Parametric Flows: Automated Behavior Equivalencing for Symbolic Analysis of Races in CUDA Programs

Peng Li,* Guodong Li,† and Ganesh Gopalakrishnan*
*School of Computing
University of Utah
{peterlee,ganesh}@cs.utah.edu
†Fujitsu Labs of America, CA
gli@us.fujitsu.com

***Abstract*—The growing scale of concurrency requires automated abstraction techniques to cut down the effort in concurrent system analysis. In this paper, we show that the high degree of behavioral symmetry present in GPU programs allows CUDA race detection to be dramatically simplified through abstraction. Our abstraction techniques is one of *automatically* creating *parametric flows*—control-flow equivalence classes of threads that diverge in the same manner—and checking for data races only across a pair of threads per parametric flow. We have implemented this approach as an extension of our recently proposed GKLEE symbolic analysis framework and show that all our previous results are dramatically improved in that (i) the parametric flow-based analysis takes far less time, and (ii) because of the much higher scalability of the analysis, we can detect even more data race situations that were previously missed by GKLEE because it was forced to downscale examples to limit analysis complexity. Moreover, the parametric flow-based analysis is applicable to other programs with SPMD models.**

***Index Terms*—GPU programming, Data Races, Formal Analysis, Parameterized Reasoning**

I. Introduction

GPUs represent an exciting platform for parallelization, as they are readily available to almost everyone, and offer impressive speedups on many problems. Unfortunately, the detection and elimination of bugs in GPU programs is a serious productivity impediment, with undetected bugs being worse. We focus on one important class of bugs in this paper—namely, data races (we also detect deadlocks—elaborated later). A data race in a concurrent program occurs when two concurrent accesses (one of which is a write access) occurs on a data variable, resulting in an unpredictable final value. Data races can not only affect a program's final answer—it can also allow a compiler to perform completely illegal transformations, because many compilers are known to aggressively transform programs *assuming* data-race freedom.

Conventional GPU debuggers [1], [2], [3] are ineffective at finding and root-causing races. One has to be lucky to have picked the right set of inputs to have triggered data races; then run the right set of thread and warp schedules to have caused a racing access; then be lucky to actually discern a data corruption amidst the final result. Many formal and semi-formal analysis tools have recently been proposed for GPU program analysis; they employ a combination of static/dynamic [4] or symbolic [5], [6], [7] analysis.

This paper is an extension of our *symbolic* approach to GPU program analysis published recently in [8] and supported by our recently released tool GKLEE. GKLEE employs a formal analysis approach that is very easy to use for practitioners, yet effective at finding deep-seated bugs. A GKLEE user writes standard C++ CUDA programs, indicating some of the program variables to be *symbolic* (the rest are assumed to be concrete variables). These programs are compiled into LLVM byte-code, with GKLEE serving as a symbolic virtual machine. When GKLEE runs such a byte-code program, it generates and records constraints relating the values of symbolic variable. Conditional expressions in the C++ code (*e.g.*, switch statements) generate constraints covering both outcomes of a branch; these are solved by instantiating the symbolic variables to cover all feasible branching options (or as per user-control of how much to cover). The result is that users automatically obtain path-coverage to the desired degree. GKLEE also writes out these cases into test files that then form test suites to be run on any platform, ensuring high coverage. Because of the recent growth in the power of SMT-solvers used to solve these constraints [9], a tool such as GKLEE is able to handle non-trivial SDK kernel functions.

This paper addresses a major drawback of all the semi-formal tools described so far—including GKLEE: *these tools model and solve the data-race detection problem over the explicitly specified number of GPU threads*. This makes these tools difficult to apply in many situations in supercomputing where many program modules (*e.g.*, library modules) often assume a certain minimum number of threads to be involved, where these minimum numbers themselves are very large. While it may be possible to manually downscale the number of threads, unfortunately many program modules do not document how such downscalings of size parameters can be done consistently (if at all that is feasible for a particular implementation). Thus, automatically handling large numbers of threads is a necessity.

In this paper, we provide an extension of GKLEE that *exploits thread symmetry and provides a way to analyze GPU programs containing large (bounded) numbers of threads in*

* Supported in part by NSF awards OCI 1148127 and CCF 1241849.
SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

real kernels. In a nutshell, our method partitions the space of executions of a GPU program into *parametric flow equivalence classes* (PFE), and models the race analysis problem over two parametric threads in one PFE equivalence class. This analysis method over parametric flows has been implemented in a new version of GKLEE called GKLEE$_p$:

- GKLEE$_p$ has found all the data-races found by GKLEE, plus many new ones that GKLEE missed (because we had to deliberately keep thread-population sizes low under GKLEE).
- GKLEE$_p$ represents a major revision of GKLEE to efficiently represent PFE classes; yet, its basic operation of finding these equivalence classes uses the same symbolic analysis methods as used in GKLEE, hence inherits all powerful symbolic facilities from GKLEE.
- In the best case (*e.g.*, in kernels without loops), GKLEE$_p$ produces the most impressive results, by modeling race-checking over conflicting (read/write) configuration over just 2 threads (as opposed to N threads under GKLEE).
- In cases with loops whose iteration counts depend on the number of threads and thread-blocks, GKLEE$_p$ still reduces one dimension of complexity. More specifically, in a situation where GKLEE encodes races over N threads of (loop-unravelled) length N, GKLEE$_p$ encodes races over 2 threads of (loop-unravelled) length N. Since GKLEE$_p$ does not over-approximate the loops, it has a low false alarm rate (none observed so far), making it particularly useful for realistic programs which may contain loops that cannot be precisely abstracted.
- We describe the conditions under which GKLEE$_p$ is an exact race-checking approach, and also present when it can miss bugs or give false alarms. In all our experiments so far, these unusual patterns have not arisen, suggesting that GKLEE$_p$ is practical.

II. Background

A. Overview of Symbolic Execution

GKLEE$_p$ is a significant rewrite of GKLEE, but retains the basic framework of its symbolic execution.

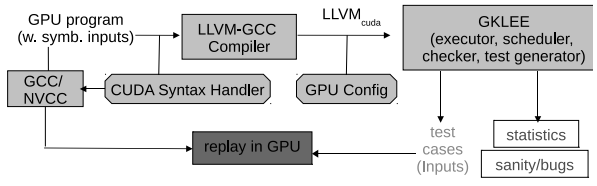

Fig. 1. GKLEE's architecture.

GKLEE's front-end compiles CUDA C++ programs into LLVM$_{\text{cuda}}$. Users may select some of the variables in their C++ program to be *symbolic*. This is the only change necessary to prepare a program for analysis under GKLEE. The GKLEE executor is a symbolic virtual machine that understands the CUDA memory- and execution models [10]. When the executor encounters an assignment statement containing symbolic variables, it generates constraints relating the pre/post states. When it encounters a conditional involving a symbolic variable, it forks executions: (i) for one path, the underlying constraint solver tries to make the conditional true by suitably instantiating the symbolic variables, (ii) for the other path, the instantiatons force a false outcome. At the end of a GKLEE run, the user minimally gains a rich collection of test files that ensure path coverage (these tests can be run on the GPU hardware). Additionally, the user is able to obtain three classes of analysis results: (i) data races, (ii) degree of warp divergences, bank conflicts, and non-coalesced accesses, (iii) deadlocks. CUDA-specific test pruning heuristics in GKLEE help retain good coverage while dramatically reducing the number of symbolic executions [8]. One can try GKLEE out on our web-hosted remote execution portal, as well as download it.[1]

B. Race Checking

Race checking in concurrent programs has been studied extensively. GPU program race checking differs in many essential way from that studied in the non-GPU contexts: (i) GPU programs are largely computation-oriented, synchronizing sparingly through barriers and atomic operations, (ii) the number of threads involved in GPU programs is vastly more than entertained in non-GPU areas. Formal and semi-formal methods in non-GPU contexts employ various lock-set and happens-before based methods [11], [12]. In terms of finding races with high assurance, one of the main impediments has been *schedule (or interleaving) explosion*. For example, five threads carrying out five sequential instructions each generate $25!/(5!)^5 \approx 13$ trillion interleavings. While methods such as partial order reduction [13], [14] dramatically reduce the number of interleavings to be examined, in the case of CUDA programs we can do much better.

In [8], we show that symbolically executing *one* schedule (called the *canonical schedule*) through all the threads and recording potential conflicting pairs during this schedule gives us the ability to detect *a race* if there is *any race*. To understand this method, consider two threads t_0 and t_1 that have encountered a GPU barrier B_0 (such as `__syncthreads()` as in CUDA) and are proceeding towards the next barrier B_1. Let t_0 perform N shared-memory accesses $s_{0,0}; s_{0,1}; \ldots; s_{0,N-1}$ in this *barrier interval*, and likewise t_1 perform N shared-memory accesses $s_{1,0}; s_{1,1}; \ldots; s_{1,N-1}$. In GKLEE, we run these threads sequentially; meaning $s_{00}; s_{01}; \ldots; s_{0,N-1}; s_{10}; s_{11}; \ldots; s_{1,N-1}$.

thread t_0	thread t_1
$p_{0,0}$? $s_{0,0}$	$p_{1,0}$? $s_{1,0}$
$\ldots$	$\ldots$
$p_{0,N-1}$? $s_{0,N-1}$	$p_{1,N-1}$? $s_{1,N-1}$

During this canonical run, for every potentially conflicting pair $s_{0,i}$ and $s_{1,j}$, we record (i) the path predicates under which

[1] http://www.cs.utah.edu/fv/GKLEE

these accesses are performed (say $p_{0,i}$ for $s_{0,i}$ and $p_{1,j}$ for $s_{1,j}$). We then try to determine if $p_{0,i} \wedge p_{1,j}$ is satisfiable, and if so whether under this constraint $s_{0,i}$ and $s_{1,j}$ are accessing the same location with one of them being a write. In [15], we prove that if there is *any* data race between these two threads, one of the $\langle s_{0,i}, s_{1,j} \rangle$ pairs will be found to be racing. Intuitively, the idea is that if there is a race R under an *arbitrary* thread schedule, then the canonical schedule will either run race-free till R, or encounter another race $R^{'}$ before encountering R. The intuition is that *any* shared-memory communication between two threads in a barrier interval is a race; and if there were no prior communication before race R, then *all schedules* leading to R are equivalent; else, $R^{'}$, the prior communication is the "earlier race."

The saving due to canonical scheduling is essential for the success of GKLEE. For example, with N being 5 and there being 5 threads, instead of examining 13-trillion schedules to check for races, under the canonical scheduling, *one* schedule finds *a race* such as $R^{'}$ (or finds R itself) if there is *any race*.

In general, GKLEE will take k threads each with N steps and run *one* schedule of length $k \times N$, and encode all possible pairs of accesses over the $k \times N = O(k^2N^2)$ total accesses. Under parametric flow equivalencing, GKLEE$_p$ will, whenever possible, safely reduce the problem to 2 threads each with N steps and run *one* canonical schedule of length $2 \times N$, and encode all pairs over $2 \times N = O(4N^2)$. If N is independent of the number of threads (as is the case in GPU programs where loop iteration counts are independent of the number of threads or thread-blocks), then the savings are even more dramatic. In fact, for debugging purposes, a loop abstraction that does not go through all loop iterations is often the most efficient compromise.

III. A Motivating Example

Let bid and tid stand for block ID and thread ID respectively. A GPU program consists of tid-independent conditionals (TIC) and tid-dependent conditionals (TDC), notice that the bid-depdendent conditionals are also categorized into the TDC domain. For simplicity purpose, we do not discuss the cases where the conditions depend on symbolic inputs in this section. An assignment statement such as $x = 1$ is a TIC (with condition "true"). A conditional expression that does not involve the tid is also a TIC. Since TICs only have one successor state, we can group TICs into maximal *basic blocks*. TDCs are conditionals that involve the tid. For instance, if $(tid\%2)$ is used as a conditional expression, the even threads will branch one way and the odd threads another way. We put the conditional expression of a TDC into a basic block of its own. Since basic blocks are basic units of execution, we will model them as our GPU program "instructions." Thus, after a TDC instruction, some threads will be executing the "then" sequence of instructions while the other threads will be executing the "other" sequence.

A motivating example 2 manifests the advantages of GKLEE$_p$. In this kernel, 8K threads are involved, with four blocks and 2K threads per block. Two arrays *a* and *b* are

```
// a[4 * 2048] in device memory;
// b[2048] in shared memory;
__global__ void test(unsigned * a) {
1:   unsigned bid = blockIdx.x;
2:   unsigned tid = threadIdx.x;
3:
4:   if (bid % 2 != 0) {
5:     if (tid < 1024) {
6:       unsigned idx = bid * blockDim.x + tid;
7:       b[tid] = a[idx] + 1;    // Write of Race-1
8:       if (tid % 2 != 0) {
9:         b[tid] = 2;           // Write of Race-2
10:      } else {
11:        if (tid > 0)
12:          b[tid] = b[tid-1]+1; // Read of Race-2
13:        }
14:    } else {
15:      b[tid] = b[tid-1];       // Read of Race-1
16:    }
17:  } else {
18:    unsigned idx = bid * blockDim.x + tid;
19:    b[tid] = a[idx] + 1;
20:  }
}
```

Fig. 2. The motivating example

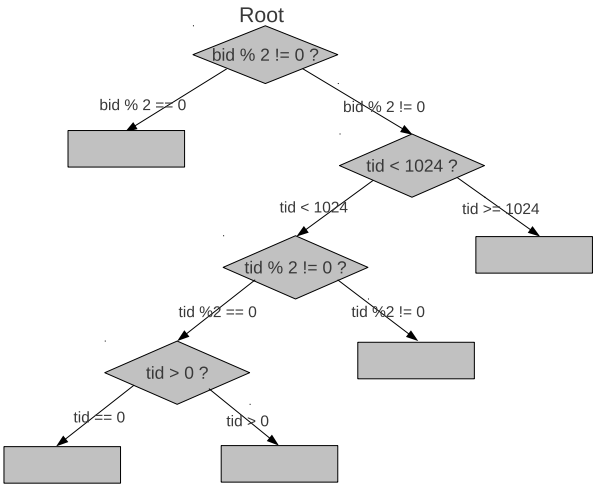

Fig. 3. Parametric flows for the motivating example

created and located in the device memory and shared memory respectively.

Parametric flows are the control-flow equivalence classes of threads that diverge in the same manner. In more detail, GKLEE$_p$'s race checking approach is one of (i) checking for data races across a pair of threads *within* a single parametric flow, and (ii) race checking between one thread (each) of *two different* flows. The former is to cover intra-warp races while the latter is to cover inter-warp races.

In our example, GKLEE$_p$ yields four parametric flows. Each lozenge denotes a TDC, and each rectangle in the diagram represent the TICs. GKLEE$_p$ starts its execution within one thread. When a TDC is encountered, it spawns a new flow. For example, when $bid\%2 \neq 0$ is encountered, two flows are generated with the appropriate conditional path conditions (namely $bid\%2 \neq 0$ and $bid\%2 = 0$).

a) A Data Race: Whenever two memory accesses involving a common location are performed concurrently by two threads, with one of the accesses being a write, a data race situation is created. Data races are almost impossible to

discern manually. They may *never* produce corrupt data results upon testing because of the restricted nature of scheduling employed by typical GPU hardware. Most damaging of all, races may license compiler transformations that are "unwarranted," resulting in an error that appears completely unrelated to the root-cause, and is potentially *very* confusing.[2] This example has two race conditions:

- The Write access in Line 7 (done by Thread 1023) and the Read access done by Thread 1024 in Line 15. This is an inter-warp race – well-understood by anyone who has studied GPUs and the CUDA execution semantics.
- Any odd-numbered thread (*e.g.*, thread 1) and an even-numbered thread that is numbered one higher (*e.g.*, thread 2), both of which being in the range $0, \ldots, 1023$, involving the Write access on Line 9 and the Read access on line 12. These lines are *mutually exclusive*; then why is it a race? Reason: on one GPU, line 9 may be executed before Line 12, and vice versa on another. Thus on GPU1, the write occurs before the read, while on GPU2, it is the other way. This "race" is noticed when programmers port the code from GPU1 to GPU2. This race type was first identified in [8] where it is called a *porting race*.

GKLEE requires around 30 seconds to explore all pairs of potential conflicts and reveal these two errors. In constrast, GKLEE$_p$ only needs 1.3 seconds.

Furthermore, for this example, GKLEE$_p$ reports a race if and only if GKLEE does so too, making GKLEE$_p$ a sound and lossless reduction of GKLEE.

IV. Foundation

A. Parameterized Race and Deadlock Checking

To better understand GKLEE$_p$, let us study the following example where $f1$ and $f2$ are functions over the block id bid and thread id tid.

```
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[f1(idx)] + b;
  __syncthreads(); // A barrier
  if (idx < N) a[idx] = temp[f2(idx)];
}
```

Suppose the barrier is removed from this example; we can observe that the accesses $a[idx]$ and $a[f1(idx)]$ by different threads may race depending on function $f1$. This can be detected by examining the symbolic models of two threads as follows (private variables in a thread are superscripted by the thread id, and for simplicity we assume that threads t_1 and t_2 are in the same block but in different warps). More formally, a race occurs if predicate $t_1.x \neq t_2.x \ \wedge \ id^{t_1} < N \wedge id^{t_2} < N \wedge f1(idx^{t_1}) = idx^{t_2} \wedge |t_2.x - t_1.x| \geq 32$ holds. A constraint solver (an SMT tool for us [9]) can determine whether this predicate is satisfiable, and if so, it would return a concrete satisfying instance. Accesses to $temp$ can be analyzed similarly (knowing $f2$).

[2]We are grateful to Vinod Grover of Nvidia for this insight.

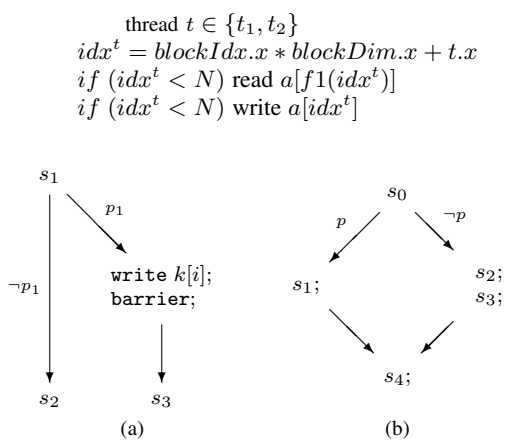

Fig. 4. Example CFGs.

To further illustrate these ideas, consider the control-flow graph (CFG) given in Figure 4(a). This diagram shows how statements s_1 through s_3 might be situated in some example program. At first glance, this appears ill-synchronized: one thread may take the s_1 to s_2 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 a deadlock (due to textually non-aligned barriers [16]) if so. Our approach checks whether all threads make the same decision on the condition.

In Figure 4(b), both the left and the right branch contain no barrier; thus they are considered well synchronized. We now check for conflicting accesses some of which involve conditionals. The conflict check includes the following expressions (here $\sim$ denotes the *conflict* relation, and $\not\sim$ 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 one *barrier interval (BI)* containing five accesses (1) s_0, (2) $p\,?\,s_1$, (3) $\neg p\,?\,s_2$, (4) s_4 and (5) $\neg p\,?\,s_3$. Conflict freedom requires the pairwise comparison of these five accesses for two parameterized threads; the $5 \times 5 = 25$ comparisons include the following. In GKLEE$_p$, this procedure is done by constructing a flow for each path condition and checking the conflicts over two representative threads (with symbolic ids).

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

Note that GKLEE$_p$ makes such case analysis scale for very large numbers of threads by choosing representative threads from each flow equivalence class.

B. Soundness of the Reduction

Terminology: Here we refine the definitions of TIC and TDC mentioned before: a variable is *X-dependent* if it is data-flow- or control-flow- dependent on X, where X is a thread ID (tid), block ID (bid), or symbolic input. If a variable v (or condition c) is dependent on only the tid, then we can denote

it as $v(tid)$ (or $c(tid)$). Access $c(tid)$? $v(tid)$ represents all the accesses on v for all valid tids under condition c. In GKLEE$_p$, we use representative values of tid rather than all concrete tids. The key of parameterized checking is to use one representative thread to represent all threads with similar behaviors. Or, from another perspective, the behaviors of all these threads can be reduced to that of one representative thread t_r. The reduction is sound if and only if thread t's each access appears in the execution path is *renaming equivalent* to that of t_r, in the sense that any access performed by t must be obtainable from those performed by t_r by taking each access expression—address read/written—and substituting t for t_r. Hence, if the variable or condition depends only on the tid, then using a parameterized tid is sufficient; if the variable or condition depends only the tid and bid, then it suffices to use a parameterized tid and bid pair. When race checking is performed, the parameterized expression is instantiated using two distinct symbolic values for tid, bid, etc. All the accesses within the same BI are compared pairwise to see whether their addresses overlap.

The case when a variable or condition is (symbolic) input-dependent is similar: we need to parametrize the inputs with respect to the threads. Typically, CUDA threads access an input, say an array A, in a space-dividing manner such that thread tid in block bid typically accesses $A(tid, bid)$. It is also possible that array A is dependent on symbolic inputs *and* on tid, bid; in this case, GKLEE$_p$ generates symbolic instances for the symbolic inputs in the usual way as GKLEE does (based on conditional-imposed constraints), and for each instance it employs representative values for tid, bid.

When a variable depends on the accumulation of values across more than two threads (we call such variables *accumulative* variables), parametric reduction may be unsafe since it considers only two threads rather than N threads. The Appendix discusses an approach which will be fully implemented in future. Handling accumulative variables is very similar to race checking – except that we consider multiple BIs rather than one BI. When a kernel restricts the usage of accumulative variables, our race checker satisfies some desirable properties.

Claim 1. *GKLEE$_p$ reports a race if and only if GKLEE reports this race for kernels where the addresses of all accesses are not dependent on accumulative variables. That is, GKLEE$_p$ is a sound abstraction of GKLEE for these kernels.*

Currently GKLEE$_p$ over-approximates the value of an accumulative variable, this may introduce some false alarms, *e.g.* when the accesses are control-dependent on this variable. However GKLEE$_p$ will not miss the races; it will also use the techniques described in the Appendix to rule out false alarms.

Claim 2. *GKLEE$_p$ will report every race reported by GKLEE for all kinds of kernels.*

C. Parameterized Checking with SIMD

A feature of CUDA is SIMD execution: threads are grouped in warps; the threads within a warp are executed in a lock-step manner; while the threads in different warps (but in the same block) are synchronized through explicit barriers. Two intra-warp threads can race only if they simultaneously write to the same shared variable at the same instruction. Inter-warp threads may race at different instructions under different path conditions since warp scheduling is non-deterministic in CUDA. Our parameterized method must account for the SIMD characteristic when checking races.

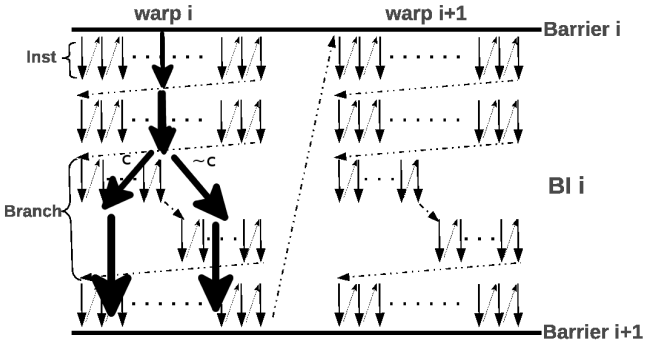

Fig. 5. Canonical schedule with SIMD.

GKLEE performs scheduling with respect to SIMD. The threads within a warp are executed in a lock-step manner. The warps (or blocks) themselves follow the usual canonical method, synchronizing at the CUDA barriers. Figure 5 shows how multiple warps are executed. In particular, in cases where the threads in a warp diverge (*i.e.* make different decisions over the same branch), the lock-step requirement is met by the hardware by executing the two sides sequentially and merging them at the first convergence point (*e.g.* the nearest common post-dominator). This is adopted by GKLEE to take care of all nuances of the CUDA semantics. GKLEE$_p$ inherits GKLEE's SIMD-aware scheduling scheme and makes it parameterized (the parametric flow is marked in Figure 5 as bold arrows). For an unconditional instruction, its execution by N threads is modeled by using one parameterized thread. This thread represents other threads in the same warp, in the different warp, in the different group, and so on. Suppose this instruction accesses shared location $a(tid)$, then threads may cause a race when $t_1 \neq t_2 \wedge a(t_1) \sim a(t_2)$ regardless whether t_1 and t_2 are within the same warp or not.

When a conditional instruction is encountered, the threads within a warp may diverge into two parts whose execution order is not fixed. GKLEE$_p$ forks the flow and produces two new nodes representing the two branches of the condition. These nodes will not be merged, and subsequent executions will start from each one. No matter what the execution order of these two parts is, the race between the two parts can be detected by examining whether the involved accesses conflict.

$$\exists t_1, t_2 \text{ in the same warp} : (c(t_1) \ ? \ a(t_1)) \sim (\neg c(t_2) \ ? \ a(t_2))$$

Clearly, this constraint is also applied when t_1 and t_2 are in the different warps. Hence $(c(t_1) \ ? \ a(t_1)) \sim (\neg c(t_2) \ ? \ a(t_2))$ is a generic constraint for detecting races relevant to condition c; and we need not to distinguish the intra-warp and inter-warp cases. This illustrates the following general principles

of using parametric flow tree to check races when respecting the SIMD model. In sum, our parametric flow based analysis respects SIMD by considering both intra-warp and inter-warp. The case of intra- and inter- blocks is analogous.

- A node $c_1 \ ? \ a_1$ may conflict with node $c_2 \ ? \ a_2$ if $c_1 \neq c_2$, *e.g.* even for intra-warp threads.
- A node $c \ ? \ a_1$ (note that c may be empty) may conflict with node $c \ ? \ a_2$. If a_1 and a_2 are at different instructions, then only inter-warp threads (*e.g.* $|t_2 - t_1| \leq$ warp_size) should be considered.

So far we have discussed parameterized race checking, soundness issues, and how to address SIMD. The next section shows how to build the parametric flow tree.

V. Parameterized Checking: Algorithm

```
__shared__ unsigned shared[NUM];

__global__ void BitonicKernel(unsigned* values) {
1:   unsigned int tid = threadIdx.x;
2:   // Copy input to shared mem.
3:   shared[tid] = values[tid];
4:   __syncthreads();
5:
6:   // Parallel bitonic sort.
7:   for (unsigned k = 2; k <= blockDim.x; k *= 2)
8:     for (unsigned j = k / 2; j > 0; j /= 2) {
9:       unsigned ixj = tid ^ j;
10:      if (ixj > tid) {
11:        if ((tid & k) == 0)
12:          if (shared[tid] > shared[ixj])
13:            swap(shared[tid], shared[ixj]);
14:        else
15:          if (shared[tid] < shared[ixj])
16:            swap(shared[tid], shared[ixj]);
17:      }
18:      __syncthreads();
19:    }
20:
21:  // Write result.
22:  values[tid] = shared[tid];
}
```

Fig. 6. The Bitonic Sort Kernel

We perform parameterized race checking by exploring a parametric flow tree (PFT) for two representative threads. One way is to construct a PFT for the entire program once and for all, then instantiate this tree with two parameterized threads. Another way (used in GKLEE$_p$) builds the tree and performs instantiations on the fly during symbolic execution. This approach fits well with our overall implementation approach and facilitates dynamic conflict checking (*e.g.* with respect to SIMD).

In a program, conditions may be purely concrete and be evaluated by GKLEE$_p$ to true or false immediately. Other conditions may depend on tid, bid, and/or symbolic inputs, and will be evaluated by forking new nodes in the PFT.

Roughly speaking, the construction of a PFT proceeds as follows (here we focus on how the symbolic executor constructs the tree, skipping most details discussed in §IV).

1) Starting with each *barrier* statement (the initial state of the program can be assumed to have one), GKLEE$_p$ launches one representative thread tid_0 — a symbolic value — for execution. So long as a conditional statement is not encountered, this representative thread would keep running until the next barrier is encountered.
2) When a $tid-$ or $bid-$ dependent condition is encountered, Two nodes are forked, one representing the threads satisfying the condition, the other one for those satisfying the negation of the condition.
3) Similarly, when a symbolic-input-dependent condition is encountered, two nodes may be forked to represent the true and false cases of the condition.
4) GKLEE$_p$ visits the nodes in an order consistent with SIMD. The nodes sharing a common ancestor will be "synchronized" at the first convergence point (*i.e.* the first post-dominator in the program). This mimics how diverged warps are executed in the hardware (see Section IV-C for more details).
5) Once all nodes reach an explicit barrier $\_\_syncthreads()$, the tree enters a synchronization status, and starts checking various kinds of errors (including intra-warp races).

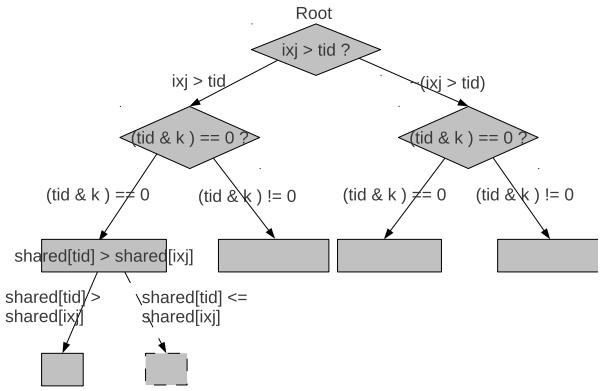

Fig. 7. Parametric flow tree for Bitonic Sort.

Figure 7 shows a portion of the PFT of the Bitonic Sort kernel. Since the first BI contains no conditional statements, no forking is needed. In the next BI, the PFT shown in Figure 7 is constructed during the execution, conditions in two outer loops are evaluated to be concrete values, so not shown in the parametric flow tree. The top three conditions are TDCs leading to node forking. The other four depend on both the symbolic inputs and built-in variables (*e.g.* tid and bid), and will lead to node forking too. The figure shows that flow 0 takes the path with path condition $ixj > tid$, $(tid\&k) == 0$ and $shared[tid] > shared[ixj]$. Note that memory accesses such as $shared[tid_0]$ are guarded by $ixj > tid_0 \wedge (tid_0\&k) == 0$.

In essence, GKLEE$_p$ follows a "canonical+SIMD" scheduling approach to build a PFT for parameterized thread tid_0. Recall that we need another thread, say tid_1, for conflict checking. Naturally, tid_1's PFT can be obtained through t_0's PFT by simply replacing tid_0 with tid_1. That is, by utilizing the symmetry of CUDA kernels, we can avoid executing tid_1 again to obtain its PFT. GKLEE$_p$ provides a facility to replace and simplify symbolic expressions, making it convenient to duplicate a PFT through cloning and thread id renaming.

For example, the bank conflict check requires two threads

involved, the write access $shared[tid_0]$ is guarded by a *TDC constraint*: $ixj_0 > tid_0 \wedge (tid_0 \& k) == 0$. Through *renaming*, thread tid_1's write access becomes guarded by its own *TDC constraint* $ixj_1 > tid_1 \wedge (tid_1 \& k) == 0$.

Extra care must be taken on the relation of two threads, which can be within the same warp, in different warps but in the same block, or in different blocks. The following shows the constraints over the thread ids for these scenarios.

Same Block and Same Warp:
$(bid_0 = bid_1) \wedge (tid_0 \neq tid_1) \wedge (\frac{tid_0}{WarpSize} = \frac{tid_1}{WarpSize})$

Same Block and Different Warps:
$(bid_0 = bid_1) \wedge (tid_0 \neq tid_1) \wedge (\frac{tid_0}{WarpSize} \neq \frac{tid_1}{WarpSize})$

Different Blocks: $(bid_0 \neq bid_1)$

VI. Experimental Results

GKLEE$_p$ supports (through command-line arguments) race and bank conflict detection for programs written with respect to CUDA Compute Capability 1.x (also called "SDK 1.x") as well as Capability 2.x (memory coalescing checks cover 1.0 through 1.3 models). All experiments are performed on a machine with Intel(R) Xeon(R) CPU @ 2.40GHz and 12GB memory. Our results about bank conflict and memory coalescing checks were done for 2.x device capabilities.

Table I presents results from SDK 2.0 kernels while Table II presents those from SDK 4.0 (many of these are also available in 2.x). Here, #T denotes the number of threads analyzed. Each cell contains a WW (write-write race), Ben. (benign race, meaning same value written by two concurrent writes), a Y or N (yes/no), or two numbers of the form A/B, where A is the tool runtime (in seconds) and B is the number of control-flow paths analyzed (*i.e.*, the TICs branched in so many ways). Most examples only generate one path, as there are no data-dependent control flow variations (except Bitonic sort, where these variations are essential to sorting).

Note that for Histogram64, GKLEE or GKLEE$_p$ explore multiple paths when $\#T = 32$ even though this example does not contain data dependent control flows. The reason for path generation is due to out-of-bound memory accesses happening (these generate a case analysis as explained in [15]). As another example, matrix multiplication using SDK 2.0 takes 362 seconds to explore the sole path using GKLEE while it only takes 3.4 seconds under GKLEE$_p$.

Since the occurrence of a race aborts the execution of GKLEE or GKLEE$_p$, for benchmarks involving races, we measured runtimes after switching off race checking.

Many of our results were obtained with respect to symbolic inputs[3]. For instance, as reported in [8] for runs using GKLEE, the Histogram64 example's race will be almost impossible to detect unless the first 10 bytes of a certain array are made symbolic (the same symbolic setting was used in runs using GKLEE$_p$ also).

[3]For details, please see our online benchmarks at http://www.cs.utah.edu/fv/gklee-parametric-flow-benchmarks/

Tables (I and II): These tables show that (i) all barriers were found to be well synchronized; (ii) the performance issues detected (bank conflict and non-coalesced memory accesses) were calibrated to the same degree of severity both by GKLEE and GKLEE$_p$. (We suppress detailed results in terms of the percentage of barrier intervals suffering from these performance issues, summarizing the results as Y/N.)

While none of our examples have a deadlock, it is the case that GKLEE$_p$'s ability to detect deadlocks has the same power as that in GKLEE. This is because GKLEE$_p$ accurately models all the flows that may result in deadlocks (modeling more threads within each flow equivalence class will not increase the number or kinds of deadlocks detected).

As for data races, (iii) all races listed in [8] were also detected by GKLEE$_p$. We also found additional inter-warp write-write races in SDK 2.0 kernels, thanks to the fact that we ran those examples with more than one thread block.

The races in kernels named Reduction4-6 were similar. Let us consider Reduction4 in some detail (more details on our website). This example has an instruction `if (blockSize ≥ 64) sdata[tid] += sdata[tid + 32]; EMUSYNC;` involving a read operation `sdata[tid + 32]` and a write operation `sdata[tid]`; these are involved in a read-write race by thread 0 and thread 32 that belong to two distinct warps. GKLEE$_p$ was able to automatically instantiate those two threads' identifiers.

Figure 8 and Figure 9 show that GKLEE$_p$ outperforms GKLEE with respect to different scales of number of threads, when $\#T = 8K$, GKLEE$_p$ speeds up matrix multiplication by a factor of 300 times. When $\#T = 16K$, GKLEE$_p$ speeds up transposeCoalesced by a factor of around 500 times.

Figure 10 presents the only kernel that GKLEE$_p$ did not perform better than GKLEE, since under GKLEE, all memory accesses are evaluated to be ones with the concrete address, whereas the time spent on constraint solving is still the main overhead of GKLEE$_p$. Figure 11 illustrates *clock* benchmark in which GKLEE$_p$ performs several times faster than GKLEE.

New results on Histogram64. In [8] we reported that GKLEE identified a possible WW race occurring within a warp. It has been an open question whether such race will manifest in the inter-warp cases (we could not run these larger models using GKLEE). GKLEE$_p$ checks whether two threads from different warps may cause such a race, and confirms that it will not (this race is present only within a warp). This demonstrates the added analysis power offered by GKLEE$_p$.

VII. Related Work

Past techniques [17], [18] generate finite-state abstractions of parameterized systems, apply induction [19], or seek cut-off bounds [20]. These techniques either require manual effort and do not apply to GPUs. Others focus on multi-threaded programs synchronizing using locks and semaphores [21]. These methods are impractical for GPU kernels.

There have been only a limited number of GPU-specific formal analyzers. An instrumentation based technique is reported [22] to find races and shared memory bank conflicts; in this

Kernels	Race	#T = 32		#T = 64		#T = 256		#T = 1,024		#T = 2,048		BC	MC
		GKLEE	GKLEE$_p$	GKLEE	GKLEE$_p$	GKLEE	GKLEE$_p$	GKLEE	GKLEE$_p$	GKLEE	GKLEE$_p$		
Bitonic Sort		T.O.	7.7/20	T.O	29.3/27	T.O	177.2/44	T.O	198/65	T.O	T.O	N	Y
Histogram64	WW	15.8/10	25.9/9	23.2/1	25.8/1	75.4/1	120/1	725.3/1	387.6/1	2682.0/1	904.6/1	Y	Y
Scalar Product		0.7/1	16.1/1	0.6/1	4.3/1	0.8/1	0.8/1	1.3/1	0.9/1	2.6/1	1.2/1	N	Y
Matrix Mult		0.2/1	4.5/1	0.4/1	4.0/1	2/1	3.2/1	19/1	2.8/1	362.1/1	3.4/1	N	Y
Reduction0		0.02/1	0.07/1	0.1/1	0.03/1	0.3/1	0.2/1	2.9/1	0.3/1	10.5/1	0.4/1	N	Y
Reduction1		0.01/1	0.1/1	0.1/1	0.1/1	0.8/1	0.2/1	8.1/1	0.3/1	24.0/1	0.5/1	Y	Y
Reduction2		0.02/1	0.1/1	0.03/1	0.1/1	0.2/1	0.1/1	2.9/1	0.3/1	10.2/1	0.4/1	N	Y
Reduction3		0.01/1	0.1/1	0.03/1	0.1/1	0.3/1	0.1/1	2.7/1	0.3/1	10.0/1	0.4/1	N	Y
Reduction4	WW	0.1/1	0.04/1	0.3/1	0.03/1	2.8/1	0.2/1	17.3/1	0.4/1	42.4/1	0.6/1	N	Y
Reduction5	WW	0.1/1	0.04/1	0.3/1	0.03/1	2.8/1	0.2/1	11.4/1	0.4/1	21.3/1	0.5/1	N	Y
Reduction6	WW	0.1/1	0.05/1	0.3/1	0.04/1	2.8/1	0.2/1	11.5/1	0.4/1	22.6/1	0.6/1	N	Y
Scan Best		0.3/1	3.6/1	2.1/1	5.1/1	48.8/1	8.1/1	923.3/1	12.5/1	T.O.	26.6/1	Y	Y
Scan Naive		0.04/1	0.2/1	0.2/1	0.4/1	3.4/1	0.5/1	66.0/1	0.9/1	291.8/1	15.2/1	N	N
Scan WorkEfficient		0.1/1	0.6/1	0.4/1	0.8/1	12.1/1	1.2/1	250.8/1	2.1/1	T.O.	3.1/1	Y	N
Scan Large		0.2/1	2.3/1	1.4/1	3.0/1	40.0/1	3.9/1	736.1/1	2.1/1	T.O.	2.1/1	Y	Y
Bisect Small	Ben.	2.2/1	105.9/1	3.5/1	108.8/1	10.6/1	108.7/1	36.0/1	108.8/1	58.1/1	233.7/1	N	Y
Bisect Large	Ben.	T.O.	226.0/1	T.O.	203.0/1	T.O.	212.6/1	T.O.	218.5/1	T.O.	248.1/1	Y	Y

TABLE I
SDK 2.0 KERNEL RESULTS. WE SET 7200 SECONDS AS THE THRESHOLD FOR TIME OUT (ABBREVIATED AS T.O.) "BC" AND "MC" ARE THE ABBREVIATIONS OF "BANK CONFLICT" AND "COALESCED GLOBAL MEMORY ACCESSES", AND THE RESULTS OF THESE TWO CATEGORIES ARE ACQUIRED THROUGH GKLEE$_p$.

Kernels	Race	#T = 1,024		#T = 2,048		#T = 4,096		#T = 8,192		#T = 16,384		BC	MC
		GKLEE	GKLEE$_p$	GKLEE	GKLEE$_p$	GKLEE	GKLEE$_p$	GKLEE	GKLEE$_p$	GKLEE	GKLEE$_p$		
Clock		3.8/1	12.1/1	6.1/1	12.2/1	9/1	12.6/1	26.6/1	13/1	92.2/1	13.9/1	N	Y
Scalar Product		50.9/1	97.9/1	213.2/1	200.2/1	902.9/1	410.9/1	T.O.	859.4/1	T.O.	1812.2/1	N	Y
Histogram64		122.3/1	50.8/1	158.7/1	55.7/1	283.8/1	65.9/1	511.8/1	85.1/1	T.O.	120.0/1	Y	N
Scan Short		36.3/1	18.1/1	92.5/1	32.3/1	216.4/1	62.6/1	714.8/1	116.7/1	3222.7/1	227.0/1	Y	N
Scan Large		40.1/1	92.9/1	107.5/1	133.9/1	336.6/1	482.2/1	1175.1/1	761.4/1	6134.4/1	555.7	Y	N
Copy		0.1/1	0.1/1	0.3/1	0.1/1	0.8/1	0.1/1	2.8/1	0.1/1	10.2/1	0.1/1	N	Y
copySharedMem		0.8/1	0.3/1	1.6/1	0.6/1	11.1/1	0.3/1	23.2/1	0.7/1	172.7/1	0.6/1	N	Y
transposeNaive		0.2/1	0.2/1	0.3/1	0.1/1	1.0/1	0.1/1	3.4/1	0.2/1	11.6/1	0.2/1	N	N
transposeCoalesced		4.7/1	0.2/1	9.6/1	0.3/1	27.3/1	0.3/1	55.3/1	0.4/1	242.4/1	0.5/1	Y	Y
transposeNoBankConflicts		0.8/1	0.4/1	1.8/1	0.4/1	11.3/1	0.4/1	24.1/1	0.5/1	179.2/1	0.7/1	N	Y
transposeDiagonal		0.8/1	0.3/1	1.7/1	0.4/1	11.2/1	0.4/1	23.5/1	0.5/1	172.1/1	0.6/1	N	Y
transposeFineGrained		0.8/1	0.3/1	1.7/1	0.4/1	11.1/1	0.4/1	23.2/1	0.5/1	170.0/1	0.6/1	N	Y

TABLE II
SDK 4.0 KERNEL RESULTS. NO RACES ARE FOUND IN THESE KERNELS.

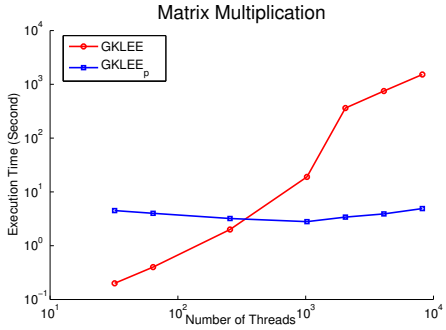

Fig. 8. Matrix Multiplication (SDK 2.0)

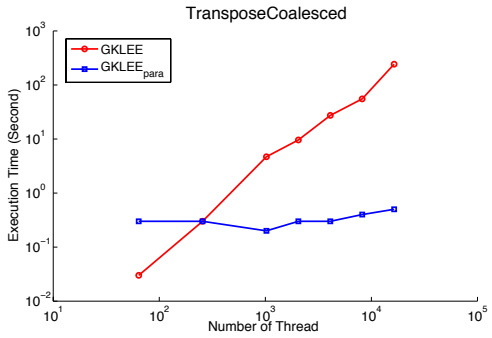

Fig. 9. TransposeCoalesced (SDK 4.0)

work, the program is instrumented with checking code, and only those executions occurring in a platform-specific manner are considered. A similar method [4] is used to find races. Static analysis is performed first to locate possible candidates for further dynamic analysis. These runtime methods cannot accept symbolic inputs, nor are able to handle a large number of threads in a parameterized way.

PUG [5] employs symbolic static analysis on individual kernels—not whole programs. It also needs considerable manual effort and suffers from the false alarm problem. It uses loop invariants to avoid unrolling any loop. GPUVerify [23] employs a similar analysis but improves the loop invariants for accumulative variables so as to reduce the false alarm rate. However finding a sufficient set of loop invariants for more complicated patterns is still a challenge. In a recent workshop paper [24], we have shown that for a class of CUDA programs, one can parametrically check functional correctness (rather than races). It requires the loops to be of particular formats. This approach has not been shown to work for practically sized CUDA programs and requires manual annotation effort. In [25] the authors provide a static analysis based approach to identify when concrete tests can represent families of tests that cause similar (for race checking) control flows.

GKLEE [8] builds a virtual machine (VM) modeling thread

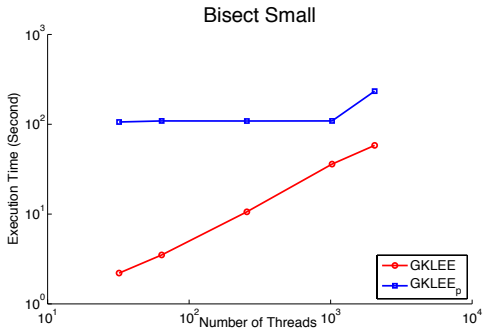

Fig. 10. Bisect Small (SDK 2.0)

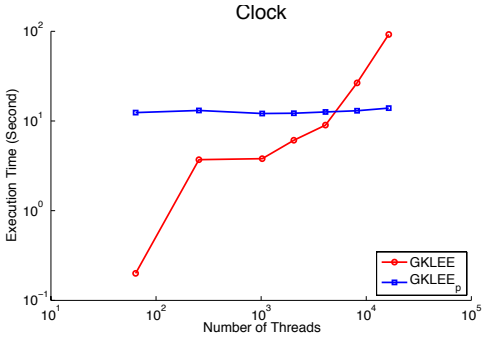

Fig. 11. Clock (SDK 4.0)

computations on GPUs. When a GPU program is executed in the VM, the tool checks for deadlocks, data races, and performance bugs on a limited number of threads, but more exhaustively than GKLEE$_p$. However, as already shown, even on the BitonicSort kernel (of about 50 lines of code), GKLEE becomes impractical when the thread number is greater than 8. Another symbolic execution tool KLEE-CL [6] is not parameterized and suffers from the same problem.

The work in this paper scales well, is tailored for CUDA and also will work with emerging standards (*e.g.*, OpenCL [26]).

VIII. Concluding Remarks

Parameterized reasoning is a formally undecidable problem, and therefore in every domain, one has to find syntactic restrictions under which one can check or prove correctness parameterically. Symbolic execution based verification is highly attractive in that one is able to bring the benefits of formal analysis to real code (not models of the code) written in practical languages (e.g., C++) and compiled using actual compilers (e.g., LLVM). In this paper, we employ parameteric-style reasoning in the symbolic execution context for race detection in the context of GPU programs. Despite the theoretical inexactness of this approach, our results show that we have caught all the races found in our earlier efforts, found some new ones, and have been able to scale verification to 16K threads, finishing verification within acceptable runtimes. This makes GKLEE$_p$ a practical race checking facility—the first of its kind—that also allows programmers to choose symbolic inputs and obtain code coverage over all the branches that depend on these inputs.

In our future work, GKLEE$_p$ will be extended to handle many of the advanced CUDA features including CUDA atomics, multiple contexts, and libraries such as Thrust. Integration into high-productivity development frameworks (e.g., Ecplise Parallel Tools Platform) and handling of accumulative variables will also be a priority.

References

[1] NVIDIA, "CUDA-GDB," Jan. 2009, an extension to the GDB debugger for debugging CUDA kernels in the hardware.

[2] A. S. Ltd, "Allinea DDT," http://www.allinea.com/products/ddt.

[3] NVIDIA, "CUDA-MEMCHECK," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/cuda-memcheck.pdf.

[4] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "GRace: A low-overhead mechanism for detecting data races in GPU programs," in *PPoPP*, 2011.

[5] G. Li and G. Gopalakrishnan, "Scalable SMT-based verification of GPU kernel functions," in *SIGSOFT FSE*, 2010, www.cs.utah.edu/fv/PUG.

[6] P. Collingbourne, C. Cadar, and P. Kelly, "Symbolic testing of OpenCL code," in *Haifa Verification Conference (HVC)*, 2011.

[7] P. Collingbourne, C. Cadar, and P. H. Kelly, "Symbolic crosschecking of floating-point and simd code," in *EuroSys*, 2011.

[8] G. Li, P. Li, G. Sawaga, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "GKLEE: Concolic verification and test generation for GPUs," in *PPoPP*, 2012, www.cs.utah.edu/fv/GKLEE.

[9] "Satisfiability Modulo Theories Competition (SMT-COMP). http://www.smtcomp.org/2012."

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

[11] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," in *SOSP*, 1997.

[12] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, pp. 558–565, 1978.

[13] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *POPL*, 2005, pp. 110–121.

[14] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Efficient stateful dynamic partial order reduction," in *SPIN*, 2008.

[15] G. Li, P. Li, G. Sawaga, and G. Gopalakrishnan, "GKLEE: Concolic verification and test generation for GPUs," University of Utah, Tech. Rep., 2012, available at www.cs.utah.edu/fv/GKLEE.

[16] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. W. Hwu, "Efficient compilation of fine-grained spmd-threaded programs for multicore CPUs," in *Code Generation And Optimization*, 2010.

[17] A. Pnueli, J. Xu, and L. D. Zuck, "Liveness with (0, 1, infty)-counter abstraction," in *CAV*, 2002.

[18] E. M. Clarke, M. Talupur, and H. Veith, "Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems," in *TACAS*, 2008.

[19] A. Pnueli, S. Ruah, and L. D. Zuck, "Automatic deductive verification with invisible invariants," in *TACAS*, 2001.

[20] E. A. Emerson and K. S. Namjoshi, "Reasoning about rings," in *POPL*, 1995.

[21] C. Flanagan and S. N. Freund, "Type-based race detection for Java," in *PLDI*, 2000.

[22] M. Boyer, K. Skadron, and W. Weimer, "Automated dynamic analysis of CUDA programs," in *STMCS*, 2008.

[23] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: A verifier for GPU kernels," in *SPLASH*, 2012.

[24] G. Li and G. Gopalakrishnan, "Parameterized verification of GPU kernel programs," in *PLC Workshop (part of IPDPS)*, May 2012.

[25] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner, "Verifying GPU kernels by test amplification," in *PLDI*, 2012.

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

APPENDIX

When a variable depends on the accumulation of values across more than two threads, our method either (1) introduces approximations by keeping the accumulative variable unconstrained (*i.e.* it can have any value), or (2) uses N threads to calculate the exact value of the accumulative variable. Specifically, consider the following concocted example where the shared variable v's value is the sum of $A[i]$ for $0 \leq i < N$. A loop containing a barrier is used to avoid Write-Write races on v.

```
__shared__ int A[N]; // symbolic array A
for (int i = 0 ; i < N; i++) {
  if (i == tid) v += A[i];
  __syncthreads();
}
v is used in some accesses;
```

The first method suggested above will keep v's final value to be in $(-\infty, +\infty)$. To obtain accurate results, we can fully unroll the loop (note that N is a concrete bound), resulting in $v = \Sigma_{i=0}^{N-1} A[i]$.

The difficulty is to find out whether a variable is accumulative or not. We use a simple strategy which marks variable v accumulative at an execution point when (1) it is a shared/global variable (we also use variable to refer to an element in an array), (2) it is written (updated) by the threads with different values up to this execution point. It should be noted that, while this seems to be similar to the definition of data races, here we actually combine the accesses in all BIs to check overlapping. The idea behind this strategy is that when a shared variable is updated by multiple threads with different values (*e.g.* in different BIs), its final value may depend on the contributions from more than two threads. Any variable that is data-flow-dependent of an accumulative variable is also accumulative.

For illustration let us consider three case studies:

- In the motivating example, obviously local variable idx is non-accumulative (it is actually tid- and bid- dependent). Variable $b[tid]$ is non-accumulative since it is "private" to thread tid such that all the updates during the entire execution are made by only this thread.
- In the above dummy loop example, shared variable v is updated by thread i at iteration i for $0 \leq i < N$. Each thread writes a different value to v (unless all the elements in A is 0). Hence v is an accumulative variable.
- In the BitonicSort kernel in Figure 6, the first assignment, shared[tid] = values[tid], writes to shared array $shared$. Since each thread updates only its "private" element in the array, $shared[tid]$ is non-accumulative at this point. Similarly local variables k, j, ixj are non-accumulative (they are tid- and bid- dependent). The cases about $shared[tid]$ and $shared[ixj]$ are more tricky. At the first iteration, they are updated by only one thread, hence are non-accumulative. However, in the subsequent iterations, they may be updated by other threads, turning them into accumulative variables. In other words, the same location in $shared$ may be updated by different threads at different iterations; in this case its elements become accumulative, and condition $shared[tid] > shared[ixj]$ may be evaluated imprecisely. Nevertheless, we will show that this will not affect our ability to find out the races.

A. Handling Accumulative Variables

In a typical CUDA kernel, accumulative variables are seldom referred by the addresses of accesses. However, as shown in the BitonicSort kernel, it is not uncommon that accumulative variables appear in the conditions guarding these accesses.

For this, a simple two-phase solving scheme can facilitate handling accumulative variables during conflict checking. For instance, suppose that accumulative variable v is used in a condition guarding access $A[f(tid)]$. One optimization is to first disregard condition $c(v)$ (*e.g.* by havocing the value of v) and check where $A[f(tid)]$ leads to a race, if not then no race exists. Otherwise we can take the second step to examine $c(v)\ ?\ A[f(tid)]$. This simple optimization is able to boost the performance substantially in the presence of accumulative variables. In fact, for virtually all of our benchmark programs, we do not need the second step since all race checks can be resolved without considering accumulative variables – which is typical for CUDA kernels.

```
... // the above loop
if (c(v)) {
  A[f(tid)] = ...;
}
```

For example, for accesses $shared[tid]$ and $shared[ixj]$ in the BitonicSort kernel, race checking requires examining whether non-accumulative variables tid and ixj can overlap in a BI. We can give the answer without considering the guard condition $share[tid] > shared[ixj]$ and its negation. The fact of race-freedom is warranted by the following constraints (where we subscript the variables with thread ids).

$$tid_1 \neq tid_2 \Rightarrow ixj_1 \neq ixj_2 \wedge \\ ixj_1 > tid_1 \ \wedge \ ixj_2 > tid_2$$

The exclusion of an accumulative condition c when evaluating $c\ ?\ a$ can be done implicitly. Suppose we do not accurately model c's value, *e.g.* another symbolic condition c' is used, then by evaluating both $c'\ ?\ a$ and $\neg c'\ ?\ a$, we can know whether a causes races due to the fact that $(c' \wedge a \ \vee \ \neg c' \wedge a) = a$. This method allows us, as indicated in Section V, to dynamically build a branching tree and check races without keeping track of where c is accumulative. Only when a causes races should we consider c to eliminate false alarms. For instance, when considering $share[tid] > shared[ixj]$, the values of $shared[tid]$ and $shared[ixj]$ may be inaccurate when checking races over the accesses under this condition. But this matters only when a race is found, in which case we need to use the accurate values of the two variables to rule out false race alarms. This trick releases us from the burden of keeping track of most accumulative variables, which are more related to functional correctness rather than races for typical GPU programs.