_2[1] = 1 + 1 \ \& \ ... \ \& \ v_n[n-1] = n - 1 + 1
\]

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.

\[
\begin{align*}
\text{Thread } t_0 : & \quad v_1[i] = 0 + 1 \quad ... \quad v_{n-1}[i] = v_{n-1}[j] + n \\
\text{Thread } t_n : & \quad v_n[k] = v_n[k] + 1 \quad ... \quad v_2n[k] = v_2n[k] + 1
\end{align*}
\]

Formally, the combined transition system for \( n \) threads is

\[
\text{TRANS}(t_x, n) \equiv \forall x_{i+1} \equiv v_{x_i}[j] + 1 \ \& \ v_{n+x_i+1}[k] = v_{n+x_i}[k] + 1 \quad \text{TRANS}(t, n) \equiv \bigwedge_{x \in [0, n-1]} \text{TRANS}(t_x, n).
\]

We give below the model of the naive transpose 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 \( \text{TRANS}'(t, n) \) 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.

\[
\begin{align*}
\text{TRANS}'(s, n) & \equiv \exists xIndex^{s_i} = \text{bid}^{s_i}.x \ * \ \text{bdim}.x + s_i.x \ \& \ \text{yIndex}^{s_i} = \text{bid}^{s_i}.y \ * \ \text{bdim}.y + s_i.y \\
\text{ite}(\text{xIndex}^{s_i} < \text{width}_0 \ \& \ \text{yIndex}^{s_i} < \text{height}_0 \ \& \ \text{index}_{in}^{s_i} = \text{xIndex}^{s_i} \ + \ \text{width}_0 \ \& \ \text{yIndex}^{s_i} \ + \ \text{index}_{out}^{s_i} = \text{yIndex}^{s_i} \ + \ \text{height}_0 \ \& \ \text{index}_{out}^{s_i} = \text{yIndex}^{s_i} \ + \ \text{height}_0 \ \& \ \text{index}_{in}^{s_i} \ \& \ \text{odata}_{in}^{s_i} = \text{odata}_{in}^{s_i} \ \& \ \text{data}_{in}^{s_i} \ (\text{index}_{out}^{s_i} \ + \ \text{height}_0) \ \& \ \text{odata}_{out}^{s_i} = \text{odata}_{out}^{s_i} \ + \ \text{height}_0)
\end{align*}
\]

Equivalence Checking Given the models \( \text{TRANS}' \) and \( \text{TRANS} \) for two kernels with inputs \( \vec{r} \) and output \( \vec{d} \), 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.

\[ \forall n. \text{TRANS}^s(n, n) \land \text{TRANS}^t(t, n) \land (\bar{t}^3 = \bar{n}^3) \Rightarrow (\bar{a}^s = \bar{a}^t). \]

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 $idata$.

\[
\text{for } (i = 1; i \leq n; i++) \{ odata += idata[i]\}
\]

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.

\[
g_{\text{odata}}[0] = 0 \land 
(0 < i < n - 1) \Rightarrow g_{\text{odata}}[i+1] = g_{\text{odata}}[i] + g_{\text{idata}}[i]
\]

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.

\[
\begin{align*}
\text{if } (\text{bid}.x \times \text{bdim}.x + \text{tid}.x \times \text{width} \&\& \\
\text{bid}.y \times \text{bdim}.y + \text{tid}.y \times \text{height}) \\
\text{odata}[\text{bid}.y \times \text{bdim}.y + \text{tid}.y] + \\
\text{height} \times (\text{bid}.x \times \text{bdim}.x + \text{tid}.x) = \\
\text{idata}[\text{bid}.x \times \text{bdim}.x + \text{tid}.x] + \\
\text{width} \times (\text{bid}.y \times \text{bdim}.y + \text{tid}.y);
\end{align*}
\]

This can be interpreted by an mapping from $odata$ to $idata$. Let $c(tid)$, $addr_d(tid)$ and $addr_s(tid)$ denote the condition $\text{bid}.x \times \text{bdim}.x + \text{tid}.x < \text{width} \land \text{bid}.y \times \text{bdim}.y + \text{tid}.y < \text{height}$, the destination address $(\text{bid}.y \times \text{bdim}.y + \text{tid}.y) + \text{height} \times (\text{bid}.x \times \text{bdim}.x + \text{tid}.x)$ and the source address $(\text{bid}.x \times \text{bdim}.x + \text{tid}.x) + \text{width} \times (\text{bid}.y \times \text{bdim}.y + \text{tid}.y)$ respectively. This CA can be denoted as $c$.

\[
\text{oadata}[	ext{addr}_d] := \text{idata}[	ext{addr}_s], \text{ where } \text{oadata}[	ext{addr}_d] \text{ and } \text{idata}[	ext{addr}_s] \text{ are called the range and domain of the CA respectively. Now consider the } k^{th} \text{ element in the output array, oadata}[k]. \text{ Its value comes from either (1) idata}[	ext{addr}_s(s_1)] \text{ for some } s_1 \text{ provided that } k = \text{addr}_d(s_1) \text{ and the guard holds (there is only one such } s_1 \text{ since no race occurs on odata); or (2) the old value of odata}[k] \text{ if } \exists s_i : k = \text{addr}_d(s_i) \land c(s_i). \text{ For brevity we write } p(s_i) \text{ for the predicate } (k = \text{addr}_d(s_i)) \land c(s_i). \text{ The diagram in Figure 1 indicates how oadata}[k] \text{ is computed: if } p \text{ holds for thread } s_1, \text{ then oadata}[i] = \text{idata}[	ext{addr}_s(s_1)], \text{ otherwise thread } s_2 \text{ is investigated, and so on. Here we use the "xor" operator } \oplus \text{ to emphasize that at most one thread satisfies } p. \text{ If no thread satisfies } p, \text{ then the old value of odata}[k] \text{ is used. As before we will use SSA indices to subscript the accesses, e.g. odata}_1 \text{ denote the first write to odata.}
\]

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 $s_i$).

\[
(\exists s_i : p(s_i)) \Rightarrow odata_1[i] = idata_0[\text{addr}_s(s_i)] \text{ for that } s_i, \text{ and } (\forall s_i : \neg p(s_i)) \Rightarrow odata_1[k] = odata_0[k]
\]

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 $odata_0[k] = \text{ite}(p(s_i), idata_0[\text{addr}_s(s_i)], odata_0[k])$. Instead, $odata_0[k] = odata_0[k]$ 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 $p(s_i) \Rightarrow odata_1[k] = idata_0[\text{addr}_s(s_i)]$ 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 $idata_0[\text{addr}_s(s_i)]$. 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.
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, \( v[a_1] \) or \( v[a_2] \), 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 \( v[a_1] \) or \( v[a_2] \) in terms of \( w \)? Or more specifically, suppose \( p \) is a predicate on \( v[a_1] \) or \( v[a_2] \); then what is the value of \( v[a_1] \) or \( v[a_2] \) 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, \( (p(s_1) \land a_1 = e(s_1)) \Rightarrow v[a_1] = w(s_1) \). 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 \( (p(s_2) \land a_2 = e(s_2)) \Rightarrow v[a_2] = w(s_2) \). These two formulas connect expression \( v[a_1] \) or \( v[a_2] \) 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 \( P(v[a_1] \lor v[a_2]) \) is shown below. It reduces the checking on \( v[a_1] \) and \( v[a_2] \) to that on \( w_1(s_1) \) and \( w_2(s_2) \).

\[
p(s_1) \land a_1 = e(s_1) \land p(s_2) \land a_2 = e(s_2) \Rightarrow P(w_1(s_1) \lor w_2(s_2))
\]

For instance, consider the optimized Transpose kernel. Let \( X(i) \) and \( Y(i) \) be \( bid.x \times bidm.x + i \) and \( bid.y \times bidm.y + i \) respectively. This kernel contains two CAs:

- if \( (X(tid.x) < \text{width} \&\& Y(tid.y) < \text{height}) \)
  \[
  \text{block}[tid.y][tid.x] = \text{idata}[Y(tid.y) \times \text{width} + X(tid.x)];
  \]

- if \( (Y(tid.x) < \text{height} \&\& X(tid.y) < \text{width}) \)
  \[
  \text{odata}[X(tid.y) \times \text{height} + Y(tid.x)] = \text{block}[tid.x][tid.y];
  \]

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 \( \text{block}^1_{1}[t_2.x][t_2.y] \) and the CA’s domain \( \text{block}^1_{0}[t_1.y][t_1.x] \) using constraint \( t_2.x = t_1.y \land t_2.y = t_1.x \).

\[
i = X(t_2.y) \times \text{height} + Y(t_2.x) \land \ (Y(t_2.x) < \text{height} \land X(t_2.y) < \text{width}) \Rightarrow \text{odata}^1[i] = \text{block}^1_{1}[t_2.x][t_2.y]
\]

\[
(t_2.x = t_1.y \land t_2.y = t_1.x) \land (X(t_1.x) < \text{width} \land Y(t_1.y) < \text{height}) \Rightarrow \text{block}^1_{0}[t_1.x][t_1.y] = \text{idata}^1[Y(t_1.y) \times \text{width} + X(t_1.x)]
\]

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

\[
i = X(t_2.y) \times \text{height} + Y(t_2.x) \Rightarrow \text{odata}^1[i] = \text{block}^1_{1}[t_2.x][t_2.y]
\]

\[
(t_2.x = t_1.y \land t_2.y = t_1.x) \Rightarrow \text{block}^1_{0}[t_1.x][t_1.y] = \text{idata}^1[Y(t_1.y) \times \text{width} + X(t_1.x)]
\]

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 \( X(t.x), Y(t.y) < \text{min(width, height)} \) should participate in the computation. In fact, our encoding models exactly this design and also helps reveal hidden assumptions. For example, \( \text{PUG}_{para} \) reports a bug when the block is not square by relying on the following check for valid configurations:

\[
\text{odata}^1[X(t_1.x) \times \text{height} + Y(t_1.y)] = \text{idata}^1[Y(t_1.y) \times \text{width} + X(t_1.x)]
\]

**Equivalence Checking.** The equivalence of the two example kernels requires \( \text{odata}^1[i] = \text{odata}^1[i] \) provided that all above constraints hold and \( \text{idata}^1 = \text{idata}^0 \). Note that only \( \text{odata}^1[i] \) is instantiated only once for each kernel. We need to reducing the checking on \( \text{odata}^1[i] \) and \( \text{odata}^1[i] \) to that on the elements in the input array \( \text{idata} \).

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 \( v \).
\[ p(s_1) \land v[e(s_1)] := w(s_1) \quad \text{and} \quad p(s_2) \land v[e(s_2)] := w(s_2) \]

Figure 2. Instantiation of conditional assignments.

\[
\begin{array}{c|c}
\text{BI 1} & p_1 \land v[e_1] := w_1 \\
p_2 \land v[e_2] := w_2 \\
\text{BI 2} & P(v[1]) \land P(v[2]) \\
\end{array}
\]

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.

\[
\begin{align*}
\text{ite}(a = e_1 \land p_1, P(w_1), \\
\text{ite}(a = e_2 \land p_2, P(w_2), \\
\ldots) \text{ and} \\
(\exists t \in [1..n] : a = e_i \land p_i) \Rightarrow P(v_i[a])
\end{align*}
\]

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: \( c_1 \land c_2 \land v[e_1] = w_1 \), \( c_1 \land \neg c_2 \land v[e_2] = w_2 \) and \( \neg c_2 \land v[e_3] = w_3 \), 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.

\[
\begin{align*}
\text{if } (c_1) \{ \\
\phantom{\text{if } (c_1) } \text{ite}(e_1, \text{ite}(e_2, a = e_1 \Rightarrow P(w_1), a = e_2 \Rightarrow P(w_2), a = e_3 \Rightarrow P(w_3))) \\
\text{else } v[e_3] = w_3;
\end{align*}
\]

D. Quantified Formulas

We attempt to convert a quantified formula into an equivalent quantifier-free formula whenever possible. One quantified formula 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:

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

We introduce a function \( g : \text{int} \to \text{int} \) by defining \( g(t) = a \) if \( (a = f(t)) \land c(t) \) and \( g(t) = \text{undefined otherwise} \). That is, \( g(t) \) returns the address \( a \) satisfying \( (a = f(t)) \land c(t) \).

Let the integer space \( S \) 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 \( S \) 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).

\[
(\forall t \in [1..n] : \neg(a = g(t))) \iff (\exists t \in [0..n] : g(t) = a < g(t+1))
\]

where \( g \) is an increasing function

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 \( S \) 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 \( e_3 = e_4 \land p_2 \) and \( e_2 = e_1 \land p_1 \) hold. Thus any violation of the predicate \( (e_3 = e_4 \land p_2 \land e_2 = e_1 \land p_1) \Rightarrow P(f(w)) \) reveals a real bug. \( \text{PUG}_{\text{para}} \) is able to find such bugs fast.

\[
p_1 \land v[e_1] := w_1; \quad p_2 \land v[e_3] := f(v[e_2]); \quad \text{assert } P(v[e_4])
\]

Coverage. Our approach may be criticized on many
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

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

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. These benchmarks employ the multiplication operation extensively, and hence, the analysis time of \( \text{PUG}_{\text{para}} \) is quite high.
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 Transpose 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.

### Table II

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Non-parameterized</th>
<th>Parameterized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n = 4 8 16(+C.)</td>
<td>-C.  +C.</td>
</tr>
<tr>
<td>Transpose (8b)</td>
<td>&lt;1 &lt;1 7.3</td>
<td>TO &lt;0.1</td>
</tr>
<tr>
<td>Transpose (16b)</td>
<td>28 &lt;1&quot; T.O(1.2)</td>
<td>37(14.3)&quot; TO&lt;0.1</td>
</tr>
<tr>
<td>Transpose (32b)</td>
<td>T.O 1.5&quot; T.O(4.3)T.O(31)</td>
<td>TO 0.16</td>
</tr>
<tr>
<td>Reduction (8b)</td>
<td>1 41 T.O(T.O) T.O(T.O) 0.2 0.2</td>
<td></td>
</tr>
<tr>
<td>Reduction (12b)</td>
<td>21 T.O T.O T.O 15 11</td>
<td></td>
</tr>
</tbody>
</table>

Table III

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Non-param.</th>
<th>Param.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n = 4 8 16</td>
<td></td>
</tr>
<tr>
<td>Transpose (16b)</td>
<td>0.16 0.53 2.7</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>Transpose (32b)</td>
<td>0.54 1.8 7.9</td>
<td>0.26</td>
</tr>
<tr>
<td>Reduction (8b)</td>
<td>0.2 3.4 T.O</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>Reduction (16b)</td>
<td>0.3 7 T.O</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>Reduction (32b)</td>
<td>0.8 9.2 T.O</td>
<td>0.1</td>
</tr>
</tbody>
</table>

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


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


