GKLEE: Concolic Verification and Test Generation for GPUs

Guodong Li [*]
Fujitsu Labs of America, CA
gli@us.fujitsu.com

Peng Li Geof Sawaya
Ganesh Gopalakrishnan
School of Computing, University of Utah, UT
{peterlee,sawaya,ganesh}@cs.utah.edu

Indradeep Ghosh
Sreeranga P. Rajan
Fujitsu Labs of America, CA
{ighosh,sree.rajan}@us.fujitsu.com

Abstract

Programs written for GPUs often contain correctness errors such as races, deadlocks, or may compute the wrong result. Existing debugging tools often miss these errors because of their limited input-space and execution-space exploration. Existing tools based on conservative static analysis or conservative modeling of SIMD concurrency generate false alarms resulting in wasted bug-hunting. They also often do not target performance bugs (non-coalesced memory accesses, memory bank conflicts, and divergent warps). We provide a new framework called GKLEE that can analyze C++ GPU programs, locating the aforesaid correctness and performance bugs. For these programs, GKLEE can also automatically generate tests that provide high coverage. These tests serve as concrete witnesses for every reported bug. They can also be used for downstream debugging, for example to test the kernel on the actual hardware. We describe the architecture of GKLEE, its symbolic virtual machine model, and describe previously unknown bugs and performance issues that it detected on commercial SDK kernels. We describe GKLEE's test-case reduction heuristics, and the resulting scalability improvement for a given coverage target.

Keywords: GPU, CUDA, Parallelism, Symbolic Execution, Formal Verification, Automatic Test Generation, Virtual Machine

1. Introduction

Multicore CPUs and GPUs are making inroads into virtually all aspects of computing, from portable information appliances to supercomputers. Unfortunately, programming multicore systems to achieve high performance often requires many intricate optimizations involving memory bandwidth and the CPU/GPU occupancy. A majority of these optimizations are still being carried out manually. Given the sheer complexity these optimizations in the context of actual problems, designers routinely introduce correctness and performance bugs. Locating these bugs using today's commercial debuggers is always a 'hit-or-miss' affair: one has to be *lucky* in so many ways, including (i) picking the right test inputs, (ii) ability to observe of data corruption (and be able to reliably attribute it to races), (iii) whether the compiler optimization match programmer assumptions, and (iv) whether the platform masks bugs because of the specific thread/warp scheduling algorithms used. If the execution deadlocks, one has to manually reason out the root-cause.

Recent formal and semi-formal analysis based tools [1, 2, 3] have improved the situation in many ways. They, in effect, examine whole classes of inputs and executions, by resorting to symbolic analysis or static analysis methods. They also analyze abstract GPU models without making hardware-specific thread scheduling assumptions. These tools also have many drawbacks. The first problem with predominantly static analysis based approaches is *false alarms*. False alarms waste precious designer time and may dissuade them from using a tool. Another limitation of today's tools is that they do not help generate tests that achieve high code coverage. Such tests are important for unearthing compiler bugs or "unexpected" bugs that surface during hardware execution. Existing tools also do not cover one new data race category that we identify (we call it *warp-divergence race*). Compilation based approaches can, in many cases, eliminate the drudgery of GPU program optimization; however, their code transformation scripts are seldom separately formally verified.

We present a new tool framework called GKLEE for analyzing GPU programs with respect to important correctness and performance issues (the tool name coming from "GPU" and "KLEE [4]). GKLEE profits from KLEE's code base and philosophy of testing a given program using *concrete plus symbolic* ("concolic") execution. GKLEE is the first concolic verifier and test generator tailored for GPU programs. Concolic verifiers allow designers to declare certain input variables as 'symbolic' (the remaining inputs are concrete).

The execution of a program expression containing symbolic variables results in constraints amongst the program variables, including constraints due to conditionals, and explicit constraints (`assume` statements) on symbolic inputs. Conditionals are resolved by invoking decision procedures ("SMT solvers [5]") that find solutions for symbolic program inputs. This approach helps concolic verifiers do something beyond bug-hunting: they can automatically enumerate test inputs in a *demand-driven* manner. That is, if there is a control/branch decision that can be affected by some input, a concolic verifier can automatically compute and record the input value in a test which is valuable for downstream debugging. Recent experience shows that formal methods often have the biggest impact when they can compute tests automatically, exposing software defects and vulnerability [6, 7, 8].

The architecture of GKLEE is shown in Figure 1. It employs a C/C++ front-end based on LLVM-GCC (with our customized extensions for CUDA syntax) to parse CUDA programs. It supports the execution of both CPU code and GPU code. GKLEE employs a new approach to model the symbolic state (recording the execution status of a kernel) with respect to the CUDA memory model.

[*] Guodong Li started working on this project as a student at the University of Utah.

Contributions: Our main contribution is a symbolic virtual machine (VM) to model the execution of GPU programs on open inputs. We detail the construction and operation of this virtual machine, showing exactly how it elegantly integrates error-detection and analysis, while not generating false alarms or missing execution paths when generating concrete tests. This approach also allows one to effect scalability/coverage tradeoffs. The following features are integrated into our symbolic VM approach:

• GPU programs can suffer from several classes of insidious data races. GKLEE finds such races (sometimes even in well-tested GPU kernels).

• GKLEE detects and reports occurrences of divergent thread warps (branches inside SIMD paths), as these can degrade performance. In addition, GKLEE guarantees to find deadlocks caused by divergent warps in which two threads may encounter different sequences of barrier (`__syncthreads()`) calls.

• GKLEE's symbolic virtual machine can systematically generate concrete tests while also taking into account any input constraints the programmer may have expressed through `assume` statements.

• While tests generated by GKLEE guarantee high coverage, it may lead to test explosion. GKLEE employs powerful heuristics for reducing the number of tests. We evaluate these heuristics on a variety of examples and identify those heuristics that result in high coverage while still only generating fewer tests.

• We can automatically run GKLEE-generated tests on the actual hardware; one such experiment alerted us to the need for a new error-check type, which we have added to GKLEE: *has a volatile declaration been possibly forgotten?* This can help eliminate silent data corruption caused by reads that may pick up stale write values.

• We target two classes of memory access inefficiencies, namely non-coalesced global memory accesses and shared memory accesses that result in bank conflicts, and show how GKLEE can spot these inefficiencies, also "understanding" platform rules (*i.e.*, compute capability 1.x or 2.x). Some kernels originally thought free of these errors are actually not so.

• GKLEE's VM incorporates the CUDA memory model within its concolic execution framework, while (i) accurately modeling the SIMD concurrency of GPUs, (ii) avoiding interleaving enumeration through an approach based on race checking, and (iii) scaling to large code sizes.

• GKLEE handles many C++/CUDA features including: struct, class, template, pointer, inheritance, CUDA's variable and function derivatives, and CUDA specific functions.

• GKLEE's analysis occurs on LLVM byte-codes (also targeted by Fortran and Clang). Byte-code level analysis can help cover pertinent compiler-induced bugs in addition to supporting future work on other binary formats.

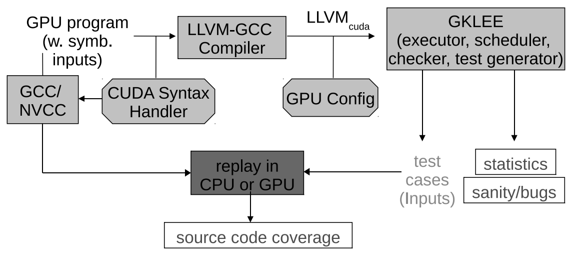

Figure 1. GKLEE's architecture.

Roadmap: § 2 explains the error-classes covered by GKLEE. § 3 presents GKLEE's concolic verification: state model, memory type inference, and concolic execution (§ 3.1) and error checking/analysis (§ 3.2). § 5 presents experimental results, covering issues pertaining to correctness checking/performance (§ 5.1) and test set generation/reduction (§ 5.2). § 6 presents related work and § 7 concludes.

2. Examples of our Analysis/Testing Goals

2.1 Basics of GPU Programs

GKLEE currently supports the CUDA [9] syntax (with OpenCL [10] to be addressed in future). A CUDA kernel is launched as an 1D or 2D *grid* of *thread blocks*. The total size of a 2D grid is `gridDim.x` × `gridDim.y`. Each block at location ⟨`blockIdx.x`, `blockIdx.y`⟩ 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*. Threads belonging to distinct blocks must use the much slower *global memory* to communicate, and may not synchronize using barriers. The values of `gridDim` and `blockDim` determines the *configuration* of the system, *e.g.* the sizes of the grid and each block. For a thread, `blockIdx` and `threadIdx` give its block index in the grid and its thread index in the block respectively. For brevity, we use $gdim$ to denote $gridDim$, bid for $blockIdx$, $bdim$ for $blockDim$, and tid for $threadIdx$. The constraints $bid.* < gdim.*$ for $* \in \{x, y\}$ and $tid.* < bdim.*$ for $* \in \{x, y, z\}$ always hold. Groups of 32 (a "warp") consecutively numbered threads within a thread block are scheduled at a time in a Single Instruction Multiple Data (SIMD) fashion.

2.2 CUDA Error Classes and Test Generation

2.2.1 Deadlocks

Deadlocks occur when any two threads in a thread block fail to encounter the same *textually aligned* barriers [11], as in kernel `deadlock` below. Here, threads satisfying `tid.x + i > 0` invoke the barrier while the other threads do not:

```
__global__ void deadlock(int i) {
  if (tid.x + i > 0)
    { ...; __syncthreads(); }   }
```

Random test input generation does not guarantee path coverage especially when conditionals are deeply embedded, whereas GKLEE's *directed test generation* based on SMT-solving ensures coverage. While the basic techniques for such test generation have been well researched in the past, GKLEE's contributions in this area include addressing the CUDA semantics and memory model, and detecting non-textually aligned barriers, a simple example of which is below. Here, the threads encounter different barrier calls if they diverge on the condition $tid.x + i > 0$.

```
  if (tid.x + i > 0) { ...; __syncthreads(); }
  else { ...; __syncthreads(); }
```

2.2.2 Data Races

There are three broad classes of races: intra-warp races, inter-warp races, and device/CPU memory races. Intra-warp races can be further classified into intra-warp races without warp divergence, and intra-warp races with warp divergence.

Intra-warp Races Without Warp Divergence: Given that any two threads within a warp execute the same instruction, an intra-warp race (without involving warp divergence) has to be a write-write race. The following is an example of such a race which GKLEE can successfully report. In this example, writes to shared array `v[]` overlap; *e.g.*, thread 0 and 1 concurrently write four bytes beginning at `v[0]` (in a 32-bit system).

```
__global__ void race()
{  x = tid.x >> 2; v[x] = x + tid.x; }
```

Intra-warp Races With Warp Divergence: In a divergent warp, a conditional statement causes some of the threads to execute the *then* part while others execute the *else* part. But because of the SIMD nature, *both* parts are executed with respect to all the threads in *some unspecified order* (undefined in the standard). Thus, in example ‘race’, depending on the hardware platform: (i) the even threads may read v first, and then the odd threads write v; or (ii) the odd threads may write v and then the even threads may read v:

```
__global__ void race()          {
  if (tid.x % 2) { ... = v ; }
  else { v = ... ; }            }
```

While on a given machine the results are predictable (either the then or the else happens first) an unpleasant surprise can result when this code is ported to a future machine where the else happens first (think of it as a “*porting race*”—race-like outcome that surfaces when the code is ported). The culprit is of course overlapped accesses across divergent-warp threads, but if v is a complicated array expression, this fact is virtually impossible to discern manually. GKLEE’s novel contribution is to detect such overlaps exactly regardless of the complexity of the conditionals or the array accesses. (For simplicity, we do not illustrate a variant of this example where both accesses are updates to v.)

This example also covers another check done by GKLEE: it reports the number of occurrences of divergent warps over the whole program.

Inter-warp Races: Inter-warp races could be read-write, write-read, or write-write: we illustrate a read-write race below. Here there is the danger that thread 0 and thread $bdim.x - 1$ may access $v[0]$ simultaneously while these two threads also belong to different warps in a thread block.

```
__global__ void race() {
  v[tid.x] = v[(tid.x + 1) % bdim.x]; }
```

Testing may fail to reveal this bug because this bug is typically noticed only when the write by one thread occurs before the read by the other thread. However, the execution order of threads in a GPU is non-deterministic depending on the scheduling, and latencies of memory accesses. GKLEE guarantees to expose this type of race.

Global Memory Races: GKLEE also detects and reports races occurring on global device variables:

```
__device__ x;
__global__ void race()
{  ...conflicting accesses to x by two threads... }
```

2.2.3 Memory Access Inefficiencies

There are two kinds of memory access inefficiencies: bank conflicts and non-coalesced memory accesses. GKLEE reports their severity by reporting the absolute number and the percentage of accesses that suffer from this inefficiency, as described in § 5.1 in detail.

Shared Memory Bank Conflicts: Bank conflicts result when adjacent threads in a half warp (for the CUDA compute capability 1.x model) or entire warp (for capability 1.2) access the same memory bank. GKLEE checks for conflicts by symbolically comparing whether two such accesses can fall into a memory bank.

Non-coalesced Device Memory Accesses: Non-coalesced memory accesses waste considerable bus bandwidth when fetching data from the device memory. Memory coalescing is achieved by following access rules specific to the GPU compute capability. GKLEE faithfully models all 1.x and 2.x compute capability coalescing rules, and can be run with the compute capability specified as a flag option (illustrates the flexibility to accommodate future such options from other manufacturers).

2.2.4 Test Generation

The ability to automatically generate high quality tests and verify kernels over all possible inputs is a unique feature of GKLEE. The BitonicSort (Figure 2) kernel taken from CUDA SDK 2.0 [9] sorts $values$’s elements in an ascending order. The steps taken in this kernel to improve performance (coalescing global memory accesses, minimizing bank conflicts, avoiding redundant barriers, and better address generation through bit operations) unfortunately end up obfuscating the code. Manual testing or random input-based testing does not ensure sufficient coverage. Instead, given a post-condition pertaining to the sortedness of the output array, GKLEE generates targeted tests that help exercise all conditional-guarded flows. Also, running this kernel under GKLEE by keeping all configuration parameters symbolic, we could learn (through GKLEE’s error message) that this kernel works only if $bdim.x$ is a power of 2 (an undocumented fact).

Covering all control-flow branches can result in too many tests. GKLEE includes for test-case minimization, as detailed in § 4.

```
__shared__ unsigned shared[NUM];

inline void swap(unsigned& a, unsigned& b)
{  unsigned tmp = a; a = b; b = tmp; }

__global__ void BitonicKernel(unsigned* values) {
  unsigned int tid = tid.x;
  // Copy input to shared mem.
  shared[tid] = values[tid];
  __syncthreads();

  // Parallel bitonic sort.
  for (unsigned k = 2; k <= bdim.x; k *= 2)
    for (unsigned j = k / 2; j > 0; j /= 2) {
      unsigned ixj = tid ^ j;
      if (ixj > tid) {
        if ((tid & k) == 0)
          if (shared[tid] > shared[ixj])
            swap(shared[tid], shared[ixj]);
        else
          if (shared[tid] < shared[ixj])
            swap(shared[tid], shared[ixj]);
      }
      __syncthreads();
    }

  // Write result.
  values[tid] = shared[tid];
}
```

Figure 2. The Bitonic Sort Kernel

3. Algorithms for Analysis, Test Generation

Given a C++ program, the GKLEE VM (Figure 1) executes the following steps, in order, for each control-flow path pursued during execution (to a first approximation, one can think of a control-flow tree and imagine all the following steps occurring *for each tree path* and *for each barrier interval along the path*). Deadlock checking and test generation occur per path (spanning barrier intervals; the notion of barrier intervals is explained in § 3.2). GKLEE checks for barriers being textually aligned and applies a canonical schedule going from one textually aligned barrier to another one.

- Create the GPU memory objects as per state model; infer memory regions representing GPU memory dynamically (§ A.5.2)
- Execute GPU kernel threads via the *canonical schedule* (§ 3.2)
- Fork new states upon non-determinism due to symbolic values, apply search heuristics and path reduction if needed (§ 2.2.4)
- In a state, at the end of the barrier interval or other synchronization points, perform checks for data races, warp divergence, bank conflicts, and non-coalesced memory accesses (§ 3.2)

- When execution path ends, report deadlocks and global memory races (if any), perform test-case selection, and write out a concrete test file (§ 4)

3.1 LLVM$_{cuda}$

The front-end compiles a C/C++ kernel program into LLVM bytecode with extensions for CUDA. Figure 3 shows an excerpt of its syntax. One main extension is that a variable is attached with its memory sort indicating which memory it refers to.

var	$:=$	$var_{cuda} \mid v : \tau$	variable
var_{cuda}	$:=$	$tid, bid, \ldots$	CUDA built-in
τ	$:=$	$\tau_{\_}, \tau_l, \tau_s, \tau_d, \tau_h$	memory sort
lab	$:=$	$l_1, l_2, \ldots$	label
e	$:=$	$var \mid n$	atomic expression
$instr$	$:=$	$br\ v\ lab1\ lab2$	conditional branch
	$\mid$	$br\ lab$	unconditional jump
	$\mid$	$\texttt{store}\ e\ v$	store
	$\mid$	$v = \texttt{load}\ v$	load
	$\mid$	$v = \texttt{binop}\ e\ e$	binary operation
	$\mid$	$v = \texttt{alloc}\ n\ \tau$	memory allocation
	$\mid$	$v = \texttt{getelptr}\ v\ e$	address calculation
	$\mid$	$\texttt{sync}$	synchronization barrier

Figure 3. Syntax of LLVM$_{cuda}$ (excerpt)

Figure 4 gives a small-step operational semantics of LLVM $_{cuda}$ using the following elements. A program is a map from labels to instructions; a value consists of one or more bytes (our model has byte-level accuracy); a memory or store maps variables to values, where each variable is assigned an integer address by the compiler. GKLEE models CUDA's memory hierarchy in a symbolic state as in Figure 5: each thread has its own local memory and stack (we combine them into a single local state in GKLEE); the threads in a block shares the shared memory; and all blocks share the device memory and the CPU memory. Each thread has a program counter (pc) recording the label of the current instruction.

Program	$:=$	$\mathbb{L} \subset lab \mapsto instr$
Value	$:=$	$\mathbb{V} \subset \texttt{byte}^+$
Memory, Store	$:=$	$M \subset var \mapsto \mathbb{V}$
Shared state	$:=$	$\mathbb{M} \subset (bid \mapsto M) \times M \times M$
Local state	$:=$	$\sigma \subset var \mapsto \mathbb{V}$
Data State	$:=$	$\Sigma \subset (tid \mapsto \sigma) \times \mathbb{M}$
Program counter	$:=$	$\mathbb{P} \subset tid \mapsto lab$
State	$:=$	$\Phi \subset \Sigma \times \mathbb{P}$

A state Φ consists of a data state Σ and a PC $\mathbb{P}$. Thread t's pc is given by $\mathbb{P}[t]$. Notations $\Sigma[v]$ and $\Sigma[v \mapsto k]$ indicate reading v's value from Σ and updating v's value in Σ to k respectively. Notation $\Sigma \vdash e$ evaluates e's value over Σ, *e.g.* $\Sigma \vdash e_1 = e_2$ is true if $\Sigma[e_1] = \Sigma[e_2]$. The semantics of an instruction is modeled by a state transition, *e.g.* the execution of an instruction br l$'$ at thread t updates the t's pc to l' and keeps the data state unchanged. Rule 9 specifies the barrier's semantics: a thread can proceed to the next instruction only after all the threads in the same block have reached the barrier. As indicated by other rules, non-barrier instructions are executed without synchronizing with other threads.

Memory Typing. After a source program is compiled into LLVM bytecode, it is difficult to determine which memory is used when an access is made because the address of this access may be calculated by multiple bytecode instructions. We employ a novel and simple GPU-specific memory sort inference method by computing for each (possibly symbolic) expression a sort τ which is either $\tau_{\_}$ (unknown), τ_l (local), τ_s (shared), τ_d (device), or τ_h (host), as per the rules (here we present the simplified version) in Figure 4. In our

1. $$\frac{\mathbb{L}[l]{=}\texttt{br l}'}{(\Sigma,\mathbb{P}){\longrightarrow}_t(\Sigma,\mathbb{P}[t{\mapsto}l'])}$$

2. $$\frac{\mathbb{L}[l]{=}\texttt{br v l}_1\ \texttt{l}_2 \quad \Sigma\vdash v}{(\Sigma,\mathbb{P}){\longrightarrow}_t(\Sigma,\mathbb{P}[t{\mapsto}l_1])} \qquad \frac{\mathbb{L}[l]{=}\texttt{br v l}_1\ \texttt{l}_2 \quad \Sigma\vdash \neg v}{(\Sigma,\mathbb{P}){\longrightarrow}_t(\Sigma,\mathbb{P}[t{\mapsto}l_2])}$$

3. $$\frac{\mathbb{L}[l]{=}(v{=}\texttt{alloc n}\ \tau)}{(\Sigma,\mathbb{P}){\longrightarrow}_t(\Sigma[(v{:}\tau){\mapsto}0^n],\mathbb{P}[t{\mapsto}l{+}1])}$$

4. $$\frac{\mathbb{L}[l]{=}(v_2{=}\texttt{getelptr}\ v_1{:}\tau\ e)}{(\Sigma,\mathbb{P}){\longrightarrow}_t(\Sigma[v_2{:}\tau{\mapsto}\Sigma[v_1]{+}\Sigma[e]],\mathbb{P}[t{\mapsto}l{+}1])}$$

5. $$\frac{\mathbb{L}[l]{=}(v{=}\texttt{binop}\ e_1{:}\tau\ e_2{:}\tau_{\_})}{(\Sigma,\mathbb{P}){\longrightarrow}_t(\Sigma[v{:}\tau{\mapsto}\texttt{binop}(\Sigma[e_1],\Sigma[e_2])],\mathbb{P}[t{\mapsto}l{+}1])}$$

6. $$\frac{\mathbb{L}[l]{=}(v_2{=}\texttt{load}\ v_1{:}\tau) \quad \tau{\neq}\tau_{\_}}{(\Sigma,\mathbb{P}){\longrightarrow}_t(\Sigma[v_2{:}\tau_{\_}{\mapsto}\Sigma[v_1]],\mathbb{P}[t{\mapsto}l{+}1])}$$

7. $$\frac{\mathbb{L}[l]{=}(\texttt{store}\ e\ v{:}\tau) \quad \tau{\neq}\tau_{\_}}{(\Sigma,\mathbb{P}){\longrightarrow}_t(\Sigma[v{:}\tau{\mapsto}\Sigma[e]],\mathbb{P}[t{\mapsto}l{+}1])}$$

8. $$\frac{v{:}\tau_{\_} \quad ((v'{:}\tau'){\mapsto}k)\in\Sigma \quad \Sigma\vdash v'{\leq}v{\leq}v'{+}\texttt{sizeof}(k)}{v{:}\tau'}$$

9. $$\frac{\mathbb{L}[l]{=}\texttt{sync} \quad \forall t'{\in}\texttt{blk\_of}(t):\mathbb{P}[t']{\in}\{l,l{+}1\}}{(\Sigma,\mathbb{P}){\longrightarrow}_t(\Sigma,\mathbb{P}[t{\mapsto}l{+}1])}$$

Figure 4. Operational semantics of LLVM$_{cuda}$ (excerpt)

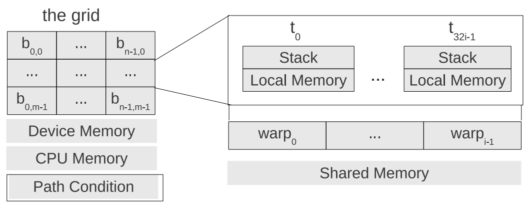

Figure 5. Components in a Symbolic State.

experience, these rules have been found to be sufficiently precise on all the kernels we have applied GKLEE to.

For example, Rule 4 models *getelptr* which refers to pointer dereferencing where v_2's type is obtained from v_1's type. Rule 6 indicates that a load instruction can be executed only if the address type is known; and the value loaded from memory has unknown type. Rule 8 says that a valid type is found for v if there exists a memory object associated with v' such that v's value falls within this object. Basically it searches the memory hierarchy to locate the target memory when the previous analysis fails to find v's type. If v represents a pointer which can refer to multiple objects (determined by SMT solving), then multiple states are generated, each of which needs to apply this rule. This often reveals memory type related bugs in the source kernel, *e.g.* mixing up the CPU and GPU memory. We plan to use Clang's ongoing support for LLVM+CUDA [12] to simplify such inference. More semantics rules (with sort inference) are available in [13].

State Model. In a symbolic state in GKLEE, each thread (in a block) has its own stack and local memory; each block has a shared memory; all blocks can access the device memory in the GPU and the main memory in the CPU. Figure 5 gives an example state for a GPU with grid size $n \times m$ and block size $32 \times i$. Each block consists of i of warps; each warp contains 32 threads. To support test generation, a state also contains a path condition recording the branching decisions made so far.

CUDA Built-in Variables. CUDA built-in variables include the block size, block id, thread id, and so on, The executor accesses these variables during the execution. GKLEE sets their values in respective memories before the execution. For example, the variable for the thread id, tid, is assigned *three* 32 bit words in the local memory of each thread. These words record the tid's values in dimension x, y and z respectively.

tid : τ_l (96b)	...
{x : 32b, y : 32b, z : 32b}	...

3.2 Canonical Scheduling and Race Checking

We now focus on the interleavings of all the threads within a thread block *from one barrier call to another* (global memory accesses across thread blocks are discussed later). Naively interleaving these threads will result in an astronomical number of interleavings. GKLEE employs the following schedule generation approach:

- Pursue just one schedule, namely the canonical schedule shown in Figure 6 where each thread is fully executed within a barrier interval before moving on to another thread.
- During the execution of all the threads in the current barrier interval, build a read-set $\mathcal{R}$ and a write set $\mathcal{W}$, recording in them (respectively) all loads and stores (these will be in mixed symbolic/concrete form) encountered in the execution.
- After the check points (as shown in Figure 6), build all possible conflict pairs, where a pair $\langle r_1, w_1\rangle$ or $\langle w_2, w_1\rangle$ is any pair that could potentially race or other conflicts.
- Through SMT-solving, decide whether any of these conflicts are races. If none are races (do not overlap in terms of a memory address), then the canonical schedule is equivalent to any other schedule. Thus, we can carry on to the next barrier interval with the next-state calculated as per the canonical schedule.

We now argue, with some caveats, that the canonical scheduling method is sound (will neither result in omissions or false alarms). The caveats are that C/C++ has no standard shared memory consistency semantics to define safe compiler optimizations, and the CUDA programming guide [14] provides only an informal characterization of CUDA's weak execution semantics. Assume that the instructions within CUDA threads in a barrier interval can be reordered; then under no conflicts (DRF), reordering transformations are sound [15]. This result also stems from [16] where it is shown that race detectors for sequential consistency can detect the *earliest race* even under weak orderings. One can also infer this result directly from [17] where it is shown that under the absence of conflict edges, the *delay set* (set of required program orderings) can be empty. We further elaborate on the soundness of the canonical scheduling method (also considering SIMD execution) in [13].

Consider the following two schedules, we record the writes and reads on v and see whether these accesses overlap at the end point (the check is denoted by a "!"). A race occurs in schedule 2 if and only if it also occurs in schedule 1.

$$\text{Schedule 1} : \Phi_0 \xrightarrow{\text{write v}}_{t_1} \Phi_1 \xrightarrow{\text{read v}}_{t_2} \Phi_2 \rightarrow \cdots \rightarrow \Phi_n(!)$$
$$\text{Sscheule 2} : \Phi'_0 \xrightarrow{\text{read v}}_{t_2} \Phi'_1 \xrightarrow{\text{write v}}_{t_1} \Phi'_2 \rightarrow \cdots \rightarrow \Phi'_n(!)$$

Intra-warp scheduling. A schedule is a sequence of state transitions made by the threads. The threads within a warp are executed in lock-step manner, and if they diverge on a condition, then one side (*e.g.* the "then" side) is executed first, with the threads in the other side blocked; and then the other side is executed (this is sound after checking for the absence of intra-warp races). (Note that GKLEE executes LLVM byte-codes, and is therefore able to capture the effect of compiler optimizations.)

In GKLEE, we schedule these threads in a lock-step manner, and provide an option to not execute the two sides sequentially. Now we show that these two scheduling methods are equivalent if no data race occurs. Specifically, the sequence

$$\Phi_0 \xrightarrow{c}_{t_1} \Phi_1 \xrightarrow{c}_{t_2} \cdots \xrightarrow{c}_{t_n} \Phi_n \xrightarrow{\neg c}_{t_1} \cdots \xrightarrow{\neg c}_{t_n} \Phi_{2n}$$

can be shuffled into the following one provided that it is race-free. We use $\xrightarrow{c}_{t_i}$ to indicate that thread t_i makes the transition with condition c.

$$\Phi_0 \xrightarrow{c}_{t_1} \Phi_1 \xrightarrow{\neg c}_{t_1} \Phi'_2 \xrightarrow{c}_{t_2} \cdots \xrightarrow{c}_{t_n} \Phi'_{2n-1} \xrightarrow{\neg c}_{t_n} \Phi_{2n}$$

Since c exclusive-or ($\oplus$) $\neg c$ holds for a thread, the sequence is equivalent to the following one (where $\Phi''_n = \Phi_{2n}$) which GKLEE produces. This is the *canonical schedule for intra-warp* steps.

$$\Phi_0 \xrightarrow{c\oplus\neg c}_{t_1} \Phi''_1 \xrightarrow{c\oplus\neg c}_{t_2} \cdots \xrightarrow{c\oplus\neg c}_{t_n} \Phi''_n$$

Hence GKLEE's intra-warp scheduling is an equivalent model of the CUDA hardware's. It eases formal analysis and boosts the performance of GKLEE. Similarly, as in Figure 6 we can reduce a race-free schedule to a canonical one for inter-warps, multi-blocks, and barrier intervals (BIs). These transition relations are represented by $\rightarrow_w$, $\rightarrow_b$, and $\rightarrow_{bi}$ respectively.

$$\underbrace{\underbrace{\rightarrow_{t_0}\cdot\rightarrow_{t_1}\cdots\rightarrow_{t_{31}}\cdot}_{\rightarrow_{w_0}\cdot(!_1\ :\ \text{intra\_warp})}\quad\underbrace{\rightarrow_{t_{32}}\cdot\rightarrow_{t_{33}}\cdots\rightarrow_{t_{63}}\cdot}_{\rightarrow_{w_1}\cdot(!_1\ :\ \text{intra\_warp})}\quad\cdots}_{\rightarrow_{b_0}\cdot(!_2\ :\ \text{inter\_warp})}$$

$$\underbrace{\rightarrow_{b_0}\cdot\rightarrow_{b_1}\cdots}_{\rightarrow_{bi_0}\cdot}\quad\cdots\quad\underbrace{\rightarrow_{b_0}\cdot\rightarrow_{b_1}\cdots}_{\rightarrow_{bi_m}\cdot(!_3\ :\ \text{global\_mem})}$$

Figure 6. Canonical scheduling and conflict checking in GKLEE.

Conflict checking: Figure 6 indicates that GKLEE supports various conflict checking:

- Intra-warp race (denoted as $!_1$), checked at the end of a warp. Threads t_1 and t_2 incur such a WW race if they write different values to the same memory location in the same store instruction: $\exists l : \mathbb{L}[l] = \texttt{store}\ e\ v \ \wedge\ \mathbb{P}[t_1] = \mathbb{P}[t_2] = l$ and $\Sigma \vdash v_{t_1} = v_{t_2} \ \wedge\ e_{t_1} \neq e_{t_2}$ (GKLEE issues a warning if $e_{t_1} = e_{t_2}$). For a diverged warp, RW and WW races are also checked by considering whehter the accesses from both sides can conflict (discussed in Section 2.2).
- Inter-warp race (denoted as $!_2$), checked at the end of a block for each BI. Thread t_1 and t_2 (in different warps) incur such a race if they access the same memory location, and one of them is a write, and different values are written if both accesses are writes. Formally, let $R\langle t, v, e\rangle$ and $W\langle t, v, e\rangle$ denote that thread t reads e from location v and writes e to v respectively. Then a RW race occurs if $\exists R\langle t_1, v_1, e_1\rangle, W\langle t_2, v_2, e_2\rangle \ :\ \Sigma \vdash v_1 = v_2$ (or the case of exchanging t_1 and t_2); a WW race occurs if $\exists W\langle t_1, v_1, e_1\rangle, W\langle t_2, v_2, e_2\rangle \ :\ \Sigma \vdash v_1 = v_2 \ \wedge\ e_1 \neq e_2$ (again GKLEE will prompt for investigation if $e_{t_1} = e_{t_2}$).
- Global race (denoted as $!_3$), checked at the end of the kernel execution. Similar to inter-warp race but on the device or CPU memory. Deadlocks are also checked at $!_3$.

Conflict checking is performed at the byte level to faithfully model the hardware. Suppose a thread reads n_1 bytes starting from address a_1, and another thread writes n_2 bytes starting from address a_2, then a overlap exists iff the following constraint holds.

$$(a_1 \leq a_2 \ \wedge\ a_2 < a_1 + n_1) \ \vee\ (a_2 \leq a_1 \ \wedge\ a_1 < a_2 + n_2)$$

Without abstracting pointers and arrays, GKLEE inherits KLEE's methods for handling them: suppose there are n arrays declared in a program. Then, when $*p$ is evaluated, for every array the concolic executor will check whether p can fall within the array, spawning a

```
__global__ void histogram64Kernel(unsigned *d_Result,
                                  unsigned *d_Data, int dataN){
   const int threadPos =
        ((threadIdx.x & (~63)) >> 0) |
        ((threadIdx.x &    15) << 2) |
        ((threadIdx.x &    48) >> 4);      ...
    __syncthreads();
    for(int pos = IMUL(blockIdx.x, blockDim.x) + threadIdx.x;
        pos < dataN; pos += IMUL(blockDim.x, gridDim.x))    {
      unsigned data4 = d_Data[pos]; // top 10 is symb. for t5,
      ...
      addData64(s_Hist, threadPos, (data4 >> 26) & 0x3FU);  }
   __syncthreads(); ...
}
inline void addData64(unsigned char *s_Hist, int threadPos,
                      unsigned int data)                    {
    // Race of T5 and T13 with threadPos of 20,52 resp.
    s_Hist[threadPos + IMUL(data, THREAD_N)]++; //<- Race!  }
```

Figure 7. Write-write race in Histogram64 (SDK 2.0)

new state if so (works particularly well for CUDA, where pointers are usually used for indexing array elements).

Note that our method reports accurate results in contrast to static analysis methods such as [18] (where no decision procedures are applied) and [1] (which uses SMT solving but relies heavily on abstractions). The method in [2] uses run-time checking to rule out false alarms produced by its static analyzer; while GKLEE builds all the checks into its VM and produces no false alarms.

3.3 Power of Symbolic Analysis

We now present how GKLEE detected a WW race condition in `histogram64Kernel` (Figure 7), a CUDA SDK 2.0 kernel. Since the invocation of this kernel in `main` passes `d_Data` that can be quite large, a user of GKLEE (in this case, us) chose to keep only the first ten locations of this array symbolic, and the rest concrete at value `0`. (This is the only manual step needed; without this, GKLEE's solver will be inundated, trying to enumerate every array location). GKLEE now determines that `addData64` can be called concurrently by two distinct threads. Drilling into this function, GKLEE generates constraints for `s_Hist[threadPos + IMUL(data, THREAD_N)]++` (not marked `atomic`) to race. The SMT solver picks two thread IDs 5 and 13; for this, `threadPos` assumes values 20 and 52, respectively. What flows into `data` is `data4 >> 26 & 0x3FU`, where `data4` obtains the value of `d_Data[pos]`. Since the top 10 elements of `d_Data[DATA_N]` are symbolic, thread 5 assigns a symbolic value denoted by `d_Data[5]` to `data4`, while thread 13 assigns the concrete value of `0` to `d_Data[13]`. The SMT solver now solves $20 + ((d\_Data[5] \gg 21) \& 2016) = 52 + 0$ ($\gg 26$ changed to $\gg 21$ because `THREAD_N` is 32), resulting in `d_Data[5]` obtaining value `0x04040404` which causes a race! The user not only obtains an automatic race alert, but also the concrete input of `0x04040404` to set `d_Data[5]` to, in case they want to study this race through any other means.

4. Test Generation

During its symbolic execution, GKLEE's VM has the ability to *fork* two execution paths whenever it "encounters a non-deterministic situation;" *e.g.* when a conditional is evaluated and both choices are true, or when a symbolic pointer is accessed, and it may point to multiple memory objects. GKLEE organizes the resulting execution states as a tree. The initial state of the GPU kernel forms the root of this tree. It then searches the state space guided by various search reduction heuristics.

The essence of the VM executor: GKLEE can be regarded as a symbolic model checker (for GPU kernels) with the symbolic state modeling the hardware state and the transitions modeling non-determinism due to symbolic inputs.

With this view, it is natural that GKLEE supports facilities such as state caching and search heuristics (*e.g.* depth-first, weighted-random, bump-merging, *etc.*), all of which are inherited from KLEE. The checks discussed in Section 3 are essentially built-in global safety properties examined at each state. In the state space tree, a path from the root to a leaf represents a valid computation with a path condition recording all the branching decisions made by all the threads. At a leaf state, we can generate a test case by solving the satisfiability of this path condition. This ability makes GKLEE a powerful test generator.

Soundness and completeness of the test generator: Given a race free kernel with a set of symbolic inputs, GKLEE visits a path if and only if there exists a schedule where the decisions made by threads (recorded in the path condition) are feasible.

Note that the feasibility of a path condition is calculated by SMT solving, which is precise without any approximation. At the first glance, the completeness of test generation may be not be obvious since we consider only one (canonical) schedule, while another schedule may apply the branchings in a different order.

To clarify this, consider the following situation where thread t_0 (t_1) branches on conditions $c_{0,0}$ ($c_{1,0}$):

$$\begin{array}{cc} t_0 & t_1 \\ \text{if } (c_{0,0}) \ldots ; & \text{if } (c_{1,0}) \ldots ; \end{array}$$

If t_0 executes before t_1, then a depth-first search visits 4 paths with path conditions $c_{0,0} \wedge c_{1,0}$, $c_{0,0} \wedge \neg c_{1,0}$, ... If t_1 executes before t_0, then the 4 path conditions become $c_{1,0} \wedge c_{0,0}$, $c_{1,0} \wedge \neg c_{0,0}$ .... The commutativity of the $\wedge$ operator ensures, under the race-free constraint, the equivalence of these two path sets. Hence, it suffices to consider only one canonical schedule in test generation as in conflict checking (Section 3).

Example. Consider the Bitonic kernel running on one block with 4 threads. Suppose the input $values$ is of size 4 and has symbolic value v. Lines 1-4 copy the input to $shared$: $\forall i \in [0, 3] : shared[i] = v[i]$. For thread 0, since lines 7-8 involve no symbolic values, they are executed concretely. In the first iteration of the inner loop, we have $k = 2$, $j = 1$, and $ixj = 1$. The conditional branch at line 10 is evaluated to be true; so does that at line 11. Then the execution reaches the branch at line 12. GKLEE queries the constraint solver to determine that both branches are possible; it explores both paths and proceeds to the loop's next iteration. Finally the execution terminates with 28 paths (and test cases).

Coverage Directed State/Path Reduction. Given that a kernel is usually executed by a large number of threads, there is a real danger, especially with complex/large kernels, that multiple threads may end up covering some line/branch while no threads visit other lines/branches.[1] We have experimented with several heuristics that help GKLEE achieve *coverage directed* search reduction. Basically, we keep track of whether some feature (line or branch) is covered by all the threads at least once, or some thread at least once. These measurements help GKLEE avoid exploring states/paths that do not result in added coverage.

Another usage of these metrics is to perform *test case selection* which still explores the entire state space, but outputs only a subset of test cases (for downstream debugging use) after the entire execution is over, with no net loss of coverage. Details of these heuristics are discussed in § 5.2. To the best of our knowledge, coverage measures for SIMD programs have not been previously studied.

[1] We have extended GKLEE's symbolic VM to measure statement and branch coverage in terms of LLVM byte-code instructions.

5. Experimental Results

As described in Section 1, a GPU kernel along with a CPU driver is compiled into LLVM bytecode, which is symbolically executed by GKLEE. Since GKLEE can handle GPU and CPU style code, we can mix the computation of CPU and GPU, *e.g.* execute multiple kernels in a sequence.

```
   CPU code; GPU code; CPU code; GPU code; ...
```

Driver. The user may give as input a kernel file to test together with a driver representing the main (CPU side) program. To cater for the need of LLVM-GCC, we redefine some CUDA specific directives and functions, *e.g.* we use C attributes to interpret them, as illustrated by the following definition of `__shared__`.

```
#define __shared__
        __attribute((section ("__shared__")))

#define cutilSafeCall(f) f
void cudaMalloc(void** devPtr, size_t size) {
  *devPtr = malloc(size);
}
void cudaMemcpy(void* a, void* b, size_t size, ...)
{ memcpy(a,b,size); };
```

We show below an example driver for the Bitonic Sort kernel. The user specifies what input values should have symbolic values; and may place `assert` assertions anywhere in the code, which will be checked during execution. Particularly, the pre- and post- conditions are specified before and after the GPU code respectively. Function `__begin_GPU(NUM)` (a more general format is `__begin_GPU(bdim.x,bdim.y,bdim.z,gdim.x,gdim.y,gdim.z)`) specifies that the x dimension of the block size is NUM.

```
int main() {
  int values[NUM];
  gklee_make_symbolic(values, NUM, "input");

  int* dvalues;
  cutilSafeCall(cudaMalloc((void**)&dvalues,
               sizeof(int)*NUM));
  cutilSafeCall(cudaMemcpy(dvalues, values,
     sizeof(int)*NUM, cudaMemcpyHostToDevice));

  // <<<...>>>(BitonicKernel(dvalues))
  __begin_GPU(NUM);        // block size = <NUM>
  BitonicKernel(dvalues);
  __end_GPU();

  // the post-condition
  for (int i = 1; i < NUM; i++)
    assert(dvalues[i-1] <= dvalues[i]);

  cutilSafeCall(cudaFree(dvalues));
}
```

A concrete GPU configuration can be specified at the command line. For instance, option −blocksize=[4,2] indicates that each block is of size 4×2. These values can also be made symbolic so as to reveal configuration limitations.

5.1 Results I: Symbolic Verification

GKLEE supports (through command-line arguments) bank conflict detection for 1.x (memory coalescing checks cover 1.0 & 1.1, and 1.2 & 1.3), as well as 2.x device capabilities. Table 1 presents results from SDK 2.0 kernels while Table 2 presents those from SDK 4.0 (many of these are written for 2.x). These are widely publicized kernels. Our results are with respect to symbolic inputs. **Tables (1 and 2)**: (#T denoting the number of threads analyzed) asserts that, under valid configurations, (i) all barriers were found to be well synchronized; (ii) the functional correctness is verified (w.r.t the configurations); *but only the canonical schedule is considered for cases with races (marked with *)* (thus for cases with fatal races, we are unsure of the overall functional correctness); (iii) performance defects (to specific degrees) were found in many kernels; (iv) two races were observed (Histogram64 and RadixSort kernels); and (v) several alerts pertaining to the use of `volatile` declarations were reported. 'WW' denotes write-write races; they are marked *benign* (ben.) if the same value is written in our concrete execution trace. The computation is expected to be deterministic.

The race in Radix Sort was within function `radixSortBlockKeysOnly()` involving `sMem1[0] = key.x` for distinct `key.x` written by two threads. In `Histogram64`, we mark the race WW$^?$ as we are unsure whether `s_Hist[..]++` of Figure 7 executed by two threads *within one warp* is fatal (apparently, CUDA guarantees[2] a net increment by 1). It is poor coding practice anyhow (we notate correctness as 'Unknown').

Two rows of results are presented for Bank Conflicts, Memory Coalescing, and Warp Divergence, the upper row averaging over barrier intervals and the lower row averaging over Warps. The 94% for Scalar Product under Bank Conflict (compute capability 2.x) is obtained by: 57 BIs were analyzed, and out of it, 54 had bank conflicts, which is 94%. All other "`z%`" entries may be read similarly. This sort of a feedback enables a programmer to attempt various optimizations to improve performance. When a kernel's execution contains multiple paths (states), the average numbers for these paths are reported.

Also, with GKLEE's help, we tried a variety of configurations (*e.g.* symbolic configurations) and discovered undocumented constraints on kernel configurations and inputs.

To show that the numbers reported by GKLEE track the profile, we employed GKLEE-generated concrete test cases and ran selected kernels on the Nvidia GTX 480 hardware. GKLEE includes a utility script, gklee-replay, that compiles the kernels using nvcc, executes them on the hardware and optionally invokes the NVIDIA command line profiler (which is the back end to their Compute Visual Profiler). GKLEE's statistics can be used for early detection of these performance issues on symbolic inputs.
Future work: GKLEE's reporting of bank conflicts, non-coalesced accesses, and warp divergence instances can help future performance analysis tools do a more accurate measurement in terms of how many encounters of these issues (dynamic instances) will be caused by typical input data sets.

Volatile Checking Heuristic GKLEE employs a heuristic to help users check for potentially missed volatile qualifiers. Basically, GKLEE analyzes for data sharings between threads within one warp involving two *distinct* SIMD instructions. The gist of an example (taken from the CUDA SDK 2.0) when it was compiled for device capability of 2.x, was as follows: a sequence '`a;b`' occurred inside a warp where SIMD instruction '`a`' writes a value into addresses a_1 and a_2 on behalf of t_0 and t_1, respectively; and SIMD instruction '`b`' reads a_0 and a_1 in t_0 and t_1, respectively. Now t_1 was meant to see the value written into a_1, but it did not, as the value was held in a register and not written back (a volatile declaration was missing in the SDK 2.0 version of the example). An Nvidia expert confirmed our observation and has updated the example to now have the volatile declaration.

We now provide a few more details on this issue. The SDK 4.0 version of this example *has* the volatile declaration in place. We exposed this bug when we took a newer release of the `nvcc` compiler (released around SDK 4.0 and does volatile optimizations), compiled the SDK 2.0 version of this example (which omits the volatile), ran the program on our GTX 480 hardware, finding incorrect results emerging. The solution in GKLEE is to flag for potentially missed volatiles in the aforesaid manner; in future, we hope

[2] As confirmed through discussions with engineers at Nvidia.

to extend GKLEE to "understand" compiler optimizations and deal with this issue more thoroughly.

Table 3 compares the execution times of GKLEE and our functional correctness checking tool PUG [1]. This result shows the pros and cons of a full SMT based static analyzer (like PUG) or a testing based approach (like GKLEE) which is far more scalable. We performed experiments on a laptop with an Intel Core(TM)2 Duo 1.60GHz processor and 2GB memory. Here the GPU times in GKLEE count in sanity checking and test generation. Similar to GKLEE, PUG also sequentializes the threads and unrolls the loops when checking functional correctness. GKLEE outperforms PUG due partially to its various optimizations such as expression rewriting, value concretization, constraint independence, and so on. A more important factor is that GKLEE is a *concolic* tool which simplifies the expressions on-the-fly and puts much less burden to the SMT solver, in addition to generating concrete tests, which PUG does not. Both tools perform poorly on the "Bitonic Sort" kernel since the relation between this kernel's input elements are complicated, *e.g.* thus GKLEE needs to explore many paths. Section 2.2.4 presents GKLEE's reduction heuristics to ameliorate this.

As an added check, we tested GKLEE on the same 57 kernels used in [1]. GKLEE found the same 2 real bugs (one deadlock and one WR race). It also revealed that 4 of other kernels contain functional correctness bugs.

5.2 Results II: Testing and Coverage

We assess GKLEE with respect to newly proposed coverage measures and coverage directed execution pruning. In Table 4, we attempt to measure the source-code coverage by converting the given kernel into a sequential version (through Perl scripts) and applying the `gcov` tool (better means are part of future work). The point is that source-code coverage may be deceptively high, as shown ("`a/b`" means "statements/branches" covered; collectively, we call this a *target*). This is the reason we rely upon only byte-code measures, described in the sequel.

GKLEE first generated tests for the shown kernels covering all feasible paths, and subsequently performed *test case selection*. For example, it first generated the 28 execution paths of Bitonic Sort; then it trimmed back the paths to just 5 because these five tests covered all the statements and branches at the byte-code level.

Four byte-code based target coverage measures were assessed first: (i) avg. Cov^t measures the number of targets covered by threads across the whole program, averaged over the threads, (ii) max. Cov^t that measures the maximum by any thread, (iii) avg. CovBI^t computes Cov^t separately for each barrier interval and reports the overall average, and (iv) max. CovBI^t is similar to avg. CovBI^t except for taking a maximum value. From Table 4, we conclude that the maximum measures give an overly optimistic impression, so we set them aside. We choose avg. CovBI^t for our baseline because activities occurring within barrier intervals are closely related, and hence separately measuring target coverage within BIs tracks programmer intent better.

Armed with avg. CovBI^t and min #tests, we assess several benchmarks (Table 5) with 'No Reductions', and two test reduction schemes. Runs with 'No Reductions' and no *test case selection* applied show the total number of paths in the kernels, and the upper limits of target coverage (albeit at the expense of considerable testing time). Red_{TB} is a reduction heuristic where we separately keep track of the coverage contributions by different threads. We continue searching till each thread is given a chance to hit a test target. For instance, in one barrier interval, if one target is reachable by all the threads, we continue exploring all these threads; but if the same target is reachable again (say in a loop), we cut off the search through the loop. In contrast, Red_{BI} only looks for some thread reaching each target; once that thread has, subsequent thread explorations to that target are truncated (more aggressive reductions). While the coverage achieved is nearly the same (due to the largely SIMD nature of the computations), it is clear that Red_{TB} is a bit more thorough.

The overall conclusion is that to achieve high target coverage (virtually the same coverage as with 'No Reductions'), reduction heuristics are of paramount importance, as they help contain test explosion. Specifically, the number of paths explored with reductions is much lower than that done with 'No Reductions.' A powerful feature of GKLEE is therefore its ability to output these minimized high-quality tests for downstream debugging.
Additional sanity-checking: we generated purely random inputs (as a designer might do); in all cases, GKLEE's test generation and test reduction heuristics provided far superior coverage with far fewer tests.

6. Related Work

Traditional CUDA program debuggers [19, 20, 21, 22] do not solve path constraints to home into relevant inputs that can trigger bugs. They examine bugs that occur only within platform executions.

Many past efforts have focused on multi-threaded programs synchronizing using locks and semaphores. The work of [23] uses code instrumentation and a model checker to analyze thread interleavings, using the model checker's partial order and symmetry reduction to combat state space explosion. Symbolic techniques for program analysis go back to works such as [24, 25] with concolic versions proposed in [6, 8] and more recently in KLEE [4]. GKLEE's approach is inspired by [4] much like five other recent open projects [7] have achieved. Concolic-execution based solvers for special domains also exist. None of these methods incorporate ways to deal with SIMD concurrency in GPUs and look for GPU-specific correctness or performance issues.

Except for GKLEE, there are only few GPU-specific checkers reported in the past. Table 6 gives a comparison of these tools. An instrumentation based technique is reported [3] to find races and shared memory bank conflicts. This is an ad-hoc testing approach, where the program is instrumented with checking code, and only those executions occurring in a platform-specific manner are considered. A similar method [2] is used to find races with the help of a static analysis phase. Static analysis is performed first to locate possible candidates so as to reduce the runtime overheads caused by instrumented code. These runtime methods cannot accept symbolic inputs and verify function correctness on open inputs, not to mention test generation. Moreover GKLEE supports a rich set of C++ language features (including those considered specifically in tools such as [26]) which other tools do not handle. In [27], a static analysis based method for divergence analysis and code optimization is presented.

While the approach of PUG [1] is SMT-based, it is not very scalable as shown in Table 3. Recently, simple analysis for memory coalescing was added to it [28]. PUG is also a kernel-at-a-time analyzer while GKLEE can analyze whole GPU programs.

Even if we narrow down to race detection on concrete inputs, instrumentation based tools may suffer from performance or extensibility problems because it is hard to implement sophisticated execution controls and decision procedures on the source level, while GKLEE does everything over an optimized symbolic virtual machine. As pointed out by Boyer [3], although it is possible to run an instrumentation based tool on the GPU (thus parallelizing its execution), CUDA only supports useful features (*e.g.* display debugging information, or recording traces in a file) in emulation mode which disables parallelism in GPU. Note that GKLEE supports test case replaying on the GPU. it also supports kernel simulation on the CPU as the CUDA debugger does. Last but not least, GKLEE can look for compiler-related bugs due to omitted volatiles.

Kernels	Loc	Race	Func. Corr.	#T	Bank Conflict ($\downarrow$ perf.)		Coalesced Accesses ($\uparrow$ perf.)			Warp Diverg. ($\downarrow$ perf.)	Volatile Needed
					1.x	2.x	1.0 & 1.1	1.2 & 1.3	2.x		
Bitonic Sort	30		yes	4	0%	0%	100%	100%	100%	60%	no
Scalar Product	30		yes	64	0%	0%	11%	100%	100%	100%	yes
Matrix Mult	61		yes	64	0%	0%	100%	100%	100%	0%	no
Histogram64$^{tb.}$	69	WW$^{?}$	Unknown	32	66%	66%	100%	100%	100%	0%	yes
Reduction (7)	231		yes	16	0%	0%	100%	100%	100%	16$\sim$83%	yes
Scan Best	78		yes	32	71%	71%	100%	100%	100%	71%	no
Scan Naive	28		yes	32	0%	0%	50%	100%	100%	85%	yes
Scan Workefficient	60		yes	32	83%	16%	0%	100%	0%	83%	no
Scan Large	196		yes	32	71%	71%	100%	100%	100%	71%	no
Radix Sort	750	WW	yes*	16	3%	0%	0%	100%	100%	5%	yes
Bisect Small	1,000	WW	–	16	38%	0%	97%	100%	100%	43%	yes
Bisect Large$^{tb.}$	1,400	ben.	–	16	15%	0%	99%	100%	100%	53%	yes

Table 1. SDK 2.0 Kernel results. "Reduction" contains 7 kernels with different implementations; we average the results. Results for "Histogram64," and "Bisect Large" are time-bounded (tb.) to 20 mins. Func. Corr. results about float values are skipped at –. We checked the integer version of "Radix Sort"; and CUDPP library calls involved in "Radix Sort" were not analyzed.

Kernels	Loc	Race	#T	Bank Conflict ($\downarrow$ perf.)		Coalesced Accesses ($\uparrow$ perf.)			Warp Diverg. ($\downarrow$ perf.)	Volatile needed/missed
				1.x	2.x	1.0 & 1.1	1.2 & 1.3	2.x		
Clock	38		64	0%	0%	0%	100%	100%	85%	no/no
Scalar Product	47		128	0%	0%	50%	100%	100%	36%	no/no
Histogram64$^{tb.}$	70		64	0%	33%	0%	0%	0%	0%	no/no
Scan Short	103		64	0%	0%	0%	100%	100%	0%	yes/no
Scan Large	226		64	0%	0%	0%	67%	67%	25%	yes/no
Transpose (8)	172		256	0$\sim$50%	0$\sim$100%	0$\sim$100%	0$\sim$100%	0$\sim$100%	0%	no/no
Bisect Small	1,000	WW	16	38%	0%	97%	100%	100%	43%	yes/yes

Table 2. SDK 4.0 Kernel results. If volatiles needed (N) is 'yes' and missed (M) is 'no', the code annotation is correct. Examples with both 'yes' (missed volatiles) were found. Transpose contains 8 different implementations; we report the results as a range through "$\sim$". Kernels having the same results as their SDK 2.0 versions, including Bitonic Sort, MatrixMult and Bisect Large, are not presented.

Kernels	#T = 4		#T = 16		#T = 64	#T = 256	#T = 1,024
	PUG	GKLEE	PUG	GKLEE	GKLEE	GKLEE	GKLEE
Simple Reduct.	2.8	$< 0.1(< 0.1)$	T.O	$< 0.1(< 0.1)$	$< 0.1(< 0.1)$	0.2(0.3)	2.3(2.9)
Matrix Transp.	1.9	$< 0.1(< 0.1)$	T.O	$< 0.1(0.3)$	$< 0.1(3.2)$	$< 0.1(63)$	0.9(T.O)
Bitonic Sort	3.7	0.9(1)	T.O	T.O	T.O	T.O	T.O
Scan Large	–	$< 0.1(< 0.1)$	–	$< 0.1(< 0.1)$	0.1(0.2)	1.6(3)	22(51)

Table 3. Execution times (in seconds) of GKLEE and PUG [1] on some kernels for functional correctness check. #T is the number of threads. Time is reported in the format of GPU time (entire time); T.O means $>$ 5 minutes.

Kernels	src. code coverage	min #test	avg. Cov^{t}	max. Cov^{t}	avg. CovBI^{t}	max. CovBI^{t}	exec. time
Bitonic Sort	100%/100%	5	78%/76%	100%/94%	79%/66%	90%/76%	1s
Merge Sort	100%/100%	6	88%/70%	100%/85%	93%/86%	100%/100%	1.6s
Word Search	100%/100%	2	100%/81%	100%/85%	100%/97%	100%/100%	0.1s
Suffix Tree Match	100%/90%	7	55%/49%	98%/66%	55%/49%	98%/83%	31s
Histogram64$^{tb.}$	100%/100%	9	100%/75%	100%/75%	100%/100%	100%/100%	600s

Table 4. Cov^{t} and CovTB^{t} measure bytecode coverage w.r.t threads. min #test tests are obtained by performing test case selection after the execution. Result for "Histogram64" is limited to 600 s. No test reductions used in generating this table. Exec. time on typical workstation.

SPMD programs are prone to incorrect synchronization patterns, especially when barriers are within conditional statements. Aiken and Gay [18] proposed a type system to check global synchronization errors by applying a simple single-value analysis, which may produce false alarms by rejecting correct programs. GKLEE uses SMT solving to compare expressions and is more precise.

The KLEE-FP [29] tool extends KLEE to cross-check IEEE 754 floating-point programs and their SIMD-vectorized versions. Two floating-point expressions are equivalent if they can be normalized to the same form through rewriting. This tool does not address the same class of correctness and performance bugs as GKLEE, neither does it produce concrete test cases. However, KLEE-FP's floating-point package along with real number support can help overcome GKLEE's current inability to handle float numbers.

Some Limitations of GKLEE. GKLEE cannot be used to analyze the functional correctness of CUDA applications that involve floating-point calculations (efficient SMT methods for floating-point arithmetic, when available, will help here). The concolic nature of GKLEE can help ameliorate this drawback by sometimes "concretizing" the floating numbers to integers. All other analyses done by GKLEE are unaffected by floating-point types, as typically variable addresses involve only unsigned integers.

Kernels	With No Reductions		Red_{TB}		Red_{BI}	
	#path	avg. CovBI^t	#path	avg.CovBI^t	#path	avg. CovBI^t
Bitonic Sort	28	79%/66%	5	79%/66%	5	79%/65%
Merge Sort	34	93%/86%	4	92%/84%	4	92%/84%
Word Search	8	100%/97%	2	100%/97%	2	94%/85%
Suffix Tree Match	31	55%/49%	6	55%/49%	6	55%/49%
Histogram64	13	100%/100%	5	100%/100%	5	100%/100%

Table 5. Reduction Heuristic Comparisons.

Comparison Categories	GKLEE	PUG [1]	GRace[2]	[3]
Methodology	Concolic Exec. in virtual machine	Symbolic Analysis	Static Analysis + Dyn. Check	Dynamic Check
Level of Analysis	LLVM Bytecode	Source Code	Source Code (Instrument.)	Source Code (Instrument.)
Bugs Targeted	Race (intra-/inter- warp, all memory), Warp Divergence, Deadlocks, Memory Coalesce, Bank Conflicts Compilation level bugs (e.g. Volatiles)	Shared Mem. Race, Deadlocks, Bank Conflict	Intra-/Inter- Warp Race	Shared Mem. race, Bank Conflict
False alarm elim.	SMT-solving, GPU replaying	Auto./Manual Refinement	Dynamic Execution	Dynamic Execution
Test Generation	Automatic, Hardware Execution, Coverage Measures, Test Reduction	Not supported	Not supported	Not supported

Table 6. Comparison of Formal Verifiers of GPU Programs

7. Concluding Remarks

We presented GKLEE, the first symbolic virtual machine based correctness checker and test generator for GPU programs written in C++. It checks several error categories, including one previously unidentified race type. We reported correctness errors and performance issues detected by GKLEE in real-world kernels. For many realistic kernels, finding a correctness violation takes less than a minute on a modern workstation. We propose several novel code coverage measures and show that GKLEE's test generation and test reduction heuristics achieve high coverage. Several future directions are planned: (i) OpenCL [10] support, (ii) handling formats other than LLVM (*e.g.*, Nvidia's PTX) using frameworks such as Ocelot [30], and (iii) scalability enhancement, including parameterized methods for SIMD programs [31].

Acknowledgements: We are indebted to the authors of [4] for releasing KLEE to the community well designed and documented.

References

[1] G. Li and G. Gopalakrishnan, "Scalable SMT-based verification of GPU kernel functions," in *SIGSOFT FSE*, 2010.

[2] 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.

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

[4] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI, 8th USENIX Symposium*, 2008.

[5] "SMT-COMP. http://www.smtcomp.org/2011."

[6] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*, 2005.

[7] "Klee open projects," http://klee.llvm.org/OpenProjects.html.

[8] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *10th ESEC/FSE*, 2005.

[9] "CUDA zone. http://www.nvidia.com/object/cuda_home.html."

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

[11] A. Kamil and K. A. Yelick, "Concurrency Analysis for Parallel Programs with Textually Aligned Barriers," in *LCPC*, 2005.

[12] "The LLVM compiler infrastructure. http://www.llvm.org/."

[13] "GKLEE Technical Report. http://www.cs.utah.edu/fv/GKLEE."

[14] "Cuda programming guide version 4.0. http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf."

[15] J. Sevcik, "Safe Optimisations for Shared-Memory Concurrent Programs," in *PLDI*, 2011.

[16] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. Netzer, "Detecting data races on weak memory systems," in *ISCA*, 1991.

[17] D. Shasa and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM TOPLAS*, vol. 10, no. 2, pp. 282–312, 1988.

[18] A. Aiken and D. Gay, "Barrier inference," in *POPL*, 1998.

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

[20] Nvidia, "Parallel Nsight," Jul. 2010.

[21] Allinea, "DDT," Nov. 2009.

[22] Rogue Wave, "Totalview for CUDA," Jan. 2010.

[23] S. Khurshid, C. S. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *TACAS*, 2003.

[24] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil, "Flavers: A finite state verification technique for software systems," *IBM Systems Journal*, vol. 41, no. 1, 2002.

[25] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *ICSE*, 2008.

[26] S. K. Lahiri, S. Qadeer, and Z. Rakamaric, "Static and precise detection of concurrency errors in systems code using SMT solvers," in *21st Computer Aided Verification (CAV)*, 2009.

[27] B. Coutinho, D. Sampaio, F. M. Quintao Pereira, and W. Meira Jr., "Divergence analysis and optimizations," in *PACT*, 2011.

[28] J. Lv, G. Li, A. Humphrey, and G. Gopalakrishnan, "Performance degradation analysis of GPU kernels," in *EC2 Workshop*, 2011.

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

[30] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *PACT*, 2010.

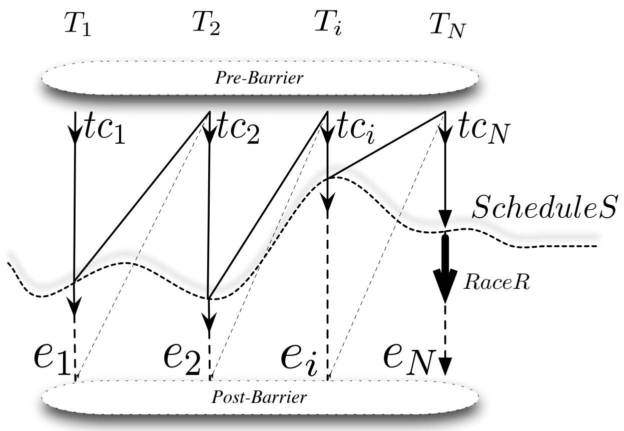

Figure 8. Proof of the Canonical Schedule Theorem

[31] G. Li, "Formal verification of programs and their transformations," Ph.D. dissertation, School of Computing, University of Utah, Aug. 2010, http://www.cs.utah.edu/fv.

A. Appendices

A.1 A Theorem About Canonical Scheduling

For the simplicity of exposition, let us first ignore SIMD scheduling (which will be considered momentarily). Consider an arbitrary schedule S under which threads T_1 through T_N have advanced from the initial state after the *Pre-Barrier* up to the wavy line of program counters. Regions t_{C_i} and e_i consist of labeled moved of the form $w \leftarrow f(r)$ where w is a location written into, and r is a location that is read. S consists of an arbitrary sequence of labeled moves taken from the initial state up to the wavy line, respecting individual thread program orders. Without loss of generality, let thread T_N execute one more step called $Race\ R$ labeled by $w_R \leftarrow f(r_R)$. The sequence $S; R$ results in the first observed race between R and one of the actions in S. Let us call $TCS = t_{C_1}, t_{C_2}, \ldots, t_{C_i}, \ldots, t_{C_N}$ the *truncated canonical schedule*. Clearly, $TCS; R$ will also cause the same race, since $S \equiv TCS$ because of the race-freedom of S.

Call the sequence $CS = t_{C_1}, e_1, t_{C_2}, e_2, \ldots, t_{C_i}, e_i, \ldots, t_{C_N}$ (omitting e_N) the *canonical schedule*; this is essentially a *sequential* execution of the threads from the *Pre-Barrier* to the *Post-Barrier*. It is clear that $CS \equiv TCS$ if none of the labeled moves within $e_1, \ldots, e_{N-1}$ races with any of the steps in TCS (if this happens, we would detect this "earlier race" and stop). Thus, unless we detect such an earlier race, $CS; R$ will also detect race R.

SIMD Scheduling: Each thread within a thread warp executes a SIMD instruction. The canonical scheduling method is sound so long as we model each labeled move mentioned above in terms of SIMD execution steps, where W includes all the locations updated by a SIMD instruction, and R includes all locations read. This extension has not been implemented explicitly in GKLEE yet, but is imminent. The scheduling order between the *then* and *else* parts of a divergent warp does not matter, in the absence of porting races (see "porting race" discussed in § 2.2.2).

A.2 Missing Volatile Violation Scenario

```
__global__ void
reducekernel() {
 //load into shared memory from global
 __syncthreads();
 for(s = 1; s < blockDim.x; s *= 2) {
   if (threadIdx.x % (2*s) == 0)
     //THE READ AND STORE
     sdata[tid] += sdata[tid + s];
   __syncthreads();
 }
 //store result to global memory
}
```

In this example (taken from the CUDA 2.0 SDK) when it was compiled for device capability of 2.0, the nvcc compiler optimized the store to shared memory such that the result was kept in a register and not written through until after the warps execution of the barrier interval containing 'THE READ AND STORE'.

When this kernel was executed with a block size of 4, and an initial data set of 5,6,7,8 (loaded into the shared memory array 'sdata'), the execution went as follows at 'THE READ AND STORE': THREAD 0, s = 1: read sdata[0] → 5, read sdata[1] → 6, store sdata[0] → 11 THREAD 2, s = 1: read sdata[2] → 7, read sdata[3] → 8, store sdata[2] → 15 THREAD 0, s = 2: read sdata[0] → 11, read sdata[2] → **7**, store sdata[0] → 18. Thread 0 could not see the result stored at sdata[2] by thread 2, 15, and read the initial value of 7. The fix was to declare `sdata` volatile.

A.3 Bank Conflict Case Study

```
__global__ void
transposeNoBankConflicts(float *odata, float *idata,
                            int width, int height, int nreps)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1]; // 16->TILE_DIM
  ... code omitted ...
  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
     { tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];}
    __syncthreads();
    ... code omitted ...
  }
}
```

Figure 9. A bank conflict detected by GKLEE from transpose kernel in SDK 4.0

Kernel transpose in Figure 9 claims there is no bank conflict for shared memory access. GKLEE reports the bank conflict occurs under 2.x device capability and the GPU configuration: 1 block with 16×16 threads. The thread $\langle 0, 4\rangle$ and the thread $\langle 15, 5\rangle$ belong to the same warp incur a write-write bank conflict when they both access `tile[threadIdx.y+i][threadIdx.x]`. Thread $\langle 0, 4\rangle$ accesses `tile[4][0]` and thread $\langle 15, 5\rangle$ accesses `tile[5][15]`, and those two array accesses can be flattened into one-dimensional access, that is, `tile[68]` and `tile[100]` respectively. Since the type of `tile` array is `float` and bank number is 32, as a result, those two accesses fail into the same bank: 4.

A.4 Comparing GKLEE and Random Testing

We comparing random testing and GKLEE (with path reduction) on our benchmark programs. We use the refined metrics Cov and CovBI (see Section 4) to give better measurement of the coverages. Table 7 gives the results: each item is reported of format c_1/c_2, where c_1 and c_2 represent the comparisons with respect to Cov and CovBI respectively. The values of c_1 and c_2 could be the following.

Relation	Meaning
$>$	better than, *i.e.* more cases report higher coverage
$<$	worse than, *i.e.* more cases report lower coverage

For example, the result of Red$_{BI}$ vs. Random is "$>$ / $>$", which indicates that the Red$_{BI}$ method beats the random one in terms

ψ	:=	`void`	void
	|	`i1, i2, ...`	bitvector
	|	`float`	float
	|	$[\mathrm{n} \times \phi]$	array
	|	$(\psi, \psi, \dots) \rightarrow \psi$	function
	|	$\psi *$	pointer
	|	$\{\psi, \psi, \dots\}$	struct
τ	:=	$\tau_{\_}$	unknown memory sort
	|	τ_l	local memory sort
	|	τ_s	shared memory sort
	|	τ_d	device memory sort
	|	τ_h	host memory sort
v	:=	`undef`	undefined value
	|	$n : \psi$	constant
	|	$@0 : \langle \psi, \tau \rangle, \dots$	global unamed variable
	|	$@id : \langle \psi, \tau \rangle$	global named variable
	|	$\%0 : \langle \psi, \tau \rangle, \dots$	unamed register
	|	$\%id : \langle \psi, \tau \rangle$	named register
	|	$tid, bid, bdim, gdim, \dots$	CUDA built-in variable
lab	:=	$l_1, l_2, \dots$	label
$instr$	:=	`br` $v,\ lab,\ lab$	conditional branch
	|	`br` lab	unconditional jump
	|	$v =$ `call` $(v, v, \dots)$	function call
	|	`ret` v | `ret void`	function return
	|	$v =$ `alloc` $\psi,\ n$	memory allocation
	|	$v =$ `getelptr` $v,\ \dots$	address calculation
	|	$v =$ `load` v	load
	|	`store` $v,\ v$	store
	|	$v =$ `binop` $v,\ v$	binary operation
	|	$v =$ `cast` $v,\ \psi$	type casting
	|	$v =$ `icmp` $v,\ v$	compare
	|	$v =$ `phi` $(v, lab),\ \dots$	ϕ-node for SSA
	|	$v =$ `syncthreads`	synchronization barrier
$block$	:=	$lab\ :\ instr,\ \dots,$ `br` .	basic block
$func$	:=	$fid(v, \dots)\ \{instr,\ \dots,\}$	function

Figure 10. Main syntax of LLVM$_{cuda}$ bytecode

of Cov and CovBI, *i.e.* Red$_{BI}$ reports higher coverage than the random one in more cases. Here Red$_{lin\_tid}$ (and Red$_{lin\_br}$) pick the threads (and branches) to explore fully in a linear manner. To make a fair comparison, when GKLEE produces n tests, the random testing picks n random inputs. Since the random testing method depends on the random seeds, and GKLEE's results depend on the drivers and the arguments given to the heuristics, we do not give exact, quantified numbers here. Yet the results show that GKLEE is able to beat the random testing method except for the cases where linear filtering is used to pick a subset of threads whose paths will be explored fully.

Compare	Random	Red BI	Red TB	Red lin_tid	Red lin_br
Random	—	< / <	< / <	< / >	< / <
Red$_{BI}$	> / >	—	< / <	> / >	> / >
Red$_{TB}$	> / >	> / >	—	> / >	> / >
Red$_{lin\_tid}$	> / <	< / <	< / <	—	< / <
Red$_{lin\_br}$	> / >	< / <	< / <	> / >	—

Table 7. Random testing against GKLEE with reduction heuristics. Item c_1/c_2 compares the method at a row with the one at a column with respect to Cov / CovBI.

A.5 LLVM$_{cuda}$

This section describes in more details the LLVM$_{cuda}$ language that GKLEE works on, with emphasis on CUDA-specific extensions. LLVM bytecode is a Static Single Assignment (SSA) based representation providing low-level operations for high-level languages like C/C++. It is the common code representation used throughout all phases of the LLVM compilation and optimization flow. LLVM$_{cuda}$ extends LLVM [12] to handle CUDA specific features.

Figure 10 shows LLVM$_{cuda}$'s main syntax (for brevity we do not present the attributes on variables, instructions and functions). The rest are handled as in KLEE. An LLVM program is collection of functions plus global variables; a function consists of a set of labeled blocks; and each block contains a list of instructions. In a function, an instruction can be identified with its block number and the label within this block. Binary instructions `binop` include `add`, `fadd`, `sub`, and so on. LLVM assumes unlimited registers. A register or variable can of bit-vector type, pointer type, *etc.* It may be cast into a pointer during address indexing.

One main extension in LLVM$_{cuda}$ is that a variable is attached with its memory sort τ indicating which memory in GPU it refers to. A variable $v : \langle \psi, \tau \rangle$ has data type ψ and memory sort τ. For example, $\%1 : \langle i32, \tau_s \rangle$ with value 1000 indicates that register $\%1$ is a 32-bit reference or pointer pointing at location 1000 in the shared memory. For a non-pointer and non-reference variable the memory sort information τ is not used.

To make the semantics more readable and easier to understand, we present the semantics rules by first showing the semantics for LLVM itself and then the CUDA extensions. In general, GKLEE performs symbolic execution according to these rules.

A.5.1 LLVM Semantics

The semantics of an instruction is modeled by a state transition. In a transition rule, notation $\mathbb{L}[\langle bk, l \rangle]$ gives the l^{th} instruction in block bk; $\mathbb{L}$ also maintains the function signatures and definitions, *e.g.* $fid(\dots) \in \mathbb{L}$ indicates that function fid is defined in the program. In a state $(\langle bk_{in}, l \rangle, \sigma, \Gamma, \mathbb{A})$, element l refers to the current program pointer. Component bk_{in} records the incoming block label which will be used to resolve phi instructions. Component σ models the memory that maps addresses to values. It consists of memory blocks each of which has a constant address and a constant size, and is an array of bytes. Γ models the stack frame: when a function is called, a new stack γ is attached to Γ to result in stack $\Gamma; \gamma$. When the function returns, the top stack γ is popped. Although LLVM allows allocating space either from the stack or the global memory, we do not distinguish them and model it always as allocating space from a common memory. To be faithful to LLVM's semantics, we use $\mathbb{A}$ to record the "stack" spaces allocated during a function call, and remove them from the memory upon the function exit.

For example, the execution of instruction br bk' updates the pc to $\langle bk', 0 \rangle$, stores the current block id in the bk_{in} field, and keeps other components unchanged. For an addition instruction $\%2 =$ `add i32 %0, %1`, the stack is updated by setting the value of $\%2$ to be the sum of $\%0$'s value and $\%1$'s value, and the pc is increased by 1. Here notations $\sigma[v]$ and $\sigma[v \mapsto k]$ indicate reading v's value from σ and updating v's value in σ to k respectively; $\Sigma \vdash v$ evaluates v's value over σ, *e.g.* $\sigma \vdash v_1 = v_2$ is true if $\sigma[v_1] = \sigma[v_2]$. We also use $[i : P_1 \mapsto v]$ to denote an array with domain satisfying P. For the stack frame Γ, the reads and writes are assumed to be at the top stack, *e.g.* $\Gamma[v]$ is an abbreviation of $\Gamma'; \gamma[v]$ for $\Gamma = \Gamma'; \gamma$. Now we elaborate some important issues.

Function Call. When a call instruction $v' =$ `call` $fid(v_0, \dots, v_n)$ is executed, a new stack is created and arguments $v_0, \dots, v_n$ are pushed into this stack by assigning v_i's value to the responding formal argument a_i. During the execution of the function body, all stack allocations are recorded in $\mathbb{A}$. When the `ret` instruction is called to exit the function body, the memory space by these allocations are released ($\sigma - \mathbb{A}$ gives the new σ by removed all allocations in $\mathbb{A}$). The top stack is popped from the stack frame, and the return value is written into the caller's stack (denoted by $\Gamma[ret \mapsto k]$), which will be fetched by the caller. The antecedent of the `call` rule includes the precondition that we be able to infer the

$$\frac{\mathbb{L}[\langle bk, l\rangle] = \texttt{br}\ \texttt{bk}'}{(\langle bk, l\rangle, bk_{in}, \sigma, \Gamma, \mathbb{A}) \to (\langle bk', 0\rangle, bk, \sigma, \Gamma, \mathbb{A})}$$

$$\frac{\mathbb{L}[\langle bk, l\rangle] = \texttt{br}\ v,\ bk_1,\ bk_2 \qquad \Gamma \vdash v}{(\langle bk, l\rangle, bk_{in}, \sigma, \Gamma, \mathbb{A}) \to (\langle bk_1, 0\rangle, bk, \sigma, \Gamma, \mathbb{A})}$$

$$\frac{\mathbb{L}[\langle bk, l\rangle] = \texttt{br}\ v,\ bk_1,\ bk_2 \qquad \Gamma \vdash \neg v}{(\langle bk, l\rangle, bk_{in}, \sigma, \Gamma, \mathbb{A}) \to (\langle bk_2, 0\rangle, bk, \sigma, \Gamma, \mathbb{A})}$$

$$\frac{\mathbb{L}[\langle bk, l\rangle] = (v' = \texttt{binop}\ v_1,\ v_2)}{\begin{array}{l}(\langle bk, l\rangle, bk_{in}, \sigma, \Gamma, \mathbb{A}) \to \\ (\langle bk, l+1\rangle, bk_{in}, \sigma, \Gamma[v' \mapsto \Gamma[v_1]\ \texttt{binop}\ \Gamma[v_2]], \mathbb{A})\end{array}}$$

$$\frac{\mathbb{L}[\langle bk, l\rangle] = (v' = \texttt{phi}\ dsts) \qquad (v, bk_{in}) \in dsts}{(\langle bk, l\rangle, bk_{in}, \sigma, \Gamma, \mathbb{A}) \to (\langle bk, l+1\rangle, bk_{in}, \sigma, \Gamma[v' \mapsto \Gamma[v]], \mathbb{A})}$$

$$\frac{\begin{array}{c}\mathbb{L}[\langle bk, l\rangle] = (v' = \texttt{call}\ fid(v_1, \ldots, v_n)) \\ fid(a_1, \ldots, a_n) \in \mathbb{L} \\ \begin{array}{l}(\langle bk, l\rangle, bk, \sigma, (\Gamma; [a_1 \mapsto v_1, \ldots, a_n \mapsto v_n]), \{\}) \to \\ (\langle bk', l_{undef}\rangle, \sigma', \Gamma[ret \mapsto k], \{\})\end{array}\end{array}}{\begin{array}{l}(\langle bk, l\rangle, bk_{in}, \sigma, \Gamma, \mathbb{A}) \to \\ (\langle bk, l+1\rangle, bk_{in}, \sigma', (\Gamma - \{ret\})[v' \mapsto k], \mathbb{A})\end{array}}$$

$$\frac{\mathbb{L}[\langle bk, l\rangle] = \texttt{ret}\ v}{(\langle bk, l\rangle, bk_{in}, \sigma, \Gamma; \gamma, \mathbb{A}) \to (\langle bk, l_{undef}\rangle, (\sigma - \mathbb{A}), \Gamma[ret \mapsto \gamma[v]], \{\})}$$

$$\frac{\begin{array}{c}\mathbb{L}[\langle bk, l\rangle] = (v = \texttt{alloc}\ \psi,\ n) \\ a' = \texttt{sum}(\{\texttt{sizeof}(k) \mid \langle a, k\rangle \in \sigma\})\end{array}}{\begin{array}{l}(\langle bk, l\rangle, bk_{in}, \sigma, \Gamma, \mathbb{A}) \to \\ (\langle bk, l+1\rangle, bk_{in}, \sigma[a' \mapsto [(i : 0 \le i < n \times \texttt{sizeof}(\psi)) \ \mapsto\ 0]], \\ \Gamma[v \mapsto k'], \mathbb{A} \cup \{v'\})\end{array}}$$

$$\frac{\mathbb{L}[\langle bk, l\rangle] = (v' = \texttt{getelptr}\ v_1, \ldots, v_n)}{\begin{array}{l}(\langle bk, l\rangle, bk_{in}, \sigma, \Gamma, \mathbb{A}) \to \\ (\langle bk, l+1\rangle, bk_{in}, \sigma, \Gamma[v' \mapsto \Gamma[v_1] + \cdots + \Gamma[v_n]], \mathbb{A})\end{array}}$$

$$\frac{\begin{array}{c}\mathbb{L}[\langle bk, l\rangle] = (v' : \psi = \texttt{load}\ v) \\ \exists [a \mapsto k] \in \sigma : a \le v + \texttt{sizeof}(\psi) < a + \texttt{size}(k)\end{array}}{\begin{array}{l}(\langle bk, l\rangle, bk_{in}, \sigma, \Gamma, \mathbb{A}) \to \\ (\langle bk, l+1\rangle, bk_{in}, \sigma, \Gamma[v' \mapsto k[\Gamma[v] - a]], \mathbb{A})\end{array}}$$

$$\frac{\begin{array}{c}\mathbb{L}[\langle bk, l\rangle] = (\texttt{store}\ v_1,\ v_2) \\ \exists [a \mapsto k] \in \sigma : a \le v + \texttt{sizeof}(\psi) < a + \texttt{size}(k)\end{array}}{\begin{array}{l}(\langle bk, l\rangle, bk_{in}, \sigma, \Gamma, \mathbb{A}) \to \\ (\langle bk, l+1\rangle, bk_{in}, \sigma[a \mapsto (k[\Gamma[v] - a \mapsto \sigma[v_1]])], \Gamma, \mathbb{A})\end{array}}$$

Figure 11. Main operational semantics rules of LLVM

entire computation of the function body in the extended environment $\Gamma; [a_1 \mapsto v_1, \ldots, a_n \mapsto v_n]$. For an external function call, the inputs are copied into the physical memory, then the function is executed using LLVM's JIT support, finally the ouptut is copied back to the VM.

Individual Memory. GKLEE uses a block based memory model, where a memory is organized as a set of blocks, each of which has a concrete address and an array of bytes recording the value. A memory object of known size (*e.g.* a LLVM struct or array) is allocated a block of memory. In the following example, memory object 1 locates at address a_1 and contains m bytes; memory object 2 is at address $a_1 + m$ containing n bytes. They are also allocated in the physical memory as in the VM; hence there exists no overlapping. The free space starts from $a_1 + m + n$. In the load instruction rule, the address is first calculated, then the value in the memory object is read ($k[\Gamma[v] - a]$ reads $\texttt{sizeof}(\psi)$ from array k starting from address $\Gamma[v] - a$). When the address is not fixed (*e.g.* a symbolic pointer), then this address will be tested on each possible address, and each possible match create a branch case and a new state is spawned. Hence GKLEE does not abstract pointers or arrays, and enumerates all posibilities of pointer aliasing.

address	$\texttt{mo}_1 : a_1$	$\texttt{mo}_2 : a_1$+m	a_1+m+n
value	$(b_0, \ldots, b_{m-1})$	$(b_0, \ldots, b_{n-1})$	

A.5.2 CUDA Related Semantics

The main issues on modeling CUDA include:

- **The architecture of GPU.** In particular, CUDA has a specific memory hierarchy as shown in Figure 5.
- **Thread scheduling.** In particular, GKLEE performs canonical scheduling to model CUDA's SIMD computations. This avoids interleaving blow-up.

In LLVM$_{cuda}$, the state Φ is extended to consisting of a data state Σ and a PC $\mathbb{P}$, where Σ contains the entire memory hierarchy and $\mathbb{P}$ records the pcs of all threads. Thread t's pc is given by $\mathbb{P}[t]$. In this section we hide the details of the stack frame Γ and heap allocation record $\mathbb{A}$ described in Section A.5.1 and use M to compactly denote the various components $\langle \sigma, \Gamma, \mathbb{A}\rangle$. We also reuse the notation σ to refer to the local store consisting of the stack and local memory, *i.e.* thread i's local store maps its local variables to values. A value consists of one or more bytes (our model has byte-level accuracy). We also use a single unique label l (rather than a block id and an offset within the block) to represent the label of an instruction. These simplifications allows us to skip unimportant details and present when only the core semantics about CUDA.

Program	:=	$\mathbb{L} \subset lab \mapsto instr$
Value	:=	$\mathbb{V} \subset \texttt{byte}^+$
Shared state	:=	$\mathbb{M} \subset (bid \mapsto M) \times M \times M$
Local state	:=	$\sigma \subset var \mapsto \mathbb{V}$
Data State	:=	$\Sigma \subset (tid \mapsto \sigma) \times \mathbb{M}$
Program counter	:=	$\mathbb{P} \subset tid \mapsto lab$
State	:=	$\Phi \subset \Sigma \times \mathbb{P}$

GKLEE models CUDA's memory hierarchy in a symbolic state as in Figure 5: each thread has its own local store ($tid \mapsto \sigma$); the threads in a block shares a shared memory ($bid \mapsto M$); and all blocks share the device memory and the CPU memory ($M \times M$). Each thread has a program counter (pc) recording the label of the current instruction. Now read operation $\Sigma[v]$ and write operation $\Sigma[v \mapsto k]$ occur in the appropriate memory/store according to v's memory sort, *e.g.* if v's sort is τ_s then the shared memory of the current block is accessed.

Scheduling. GKLEE uses canonical scheduling to execute the instructions while the GPU exhibits non-determinism. We have shown that, if not races occur, such scheduling suffices to model the non-deterministic executions of the threads even under relaxed memory models. Besides non-deterministic semantics, we also present the semantics with respect to the canonical scheduling. Recalled that $(\Sigma, \mathbb{P})$ records the state and the pcs; we also write $(\Sigma, \mathbb{P}, t)$ to indicate that the current thread is t in the canonical schedule. The canonical scheduling is defined by how this t changes.

The basic transition is a local transition made by any thread t: in Section A.5.1 we use $(l, \Sigma) \to (l', \Sigma')$ to depict the execution of a LLVM instruction at label l, which updates the pc to l' and the data state to Σ'. We denote it as $(\Sigma, \mathbb{P}) \hookrightarrow_t (\Sigma', \mathbb{P}[t \mapsto l'])$. In general, a thread is free to make local transitions until synchronization points are reached. CUDA has implicit barriers in an intra-warp and may have explicit barriers in a block.

- **Intra-warp scheduling**. For each instruction, GKLEE schedules the threads within a warp sequentially until all threads finishing execution, then the next instruction starts. The following rule enforces that thread t executes only when the threads in the same warp has just executed or is about to execute the same instruction. Another rule (not shown here) is to schedule a divergent warp to sequentially execute its two diverged parts.

[intra_warp_schd$_1$]:

$$\frac{(l, \Sigma) \rightarrow (l', \Sigma') \quad \forall t' \in \texttt{warp\_of}(t) : \mathbb{P}[t'] \in \{l, l'\}}{(\Sigma,\, \mathbb{P}) \hookrightarrow_t (\Sigma',\, \mathbb{P}[t \mapsto l'])}$$

As mentioned in Section 3.2, GKLEE also supports using a slightly different rule which does not require all the threads are at the same instruction so that a divergent warp needs not to be executed twice (note that, in the hardware GPU, two divergent threads won't execute the same instruction at the same time); instead the threads are executed in a round-robin manner, and at the end of a warp the first thread is the next one to execute. Note that thread id t is global (as in Figure 6).

[intra_warp_schd$_2$]:

$$\frac{\mathbb{L}[l] \neq \texttt{syncthreads} \qquad (l, \Sigma) \rightarrow (l', \Sigma') \quad t' \doteq (t\%\, 32 = 31)\ ?\ t - 31 : t + 1}{(\Sigma, \mathbb{P}, t) \hookrightarrow_t (\Sigma',\, \mathbb{P}[t \mapsto l'], t')}$$

Consider the following example, suppose t_1 and t_2 diverge on condition c, then the GPU hardware and Rule intra_warp_schd$_1$ will execute the c and then the $\neg c$ part (or in the reverse order). Rule intra_warp_schd$_2$ schedules $t_1 : instr_1 \hookrightarrow t_2 : instr_1 \hookrightarrow t_1 : instr_2 \hookrightarrow t_2 : instr_2$, where each instruction is executed if its guarded condition holds. In the race-free case this schedule is equivalent to the one by intra_warp_schd$_1$.

t_1	t_2
br c, l_1 l_2	br c, l_1 l_2
$l_1 : instr_1$	$l_1 : instr_1$
$l_2 : instr_2$	$l_2 : instr_2$

- **Inter-warp scheduling**. Warps in a block will synchronize with each other only at explicit syncthreads barriers. The following rule specifies that a thread upon a barrier can proceed to the next instruction only after all the threads in the same block have reached the barrier. Note that this rule does not specify the order in which the threads execute the barrier.

$$\frac{\mathbb{L}[l] = \texttt{syncthreads} \quad \forall t' \in \texttt{blk\_of}(t)\ :\ \mathbb{P}[t'] \in \{l, l+1\}}{(\Sigma,\, \mathbb{P}) \hookrightarrow_t (\Sigma,\, \mathbb{P}[t \mapsto l+1])}\ \text{inter\_warp\_schd}_1$$

The canonical schedule executes the $i + 1^{th}$ block after the i^{th} block: the first thread in block $i+1$ executes right after the last thread in block i (here $bdim$ denotes the total number of threads in a block).

[inter_warp_schd$_2$]:

$$\frac{\mathbb{L}[l] = \texttt{syncthreads} \quad t' \doteq ((t+1)\%\, bdim = 0)\ ?\ t - bdim : t + 1}{(\Sigma, \mathbb{P}, t) \hookrightarrow_t (\Sigma',\, \mathbb{P}[t \mapsto l+1], t')}$$

- **Block scheduling**. This case is similar to inter-warp scheduling excepts that the threads won't synchronize at syncthreads. Instead other global barriers (*e.g.* the implicit one at the end of a kernel) apply.

Memory Sort. After a source program is compiled into LLVM bytecode, it is not straight-forward to determine which memory is used when an access is made because the address of this access may be calculated by multiple bytecode instructions. We employ a simple GPU-specific memory type inference method by computing for each (possibly symbolic) expression a sort τ which is either $\tau_{\_}$ (unknown), τ_l (local), τ_s (shared), τ_d (device), or τ_h (host), as per the rules in Figure 12. In our experience, these rules have been found to be sufficiently precise on all the kernels we have applied GKLEE to.

$$\frac{\mathbb{L}[l] = (v = \texttt{alloca}\ \psi, n)}{(\Sigma,\, \mathbb{P}) \hookrightarrow_t (\Sigma[(v : \tau_l) \mapsto 0^{n \times \texttt{sizeof}(\psi)}],\, \mathbb{P}[t \mapsto l+1])}$$

$$\frac{\mathbb{L}[l] = (v = \texttt{getelptr}\ (v_1 : \tau),\ v_2\ \ldots\ v_n)}{(\Sigma,\, \mathbb{P}) \hookrightarrow_t (\Sigma[v : \tau \mapsto \Sigma[v_1] + \Sigma[v_2] + \cdots + \Sigma[v_n]],\, \mathbb{P}[t \mapsto l+1])}$$

$$\frac{\mathbb{L}[l] = (v = \texttt{binop}\ (v_1 : \tau),\ (v_2 : \tau_{\_}))}{(\Sigma,\, \mathbb{P}) \hookrightarrow_t (\Sigma[v : \tau \mapsto \texttt{binop}(\Sigma[v_1], \Sigma[v_2])],\, \mathbb{P}[t \mapsto l+1])}$$

$$\frac{\mathbb{L}[l] = (v = \texttt{binop}\ v_1,\ v_2) \qquad \tau_{\_} \notin \{v_1, v_2\}}{(\Sigma,\, \mathbb{P}) \hookrightarrow_t (\Sigma[v : \tau_{\_} \mapsto \texttt{binop}(\Sigma[v_1], \Sigma[v_2])],\, \mathbb{P}[t \mapsto l+1])}$$

$$\frac{\mathbb{L}[l] = (v_2 = \texttt{load}\ v_1 : \tau) \qquad \tau \neq \tau_{\_}}{(\Sigma,\, \mathbb{P}) \hookrightarrow_t (\Sigma[v_2 : \tau_{\_} \mapsto \Sigma[v_1]],\, \mathbb{P}[t \mapsto l+1])}$$

$$\frac{\mathbb{L}[l] = (\texttt{store}\ v_1,\ v_2 : \tau) \qquad \tau \neq \tau_{\_}}{(\Sigma,\, \mathbb{P}) \hookrightarrow_t (\Sigma[v_2 : \tau \mapsto \Sigma[v_1]],\, \mathbb{P}[t \mapsto l+1])}$$

$$\frac{v : \tau_{\_} \qquad ((v' : \tau') \mapsto k)\ \in\ \Sigma \qquad \Sigma\ \vdash\ v' \leq v \leq v' + \texttt{sizeof}(k)}{v : \tau'}$$

Figure 12. Main memory operation rules with sort inference in LLVM$_{cuda}$

For example, instruction alloca allocates n elements of type ψ from the local memory, the sort of the address v is τ_l, indicating that it refers to a block in the local store. A getelptr instruction calculates the address by adding the offsets $v_2, \ldots, v_n$ to basic address v_1, the final address v points to the same memory as v_1. Address calculation may also be performed through data processing instructions, *e.g.* a binop instruction returns an address of sort τ when one of its operands has sort τ (thus being the base address) and the other one is not associated with a known memory. If both operands have known sorts, then the address calculation fails in identifying the sort of the calculated address; and the search rule will be applied to locate the right memory.

A load or store instruction can be executed only if the address sort is known; and the value loaded from memory has unknown sort. The last rule says that a valid sort τ' is found for v if there exists a memory object associated with v' in memory τ' such that v's value falls within this object. Basically it searches the memory hierarchy to locate the target memory when the previous analysis fails to find v's sort. It is the extended version of KLEE's method to resolve pointer aliasing, which addresses CUDA's memory hierarchy. If v represents a pointer which can refer to multiple objects (determined by SMT solving), then multiple states are generated, each of which needs to apply this rule. This often reveals memory sort related bugs in the source kernel, *e.g.* mixing up the CPU and GPU memory. We plan to use Clang's ongoing support for LLVM +CUDA [12] to simplify or even eliminate such inference.