Practical Symbolic Race Checking of GPU Programs

Peng Li
School of Computing
University of Utah, UT, USA
Email: peterlee@cs.utah.edu

Guodong Li
Fujitsu Labs of America, CA, USA
Email: gli@fla.fujitsu.edu

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

***Abstract*—Even the careful GPU programmer can inadvertently introduce data races while writing and optimizing code. Currently available GPU race checking methods fall short either in terms of their formal guarantees, ease of use, or practicality. Existing symbolic methods: (1) do not fully support existing CUDA kernels; (2) may require user-specified assertions or invariants; (3) often require users to guess which inputs may be safely made concrete; (4) tend to explode in complexity when the number of threads is increased; and (5) explode in the face of thread-ID based decisions, especially in a loop. We present SESA, a new tool combining Symbolic Execution and Static Analysis to analyze C++ CUDA programs that overcomes all these limitations. SESA also scales well to handle non-trivial benchmarks such as Parboil and Lonestar, and is the only tool of its class that handles such practical examples. This paper presents SESA's methodological innovations and practical results.**

Keywords: GPU, CUDA, Parallelism, Symbolic Execution, Formal Verification, Virtual Machine, Taint Analysis, Data Flow Analsis

I. Introduction

GPUs offer impressive speedups on many problems; unfortunately, the detection and elimination of bugs in GPU programs is a serious productivity impediment. In this work, we focus on accurate (false-alarm free) detection of data races in GPU programs. A data race exists in an execution when two threads make a happens-before [13] unordered access to the same memory location where one access is a write. Races can not only affect a program's result, it can also cause a compiler to optimize code incorrectly, as many such optimizations assume data-race freedom.

Conventional GPU debuggers [21], [2], [20] are ineffective at finding and root-causing races that are dependent on thread schedules and inputs. Besides they check only for the races manifested by the particular platform, and not the races according to the execution model (that may occur on some unknown future platform). To enhance coverage, many formal and semi-formal analysis tools have recently been proposed. They employ a combination of static/dynamic [27], or symbolic analysis, as employed by the tools PUG [15], KLEE-CL [9], KLEE-FP [10], and GPUVerify [8]. In almost all these tools, symbolic analysis is supported by SMT-solvers [24] which are steadily growing in their capacity. GPUVerify often requires user input to specify input- and loop constraints. KLEE-CL detects races and also helps cross-check Open-CL codes against sequential code. KLEE-FP focuses on floating-point result cross-checking across sequential and parallel codes.

SC14, November 16-21, 2014, New Orleans, USA
978-1-4799-5500-8/14/$31.00

Concolic ("concrete plus symbolic") execution tools such as GKLEE [17], [18] have also recently been proposed, and have helped find bugs in real-world GPU kernels. As the work in this paper is based on concolic execution, we make more detailed comparisons against these tools. A GKLEE user writes standard C++ CUDA programs, indicating some of the program variables to be *symbolic* (the rest are assumed to be concrete or *ground* 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 (as governed by user input and available resources) branches. As added benefit, concolic execution tools can also generate concrete tests.

Existing concolic execution based GPU program correctness analysis tools suffer from two major drawbacks. First, they require users to pick the desired symbolic inputs. Inadvertently picking less symbolic inputs causes omissions, while picking excessively burdens the symbolic analysis engine (typically an order of magnitude slower than concrete execution). These tools also *model and solve the data-race detection problem over an explicitly specified (and often small) number of GPU threads.* This makes these tools difficult to apply to analyze realistic programs that assume a certain minimum number (and often much larger) number of threads. Moreover, downscaling the number of threads together with other problem parameters in a consistent way is often impractical.

A recent tool GKLEE$_p$ [18] provides more scalability by exploiting thread symmetry, but still requires users to pick the symbolic inputs. In particular, this tool partitions the space of executions of a GPU program into *parametrically equivalent flows*, and models the race analysis problem over *two parametric threads* in each equivalence class. GKLEE$_p$ can scale to thousands of threads for *non-divergent programs* (*i.e.* programs not forking paths w.r.t. symbolic inputs). Unfortunately, GKLEE$_p$ still suffers from search explosion: even if only two symbolic threads are considered, each thread may create a large number of flows or symbolic states. For instance,

if a thread contains n feasible branches, then $O(2^n)$ flows or paths may be generated. As shown later, this scenario is not uncommon in realistic CUDA programs. We present a new tool SESA (Symbolic Executor with Static Analysis) that significantly improves over prior tools in its class both in terms of new ideas and new engineering:

1) SESA implements a new front-end (based on Clang). It also supports all core CUDA C++ instructions, 95 arithmetic intrinsics, 25 type conversion intrinsics, all atomic intrinsics and 82 CUDA runtime functions.
2) SESA is the first tool to employ data-flow analysis to combine parametrically equivalent flows that may otherwise exponentially grow in many examples.
3) SESA employs static ("taint") analysis to identify inputs that can be concretized without loss of verification coverage, while significantly speeding up verification. This analysis has yielded fairly precise results in practice (very little over-approximation), partly helped by the selective use of loop unrollings.
4) SESA can scale to thousands of threads for typical CUDA programs. It has been used to analyze over 50 programs in the SDK and popular libraries such as Parboil [23] and Lonestar [19]. It reveals at least 3 new bugs that have not been reported by any other tool before. Previously reported formally-based GPU analysis tools have not handled such practical examples before.
5) We describe conditions under which SESA is an exact race-checking approach, and also present when it can miss bugs. In all our experiments so far, these unusual patterns have not arisen.

II. Background

A CUDA kernel is launched as an 1D, 2D or 3D *grid* of *thread blocks*. The total size of a 3D grid is `gridDim.x` × `gridDim.y` × `gridDim.z`. Each block at location ⟨`blockIdx.x`, `blockIdx.y`, `blockIdx.z`⟩ has dimensions `blockDim.x`, `blockDim.y` and `blockDim.z`. Each block contains `blockDim.x` × `blockDim.y` × `blockDim.z` threads, with IDs ⟨`threadIdx.x`, `threadIdx.y`, `threadIdx.z` ⟩. These threads can share information via *shared memory*, and synchronize via *barriers*. However, threads belonging to distinct blocks must use (the slower) *global memory* to communicate and synchronize. The values of `gridDim` and `blockDim` determines the *configuration* of the system, *e.g.* the sizes of the grid and each block. For a thread, `blockIdx` and `threadIdx` give its block index in the grid and its thread index in the block respectively. For brevity, we use $gdim$ to denote $gridDim$, bid for $blockIdx$, $bdim$ for $blockDim$, and tid for $threadIdx$. For $* \in \{x, y, z\}$, the constraints $bid.* < gdim.*$ for $* \in \{x, y, z\}$ and $tid.* < bdim.*$ always hold. Groups of 32 (a "warp") consecutively numbered threads within a thread block are scheduled in a Single Instruction Multiple Data (SIMD) fashion.

Data Race Examples. The following example '`race`' contains two classes of races: (1) In the statement before the barrier, thread 0 and thread $bdim.x - 1$ may race on $v[0]$ (and similarly for the remaining adjacent threads). (2) In the conditional after the barrier, one thread may execute the *then* part while others execute the *else* part, and there is no guarantee that these accesses are ordered in a specific way (hence may change across GPU families).

```
__global__ void race() {
  v[tid.x] = v[(tid.x + 1) % bdim.x];
  __syncthreads();
  if (tid.x % 2 == 0) { ... = v[tid.x] ; }
  else { v[tid.x >> 2] = ... ; }
}
```

For race checking, SESA records the *Read Set* and *Write Set* of shared variables. For the above example, the code from the beginning to the barrier constitutes the first *barrier interval*. In this barrier interval, the read set and write set of thread tid are $\{v[(tid.x + 1) \,\%\, bdim.x]\}$ and $\{v[tid.x]\}$ respectively. To check races, we instantiate the read set and write set for two different threads as following. For WW (Write-Write) races, we check whether an access in t_1's write set can have the same address as an access in t_2's write set. This is reduced to checking where $t_1.x = t_2.x$ holds, which is false since t_1 and t_2 are different threads. Similarly, to check WR races, each element in t_1's write set is compared with each element in t_2's read set, *i.e.* where $t_1.x = (t_2.x + 1)\%bdim.x$ is satisfiable for $t_1.x \neq t_2.x \wedge t_1.x < bdim.x \wedge t_2.x < bdim.x$. A constraint solver can find a solution, *e.g.* $t_1.x = 0$ and $t_2.x = bdim.x - 1$ for any $bdim.x \neq 0$. This gives a witness of the WR race by threads 0 and $bdim.x - 1$. Note that we need not to compare $v[t_2.x]$ and $v[(t_1.x+1) \,\%\, bdim.x]$ since t_1 and t_2 are symmetric.

	thread t_1	thread t_2
WriteSet :	$\{v[t_1.x]\}$	$\{v[t_2.x]\}$
ReadSet :	$\{v[(t_1.x + 1)\%bdim.x]\}$	$\{v[(t_2.x + 1)\%bdim.x]\}$

The code after the barrier constitutes the second barrier interval. The read set and write set of thread tid contain conditional accesses of format '`condition?access`'. This accurately models the cases of divergent threads. That is, the sets are the same whether the *then* part or the *else* is first executed. Race checking is similar to the above procedure, except that the conditions must be taken into account. For instance, there exists a RW race since formula $t_1.x \,\%\, 2 = 0 \wedge t_2.x \,\%\, 2 \neq 0 \wedge t_1.x = t_2.x \gg 2$ is satisfiable, *e.g.* when $t_1.x = 0$ and $t_2.x = 1$. This happens no matter whether t_1 and t_2 are within a warp or not. For instance, the race manifests when t_1 executes the *else* part while t_2 idles, then t_2 executes the *then* part while t_2 idles.

	thread t_1	thread t_2
WriteSet :	$\{t_1.x\%2 \neq 0 \;?\; v[t_1.x \gg 2]\}$	$\{t_2.x\%2 \neq 0 \;?\; v[t_2.x \gg 2]\}$
ReadSet :	$\{t_1.x\%2 = 0 \;?\; v[t_1.x]\}$	$\{t_2.x\%2 = 0 \;?\; v[t_2.x]\}$

SESA inherits several features from its predecessors [17], [18]. In particular, it has the ability to do race-checking under standard warp-sizes, or a warp-size of 1. The latter option is important because many programmers rely on warp semantics and run into inexplicable races, as studied and explained in [25], [26]. A CUDA compiler can "assume" that the warp size is 1, and may mis-compile codes that race under this

view. SESA also checks for global memory races—a feature missing in commercial tools such as [20].

III. New Techniques

```
/*-----------------------------------------------*/
// Generic Example

// local: u, v, w, z;
// global: a, b, c, array A
   ...  // update c
1:    if (e1(tid)) then v = a;
2:    else v = b;
3:    if (e3(c)) u = e2(tid);
4:    A[w] = v + z;

/*-----------------------------------------------*/

// Reduction kernel

__shared__ int sdata[NUM * 2];
__global__ void reduce(float *idata, float *odata) {
1:  ...;   // copy idata to sdata
2:  for(unsigned int s = 1; s < blockDim.x; s *= 2) {
3:    if (tid % (2*s) == 0)
4:      sdata[tid] += sdata[tid + s];
5:    __syncthreads();
6:  } // end for
7:}...;  // copy sdata to odata

/*-----------------------------------------------*/

// Bitonic kernel

__global__ void BitonicKernel(unsigned *values) {
1:  for (unsigned int k = 2; k <= blockDim.x; k *= 2) {
2:    for (unsigned int j = k / 2; j>0; j /= 2) {
3:      unsigned int ixj = tid ^ j;
4:      if (ixj > tid) {
5:        if ((tid & k) == 0) {
6:          if (shared[tid] > shared[ixj])
7:            swap(shared[tid], shared[ixj]);
8:        }
9:        else {
10:         if (shared[tid] < shared[ixj])
11:           swap(shared[tid], shared[ixj]);
12:     }}
13:    __syncthreads();
14:   } // end for 2
15:}} //end for 1
```

Fig. 1. Example kernels Generic, Reduction, and Bitonic

To explain the features in SESA, consider three examples, namely Generic, Reduction, and Bitonic (Figure 1). One of the central problems of GKLEE stemmed from its explicit modeling of every thread in a CUDA program. GKLEE$_p$ improved the situation by capitalizing on the symmetry inherent in a CUDA program. For example, consider the Reduction example in Figure 1. If there is a data race experienced by a thread satisfying the condition listed on line 3, such a race will also be experienced by a group of threads satisfying this condition. Thus, GKLEE$_p$ splits the threads into two equivalence classes at line 3, and models *two symbolic threads t_1 and t_2 for each of these equivalence classes*. In [18], it was shown that GKLEE$_p$ can simply proceed on the assumption that $t_1 \neq t_2$ and detect the same class of races as GKLEE, and thus avoid directly facing the complexity of modeling the actual number of threads (hence the name "parametric flows"). However, this approach of GKLEE$_p$ can generate an exponential number of flows (four flows in Generic, and many more in the others). In section III-A and section III-B, we explain how such flows are combined by SESA through some examples.

A. Flow Combining With Respect to Local Variables

In our Generic example, notice that local variable `v` is set to `a` or `b` depending on condition `e1(tid)`. Since the CUDA program will be populated with thousands of threads, one has to track the behavior of threads satisfying `e1(tid)` and `!e1(tid)` separately. This can become a huge overhead, especially if such conditionals occur within loops, as in Reduction kernel, line 3, and Bitonic kernel, lines 4 and 5. (GKLEE$_p$ times out on these examples). SESA applies its static analysis to avoid this situation.

Observe the statement `A[w]` on line 4 of Generic, where `A` is a global array. If it is possible for two different threads to be performing this update concurrently, there could be a data race on `A[w]`, depending on how `w` is calculated as a function of the thread ID. However, if the index `w` in `A[w]` does not depend on `v` (determined through static analysis), the manner in which TIDs split based on `e1(tid)` and `!e1(tid)` has no influence on whether `A[w]` will incur a race or not. In this case, it suffices to maintain a single flow that is not predicated on `e1` at all. SESA employs static analysis to identify whether local variables flow into sensitive sinks, namely shared/global memory addresses or conditionals downstream in the code. In the Bitonic kernel, the conditionals at lines 4 and 5 carry on with a single flow through them, avoiding flow splitting.

B. Flow Combining With Respect to Global Variables

Now consider the Generic example where global variable `c` affects the value that a particular thread assigns to `u` (via `e2(tid)`). Suppose global variable `c` is assigned a symbolic value, then two branches are supposed to be exploited. Since `u`'s value is not used in `w`, we can combine the flows again at line 3 of the Generic example. This is how we handle the "flow explosion" due to the conditionals on lines 6 and 10 of the Bitonic kernel: even though we are updating `shared[..]` through the swap function, this updated state does not flow into any of the future sensitive sinks.

C. Symbiotic Use of Static and Symbolic Analysis

No practical tool can be entirely push-button; this is especially so for symbolic analysis tools. In particular, while SESA has static data-flow and taint analysis capabilities, *users must still intervene and set loop bounds concrete* (without bounding loops, concolic execution tools cannot finish their search; the alternative is to discover loop invariants which is far from a practical approach for the average CUDA programmer).

Fortunately, SESA provides information on *why* certain inputs must be kept symbolic. Those inputs that are deemed to be symbolic because they flow into array index expressions, the programmer must proceed treating these inputs purely symbolic. Finally, for those inputs found not to flow into array index or control expressions, SESA allows the programmer to safely set them to concrete values.

In our results section §VI, we show that the combination of the aforesaid static analysis and the flow combining approaches of §III-A and §III-B were essential to handle practical benchmarks such as the Lonestar benchmark.

IV. Parametric Execution and Race Checking

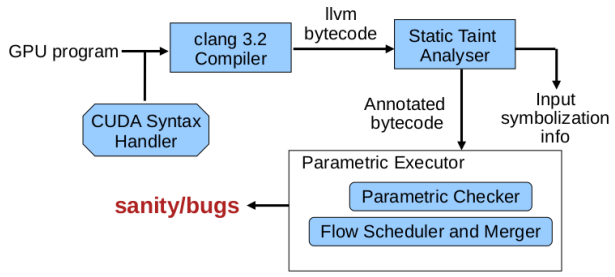

Fig. 2. SESA's infrastructure.

Figure 2 shows the infrastructure of SESA, The front-end uses the `Clang-3.2` compiler to translate a CUDA program into LLVM bytecode (with this front-end, support for OpenCL [22] is within reach, and is being planned). The bytecode is first processed by the static analyzer (§V) to add annotations on data-flow information, specifically whether a variable can flow into sensitive sinks. The annotated bytecode is then interpreted by the symbolic executor during parametric execution (§IV-A), at which time, races are checked (§IV-B). The annotated bytecode contains information about what inputs should be symbolic as well as flow into sensitive sinks; this guides the symbolic executor to carry out the functions described in §III (§V). SESA currently supports all core LLVM instructions, 95 arithmetic intrinsics, 25 type conversion intrinsics, all atomic intrinsics and 82 CUDA runtime functions. This amounts to ∼3K LOC new for the basic infra-structure. This represents substantial effort in making a practical C++ CUDA front-end; existing formal approaches to GPU correctness do not have these features.

A. Parametric Execution of CUDA Programs

τ	:=	τ_l, τ_g	memory sort
var	:=	$var_{cuda} \mid v : \tau$	variable
var_{cuda}	:=	$tid, bid, \ldots$	CUDA built-in
lab	:=	$l_1, l_2, \ldots$	label
e	:=	$var \mid n$	atomic expression
$instr$	:=	`br` v lab lab	conditional branch
		`br` lab	unconditional jump
		`store` e v	store to addr e value v
		$v =$ `load` e	load from addr e
		$v =$ `binop` e e	binary operation
		$v =$ `alloc` n τ	memory allocation
		$v =$ `getelptr` v e ...	address calculation
		$v =$ `phi` $[lab, v]$ ...	control-flow merge
		`syncthreads`	synchronization barrier

Fig. 3. Summary syntax of CUDA bytecode.

In this section, we recap the essential ideas of parametric execution introduced in GKLEE$_p$, and point out key innovations made in this work. We present the basics of the CUDA syntax handled (as captured at the LLVM level), our store model, how SIMD execution is carried out, and the basics of preserving parametricity. In §IV-B, we present the highlights of parametric race checking.

Figure 3 shows an excerpt of the syntax of CUDA LLVM bytecode. We now provide a high level view of how a collection of CUDA threads execute from the perspective of race checking. For this, we focus on the "Race State" of each thread. Given a thread ID and a flow conditional (flow_cond), a single access is either a read (r) or a write (w) followed by the address being accessed. For brevity, consider a sequence of race states of the form $c_1?r(a_1), c_2?r(a_2), c_3?w(a_3), \ldots$ that a single thread evolves over. We now define the notion of a *Parametric Execution*.

Parametric Execution. Because of the thread symmetry, within each barrier interval in a CUDA program, threads execute across an "identical-looking" race history. Specifically, consider a barrier interval containing instruction I. This instruction syntactically looks the same across all threads. Thus, given one race history of thread t_i

$$c_1(i)?r(a_1(i)), c_2(i)?r(a_2(i)), c_3(i)?w(a_3(i)), \ldots$$

we can obtain the race history of another thread t_j by simply replacing every occurrence of i with j.

Now, when can we claim that regardless of the number of threads executing a barrier interval, we can detect all data races within the barrier interval by simply modeling two symbolic threads t_i and t_j and checking races across just these threads (this is the key idea of parametric checking). In §IV-B, we proceed to describe these ideas can be used to efficiently perform race checking without incurring parameterized flow explosion.

B. Parametric Flows and Race Checking

A barrier interval may contain multiple conditions where the threads diverge over. We discuss divergent computations in terms of conditional SIMD instructions. A conditional SIMD instruction is of format $c\ ?\ is_l : is_r$, where c is the condition, instruction sequences is_l and is_r will be executed when c is true and false respectively. When n threads diverge over this instruction, the threads satisfying the condition will execute is_l while other threads are idle. The is_r case is analogous. The execution orders of is_l and is_r are not deterministic.

For the ease of exposition, consider the case where the conditions c are a function of thread id tid. Such conditions divide the threads into 2 groups, one satisfying c and one satisfying $\neg c$. We say that this condition creates two parametric flows. Each flow represents a group of threads with a flow condition. Since the threads within a flow perform similar operations, they can be reasoned about using a parametric thread.

Observation. A parametric flow represents a set of threads which perform parametric computations under a flow condition. In the case of multiple conditions c_1, c_2, ..., c_k, each

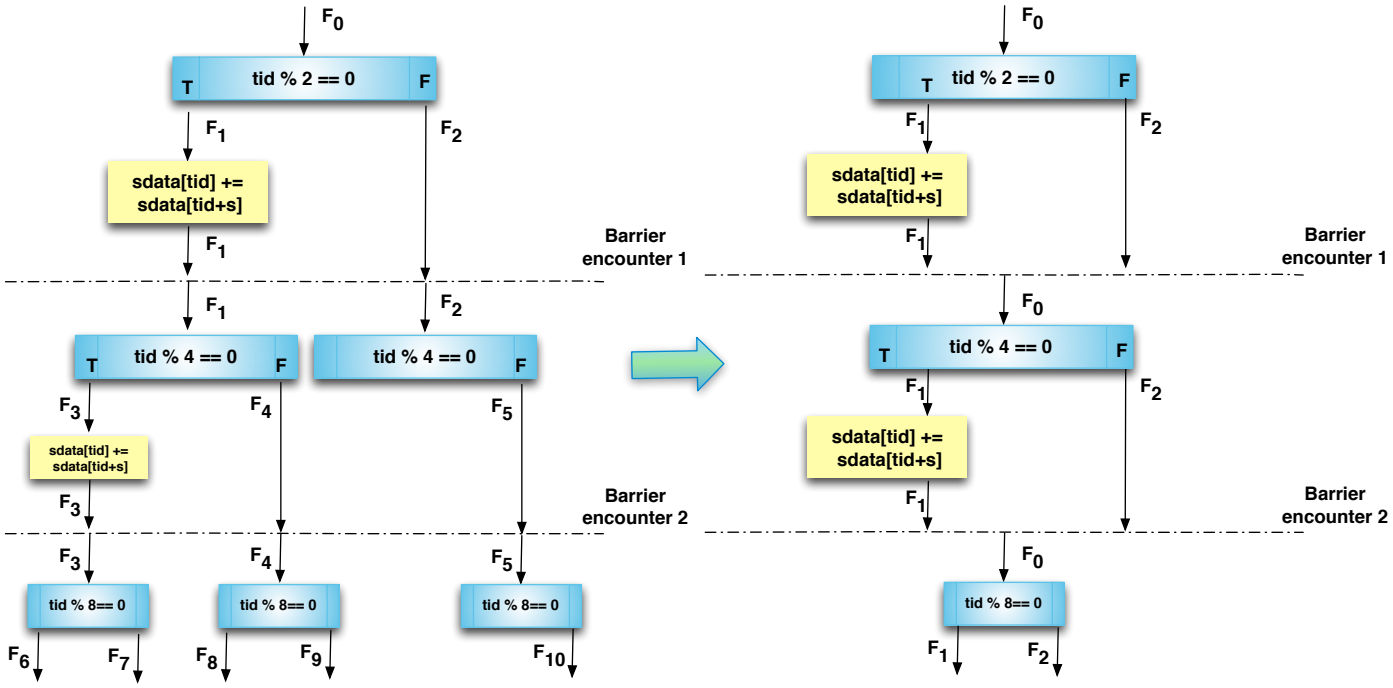

Fig. 4. Parametric flows of kernel reduction, and how flows are combined.

condition c_i partitions the threads into two flows. In general, the number of flows can grow exponentially. Here are two facts about the number of flows:

1) There is only one flow if the threads are non-divergent.
2) If all conditions depend on only the thread id, the number of threads is an upper bound of the number of distinct flows (this number can indeed be quite large).

Now we describe race checking during parametric execution. For barrier interval, one can follow a canonical (sequential [3]) schedule. That is, we symbolically execute one thread from one barrier to the other, then switch to the other thread and do the same, etc. The fact that this single sequential schedule [3] is sufficient is argued in [1], [15], [4], [17]. Both GKLEE$_p$ and SESA carry this sequential schedule using *one symbolic thread*. The resulting race history is cloned and instantiated over two thread IDs t_1 and t_2 with $t_1 \neq t_2$. This idea is applied within each *parametric flow* described below.

Details of Parameterized Flow Evolution: We begin one parametric thread that represents *all the threads* that are situated in one flow. During the execution, a flow may be split into multiple flows, each of which represents a group of threads. Regardless, each flow is executed as per the aforesaid canonical schedule. When the final thread of a barrier interval has finished running, the race history of the parametric thread is cloned, and race checking is carried out.

Example. Figure 4 show the parametric flow tree of the reduction kernel. The initial flow represents all $bdim.x$ threads. Consider the first loop iteration where $s = 1$. At line 3, the threads are divided into two groups upon condition $tid\%2 = 0$: one with thread ids $\{0, 2, 4, \dots\}$, and the other one with $\{1, 3, 5, \dots\}$. Accordingly, SESA creates two flows: flow F_1 with flow condition $tid\%2 = 0$, and flow F_2 with $tid\%2 \neq 0$. Then, SESA can execute F_1, producing read set $\{tid\%2 = 0\ ?\ sdata[tid+1], tid\%2 = 0\ ?\ sdata[tid]\}$ and write set $\{tid\%2 = 0\ ?\ sdata[tid]\}$. When this flow reaches the barrier, flow F_2 is scheduled to execute. Since F_2 contains no computation before the barrier, its read set and write set are empty. Now all the flows for the first barrier interval reach the barrier. We union the read-sets of F_1 and F_2 to produce the barrier interval read set: $\{tid\%2 = 0\ ?\ sdata[tid+1], tid\%2 = 0\ ?\ sdata[tid]\}$. Similarly we obtain the barrier interval write-set: $\{tid\%2 = 0\ ?\ sdata[tid]\}$. Then we instantiate these barrier interval sets with two symbolic threads t_1 and t_2 for race checking (for simplicity, we do not show the extra assumption $t_1, t_2 < bdim.x$ here). Since the solver returns "unsat", no race is found.

WW race: $t_1 \neq t_2 \ \wedge\ t_1\%2 = 0 \ \wedge\ t_2\%2 = 0 \ \wedge\ t_1 = t_2$
RW race: $t_1 \neq t_2 \ \wedge\ t_1\%2 = 0 \ \wedge\ t_2\%2 = 0 \ \wedge\ (t_1 + 1 = t_2 \ \vee\ t_1 = t_2)$

Suppose flow F_1 is executed first in the next barrier interval. Again two new flows are generated upon condition $tid\%4 = 0$. The leftmost flow F_3 has flow condition $tid\%2 = 0 \wedge tid\%4 = 0$, which is simplified to $tid\%4 = 0$. For flow F_4 with condition $tid\%2 \neq 0$, since $tid\%4 = 0$ conflicts with $tid\%2 \neq 0$, *i.e.* $tid\%2 \neq 0 \Rightarrow tid\%4 \neq 0$, SESA keeps only one flow with condition $tid\%2 \neq 0$. Finally we have 5 flows at barrier 2. Each flow represents a group of threads, *e.g.* F_6 represents threads $\{0, 4, 8, \dots\}$. Then we can obtain the barrier interval read/write sets by uniting the flows' read/write sets instantiate them with two threads, and check address overlapping for possible races. In the implementation, when two flows are generated, we can reuse the current flow for one of the generated flows so as to

reduce the flow cloning cost. For example, we can reuse F_0 for F_1, F_3 and F_6, and F_2 for F_5 and F_{10}.

Since m conditions over thread ids and symbolic inputs can result in $O(2^m)$ flows, we can utilize static data-flow information to combine the flows by (1) keeping $sdata$ value in "then" path, (2) keeping the value of s, *i.e.* $s = 1$, and (3) emptying the flow condition (since $(tid\%2 = 0 \vee tid\%2 \neq 0) = \text{true}$). Similarly at barrier 2 we have only 1 flow after flow-combining.

Warp Execution. SESA also supports warp execution as per CUDA's SIMD model. The threads within a warp are executed in a lock-step manner: two intra-warp threads can race only if they simultaneously write to the same shared variable at the same instruction. In case of divergence, we execute the two sides sequentially and merge them at the first convergence point (*e.g.* the nearest common post-dominator). When a conditional instruction $c\ ?\ is_1\ :\ is_2$ is encountered, SESA forks two new flows F_1 and F_2 representing the two branches of the condition, and subsequent executions will start from each one. At F_1, after is_1 is executed and the convergence point is reached, SESA executes is_2 immediately. That is, flow F_1 executes $c\ ?\ is_1$ followed by $\neg c\ ?\ is_2$. Similarly, flow F_2 executes $\neg c\ ?\ is_1$ followed by $c\ ?\ is_2$. For example, when the threads diverge on the condition at line 1 in the Generic Example of Figure 1, the order "$F_1; F_2$" produces $v = b$ while the order "$F_2; F_1$" produces $v = a$. The executor has to enumerate both orders to be complete (unless data-flow analysis helps eliminate one).

a) Soundess and Completeness: Since we use only one parametric thread t_i to model all the n threads, the soundness and completeness of our method depends on whether t_i can accurately simulate how the state is accessed by n threads, *e.g.* whether t_i can parametrically model the read and write by all the threads. We consider various cases from t_i's perspective:

1) For an access (read or write) involving no shared data, we can always obtain the accurate value of this access. That is, t_i is parametric.
2) If t_i reads the shared data written only by itself, then this read obtains the right value as in a normal single thread execution. Again, t_i is parametric. This is the case where each thread reads its own portion of the shared data.
3) If t_i reads the shared data written by other threads, then the value depends on how other threads write the data. We call the such an write *global SIMD write*.

In Generic Example, no SIMD writes occur. Hence all reads obtain the right values, and our parametric checking is accurate. In Reduction kernel, line 4 contains an SIMD write to shared variable $sdata$ such that each thread updates its own portion of the data. The next loop iteration reads this variable, whose value depends on the previous SIMD write. To study such accesses, we introduce the concept of an access being *resolvable* (whether we can estimate such read values accurately). By definition, an access $c\ ?\ v$ is resolvable if both c and v do not contain global SIMD writes.

Clearly, all reads in a computation are resolvable if this computation contains no global SIMD writes. A read over an SIMD write may be still resolvable by analyzing the relation of their addresses, *e.g.* our prior work [16] uses SMT solving to reason about resolvable reads. Currently, GKLEE resolves only the global reads and writes performed by the same (parametric) thread.

In general, the v part of an access $c\ ?\ v$ involves no global SIMD writes (*i.e.*, in such a write to a global variable, multiple threads contribute to the variable's value). For example, in the Reduction kernel, the SIMD write on $sdata$ at line 4 does not appear in the read set or write set for race checking. However, the c part is not resolvable in some kernels, *e.g.* the conditions at lines 6 and 10 in the Bitonic kernel introduce global SIMD writes into the read set and write set, and the reads pertaining to these writes are currently not resolvable with our one parametric thread model.

Proposition. If each access $c\ ?\ v$ in the read set or the write set is resolvable, *i.e.*, c and v do not contain global SIMD writes, then our parametric checker is sound and complete.

If an address expression involved in shared/global memory updates are unresolvable, then parametric checking may be unsound and incomplete. To avoid omissions, we can "havoc" (set to a fresh symbolic value) the value of a read over global SIMD update. (Our current release of SESA does not have this facility, yet.) To warrant soundness, we may use global invariants as in [15], [8], which often require manual effort. In our results section §VI, we indicate whether the race checking of a kernel involves unresolved SIMD writes, and indicate whether soundess and completeness are affected if such writes exist.

V. Taint Analysis

We now describe our taint analysis. It utilizes multiple LLVM passes: inlining, use-def, live-var, pointer-alias; then annotates LLVM instructions with live-vars relevant for race-checking. Passes are designed CUDA-feature-aware.

In addition to built-in variables (`bid.{x,y,z}`, `tid.{x,y,z}`), a kernel takes inputs from the CPU. The analysis consists of three LLVM passes, that marks how variables flow into sensitive sinks such as race related accesses:

1) Inline function calls within a kernel.
2) Determine data inputs and intermediate variables that flow into relevant sinks, employing LLVM alias analysis to handle memory accesses.
3) Annotate LLVM `br` and `switch` instructions to assist the dynamic flow removal and merging.

The second pass is the most important one since it calculates which inputs and intermediate variables flow into sensitive sinks. A traditional method is to start from all sinks and calculate the relevant variables by exploring the control flow graph backwards. We however make use of LLVM's in-built facilities and perform a forward pass to find out this set. Basically, this pass answers: *what variables will be used by the sinks from a given program point?* If the point is at the kernel entry, then we can obtain all those inputs that should be made symbolic. For flow merging, the point is at barrier statements or warp convergence places (for divergent warps). We need not track other program points.

$v = \texttt{alloc}\ n\ \tau$	$T(v) = v$ is live ? $\{v\} : \{\}$
$v = \texttt{getelptr}\ v_1, v_2\ ...\ v_n$	$T(v) = T(v_1) \cup T(v_2) \cup \cdots \cup T(v_n)$
$v = \texttt{binop}\ v_1,\ v_2$	$T(v) = T(v_1) \cup T(v_2)$
$v = \texttt{load}\ v_1$	$T(v) = T_\mu(v_1)$
$\texttt{store}\ v_1, v_2$	$T_\mu(v_1) = T(v_2)$
$v = \texttt{cmp}\ bop, v_1,\ v_2$	$T(v) = T(v_1) \cup T(v_2)$
$v = \texttt{cast}\ v_1$	$T(v) = T(v_1)$
$v = \texttt{phi}\ [l_1, v_1], [l_2, v_2]$	$T(v) = T(v_1) \cup T(v_2)$

Fig. 5. Taint propagation rules

This pass is extended from LLVM's use-def analysis. Roughly, it (i) identifies a set of live variables ("live" in terms of traditional use-def analysis) at a program point, (ii) propagates these variables along the control flow, and (iii) when a variable appears in a sink (*e.g.* in the address of a shared memory access), marks the variable as tainted. Additionally, we need to process the cases where sinks are control-dependent on active variables. Due to the existence of loops in the control-flow graph (CFG), the calculation iterates until a fixed-point is reached.

We check whether the live variables can flow into the addresses of shared memory accesses. We consider two cases: (1) the address $addr$ is data dependent on a variable v such that v appears in $addr$; and (2) $addr$ is control dependent on v such that v appears in the flow condition of the access. For the second case, we maintain flow conditions during the analysis.

Each variable v is associated with a live variable set (LVS), that records the live variables used by v. For each new instruction, we apply the taint propagation rules of Figure 5, where we use T as a short hand for LVS. One complication is about memory loads and stores. We introduce a notation $T_\mu[v]$ to represent the LVS of a variable whose memory address is v. We maintain a memory model to compute T_μ, and use LLVM's pointer alias analysis to resolve memory accesses.

For illustration, consider the following C code (for succinctness we absort the getelptr instructions into load and store). Here live variable v_1 resides at register $\%1$. Upon the store instruction, we maintain in T_μ that $T_\mu(A[1]) = \{v_1\}$. This LVS is propagated to register variable $\%2$ such that $T(\%2) = \{v_1\}$. The second load instruction results in an empty LVS for $\%3$. Finally the add instruction makes $T(\%4) = \{v_1\} \cup \{\} = \{v_1\}$.

```
  C code                  LLVM code
char A[10];             store A 1 %1
A[1] = v1;              %2 = load A 1
char c = A[1]+A[0];     %3 = load A 0
                        %4 = add %2 %3
```

In the implementation, we maintain a CFG for the inlined kernel. We follow the CFG to examine each instruction and update the LVS of the target variable. Each instruction will be visited at least once. For an instruction, if its LVS is updated, the new LVS will be propagated along the control flow. If the LVS is unchanged, then no propagation will be made. The entire analysis stops when no more propagation is needed (*i.e.* the LVS of each variable will not change anymore, thus a fixed point is reached). This is similar to the live variable calculation in compiler construction.

Example 1. We show below the LVS sets for the variables in the Generic example. Global inputs a and b are propagated to the LVS sets of local variable v and memory address $A[w]$. Since w's LVS is empty, all inputs can be made concrete.

Line 1:	T(v) = {a}
Line 2:	T(v) = {b}
Line 3:	T(v) = {a,b}, T(u) = {tid}
Line 4:	T_μ(A[w]) = {a,b}
Finally:	TaintSet = {}

Moreover, it is safe to combine the flows for the branches at lines 1 and 3. In our implementation, we instrument the LLVM instructions by adding a flag "skip" to the "else" parts of these branches. This flag tells the symbolic executor not to fork a flow. When the part contains executable code, this flag allows the executor to abandon the flow after executing this code. Since the remaining flow is a merge of the two original flows, its flow condition does not contain the branch condition.

Example 2. Consider the loop in the reduction kernel, which takes three inputs including $sdata$. We show below its bytecode (for better readability we simplify the byte-code including ignoring datatypes, e.g., "float"). Loop index s is an intermediate variable residing at register $\%1$. There are many more intermediate variables stored in the registers, *e.g.* $\%5$ stores $tid\,\%\,(2s)$ (at line 3 in the source code) and $\%7$ stores $v_2 = tid + s$ (at line 4).

```
loop:
    %2 = cmp lt %1 bdim.x       ; s < bdim.x?
    br %2 body end.for          ; branch
body:
    %3 = phi [loop,1] [if.end,%9] ; s's value
    %4 = mul 2 %1                 ; 2 * s
    %5 = mod tid %2               ; tid % (2*s)
    %6 = cmp eq %5 0           ; tid % (2*s) == 0?
    br %6 if.then if.else      ; branch
if.then:
    %7 = add tid %3            ; tid + s
    %8 = load sdata %7         ; read sdata[tid+s]
    store sdata tid %8         ; write sdata[tid]
    br if.end                  ; jump to if.end
if.else:
    br if.end
if.end:
    %9 = mul %3 2              ; s *= 2
    call __syncthreads         ; barrier
    br loop                    ; continue the loop
end.for:
```

There are two program points of interest: $pnt1$ at the kernel entry, and $pnt2$ at the barrier at line 5. Consider $pnt1$, where the live variables are the inputs including $sdata$. The analyzer's task is to, given a program point, determine which live variables will flow into the sinks. For $pnt1$, we check whether the live variables (*i.e.* the inputs) can flow into the two shared memory accesses at line 4. The calculation is done by investigating each instruction and propagating the variables according to the rules shown in Figure 5.

Initially, the LVS of each variable is empty. At the first instruction, $\%1$'s LVS is empty since $\%1$ stores local variable s (hence involving no global inputs). Instruction "%2 = cmp lt %1 bdim.x" propagates %1's LVS to %2, whose LVS is also empty. The "phi" instruction propagates %9's LVS to %3, and so on. At line 4, we check the addresses of $sdata[\%7]$

and $sdata[\%8]$ for possible races, hence the variables in %7's LVS and %8's LVS need to be marked as tainted variables. Since these two sets are empty, no variable will be marked, indicating that no inputs should be made symbolic.

The computation goes on when the control jumps back to label "loop". Since processing the instruction does not change the LVS, propagation stops here, reaching a fixpoint, concluding that all inputs can be concrete.

The analysis results can also be used to remove parametric flows. At program point $pnt2$ at the barrier, variables $\%3, \%4, \%5, \%7$ and $\%8$ are determined to be not related to the sinks (*i.e.* addresses in shared accesses), variable $\%2$ is not live, and variables $\%1$ and $\%9$ have the same values for both flows, hence only one flow is needed to be explored during symbolic execution. Basically we check whether the writes in a BI (barrier interval) will be used in subsequent BIs. A live variable not updated in any flow within the BI can be ignored since its value will be the same in all flows.

We show below more information, where the data store gives the variable values. Note that, after flow merging, the path condition of the merged flow is the union of those of the two original flows. Based on the LLVM Meta-data encoded by static analyzer, symbolic executor merely picks the left flow.

	left flow	right flow
Path Cond :	$tid\,\%\,2 = 0$	$tid\,\%\,2 \neq 0$
Data Store :	$\%1 = 0$	$\%1 = 0$
	$\%9 = 2$	$\%9 = 2$
	$\%7 = tid + 1$	$\%7 =$ undef
	$\%8 = sdata[\%7]$	$\%8 =$ undef
	...	...

VI. Experimental Results

We run SESA on the kernels in CUDA SDK, the Parboil library [23], and the LonestarGPU benchmark [19]. All experiments are performed on a machine with Intel(R) Xeon(R) CPU @ 2.40GHz and 12GB memory. We perform extensive comparisons of SESA and a start-of-the-art GPU testing tool GKLEE$_p$ [18]. The benchmarks are available at https://sites.google.com/site/sesabench/.

A. CUDA SDK

Table I shows results on a few CUDA SDK 5.5 kernels that have no thread divergence (hence both SESA and GKLEE$_p$ explore one flow). For example, for kernel vectorAdd, SESA determines that none of the 4 inputs need to be symbolic, while a typical GKLEE$_p$ user sets 2 symbolic inputs (the other two are related to the concrete thread number). SESA finishes the execution for $50,176$ threads in 0.8 second while GKLEE$_p$ needs 3.8 seconds. This shows the advantages of reduced symbolic inputs, even for small examples. The case of kernel matrixMul is similar.

For some benchmarks, while having excessive symbolic inputs does not affect performance (because most steps such as *e.g.* memory block matching are unrelated to inputs), it causes other problems. For example, SESA detects no symbolic inputs for kernel scalarProd, while GKLEE$_p$ sets 2 symbolic inputs and crashes without giving any useful result.

Kernels	# Threads	GKLEE$_p$		SESA	
		# Inputs	Time	# Inputs	Time
vectorAdd	50,176	2/4	3.8	0/4	0.8
Clock	16,384	2/3	4.7	0/3	4.6
matrixMul	204,800	2/5	95.25	0/5	9.8
Scan Short	4,096	1/4	179.3	0/4	181.9
Scan Large	4,096	1/4	107.1	0/4	109.1
scalarProd	32,768	2/5	Crash	0/5	77.6
Transpose	262,144	1/4	132.0	0/4	128.4
fastWalsh	1,024	2/4	54.8	0/4	44.5

TABLE I
CUDA SDK 5.5 NON-DIVERGENT KERNEL RESULTS. NO RACES ARE FOUND; TIMING RESULTS (SEC.) ARE THE AVERAGE OVER THREE RUNS.

This explains another important issue in GPU symbolic testing: *constraints on the symbolic inputs must be set properly*. By avoiding excessive symbolic inputs, this problem seems ameliorated[1]. In summary, SESA is able to detect all races and errors found by GKLEE and GKLEE$_p$ on SDK kernels. For example, SESA spends 2 seconds to find a real WW race in histogram64 in SDK 2.0 while both GKLEE$_p$ and GKLEE spend more than 20 seconds.

B. Performance improvement with flow optimizations

Table II presents the SESA's advantage over the GKLEE$_p$ for kernels whose execution produces many flows. SESA's performance improvement becomes more significant when the number of threads becomes larger. For example, with 16 threads in mergeSort kernel, GKLEE$_p$ explores 17 flows in 75.4 seconds, while SESA explores only 1 flow and finishes the execution in 3 seconds thanks to removal of duplicate flows and merging of flows at thread convergence points. For kernels bitonic and wordsearch, GKLEE$_p$ times out in 1 hour even for 16 threads, while SESA can scale to over 256 threads. SESA produces 3 flows for kernel blelloch with 64 threads, and SESA finishes the checking in 46.9 seconds while GKLEE$_p$ times out. Here we count only the execution time since the taint analysis time is negligible. For the buggy stream compaction kernel, SESA spends less than half of the time than GKLEE$_p$ to locate the bug. Note that these programs are highly divergent with large state spaces, hence may take both GKLEE$_p$ and SESA quite some time to analyze. Without flow optimizations it is very hard to check them for even small configurations such as 16 threads.

SESA may suffer from false alarms or omissions caused by unresolvable reads described in §IV-B. In our experiments, we spent effort identifying these unresolvable reads manually. Whenever feasible, we also compared our manual results against GKLEE [17], the predecessor of GKLEE$_p$ and SESA. For example, stream compaction and n stream compaction are `un-resolvable` kernels.

Table III presents the results for the LonestartGPU benchmark, which consists of irregular kernels with multiple flows. The columns with (`Conc.`) use pure concrete inputs (the thread ids are still symbolic), while the columns with (`Sym.`) apply the symbolic inputs identified by taint analyzer, with

[1]Far easier to find legal input instances than general input constraints.

Kernel Name	RSLV?	RR/OM	#T = 16		#T = 32		#T = 64		#T = 128		#T = 256	
			GKLEE$_p$	SESA	GKLEE$_p$	SESA	GKLEE$_p$	SESA	GKLEE$_p$	SESA	GKLEE$_p$	SESA
bitonic 2.0	Y	–/–	T.O.	1 (5.9)	T.O.	1 (12.1)	T.O.	1 (30.4)	T.O.	1 (79.7)	T.O.	1 (248.2)
wordsearch	Y	–/–	T.O.	1 (1.6)	T.O.	1 (4.5)	T.O.	1 (15.7)	T.O.	1 (72.1)	T.O.	1(474.2)
bitonic 4.3	Y	–/–	T.O.	1 (29.3)	T.O.	1 (105.6)	T.O.	1 (295.4)	T.O.	1 (504.5)	T.O.	1 (1,215.6)
mergeSort 4.3	Y	–/–	17 (75.4)	1 (3.0)	38 (442.5)	1 (4.5)	78 (2,174.4)	1 (7.3)	T.O.	1 (9.2)	T.O.	1 (14.1)
stream compaction*	N	RR/–	32 (9.1)	5 (6.1)	33 (16.2)	6 (11.9)	65 (36.2)	7 (21.6)	129 (81.7)	8 (34.8)	257 (181.5)	9 (50.6)
n stream compaction*	N	–/–	34 (58.7)	16 (8.3)	35 (224.7)	33 (120.6)	67 (541.5)	65 (301.9)	131 (1497.8)	129 (813.9)	259 (1596.8)	257 (936.6)
blelloch	Y	–/–	93 (1,496.1)	3 (9.7)	T.O.	3 (20.3)	T.O.	3 (46.9)	T.O.	3 (114.5)	T.O.	3 (629.0)
brentkung	Y	–/–	65 (1,476.7)	3 (9.6)	T.O.	3 (24.2)	T.O.	3 (55.5)	T.O.	3 (140.7)	T.O.	3 (315.5)

TABLE II

COMPARISON OF SESA AND GKLEE$_p$. SESA IDENTIFIES A SUBSET OF INPUTS TO BE SYMBOLIZED. EACH RESULT CELL IS OF THE FORM 'Num-Flows' ('elapsed-time-in-secs.') AND TIME-OUTS ARE AT 3,600 SECONDS, 'NUM-FLOWS' REFERS TO THE MAXIMUM OF FLOWS THAT OCCUR DURING THE EXECUTION. COLUMN `RSLV?` REPRESENTS IF THE KERNEL IS *resolvable*. COLUMN `RR/OM` DENOTES IF RACES WERE REPORTED BY SESA OR OMISSIONS OCCUR. A "-" MEANS THAT THESE OUTCOMES DID NOT HAPPEN. stream compaction SUFFERS FROM A FALSE OUT-OF-BOUND ERROR AND WRITE-WRITE RACE (MANUALLY CONFIRMED). FOR stream compaction AND n stream compaction KERNELS, 'NUM-FLOWS' REFERS TO THE NUMBER OF THE EXECUTION PATHS. THE LAST FOUR BENCHMARKS ARE FROM [8].

Kernel Name	RSLV?	RR/OM	#Threads	GKLEE$_p$ (Conc.)	SESA (Conc.)	GKLEE$_p$ (Sym.)		SESA (Sym.)	
				# Flow	# Flow	# Flow	Errors	# Flow	Errors
bfs_ls (BFS)	N	RR/–	256	9 (37.9)	9 (36.8)	T.O.	?	2 (0.9)	?
bfs_atomic (BFS)	N	RR/–	1,024	7 (7.9)	7 (7.9)	T.O.	R/W*	T.O.	R/W*
bfs_worklistw (BFS)	N	RR/–	256	4 (140.6)	4 (147.6)	19 (40.5)	?	2 (20.2)	?
bfs_worklista (BFS)	N	RR/–	1,024	5 (2.3)	5 (2.3)	19 (8.5)	?	3 (0.6)	?
BoundingBox (BH)	Y	RR/–	6,144	16 (106.7)	2 (50.1)	16 (103.9)	R/W*	2 (46.4)	R/W*
sssp_ls (SSSP)*	N	RR/–	1,024	6 (19.9)	6 (19.9)	310 (198.1)	W/W	2 (2.5)	W/W
sssp_worklistn (SSSP)*	N	RR/–	1,024	5 (318.2)	5 (327.3)	390 (617.5)	W/W	2 (21.4)	W/W

TABLE III

COMPARISON OF SESA AND GKLEE$_p$ FOR LONESTARGPU BENCHMARK WHICH IS A COLLECTION OF APPLICATIONS THAT EXHIBIT IRREGULAR BEHAVIOR. EACH `# Flow` CELL IS OF THE FORM 'Num-Flows' ('elapsed-time-in-secs.'). WE SET 3,600 SECONDS FOR T.O. AND EACH `# Flow` REFERS TO THE MAXIMUM NUMBER OF FLOWS THAT OCCUR DURING THE EXECUTION. COLUMN `RR/OM` DENOTES IF RACES WERE REPORTED BY SESA OR OMISSIONS OCCUR. IF AN R/W OR A W/W RACE IS LISTED UNDER "ERRORS," THEN THAT MEANS THAT THE RACE REPORT WAS MANUALLY CONFIRMED AS BEING A GENUINE RACE. WHENEVER A KERNEL SUFFER FROM AN `out-of-bound` ERROR, WE DISABLED THIS CHECK IN ORDER TO BE ABLE TO COLLECT THE RUNTIME. WE USE `# execution path` RATHER THAN `# Flow` FOR KERNELS MARKED WITH $*$ SYMBOLS. IN BFS_ATOMIC AND BOUNDINGBOX KERNELS, R/W RACE STEMS FROM "DON'T-CARE NON-DET." (ENSURES ONE BODY INSERTED IN EACH POSITION). ? DENOTES THOSE ERRORS ARE UNDER INVESTIGATION.

those flowing into loop bounds being excluded (as mentioned in §III-C). For example, consider kernel bfs_ls. When symbolic inputs are used, GKLEE$_p$ times out in 1 hour, and SESA finishes the checking in 0.9 seconds with only two flows.

Note that, when the symbolic inputs are not constrained with proper assumptions, many memory out-of-bound (OOB) errors may be produced, while these errors are not relative to races. GKLEE$_p$ suffers seriously from this problem, while SESA is more resilient with only a portion of inputs being symbolic. To make the comparison fair, we disable the OOB checking and OOB related state spawning in both GKLEE$_p$ and SESA. This often reduces the total execution time (*e.g.* less than that with concrete inputs). We also show intuitively in Figure 6 the speedup, *e.g.* SESA is more than $3,000$x faster than GKLEE$_p$ for kernel bfs-ls with symbolic inputs. For better readability, this figure does not show the overflowing values in an exact way, *e.g.* the red bar of bfs-ls which has over $3,000$x speed-up. Similarly, Figure 7 shows some speedups for kernels in Table II. Here SESA can outperform GKLEE$_p$ by 1-3 orders of magnitude.

C. Parboil

We have run SESA on the entire Parboil benchmark [23]. Table IV lists the results on 10 kernels, many of which contain issues revealed by SESA. For example, for histo_prescan_kernel, SESA infers that 1 out of the 3 inputs should be made symbolic. SESA can scale to $32,768$ threads. It explores two flows and detects a real RW race detailed below.

Figure 8 explains the read-write race uncovered in histo_prescan (witness was automatically generated). The write access in `SUM(stride)` is performed by thread $\langle 17,0,0\rangle$ while the read access in `SUM(16)` is by thread

Bench Name	Kernels	# Threads	# Inputs (SYM)	Errors	# Flow
bfs	BFS_in_GPU_kernel	512	4/11	W/W (Benign)	1
cutcp	cutoff_potential_lattice6overlap	15,488	1/8	W/W (Benign)	1
histo	histo_prescan_kernel	32,768	1/3	R/W	1
histo	histo_intermediates_kernel	32,370	0/5	–	1
histo	histo_main_kernel	21,504	2/9	–	1
histo	histo_final_kernel	21,504	0/8	OOB	1
mri-gridding	binning_kernel	16,896	⟨2,1⟩/7	R/W	1
mri-gridding	reorder_kernel	16,896	⟨1,0⟩/4	–	1
spmv	spmv_jds	1,152	⟨2,0⟩/7	W/W (Benign)	1
stencil	block2D_hybrid_coarsen_x	8,192	0/7	–	T.O.

TABLE IV
PARBOIL RESULTS (T.O. IS 2 HOURS). FOR mri-gridding AND spmv, THE FIRST MEMBER OF THE ⟨⟩ PAIR DENOTES THE NUMBER OF INPUTS INFERRED BY SESA TO BE MADE SYMBOLIC, WHILE THE SECOND REPRESENTS THE ACTUAL NUMBER OF SYMBOLIZED INPUTS NEEDED (REVEALED BY MANUAL ANALYSIS).

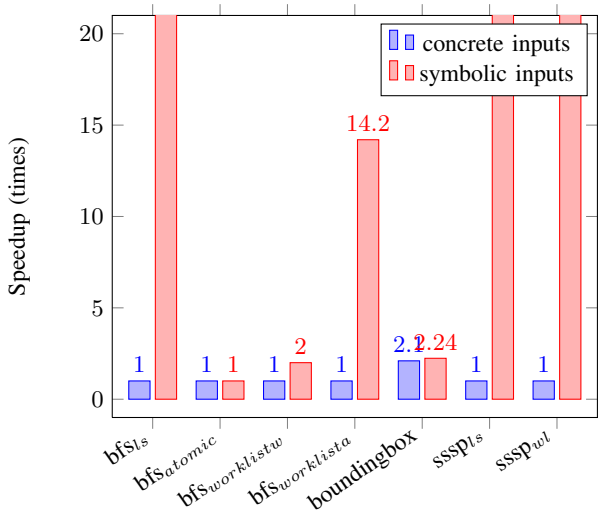

Fig. 6. Speedup of SESA over GKLEE$_p$ on LoneStarGPU.

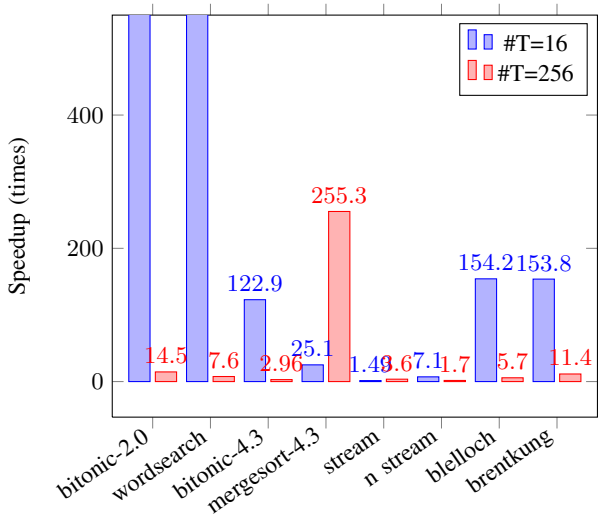

Fig. 7. Speedup of SESA over GKLEE$_p$ on kernels in Table II.

$\langle 1,0,0\rangle$; these conflict, leading to the race.

Figure 9 illustrates the out-of-bound access caught in the histo_final kernel. The OOB access occurs in the 47th iteration of the loop, the initial value of i is $tid.x + bid.x * 512$ for symbolic tid and bid. In addition, the size of memory region the pointer global_histo refers to is $8,159,232$. Then the OOB is modeled as a constraint $(tid.x + bid.x \times 512 + 47 \times 42 \times 512) * 8 < 8159230$, SESA solves this formula and identifies a thread with the configuration: $bid = \langle 24,0,0\rangle \wedge tid = \langle 0,0,0\rangle$ that could incur the OOB access. Note that it is not easy to use manual or random testing to come up with such witnesses.

Figure 10 illustrates the inter-block read-write race caught in the binning_kernel. This race is uncovered when the memory region `sample_g` is set symbolic, and the size of the memory region is $404,160$. SESA exposes that the thread 1 with $bid = \langle 32,0,0\rangle \wedge tid = \langle 64,0,0\rangle$ and the thread 2 with $bid = \langle 0,0,0\rangle \wedge tid = \langle 0,0,0\rangle$ are involved in the race, thread 1 reads the `binCount_g[binIdx]`, and thread 2 writes the same element with the `atomicAdd` instruction where `binIdx` is evaluated to be `0` because `pt` is symbolic.

These issues have not (to the best of our knowledge) been reported before by others; most of these bugs manifest only when the state space is analyzed sufficiently. For example, GKLEE$_p$ generates a huge number of flows and times-out before the bug in binning is reached. Flow-merging and tight symbolic input-set selection helps SESA reach the bug in affordable time budget.

VII. ADDITIONAL RELATED WORK, CONCLUSIONS

In [6], a dynamic race detection technique is proposed. The work in [27] brings in static analysis to locate possible candidates for further dynamic analysis. These methods do not employ symbolic execution nor parametric flows.

PUG [15] employs symbolic static analysis on individual kernels (not whole-program), requires user annotations of allowed inputs, loop invariants, and has restrictions on allowed loops. GPUVerify [4], [8] is also based on symbolic static analysis and a precisely CUDA operational semantics for predicated forms of GPU programs. It employs loop invariants to avoid loop unrolling, and barrier invariants [8] to reduce false alarms related to shared memory accesses. The main drawback of these tools is the difficulty of manual invariant

```
__global__ void histo_prescan_kernel(...) {
...
#define SUM(stride__)
if(threadIdx.x < stride__){
    Avg[threadIdx.x] += Avg[threadIdx.x+stride__];
    StdDev[threadIdx.x] += StdDev[threadIdx.x+stride__];
}
#if (PRESCAN_THREADS >= 32)
    for (int stride = PRESCAN_THREADS/2;
         stride >= 32; stride = stride >> 1) {
        __syncthreads();
        SUM(stride);
    }
#endif
#if (PRESCAN_THREADS >= 16)
    SUM(16);
#endif
}
```

Fig. 8. The Read-Write race in histo prescan kernel

```
// gridDim.x: 42, blockDim.x: 512
__global__ void histo_final_kernel (...) {
  unsigned int start_offset =
    threadIdx.x + blockIdx.x * blockDim.x;
  for (unsigned int i = start_offset;
       i < size_low_histo/4;
       i += gridDim.x * blockDim.x) {
    // out of bound error found here
    ushort4 global_histo_data =
      ((ushort4*)global_histo)[i];
    ...
  }
}
```

Fig. 9. The OOB in histo final kernel

```
__global__ void binning_kernel (...) {
  unsigned int sampleIdx = blockIdx.x*blockDim.x
                           +threadIdx.x;
  ...
  if (sampleIdx < n){
    pt = sample_g[sampleIdx];
    binIdx = (unsigned int)(pt.kZ)*size_xy_c +
             (unsigned int)(pt.kY)*gridSize_c[0] +
             (unsigned int)(pt.kX);
    if (binCount_g[binIdx]<binsize){
      count = atomicAdd(binCount_g+binIdx, 1);
  }
}
```

Fig. 10. The read write race in binning kernel

discovery. These tools also do not operate on whole programs including the CPU code, which SESA does. In addition, being an execution based tool, SESA can more naturally detect programming errors such as out-of-bounds array accesses.

The GPU program testing tool in [14] reported a 'test amplification technique' that, similar to taint-based analysis, helps identify inputs that can be set concrete without losing coverage. The key novelty of SESA in this regard is that in addition to this leverage, static analysis also helps merge parameterized flows. The tools GKLEE [17], KLEE-CL [9] and KLEE-FP [10] do not support parametric flows. The tool GKLEE$_p$ [18] does not have the ability to merge flows based on static analysis, and relies on users to select those inputs that are set to concrete values (which risks either omitted coverage or un-necessary symbolic execution).

The list of examples handled in SESA far exceeds—in size, variety, and practical importance—those handled by the above mentioned tools.

Conclusions. In this paper, we present SESA, a symbolic execution based tool for detecting data races in practical CUDA programs. SESA uses parametric execution to efficiently handle large numbers of threads. It employs static analysis to automatically identify inputs that can be safely set to concrete values, often identifying most of the inputs that can be so set without losing coverage. A key novelty of SESA is that its static analysis also informs parametric flow merging, a key technique that avoids the creation of un-necessary flows. It also provides a race-modulo analysis technique that is reminiscent of techniques developed for CPUs (e.g., [5], [11], [12], [7]), but adapts this thinking to GPU (SIMD) codes.

In this paper, we thoroughly evaluate SESA on large benchmark suits such as Parboil and Lonestar. During these experiments, the tool *automatically* found several genuine bugs, including out of bound array accesses and data races. It also found a few races later deemed to be benign—and in the process forced useful code walk-through and code understanding. All error reports are accompanied by concrete witnesses (input values and thread IDs involved in the issue). These results, together with the enriched CUDA subset that SESA handles, positions itself as (to the best of our knowledge) the foremost of formally based GPU correctness analysis tools geared primarily toward race-checking.

ACKNOWLEDGEMENTS

Thanks to: (a) Mark Baranowski and Ian Briggs for testing SESA thoroughly and finding bugs. (b) Tyler Sorensen for discussions on warp-synchronous programming and many aspects of GPU programming. (c) Vinod Grover of Nvidia for numerous discussions. (d) Nvidia for supporting Peng Li's research during 2012 through an Nvidia fellowship, and to Fujitsu Labs of America for supporting Peng's work during 2014. This work was supported by NSF grants CCF 1346756, CCF 1302449, and ACI 1148127.

REFERENCES

[1] ADVE, S. V. *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, Madison, WI, USA, 1993. UMI Order No. GAX94-07354.

[2] Allinea DDT. http://www.allinea.com/products/ddt.

[3] ATTIYA, H., GUERRAOUI, R., HENDLER, D., KUZNETSOV, P., MICHAEL, M. M., AND VECHEV, M. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL* (2011), pp. 487–498.

[4] BETTS, A., CHONG, N., DONALDSON, A. F., QADEER, S., AND THOMSON, P. GPUVerify: a verifier for GPU kernels. In *OOPSLA* (2012).

[5] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. R. RWset: Attacking path explosion in constraint-based test generation. In *TACAS* (2008).

[6] BOYER, M., SKADRON, K., AND WEIMER, W. Automated dynamic analysis of CUDA programs. In *STMCS* (2008).

[7] BUGRARA, S., AND ENGLER, D. R. Redundant state detection for dynamic symbolic execution. In *USENIX ATC* (2013).

[8] CHONG, N., DONALDSON, A. F., KELLY, P. H., KETEMA, J., AND QADEER, S. Barrier invariants: A shared state abstraction for the analysis of data-dependent gpu kernels. In *OOPSLA* (2013).

[9] COLLINGBOURNE, P., CADAR, C., AND KELLY, P. Symbolic testing of OpenCL code. In *HVC* (2011).

[10] COLLINGBOURNE, P., CADAR, C., AND KELLY, P. H. Symbolic crosschecking of floating-point and simd code. In *EuroSys* (2011).
[11] HANSEN, T., SCHACHTE, P., AND SØNDERGAARD, H. State joining and splitting for the symbolic execution of binaries. In *Runtime Verification (RV)* (2009).
[12] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient state merging in symbolic execution. In *PLDI* (2012).
[13] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.
[14] LEUNG, A., GUPTA, M., AGARWAL, Y., GUPTA, R., JHALA, R., AND LERNER, S. Verifying GPU kernels by test amplification. In *PLDI* (2012).
[15] LI, G., AND GOPALAKRISHNAN, G. Scalable SMT-based verification of GPU kernel functions. In *SIGSOFT FSE* (2010).
[16] LI, G., AND GOPALAKRISHNAN, G. Parameterized Verification of GPU Kernel Programs. In *PLC Workshop (part of IPDPS)* (May 2012).
[17] LI, G., LI, P., SAWAGA, G., GOPALAKRISHNAN, G., GHOSH, I., AND RAJAN, S. P. GKLEE: Concolic verification and test generation for GPUs. In *PPoPP* (2012).
[18] LI, P., LI, G., AND GOPALAKRISHNAN, G. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in cuda programs. In *Supercomputing (SC)* (2012).
[19] Lonestargpu benchmark. http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu.
[20] NVIDIA. CUDA-MEMCHECK. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/cuda-memcheck.pdf.
[21] NVIDIA. CUDA-GDB, Jan. 2009. An extension to the GDB debugger for debugging CUDA kernels in the hardware.
[22] OpenCL. http://www.khronos.org/opencl.
[23] Parboil Benchmark. http://impact.crhc.illinois.edu/parboil.aspx.
[24] Satisfiability Modulo Theories Competition (SMT-COMP). http://www.smtcomp.org/2012.
[25] SORENSEN, T. Towards shared memory consistency models for GPUs. Section 5.3 of http://www.cs.utah.edu/fv/theses/tyler_bs.pdf.
[26] Discussions on exploiting warp semantics and consequences, as of 2007. Expert Nvidia Note: NVIDIA makes no guarantees on warp size. Also the use of volatiles in this context no longer guaranteed. https://devtalk.nvidia.com/default/topic/377816/.
[27] ZHENG, M., RAVI, V. T., QIN, F., AND AGRAWAL, G. GRace: A low-overhead mechanism for detecting data races in GPU programs. In *PPoPP* (2011).