Scalable SMT-Based Verification of GPU Kernel Functions *

Guodong Li
School of Computing, University of Utah
UT, USA
ligd@cs.utah.edu

Ganesh Gopalakrishnan
School of Computing, University of Utah
UT, USA
ganesh@cs.utah.edu

ABSTRACT

Interest in Graphical Processing Units (GPUs) is skyrocketing due to their potential to yield spectacular performance on many important computing applications. Unfortunately, writing such efficient GPU kernels requires painstaking manual optimization effort which is very error prone. We contribute the first comprehensive symbolic verifier for kernels written in CUDA C. Called the 'Prover of User GPU programs (PUG),' our tool efficiently and automatically analyzes real-world kernels using Satisfiability Modulo Theories (SMT) tools, detecting bugs such as data races, incorrectly synchronized barriers, bank conflicts, and wrong results. PUG's innovative ideas include a novel approach to symbolically encode thread interleavings, exact analysis for correct barrier placement, special methods for avoiding interleaving generation, dividing up the analysis over barrier intervals, and handling loops through three approaches: loop normalization, overapproximation, and invariant finding. PUG has analyzed over a hundred CUDA kernels from public distributions and in-house projects, finding bugs as well as subtle undocumented assumptions.

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

General Terms: Reliability, Verification

Keywords: CUDA, GPU, Formal Verification, Concurrency, Satisfiability Modulo Theories (Decision Procedures)

1. INTRODUCTION

There is an explosive growth of interest in *Graphical Processing Units (GPU)* for speeding up computations occurring at all application scales [10, 14]. GPUs are used in iPhones for video processing, and on desktop computers for extracting features from medical images. All future supercomputers will employ GPUs. The main attraction of GPUs is that *when properly programmed*, they can yield anywhere from 20 to 100 times more performance compared to standard CPU based multi-cores. Unfortunately, obtaining this performance requires heroic acts of programming; to name a few: (i) one must keep all the fine-grained GPU threads busy; (ii) one must ensure coalesced [14] data movements from the *global memory* (that is accessed commonly by CPUs and GPUs) to the *shared memory* (that is accessed commonly by the GPU threads); and (iii) one must minimize bank conflicts when the GPU threads step through the shared memory. Data races and incorrect barrier placements are frequently introduced during CUDA programming. Few tools are available to verify CUDA programs. The emulator that comes with GPUs assumes concrete inputs and executes only a miniscule fraction of all possible schedules. Bugs often escape, either crashing or deadlocking the GPU hardware, often requiring a hardware reboot.

GPU kernels are comprised of light-weight threads. Their Single Instruction Multiple Data (SIMD) organization bears little resemblance to thread programs written in C/Java with their heterogeneous and heavy-weight threads, and use of synchronization primitives such as locks/monitors. This requires a fundamentally new approach for analyzing CUDA kernels. This paper's main result is that while Satisfiability Modulo Theories (SMT [22]) techniques are a natural choice for analyzing CUDA kernels, many innovations are essential before such analysis can scale. Efficient techniques for encoding concurrent interleavings and analyzing barrier placement must be developed. One must try to exploit the "mostly deterministic" style of programming and avoiding interleaving generation. It is efficient to divide up the analysis over barrier intervals. Finally, techniques for efficiently handling loops (rather than simply unrolling them) must be developed. We now begin with a few CUDA examples and elaborate our innovations.

Illustration of CUDA. 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` (assuming 1D or 2D blocks in this paper). 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. *This paper focuses on shared memory races.* Consider a sim-

*Supported in part by Microsoft, SRC TJ 1847.001, and NSF CCF 0935858.

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

ple example of a CUDA kernel to add `b` to all the elements of a shared array `a` of size `N`:

```
void __global__ kernel (int *a, int b) {
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < N) a[idx] = a[idx] + b;}
```

Basically, each thread accesses a different array location and adds `b` to it in parallel; *there are no data races*. Now imagine the programmer wanting to update each array location with `b` added to the previous array location. The programmer may not simply change the last line to `a[idx] = a[idx-1] + b;` because there will be data races between adjacent threads. The programmer may however change the code to the following:

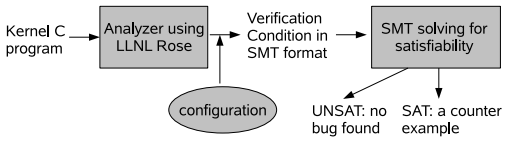
```
void __global__ kernel1 (int *a, int b) {
__shared__ int temp[N];
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < N) temp[idx] = a[idx-1] + b;
__syncthreads(); // A barrier
if (idx < N) a[idx] = temp[idx];}
```

What if the barrier is removed from this code? Obviously the accesses of $a[idx]$ and $a[idx-1]$ by different threads may cause a race. This can be detected by examining the symbolic models of two threads as following, where private variables in a thread are superscripted by the thread id, *bid* and *bdim* are the short hands for `blockIdx` and `blockDim` respectively. Threads t_1 and t_2 are assumed to be in the same block. Formally, a race occurs if predicate $t_1.x \neq t_2.x \wedge id^{t_1} < N \wedge id^{t_2} < N \wedge idx^{t_1} - 1 = idx^{t_2}$ holds. As all variables have symbolic values, we can consult with a constraint solver to determine whether this predicate is satisfiable. If so, then the solver would return a concrete counter example. If the barrier is present then we only need to check whether the writes to $a[idx^{t_1}]$ and $a[idx^{t_2}]$ conflict. Since $t_1.x \neq t_2.x$ implies $idx^{t_1} \neq idx^{t_2}$ for $t_1.x < bdim.x$ and $t_2.x < bdim.x$, these two writes will not result in a race.

thread t_1	thread t_2
$idx^{t_1} = bid.x * bdim.x + t_1.x$	$idx^{t_2} = bid.x * bdim.x + t_2.x$
$if\ (idx^{t_1} < N)$ read $a[idx^{t_1} - 1]$	$if\ (idx^{t_2} < N)$ read $a[idx^{t_2} - 1]$
$if\ (idx^{t_1} < N)$ write $a[idx^{t_1}]$	$if\ (idx^{t_2} < N)$ write $a[idx^{t_2}]$

As another example, the `scalarProdGPU` (Figure 1) kernel computes the scalar product of `vN` pairs of vectors with `eN` elements in each vector (both sequential and CUDA parallel versions are shown). This kernel coalesces global memory accesses, minimizes bank conflicts, avoids redundant barriers, and reduces serial penalties through tree summation. Without such hand-crafting steps, kernels such as this will perform poorly. In this paper, we present our tool PUG that helps detect bugs introduced during kernel design.

Internal Architecture of PUG. PUG takes a kernel program written in C (called Kernel C) as input. It first uses the Rose Compiler [21] to parse the kernel and generates an immediate format, then produces an SMT expression according to the configuration information supplied (*e.g.* the properties to be checked or the number of threads). We consider only two threads with symbolic identifiers (IDs) for race and synchronization checking. Users must specify the number of threads for assertion (user-defined property) checking. The PUG generated SMT expressions are processed by an SMT solver (currently Yices [24]) for satisfiability checking. If the expression is satisfiable, the solver will return a concrete counter-example; otherwise the kernel is deemed free of the bugs targeted by our analysis.



Organization. We now list some of our novel contributions, each of which is later elaborated in its own section.

- PUG employs a C front-end based on the LLNL Rose [21] framework (with customized extensions). It handles many CUDA C features including: (i) arrays and records, (ii) loops, conditional statements and function calls, (iii) variable aliases due to pointer expressions, and (iv) lexical scopes. Many features such as heap allocation and recursive calls are not allowed in CUDA, simplifying our translation. § 2
- We contribute a novel approach to capture all possible interleavings between CUDA threads as compact SMT formulae. In practice, working with this SMT representation is far more efficient than explicitly enumerating all schedules. § 3
- We propose a way to model the semantics of barriers exactly. We generate SMT formulae that help verify that despite the presence of branches and loops, all barriers are well synchronized. § 4
- While we have the ability to model all possible concurrent interleavings, *it is preferable to avoid resorting to this approach whenever possible*. Our observation that enables this optimization is based on the fact that in many cases, the existence of races between a given pair of variables is predicated on the existence of conflicts on other variables. The existence of conflicts can be checked over just one canonical interleaving – say the one that simply runs one thread till it blocks and then switching over to another. This helps dramatically improves the overall efficiency. We propose a way to further scale up this approach by analyzing one *barrier interval* (the portion before and after `__syncthreads()`) at a time. This *divide-and-conquer* approach also helps boost efficiency. § 5.
- The translation of loops can become extremely involved – especially if the loops are nested and they employ non-linear strides. Our multi-pronged attack is as follows: (i) we normalize loops through program transformation into a unit-stride loop; (ii) we over-approximate loop computations; and (iii) we can automatically discover *compensating invariants* that compensate for non-linear loop strides frequently found in practice. § 6.
- For many kernels, an SMT tool may generate a false alarm (false bug report) when it cannot determine how the kernel formal parameters are constrained by the main program (caller). For example, PUG assures that the matrix multiplication kernel in the CUDA Programming Guide [7] works only when the size of matrix B is greater or equal to the block size. PUG is able to reveal such undocumented assumptions.
- We have obtained very encouraging results using PUG on real examples. As one example of its multiple uses, with respect to `scalarProdGPU`, we could obtain many valuable analysis results using PUG: (i) One may not remove the

```
void scalarProdSeq // Sequential version
(float *d_C, float *d_A, float *d_B, int vN, int eN) {
1:  for(int vec = 0; vec < vN; vec++){
2:    int vBase = eN * vec; int vEnd  = vBase + eN;
3:    double sum = 0;
4:    for(int pos = vBase; pos < vEnd; pos++)
5:      sum += d_A[pos] * d_B[pos];
6:    d_C[vec] = (float)sum;
7:  }}

// Parallel version: Nvidia CUDAZone site
__global__ void scalarProdGPU (float *d_C, float *d_A,
  float *d_B, int vN, int eN) {
1:  __shared__ float acc[ACC_N];
2:
3:  for(int vec = blockIdx.x; vec < vN; vec += gridDim.x) {
4:    int vBase = eN * vec; int vEnd  = vBase + eN;
5:
6:    for(int i = threadIdx.x; i < ACC_N; i += blockDim.x){
7:     float sum = 0;
8:     for(int pos = vBase + i; pos < vEnd; pos += ACC_N)
9:       sum += d_A[pos] * d_B[pos];
10:    acc[i] = sum;
11:   }
12:
13:   for(int stride = ACC_N / 2; stride > 0; stride >>= 1) {
14:     __syncthreads();
15:     for(int i = threadIdx.x; i < stride; i += blockDim.x)
16:       acc[i] += acc[stride + i];
17:   }
18:
19:   if(threadIdx.x == 0) d_C[vec] = acc[0];
20:  }}
```

Figure 1: Scalar Product: Sequential and CUDA Parallel Versions

barrier on line 14 (it will result in a data race), but this single barrier suffices to remove all races with respect to the variables *d_A*, *d_B*, *d_C* and *acc*. (ii) It is formally guaranteed that no bank conflicts (by different threads) occur in this example for all possible values of vN, eN and ACC_N; (iii) Analysis by PUG helped us confirm the assumption that ACC_N must be a power of two; (iv) We could establish the equivalence of this kernel to `scalarProdSeq` for small instances of the problem parameters.

We have also encountered examples where some kernels have benign races; *i.e.*, they are still functionally correct. PUG has caught some serious (but non-obvious) bugs in beginner examples. It has also handled many large examples from the CUDA SDK site.§ 7 All these examples and PUG itself are freely downloadable [16].

2. ENCODING SERIAL CONSTRUCTS

$prog$	::= $\langle var\_decl \mid fun\_decl \rangle_;$	program
var_decl	::= $[md_v]\ ty\ id_v[= exp]$	variable
fun_decl	::= $ty\ id_f(\langle ty\ id_v \rangle_,) = block$	function
$block$	::= $\{\langle stmt \rangle_;\}$	basic block
$stmt$	::= `if` $exp\ block$ [`else` $block$]	conditional
	| `for`(exp; exp; exp) $block$	loop
	| $block$	
	| var_decl	
	| exp	expression
	| $id_f(\langle exp \rangle_,)$	function call
ty	`int` | $ty*$ | $ty[]$ |	type
md_v	`shared` | `global`	modifier

Figure 2: Summary syntax of Kernel C

This section describes the encoding of serial constructs; it gives the formal semantics of a kernel assuming no concurrency. Concurrency is handled in the next section.

The main syntax of Kernel C is given in Figure 2, and are illustrated by the kernel examples given so far. The notation $\langle term \rangle_{separator}$ (used in *fun_decl*, *block*, etc.) denotes a sequence of *term*'s separated by *separator*. Expression *exp* represents usual C expressions including assignments. Identifiers id_f and id_v represent names of functions and variables respectively. Shared and global variables reside in the GPU and the CPU respectively. A variable declared without modifier is local to each thread. We now present the encoding of sequential program structures.

Basic Statements. Our encoding assigns SSA indexes 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. We also give below a simple example of applying Γ.

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

$$
\begin{array}{l}
\texttt{int k = 0;} \\
\texttt{int a[3];} \\
\texttt{int i = a[1] + k;} \\
\texttt{a[0] = i * k;} \\
\texttt{i++;}
\end{array}
\quad \overset{\Gamma}{\rightarrow} \quad
\begin{array}{l}
k_1 = 0 \wedge \\
i_1 = a_0[1] + k_1 \wedge \\
a_1 = a_0([0] \mapsto i_1 * k_1) \wedge \\
i_2 = i_1 + 1
\end{array}
$$

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 so that later on i_0 is invisible and only variable i_1 will be referred. Here notation `ite` stands for "`if then else`".

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

Such synchronization is done at the join node by inserting the following formula into $\Gamma(blk_1)$ (and similarly to $\Gamma(blk_2)$), where $\texttt{cur}(blk, v)$ returns v's last SSA index in blk.

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

Variable Aliasing.

Variables may be aliased due to the use of pointers or references. Typically, when the formal parameters of a function are of pointer or reference types, the parameters are the aliases of the incoming actual arguments. When converting the programs, we map an alias to its corresponding variable and use the variable rather than the alias. For the alias updated in different paths, we add an `ite` expression at the join. Note that most aliases in CUDA kernels occur at function entry.

```
int a[3]; int *i = a;
int j = i[1] + a[2];
i[0]++;
```
$\xrightarrow{\Gamma}$ $j_1 = a_0[1] + a_0[2] \wedge a_1 = a_0([0] \mapsto a_0[0] + 1)$

However we do not model complicated pointer operations (*e.g.* pointer dereference) although it can be implemented by using a global array to represent the shared memory. Since typical CUDA programs exhibit very limited pointer arithmetic operations, PUG does not encounter this problem in practice.

Scopes and Function Calls. Each basic block has its own scope. A variable should be distinguished from another one with the same name but in a different scope. For this, a variable is prepended by its scope number: ${}^n v$ indicates that v is in scope n. The scope numbers of top level variables are skipped. When a function is inlined, its body constitutes a new scope. In the following example, the top level code consists of a "if" statement, whose left clause (a basic block) contains a call to f. Note that j is passed as a pointer.

```
int f (int i, int* j) {
 int k = i - j;
 return (i * k);
}
if (i > 10) {
 int i = 2;
 int j = f(i, j);
}
```
$\xrightarrow{\Gamma}$ $\neg(i_0 > 10) \vee {}^1i_1 = 2 \wedge {}^2i_1 = {}^1i_1 \wedge {}^2k_1 = {}^2i_1 - {}^1j_1 \wedge {}^1j_1 = {}^2i_1 * {}^2k_1$

3. ENCODING CONCURRENCY

A variable with modifier `shared` is "shared" for all threads within a block. Private variables have no modifiers. We now illustrate the translation of `shared` variable updates.

2-thread translation of `shared` *updates.*

Suppose we have to translate a shared assignment $v = 1$. Note that two threads are being allowed to concurrently perform this assignment. Our approach is to treat v as an array indexed by *Schedule IDs* (SID$\in \{0, 1, 2, \ldots\}$). (If v were an array, we would simply add one more dimension to v indexed by SID.) An SID has the same root name as the variable, but has a subscript and a superscript. It is like a timestamp and combines two pieces of information: which thread is accessing it (superscript), and where in the code the access is occurring (subscript, forming the single static assignment or SSA index [19]). With these, the translation of $v = 1$ is as follows:

$$v = 1 \quad \xrightarrow{\Gamma} \quad v[v_1^{t_1}] = 1 \ \wedge \ v[v_1^{t_2}] = 1$$

Here, the SIDs $v_1^{t_1}$ and $v_1^{t_2}$ range over $\{0, 1\}$. To say that t_1 accesses (writes) into v first, we can throw in the constraint $v_1^{t_1} < v_1^{t_2}$. To say that either access order is possible, we do not throw in any constraint. Now, things get more interesting when we translate $v = v + 1$:

$$v = v + 1; \quad \xrightarrow{\Gamma} \quad \begin{array}{l} v[v_2^{t_1}] = v[v_1^{t_1}] + 1 \ \wedge \ v[v_2^{t_2}] = v[v_1^{t_2}] + 1 \\ \wedge\ (v_2^{t_1} > v_1^{t_1}) \ \wedge \ (v_2^{t_2} > v_1^{t_2}) \wedge \\ v[v_1^{t_1}] = v[v_1^{t_1} - 1] \ \wedge \ v[v_1^{t_2}] = v[v_1^{t_2} - 1] \end{array}$$

and further $v_1^{t_1}$, $v_1^{t_2}$, $v_2^{t_1}$, and $v_2^{t_2}$ should be pairwise distinct and must belong to the set $\{0, \ldots, 3\}$.

First, let us look at the "pairwise distinct" requirement. This can be elegantly modeled by using an un-interpreted function f. More specifically, consider two variables l and m that range over $v_1^{t_1}$, $v_1^{t_2}$, $v_2^{t_1}$, and $v_2^{t_2}$. Then we can say $f(l) \neq f(m)$. Since f is a function, this forces $l \neq m$.

Now what about the rest of the constraints? It is clear that $v[v_2^{t_1}] = v[v_1^{t_1}] + 1$ and $v[v_2^{t_2}] = v[v_1^{t_2}] + 1$ model how "assignment works." It is also clear that $v_2^{t_1} > v_1^{t_1}$ and $v_2^{t_2} > v_1^{t_2}$ model that the L-value is updated only after the R-value is obtained. Now what about the R-value itself? This depends on "who wrote v last." This is precisely why we include $v[v_1^{t_1}] = v[v_1^{t_1} - 1]$ and $v[v_1^{t_2}] = v[v_1^{t_2} - 1]$. It is interesting that this system, in one fell swoop, models all the six schedules possible.

Example: Suppose $v_1^{t_1} = 0$, $v_2^{t_1} = 3$, $v_1^{t_2} = 1$, and $v_2^{t_2} = 2$. Then we have expressed these constraints: $v[3] = v[0] + 1 \ \wedge \ v[2] = v[1] + 1 \ \wedge \ v[1] = v[0]$. In this example, we are modeling the following schedule that, overall, increments v by 1, and not 2: (i) $v[1] = v[0]$ models that thread t_2 also "enjoys" the initial value of v in addition to t_1 (we take $v[-1]$ to be the initial value of v, which is what t_1 gets); (ii) $v[2] = v[1]+1$ models that thread t_2 now does the update of this v; (iii) finally $v[3] = v[0]+1$ models that t_1 now takes the value it had read "long ago," is incrementing that value, and depositing it into v.

An Advanced Example Showing Barrier Encoding.

We now illustrate advanced features of our encoding scheme through an example (details in [16]). In this kernel, `k` is allocated in the `shared` memory.

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

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

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

• To capture the semantics of barriers, we assign them a single SID (*e.g.*, *bar*0 in our example) and constrain them with respect to SIDs of *all* threads.
• Each thread t has a *private* copy of local variables like v. They are referred to by v^t. Since its value is independent of the schedule, there is no SID associated with it.
• We can now derive inequalities to model all these facts (the cases under ORDER are the numbers we refer to here): (1) the program order within each thread must be respected; (2) all the SIDs of all threads constitute a natural number

interval $[0, 4n+1]$ where n is the number of threads; and (3) all the SIDs must be distinct.

A valid schedule of the given example for two threads is depicted below (note that k is the only shared variable):

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

Race Detection. In [16] we present an approach to detect races by encoding *Access IDs* into the formulas. It guarantees that all valid schedules are investigated, a race exhibiting in any particular schedule will not be missed. However it does not scale well [17]; thus we have replaced it with the method described in §5, which needs to consider only one schedule as Feng and Leiserson [9] did for multi-threaded programs represented by series-parallel DAGs.

4. CONDITIONAL BARRIERS AND CROSS-BRANCH CONFLICTS

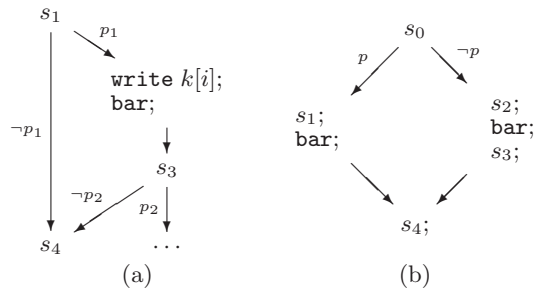


Figure 3: Example CFGs.

The presence of conditional statements makes it imperative that we have the precision of the SMT technology when we check whether all barriers are well-synchronized. It also influences the determination of whether races occur. Work such as [1] which rely purely on static analysis can generate too many false alarms in codes where there are many conditionals.

To illustrate these ideas, consider the control-flow graph (CFG) given in Figure 3(a). This diagram shows how statements s_1 through s_4 are situated in some example program (in (a) s_2 itself is shown expanded in terms of `write k[i]` followed by the barrier `bar`). At first glance, this appears ill-synchronized: one thread may take the s_1 to s_4 path encountering no barriers while another may take the path through p_1 encountering a barrier. Our SMT techniques can determine whether these paths are feasible, and flag an error if so. PUG's approach to checking for well synchronized barriers is as follows: either (i) two branches must execute the same number of barriers; or (ii) all threads must make the same decision on the condition.

In Figure 3(a), if all threads make the same decision on condition p_1 , *i.e.* $\forall t_1, t_2 : p_1^{t_1} = p_1^{t_2}$, then all threads will execute the same branch, which is synchronization safe even if the two branches contain different numbers of barriers. In Figure 3(b), both the left and the right branch contains only one barrier, thus they are considered well synchronized.

Now assume that all barriers are well synchronized. We must now check for conflicting accesses that occur in programs involving conditionals. If for instance the formula $(i^{t_1} = i^{t_2}) \wedge p_1^{t_1} \wedge p_1^{t_2}$ is true in Figure 3(a), both threads can take the p_1 branch *and* conflict on the same k location, causing a race.

A more general analysis is captured by the CFG in Figure 3(b). The conflict check includes the following expressions (here $\not\sim$ denotes *non-conflicting*). Also let us use $p \,?\, s$ to denote an expression s guarded by path condition p. Now, this CFG may be regarded as consisting of two *barrier intervals*: the first one containing s_0, $p \,?\, s_1$ and $\neg p \,?\, s_2$, and the second one containing s_4 and $\neg p \,?\, s_3$. Conflict freedom requires the pairwise comparison of the elements in each barrier interval:

$$p^{t_2} \Rightarrow s_0^{t_1} \not\sim s_1^{t_2} \qquad \neg p^{t_2} \Rightarrow s_0^{t_1} \not\sim s_2^{t_2} \\ p^{t_1} \wedge \neg p^{t_2} \Rightarrow s_1^{t_1} \not\sim s_2^{t_2} \qquad \neg p^{t_2} \Rightarrow s_4^{t_1} \not\sim s_3^{t_2}$$

5. EXPLOITING SERIALIZABILITY AND BARRIER INTERVALS

CUDA programmers often intend to write *deterministic* programs whose final results are independent of the concurrent schedule. Thus it is natural to seek analysis methods that also try to avoid having to generate schedules. Our insights are explained with respect to a simple example:

thread t_1	thread t_2
write $k[i]$;	read v;
...;	...;
write v;	read $k[j]$;

Let us ignore `write v` and `read v` for the moment. Suppose k is the only shared variable. Now if both i and j are (control- and data-) dependent only on thread-local variables, then their values are the same in all schedules. In that case, in order to check whether write $k[i]$ and read $k[j]$ conflict, it suffices to examine only one arbitrary schedule that respects program order.

Now suppose j depends on a shared variable v. Then j's value in thread t_2 may be different in different schedules. *However in this case there exists a conflict on v.* Furthermore, this conflict can be detected by executing v's accesses according to one schedule (any schedule) that simply respects the program order. If we find two accesses within the same barrier interval that conflict, we are done detecting the conflict. This conflict does not go away under another schedule. For this reason we say that k's conflict is *reduced* to v's.

Theorem (Serializability). Consider each pair of accesses to shared variables where one access in the pair is a write. Suppose these access pairs can be shown to be non-conflicting. Then the entire code containing these accesses is race free and can be serialized.

PUG implements such conflict checks and is able to eliminate generating concurrency schedules in our realistic examples. We now show how the ideas in this theorem apply to programs that are decomposed in terms of barrier intervals.

Barrier Intervals (BI) and Incremental Modeling.

CUDA intra-block thread executions exhibit a regular pattern: $\{t_1, \cdots, t_n\}$ execute $\rightarrow$ barrier $\rightarrow$ $\{t_1, \cdots, t_n\}$ execute $\rightarrow \cdots$. Since an access before a barrier will never conflict

with an access after this barrier, we may focus on the accesses between two consecutive barriers (so called a *barrier interval* or *BI*). If the accesses in a BI are non-conflicting, we build a transition constraint by serializing (sequentializing) them; then we move on to the next BI and hope to repeat this treatment. This approach also goes hand in hand with our SMT solver Yices's [24] *incremental SMT solving* facility that reuses existing conflict clauses in the context when checking new expressions. As an illustration we consider the following program *where shared variables are marked with a hat for readability.*

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

Let us consider the case of two threads t_1 and t_2. The first BI consists of statement 1. Since there are no writes to shared variables, accesses to $\widehat{i}$ at t_1 and t_2 are non-conflicting. Both of them can be set to $i[0]$, *i.e.* their SIDs can both be forced to be 0. Using this approach, the transition relation up to statement 2 can be simplified and rewritten as follows (the js are private variables):

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

Now, the second BI consists of a read and a write to shared variable $\widehat{k}$. We need to determine whether their addresses may overlap for different threads. Given $\texttt{TRANS}(t_1, t_2)_2 \wedge t_1 \neq t_2$, expression $j_1^t = i[0]$ is unsatisfiable for $t \in \{t_1, t_2\}$. Therefore the read and write of $\widehat{k}$ do not conflict. Also, we have $j_1^{t_1} = j_1^{t_2}$. Therefore even the writes to $\widehat{k}$ are non-conflicting. We can follow the approach used before and (re-)use the SIDs 0 and 1 for $\widehat{i}$ and $\widehat{j}$ respectively, and write the translation up to statement 4 as:

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

Things are fine if we keep the barrier (__syncthreads) at statement 5. Let us remove it and see what happens. Then, the second BI includes statement write $\widehat{i}^t$. Now, we do not know what write $\widehat{i}^t$ will write into $\widehat{i}^t$. It is possible that expression $j_1^t = \widehat{i}$ can be satisfied. The key observation is that *the conflict between statements 3 and 4 is reducible to a conflict between statements 3 and 6.*

The key point here is that we can keep building constraints without considering interleavings (just by following a canonical interleaving). If there is any race at all in the program, we will reach a point where there will be one conflict somewhere. Since we assume conflicts are rare, this optimistic approach has the ability to process many CUDA kernels successfully without finding any conflicts (and hence races).

In practice, instead of coalescing the SIDs among multiple threads, PUG builds the transitions in a *thread modular* manner: after constructing one single parameterized transition `TRANS(t)`, it instantiates the SIDs with concrete values so as to serialize the concurrent execution of all threads.

We give below the entire model of the example kernel in §3 for n threads. There are two BIs each of which contains only one write. The serialization makes t_i happens before t_j for $i < j$ for each BI. Hence the SIDs of the writes in $t_1, t_2, \ldots, t_n$ in the first BI are $1, 2, \ldots, n$; and those in the second BI are $n+1, n+2, \ldots, 2n$. Clearly this enforces that (1) within a BI, accesses in thread t_i happen before those in t_j for $i < j$; and (2) in a thread, accesses in BI i happen before those in BI j for $i < j$.

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

$$\texttt{TRANS}(t_1, \cdots, t_n) \equiv \textstyle\bigwedge_{x\in[1,n]} \texttt{TRANS}(t_x, n)$$

6. LOOP ABSTRACTION

While it is possible to unroll loops for precise checking, loop unrolling may not scale, especially with nested loops. Also, the loop bounds may involve symbolic values, making it impossible to perform loop unrolling. Consider the scalar product example shown in Figure 1. The outermost loop iterates through every pair of vectors. Each iteration first cycles through vectors with stride `ACC_N`, then performs tree-like reduction of the results. In practice, the grid size, the block size and the stride are large numbers, making it impractical to unroll, particularly the nested loops. One solution is to downscale the problem size by reducing these sizes to small numbers while preserving the program's behaviors (this is tedious if done manually). Another solution – the focus of this section – is to perform loop abstraction to reduce or even eliminate loop unrolling.

6.1 Loop Normalization

A standard result in program analysis [2] is that if the stride part of a loop is a linear function of the loop index i (*i.e.* of format $i = i \pm e$ where e is an expression), then we can *normalize* such loops so they have a stride of one. For example, the loop header

$$\text{for (int i = lb; i} \leq \text{ub; i += stride)}$$

can be normalized to

$$\text{for (int i = 0; i} \leq \text{(ub - lb) / stride; i++),}$$

and each reference to i within the original loop is replaced by $i * stride + lb$. After normalization, the precise value range of the loop index is $[0, (ub - lb)/stride]$. When the stride is not a linear function on the loop index, we do not perform normalization to avoid making the range imprecise. Consider lines 13-17 of the example in Figure 1. Since the stride of the loop at line 13 is non-linear, we leave it alone. Since the stride of the loop at line 15 is linear, we change it. The transformation results in this code:

```
for(int stride = ACC_N/2; stride > 0; stride >>= 1) {
  __syncthreads();
  for(int i' = 0; i' < (stride-threadIdx.x)/blockDim.x; i'++) {
    int i = i' * blockDim.x + threadIdx.x;
    acc[i] += acc[stride + i];
  }}
```

To determine whether this code is conflict-free (no race on `acc` on line 16), we need to check, for threads t_1 and t_2, two cases:

• Whether ($i^{t_1} = i^{t_2}$). Luckily, this is false because i is initialized to `threadIdx.x` (different for different threads) and stays different.

• Or, whether ($i^{t_1} = stride^{t_2} + i^{t_2}$). This is also false because $i^{t_1} < stride^{t_2}$ holds.

The logical formula for conflict checking incorporates all this knowledge and also that $stride \in (0, ACC\_N/2]$ and $i \in [0, (stride - threadIdx.x)/blockDim.x)$; it also emerges unsatisfiable:

$$\bigwedge_{t\in\{t_1,t_2\}} \left(\begin{array}{l} stride^t > 0 \ \wedge \ stride^t \leq ACC_N/2 \ \wedge \\ i'^t \geq 0 \ \wedge \ i' < (stride^t - t)/blockDim.x \\ \wedge \ i^t = i'^t * blockDim.x + t \end{array} \right)$$
$$\wedge \ (t_1 \neq t_2) \ \wedge \ (i^{t_1} = i^{t_2} \ \vee \ i^{t_1} = stride^{t_2} + i^{t_2})$$

Similar analysis can also be applied to the loop at lines 6-11.

6.2 Automatic Refinement

In addition to the loop index, we need to handle the variables in the loop body. Consider the following example, the constraints generated for j, l, n and k depend on whether they are loop carrying.

```
int m = 0; int k = a;
for(int i = lb; i < ub; i++)
{ int j = i * 2; int l = j + i;
  int n = m - l; k = j * k; ... }
```

A variable is *non loop-carrying* if (1) it is the loop index variable, or (2) it is not updated in the loop, or (3) any of its updates (if there is any) involves only non loop-carrying variables. We simplify our analysis by generating constraints only for non loop-carrying variables, and over-approximate loop-carrying variables to have range $(-\infty, +\infty)$. In this example, i, j, m, n are non loop-carrying while k is loop-carrying. The formula $i \in [lb, ub) \ \wedge \ j = i*2 \ \wedge \ l = j+i \ \wedge \ n = m - i$ accurately specifies the value ranges of i, j, l and n, and k (because it is loop-carrying) is over-approximated by leaving it unconstrained.

Sometimes over-approximating the range of a loop-carrying variable may lead to false alarms. If j below is unconstrained, then a false race will be reported on s[`threadIdx` $* \ n + j$].

```
int j = 1; int n = blockDim.x;
for(int i = n; i > 0; i >>= 1)
{ s[threadIdx * n + j] = ...; j = j * 2; }
```

To overcome this, PUG incorporates simple rules for syntactically deriving common invariants safely, and automatically adds them to the constraints. For the above example, PUG derives an invariant $i * j = n$, which follows from the relation between $*2$ and right shift ($\gg 1$). This implies $j < n$ and threadIdx$*n+j$ are different in different threads. PUG derives invariants for similar simple patterns involving $+$ and $-$, $*$ and /, and so on, but only for the variables used in the addresses of shared variables.

As another example, invariant $j = v + i * k$ can be derived for the following loop since j can be normalized to have the same stride as loop index i does.

```
int j = v;
for (int i = 0; i < ub; i++)
  { ...; j += k;}
```

6.3 Inter-Iteration Race Checking

Within a loop, accesses to shared variables may conflict with themselves in previous iterations, thus causing *inter-iteration* conflicts. For example, in the following loop,

```
for(int i = lb; i < ub; i++)
{ __syncthreads(); acc[i+1+tid] += acc[i]; }
```

access $acc[i + 1 + tid]$ may not conflict with $acc[i + 1 + tid]$ and $acc[i]$ in the same iteration. However, if the barrier is removed then $acc[i + 1 + tid]$ may conflict with $acc[(i - 1) + 1 + (tid + 1)]$, *i.e.* the access by a neighboring thread in the previous iteration.

PUG considers two cases:

- The loop body is not barriered. Different threads may be in different iterations, *i.e.* i's values in different threads may be regarded to be unrelated. If the barrier is removed in the above example, the constraint for conflict checks is as follows, which is clearly satisfiable for $t_1 \neq t_2$.

  $$i^{t_1} \in [lb, ub] \ \wedge \ i^{t_2} \in [lb, ub] \ \wedge$$
  $$(i^{t_1} + 1 + t_1 = i^{t_2} \ \vee \ i^{t_1} + 1 + t_1 = i^{t_2} + 1 + t_2)$$

- The loop body is barriered (*e.g.* ends with a barrier). If the body satisfies the synchronization correctness requirement described in § 4, then all threads will always be in the same iteration. In other words, loop index variable i should have the same value at all threads (*i.e.* $i^{t_1} = i^{t_2}$) (we say i is *single valued*); and the following constraint is unsatisfiable for $t_1 \neq t_2$.

  $$i \in [lb, ub] \ \wedge \ (i + 1 + t_1 = i \ \vee \ i + 1 + t_1 = i + 1 + t_2)$$

  Even after i is set to single valued, we may still need to consider two consecutive iterations. For the following code, PUG considers the possibility that accesses in s2 at iteration i conflict with those in s1 at iteration $i+1$.

  ```
  for(int i = lb; i < ub; i++)
    { s1; __syncthreads(); s2; }
  ```

In the scalar product example, the loop in lines 6-11 belongs to the first case, while the loop in lines 13-17 belongs to the second case. If the barrier at line 14 in the second loop is removed, then accesses on $acc[i]$ and $acc[stride + i]$ may conflict when $stride^{t_1} \neq stride^{t_2}$.

7. IMPLEMENTATION AND EXPERIMENTAL RESULTS

As described earlier, PUG is based on the Rose framework for C program analysis. The user may input a file containing multiple kernels together with the main (CPU side) program. The kernel to be analyzed is syntactically flagged, and this kernel alone will be analyzed. Within the kernel of interest, the user may place `assert` assertions anywhere in the code, which will be checked during analysis.

Overall Orchestration of PUG. Given an annotated program, PUG works in a push-button fashion and is totally syntax driven (similar to a precise type checker, more details in [16]). It first parses the program and triggers rules

for each syntactic category, building constraints in an intermediate format. For instance, for handling loops, it first checks if whether the loop body contains barriers. It then performs loop normalization and loop refinement, and analyzes the loop body which may contain multiple BIs. For a BI, PUG first checks whether there is a race (conflict), if so then report the bug and terminates. Otherwise it serializes all the accesses to shared variables and moves to the next BI.

Expressions in the intermediate language (IL) are converted to Yices' expressions for satisfiability checking. Yices' expressions are based on bit vectors (bounded integers). We found that the correctness of most CUDA kernels relied on the assumption that no overflows will occur in arithmetic operations. To model this, the user has the ability to request (through the "+O" flag) whether non-overflow constraints must be incorporated (for unsigned bit vectors). Setting the +O flag causes PUG to generate and incorporate these additional constraints for + and ∗:

IL Expr.	Yices Expr.	Constraint
$e_1 + e_2$	$e_1 + e_2$	$e_1 < 2^n - 1 \ \wedge \ e_2 < 2^n - 1$
$e_1 * e_2$	$e_1 * e_2$	$e_1 < 2^{n/2} \ \wedge \ e_2 < 2^{n/2}$
$e_1 \,/\, e_2$	q	$e_2 * q + r = e_1 \ \wedge \ r < e_2$
$e_1 \,\%\, e_2$	r	$e_2 * q + r = e_1 \ \wedge \ r < e_2$

In addition, since Yices does not provide the "div" and "mod" operator directly, we implement them using multiplication and addition. Some optimizations are performed when e_1 or e_2 are constants. For example, $2^m * e_2$ and $e_1 \,/\, 2^m$ are converted to $e_2 \ll m$ and $e_1 \gg m$ respectively ($\gg$ is a shift operator).

The user may further use two more types of annotations within the kernel of interest:

• An `assume` that defines the problem configuration parameters and input constraints (*e.g.*, whether a matrix is assumed to be square, what the input data constraints are). We capture this assume class as if it were a flag, "+C".

• In some examples, the user has to help PUG out by providing simple loop invariants or simple predicates on shared variables. These are assumed to be true (for now; future work will try to semi-automate their formal verification). These are shown as the "+R" flag. We do not include the syntactic invariants automatically generated by PUG into +R (these are guaranteed to be correct invariants).

We performed experiments using PUG on a machine with a single CPU (Intel Pentium-4 3.60 GHz processor with only 1 GB of memory). Our table of results in Table 1 shows which examples required these flags for verification to succeed, and not fail through false alarms. All the examples in this table are widely cited kernels from the CUDA SDK, and naturally PUG found them all to be correct. "Pass" in this table asserts that (i) *All barriers were found to be well synchronized*, and (ii) *No races were found*. When a benchmark program (*e.g.* `Reduction`) contains multiple kernels, we invoke them one by one – but in a single run – and report the total time of this run.

PUG has checked many more CUDA SDK kernels than shown in Figure 1. While the computation of a large application is usually broken into multiple kernels, we have successfully checked some very large kernels (*e.g.*, Eigenvalues, at 2,200 LOC). The translation time into IL and to the Yices constraints is negligible, and not counted in.

Kernels	loc	+O	+C	+R	B.C.	Time(pass)
Bitonic Sort	65				LO	2.2
MatrixMult	102	*	*		HI	<1
Histogram64	136				LO	2.9
Sobel	130	*			HI	5.6
Reduction	315	*			HI	3.4
Scan	255	*	*	*	LO	3.5
Scan Large	237	*	*		LO	5.7
Nbody	206	*			HI	7.4
Bisect Large	1,400	*	*		HI	44
Radix Sort	1,150	*	*	*	LO	39
Eigenvalues	2,200	*	*	*	HI	68

Table 1: Experimental results of checking some SDK kernel programs for synchronization errors, races and bank conflicts.

PUG is able to check most programs smoothly. The radix sort kernel is the most difficult one to analyze since the addresses of a few shared variable accesses cannot be resolved locally, *i.e.* they are control-dependent on the shared arrays which may be updated by multiple threads. This makes the checking difficult. In our present attack, we added +C constraints indicating the the shared arrays are (partially) sorted to overcome this limitation.

Bank Conflict Checking. A fascinating direction to evolve PUG is in giving designers feedback on performance metrics. Thanks to our use of SMT, we can use the infrastructure for race checking in order to check for bank conflicts also. Specifically, access $k[i]$ and $k[j]$ incurs a race when $i = j$, and incurs a bank conflict when $i\%16 = j\%16$. Column "B.C." indicates how serious the bank conflict is, which is measured by the *percentage of the barrier intervals (BI) containing bank conflicts*: HI (High) and LO (Low) denote $\geq 50\%$ and $< 50\%$ respectively. Since only two threads are considered and the loops are not unrolled, these results are quite preliminary; yet, the promise is clear. We plan to give more accurate measurement in the future work.

Road-Testing PUG. We took 57 assignment submissions from a recently completed graduate GPU class taught in our department. The "Defects" column in the table below

`Defects`	`Race`		`Refinement`	
	benign	fatal	over #kernel	over #loop
13 (23%)	3	2	17.5%	10.5%

indicates how many kernels were found to be not well parameterized – *i.e.*, work only in certain configurations (e.g. the grids and blocks must have specific sizes). We had to manually find this out by guessing and trying different +C settings. This is a promising way to reverse-engineer unstated assumptions and provide feedback to a programmer to improve their kernel.

There were *three benign races* and *two fatal races* in these (presumably tested) codes. These fatal races can be attributed to missing barriers in the loop body or incorrect indexing at the boundary between two thread data spaces.

While PUG always does its set of automatic loop refinements, we were curious as to how many of these cases could have passed through without them. When we turned off the automatic loop invariants, we found that only 17.5% of the kernels (measured in terms of loops, only 10.5% of the total number of loops) would have failed (by giving false alarms).

Thus it appears that for small to medium kernels represented by a class, about 90% of the kernels can be verified even without loop refinements.

Assertion Checking (Functional Correctness). Users can specify the properties to be checked using our `assume` and `guarantee` directives. If a precondition $assume(P)$ and a postcondition $guarantee(Q)$ are specified, formula $P \wedge \neg Q$ is added into the constraint. For example, we can specify the correctness of the bitonic sort kernel

```
__global__ bitonic (int vals[]) {
  ...
  guarantee(i < j ⟹ vals[i] ≤ vals[j]);
}
```

Functional correctness check requires accurate models of the programs. PUG translates the program into a bounded one by unrolling the loops dynamically in the incremental modeling phase. The number of threads must be specified explicitly. Since CUDA programs are highly symmetric, we only need to consider a few threads.

The following table shows the SMT solving time in seconds. To speed up the checking we turn off the overflow detection, assign small values to the loop bounds, and use smaller bitvectors. Here n denotes the number of threads; T.O denotes Time Out (> 5 minutes). Correctness is proven for bug-free programs, and bugged programs are obtained by disabling some required constraints or specifying false assertions. Correctness check takes much longer time since the solver needs to prove unsatisfiability (*i.e.* absence of bugs) for all cases. In general, the degree of loop unrolling needed is proportional to the number of threads n, making the solving time blow up on n.

Kernels	n = 2		n = 4		n = 8	
	Corr.	Bug	Corr.	Bug	Corr.	Bug
simple reduct.	< 1	< 1	2.8	< 1	T.O	4.42
matrix transp.	< 1	< 1	1.9	< 1	28	6.5
bitonic sort	< 1	< 1	3.7	< 1	T.O	255
scalar product	< 1	< 1	6.9	2	T.O	137

This checker identifies several "bugs" in these programs: (i) the "bitonic sort" is incorrect when the number of threads is not the power of 2; (ii) the "scalar product" is incorrect when ACC_N is not the power of 2; and (iii) the "matrix tranpose" is incorrect when the sizes of two input matrixes are smaller than the block size.

As the property checker does not scale well with respect to the number of threads, it is intended to be used as a *unit* tester/verifier for functional correctness.

Performance Improvement. PUG utilizes Yices's *incremental SMT solving* technique to avoid evaluating an expression multiple times. This technique is primarily used to manage the built transitions. For example, when the solver is called for evaluating e over transitions $\mathbb{E}$ provided that path condition $\mathbb{C}$ holds, we first assert $\mathbb{E}$ and push the context containing $\mathbb{E}$ into Yices' context stack, then assert $\mathbb{C}$ and e to evaluate the entire expression. After that, when we want to evaluate e_1 on $\mathbb{E}$ and $\mathbb{C}_1$, we pop the context stack so as to restore the context containing the existing clauses for $\mathbb{E}$, then we assert $\mathbb{C}_1$ and e_1. This enables us to avoid evaluating $\mathbb{E}$ again.

We also apply a simple *slicing* algorithm to exclude useless transitions from the transition stack. A use-def analysis is performed to identify the variables which will be used by the addresses of shared variables. We do not build transitions for the assignments involving other variables. For instance, in the scalar product example of Figure 1, no transitions corresponding to the assignments on line 9 and line 16 will be added into the transition stack.

Some Limitations of PUG. Present day SMT solvers provide limited support for real numbers. PUG cannot prove the functional correctness of many CUDA applications that operate on float or double numbers. Fortunately, this doesn't limit PUG's conflict checking power because the addresses of shared variables involves only unsigned integers.

PUG may report false alarms if it fails to derive loop invariants for complicated program patterns. In this case, the user is required to provide sufficient invariants.

PUG cannot handle kernels containing complicated pointer arithmetic operations. In addition, PUG requires manual transformation of the source programs to Kernel C format (*e.g.* by converting "while" loops to "for" loops and eliminating advanced C++ features).

Other Programs. Although focusing on CUDA kernels, PUG can be easily extended to other domains such as lock based multi-threaded programs. It is particularly suitable for checking such programs over relaxed memory models: we just need to loosen the constraint on the accesses orders w.r.t the memory model. The main challenge, however, is to model involved APIs and system calls. One solution is to build light-weight models or abstract interpretations for these APIs as we did for MPI 2.0 [18].

8. CONCLUDING REMARKS

Other Related Work. Traditional testing methods are ineffective at locating CUDA bugs because they assume concrete input values as well as a fixed numbers of threads. They cannot generate all possible schedules – an exponentially growing number even for short programs. They have no mechanisms to focus on *relevant* schedules that trigger bugs. Interleaving reduction methods such as [12] are inapplicable to CUDA. Many past efforts have focused on multi-threaded programs synchronizing using locks and semaphores [11]. These methods are inapplicable for kernels.

Symbolic techniques for program analysis go back to works such as [5] and more recently [6]. Recent exact symbolic concurrent C program analysis techniques (*e.g.*, [15]) have not been shown to be effective for vector computations found in CUDA kernels. In PUG, we do not worry about modeling recursive functions or heap allocated structures – something considered in tools such as [15]. Our work is tailored for CUDA which is very widely used; it will easily apply to emerging standards (*e.g.*, OpenCL [20]).

Only two CUDA-specific checkers have been reported on the past. An instrumentation based technique is reported [3] to find races and shared memory bank conflicts. This is an ad-hoc testing approach, where the program is instrumented with checking code, and only those interleavings occurring in a platform-specific manner are considered. A determinism (*i.e.* no races) checking tool [23] constructs constraints from an automaton without considering the communication (*e.g.* value passing) among threads. This tool makes many assumptions on the input programs to facilitate noninterference checking, which include: (i) the source program cannot contain loops, conditional barriers, functions calls and

pointers; (ii) all variables must already in SSA format, etc. In contrast, PUG works on source programs directly and does not make these restrictions. PUG is also able to model communicating programs. Our static race detection is similar to the non-interference checking in [23]; our method is capable of handling more general cases.

SPMD programs are prone to incorrect synchronization patterns, especially when barriers are within conditional statements. Aiken and Gay [1] proposed a type system to check global synchronization errors. They check whether program branches make the same decision and execute the same number of barriers by recording the single-valued variables that have the same values among all threads. They may produce false alarms by rejecting correct programs. PUG is able to check such global synchronization errors as well. As PUG relies on SMT solving to compare the values of expressions, it produces more precise results on determining whether different threads make the same decision, thus giving more accurate reports on synchronization errors.

We have shown many innovative uses of PUG including providing performance estimates relating to bank conflicts. Related work on symbolic techniques for performance evaluation [13] is of interest here.

Summary of PUG, and Future Work. We presented the first realistic analyzer for GPU kernels called PUG. PUG takes an annotated CUDA C program and analyzes a kernel flagged to be of interest in it (this single kernel may itself be thousands of lines long). The user specifies the number of threads (usually two) for which the kernel is to be analyzed. There is really no way to tell whether two threads are sufficient for either race checking or for assertion checking. This can easily be shown to be an undecidable problem as a special instance of the undecidability of parameterized verification problem [4]. We are interested in exploring whether finite cut-off results can be obtained (*e.g.*, [8]). Abstraction/refinement methods may be another approach.

PUG can be supported by any of the highly developed SMT solvers available today. Even given the phenomenal advances in the SMT technology, a straightforward (naïve) approach of unrolling all loops and solving will not work. PUG employs a number of innovative approaches for reducing the analysis complexity.

We have already documented a number of limitations of PUG. One additional limitation that needs to be overcome pertains to the calling context of CUDA kernels. The calling context may most likely determine the `assume` clauses that a user has to provide through the "+C" flag. Any method for obtaining some of these constraints automatically can help improve the degree of automation. Loop invariant discovery methods that determine the +R annotations will also enhance the usability of PUG.

9. REFERENCES

[1] Aiken, A., and Gay, D. Barrier inference. In *Symposium on the Principles of Programming Languages (POPL)* (1998).

[2] Allen, R., and Kennedy, K. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[3] Boyer, M., Skadron, K., and Weimer, W. Automated dynamic analysis of CUDA programs. In *Third Workshop on Software Tools for MultiCore Systems* (2008).

[4] Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking*. MIT Press, 2000.

[5] Cobleigh, J. M., Clarke, L. A., and Osterweil, L. J. Flavers: A finite state verification technique for software systems. *IBM Systems Journal 41*, 1 (2002).

[6] Csallner, C., Tillmann, N., and Smaragdakis, Y. DySy: Dynamic symbolic execution for invariant inference. In *International Conference on Software Engineering (ICSE)* (2008), pp. 281–290.

[7] Cuda programming guide version 1.1. `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`.

[8] Emerson, E. A., and Kahlon, V. Reducing model checking of the many to the few. In *International Conference on Automated Deduction (CADE)* (2000), pp. 236–254.

[9] Feng, M., and Leiserson, C. E. Efficient detection of determinacy races in cilk programs. In *Parallel Algorithms and Architectures (SPAA)* (1997).

[10] Fermi. `http://www.nvidia.com/object/fermiarchitecture.html`.

[11] Flanagan, C., and Freund, S. N. Type-based race detection for Java. In *Programming Language Design and Implementation (PLDI)* (2000).

[12] Flanagan, C., and Godefroid, P. Dynamic partial-order reduction for model checking software. In *Symposium on the Principles of Programming Languages (POPL)* (2005), pp. 110–121.

[13] Gulwani, S. Speed: Symbolic complexity bound analysis. In *Computer Aided Verification (CAV)* (2009), pp. 51–62.

[14] Kirk, D. B., and mei W. Hwu, W. *Programming Massively Parallel Processors*. Morgan Kauffman, 2010.

[15] Lahiri, S. K., Qadeer, S., and Rakamaric, Z. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Computer Aided Verification (CAV)* (2009), pp. 509–524.

[16] Li, G., and Gopalakrishnan, G. Technical Report and PUG Tool Download: `http://www.cs.utah.edu/fv/PUG`.

[17] Li, G., Gopalakrishnan, G., Kirby, R. M., and Quinlan, D. A symbolic verifier for CUDA programs. In *PPoPP, Poster Session* (2010), pp. 357–358.

[18] Li, G., Palmer, R., DeLisi, M., Gopalakrishnan, G., and Kirby, R. M. Formal specification of MPI 2.0: Case study in specifying a practical concurrent programming API. *Sci. Comp. Prog. 75* (2010).

[19] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of Program Analysis*. Springer-Verlag, 1999.

[20] OpenCL. `http://www.khronos.org/opencl`.

[21] The ROSE compiler. `http://www.rosecompiler.org/`.

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

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

[24] Yices: An SMT solver. `http://yices.csl.sri.com`.