# PRACTICAL SYMBOLIC EXECUTION ANALYSIS AND METHODOLOGY FOR GPU PROGRAMS

by

Peng Li

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2015

The University of Utah Graduate School

## STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Peng Li**

has been approved by the following supervisory committee members:

| | | |
|---|---|---|
| **Ganesh Gopalakrishnan** | , Chair | **10/28/2014**<br>Date Approved |
| **John Regehr** | , Member | **10/01/2014**<br>Date Approved |
| **Matthew Might** | , Member | **10/01/2014**<br>Date Approved |
| **Zvonimir Rakamarić** | , Member | **10/01/2014**<br>Date Approved |
| **Guodong Li** | , Member | **10/02/2014**<br>Date Approved |

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

# ABSTRACT

Graphics processing units (GPUs) are highly parallel processors that are now commonly used in the acceleration of a wide range of computationally intensive tasks. GPU programs often suffer from data races and deadlocks, necessitating systematic testing. Conventional GPU debuggers are ineffective at finding and root-causing races since they detect errors with respect to the specific platform and inputs as well as thread schedules. The recent formal and semiformal analysis based tools have improved the situation much, but they still have some problems. Our research goal is to aply scalable formal analysis to refrain from platform constraints and exploit all relevant inputs and thread schedules for GPU programs. To achieve this objective, we create a novel symbolic analysis, test and test case generator tailored for C++ GPU programs, the entire framework consisting of three stages: GKLEE, GKLEE$_p$, and SESA. Moreover, my thesis not only presents that our framework is capable of uncovering many concurrency errors effectively in real-world CUDA programs such as latest CUDA SDK kernels, Parboil and LoneStarGPU benchmarks, but also demonstrates a high degree of test automation is achievable in the space of GPU programs through SMT-based symbolic execution, picking representative executions through thread abstraction, and combined static and dynamic analysis.

To my parents and little sister, for their persistent love, support, and encouragement

# CONTENTS

# LIST OF TABLES

## ACKNOWLEDGEMENTS

The 7-year scholarly pursuit of a Ph.D. in Utah has been the greatest challenge in my life. I would like to gratefully thank all the people that have influenced this endeavor.

First, I am deeply indebted to my advisor, Ganesh Gopalakrishnan, for his invaluable research advice, great patience, and financial support. As a venerable mentor, he spent innumerous time guiding me to do research independently and effectively, answering my questions, keeping tracking of my research direction and teaching me to prepare presentations and write technical papers properly. I am fortunate and grateful to be supervised by Ganesh; without his guidance, this dissertation would never be completed.

I also thank my thesis committee members: John Regehr, Zvonimir Rakamarić, Matt Might, and Guodong Li. They have all eagerly provided me with advice on both my research and career. I am especially indebted to Guodong Li, who is my research collaborator as well as the research "elder brother"; I benefited much from his insightful guidance and ideas, and he initialised GKLEE, which forms a basis for my research.

I want to thank Indradeep Ghosh, who supervised me at Fujitsu Labs of America. He introduced me an interesting project named KLOVER, and his wisdom and knowledge about program analysis, software reliability, and software security helped me much.

I also want to thank Vinod Grover, who supervised me throughout my internship at NVIDIA. His wisdom about program analysis of CUDA programs helped me overcome many problems. I really spent a great summer working with him.

I thank all my computer science fellow friends: Yang Chen, Xuejun Yang, Jianjun Duan, Lu Zhao, Zhisong Fu, Qingyu Meng, etc., and all GAUSS members: Geof Sawaya, Sriram Aananthakrishnan, etc.

Finally, I owe many thanks to my parents, Jianxin Li and Guilan Chen, and my little sister, Yaqi Li. Their unceasing love, support, and encouragement are the main driving force for me to accomplish my PHD; this dissertation is lovely dedicated to them.

# CHAPTER 1

# INTRODUCTION

This thesis is about the techniques for building more reliable software, especially the GPU software, which is easier to induce errors but more difficult to guarantee the correctness. To enable practical tools, we require scalable testing techniques that are capable of reasoning about real programs. This thesis examines the problem of precise and scalable formal analysis techniques in the context of data-parallel GPU programs and presents the first dynamic symbolic analysis, test, and test case generator tailored for GPU programs.

## 1.1 Motivation

Modern GPUs are parallel many-core processors designed for throughput processing. GPUs are widely used as parallel coprocessors under the control of a host CPU in a heterogeneous system. The host CPU is responsible for copying input data to the GPU, launching parallel programs on the GPU and fetching resulting data back from the GPU, and the compute-intensive part of an application is always offlined to GPUs. A typical GPU consists of multiple multiprocessors each made up of many simple cores. Each multiprocessor executes instructions in a SIMD fashion where multiple cores execute the same instruction on different data.

At present, GPUs are making inroads into virtually many aspects of our daily lives, ranging from GPU-based mobile chips [49], image processing and scientific computing [21], and system software accelaration [47, 48] to high performance computing [50]. In top 500 [51] of July 2012 [1], there are 52 supercomputers built upon NVidia GPUs. In order to accomodate various GPU architectures, many GPU-based programming models have been proposed, such as NVidia CUDA [17], OpenCL [38], and C++ AMP [12], etc. In particular, NVidia CUDA (Compute Unified Device Architecture) is drawing widespread interest, and C/C++ CUDA programs are being developed and used extensively, so improving the reliability of CUDA programs becomes a critical issue.

Random testing is the most prominent method to improve software reliability in industry; however, it always detects errors with respect to the specific platform and execution path as well as the thread schedule. In this thesis we focus on *formal analysis based testing* to improve CUDA programs' correctness. Testing is expected to be exponential. For example, testing for a program is expected to be exerted with respect to all platforms. In sequential programs testing is supposed to exploit all feasible program paths, while in concurrent scenarios it is complete to take into account of all thread schedules. Our testing goals are to employ formal analysis based testing to refrain from porting issues (i.e., platform constraints) while exploiting all execution paths and thread schedules.

## 1.2 CUDA

A CUDA program is a hybrid of host code and kernel code, host code is executed sequentially in the CPU, and kernel code is executed with the parallel SIMD mode in the GPU. A CUDA kernel is launched as a 3D *grid* of *thread blocks*. The total size of a 3D grid is `gridDim.x` $\times$ `gridDim.y` $\times$ `gridDim.z`. Each block has block ID $\langle$ `blockIdx.x`, `blockIdx.y`, `blockIdx.z` $\rangle$ and contains `blockDim.x` $\times$ `blockDim.y` $\times$ `blockDim.z` threads, each of which has thread ID $\langle$ `threadIdx.x`, `threadIdx.y`, `threadIdx.z` $\rangle$. For brevity, we use $gdim$ to denote $gridDim$, $bid$ for $blockIdx$, $bdim$ for $blockDim$, and $tid$ for $threadIdx$. The constraints $bid.* < gdim.*$ for $* \in \{x, y, z\}$ and $tid.* < bdim.*$ for $* \in \{x, y, z\}$ always hold. Figure 1.1 presents a simple CUDA program containing $3 \times 2$ blocks, each of which consists of $4 \times 3$ threads. Threads within the same block 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. Each thread uses its own local memory and register. 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.

## 1.3 The Difficulty of Debugging CUDA Programs

CUDA programs are extensively used, but programming CUDA programs to achieve high performance often requires many intricate optimizations involving memory bandwidth and the CPU/GPU occupancy. Most of these optimizations are being carried out manually,

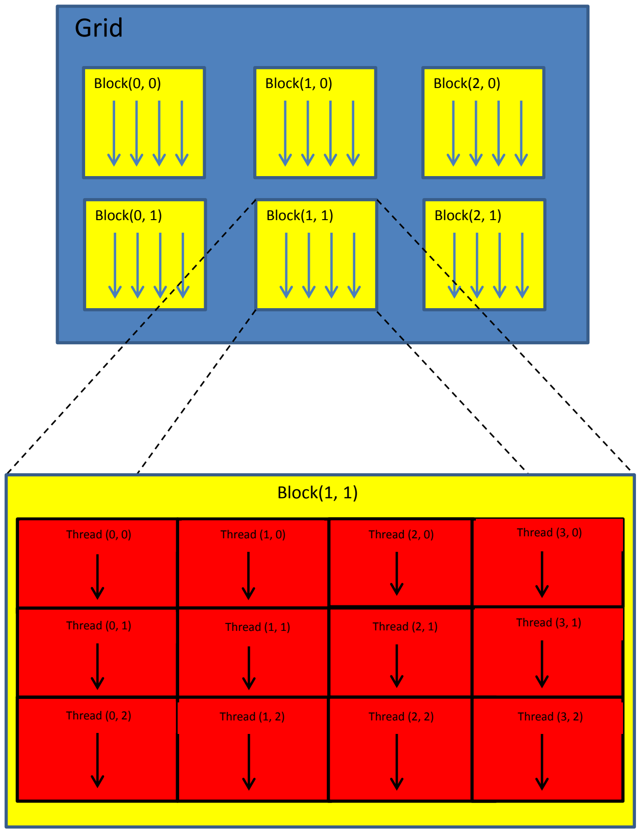

**Figure 1.1**: Grid of thread blocks, adapted from [18].

and due to the complexity of these optimizations in the context of actual problems, designers routinely introduce insidious correctness and performance bugs. Conventional GPU debuggers [34–36] are ineffective at finding and root-causing races that are dependent on thread schedules and inputs. 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) choosing the appropriate thread interleaving, (iii) ability to observe data corruption (and be able to reliably attribute it to races), (iv) ability to demonstrate all possible thread/warp schedules in the specific platform, and (v) ability to diagnose defects.

## 1.4 Improving Correctness of GPU Kernels

In the following we review techniques for improving correctness of GPU kernels. We broadly classify the techniques as *dynamic* or *static*. Dynamic techniques execute programs on concrete test cases to derive information about the executions of the program, whereas static techniques use program verification to reason about all executions of the program without having to run it.

### 1.4.1 Dynamic Analysis with Code Instrumentation

Given a parallel program, dynamic instrumentation works by instrumenting and executing the program to dynamically record all memory accesses. Then, after running the program, we can determine if the execution has a data race by examining the logs for conflicting accesses. This technique is simple to apply and races that are uncovered are true bugs (no false positives) because the program is executed concretely. The main weakness of dynamic instrumentation is that it cannot guarantee the completeness (completeness means no false negatives or omissions) since it only checks a single path (out of many). Tools that use dynamic instrumentation include work by Boyer et al. [10], CUDAMEMCHECK [35] and GRace [54], the GRace tool additionally uses static analysis to reduce the number of memory access pairs for further dynamic analysis.

## 1.5 Static Verification

Unlike dynamic techniques, static verification can offer guarantees of completeness. However, a recurring weakness of these techniques is the need for invariants to refrain from false positives. An invariant is a property that always holds at a particular program point (e.g., that a variable is always greater than 0).

The PUG tool [28] verifies GPU kernels for race-freedom through encoding thread interleavings and translating CUDA kernels to verification conditions in SMT format. Given a CUDA program, PUG generates a verification condition that reflects the accesses of the program from the point of view of two symbolic (arbitrary) threads. PUG uses the canonical schedule (a sequential thread schedule) to reduce the number of thread schedules that must be considered. PUG also uses the idea of barrier intervals to incrementally consider one barrier interval at a time. PUG addresses the problem of generating loop invariants by loop normalization, overapproximation and invariant finding. The tool recognises certain loop patterns and automatically encode invariants into the verification condition. The tool allows the programmer to annotate loops with invariants.

GPUVerify [8, 14] verifies race- and divergence-freedom of GPU kernels, which are written in mainstream GPU programming languages such as CUDA and OpenCL. At a high-level, GPUVerify is similar to PUG. The main difference is the technique used to generate verification conditions. GPUVerify also exploits the property that race-freedom is a pairwise property. Rather than directly encoding the kernel as a verification condition, GPUVerify translates the kernel into a sequential program that models the behavior of an arbitrary pair of threads; the correctness of this program implies the race-freedom of the given kernel. However, GPUVerify suffers from the problem of precise reasoning for *data-dependent* GPU kernels, whose control- or data-flow is dependent on shared state. Barrier invariants, manually written invariants with respect to *data-dependent* kernels, enable precise reasoning while retaining the two-thread reduction necessary for scalable verification. But barrier invairants are difficult to be written unless the kernel behaviors are clear to users, and ambiguous invariants will lead to incorrect verification results.

## 1.6 Hybrid Technique

A hybrid technique uses both dynamic and static techniques in conjunction. Work by Leung et al. [27] has applied test amplification to GPU kernels. Test amplification is to use single dynamic run to learn much more information about a program's behavior. First, a CUDA kernel is instrumented and the behavior of the kernel with some fixed test input and under a particular thread interleaving is logged. Second, they compute if the kernel is access invariant: the input variables of a kernel will not actually flow-to or affect the integrity of the variables appearing in the property to be verified. A taint analysis conservatively tracks the inputs of the kernel and ensures that (i) no tainted variable is used in the address computation of any memory access and (ii) no memory access is control-dependent on a tainted variable. Finally, if a kernel is access invariant, then race-freedom can be amplified to all executions of the program. The main advantage of this technique is that it combines the complementary strengths of dynamic and static approaches. However, test amplification is inapplicable for access variant kernels, which exist extensively.

### 1.6.1 Dynamic Symbolic Execution

Dynamic symbolic execution uses symbolic execution to explore the executions of a program. Under symbolic execution, the program is executed in a symbolic virtual machine (VM) where program variables can be marked as concrete or symbolic. During execution the VM can fork execution paths whenever it encounters a nondeterministic situation (e.g., a conditional where both choices can be true or dereferencing a symbolic pointer that may point to multiple objects). Execution can be forced along a particular path by applying constraints over symbolic variables. For example, if execution reaches an if-statement with condition $(x == 0)$ where x is an unconstrained symbolic variable then we will explore execution paths where the condition holds $(x == 0)$ and not explore paths where $(x \neq 0)$.

The main advantage of this technique is that it executes the program and so does not suffer from false positives. Additionally, if a bug is uncovered (e.g., an assertion failure or a null pointer dereference), then the constraints over symbolic variables (called the path condition) can be used to automatically generate a concrete test case as the witness.

Tools that have applied dynamic symbolic execution to GPU kernels include KLEE-FP [15] and KLEE-CL [16]. KLEE-FP [15] presents an effective technique to cross-check an IEEE 754 floating-point program against its SIMD-vectorized version. The key insight behind their approach is that floating-point values are only reliably equal if they are essentially built by the same operations. As a result, their technique works by lowering the Intel Streaming SIMD Extension (SSE) instruction set to primitive integer and floating-point operations and then using an algorithm based on symbolic expression matching augmented with canonicalization rules. In addition, the analogue technique is used by KLEE-CL [16] to detect races and help cross-check OpenCL code against sequential code.

## 1.7 Thesis Statement

We always aim at building effective yet practical analysis and methodology for GPU programs, and the framework we create to realize this objective must be precise, fully automatic and complete as much as possible. To achieve the objective, we construct the first symbolic analysis, testing and test case generator tailored for GPU programs. Since CUDA is the most prominent GPU programming model, our framework targets CUDA programs, this framework supports a significant subset of CUDA/C++ and is able to uncover many concurrency errors effectively in real-world CUDA programs. In addition, our thesis statement is summarized as follows:

*A high degree of test automation is achievable in the space of GPU programs through SMT-based symbolic execution, picking representative executions through thread abstraction and combined static and dynamic analysis.*

## 1.8 Research Outline

The entire symbolic analysis framework for CUDA programs is realized through three stages:

First, we implemented GKLEE [31], which is the first symbolic analyser and test generator tailored for C++ CUDA programs. GKLEE analyzes whole programs, which means that it verifies the "`main`" (CPU) program together with the collection of GPU kernel functions that it calls (we call these functions "GPU kernels"). GKLEE extends KLEE [13] in many ways. While KLEE provides the basic capabilities for sequential program analysis, GKLEE ("GPU + KLEE") extends KLEE to provide self-contained and powerful facilities for analyzing GPU programs. GKLEE executes the programmer's CUDA application in a symbolic environment. With the user declared symbolic assignment of program variables, GKLEE can execute through all different branches in the code where the predicates are based on the symbolic variables. These executions output as test cases, with concrete values substituted for the symbolic ones. GKLEE fully executes the CUDA program, including the kernel portion, closely following CUDA semantics, and using a canonical schedule fully respecting warp-based SIMD execution (which is proven to be sound for race detection).

Second, GKLEE encounters a major drawback of all the semiformal tools described so far: *these tools model and solve the data-race detection problem over the explicitly specified number of GPU threads*. This makes these tools difficult to apply in many situations in supercomputing where many program modules (*e.g.*, library modules) often assume a certain minimum number of threads to be involved, where these minimum numbers themselves are very large. To address this problem, we provide an extension of GKLEE that exploits thread symmetry and provides a way to analyze GPU programs containing large (bounded) numbers of threads in real kernels. In a nutshell, our method partitions the space of executions of a GPU program into *parametric flow equivalence classes* (PFE) and models the race analysis problem over two parametric threads in one PFE equivalence class. This analysis method over parametric flows has been implemented in a new version of GKLEE called GKLEE$_p$ [32].

Last but not least, GKLEE$_p$ still requires users to pick the symbolic inputs. Choosing too many symbolic inputs can result in slower symbolic evaluation while choosing less might miss important errors. In addition, GKLEE$_p$ still suffers from search explosion: even if only two symbolic threads are considered, each thread may create a large number of flows or symbolic states. For instance, if a thread contains $n$ feasible branches, then $O(2^n)$ flows or paths may be generated. To resolve those problems, we present a new tool SESA [33] (Symbolic Executor with Static Analysis) that significantly improves over prior tools in its class both in terms of new ideas and new engineering:

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

## 1.9 Summary of Contributions

After comparing against all aforesaid formal analysis tools with GKLEE in Table 1.1, the main contributions of this dissertation are highlighted as follows:

- The automation of GPU programs testing is achieved through SMT-based symbolic execution; we built the first symbolic analyser and test generator tailored for CUDA programs: GKLEE. In GKLEE, the hierarchical memory model is supported fully and accurately, a novel sequential canonical thread schedule, which is sound for detecting races, is provided to improve GKLEE's scalability, and the SMT-based race detection is applied to uncover races effectively.

**Table 1.1**: Comparison of formal analysis and testing frameworks of GPU programs.

| **Comparison Categories** | Boyer et al. [10] | PUG [28] | GRace [54] | KLEE-FP [15] KLEE-CL [16] | TestAmp [27] | GPUVerify [8, 14] | GKLEE [31] GKLEE$_p$ [32] SESA [33] |
|---|---|---|---|---|---|---|---|
| Methodology | Dynamic Check | Symbolic Static Analysis | Static Analysis + Dynamic Check | Dynamic Symbolic Execution | Static Analysis + Dynamic Check | Static Verification + Barrier Invarient | Dynamic Symbolic Execution [31] + Parametric Flow [32, 33] + Static Analysis [33] |
| Target Program | CUDA | CUDA | CUDA | IEEE 754 floating-point program OpenCL | CUDA | CUDA OpenCL | CUDA |
| Level of Analysis | Source Code (Instrument.) | Source Code | Source Code (Instrument.) | LLVM Bytecode | Source code (Instrument.) | LLVM Bytecode Boogie | LLVM Bytecode |
| Bugs Targeted | Shared Mem. Race, Bank Conflicts | Shared Mem. Race, Deadlocks, Bank Conflicts | Intra-/Inter- Warp Race | Race, Functional Correctness, Memory Errors | Race | Race Deadlock | Race (intra-/inter- warp, all memory), Memory Errors, Deadlocks, Warp Divergence, Memory Coalesce, Bank Conflicts, Compilation level bugs (e.g. Volatiles) |
| False alarm elim. | Auto./Manual Refinement | Dynamic Execution | Dynamic Execution | SMT-solving | Dynamic Execution | SMT Solving, Barrier Invariant | SMT-solving, GPU replaying |
| Test Generation | Not supported | Not supported | Not supported | Automatic | Not supported | Not Supported | Automatic, Hardware Execution, Coverage Measures, Test Reduction |

- Thread abstraction is proposed and used to further improve scalability and simplify race detection significantly. GKLEE$_p$ is an important extension to implement the thread abstraction.
- Symbolic execution can take advantage of static analysis in terms of automatic selection of symbolic inputs and avoidance of redundant symbolic execution. GKLEE$_p$ evolves to SESA after combining static taint anlysis.
- All these tools are built based upon LLVM/Clang-3.2 and can be used independently or as a whole. They support a significant subset of CUDA language features, provide high-degree automation, and uncover actual errors from the latest CUDA SDK kernels and other popular CUDA software.

## 1.10 Dissertation Outline

The rest of the dissertation is organized into chapters as follows:

- **Chapter 2** presents the basic SMT-based symbolic analysis approaches and the first CUDA symbolic analyser and test generator called GKLEE.
- **Chapter 3** introduces the thread abstraction idea through grouping threads that diverge in the same manner into parametric flows and detecting races based on parametric flows.
- **Chapter 4** demonstrates static analysis to automatically symbolize inputs and prune off redundant symbolic execution that will not contribute to race detection. In this chapter, we present the latest framework called SESA and show this framework is useful to uncover insidious errors in the latest CUDA SDK 5.5 kernels and some well-known CUDA software, such as Parboil [3], etc.
- **Chapter 5** concludes our work and looks ahead to the future research directions.

# CHAPTER 2

# BASIC SYMBOLIC ANALYSIS APPROACHES

Programs written for GPUs often contain correctness errors such as races or deadlocks, leading to 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 (noncoalesced memory accesses, memory bank conflicts, and divergent warps).

This chapter presents 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" [13]). GKLEE profits from KLEE's code base and philosophy of testing a given program using symbolic execution. GKLEE is the first symbolic analyser and test generator tailored for GPU programs.

In GKLEE, 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 KLEE's decision procedures ("SMT solvers" [44]) that find solutions for symbolic program inputs. This approach helps the symbolic analyser 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 symbolic analyser can automatically compute and record the input value in a test that 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 [20, 24, 41].

The architecture of GKLEE is shown in Figure 2.1. It employs a C/C++ front-end based on LLVM-GCC (with our customized extensions for CUDA syntax) to parse CUDA

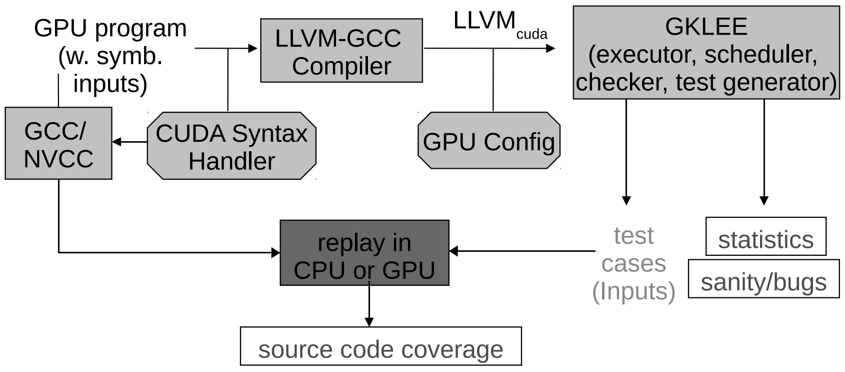

**Figure 2.1**: GKLEE's architecture.

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.

## 2.1 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 ex-

plosion. 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 noncoalesced global memory accesses and shared memory accesses that result in bank conflicts and show how GKLEE can spot these inefficiencies and also understand 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.

## 2.2 Roadmap

§ 2.3 explains the error-classes covered by GKLEE. § 2.4 presents GKLEE's concolic verification: state model, memory type inference, and concolic execution (§ 2.4.2) and error checking/analysis (§ 2.4.3). § 2.5 presents the test generation as witness of errors. § 2.6 presents experimental results, covering issues pertaining to correctness checking/performance (§ 2.6.1) and test set generation/reduction (§ 2.6.2).

## 2.3 Examples of our Analysis/Testing Goals

GKLEE currently supports the CUDA [17] syntax. CUDA programs suffer from not only many common concurrent errors, i.e., race conditions, but also CUDA specific errors.

This section demonstrates the CUDA error classes in detail.

### 2.3.1 Deadlocks

Deadlocks occur when any two threads in a thread block fail to encounter the same *textually aligned* barriers [23], 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 nontextually 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(); }
```

### 2.3.2 Data Races

There are three broad classes of races: intrawarp races, interwarp races, and device/CPU memory races. Intrawarp races can be further classified into intrawarp races without warp divergence, and intrawarp races with warp divergence.

#### 2.3.2.1 Intrawarp Races Without Warp Divergence

Given that any two threads within a warp execute the same instruction, an intrawarp 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; }
```

### 2.3.2.2 Intrawarp 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.

### 2.3.2.3 Interwarp Races

Interwarp 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 nondeterministic depending on the scheduling and latencies of memory accesses. GKLEE guarantees to expose this type of race.

#### 2.3.2.4 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... }
```

### 2.3.3 Memory Access Inefficiencies

There are two kinds of memory access inefficiencies: bank conflicts and noncoalesced memory accesses. GKLEE reports their severity by reporting the absolute number and the percentage of accesses that suffer from this inefficiency, as described in § 2.6.1 in detail.

#### 2.3.3.1 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.

#### 2.3.3.2 Noncoalesced Device Memory Accesses

Noncoalesced 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).

### 2.3.4 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 2.2) kernel taken from CUDA SDK 2.0 [17] sorts $values$'s elements in an ascending order. The steps

```
__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 2.2**: The Bitonic Sort kernel.

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 postcondition 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 heuristics for test-case minimization, as detailed in § 2.5.

## 2.4 Algorithms for Analysis, Test Generation

Given a C++ program, the GKLEE VM (Figure 2.1) 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 § 2.4.3). 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 (§ 2.4.2)
- Execute GPU kernel threads via the *canonical schedule* (§ 2.4.3)
- Fork new states upon nondeterminism due to symbolic values, apply search heuristics and path reduction if needed (§ 2.3.4)
- In a state, at the end of the barrier interval or other synchronization points, perform checks for data races, warp divergence, bank conflicts, and noncoalesced memory accesses (§ 2.4.3)
- When execution path ends, report deadlocks and global memory races (if any), perform test-case selection, and write out a concrete test file (§ 2.5)

### 2.4.1 PTX versus LLVM

Parallel Thread Execution (PTX) [39] is a pseudo-assembly language used in Nvidia's CUDA programming environment. The nvcc compiler translates code written in CUDA into PTX, and the graphics driver contains a compiler that translates the PTX into a binary code that can be run on the processing cores. LLVM is a low level intermediate representation which is platform-independent.

GKLEE works on LLVM bytecode instead of PTX becuase the predecessor of GKLEE, KLEE, is a LLVM symbolic virtual machine for C/C++ programs, a CUDA program is a C++ program with many GPU-specific language features and semantics, so it is straightforward to support CUDA in KLEE virtual machine after we provide some customized transformations to translate CUDA programs to LLVM bytecode. If GKLEE is applied on top of PTX, we may take advantage of simpler compilation in that all front-end customizations for CUDA are not needed. However, PTX is a assembly language, similar to C/C++ assembly language; it is more difficult and error-prone to implement the symbolic interpreter on assembly laugnage instead of IR. NVVM [37] may be a better candidate over PTX.

### 2.4.2 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 to handle CUDA specific features.

Figure 2.3 shows an excerpt of its syntax. One main extension in LLVM$_{cuda}$ is that a variable is attached with its memory sort $\tau$ indicating which memory in GPU it refers to. A variable $v : \langle\psi, \tau\rangle$ has data type $\psi$ and memory sort $\tau$. 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 nonpointer and nonreference variable the memory sort information $\tau$ is not used.

#### 2.4.2.1 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 2.4 visualizes 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.

Moreover, Figure 2.5 presents the formal description of CUDA memory and state model in GKLEE. The state $\Phi$ is extended to consisting of a data state $\Sigma$, a PC $\mathbb{P}$, and a path condition $\mathbb{PC}$, as well as the current scheduled thread $\mathbb{T}$, where $\Sigma$ contains the entire memory hierarchy, $\mathbb{P}$ records the pcs of all threads, and $\mathbb{PC}$ includes the constraints on inputs gathered from conditional branches encountered along the execution. It is a conjunction of a series of constraints. Thread $t$'s pc is given by $\mathbb{P}[t]$. In this section we hide

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

**Figure 2.3**: Syntax of LLVM$_{cuda}$ (excerpt).

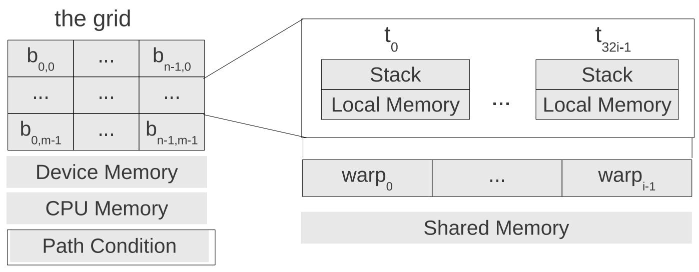

**Figure 2.4**: Components in a symbolic state.

| | | | | |
|---|---|---|---|---|
| Program | := | $\mathbb{L}$ | $\subset$ | $lab \mapsto instr$ |
| Value | := | $\mathbb{V}$ | $\subset$ | $\texttt{byte}^+$ |
| Memory Store | := | $M$ | $\subset$ | $var \mapsto \mathbb{V}$ |
| Local Memory | := | $\sigma_l$ | $\subset$ | $tid \mapsto M$ |
| Read Set | := | $R$ | $\subset$ | $tid \mapsto M$ |
| Write Set | := | $W$ | $\subset$ | $tid \mapsto M$ |
| Shared Memory | := | $\sigma_s$ | $\subset$ | $bid \mapsto \langle M, R, W\rangle$ |
| Device Memory | := | $\sigma_d$ | $\subset$ | $\langle M, R, W\rangle$ |
| Data State | := | $\Sigma$ | $\subset$ | $\sigma_l \times \sigma_s \times \sigma_d$ |
| Program Counter | := | $\mathbb{P}$ | $\subset$ | $tid \mapsto lab$ |
| Path Condition | := | $\mathbb{PC}$ | $\subset$ | $exp$ |
| Current Thread | := | $\mathbb{T}$ | $\subset$ | $tid$ |
| State | := | $\Phi$ | $\subset$ | $\Sigma \times \mathbb{P} \times \mathbb{PC} \times \mathbb{T}$ |

**Figure 2.5**: Formal descripion of state model in GKLEE.

the details of the stack frame $\Gamma$ and heap allocation record $\mathbb{A}$ and use $M$ to compactly denote the various components $\langle\sigma, \Gamma, \mathbb{A}\rangle$. We also reuse the notation $\sigma_l$ to refer to the local store consisting of the stack and local memory, *e.g.,* 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 allow us to skip unimportant details and present when only the core semantics about CUDA. Each thread has its own local store ($\sigma_l$); the threads in a block share a shared memory ($\sigma_s$), and all blocks share the device memory $M$. Note that the shared memory and device memory are both associated with the read set $R$ and write set $W$. These two sets are the mappings from thread IDs to shared/device memory stores. 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 load/store according to $v$'s memory sort, *e.g.,* if $v$'s sort is $\tau_s$, then the shared memory of the current block is accessed.

In Figure 2.5, the thread local store $\sigma_l$, the shared store $\sigma_s$, the device store $\sigma_d$, and the program counter $\mathbb{P}$, as well as the path condition $\mathbb{PC}$ constitute the program state $\Phi$. We use $\Sigma.\sigma_l$, $\Sigma.\sigma_s$, etc., to refer to the components of $\Sigma$. An initial state $\Phi^0 = (\Sigma^0, \mathbb{P}^0, \mathbb{PC}^0, \mathbb{T}^0)$ is valid if

1. $\Sigma^0.\sigma_l(i).v = 0$ for all $v \in V$ $(0 \leq i < n)$
2. $\Sigma^0.\sigma_s.R = \Sigma^0.\sigma_s.W = \emptyset$
3. $\Sigma^0.\sigma_d.R = \Sigma^0.\sigma_d.W = \emptyset$
4. $\mathbb{P}^0(0) = \mathbb{P}^0(i) = ... = \mathbb{P}^0(n-1) = 0$ $(0 \leq i < n)$
5. $\mathbb{PC}^0 = true$
6. $\mathbb{T}^0 = 0$

The first requirement ensures that local variables are initialized to the value in `Word` corresponding to *0*, the second and third requirements are straightforward, the read and write sets of shared/device memories must be empty, and all threads' initial program counters must be same and set as the entry point of kernel, indicated in the fourth requirement. The last requirement specifies that the path condition of the initial state must be *true*.

#### 2.4.2.2 Memory Typing

After a source program is compiled into LLVM bytecode, it is not straightforward 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 $\tau$, which is either $\tau_{\_}$ (unknown), $\tau_l$ (local), $\tau_s$ (shared), $\tau_d$ (device), or $\tau_h$ (host), as per the rules in Figure 2.6. The rules in Figure 2.6 also define a transition: $\mapsto_t \subseteq \Phi \rightarrow \Phi'$. That is, after thread $t$'s statement is executed, the state moves forward.

In our experience, these rules have been found to be sufficiently precise on all the kernels we have applied GKLEE to. Rule `T-alloc` abstracts the LLVM instruction `alloca`, it allocates $n$ elements of type $\psi$ in the local memory, the sort of the address $v$ is $\tau_l$, indicating that it refers to a memory block in the local store. A `getelementptr` instruction calculates the address by adding the offsets $v_2, \ldots, v_n$ to basic address $v_1$, the final address $v$ points to

[T-alloc]:

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

[T-getelementptr]

$$\frac{\mathbb{P}[t] = l \;\; \mathbb{L}[l] = (v = \texttt{getelementptr}\, (v_1 : \tau), v_2 ... v_n)}{(\Sigma, \mathbb{P}, \mathbb{PC}, \mathbb{T}) \mapsto_t (\Sigma[v : \tau \mapsto \Sigma[v_1] + \Sigma[v_2] + ... + \Sigma[v_n]], \mathbb{P}[t \mapsto l+1], \mathbb{PC}, \mathbb{T})}$$

[T-binop]

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

[T-load]:

$$\frac{\mathbb{P}[t] = l \;\; \mathbb{L}[l] = (v = \texttt{load}\, v_1 : \tau) \;\; \tau \neq \tau_{\_}}{(\Sigma, \mathbb{P}, \mathbb{PC}, \mathbb{T}) \mapsto_t (\Sigma[v : \tau_{\_} \mapsto \Sigma[v_1], \;\tau = \tau_{s|d} \mapsto \sigma_{s|d}.R \cup \{v_1\}], \mathbb{P}[t \mapsto l+1], \mathbb{PC}, \mathbb{T})}$$

[T-store]:

$$\frac{\mathbb{P}[t] = l \;\; \mathbb{L}[l] = (\texttt{store}\, v_1, v_2 : \tau) \;\; \tau \neq \tau_{\_}}{(\Sigma, \mathbb{P}, \mathbb{PC}, \mathbb{T}) \mapsto_t (\Sigma[v_2 : \tau \mapsto \Sigma[v_1], \;\tau = \tau_{s|d} \mapsto \sigma_{s|d}.W \cup \{v_2\}], \mathbb{P}[t \mapsto l+1], \mathbb{PC}, \mathbb{T})}$$

[T-br-conc]:

$$\frac{\mathbb{P}[t] = l \;\; \mathbb{L}[l] = (\texttt{br}\, v, \; lab1, \; lab2)}{(\Sigma, \mathbb{P}, \mathbb{PC}, \mathbb{T}) \mapsto_t (\Sigma, \mathbb{P}[((\Sigma[v] = true) \mapsto (t \mapsto lab1)) \; || \; ((\Sigma[v] = false) \mapsto (t \mapsto lab2), \mathbb{PC}, \mathbb{T})}$$

[T-br-sym]:

$$\frac{\mathbb{P}[t] = l \;\; \mathbb{L}[l] = (\texttt{br}\, v, \; lab1, \; lab2) \;\; \Sigma[v] = unknown}{(\Sigma, \mathbb{P}, \mathbb{PC}, \mathbb{T}) \mapsto_t (\Sigma, \mathbb{P}[t \mapsto lab1], \mathbb{PC} \wedge v, \mathbb{T}) \cup (\Sigma', \mathbb{P}'[t \mapsto lab2], \mathbb{PC}' \wedge \neg v, \mathbb{T})}$$

[Memory-inference]:

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

[T-barrier]:

$$\frac{\mathbb{P}[t] = l \;\; \mathbb{L}[l] = (\_\_syncthreads)}{(\Sigma, \mathbb{P}, \mathbb{PC}, \mathbb{T}) \mapsto_t (\Sigma, \mathbb{P}[t \mapsto l+1], \mathbb{PC}, \mathbb{T}[t := (t+1)\%\#T]) \; (t = 0 \mapsto \text{T-race})}$$

[T-race]:

$$Race = \begin{array}{lll} \forall(tid_1 \mapsto M_1) \in \sigma_{s|d}.R & \forall(tid_2 \mapsto M_2) \in \sigma_{s|d}.W & tid_1 \neq tid_2 \; \&\& \; M_1.var = M_2.var \\ \forall(tid_1 \mapsto M_1) \in \sigma_{s|d}.W & \forall(tid_2 \mapsto M_2) \in \sigma_{s|d}.W & tid_1 \neq tid_2 \; \&\& \; M_1.var = M_2.var \end{array}$$

**Figure 2.6**: Rules for thread-level operation.

the same memory as $v_1$. A `binop` instruction returns the calculated value of two operands, if both operands have known sorts, then the calculation fails in identifying the sort of the return value. 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. A `br` instruction is executed in a different manner according to the evaluation result of the conditional $v$. If $v$ is evaluated to a concrete result, $true$ or $false$, then the program counter $\mathbb{P}$ is updated based on the evaluation result of the conditional $v$, shown in rule `T-br-conc`. If the evaluation of $v$ is nondeterministic, then a new state $\Sigma'$ is spawned, and the original and newly produced states' $\mathbb{PC}$ are extended with $v$ and its negation. This procedure is illustrated with rule `T-br-sym`. In this rule, we employ $(\mathbb{PC}, e)$ to represent the combination of the path condition $\mathbb{PC}$ and an expression $e$. The last rule says that a valid sort $\tau'$ is found for $v$ with unknown memory sort if there exists a memory object $v'$ residing in memory sort $\tau'$ and $v$'s value falls within this object. GKLEE traverses the memory hierarchy to reason about the target memory if the previous analysis fails to identify $v$'s sort. It is the extended version of KLEE's method to resolve pointer aliasing, which addresses CUDA's memory hierarchy. In the rule T-barrier, if the current thread $t$ encounters the barrier, then next thread is to be executed. If all threads encounter the barrier, then thread number rolls back to *0*, GKLEE starts race detection through comparing read/write and write/write pairs in shared/device memory. If race is detected, then symbolic execution aborts. The entire race detection is formally described in the rule T-race and depicted in Section 2.4.3.

#### 2.4.2.3 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 : $\tau_l$ (96b) | ... |
|---|---|
| {x : 32b, y : 32b, z : 32b} | ... |

### 2.4.3 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 2.7 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 2.7), 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.

Canonical scheduling is sound for safety properties (will neither result in omissions or false alarms). The caveats that go with this argument are that C/C++ has no standard shared memory consistency semantics to define safe compiler optimizations, and the CUDA programming guide 5.5 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 [42]. This

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

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

**Figure 2.7**: Canonical scheduling and conflict checking in GKLEE.

result also stems from [5] 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 [43] 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 [30].

Consider the following two schedules, we record the writes and reads on $v_1$ and $v_2$ 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.

$$\begin{array}{l}\text{Schedule 1}: \cdot \xrightarrow{\text{W } v_1}_{t_1} \cdot \xrightarrow{\text{W } v_2}_{t_1} \cdot \xrightarrow{\text{R } v_1}_{t_2} \cdot \to \cdots \to \cdot(!) \\ \text{Schedule 2}: \cdot \xrightarrow{\text{R } v_1}_{t_2} \cdot \xrightarrow{\text{W } v_2}_{t_1} \cdot \xrightarrow{\text{W } v_1}_{t_1} \cdot \to \cdots \to \cdot(!)\end{array}$$

### 2.4.3.1 Intrawarp 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 intrawarp 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 (up to the next joint point)

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

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 intrawarp* 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 intrawarp scheduling is an equivalent model of the CUDA hardware.

It eases formal analysis and boosts the performance of GKLEE. Similarly, as Figure 2.7 we can reduce a race-free schedule to a canonical one for interwarps, multiblocks, and barrier intervals (BIs). These transition relations are represented by $\rightarrow_w$, $\rightarrow_b$, and $\rightarrow_{bi}$ respectively.

#### 2.4.3.2 Conflict checking

Figure 2.7 indicates that GKLEE supports various conflict checking:

- Intrawarp 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 whether the accesses in both sides can conflict (discussed in Section 2.3).
- Interwarp race (denoted as $!_2$), checked at the end of a block. 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 interwarp 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 if 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 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 [6] (where no decision procedures are applied) and [28] (which uses SMT solving but relies heavily on abstractions). The method in [54] 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.

Figure 2.8 visualizes the sequential thread schedule. GKLEE provides two types of sequential schedule, pure canonical schedule shown in the left of Figure 2.8 and SIMD-aware canonical schedule shown in the right of Figure 2.8. The pure canonical schedule is the default one for GKLEE, and it shows how GKLEE moves away from generating all (exponential) interleavings to merely generating *one* canonical interleaving *per barrier interval* (barrier intervals are thread code paths from one `__syncthread()` to the next). This approach is sound (no false alarms) and complete (no omissions) for safety properties. The basic point of this approach is that if no races are found, then the canonical schedule is equivalent to any other schedule.

### 2.4.4 Power of Symbolic Analysis

We now present how GKLEE detected a WW race condition in `histogram64Kernel` (Figure 2.9), 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 10 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]>>21)&2016) = 52+0` (`>>26` changed to `>>21` because `THREAD_N` is `32`), resulting in `d_Data[5]` obtaining value `0x04040404`,

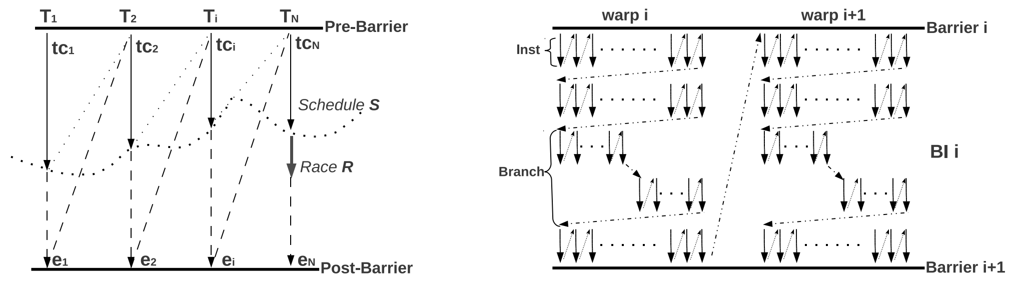

**Figure 2.8**: GKLEE's canonical schedule (left) and SIMD-aware canonical schedule (right).

```
__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 2.9**: Write-write race in Histogram64 (SDK 2.0).

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.

## 2.5 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, 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.

### 2.5.1 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 nondeterminism 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 2.4 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.

### 2.5.2 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 four paths with path conditions $c_{0,0} \wedge c_{1,0}$, $c_{0,0} \wedge \neg c_{1,0}$, $\ldots$. If $t_1$ executes before $t_0$, then the 4 path conditions become $c_{1,0} \wedge c_{0,0}$, $c_{1,0} \wedge \neg c_{0,0}$ $\ldots$. 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 2.4).

### 2.5.3 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).

### 2.5.4 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. 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 § 2.6.2. To the best of our knowledge, coverage measures for SIMD programs have not been previously studied.

## 2.6 Experimental Results

As shown in Figure 2.1, 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; ...
```

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 postconditions are specified before and after the GPU code, respectively. Function `__begin_GPU(NUM)` (a more general format is `__begin_GPU(gdim.{x,y,z}, bdim.{x,y,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 $4 \times 2$. These values can also be made symbolic so as to reveal configuration limitations.

### 2.6.1 Results I: Symbolic Identification of Issues

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 2.1 presents results from SDK 2.0 kernels while Table 2.2 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 (2.1 and 2.2): (#T denoting the number of threads analyzed) assert 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 2.9 executed by two threads *within one warp* is fatal (apparently, CUDA guarantees a net increment by 1). It is poor coding practice anyhow (we notate correctness as 'Unknown'). In Table 2.1,

**Table 2.1**: SDK 2.0 kernel results concluded by GKLEE.

| **Kernels** | Loc | Race | Func. Corr. | #T | Bank Conflict (↓ perf.) | | Coalesced Accesses (↑ perf.) | | | Warp Diverg. (↓ 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∼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 | ben. | – | 16 | 38% | 0% | 97% | 100% | 100% | 43% | yes |
| Bisect Large$^{tb.}$ | 1,400 | ben. | – | 16 | 15% | 0% | 99% | 100% | 100% | 53% | yes |

**Table 2.2**: SDK 4.0 kernel results concluded by GKLEE.

| **Kernels** | Loc | Race | #T | Bank Conflict(↓ perf.) | | Coalesced Accesses (↑ perf.) | | | Warp Diverg. | Volatile(N/M) |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | 1.x | 2.x | 1.0 & 1.1 | 1.2 & 1.3 | 2.x | (↓ perf.) | |
| 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∼50% | 0∼100% | 0∼100% | 0∼100% | 0∼100% | 0% | no/no |
| Bisect Small | 1,000 | ben. | 16 | 38% | 0% | 97% | 100% | 100% | 43% | yes/yes |

"Reduction" contains seven 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. In Table 2.2, 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 eight different implementations; we report the results as a range through "∼." Kernels having the same results as their SDK 2.0 versions, including Bitonic Sort, MatrixMult and Bisect Large, are not presented.

One row result is presented for Bank Conflicts, Memory Coalescing, and Warp Divergence, this row averaging over barrier intervals. The 71% for Scan Best under Bank Conflict (compute capability 2.x) is obtained by 14 BIs being analyzed, and out of it, 10 had bank conflicts, which is 71%. 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 CUDA profiler reports, 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). We found GKLEE's findings to be in agreement with that discovered by the profiler. GKLEE's statistics can be used for early detection of these performance issues on symbolic inputs.

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), and 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 the future, we hope to extend GKLEE to "understand" compiler optimizations and deal with this issue more thoroughly.

### 2.6.2 Results II: Testing and Coverage

We assess GKLEE with respect to newly proposed coverage measures and coverage directed execution pruning. In Table 2.3, 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.

**Table 2.3**: Cov$^t$ and CovTB$^t$ measure bytecode coverage w.r.t threads.

| **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 |

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 five 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 2.3, we conclude that the maximum measures give an overly optimistic impression, so we set them aside. min #test tests are obtained by performing test case selection after the execution. The result for "Histogram64" is limited to 600 s. No test reductions used in generating this table. Exec. time on typical workstation. 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 2.4) 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). $\mathrm{Red}_{TB}$ 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, $\mathrm{Red}_{BI}$ only looks for some thread reaching each

**Table 2.4**: Reduction heuristic comparisons.

| **Kernels** | No Reductions | | $\mathrm{Red}_{TB}$ | | $\mathrm{Red}_{BI}$ | |
|---|---|---|---|---|---|---|
| | #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% |

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 $\mathrm{Red}_{TB}$ 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. 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.

# CHAPTER 3

# THREAD ABSTRACTION FOR SCALABILITY

This chapter introduces an extension of our *symbolic* approach to GPU program analysis published recently in [31] and supported by our recently released tool GKLEE. GKLEE employs a formal analysis approach that is convenient to use for practitioners, yet effective at finding deep-seated bugs. A GKLEE user writes standard C++ CUDA programs, indicating some of the program variables to be *symbolic* (the rest are assumed to be concrete variables). These programs are compiled into LLVM byte-code, with GKLEE serving as a symbolic virtual machine. When GKLEE runs such a byte-code program, it generates and records constraints relating the values of symbolic variable. Conditional expressions in the C++ code (*e.g.*, switch statements) generate constraints covering both outcomes of a branch; these are solved by instantiating the symbolic variables to cover all feasible branching options (or as per user-control of how much to cover). The result is that users automatically obtain path-coverage to the desired degree. GKLEE also writes out these cases into test files that then form test suites to be run on any platform, ensuring high coverage. Because of the recent growth in the power of SMT-solvers used to solve these constraints [44], a tool such as GKLEE is able to handle nontrivial SDK kernel functions.

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

In this chapter, we provide an extension of GKLEE that *exploits thread symmetry and provides a way to analyze GPU programs containing large (bounded) numbers of threads in real kernels*. In a nutshell, our method partitions the space of executions of a GPU program into *parametric flow equivalence classes* (PFE) and models the race analysis problem over two parametric threads in one PFE equivalence class. This analysis method over parametric flows has been implemented in a new version of GKLEE called GKLEE$_p$:

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

## 3.1 Background

This section presents background about GKLEE$_p$. GKLEE$_p$ inherits the symbolic execution power from GKLEE, whose infrastructure is introduced in Figure 2.1. GKLEE$_p$ differs from GKLEE in the part of executor and scheduler. They share the same front-end and test generator.

Race checking in concurrent programs has been studied extensively. GPU program race checking differs in many essential ways from that studied in the non-GPU contexts: (i) GPU programs are largely computation-oriented, synchronizing sparingly through barriers and atomic operations, and (ii) the number of threads involved in GPU programs is vastly more than entertained in non-GPU areas. Formal and semiformal methods in non-GPU contexts employ various lock-set and happens-before based methods [26, 40]. In terms of finding races with high assurance, one of the main impediments has been *schedule (or interleaving) explosion*. For example, five threads carrying out five sequential instructions each generate $25!/(5!)^5 \approx$ 13 trillion interleavings. While methods such as partial order reduction [19,53] dramatically reduce the number of interleavings to be examined, in the case of CUDA programs we can do much better.

In the previous section, we show that symbolically executing *one* schedule (called the *canonical schedule*) through all the threads, and recording potential conflicting pairs during this schedule gives us the ability to detect a *race* if there is *any race*. The saving due to canonical scheduling is essential for the success of GKLEE. For example, with $N$ being $5$ and there being five threads, instead of examining 13 trillion schedules to check for races, under the canonical scheduling, *one* schedule finds *a race* such as $R'$ (or finds $R$ itself) if there is *any race*.

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

## 3.2 A Motivating Example

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

A motivating example (Figure 3.1) manifests the advantages of GKLEE$_p$. In this kernel, 8K threads are involved, with four blocks and 2K threads per block. Two arrays *a* and *b* are created and located in the device memory and shared memory, respectively.

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

As shown in Figure 3.2, GKLEE$_p$ yields four parametric flows. Each lozenge denotes a TDC, and each rectangle in the diagram represent the TICs. GKLEE$_p$ starts its execution within one thread. When a TDC is encountered, it spawns a new flow. For example, when $bid\%2 \neq 0$ is encountered, two flows are generated with the appropriate conditional path conditions (namely $bid\%2 \neq 0$ and $bid\%2 = 0$).

### 3.2.1 A Data Race

Whenever two memory accesses involving a common location are performed concurrently by two threads, with one of the accesses being a write, a data race situation is created.

```
// a[4 * 2048] in device memory;
// b[2048] in shared memory;
__global__ void test(unsigned * a) {
  unsigned bid = blockIdx.x;
  unsigned tid = threadIdx.x;

  if (bid % 2 != 0) {
    if (tid < 1024) {
      unsigned idx = bid * blockDim.x + tid;
      b[tid] = a[idx] + 1; // Write of Race-1
      if (tid % 2 != 0) {
        b[tid] = 2; // Write of Race-2
      } else {
        if (tid > 0)
          b[tid] = b[tid -1]+1; // Read of Race-2
        }
    } else {
      b[tid] = b[tid -1]; // Read of Race-1
    }
  } else {
    unsigned idx = bid * blockDim.x + tid;
    b[tid] = a[idx] + 1;
  }
}
```

**Figure 3.1**: The motivating example.

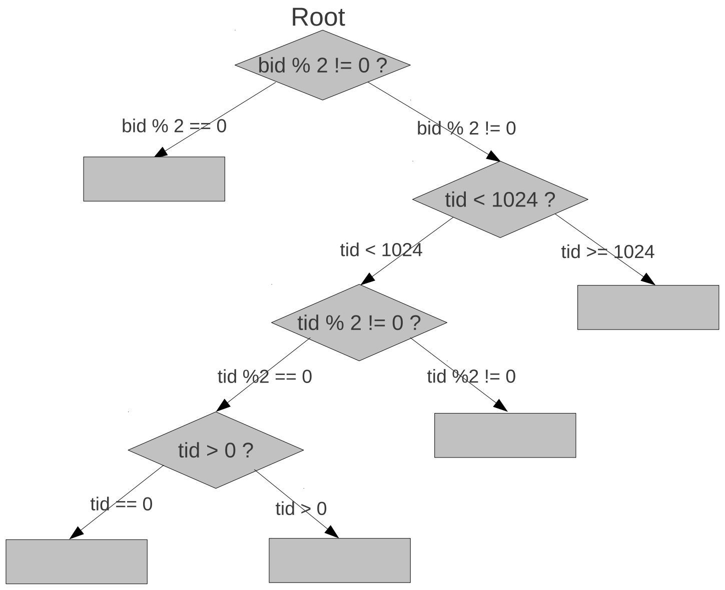

**Figure 3.2**: Parametric flows for the motivating example.

Data races are almost impossible to discern manually. They may *never* produce corrupt data results upon testing because of the restricted nature of scheduling employed by typical GPU hardware. Most damaging of all, races may license compiler transformations that are "unwarranted," resulting in an error that appears completely unrelated to the root-cause and is potentially *very* confusing.[1] This example has two race conditions:

- The Write access in Line 10 (done by Thread 1023) and the Read access done by Thread 1024 in Line 18. This is an interwarp race – well-understood by anyone who has studied GPUs and the CUDA execution semantics.
- Any odd-numbered thread (*e.g.,* thread $1$) and an even-numbered thread that is numbered one higher (*e.g.,* thread $2$), both of which are in the range $0, \ldots, 1023$, involving the Write access on Line 12 and the Read access on line 15. These lines are *mutually exclusive*; then why is it a race? Reason: on one GPU, line 12 may be executed before Line 15, and vice versa on another. Thus on GPU1, the write occurs before the read, while on GPU2, it is the other way. This "race" is noticed when programmers port the code from GPU1 to GPU2. This race type was first identified in [31] where it is called a *porting race*.

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

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

## 3.3 Foundation

We first introduce the race and deadlock checking in the parameterized circumstance, and then prove the soundness of parameterized checking. Finally, we demonstrate that the parameterized checking respects the SIMD property of CUDA programs while not losing analysis precision.

### 3.3.1 Parameterized Race and Deadlock Checking

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

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

```
void __global__ kernel1 (int *a, int b) {
  __shared__ int temp[N];
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < N) temp[idx] = a[f1(idx)] + b;
  __syncthreads(); // A barrier
  if (idx < N) a[idx] = temp[f2(idx)];
}
```

Suppose the barrier is removed from this example; we can observe that the accesses $a[idx]$ and $a[f1(idx)]$ by different threads may race depending on function $f1$. This can be detected by examining the symbolic models of two threads as follows (private variables in a thread are superscripted by the thread id, and for simplicity we assume that threads $t_1$ and $t_2$ are in the same block but in different warps). More formally, a race occurs if predicate $t_1.x \neq t_2.x \ \wedge \ id^{t_1} < N \ \wedge \ id^{t_2} < N \ \wedge \ f1(idx^{t_1}) = idx^{t_2} \ \wedge \ |t_2.x - t_1.x| \geq 32$ holds. A constraint solver (an SMT tool for us [44]) can determine whether this predicate is satisfiable, and if so, it would return a concrete satisfying instance. Accesses to $temp$ can be analyzed similarly (knowing $f2$).

$$\begin{array}{l} \quad\text{thread } t \in \{t_1, t_2\} \\ idx^t = blockIdx.x * blockDim.x + t.x \\ if\ (idx^t < N) \text{ read } a[f1(idx^t)] \\ if\ (idx^t < N) \text{ write } a[idx^t] \end{array}$$

To further illustrate these ideas, consider the control-flow graph (CFG) given in Figure 3.3(a). This diagram shows how statements $s_1$ through $s_3$ might be situated in some example program. At first glance, this appears ill-synchronized: one thread may take the $s_1$ to $s_2$ path encountering no barriers while another may take the path through $p_1$ encountering a barrier. Our SMT techniques can determine whether these paths are feasible and flag a

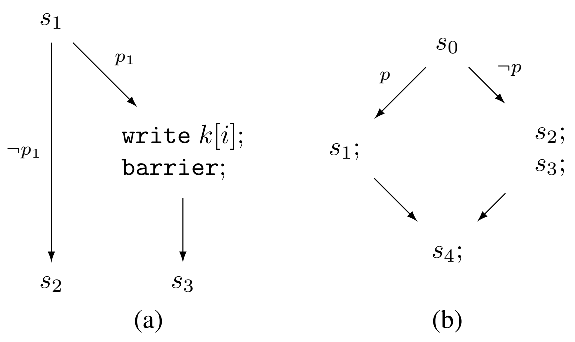

**Figure 3.3**: Example CFGs.

deadlock (due to textually nonaligned barriers [46]) if so. Our approach checks whether all threads make the same decision on the condition.

In Figure 3.3(b), both the left and the right branch contain no barrier; thus they are considered well synchronized. We now check for conflicting accesses, some of which involve conditionals. The conflict check includes the following expressions (here $\sim$ denotes the *conflict* relation, and $\nsim$ denotes *nonconflicting*). Also let us use $p\,?\,s$ to denote an expression $s$ guarded by path condition $p$. Now, this CFG may be regarded as consisting of one *barrier interval (BI)* containing five accesses (1) $s_0$, (2) $p\,?\,s_1$, (3) $\neg p\,?\,s_2$, (4) $s_4$, and (5) $\neg p\,?\,s_3$. Conflict freedom requires the pairwise comparison of these five accesses for two parameterized threads; the $5 \times 5 = 25$ comparisons include the following. In GKLEE$_p$, this procedure is done by constructing a flow for each path condition and checking the conflicts over two representative threads (with symbolic ids):

$$\begin{array}{ll} p^{t_2} \Rightarrow s_0^{t_1} \nsim s_1^{t_2} & \neg p^{t_2} \Rightarrow s_0^{t_1} \nsim s_2^{t_2} \\ p^{t_1} \wedge \neg p^{t_2} \Rightarrow s_1^{t_1} \nsim s_2^{t_2} & \neg p^{t_2} \Rightarrow s_4^{t_1} \nsim s_3^{t_2} \end{array}$$

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

### 3.3.2 Parameterized Checking with SIMD

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

GKLEE performs scheduling with respect to SIMD. The threads within a warp are executed in a lock-step manner. The warps (or blocks) themselves follow the usual canonical method, synchronizing at the CUDA barriers. Figure 3.4 shows how multiple warps are executed. In particular, in cases where the threads in a warp diverge (*i.e.,* make different decisions over the same branch), the lock-step requirement is met by the hardware by executing the two sides sequentially and merging them at the first convergence point (*e.g.,*

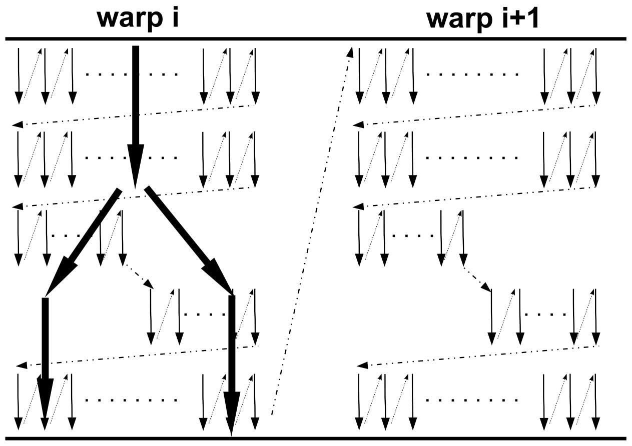

**Figure 3.4**: Canonical schedule with SIMD.

the nearest common postdominator). This is adopted by GKLEE to take care of all nuances of the CUDA semantics. GKLEE$_p$ inherits GKLEE's SIMD-aware scheduling scheme and makes it parameterized (the parametric flow is marked in Figure 2.8 as bold arrows). For an unconditional instruction, its execution by $N$ threads is modeled by using one parameterized thread. This thread represents other threads in the same warp, in the different warp, in the different group, and so on. Suppose this instruction accesses shared location $a(tid)$, then threads may cause a race when $t_1 \neq t_2 \ \wedge \ a(t_1) \sim a(t_2)$, regardless of whether $t_1$ and $t_2$ are within the same warp or not.

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

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

Clearly, this constraint is also applied when $t_1$ and $t_2$ are in the different warps. Hence $(c(t_1) \ ? \ a(t_1)) \sim (\neg c(t_2) \ ? \ a(t_2))$ is a generic constraint for detecting races relevant to condition $c$, and we need not to distinguish the intrawarp and interwarp cases. This illustrates the following general principles of using parametric flow tree to check races when respecting the SIMD model. In sum, our parametric flow based analysis respects

SIMD by considering both intrawarp and interwarp. The case of intra- and interblocks is analogous.

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

## 3.4 Parameterized Checking: Algorithm

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

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

Roughly speaking, the construction of a PFT proceeds as follows (here we focus on how the symbolic executor constructs the tree, skipping most details discussed in §3.3).

1. Starting with each *barrier* statement (the initial state of the program can be assumed to have one), GKLEE$_p$ launches one representative thread $tid_0$ – a symbolic value – for execution. So long as a conditional statement is not encountered, this representative thread would keep running until the next barrier is encountered.
2. When a $tid$- or $bid$-dependent condition is encountered, two nodes are forked, one representing the threads satisfying the condition, the other one for those satisfying the negation of the condition.
3. Similarly, when a symbolic-input-dependent condition is encountered, two nodes may be forked to represent the true and false cases of the condition.
4. Once all nodes reach an explicit barrier $\_\_syncthreads()$, the tree enters a synchronization status and starts checking various kinds of errors (including intrawarp races).

Figure 3.5 shows a portion of the PFT of the Bitonic Sort kernel. Since the first BI contains no conditional statements, no forking is needed. In the next BI, the PFT shown in Figure 3.5 is constructed during the execution. Conditions in two outer loops are evaluated to be concrete values, so they are not shown in the parametric flow tree. The top three conditions are TDCs leading to node forking. The other four depend on both the symbolic inputs and built-in variables (*e.g.,* $tid$ and $bid$) and will lead to node forking too. The figure shows that flow $0$ takes the path with path condition $ixj > tid$, $(tid \ \& \ k) == 0$ and $shared[tid] > shared[ixj]$. Note that memory accesses such as $shared[tid_0]$ are guarded by $ixj > tid_0 \wedge (tid_0 \ \& \ k) == 0$.

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

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

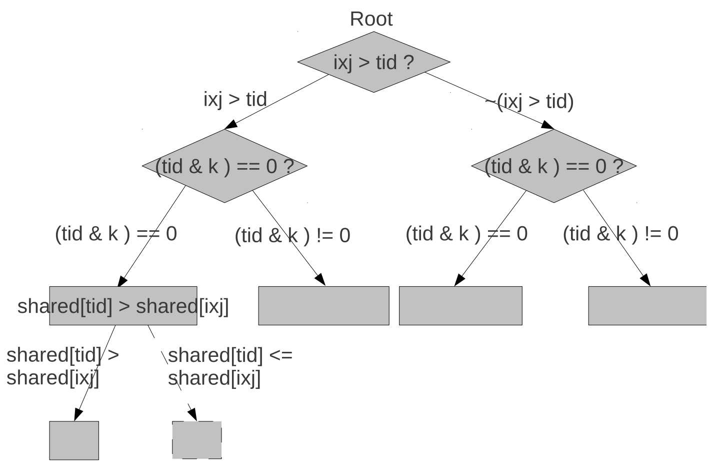

**Figure 3.5**: Parametric flow tree for Bitonic Sort.

$tid_1 \wedge (tid_1 \ \& \ k) == 0$.

Extra care must be taken on the relation of two threads. In the scenario without warp involved, the following constraints specify the threads' relation.

Same Block:
$$(bid_0 = bid_1) \wedge (tid_0 \neq tid_1)$$

Different Blocks:
$$(bid_0 \neq bid_1)$$

If warp is taken into account, then these two threads can be within the same warp, in different warps but in the same block, or in different blocks. The following shows the constraints over the thread ids for these scenarios.

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

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

Different Blocks:
$$(bid_0 \neq bid_1)$$

## 3.5 Formal Description of GKLEE$_p$

After introducing the parametric flow and the construction of parametric flow tree, this section formally presents state model of GKLEE$_p$, parametric flow operation, and the construction of parametric flow tree, as well as state transition within GKLEE$_p$.

### 3.5.1 Formal Description of State Model Upon GKLEE$_p$

With respect to the parametric flows, we should define new state models upon GKLEE$_p$. Since *bid* and *tid* are symbolic in GKLEE$_p$, CUDA model semantics must be changed accordingly. We use *fid* to denote the parametric flow's identifier.

Similar to GKLEE, GKLEE$_p$ maintains the $\sigma_l$ and $\sigma_s$, as well as $\sigma_d$ to represent the local memory, shared memory, and device memory, respectively. However, these memory resorts are not operated by real threads; for example, $\sigma_l$ represents a mapping from flows to memory store; that is, each flow owns and manipulates its own local memory. $\sigma_s$ in GKLEE$_p$ is slightly different from that in GKLEE since shared memory in GKLEE$_p$ is used to denote any of shared memories. Moreover, the read set $R$ and write set $W$ consist of memory accesses, each of which is associated with a predicate represented as $pred$, acquired through $\mathbb{PFT}$. Intuitively, $(pred, r)$ represents a read access that should be operated if $pred$ holds. For instance, in the code $if(threadIdx.x \ \% \ 2 == 0) \ \{shared[threadIdx.x] = ...\}$, the write access on shared memory $shared[threadIdx.x]$ is constrained by $threadIdx.x \ \% \ 2 == 0$. In constrast to GKLEE, each flow includes a program counter (pc) recording the label of the current instruction. Instead of the path constraint $\mathbb{PC}$, GKLEE$_p$ maintains a parametric flow tree $\mathbb{PFT}$. And this tree is interpreted as the mapping between $fid$ to $pred$ to specify the parametric flow. GKLEE$_p$ defines the notations analoguely — read operation $\Sigma[v]$ and write operation $\Sigma[v \mapsto k]$ occur in the appropriate memory load/store accordingto $v$'s memory resorts. In Figure 3.6, the data state $\Sigma$, the program counter $\mathbb{P}$ and parametric flow tree $\mathbb{PFT}$ constitute the program state $\Phi$.

Since parametric flows are produced on the fly, it is difficult to obtain the exact number of flows initially, and we use $n$, the number of real threads, as the possible upper bound of number of flows. An initial state $\Phi^0 = (\Sigma^0, \mathbb{P}^0, \mathbb{PFT}^0, \mathbb{F}^0)$ is valid if

1. $\Sigma^0.\sigma_l(i).v = 0$ for all $v \in V \ (0 \leq i < n)$
2. $\Sigma^0.\sigma_s.R = \Sigma^0.\sigma_s.W = \emptyset$
3. $\Sigma^0.\sigma_d.R = \Sigma^0.\sigma_d.W = \emptyset$
4. $\mathbb{P}^0(0) = \mathbb{P}^0(i) = ... = \mathbb{P}^0(n-1) = 0 \ (0 \leq i < n)$
5. $\mathbb{PFT}^0 = \emptyset$
6. $\mathbb{F}^0 = 0$

| | | | | |
|---|---|---|---|---|
| Program | := | $\mathbb{L}$ | $\subset$ | $lab \mapsto instr$ |
| Value | := | $\mathbb{V}$ | $\subset$ | $\texttt{byte}^+$ |
| Memory Store | := | $M$ | $\subset$ | $var \mapsto \mathbb{V}$ |
| Local Memory | := | $\sigma_l$ | $\subset$ | $fid \mapsto M$ |
| Read Set | := | $R$ | $\subset$ | $fid \mapsto (pred, M)$ |
| Write Set | := | $W$ | $\subset$ | $fid \mapsto (pred, M)$ |
| Shared Memory | := | $\sigma_s$ | $\subset$ | $\langle M, R, W \rangle$ |
| Device Memory | := | $\sigma_d$ | $\subset$ | $\langle M, R, W \rangle$ |
| Data State | := | $\Sigma$ | $\subset$ | $\sigma_l \times \sigma_s \times \sigma_d$ |
| Program Counter | := | $\mathbb{P}$ | $\subset$ | $fid \mapsto lab$ |
| Parametric Flow Tree | := | $\mathbb{PFT}$ | $\subset$ | $fid \mapsto pred$ |
| Current Flow | := | $\mathbb{F}$ | $\subset$ | $fid$ |
| State | := | $\Phi$ | $\subset$ | $\Sigma \times \mathbb{P} \times \mathbb{PFT} \times \mathbb{F}$ |

**Figure 3.6**: Formal description of state model in GKLEE$_p$.

The first requirement ensures that local variables are initialized to the value in `Word` corresponding to *0*, the second and third requirements are straightforward, the read and write sets of shared/device memories must be empty, and all threads' initial program counters must be the same, indicated in the fourth requirement. The last requirement specifies that the parametric flow tree of the initial state must be *empty*.

### 3.5.2 Formal Description of Flow Operation

Similar to GKLEE, we must employ the GPU-specific memory type inference method by computing for each expression a sort $\tau$. The type inference rules based on parametric flow are described in combination of flow level operation, illustrated in Figure 3.7.

The rules in Figure 3.7 and 3.8 also define a transition: $\mapsto_f \subseteq \Phi \to \Phi'$. That is, after flow $f$'s statement is executed, the state moves forward.

Rule `F-alloc` abstracts the LLVM instruction `alloca`. It allocates $n$ elements of type $\psi$ in the local memory, and the sort of the address $v$ is $\tau_l$, indicating that it refers to a memory block in the local store. A `getelementptr` instruction calculates the address by adding the offsets $v_2$, ..., $v_n$ to basic address $v_1$. The final address $v$ points to the same memory as $v_1$. A `binop` instruction returns the calculated value of two operands. If both operands have known sorts, then the calculation fails in identifying the sort of the return value, 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. Note that each `load` or `store` access must be associated with the $pred$ expression, which is acquired from the $\mathbb{PFT}$. A `br` instruction is executed in a different manner according to the evaluation result of the conditional $v$. If $v$ is evaluated to a concrete result, $true$ or $false$, then the program counter $\mathbb{P}$ is updated based on the evaluation result of the conditional $v$ ($Eval(\Sigma[v])$), shown in the rule `F-br-conc`. Note that $Eval(V)$ represents the evaluation result acquired through SMT solver under the constraint of the predicate acquired from $\mathbb{PFT}$. If the evaluation of $v$ is nondeterministic and $v$ is not categorized into the pure `bid-` or `tid-`dependent conditional, then a new state $\Phi'$ is spawned, and the original state updates its $\mathbb{P}$ and extends its $\mathbb{PFT}$ with the conditional $\Sigma[v]$. $\Phi'$ differs from $\Phi$ in two ways: (i) the program counter $\mathbb{P}'$ and (ii) the parametric tree $\mathbb{PFT}'$. Then $\Phi'$ is preserved and will be scheduled when the current state terminates.

[F-alloc]:

$$\frac{\mathbb{P}[f] = l \ \ \mathbb{L}[l] = (v = \texttt{alloc}\ \psi, n)}{(\Sigma, \mathbb{P}, \mathbb{PFT}, \mathbb{F}) \mapsto_f (\Sigma[v : \tau_l \mapsto 0^{n \times \texttt{sizeof}(\psi)}], \mathbb{P}[f \mapsto l + 1], \mathbb{PFT}), \mathbb{F}}$$

[F-getelementptr]

$$\frac{\mathbb{P}[f] = l \ \ \mathbb{L}[l] = (v = \texttt{getelementptr}\ (v_1 : \tau), v_2 \ ... \ v_n)}{(\Sigma, \mathbb{P}, \mathbb{PFT}, \mathbb{F}) \mapsto_f (\Sigma[v : \tau \mapsto \Sigma[v_1] + \Sigma[v_2] + ... + \Sigma[v_n]], \mathbb{P}[f \mapsto l + 1], \mathbb{PFT}, \mathbb{F})}$$

[F-binop]

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

[F-load]:

$$\frac{\mathbb{P}[f] = l \ \ \mathbb{L}[l] = (v = \texttt{load}\ v_1 : \tau) \ \ \tau \neq \tau_{\_} \ \ pred \mapsto \mathbb{PFT}[f]}{(\Sigma, \mathbb{P}, \mathbb{PFT}, \mathbb{F}) \mapsto_f (\Sigma[v : \tau_{\_} \mapsto \Sigma[v_1], \ \tau = \tau_{s|d} \mapsto \sigma_{s|d}.R \cup \{(pred, v_1)\}], \mathbb{P}[f \mapsto l + 1], \mathbb{PFT}, \mathbb{F})}$$

[F-store]:

$$\frac{\mathbb{P}[f] = l \ \ \mathbb{L}[l] = (\texttt{store}\ v_1, v_2 : \tau) \ \ \tau \neq \tau_{\_} \ \ pred \mapsto \mathbb{PFT}[f]}{(\Sigma, \mathbb{P}, \mathbb{PFT}, \mathbb{F}) \mapsto_f (\Sigma[v_2 : \tau \mapsto \Sigma[v_1], \ \tau = \tau_{s|d} \mapsto \sigma_{s|d}.W \cup \{(pred, v_2)\}], \mathbb{P}[f \mapsto l + 1], \mathbb{PFT}, \mathbb{F})}$$

[F-br-true]:

$$\frac{\mathbb{P}[f] = l \ \ \mathbb{L}[l] = (\texttt{br}\ v, \ lab1, \ lab2) \ \ Eval(\Sigma[v]) = true}{(\Sigma, \mathbb{P}, \mathbb{PFT}, \mathbb{F}) \mapsto_f (\Sigma, \mathbb{P}[f \mapsto lab1], \mathbb{PFT}, \mathbb{F})}$$

[F-br-false]:

$$\frac{\mathbb{P}[f] = l \ \ \mathbb{L}[l] = (\texttt{br}\ v, \ lab1, \ lab2) \ \ Eval(\Sigma[v]) = false}{(\Sigma, \mathbb{P}, \mathbb{PFT}, \mathbb{F}) \mapsto_f (\Sigma, \mathbb{P}[f \mapsto lab2], \mathbb{PFT}, \mathbb{F})}$$

[F-br-sym]:

$$\frac{\mathbb{P}[f] = l \ \ \mathbb{L}[l] = (\texttt{br}\ v, \ lab1, \ lab2) \ \ (Eval(\Sigma[v]) = unknown \ \wedge \ v \notin TDC)}{(\Sigma, \mathbb{P}, \mathbb{PFT}, \mathbb{F}) \mapsto_f (\Sigma, \mathbb{P}[f \mapsto lab1], (\mathbb{PFT}[f], \Sigma[v]), \mathbb{F}) \cup (\Sigma', \mathbb{P}'[f \mapsto lab2], (\mathbb{PFT}'[f], \neg\Sigma[v]), \mathbb{F}')}$$

[F-fork]:

$$\frac{\mathbb{P}[f] = l \ \ \mathbb{L}[l] = (\texttt{br}\ v, \ lab1, \ lab2) \ \ (Eval(\Sigma[v]) = unknown \ \wedge \ v \in TDC)}{f' \ is \ forked \ \wedge \ (\Sigma[f'] := \Sigma[f], \mathbb{P}[f'] := \mathbb{P}[f], \mathbb{PFT}[f'] := \mathbb{PFT}[f])}$$

[F-br-tdc]:

$$\frac{\mathbb{P}[f] = l \ \ \mathbb{L}[l] = (\texttt{br}\ v, \ lab1, \ lab2) \ \ (Eval(\Sigma[v]) = unknown \ \wedge \ v \in tdc)}{(\Sigma, \mathbb{P}, \mathbb{PFT}, \mathbb{F}) \mapsto_f (\text{F-fork}) \ \wedge \ (\Sigma, \mathbb{P}[f \mapsto lab1, f' \mapsto lab2], ((\mathbb{PFT}[f], \Sigma[v]), (\mathbb{PFT}[f'], \neg\Sigma[v])), \mathbb{F})}$$

[Memory-inference]:

$$\frac{v : \tau_{\_} \quad ((v' : \tau') \mapsto k) \ \in \ \Sigma \quad \Sigma \ \vdash \ v'.base \leq v.base < v.base + v.size \leq v'.base + v'.size}{v : \tau'}$$

**Figure 3.7**: Rules for flow operation.

[F-barrier]:

$$\frac{\mathbb{P}[f] = l \;\; \mathbb{L}[l] = (\_\_syncthreads)}{(\Sigma, \mathbb{P}, \mathbb{PFT}, \mathbb{F}) \mapsto_t (\Sigma, \mathbb{P}[f \mapsto l+1], \mathbb{PFT}, \mathbb{F}[f := \text{F-next-flow}])}$$

[F-next-flow]:

$$\frac{flow\_stack \; != \emptyset \;\; f := popflow\_stack}{(\Sigma, \mathbb{P}, \mathbb{PFT}, \mathbb{F}) \mapsto (\Sigma, \mathbb{P}, \mathbb{PFT}, \mathbb{F})(flow\_stack = \emptyset \mapsto \text{F-race})}$$

[F-race]:

$$\begin{array}{ll} \forall(fid_1 \mapsto (pred_1, M_1)) \in \sigma_{s|d}.R & \forall(fid_2 \mapsto (pred_2, M_2)) \in \sigma_{s|d}.W \\ & fid_1 \neq fid_2 \,\&\&\, pred_1 \wedge pred_2 \,\&\&\, M_1.var = M_2.var \\ \forall(fid_1 \mapsto (pred_1, M_1)) \in \sigma_{s|d}.W & \forall(fid_2 \mapsto (pred_2, M_2)) \in \sigma_{s|d}.W \\ & fid_1 \neq fid_2 \,\&\&\, pred_1 \wedge pred_2 \,\&\&\, M_1.var = M_2.var \end{array}$$

**Figure 3.8**: More rules for flow operation.

Note that in $\Phi'$ the flow $f$ is not changed. The entire procedure is illustrated in the rule `F-br-sym`. In this rule, we employ $(\mathbb{PFT}[f], e)$ to represent the combination of flow $f$'s predicate and an expression $e$. The rule `F-fork` describes that if the evaluation of the conditional $v$ is nondeterministic and $v$ is TDC, a new flow $f'$ is forked through copying flow $f$'s memory space and predicate and synchronizing its program counter, and then $f'$ is preserved and scheduled when the current flow $f$ encounters the explicit barrier. Details are described in Section 3.5.3. The rule `F-br-tdc` is a continuation of `F-fork`; that is, after producing the new flow $f'$, GKLEE$_p$ further updates $\mathbb{P}[f']$ as well as $\mathbb{PFT}[f']$. In GKLEE$_p$, we use $\{base, size\}$ to model a variable. So the last rule `Memory-inference` says that a valid sort $\tau'$ is found for $v$ with unknown memory sort if there exists a memory object $v'$ residing in memory sort $\tau'$, and $v$'s value falls within this object. GKLEE$_p$ traverses the memory hierarchy to reason about the target memory if the previous analysis fails to identify $v$'s sort. It is the extended version of KLEE's method to resolve pointer aliasing, which addresses CUDA's memory hierarchy. `F-barrier` specifies that when the current flow encounters the __syncthreads() barrier, then next flow is scheduled, which is depicted in rule `F-next-flow`.

### 3.5.3 Formal Description of Parametric Flow Schedule

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ and consider a CUDA program *pgm* meant for execution within `blockIdx` $\in$ `gridDim.`$\{$`x,y,z`$\}$; then there are `gridDim.x`$\times$`gridDim.y`$\times$`gridDim.z` blocks where

`gridDim.`$\{$`x,y,z`$\}$ $\in$ $\mathbb{N}$, and a relation can be automatically inferred:

`0 ≤ blockIdx.{x,y,z} < gridDim.{x,y,z}`

Similarly, there are `threadIdx ∈ blockDim.{x,y,z}`; then there are `blockDim.x` × blockDim.y × blockDim.z threads per block where `blockDim.{x,y,z}` $\in \mathbb{N}$, and a relation can also be inferred:

`0 ≤ threadIdx.{x,y,z} < blockDim.{x,y,z}`

These two relations shown above describe that block or thread identifiers must be within the bound specified by users. And they are used as the precondition of parameterized race checking; thus parametric flow schedule is considered the *Bounded* flow schedule. A barrier interval (BI) is the interval (block of code) enclosed by two successive barriers. We employ the set $DoneBar = fid \mapsto barset$ to record the set of barriers the flow $fid$ explored, and $DoneBar$ is an empty set initially.

GKLEE$_p$ employs a simple flow-based sequential schedule, *parametric flow schedule*, by default. In addition to the flow operation semantics shown in Figure 3.7, it is often necessary to present the semantics on how the flow $f$ switches. Briefly, within a barrier interval, a flow $f$ starts executing from the top barrier and terminates at the bottom barrier. During the exploration, a child flow might be forked and preserved in the flow stack. Then all enabled flows will be scheduled until the flow stack is empty. All flows explored in the current BI will be extended in the next BI; that is, their predicates are propagated to flows in the next BI. The parametric flow schedule is formally described as follows:

$$\frac{\left(\begin{array}{l} \mathbb{P}[f] = l \ \wedge \ \mathbb{L}(l) = \texttt{syncthreads} \wedge \\ (\Sigma, \mathbb{P}, \mathbb{PFT}) \mapsto_f (\Sigma, \mathbb{P}', \mathbb{PFT}) \wedge \\ l' = l + 1 \ \wedge \ \mathbb{P}[f] = l' \ \wedge \ flow\_stack = \emptyset \wedge \\ create\ flow\_stack \end{array}\right) \wedge \texttt{Races} \wedge \texttt{Barrier\_Divergence}}{(\Sigma,\mathbb{P},\mathbb{PFT}) \mapsto (\Sigma,\mathbb{P}'[f \mapsto l'],\mathbb{PFT})}$$

The flows within the current BI are constructed based on the flows created in the previous BI, and the flows across all the BIs in a program constitute an intact $\mathbb{PFT}$. The flow schedule within a BI proceeds as follows (here we focus on how the symbolic executor constructs the tree).

1. Popping a flow from the stack, the flow continues exploring. As long as a conditional statement is not encountered, this flow keeps running until the next barrier is encountered.
2. If a BDC/TDC is encountered and its evaluation is nondeterministic, then a new flow is spawned and pushed into the stack. This is described in the rule `F-br-tdc`.

3. When a symbolic-input-dependent condition is encountered and its evaluation is also nondeterministic, a new state is produced, and the details are introduced in the rule `F-br-sym`.
4. Once the current flow reaches the explicit barrier $\_\_syncthreads()$, GKLEE$_p$ picks up the next flow from the stack, shown in the rule `F-next-flow`.
5. If the flow stack is empty, all flows reach synchronization status, then GKLEE$_p$ starts checking various kinds of errors and regrouping all the flows produced in the current BI into the stack. The rule `NextBI` depicts the details.

### 3.5.4 Soundness and Completeness of Parametric Execution

After the description of the state model and the semantics of flow-level operations, we need to know when our parametric execution w.r.t. SIMD execution is sound and complete. First we will introduce SIMD execution here.

#### 3.5.4.1 SIMD Execution

We describe the SIMD execution of a CUDA program w.r.t the data store $\sigma$. Naturally, we view that all the $n$ threads access $\sigma$ simultaneously (rather than that each thread perform the thread sequentially). We introduce the concept of *value vector* to depict the values related to $n$ threads. A value vector $v$ is a size-$n$ vector whose $i^{th}$ element (denoted by $v(i)$) stores the value for thread $i$. For example, value vector $\langle 2, 4, \ldots, 2n \rangle$ indicates that the value pertaining to thread $i$ is $2i$.

SIMD computations are parametric over the threads. In general, a computation can be modeled by a function $f$ mapping the input to output. In CUDA, the threads perform the same computation on different data. Each thread is supposed to execute the same instruction sequence (although they may diverge on conditions). By definition, a parametric computation can be modeled by a function $\lambda tid.\ f(tid)$. The result of the computation at thread $i$ is modeled by $f(i)$. In other words, the computation result at each thread $i$ can be obtained by instantiating the $\lambda$ function with $i$. Hence thread $i$'s computation is $\alpha$-*equivalent* to thread $j$'s. We call such a computation a *thread parametric* computation. An expression obtained by a thread parametric computation is a thread parametric expression.

#### 3.5.4.2 Basic Operations

We start with the cases without branches. The instructions can be divided into three categories: (1) reads from the store, (2) writes to the store, and (3) arithmetic computations. Arithmetic computations occur on local stores.

Consider an SIMD read instruction: "v = read$_\tau$ (f(tid))," where address $f(tid)$ is a function of the thread id $tid$, and type $\tau$ marks whether the read is local (when $\tau = l$) or global (when $\tau = g$). Thread $i$ reads the value $v_i$ from the store, *e.g.,* $v(i) = \sigma^\tau[f(i)]$. We can image that all the reads are done simultaneously, resulting in a value vector $< \sigma[f(0)], \sigma[f(1)], \ldots, \sigma[f(i-1)] >$.

```
thread 0    thread 1    ...    thread n-1
read f(0)   read f(1)   ...    read f(n-1)
```

For an $m$-arity operation $op$, each thread operates on its part of the data and produces an output value. That is, $n$ threads consumes $n$ value vectors $v_1,\ldots, v_m$ and produce a new one: $\langle op(v_1(i), \ldots, v_m(i)), \ldots, op(v_1(n-1), \ldots, v_m(n-1))\rangle$.

Next, consider an SIMD write instruction "write$_\tau$ (f(tid)) (v(tid))," where address $f(tid)$ and value $v(tid)$ are functions of the thread id $tid$. Thread $i$ write the value $v(i)$ to the store at address $f(i)$. We can image that all the writes are done simultaneously, but need to consider possible data races. To stress the simultaneousness, we introduce notation $\sigma^\tau[f(0) \mapsto v(0) \mid f(1) \mapsto v(1) \mid \ldots \mid f(n-1) \mapsto v(n-1)]$ (and $\sigma^\tau[f(i) \mapsto_{i\in[0,n-1]} v(i)]$ for short) to denote the SIMD update. If no data races exist, then the writes can be executed in a sequential order from thread 0 to $n-1$ and then result in a new store $\sigma^\tau[f(0) \mapsto v(0)][f(1) \mapsto v(1)] \ldots [f(n-1) \mapsto v(n-1)]$, which is equivalent to $\sigma^\tau[f(i) \mapsto_{i\in[0,n-1]} v(i)]$. Any other order is equivalent to this one as long as there exist no data races.

For a local write, each thread updates its local store; hence we have the following. A global write is more complicated since it may involve multiple threads.

$$\begin{array}{l} \sigma^l[f(i) \mapsto_{i\in[0,n-1]} v(i)] = \\ \langle \sigma^l_0[f(0) \mapsto v(0)], \ldots, \sigma^l_{i-1}[f(i-1) \mapsto v(i-1)]\rangle \end{array}$$

Consider the following instruction sequence, which first reads two values from the store, then performs an operation $op$, and then writes the result back into the store. After that, a read occurs on the updated store. This is a general case of the statement at line 7 of the reduction kernel (Figure 3.9) where $f_1 = f_3 = \lambda t.\, t$, $f_2 = \lambda t.\, t + s$ and the op is the

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

**Figure 3.9**: Reduction kernel.

addition operation.

$$\begin{array}{ll} 1: & v_1 = \text{read } f_1(tid); \\ 2: & v_2 = \text{read } f_2(tid); \\ 3: & v_3 = op(v_1, v_2); \\ 4: & \text{write } f_3(tid)\ v_3 \\ 5: & v_4 = \text{read } f_4(tid); \end{array}$$

Using the notations introduced above, we can depict the value vector evolves when $n$ threads execute this sequence.

$$\begin{array}{ll} \text{after 1}: & \langle \sigma_0[f_1(0)], \sigma_0[f_1(1)], \ldots, \sigma_0[f_1(n-1)] \rangle \\ \text{after 2}: & \langle \sigma_0[f_2(0)], \sigma_0[f_2(1)], \ldots, \sigma_0[f_2(n-1)] \rangle \\ \text{after 3}: & \langle op(\sigma_0[f_1(0)], \sigma_0[f_2(0)]), \ldots, op(\sigma_0[f_1(n-1)], \sigma_0[f_2(n-1)]) \rangle \\ \text{after 4}: & \sigma_1 = \sigma_0[f_3(i) \mapsto_{i \in [0,n-1]} op(f_1(i), f_2(i))] \\ \text{after 5}: & \langle \sigma_1[f_4(0)], \sigma_1[f_4(1)], \ldots, \sigma_1[f_4(n-1)] \rangle \end{array}$$

It is easy to see that for thread $i$, the values $v_1$ and $v_2$ are $\sigma_0[f_1(i)]$ and $\sigma_0[f_2(i)]$, respectively. That is, these two values are thread parametric. So does value $v_3$. In contrast, $v_4$'s value may or may not be thread parametric. For instance, for the statement sdata[tid] += sdata[tid + s] in the reduction kernel, the elements in the resulting $sdata$ are not thread parametric.

We need to know when our parametric method is sound and complete. Since local reads/writes and global reads are always thread parametric, we focus on global writes.

#### 3.5.4.3 Global SIMD Write

We consider only SIMD writes that will not incur races. That is, for $\sigma^\tau[f(i) \mapsto_{i \in [0,n-1]} v(i)]$, we have $\forall i, j \in [0, n-1].\, f(i) \neq f(j)$. SIMD writes and reads satisfy the following property. Specifically, if there exists a thread $j$ whose write address matches the read address of thread $i$, then $i$ obtains the value written by thread $j$. Otherwise, $i$ gets the value from the old global store.

$$
\begin{array}{l}
(\sigma^g[f_1(i) \mapsto_{i\in[0,n-1]} v(i)])[f_2(i)] = \\
\quad \begin{cases} v(j) & \texttt{if } \exists j \in [0, n-1].\, f_1(j) = f_2(i) \\ \sigma^g[f_2(i)] & \texttt{if } \forall j \in [0, n-1].\, f_1(j) \neq f_2(i) \end{cases}
\end{array}
$$

Global SIMD updates play an important role in parametric analysis. There are two cases whether such an update can be removed:

$$
\begin{array}{ll}
\text{case 1}: & f_1(i) = f_2(i) \Rightarrow \\
& (\sigma^g[f_1(i) \mapsto_{i\in[0,n-1]} v(i)])[f_2(i)] = v(i) \\
\text{case 2}: & (\forall j \in [0, n-1].\, f_1(j) \neq f_2(i)) \Rightarrow \\
& (\sigma^g[f_1(i) \mapsto_{i\in[0,n-1]} v(i)])[f_2(i)] = \sigma^g[f_2(i)]
\end{array}
$$

In the first case, thread $i$ obtains the value from its previous write since the addresses of the read and the write are equal. In the second case, the SIMD update does not affect the value since the address of the read does not match that of the write by any thread. In both cases, the read after an SIMD update is *resolved* so that the value obtained by the read is not dependent on the SIMD update. We say that such a read is a *resolved* read.

**Theorem 1** *If all the reads in an expression $e$ are* resolved*, then $e$ is* thread parametric.

**Corollary 2** *If a value $v$ involves no global SIMD updates, then $v$ is* thread parametric.

#### 3.5.4.4 Proposition

If address expressions involved in shared/global memory accesses are resolvable, parametric computation is sound and complete.

Parametric execution symbolically executes with respect to two-thread abstraction; it executes the following steps:

- Starting with the state of one symbolic thread with thread ID $t_0$ where $t_0$ is fully symbolic ($t_0 \in [0...n-1]$ where $n$ means the number of threads), symbolic thread $t_0$ executes along with a flow $E_1$, and other threads in this flow are unupdated; their states remain fully symbolic (called lazy symbolization).
- Cloning $E_1$ and substituting $t_1$ for $t_0$ everywhere, then we are given the other flow $E_2$.
- Checking whether $E_1$ and $E_2$ may incur a race under the assumption $(t_0 \neq t_1)$.

Two-thread abstraction might result in false alarms or omissions. If there exist unresolvable address expressions in shared/global memory accesses, unresolvable reads will load unupdated values. Different from the "havoced" value (more general unknown variable 'X'), those unupdated values might fool SMT solvers to make incorrect decision to execute "infeasible" paths or miss "feasible" paths, resulting in false alarms or omissions. Two examples are present below to demonstrate the false alarms and omissions caused by incorrect SMT derivation of *unresolvable* values.

Figure 3.10 presents the false alarm incurred by *unresolvable* variable `S[i]`. In this example, after the update of A1 and A2, each element of array `S` is expected to be greater than or equal to 10, so the branch including race will not be executed; GKLEE verifies so.

```
#include <stdio.h>

#define NUM 4

__global__ void test(int *dOut) {
  __shared__ int S[NUM];
  size_t i = threadIdx.x;
  S[i] = 0;
  __syncthreads();

  S[i] += 10; // A1
  __syncthreads();

  if (i < blockDim.x/2) {
    S[i] += S[i+blockDim.x/2]; // A2
  }
  __syncthreads();

  if ( S[i] >= 10 ) {
    S[i] -= 10;
  } else {
    S[i] += S[(i+1)%blockDim.x]; // race
  }
  __syncthreads();

  dOut[i] = S[i];
}

int main(void) {
  int *dOut;
  cudaMalloc((void**)&dOut, sizeof(int)*NUM);
  test<<<1, NUM>>>(dOut);
  return 0;
}
```

**Figure 3.10**: False alarms incurred by *unresolvable* values.

Under parametric computation, *unresolvable* `S[i]` fools SMT solver to explore both of the branches, leading to a "false" race.

We contrived a similar example shown in Figure 3.11 to demonstrate the omission caused by an *unresolvable* value. In the concrete scenario, after the operations in A1 and A2, thread 0 and 1 are supposed to execute the "then" branch and the rest will execute the "else" path, resulting in a race. But, under parametric computation, after A1 operation, all threads are expected to be updated to 10. Suppose that we do not abstract shared state; then because every thread *tid* only writes 10 to its corresponding element $S[tid]$ and has not modified $S[tid + blockDim.x/2]$, we erroneously preserve the condition that $S[tid + blockDim.x/2] = 0$ in A2 position. In this way, SMT solver always evaluates conditional $S[i] == 20$ false so that only false branch is executed; thus a race will not be reported.

To avoid omissions, we can "havoc" (set to a fresh symbolic value) the value of a read over global SIMD update. (Our current release of GKLEE$_p$ does not have this facility, yet.) The "havoced" values can overapproximate all nondeterminism. The evaluation result of SMT solvers for "havoced" values will absolutely subsume the expected result and the unexpected result, leading to false alarms but no omissions. To warrant soundness, we may use global invariants as in [14, 28], which often require manual effort.

If *unresolvable* memory accesses do not exist, it is guaranteed that each thread loads the value not written by other threads, and these values are merely updated by the thread itself. Then we can safely claim that the representative thread $t_0$'s memory accesses are *rename-equivalent* to each explicitly modeled thread's memory accesses. Consequently, parametric computation based on the representative symbolic thread is sound and complete if address expressions involved in shared/global memory accesses are resolvable.

## 3.6 Experimental Results

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

```
#include <stdio.h>

#define NUM 4

__global__ void test(int *dOut) {
  __shared__ int S[NUM];
  size_t i = threadIdx.x;
  S[i] = 0;
  __syncthreads();

  S[i] += 10; // A1
  __syncthreads();

  if (i < blockDim.x/2) {
    S[i] += S[i+blockDim.x/2]; // A2
  }
  __syncthreads();

  if ( S[i] == 20 ) {
    S[i] = S[i+1]; //race
  } else {
    S[i] = 5;
  }
  __syncthreads();

  dOut[i] = S[i];
}

int main(void) {
  int *dOut;
  cudaMalloc((void**)&dOut, sizeof(int)*NUM);
  test<<<1, NUM>>>(dOut);
  return 0;
}
```

**Figure 3.11**: Omissions incurred by *unresolvable* values.

Table 3.1 presents results from SDK 2.0 kernels while Table 3.2 presents those from SDK 4.0 (many of these are also available in 2.x). Here, #T denotes the number of threads analyzed. Each cell contains a WW (write-write race), Ben. (benign race, meaning the same value written by two concurrent writes), a Y or N (yes/no), or two numbers of the form A/B, where A is the tool runtime (in seconds) and B is the number of control-flow paths analyzed (*i.e.*, the TICs branched in so many ways). Most examples only generate one path, as there are no data-dependent control flow variations (except Bitonic sort, where these variations are essential to sorting). In Table 3.1, we set 7200 seconds as the threshold for time out (abbreviated as T.O.). “BC” and “MC” are the abbreviations of “Bank Conflict”

**Table 3.1**: SDK 2.0 kernel results.

| **Kernels** | Race | #T = 32 | | #T = 64 | | #T = 256 | | #T = 1,024 | | #T = 2,048 | | BC | MC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | GKLEE | GKLEE$_p$ | GKLEE | GKLEE$_p$ | GKLEE | GKLEE$_p$ | GKLEE | GKLEE$_p$ | GKLEE | GKLEE$_p$ | | |
| Bitonic Sort | | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | T.O. | N | Y |
| Histogram64 | WW | 15.8/10 | 25.9/9 | 23.2/1 | 25.8/1 | 75.4/1 | 120/1 | 725.3/1 | 387.6/1 | 2,682.0/1 | 904.6/1 | Y | Y |
| Scalar Product | | 0.7/1 | 16.1/1 | 0.6/1 | 4.3/1 | 0.8/1 | 0.8/1 | 1.3/1 | 0.9/1 | 2.6/1 | 1.2/1 | N | Y |
| Matrix Mult | | 0.2/1 | 4.5/1 | 0.4/1 | 4.0/1 | 2/1 | 3.2/1 | 19/1 | 2.8/1 | 362.1/1 | 3.4/1 | N | Y |
| Reduction0 | | 0.02/1 | 0.07/1 | 0.1/1 | 0.03/1 | 0.3/1 | 0.2/1 | 2.9/1 | 0.3/1 | 10.5/1 | 0.4/1 | N | Y |
| Reduction1 | | 0.01/1 | 0.1/1 | 0.1/1 | 0.1/1 | 0.8/1 | 0.2/1 | 8.1/1 | 0.3/1 | 24.0/1 | 0.5/1 | Y | Y |
| Reduction2 | | 0.02/1 | 0.1/1 | 0.03/1 | 0.1/1 | 0.2/1 | 0.1/1 | 2.9/1 | 0.3/1 | 10.2/1 | 0.4/1 | N | Y |
| Reduction3 | | 0.01/1 | 0.1/1 | 0.03/1 | 0.1/1 | 0.3/1 | 0.1/1 | 2.7/1 | 0.3/1 | 10.0/1 | 0.4/1 | N | Y |
| Reduction4 | | 0.1/1 | 0.04/1 | 0.3/1 | 0.03/1 | 2.8/1 | 0.2/1 | 17.3/1 | 0.4/1 | 42.4/1 | 0.6/1 | N | Y |
| Reduction5 | | 0.1/1 | 0.04/1 | 0.3/1 | 0.03/1 | 2.8/1 | 0.2/1 | 11.4/1 | 0.4/1 | 21.3/1 | 0.5/1 | N | Y |
| Reduction6 | | 0.1/1 | 0.05/1 | 0.3/1 | 0.04/1 | 2.8/1 | 0.2/1 | 11.5/1 | 0.4/1 | 22.6/1 | 0.6/1 | N | Y |
| Scan Best | | 0.3/1 | 3.6/1 | 2.1/1 | 5.1/1 | 48.8/1 | 8.1/1 | 923.3/1 | 12.5/1 | T.O. | 26.6/1 | Y | Y |
| Scan Naive | | 0.04/1 | 0.2/1 | 0.2/1 | 0.4/1 | 3.4/1 | 0.5/1 | 66.0/1 | 0.9/1 | 291.8/1 | 15.2/1 | N | N |
| Scan WorkEfficient | | 0.1/1 | 0.6/1 | 0.4/1 | 0.8/1 | 12.1/1 | 1.2/1 | 250.8/1 | 2.1/1 | T.O. | 3.1/1 | Y | N |
| Scan Large | | 0.2/1 | 2.3/1 | 1.4/1 | 3.0/1 | 40.0/1 | 3.9/1 | 736.1/1 | 2.1/1 | T.O. | 2.1/1 | Y | Y |
| Bisect Small | Ben. | 2.2/1 | 105.9/1 | 3.5/1 | 108.8/1 | 10.6/1 | 108.7/1 | 36.0/1 | 108.8/1 | 58.1/1 | 233.7/1 | N | Y |
| Bisect Large | Ben. | T.O. | 226.0/1 | T.O. | 203.0/1 | T.O. | 212.6/1 | T.O. | 218.5/1 | T.O. | 248.1/1 | Y | Y |

**Table 3.2**: SDK 4.0 kernel results.

| **Kernels** | Race | #T = 1,024 | | #T = 2,048 | | #T = 4,096 | | #T = 8,192 | | #T = 16,384 | | BC | MC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | GKLEE | GKLEE$_p$ | GKLEE | GKLEE$_p$ | GKLEE | GKLEE$_p$ | GKLEE | GKLEE$_p$ | GKLEE | GKLEE$_p$ | | |
| Clock | | 3.8/1 | 12.1/1 | 6.1/1 | 12.2/1 | 9/1 | 12.6/1 | 26.6/1 | 13/1 | 92.2/1 | 13.9/1 | N | Y |
| Scalar Product | | 50.9/1 | 97.9/1 | 213.2/1 | 200.2/1 | 902.9/1 | 410.9/1 | T.O. | 859.4/1 | T.O. | 1,812.2/1 | N | Y |
| Histogram64 | | 122.3/1 | 50.8/1 | 158.7/1 | 55.7/1 | 283.8/1 | 65.9/1 | 511.8/1 | 85.1/1 | T.O. | 120.0/1 | Y | N |
| Scan Short | | 36.3/1 | 18.1/1 | 92.5/1 | 32.3/1 | 216.4/1 | 62.6/1 | 714.8/1 | 116.7/1 | 3,222.7/1 | 227.0/1 | Y | N |
| Scan Large | | 40.1/1 | 92.9/1 | 107.5/1 | 133.9/1 | 336.6/1 | 482.2/1 | 1,175.1/1 | 761.4/1 | 6,134.4/1 | 555.7 | Y | N |
| Copy | | 0.1/1 | 0.1/1 | 0.3/1 | 0.1/1 | 0.8/1 | 0.1/1 | 2.8/1 | 0.1/1 | 10.2/1 | 0.1/1 | N | Y |
| copySharedMem | | 0.8/1 | 0.3/1 | 1.6/1 | 0.6/1 | 11.1/1 | 0.3/1 | 23.2/1 | 0.7/1 | 172.7/1 | 0.6/1 | N | Y |
| transposeNaive | | 0.2/1 | 0.2/1 | 0.3/1 | 0.1/1 | 1.0/1 | 0.1/1 | 3.4/1 | 0.2/1 | 11.6/1 | 0.2/1 | N | N |
| transposeCoalesced | | 4.7/1 | 0.2/1 | 9.6/1 | 0.3/1 | 27.3/1 | 0.3/1 | 55.3/1 | 0.4/1 | 242.4/1 | 0.5/1 | Y | Y |
| transposeNoBankConflicts | | 0.8/1 | 0.4/1 | 1.8/1 | 0.4/1 | 11.3/1 | 0.4/1 | 24.1/1 | 0.5/1 | 179.2/1 | 0.7/1 | N | Y |
| transposeDiagonal | | 0.8/1 | 0.3/1 | 1.7/1 | 0.4/1 | 11.2/1 | 0.4/1 | 23.5/1 | 0.5/1 | 172.1/1 | 0.6/1 | N | Y |
| transposeFineGrained | | 0.8/1 | 0.3/1 | 1.7/1 | 0.4/1 | 11.1/1 | 0.4/1 | 23.2/1 | 0.5/1 | 170.0/1 | 0.6/1 | N | Y |

and “Coalesced Global Memory Accesses,” and the results of these two categories are acquired through GKLEE$_p$.

Note that for Histogram64, GKLEE or GKLEE$_p$ explore multiple paths when $\#T = 32$, even though this example does not contain data-dependent control flows. The reason for path generation is due to out-of-bound memory accesses happening (these generate a case analysis as explained in [30]). As another example, matrix multiplication using SDK 2.0 takes 362 seconds to explore the sole path using GKLEE, while it only takes 3.4 seconds under GKLEE$_p$.

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

Many of our results were obtained with respect to symbolic inputs. For instance, as reported in [31] for runs using GKLEE, the Histogram64 example’s race will be almost impossible to detect unless the first 10 bytes of a certain array are made symbolic (the same symbolic setting was used in runs using GKLEE$_p$ also).

### 3.6.1 Tables (3.1 and 3.2 )

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

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

As for data races, (iii) all races listed in [31] were also detected by GKLEE$_p$. We also found additional interwarp write-write races in SDK 2.0 kernels, thanks to the fact that we ran those examples with more than one thread block.

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

Figure 3.12, 3.13, 3.14, and 3.15 show that GKLEE$_p$ outperforms GKLEE with respect to different scales of number of threads. When $\#T = 8K$, GKLEE$_p$ speeds up matrix multiplication by a factor of 300 times. When $\#T = 16K$, GKLEE$_p$ speeds up the kernel `transposeCoalesced` by a factor of around 500 times. Figure 3.14 and Figure 3.15 illustrate `ScanLarge` and `Clock` benchmark in which GKLEE$_p$ performs several times faster than GKLEE.

### 3.6.2 New Results on Histogram64

In [31] we reported that GKLEE identified a possible WW race occurring within a warp. It has been an open question whether such race will manifest in the interwarp cases (we could not run these larger models using GKLEE). GKLEE$_p$ checks whether two threads from

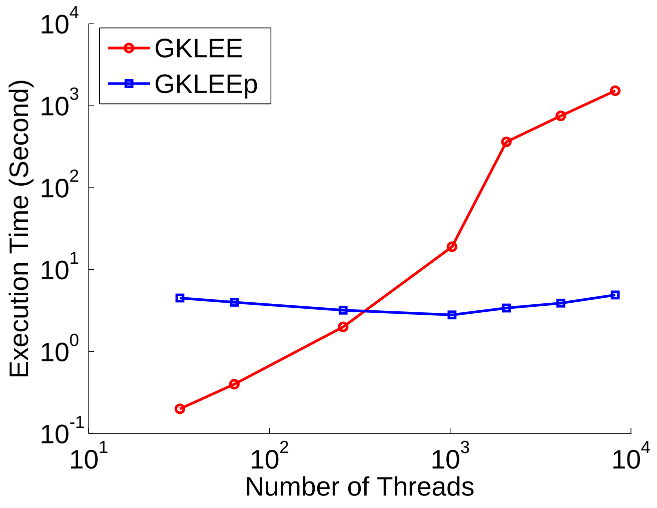

**Figure 3.12**: Matrix Multiplication (SDK 2.0, LOC 60).

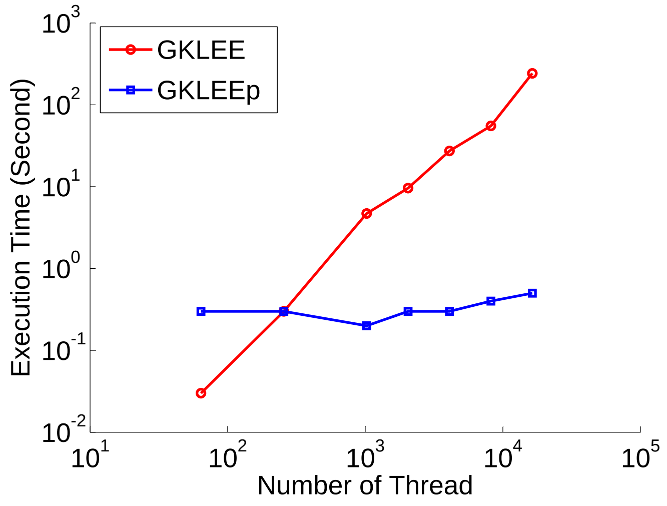

**Figure 3.13**: TransposeCoalesced (SDK 4.0, LOC 23).

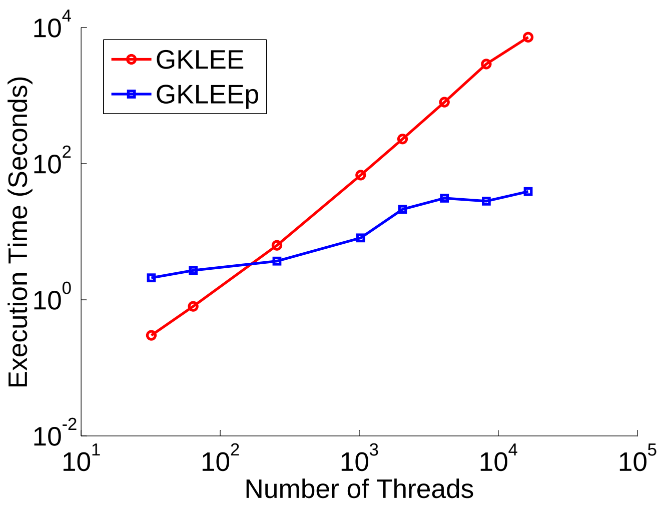

**Figure 3.14**: Scan Large (SDK 2.0, LOC 196).

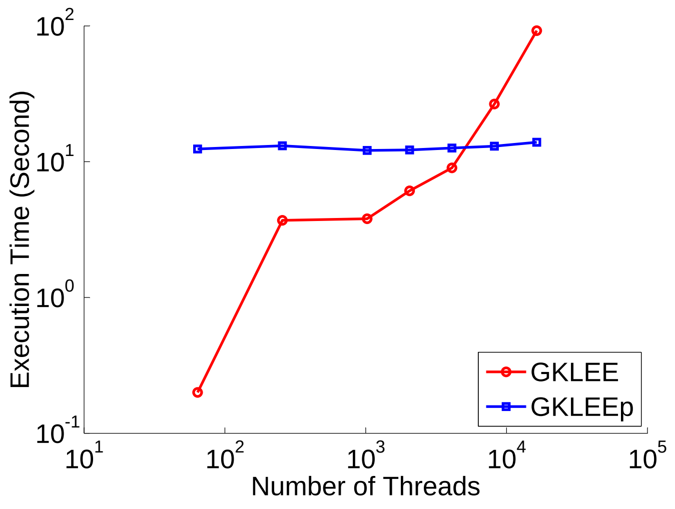

**Figure 3.15**: Clock (SDK 4.0, LOC 38).

different warps may cause such a race and confirms that it will not (this race is present only within a warp). This demonstrates the added analysis power offered by GKLEE$_p$.

# CHAPTER 4

# COMBINING ABSTRACTION WITH STATIC ANALYSIS

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

Our recent tool GKLEE$_p$ [32] provides more scalability by exploiting thread symmetry, but still requires users to pick the symbolic inputs. In particular, this tool partitions the space of executions of a GPU program into *parametric flows*, and models the race analysis problem over *two parametric threads* in each equivalence class. GKLEE$_p$ can scale to thousands of threads for *nondivergent programs* (*e.g.,* programs not forking paths with respect to symbolic inputs). Unfortunately, GKLEE$_p$ still suffers from search explosion: even if only two symbolic threads are considered, each thread may create a large number of flows or symbolic states. For instance, if a thread contains $n$ feasible branches, then $O(2^n)$ flows or paths may be generated. As shown later, this scenario is not uncommon in realistic CUDA programs. We present a new tool SESA (Symbolic Executor with Static Analysis) that significantly improves over prior tools in its class both in terms of new ideas and new engineering:

1. SESA implements a new front-end (based on Clang). It also supports all core CUDA C++ instructions, 95 arithmetic intrinsics, 25 type conversion intrinsics, all atomic

intrinsics, and 82 CUDA runtime functions.

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

## 4.1 Background

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

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

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

$$\begin{array}{lll} & \text{thread } t_1 & \text{thread } t_2 \\ \text{WriteSet}: & \{v[t_1.x]\} & \{v[t_2.x]\} \\ \text{ReadSet}: & \{v[(t_1.x+1)\%bdim.x]\} & \{v[(t_2.x+1)\%bdim.x]\} \end{array}$$

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

$$\begin{array}{lll} & \text{thread } t_1 & \text{thread } t_2 \\ \text{WriteSet}: & \{t_1.x\%2 \neq 0\ ?\ v[t_1.x \gg 2]\} & \{t_2.x\%2 \neq 0\ ?\ v[t_2.x \gg 2]\} \\ \text{ReadSet}: & \{t_1.x\%2 = 0\ ?\ v[t_1.x]\} & \{t_2.x\%2 = 0\ ?\ v[t_2.x]\} \end{array}$$

SESA inherits several features from its predecessors [31, 32]. In particular, it has the ability to do race-checking under standard warp-sizes, or a warp-size of $1$. The latter option is important because many programmers rely on warp semantics and run into inexplicable races, as studied and explained in [45, 52]. A CUDA compiler can "assume" that the warp size is $1$, and may miscompile codes that race under this view. SESA also checks for global memory races—a feature missing in commercial tools such as [35].

## 4.2 New Techniques

To explain the features in SESA, consider three examples, namely Generic (Figure 4.1), Reduction (Figure 4.2), and Bitonic (Figure 4.3). One of the central problems of GKLEE stemmed from its explicit modeling of every thread in a CUDA program. GKLEE$_p$ improved the situation by capitalizing on the symmetry inherent in a CUDA program. For example, consider the Reduction example in Figure 4.2. If there is a data race experienced by a thread satisfying the condition listed on line 3 such a race will also be experienced by a group of threads satisfying this condition. Thus, GKLEE$_p$ splits the threads into two equivalence classes at line 3 and models *two symbolic threads* $t_1$ *and* $t_2$ *for each of these equivalence classes*. In [32], it was shown that GKLEE$_p$ can simply proceed on the assumption that $t_1 \neq t_2$ and detect the same class of races as GKLEE and thus avoid directly facing the complexity of modeling the actual number of threads (hence the name "parametric flows"). However, this approach of GKLEE$_p$ can generate an exponential number of flows (four flows in Generic, and many more in the others). In Section 4.2.1 and Section 4.2.2, we explain how such flows are combined by SESA through some examples.

### 4.2.1 Flow Combining with Respect to Local Variables

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

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

**Figure 4.1**: Contrived generic kernel.

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

**Figure 4.2**: Reduction kernel.

```
__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 4.3**: Bitonic kernel.

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

### 4.2.2 Flow Combining with Respect to Global Variables

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

### 4.2.3 Symbiotic Use of Static and Symbolic Analysis

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

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

In our results section §4.5, we show that the combination of the aforesaid static analysis and the flow combining approaches of §4.2.1 and §4.2.2 were essential to handle practical benchmarks such as the Lonestar benchmark.

## 4.3 Parametric Execution and Race Checking

Figure 4.4 shows the infrastructure of SESA. The front-end uses the `Clang-3.2` compiler to translate a CUDA program into LLVM bytecode (with this front-end, support for OpenCL [38] is within reach and is being planned). The bytecode is first processed by the static analyzer (§4.4) to add annotations on data-flow information, specifically whether a variable can flow into sensitive sinks. The annotated bytecode is then interpreted by the symbolic executor during parametric execution (§4.3.1), at which time, races are checked (§4.3.2). The annotated bytecode contains information about what inputs should be symbolic as well as flow into sensitive sinks; this guides the symbolic executor to carry out the functions described in §4.2 (§4.4). SESA currently supports all core LLVM instructions, 95 arithmetic intrinsics, 25 type conversion intrinsics, all atomic intrinsics, and 82 CUDA runtime functions. This amounts to $\sim$3K LOC new for the basic infrastructure. This represents substantial effort in making a practical C++ CUDA front-end; existing formal approaches to GPU correctness do not have these features.

### 4.3.1 Parametric Execution of CUDA Programs

In this section, we recap the essential ideas of parametric execution introduced in GKLEE$_p$ and point out key innovations made in this work. We present the basics of the CUDA syntax handled (as captured at the LLVM level), our store model, how SIMD execution is carried out, and the basics of preserving parametricity. In §4.3.2, we present

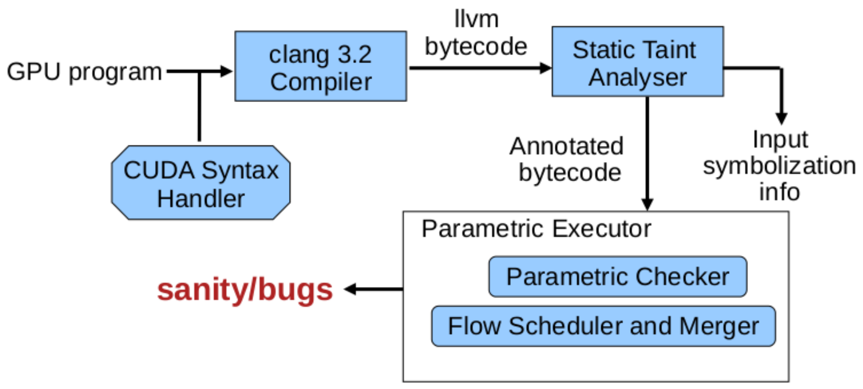

**Figure 4.4**: SESA's infrastructure.

the highlights of parametric race checking.

Figure 4.5 shows an excerpt of the syntax of CUDA LLVM bytecode. We now provide a high level view of how a collection of CUDA threads execute from the perspective of race checking. For this, we focus on the “Race State” of each thread. Given a thread ID and a flow conditional (flow_cond), a single access is either a read $(r)$ or a write $(w)$ followed by the address being accessed. For brevity, consider a sequence of race states of the form $c_1?r(a_1), c_2?r(a_2), c_3?w(a_3), \ldots$ that a single thread evolves over. We now define the notion of a *Parametric Execution*.

### 4.3.1.1 Parametric Execution

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

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

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

Now, when can we claim that regardless of the number of threads executing a barrier interval, we can detect all data races within the barrier interval by simply modeling two

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

**Figure 4.5**: Summary syntax of CUDA bytecode.

symbolic threads $t_i$ and $t_j$ and checking races across just these threads (this is the key idea of parametric checking). In §4.3.2, we proceed to describe that these ideas can be used to efficiently perform race checking without incurring parameterized flow explosion.

### 4.3.2 Parametric Flows and Race Checking

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

Suppose for now there are no symbolic inputs such that $c$ is a function of thread id $tid$. This condition divides the threads into two groups, one satisfying $c$ and one satisfying $\neg c$. We say that this condition creates two parametric flows. Each flow represents a group of threads with a flow condition. Since the threads within a flow perform similar operations, they can be reasoned about using a parametric thread.

A parametric flow represents a set of threads that perform parametric computations under a flow condition. In the case of multiple conditions $c_1$, $c_2$, ..., $c_k$, each condition $c_i$ partitions the threads into two flows. In general, the number of flows can grow exponentially. Here are two facts about the number of flows:

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

Now we describe race checking during parametric execution. For barrier interval, one can follow a canonical (sequential [7]) schedule. That is, we symbolically execute one thread from one barrier to the other, then switch to the other thread and do the same, etc. The fact that this single sequential schedule [7] is sufficient is argued in [4, 8, 28, 31]. Both GKLEE$_p$ and SESA carry this sequential schedule using *one symbolic thread*. The resulting race history is cloned and instantiated over two thread IDs $t_1$ and $t_2$ with $t_1 \neq t_2$. This idea is applied within each *parametric flow* described below.

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

Figure 4.6 shows the parametric flow tree of the reduction kernel. The initial flow represents all $bdim.x$ threads. Consider the first loop iteration where $s = 1$. At line 3, the threads are divided into two groups upon condition $tid\%2 = 0$: one with thread ids $\{0, 2, 4, \ldots\}$, and the other one with $\{1, 3, 5, \ldots\}$. Accordingly, SESA creates two flows: flow $F_1$ with flow condition $tid\%2 = 0$ and flow $F_2$ with $tid\%2 \neq 0$. Then, SESA can execute $F_1$, producing read set $\{tid\%2 = 0\ ?\ sdata[tid + 1], tid\%2 = 0\ ?\ sdata[tid]\}$ and write set $\{tid\%2 = 0\ ?\ sdata[tid]\}$. When this flow reaches the barrier, flow $F_2$ is scheduled to execute. Since $F_2$ contains no computation before the barrier, its read set and write set are empty. Now all the flows for the first barrier interval reach the barrier. We union the read-sets of $F_1$ and $F_2$ to produce the barrier interval read set: $\{tid\%2 = 0\ ?\ sdata[tid + 1], tid\%2 = 0\ ?\ sdata[tid]\}$. Similarly we obtain the barrier interval write-set: $\{tid\%2 = 0\ ?\ sdata[tid]\}$. Then we instantiate these barrier interval sets with two symbolic threads $t_1$ and $t_2$ for race checking (for simplicity, we do not show the extra assumption $t_1, t_2 < bdim.x$ here). Since the solver returns "unsat," no race is found.

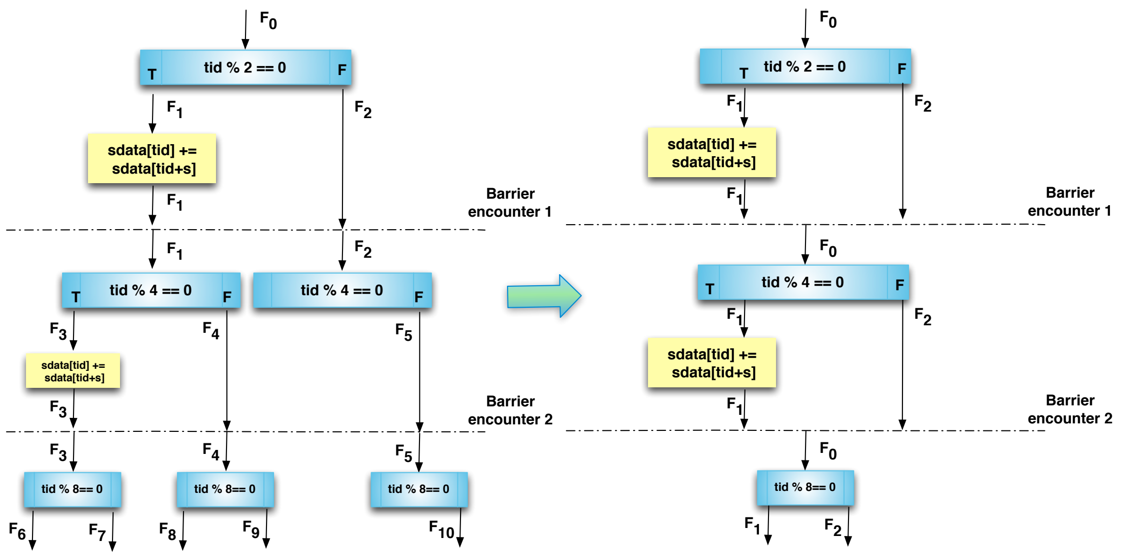

**Figure 4.6**: Parametric flows of kernel reduction, and how flows are combined.

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

Suppose flow $F_1$ is executed first in the next barrier interval. Again two new flows are generated upon condition $tid\%4 = 0$. The leftmost flow $F_3$ has flow condition $tid\%2 = 0 \wedge tid\%4 = 0$, which is simplified to $tid\%4 = 0$. For flow $F_4$ with condition $tid\%2 \neq 0$, since $tid\%4 = 0$ conflicts with $tid\%2 \neq 0$, *e.g.,* $tid\%2 \neq 0 \Rightarrow tid\%4 \neq 0$, SESA keeps only one flow with condition $tid\%2 \neq 0$. Finally we have five flows at barrier 2. Each flow represents a group of threads, *e.g.,* $F_6$ represents threads $\{0, 4, 8, \dots\}$. Then we can obtain the barrier interval read/write sets by uniting the flows' read/write sets, instantiating them with two threads, and checking address overlapping for possible races. In the implementation, when two flows are generated, we can reuse the current flow for one of the generated flows so as to reduce the flow cloning cost. For example, we can reuse $F_0$ for $F_1$, $F_3$ and $F_6$, and $F_2$ for $F_5$ and $F_{10}$.

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

SESA also supports warp execution as per CUDA's SIMD model. The threads within a warp are executed in a lock-step manner: two intrawarp threads can race only if they simultaneously write to the same shared variable at the same instruction. In case of divergence, we execute the two sides sequentially and merge them at the first convergence point (*e.g.,* the nearest common postdominator). When a conditional instruction $c \ ? \ is_1 : is_2$ is encountered, SESA forks two new flows $F_1$ and $F_2$, representing the two branches of the condition, and subsequent executions will start from each one. At $F_1$, after $is_1$ is executed and the convergence point is reached, SESA executes $is_2$ immediately. That is, flow $F_1$ executes $c \ ? \ is_1$ followed by $\neg c \ ? \ is_2$. Similarly, flow $F_2$ executes $\neg c \ ? \ is_1$ followed by $c \ ? \ is_2$. For example, when the threads diverge on the condition at line 1 in the Generic Example of Figure 4.1, the order "$F_1; F_2$" produces $v = b$ while the order "$F_2; F_1$" produces $v = a$. The executor has to enumerate both orders to be complete (unless data-flow analysis helps eliminate one).

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

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

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

Clearly, all reads in a computation are resolvable if this computation contains no global SIMD writes. A read over an SIMD write may be still resolvable by analyzing the relation of their addresses, *e.g.,* our prior work [29] uses SMT solving to reason about resolvable reads. Currently, GKLEE resolves only the global reads and writes performed by the same (parametric) thread.

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

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

If an address expression involved in shared/global memory updates are unresolvable, then parametric checking may be unsound and incomplete. To avoid omissions, we can "havoc" (set to a fresh symbolic value) the value of a read over global SIMD update. (Our current release of SESA does not have this facility, yet.) To warrant soundness, we may use global invariants as in [14, 28], which often require manual effort. In our results section §4.5, we indicate whether the race checking of a kernel involves unresolved SIMD writes and indicate whether soundess and completeness are affected if such writes exist.

## 4.4 Taint Analysis

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

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

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

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

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

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

Each variable $v$ is associated with a live variable set (LVS), which records the live variables used by $v$. For each new instruction, we apply the taint propagation rules of Figure 4.7, where we use $T$ as a short hand for LVS. One complication is about memory loads and stores. We introduce a notation $T_\mu[v]$ to represent the LVS of a variable whose memory address is $v$. We maintain a memory model to compute $T_\mu$ and use LLVM's pointer alias analysis to resolve memory accesses.

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

```
  C code                    LLVM code
char A[10];                store A 1 %1
A[1] = v1;                 %2 = load A 1
```

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

**Figure 4.7**: Taint propagation rules.

```
char c = A[1]+A[0];       %3 = load A 0
                          %4 = add %2 %3
```

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

### 4.4.1 Example 1

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

| | |
|---|---|
| Line 1: | T(v) = {a} |
| Line 2: | T(v) = {b} |
| Line 3: | T(v) = {a,b}, T(u) = {tid} |
| Line 4: | $T_{\mu}$(A[w]) = {a,b} |
| Finally: | TaintSet = {} |

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

### 4.4.2 Example 2

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

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

There are two program points of interest: $pnt1$ at the kernel entry and $pnt2$ at the barrier at line 5. Consider $pnt1$, where the live variables are the inputs including $sdata$. The analyzer's task is to, given a program point, determine which live variables will flow into the sinks. For $pnt1$, we check whether the live variables (*e.g.,* the inputs) can flow into the two shared memory accesses at line 4. The calculation is done by investigating each instruction and propagating the variables according to the rules shown in Figure 4.7.

Initially, the LVS of each variable is empty. At the first instruction, $\%1$'s LVS is empty since $\%1$ stores local variable $s$ (hence involving no global inputs). Instruction "%2 = cmp lt %1 bdim.x" propagates %1's LVS to %2, whose LVS is also empty. The "phi" instruction propagates %9's LVS to %3, and so on. At line 4, we check the addresses of $sdata[\%7]$ and $sdata[\%8]$ for possible races; hence the variables in %7's LVS and %8's LVS need to be marked as tainted variables. Since these two sets are empty, no variable will be marked, indicating that no inputs should be made symbolic.

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

The analysis results can also be used to remove parametric flows. At program point $pnt2$ at the barrier, variables $\%3, \%4, \%5, \%7$, and $\%8$ are determined to be not related to the sinks (*e.g.,* addresses in shared accesses), variable $\%2$ is not live, and variables $\%1$ and

$\%9$ have the same values for both flows; hence only one flow is needed to be explored during symbolic execution. Basically we check whether the writes in a BI (barrier interval) will be used in subsequent BIs. A live variable not updated in any flow within the BI can be ignored since its value will be the same in all flows.

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

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

## 4.5 Experimental Results

We run SESA on the kernels in CUDA SDK, the Parboil library [3], and the LonestarGPU benchmark [2]. All experiments are performed on a machine with Intel(R) Xeon(R) CPU @ 2.40GHz and 12GB memory. We perform extensive comparisons of SESA and a start-of-the-art GPU testing tool GKLEE$_p$ [32]. The benchmarks are available at `https://sites.google.com/site/sesabench/`.

### 4.5.1 CUDA SDK

Table 4.1 shows results on a few CUDA SDK 5.5 kernels that have no thread divergence (hence both SESA and GKLEE$_p$ explore one flow). And no races are found. Timing results (sec.) are the average over three runs. For example, for kernel vectorAdd, SESA determines that none of the four inputs need to be symbolic, while a typical GKLEE$_p$ user sets two symbolic inputs (the other two are related to the concrete thread number). SESA finishes the execution for $50,176$ threads in $0.8$ second, while GKLEE$_p$ needs $3.8$ seconds. This shows the advantages of reduced symbolic inputs, even for small examples. The case of kernel matrixMul is similar.

For some benchmarks, while having excessive symbolic inputs does not affect performance (because most steps such as, *e.g.,* memory block matching are unrelated to

**Table 4.1**: CUDA SDK 5.5 nondivergent kernel results.

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

inputs), it causes other problems. For example, SESA detects no symbolic inputs for kernel scalarProd, while GKLEE$_p$ sets 2 symbolic inputs and crashes without giving any useful result.

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

### 4.5.2 Performance Improvement with Flow Optimizations

Table 4.2 presents the SESA's advantage over the GKLEE$_p$ for kernels whose execution produces many flows. SESA identifies a subset of inputs to be symbolized. Each result cell is of the form 'Num-Flows' ('elapsed-time-in-secs.'), and Time-Outs are at 3,600 seconds. 'Num-Flows' refers to the maximum of flows that occur during the execution. Column `RSLV?` represents if the kernel is *resolvable*. Column `RR/OM` denotes if Races were Reported by SESA or Omissions occur. A "-" means that these outcomes did not happen. stream compaction suffers from a false out-of-bound error and write-write race (manually confirmed). For stream compaction and n stream compaction kernels, 'Num-Flows' refers to the number of the execution paths. The last four benchmarks are from [14]. SESA's performance improvement becomes more significant when the number of threads becomes larger. For example, with 16 threads in mergeSort kernel, GKLEE$_p$

**Table 4.2**: Comparison of SESA and GKLEE$_p$.

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

explores $17$ flows in $75.4$ seconds, while SESA explores only one flow and finishes the execution in $3$ seconds thanks to removal of duplicate flows. For kernels bitonic and wordsearch, GKLEE$_p$ times out in $1$ hour even for $16$ threads, while SESA can scale to over $256$ threads. SESA produces $3$ flows for kernel blelloch with $64$ threads, and SESA finishes the checking in $46.9$ seconds while GKLEE$_p$ times out. Here we count only the execution time since the taint analysis time is negligible. For the buggy stream compaction kernel, SESA spends less than half of the time than GKLEE$_p$ to locate the bug. Note that these programs are highly divergent with large state spaces, and hence it may take both GKLEE$_p$ and SESA quite some time to analyze. Without flow optimizations it is very hard to check them for even small configurations such as $16$ threads.

SESA may suffer from false alarms or omissions caused by unresolvable reads described in §4.3.2. In our experiments, we spent effort identifying these unresolvable reads manually. Whenever feasible, we also compared our manual results against GKLEE [31], the predecessor of GKLEE$_p$ and SESA. For example, stream compaction and n stream compaction are `unresolvable` kernels.

Table 4.3 presents the results for the LonestartGPU benchmark, which consists of irregular kernels with multiple flows. Each `# Flow` cell is of the form 'Num-Flows' ('elapsed-time-in-secs.'). We set 3,600 seconds for T.O., and each `# Flow` refers to the maximum number of flows that occur during the execution. Column `RR/OM` denotes if Races were reported by SESA or if OMissions occur. If an R/W or a W/W race is listed under "Errors," then that means that the Race Report was manually confirmed as being a

**Table 4.3**: Comparison of SESA and GKLEE$_p$ for LonestarGPU benchmark.

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

genuine race. Whenever a kernel suffer from an `out-of-bound` error, we disabled this check in order to be able to collect the runtime. We use `# execution path` rather than `# Flow` for kernels marked with $*$ symbols. In bfs_atomic and BoundingBox kernels, R/W race stems from "don't-care non-det." (ensures one body inserted in each position). $?$ denotes those errors are under investigation. The columns with `(Conc.)` use pure concrete inputs (the thread ids are still symbolic), while the columns with `(Sym.)` apply the symbolic inputs identified by the taint analyzer, with those flowing into loop bounds being excluded (as mentioned in §4.2.3). For example, consider kernel bfs_ls. When symbolic inputs are used, GKLEE$_p$ times out in $1$ hour, and SESA finishes the checking in $0.9$ seconds with only two flows.

Note that when the symbolic inputs are not constrained with proper assumptions, many memory out-of-bound (OOB) errors may be produced, while these errors are not relative to races. GKLEE$_p$ suffers seriously from this problem, while SESA is more resilient with only a portion of inputs being symbolic. To make the comparison fair, we disable the OOB checking and OOB related state spawning in both GKLEE$_p$ and SESA. This often reduces the total execution time (*e.g.,* less than that with concrete inputs). We also show intuitively in Figure 4.8 the speedup, *e.g.,* SESA is more than $3,000$x faster than GKLEE$_p$ for kernel bfs-ls with symbolic inputs. For better readability, this figure does not show the overflowing values in an exact way, *e.g.,* the red bar of bfs-ls, which has over $3,000$x speed-up. Similarly, Figure 4.9 shows some speedups for kernels in Table 4.2. Here SESA can outperform GKLEE$_p$ by 1–3 orders of magnitude.

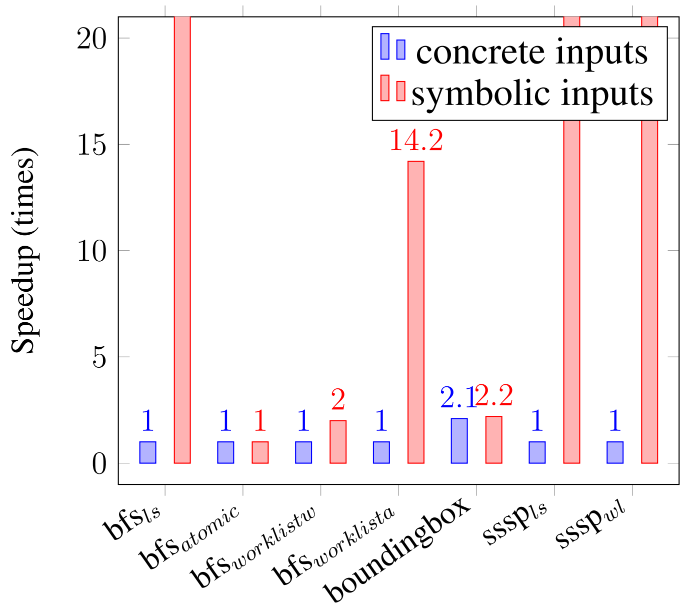

**Figure 4.8**: Speedup of SESA over GKLEE$_p$ on LoneStarGPU.

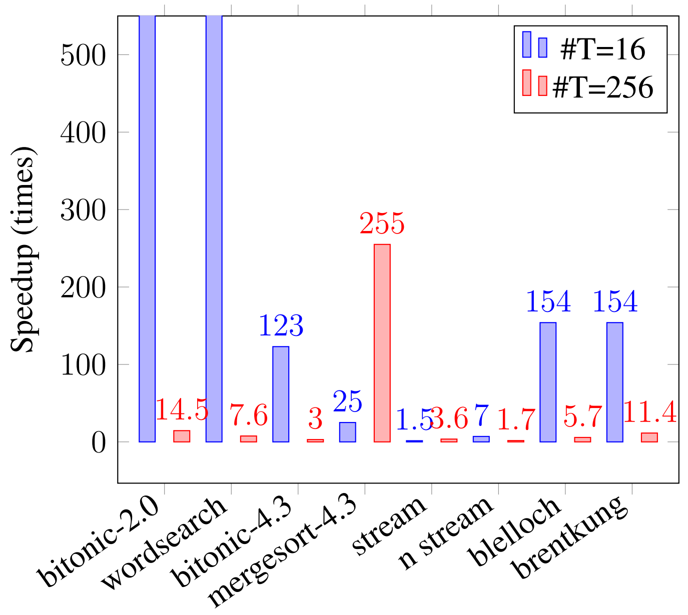

**Figure 4.9**: Speedup of SESA over GKLEE$_p$ on kernels in Table 4.2.

### 4.5.3 Parboil

We have run SESA on the entire Parboil benchmark [3]. Table 4.4 lists the results on 10 kernels, many of which contain issues revealed by SESA. We set 2 hours for T.O. in this table. For mri-gridding and spmv, the first member of the $\langle\rangle$ pair denotes the number of inputs inferred by SESA to be made symbolic, while the second represents the actual number of symbolized inputs needed (revealed by manual analysis). For example, for histo_prescan_kernel, SESA infers that one out of the three inputs should be made symbolic. SESA can scale to $32,768$ threads. It explores two flows and detects a real RW race detailed below.

Figure 4.10 explains the read-write race uncovered in histo_prescan (witness was automatically generated). The write access in `SUM(stride)` is performed by thread $\langle 17,0,0\rangle$ while the read access in `SUM(16)` is by thread $\langle 1,0,0\rangle$; these conflict, leading to the race.

Figure 4.11 illustrates the out-of-bound access caught in the histo_final kernel. The OOB access occurs in the $47$th iteration of the loop, and the initial value of $i$ is $tid.x + bid.x * 512$ for symbolic $tid$ and $bid$. In addition, the size of memory region the pointer global_histo refers to is $8,159,232$. Then the OOB is modeled as a constraint $(tid.x + bid.x \times 512 + 47 \times 42 \times 512) * 8 < 8159230$, and SESA solves this formula and identifies a thread with the configuration: $bid = \langle 24,0,0\rangle \wedge tid = \langle 0,0,0\rangle$ that could incur the OOB access. Note that it is nontrivial to use manual or random testing to come up with such witnesses.

Figure 4.12 illustrates the interblock read-write race caught in the binning_kernel. This

**Table 4.4**: Parboil results.

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

```
__global__ void histo_prescan_kernel(...) {
#define SUM(stride__)
if(threadIdx.x < stride__){
    Avg[threadIdx.x] += Avg[threadIdx.x+stride__];
    StdDev[threadIdx.x] += StdDev[threadIdx.x+stride__];
}

#if (PRESCAN_THREADS >= 32)
    for (int stride = PRESCAN_THREADS/2;
         stride >= 32; stride = stride >> 1) {
        __syncthreads();
        SUM(stride);
    }
#endif
#if (PRESCAN_THREADS >= 16)
    SUM(16);
#endif
...
}
```

**Figure 4.10**: The read-write race in histo prescan kernel.

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

**Figure 4.11**: The OOB in histo final kernel.

```
__global__ void binning_kernel (...) {
  unsigned int sampleIdx = blockIdx.x*blockDim.x
                           +threadIdx.x;
  ...
  if (sampleIdx < n){
    pt = sample_g[sampleIdx];

    binIdx = (unsigned int)(pt.kZ)*size_xy_c +
             (unsigned int)(pt.kY)*gridSize_c[0] +
             (unsigned int)(pt.kX);
    if (binCount_g[binIdx]<binsize){
      count = atomicAdd(binCount_g+binIdx, 1);
  }
}
```

**Figure 4.12**: The read-write race in binning kernel.

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

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

# CHAPTER 5

# CONCLUSION AND FUTURE WORK

Symbolic execution based verification is highly attractive in that one is able to bring the benefits of formal analysis to real code (not models of the code) written in practical languages (e.g., C++) and compiled using actual compilers (*e.g.,* LLVM).

First, we presented GKLEE, the first symbolic virtual machine based correctness checker and test generator for GPU programs written in CUDA/C++. It checks several error categories, including one previously unidentified race type. We discussed logical errors and performance bottlenecks detected by GKLEE in real-world kernels. For many realistic kernels, finding these issues 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.

To overcome the scalability problem GKLEE suffers from, parameterized reasoning is proposed. This strategy fully utilizes the thread symmetry property within CUDA programs. Despite the theoretical inexactness of this approach, our results show that we have caught all the races found in our earlier efforts, found some new ones, and have been able to scale GKLEE to huge amount of threads, finishing testing within acceptable runtimes. This makes GKLEE$_p$ a practical race checking facility—the first of its kind—that also allows programmers to choose symbolic inputs and obtain code coverage over all the branches that depend on these inputs.

The current symbolic testing frameworks require users to manually select inputs that should be symbolized, which not only costs more manual efforts, but incurs the imprecision, leading to the unnecessary symbolic execution or the missing of the chance to uncover potential defects. Hence, GKLEE$_p$ evolves to SESA, and SESA employs static analysis to automatically identify inputs that can be safely set to concrete values, often identifying most of the inputs that can be so set without losing coverage. A key novelty of SESA is that its static analysis also informs parametric flow combining, a key technique that avoids the creation of unnecessary flows. It also provides a race-modulo analysis technique that is reminiscent of techniques developed for CPUs (*e.g.,* [9, 11, 22, 25]), but adapts this thinking to GPU (SIMD) codes. In addition, SESA provides many of the advanced CUDA features including CUDA atomics, many CUDA intrinsics, etc. We thoroughly evaluate SESA on large benchmark suits such as Parboil and Lonestar. During these experiments, the tool automatically found several genuine bugs, including out of bound array accesses and data races. It also found a few races later deemed to be benign and in the process forced useful code walk-through and code understanding. All error reports are accompanied by concrete witnesses (input values and thread IDs involved in the issue). These results, together with the enriched CUDA subset that SESA handles, positions itself as (to the best of our knowledge) the foremost of formally based GPU correctness analysis tools geared primarily toward race-checking.

## 5.1 Future Work

In the future, our framework can be extended in terms of new ideas and new engineering:

1. Symbolic executor can be extended to automatically detect the "unresolvable" memory accesses. For example, we can employ SMT solvers to reason about if the shared/global data will be upated by multiple threads. If "unresolvable" values are used in sensitive sinks, false alarms or omissions might happen. To overcome the disadvantages incurred by "un-resolvable" variables, we should use a good havocing (meaning different symbolic variables are introduced for different shared writes), then no omissions.
2. GKLEE$_p$ and SESA can be parallelized through multithreaded or MPI programming model, leading to better performance.
3. Since CUDA programming model evolves fast, more CUDA runtimes and features can be supported to improve tool's power to handle host code better.
4. Since GKLEE, GKLEE$_p$, and SESA are all based on Clang/LLVM, which provides better supports for OpenCL, and OpenCL shares the similar programming model as CUDA, they can be used as a good symbolic testing platform for OpenCL as well.

## REFERENCES

[1] New Top500 List, `http://blogs.nvidia.com/blog/2012/07/02/new-top500-list-4x-more-gpu-supercomputers/`.

[2] Lonestargpu benchmark. `http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu`.

[3] Parboil benchmark. `http://impact.crhc.illinois.edu/parboil.aspx`.

[4] Sarita Vikram Adve. *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, Madison, WI, USA, 1993. UMI Order No. GAX94-07354.

[5] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H.B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA'91)* (Toronto, Ontario, Canada, Apr. 1991), ACM, 234–243.

[6] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)* (San Diego, California, USA, Jan. 1998), ACM, 342–354.

[7] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)* (Austin, Texas, USA, Jan 2011), ACM, 487–498.

[8] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: A verifier for GPU kernels. In *Proceedings of the 27th ACM International Conference on Object Oriented Programming Systems Languages and Applications(OOPSLA'12)* (Tucson, Arizona, USA, Oct. 2012), ACM, 113–132.

[9] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)* (Budapest, Hungary, Mar 2008), Springer-Verlag, 351–366.

[10] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated dynamic analysis of CUDA programs. In *Proceedings of the 3rd Workshop on Software Tools for MultiCore Systems (STMCS'08)* (Apr. 2008).

[11] Suhabe Bugrara and Dawson R. Engler. Redundant state detection for dynamic symbolic execution. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC'13)* (San Jose, California, USA, 2013), USENIX Association, 199–211.

[12] C++ AMP, `http://msdn.microsoft.com/en-us/library/vstudio/hh265137.aspx`.

[13] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)* (San Diego, California, Dec 2008), USENIX Association, 209–224.

[14] Nathan Chong, Alastair F. Donaldson, Paul H.J. Kelly, Jeroen Ketema, and Shaz Qadeer. Barrier invariants: A shared state abstraction for the analysis of data-dependent gpu kernels. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2013)* (Indianapolis, Indiana, USA, Oct 2013), ACM, 605–622.

[15] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic crosschecking of floating-point and simd code. In *Proceedings of the 5th ACM SIGOPS on European Conference on Computer Systems (EuroSys'11)* (Salzburg, Austria, Apr 2011), ACM, 315–328.

[16] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic testing of opencl code. In *Proceedings of the 7th Haifa Verification Conference (HVC'11)* (Berlin, Heidelberg, Jan 2011), Springer-Verlag, 203–218.

[17] CUDA zone. `http://www.nvidia.com/object/cuda_home_new.html`.

[18] CUDA Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[19] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages (POPL'05)* (Long Beach, California, USA, Jan. 2005), ACM, 110–121.

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)* (Chicago, Illinois, USA, June 2005), ACM, 213–223.

[21] gpgpu.org. `http://gpgpu.org/tag/image-processing`.

[22] Trevor Hansen, Peter Schachte, and Harald Søndergaard. State joining and splitting for the symbolic execution of binaries. In *Proceedings of the 9th Workshop on Runtime Verification (RV)* (Berlin, Heidelberg, Jun 2009), Springer-Verlag, 76–92.

[23] Amir Kamil and Katherine A. Yelick. Concurrency Analysis for Parallel Programs with Textually Aligned Barriers. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC'05)* (Hawthorne, New York, USA, Oct. 2005), Springer-Verlag, 185–199.

[24] Klee open projects. `http://klee.llvm.org/OpenProjects.html`.

[25] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'12)* (Beijing, China, Jun 2012), ACM, 193–204.

[26] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[27] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying GPU kernels by test amplification. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)* (Beijing, China, June 2012), ACM, 383–394.

[28] Guodong Li and Ganesh Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)* (Santa Fe, New Mexico, USA, Nov. 2010), ACM, 187–196.

[29] Guodong Li and Ganesh Gopalakrishnan. Parameterized Verification of GPU Kernel Programs. In *Multicore and GPU Programming Models, Languages and Compilers Workshop* (Shanghai, China, May 2012), IEEE, 2450–2459.

[30] Guodong Li, Peng Li, Geof Sawaga, and Ganesh Gopalakrishnan. GKLEE: Concolic verification and test generation for GPUs. Tech. rep., University of Utah, 2012. Available at `www.cs.utah.edu/fv/GKLEE`.

[31] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. GKLEE: Concolic verification and test generation for gpus. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)* (New Orleans, Louisana, USA, Feb. 2012), ACM, 215–224.

[32] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in cuda programs. In *Proceedings of the 24th ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis Conference (SC'12)* (Salt Lake City, Utah, USA, Nov. 2012), IEEE, 29:1–29:10.

[33] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Practical symbolic checking of gpu programs. In *Proceedings of the 26th ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis Conference (SC'14)* (New Orleans, Louisana, USA, Nov. 2014), IEEE, 179–190.

[34] Allinea Software Ltd. Allinea DDT. `http://www.allinea.com/products/ddt`.

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

[36] NVIDIA. CUDA-GDB, Jan. 2009. An extension to the GDB debugger for debugging CUDA kernels in the hardware.

[37] NVVM. `http://docs.nvidia.com/cuda/nvvm-ir-spec/index.html`.

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

[39] PTX. `http://www.nvidia.com/content/cuda-ptx_isa_1.4.pdf`.

[40] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'97)* (Saint Malo, France, Oct. 1997), ACM, 27–37.

[41] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)* (Lisbon, Portugal, Sep 2005), ACM, 263–272.

[42] Jaroslav Sevcik. Safe Optimisations for Shared-Memory Concurrent Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)* (San Jose, California, USA, June 2011), ACM, 306–316.

[43] Dennis Shasa and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS) 10*, 2 (1988), 282–312.

[44] SMT-COMP. `http://www.smtcomp.org/2011`.

[45] Tyler Sorensen. Towards shared memory consistency models for GPUs. Section 5.3 of `http://www.cs.utah.edu/fv/theses/tyler_bs.pdf`.

[46] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. Efficient compilation of fine-grained spmd-threaded programs for multicore CPUs. In *Proceedings of the 8th IEEE/ ACM International Symposium on Code Generation and Optimization (CGO'10)* (Toronto, Ontario, Canada, Apr. 2010), ACM, 111–119.

[47] Weibin Sun and Robert Ricci. Fast and flexible: Parallel packet processing with GPUs and Click. In *Proceedings of the 9th ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS'13)* (San Jose, California, USA, Oct. 2013), IEEE, 25–36.

[48] Weibin Sun, Robert Ricci, and Matthew J. Curry. GPUstore: Harnessing GPU computing for storage systems in the OS kernel. In *Proceedings of the 5th ACM International Systems and Storage Conference (SYSTOR'12)* (Haifa, Israe, June 2012), ACM, 9:1–9:12.

[49] Tegra4. `http://www.nvidia.com/object/tegra-4-processor.html`.

[50] Titan, `http://www.olcf.ornl.gov/titan/`.

[51] Top 500, `http://www.top500.org/`.

[52] Discussions on exploiting warp semantics and consequences, as of 2007. Expert Nvidia of `https://devtalk.nvidia.com/default/topic/377816/`.

[53] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In *Proceedings of the 15th International SPIN Workshop on Model Checking of Software (SPIN'08)* (Saint Malo, France, Aug 2008), Springer-Verlag, 288–305.

[54] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. GRace: A low-overhead mechanism for detecting data races in GPU programs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'11)* (San Antonio, TX, USA, Feb. 2011), ACM, 135–146.