Verifying GPU Kernels by Test Amplification *

Alan Leung Manish Gupta Yuvraj Agarwal Rajesh Gupta Ranjit Jhala Sorin Lerner

University of California, San Diego

{aleung,manishg,yuvraj,gupta,jhala,lerner}@cs.ucsd.edu

Abstract

We present a novel technique for verifying properties of data parallel GPU programs via *test amplification*. The key insight behind our work is that we can use the technique of static information flow to amplify the result of a single test execution over the set of all inputs and interleavings that affect the property being verified. We empirically demonstrate the effectiveness of test amplification for verifying race-freedom and determinism over a large number of standard GPU kernels, by showing that the result of verifying a single dynamic execution can be amplified over the massive space of possible data inputs and thread interleavings.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification – Validation; F.3.2 [*Semantics of Programming Languages*]: Semantics of Programming Languages – Program analysis

General Terms Languages, Reliability, Verification

Keywords Test Amplification, Determinism, GPU

1. Introduction

Despite its relative infancy, CUDA and related programming languages have already become one of the most widely used models for parallel programming. The appeal of this model is that it offers a way to exploit fine-grained data-parallelism within the comforts of C/C++ style imperative programming. The dramatic performance gains offered by the approach have led to its widespread use in a variety of domains including scientific computing, finance, and computational biology [9].

CUDA programs are very hard to get right, for all the usual reasons, and a few new ones. First, in the course of execution, a program can spawn millions of threads, which are clustered in multilevel hierarchies that mirror a multi-level shared memory hierarchy. Second, for performance reasons, well-weathered synchronization mechanisms like locks or transactions are eschewed in favor of barriers, and it is up to the programmer to make sure that concurrently executing threads do not conflict by ensuring that their algorithms make threads touch disjoint parts of memory. Third, a key source of efficiency is the extremely fine-grained memory sharing across threads. Memory bandwidth is maximized when threads with successive identifiers access, in lock-step, addresses that are physically adjacent in memory. This can lead to complex striding patterns of memory access that are hard to reason about.

An unfortunate consequence of these factors is that CUDA parallelism is difficult to analyze with existing static or dynamic approaches. Static techniques are thwarted by the complexity of the sharing patterns. A useful static analysis would have to find a succinct symbolic representation for the sets of addresses accessed by each thread. It is common for CUDA code to use modular arithmetic or bit-shifting to access memory indices according to complex linear or even non-linear patterns which makes such analyses difficult. Dynamic techniques are challenged by the combinatorial explosion of thread interleavings and space of possible data inputs: any reasonable number of tests would represent a small subset of the possible behaviors of the system. Finally, mechanisms for enforcing isolation at run-time are likely to impose unacceptable overheads, and would leave the developer the unenviable task of sifting through a huge execution trace to figure out what went wrong.

To attack these challenges, we employ *test amplification*, a general notion wherein a single dynamic run can be used to learn much more information about a program's behavior than is directly exhibited by that particular execution. Though this notion has been explored in many settings, we contribute a new formulation of test amplification specialized for verifying properties of CUDA kernels, which we call *flow-based test amplification*. Our technique skirts the limitations of existing static and dynamic approaches by combining their complementary strengths, as follows.

First, we run a dynamic analysis where we log the behavior of the kernel with some fixed test input and under a particular thread interleaving. Second, we use a static information flow analysis to compute the *property-integrity inputs*, namely, the input variables that actually flow-to, or affect the integrity of, the variables appearing in the property to be verified. Finally, we amplify the result of the test to hold over *all* the inputs that have the same values for the property-integrity inputs.

Of course, this approach would yield no mileage if all the inputs were property-integrity inputs. Our second contribution is to empirically demonstrate over a large number of CUDA benchmarks, that for key properties like race-freedom and determinism, the set of property-integrity inputs comprises just a small core of *configuration* inputs. These are typically parameters that describe the dimensions of the thread- and memory- hierarchies and dataset, and which are highly tuned for a given architecture and algorithm. Thus, test amplification allows us to use a single execution over the configuration to verify properties over the massive space of all possible *data inputs* (non-configuration inputs) and thread interleavings that constitute the behaviors of the kernel.

This simple insight ties together the complementary strengths of well-known static and dynamic approaches, yielding a clean way to analyze CUDA kernels. The static analysis is just a simple flows-to question that can be resolved via taint-propagation and

* This work was supported by NSF Expeditions in Computing grant CCF-1029783 and CAREER grants CCF-0644306 and CCF-0644361.

PLDI'12, June 11–16, 2012, Beijing, China.

an alias analysis. The dynamic analysis performs the difficult task of actually computing the sets of addresses that are accessed in complex striding patterns. Using the information flow, we can then generalize the dynamic analysis over all possible data inputs. In summary, we make the following contributions:

- We present *flow-based test amplification*, a hybrid technique specialized for analyzing GPU kernels which uses static information flow to generalize the result of a dynamic analysis over a large space of inputs (Section 3).
- We implement flow-based test amplification for CUDA using the LLVM framework, by implementing static taint-propagation and devising efficient ways to record and analyze traces of CUDA kernels (Section 4).
- We empirically demonstrate the effectiveness of flow-based test amplification for verifying race freedom and determinism of CUDA kernels, by conducting a systematic evaluation over the kernels available in the CUDA SDK [8] and demonstrating that the technique can be used for most kernels. Even when the technique does not apply, a kernel can often be slightly modified to fall under the scope of our analysis (Section 5). As a result, we believe test amplification can be a stepping stone to developing various safety and performance analyses for massively data-parallel programs.

2. Overview

We start with an overview of CUDA, and then motivate our technique with a textbook CUDA kernel.

2.1 CUDA Basics

To illustrate the CUDA programming model we use a simplified version of `scalarProd`, shown in Figure 1, a program from the CUDA SDK that performs a parallel dot product of two vectors. In a nutshell, the CUDA language is a variant of C++ extended with syntax to annotate parallel routines called *kernels*, and the shared data structures that the routines manipulate. The kernels execute on GPU hardware and must exploit the shared memory hierarchy in order to obtain maximum performance.

Thread Hierarchy A program written in CUDA decomposes parallel computation into a two-level geometric grid of `gridDim` *thread blocks*, where each block comprises `blockDim` threads, each performing a subset of the total workload. Due to hardware support, CUDA threads have much less overhead relative to CPU threads: a CUDA program might spawn more than one million threads in a single execution. The `gridDim` and `blockDim` configuration parameters are together called the *geometry* of the computation. (Although grids and thread blocks may be multi-dimensional, we omit finer grained $x-$ and $y-$ dimensions for clarity.)

Memory Hierarchy Threads communicate via a hierarchical memory model organized into three levels, in decreasing order of their thread visibility, size, and latency: *global memory* which is slowest and shared by all threads, *shared memory* which is faster but shared only by the threads within a particular block, and *registers* which are fastest but private to each thread. In our example, all threads belong to a single thread block, all with access to the shared array `accumRes` and the global arrays `d_A` and `d_B`.

Parallel Execution Threads are spawned and executed via *kernel calls*, which specify the kernel that is to be executed and the thread-geometry, *i.e.,* the delineation of threads into thread blocks. For example, a kernel call of the form

```
ScalarProd<<<1, 128>>>(d_C, d_A, d_B, 4096)
```

```
1: #define accumN 1024
2: void ScalarProd( float *d_C
                  , float *d_A
                  , float *d_B
                  , int sizeN)
   {
3:   __shared__ float accumRes[accumN];

/*******************************************/
/********* Phase 1: Partial Sums ***********/
/*******************************************/

4:   for( int i = threadIdx
        ; i < accumN
        ; i += blockDim)
     {
5:     float sum = 0;
6:     for(int j = i; j < sizeN; j += accumN)
       {
7:       sum += d_A[j] * d_B[j];
       }
8:     accumRes[i] = sum;
     }

/*******************************************/
/********* Phase 2: Tree Reduction *********/
/*******************************************/

9:   for( int stride = accumN / 2
        ; stride > 0
        ; stride >>= 1)
     {
10:    __syncthreads();
11:    for( int i = threadIdx
          ; i < stride
          ; i += blockDim)
       {
12:      accumRes[i] += accumRes[stride + i];
       }
     }
13:  if(threadIdx == 0) *d_C = accumRes[0];
   }
```

Figure 1. `scalarProd` kernel

begins execution of a single block containing 128 threads, where each thread executes the code in the `scalarProd` routine from Figure 1. Although all threads execute the same kernel, each thread is distinguished by a unique identifier in the form of its coordinate within the grid, that is by the tuple of the variables `blockIdx` and `threadIdx` (and the elided x- and y- sub-dimensions.) Thus, a thread can use the values of its `blockIdx` and `threadIdx`, and the geometry variables `gridDim` and `blockDim`, to distinguish its portion of the workload from those of other threads.

Synchronization Threads synchronize via barriers: a thread that calls `__syncthreads()` will wait until all other threads in its thread block have also reached the barrier. No corresponding synchronization mechanism exists for threads in different thread blocks. (While one can encode synchronization mechanisms using global memory, such mechanisms are brittle and inefficient and hence discouraged.)

2.2 Scalar Dot Product

Now that we are equipped with a basic understanding of the CUDA model, let us turn our attention to the `scalarProd` benchmark provided with the CUDA SDK. The original program calculates the dot products of 256 pairs of vectors, each of length $\texttt{sizeN} = 4096$. To simplify exposition, we have reduced the example to calculate the dot product of a single pair of vectors of `sizeN` elements.

The example exhibits three characteristics of CUDA kernels optimized for performance: 1) Memory is accessed in strides in order to maximize memory bandwidth, 2) Shared memory is used to cache frequently accessed data, and 3) Threads cooperate to perform reductions with minimal synchronization and communication. The example also demonstrates that even a seemingly simple computation, an elementary linear algebra primitive, can require hard-to-analyze optimizations when adapted for parallelization via CUDA. The implementation uses a parallel algorithm separated into two phases, each with a different parallelization strategy, to increase performance.

Phase 1: Partial Products In the first phase (lines 4-8), each thread computes $\texttt{accumN}/\texttt{blockDim}$ (*i.e.,* 8) partial dot-products, of sub-vectors of size $\texttt{sizeN}/\texttt{accumN}$ (*i.e.,* 4), and stores these partial products into the array `accumRes`. This phase is implemented by the nested loops on lines 4 and 6. The outer loop on line 4 iterates `i` over

$$\begin{aligned}&\texttt{threadIdx} + 0 \times \texttt{blockDim},\\ &\texttt{threadIdx} + 1 \times \texttt{blockDim},\\ &\texttt{threadIdx} + 2 \times \texttt{blockDim}, \ldots\end{aligned}$$

and at each offset, uses the inner-loop on line 6 to compute the dot-product of the sub-vectors at indices

$$\begin{aligned}&\texttt{i} + 0 \times \texttt{accumN},\\ &\texttt{i} + 1 \times \texttt{accumN},\\ &\texttt{i} + 2 \times \texttt{accumN}, \ldots\end{aligned}$$

the result of which is stored in the shared `accumRes[i]` (line 8).

Note that the threads *do not* operate on contiguous vector elements but instead access elements with a stride of `accumN`. The strided access pattern is deliberate: we achieve maximum memory bandwidth when neighboring threads access consecutive elements in global memory because the hardware optimizes the accesses to occur in parallel. This feature is known as *memory coalescing*. If instead each thread accessed contiguous vector elements, the memory accesses would occur serially, thereby severely reducing performance. Note also that the partial dot products are stored in shared memory, not global memory, in order to exploit temporal locality of access to intermediate results when performing the subsequent reduction.

Phase 2: Tree Reduction In the second phase (lines 9-13), all threads cooperate to add up the partial products to a final value. A naïve implementation would assign a single thread to perform the entire sum, reverting to a sequential algorithm. Instead, we can view the reduction as follows: each partial product is a leaf in a binary tree, and each parent is the sum of its children. We iterate up the levels of the tree by calculating the parents' values until we reach the root of the tree. Recall that the the array `accumRes` contains the partial products. Thus, in each iteration, we simply overwrite the "left child" at index `i` with the sum of its own value, and that of the "right child" at index $\texttt{i} + \texttt{stride}$, thereby obtaining the "parent" value.

Each iteration is parallelized by using thread `threadIdx` to compute the "parent" values at indices

$$\begin{aligned}&\texttt{threadIdx} + 0 \times \texttt{blockDim},\\ &\texttt{threadIdx} + 1 \times \texttt{blockDim},\\ &\texttt{threadIdx} + 2 \times \texttt{blockDim}, \ldots\end{aligned}$$

as done in the inner-loop on lines 11-12. As before, this strided access pattern enables efficiency via parallel memory access.

We iterate up the levels of the tree using the outer loop on line 9 which shrinks `stride` by half at each iteration as the number of leaf nodes halves across each iteration. Note that the cells accessed by different threads overlap *across levels*. The `__syncthreads()` at line 10 ensures that a level has been completed before the computation proceeds to the next level, and thereby preventing memory conflicts between threads executing on different levels. When the iterations have all completed, thread 0 copies the final value from $\texttt{accumRes}[0]$ into the destination `d_C` at line 15, and the kernel terminates.

2.3 Verification

Thus, the `scalarProd` kernel makes use of several sophisticated optimizations to squeeze performance out of the GPU hardware. Unfortunately, these optimizations are treacherous as it is easy enough for the usual sorts of concurrency-related errors to creep in. The GPU setting exacerbates matters, as dynamic monitoring and protection mechanisms would likely require expensive hardware support in order to be efficient. Even if such mechanisms could be implemented, errors would be difficult to debug due to the scale of the concurrency (millions of threads).

Property: Absence of Races Let us consider the concrete problem of verifying race freedom, *i.e.,* verifying that a given thread is not writing to a shared location at the same time that another is reading from the location. In the CUDA setting, the only synchronization primitive is the `__syncthreads()` barrier, and hence, the problem reduces to determining that the sets of memory locations read and written by different threads within matching barriers, are disjoint.

Difficulty of Static Verification Unfortunately, as the `scalarProd` example illustrates, existing static analyses are of little use in the face of the complex access patterns that are idiomatic in CUDA kernels. Aliasing-, Shape- or Escape- based approaches [29] can work for disjoint linked data structures, but are not precise enough to distinguish disjoint regions within shared arrays. More precise arithmetic abstractions for such regions, such as intervals, octagons, polyhedra and even interval congruences [27] would not suffice as the set of addresses accessed by each CUDA thread often follows complex patterns.

For example, in the first phase of `scalarProd` the thread `threadIdx` writes to the array `accumRes` at indices

$$\{\texttt{threadIdx} + m \times \texttt{blockDim} + n \times \texttt{accumN}\}$$

where m and n range over 0 and $\texttt{accumN}/\texttt{blockDim}$ and $\texttt{sizeN}/\texttt{accumN}$ respectively. The second phase is even more challenging, as in the k^{th} iteration of the outer-loop, each thread reads the array `accumRes` at indices

$$\{\texttt{threadIdx} + m \times \texttt{blockDim} + \texttt{accumN}/2^k\}$$

where m ranges from 0 to $\texttt{accumN}/(\texttt{blockDim} \times 2^k)$.

2.4 Our Approach

Our solution skirts the limitations of current static verification techniques with a combination of dynamic race detection and static information flow analysis. Our dynamic analysis tests for races in a *single execution* by instrumenting the kernel to log memory accesses and verifying disjointness of read and write addresses. We then *amplify* the results of the dynamic analysis to apply to *all possible executions* of a configuration if we can verify that the kernel always performs the same memory accesses, regardless of the values of its data inputs, a property we call *access invariance*. Given that this property holds, we then know that a single execution's accesses are in fact representative for all the possible executions.

X	$\doteq$	$\mathtt{threadIdx}, \mathtt{x}, \mathtt{y}, \ldots$	**Locals**
G	$\doteq$	$\mathtt{g}, \mathtt{h}, \ldots$	**Globals**
V	$\doteq$	$X \cup G$	**Variables**
e	$::=$		**Expressions**
	$\mid$	$\mathtt{x}$	local-read
	$\mid$	n	constant
	$\mid$	$e_1 \oplus e_2$	binop
c	$::=$		**Commands**
	$\mid$	$\mathtt{assume}(e)$	assume
	$\mid$	$\mathtt{x} = e$	var-assignment
	$\mid$	$\mathtt{x} = \mathtt{g}[\mathtt{x}]$	global-read
	$\mid$	$\mathtt{g}[\mathtt{x}] = e$	global-write
	$\mid$	$c; c$	sequence

Figure 2. Syntax

Putting it together, we can effectively guarantee that the kernel is race free and deterministic across *all data inputs*.

Dynamic Race Detection In addition to memory addresses we record the identity of the thread performing the access and the location of the barrier last encountered by that thread. To distinguish accesses to shared and global memory, we record the address and extent of each shared and global data structure.

We then check for races by verifying disjointness of write and read addresses between threads that can race to the same global or shared data structure. Given that the log demonstrates no such conflicting access, we provide the guarantee that the kernel is race free for the single instantiation of input values used for that execution. Because the analysis checks for the *possibility* of a race, not whether a destructive race has actually occurred, a kernel verified to be race free by this analysis is race free regardless of the particular interleaving exercised by the execution, once again for this single instantiation of input values. In addition, since the analysis verifies disjointness of accesses, we can also guarantee that the verified kernel executes deterministically on that input. However, note that this dynamic testing by itself can provide no guarantees about execution on any other instantiation of inputs.

Information Flow Although the guarantees provided by our dynamic race detector are desirable, they extend only to a single instantiation of inputs within an enormous universe of inputs, hence the need for a static analysis to amplify those guarantees further. In particular, we use a static information flow analysis that tracks flows from data inputs throughout the kernel. Intuitively, we apply a taint to data inputs values, track the flow of taint through program variables, then check that no tainted program variable is used as the address operand of a memory instruction. We additionally check that no memory access is control dependent on a tainted variable. If these two properties hold, we say that the kernel is *access invariant* and will exhibit the same memory accesses across all data inputs.

By itself, this property is interesting but not immediately useful. It is the combination of this property and the result of our dynamic analysis that produces a much more powerful guarantee: the verified kernel will exhibit no races in any execution, and so the kernel is deterministic.

3. Test Amplification via Information Flow

In this section we formalize a general framework for combining tests with information flow, and show how we instantiate it to verify CUDA kernels. We start by making precise the syntax and semantics of kernels, which enables us to define the ingredients of the main Theorem 1 which states how the results of a particular test can be amplified across different inputs via information flow. Next, we describe how the general framework is instantiated for the setting of CUDA kernels.

3.1 Syntax

Figure 2 summarizes the syntax of kernels. Informally, a kernel is a collection of concurrently executing threads which interact via shared memory.

Variables Our kernels include two kinds of variables: (thread)-local, denoted by $\mathtt{x}, \mathtt{y}$, *etc.* and global $\mathtt{g}, \mathtt{h}$, *etc.*. We write $\mathtt{v}, \mathtt{w}$ to denote either local or global variables. We assume for simplicity that all global variables are arrays. Note that a scalar global is simply an array of size 1. For clarity, we abuse notation to omit the offset 0 when reading or writing such variables.

Expressions and Commands The set of kernel expressions includes reads of local variables, primitive constants (*e.g.,* $0, 1, 2, \ldots$) and binary operations over sub-expressions. Our language of expressions is side-effect free. The set of commands includes sequences of $\mathtt{assume}(\cdot)$ (used to model branches), local assignment, and reads from and writes to global variables.

Reads and Writes A variable $\mathtt{v}$ is *read* in an expression if it appears in the expression. A variable is read in a command if the command contains an expression in which the variable is read. A variable $\mathtt{v}$ is written in a command if the command contains an assigment of the form $\mathtt{v} = e$ or $\mathtt{v}[\mathtt{x}] = e$.

Threads A *thread* t is a tuple (V_0, L, l_0, C), where V_0 is a set of *input variables*, L is a finite set of program locations, $l_0 \in L$ is a special *entry location*, and C is a *control flow map* from $L \times L$ to the set of commands. A variable is read (resp. written) on an edge if it is read (resp. written) on the command labelling the edge. A variable $\mathtt{v}$ is *immutable* in a thread if it is not written on any edge of the thread's control-flow map. We assume that V_0 includes any local or global variables that may be read before they are written. Finally, we assume that V_0 includes distinguished immutable local input variables `threadIdx` and `blockDim` that hold the (unique) identifier of the thread and the total number of threads, respectively.

Kernels A *kernel* is a pair of a set of *thread-identifiers* $Tid \subseteq \mathbb{N}$ and a thread. Intuitively, a kernel (Tid, V_0, L, l_0, C) has $|Tid|$ many distinct threads executing the same commands (given by C). However, the commands can inspect the value of the input `threadIdx`, and hence, different threads can behave differently.

Example Recall the `scalarProd` kernel from Figure 1. The body of the kernel can be mapped to a single thread's control-flow map, with locations corresponding to the program labels (`3:`, `4:`, `5:`, *etc.*.) in the standard way. The input variables of the thread are `d_A`, `d_B`, `d_C`, `sizeN`, the distinguished `threadIdx`, and `blockDim`.

3.2 Semantics

Next, we formalize the semantics via states and transitions.

Notation Let f map A to B. We write $f[a \mapsto b]$ for the map

$$\lambda x.\ \text{if } x = a \text{ then } b \text{ else } f\ x$$

Let A' be a subset of A. The *restriction* of f to A', is the map from A' to B that coincides with f on A'.

Kernel Transition $\boxed{P \vdash \sigma \hookrightarrow \sigma'}$

$$\frac{tid, C \vdash \sigma \hookrightarrow \sigma'}{(Tid, V_0, L, l_0, C) \vdash \sigma \hookrightarrow \sigma'} \text{ [T-PGM]}$$

Thread Transition $\boxed{tid, C \vdash \sigma \hookrightarrow \sigma'}$

$$\frac{l = \sigma(tid) \qquad c = C(l, l') \qquad tid, c \vdash \sigma \hookrightarrow \sigma'}{tid, C \vdash \sigma \hookrightarrow \sigma'[tid \mapsto l']} \text{ [T-THREAD]}$$

Command Transition $\boxed{tid, c \vdash \sigma \hookrightarrow \sigma'}$

$$\frac{tid, c_1 \vdash \sigma \hookrightarrow \sigma' \qquad tid, c_2 \vdash \sigma' \hookrightarrow \sigma''}{tid, c_1; c_2 \vdash \sigma \hookrightarrow \sigma''} \text{ [T-SEQ]}$$

$$\frac{\sigma(tid)(e) = true}{tid, \texttt{assume}(e) \vdash \sigma \hookrightarrow \sigma} \text{ [T-ASSUME]}$$

$$\frac{\sigma(e)(tid) = n}{tid, \texttt{x = } e \vdash \sigma \hookrightarrow \sigma[\texttt{x} \mapsto \sigma(\texttt{x})[tid \mapsto n]]} \text{ [T-ASGN]}$$

$$\frac{\sigma(y)(tid) = n' \qquad \sigma(\texttt{g})(n') = n}{tid, x \texttt{ = g}[y] \vdash \sigma \hookrightarrow \sigma[\texttt{x} \mapsto \sigma(\texttt{x})[tid \mapsto n]]} \text{ [T-READ]}$$

$$\frac{\sigma(y)(tid) = n' \qquad \sigma(x)(tid) = n}{tid, \texttt{g}[y] \texttt{ = } x \vdash \sigma \hookrightarrow \sigma[\texttt{g} \mapsto \sigma(\texttt{g})[n' \mapsto n]]} \text{ [T-WRITE]}$$

Figure 3. Semantics

States A *state* σ is a map from $Tid \cup X \cup G$ to $L \cup (Tid \to \mathbb{N}) \cup (\mathbb{N} \to \mathbb{N})$, such that $\sigma(tid)$ is the program location of thread tid, $\sigma(\texttt{x})(tid)$ is the value of the local $\texttt{x}$ in the thread tid, and $\sigma(\texttt{g})(n)$ is the value of the global array $\texttt{g}$ at the index n. We write Σ for the set of all states.

Initial States A state σ is an initial state for kernel $P = (Tid, V_0, L, l_0, C)$ written $P \vdash \sigma$ if $\sigma(\texttt{blockDim}) = |Tid|$, and for each $tid \in Tid$, we have $\sigma(tid) = l_0$ and $\sigma(\texttt{threadIdx})(tid) = tid$.

Transitions The *transition* relation of a kernel is a subset of $\Sigma \times \Sigma$. We write $P \vdash \sigma \hookrightarrow \sigma'$ if the pair σ, σ' is in the transition relation of the kernel, defined formally in Figure 3. Intuitively, the transition relation is the union of the transition relations of the individual threads. Each thread transition atomically moves the thread from its current program location to a successor location, updating the locals of the thread and the globals whilst leaving the program locations and locals of all other threads untouched.

An assume transition [T-ASSUME] succeeds only if the condition holds, a local assignment [T-ASGN], [T-READ] updates the value of the local for the executing thread, and a write [T-WRITE] updates the global array at appropriate address. A command is a *sequence* of assignments and assumes, and [T-SEQ] ensures that the sequence executes atomically, thereby allowing us to construct higher level synchronization mechanisms as described later.

Branch Determinism We say a control flow map is *branch deterministic* if for any thread tid and state σ, there is at most a single σ' such that $tid, C \vdash \sigma \hookrightarrow \sigma'$. There is a simple sufficient condition for branch determinism, namely 1) the primitive operations $\oplus$ are deterministic, and 2) the program locations have at most two successor edges, 3) the nodes with multiple successor edges have edges labeled by $\texttt{assume}(e)$ and $\texttt{assume}(\neg e)$. The control flow maps from compiled from standard structured languages are branch deterministic, so we will assume this property in the sequel.

Traces The *traces* of a kernel $\mathsf{Traces}(P)$ are finite sequences of of states $\tau = \sigma_0, \ldots, \sigma_n$ such that $P \vdash \sigma_0$ and $P \vdash \sigma_k \hookrightarrow \sigma_{k+1}$, for each $0 \leq k < n$. We write $\tau(k)$ for the k^{th} state in a trace.

3.3 Test Amplification

Now that we have formalized the semantics of kernels, we can describe what it means for a variable to flow-to another, and hence the notion of flow-based test amplification.

Projection and Equivalence Let Y be a set of variables. The *projection* of a state σ to Y, written $\sigma[Y]$ is the restriction of σ to the domain Y. We lift projection to sequences of states, *i.e.,* traces, in the natural way. A state σ is *equivalent over* Y to σ', written $\sigma \equiv_Y \sigma'$ if $\sigma[Y] = \sigma'[Y]$.

Information Flow A variable $\texttt{v}$ *flows-to* $\texttt{w}$ written $\texttt{v} \in \mathsf{FlowsTo}(P, \texttt{w})$ if there exist traces, τ and τ' in $\mathsf{Traces}(P)$ such that $\tau(0) \equiv_{V \setminus \texttt{v}} \tau'(0)$, and $\tau(k) \not\equiv_{\texttt{w}} \tau'(k)$ for some k. Intuitively, $\texttt{v}$ flows-to $\texttt{w}$ if there are two traces that agree on all variables *except* $\texttt{v}$ at the beginning, but which differ on the value of $\texttt{w}$ at some time k [20]. For a set of variables W we define

$$\mathsf{FlowsTo}(P, W) \doteq \cup_{\texttt{w} \in W} \mathsf{FlowsTo}(P, \texttt{w})$$

Thread Interleavings We say that a variable $\texttt{w}$ is non-deterministically affected by thread interleavings if there exist traces $\tau, \tau' \in \mathsf{Traces}(P)$ such that $\tau(0) = \tau'(0)$ but for some k, $\tau(k)(\texttt{w}) \neq \tau'(k)(\texttt{w})$. We make the following observations:

1. If $\texttt{w}$ is non-deterministically affected by interleavings then trivially, for all $\texttt{v}$ we have $\texttt{v} \in \mathsf{FlowsTo}(P, \texttt{w})$.
2. If $\texttt{w}$ is *not* non-deterministically affected by thread interleavings. Then $\texttt{v} \in \mathsf{FlowsTo}(P, \texttt{w})$ only if $\texttt{v}$ is an input variable.

Properties A *property* Φ *over* V_Φ is a predicate over the (program) variables V_Φ. A state σ satisfies a property, written $\sigma \models \Phi$ if the predicate evaluates to true in σ. A trace τ satisfies a property, written $\tau \models \Phi$ if for each k, we have $\tau(k) \models \Phi$. The set $\mathsf{FlowsTo}(P, V_\Phi)$ is the set of *property-integrity inputs*, that is, the set of inputs that flow-to a variable in V_Φ.

THEOREM 1. **[Test Amplification]** *Let Φ be a property over V_Φ, and Y be a subset of the input variables. If*

- $\mathsf{FlowsTo}(P, V_\Phi) \subseteq Y$
- $\tau \in \mathsf{Traces}(P)$ *is such that* $\tau \models \Phi$

then $\forall \tau' \in \mathsf{Traces}(P).\ \tau(0) \equiv_Y \tau'(0)$ *implies* $\tau' \models \Phi$.

Informally, the theorem states that if Y contains the set of property-integrity inputs, then we can amplify the success of *single* satisfying execution to conclude that all *all* executions that start with the same values for Y, regardless of the values of other inputs and thread scheduling, will also satisfy the property.

3.4 CUDA Verification

The conclusion of the test amplification theorem is trivial, and of little practical value if the set Y includes all the program variables. To analyze CUDA kernels, we must reduce Y to be as small as possible. We demonstrate that for verifying determinism of CUDA kernels, Y can be distilled down to a small core of *configuration* inputs that are highly tuned to a limited set of values for a given algorithm and architecture. Consequently, test amplification allows us to use a dynamic single execution over the configuration to verify properties over a massive space of possible data inputs and thread interleavings that constitute the behaviors of the kernel.

Thus, next, we describe how our framework can be instantiated to verify CUDA kernels. To do so, we need to: 1) encode CUDA semantics, in particular, barrier synchronization, 2) encode the properties that we wish to check, 3) describe the property variables V_Φ and configuration variables Y, 4) compute the set $\mathsf{FlowsTo}(P, V_\Phi)$.

Barriers CUDA includes a special barrier operation. Intuitively, when a thread reaches a barrier, it waits until all the other threads have reached the same barrier. We encode barriers by introducing two special variables `flag`, a local variable that has the value 1 iff the thread tid has reached a barrier, and `count`, a global variable (initialized to 0) which holds the number of threads that have reached the barrier. Now, a barrier operation between l and l' is a sequence of three CFG edges:

$$
\begin{array}{lcl}
C(l, l_{wait}) & \doteq & \texttt{assume}(\texttt{flag} = 0);\\
 & & \texttt{flag = } 1;\\
 & & \texttt{count = count} + 1\\
C(l_{wait}, l_{go}) & \doteq & \texttt{assume}(\texttt{count} = \texttt{blockDim})\\
C(l_{go}, l') & \doteq & \texttt{assume}(\texttt{threadIdx} = \texttt{count});\\
 & & \texttt{flag = } 0;\\
 & & \texttt{count = count} - 1
\end{array}
$$

where l_{wait}, l_{go} are two distinct program locations introduced for each barrier operation.

Instrumentation: Recording Global Accesses To verify that the kernel is deterministic, we need to check in the trace that the set of global addresses written by a thread is disjoint from the addresses accessed by other threads. We need only perform this check on accesses within the same *barrier interval* because two accesses separated by a barrier cannot race [24]. Thus, we track both the set of addresses read or written by each thread, as well as timestamps uniquely identifying each barrier interval. We instrument the kernel to track this information as follows.

First, we introduce a special local variable `timestamp` (initialized to 0) which holds a logical timestamp for the current barrier interval. We instrument each barrier-wait $\texttt{assume}(\texttt{count} = \texttt{blockDim})$ to become

$$
\begin{array}{l}
\texttt{assume}(\texttt{count} = \texttt{blockDim});\\
\texttt{timestamp} = \texttt{timestamp} + 1
\end{array}
$$

We introduce a global array `log`, that maps Tid, to subsets of

$$\mathbb{N} \times \{\mathsf{Rd}, \mathsf{Wr}\} \times G \times \mathbb{N}$$

That is, each tuple consists of a barrier timestamp, access type, global array name, and array index, respectively. Next, we instrument each read and write command to record the access inside `log`. Global-reads `y = g[x]` become

$$
\begin{array}{l}
\texttt{y = g[x]};\\
\texttt{log[threadIdx] = log[threadIdx]} \cup (\texttt{timestamp}, \mathsf{Rd}, \text{“g”}, \texttt{x})
\end{array}
$$

and global-writes `g[x] = y` become

$$
\begin{array}{l}
\texttt{log[threadIdx] = log[threadIdx]} \cup (\texttt{timestamp}, \mathsf{Wr}, \text{“g”}, \texttt{x});\\
\texttt{g[x] = y}
\end{array}
$$

Property: Determinism Finally, to verify determinism we define the following property Φ_{Det}.

$$
\begin{array}{l}
\forall t, i, i', \mathsf{g}, n : (t, A, \text{“}g\text{”}, n) \in \texttt{log}[i] \wedge (t, A', \text{“}g\text{”}, n) \in \texttt{log}[i']\\
\qquad \Rightarrow i = i' \vee A = A' = \mathsf{Rd}
\end{array}
$$

Intuitively, the property checks that within each barrier interval, the set of *writes* in the `log` of thread tid is disjoint from the set of *accesses* contained in the `log` of another thread tid'. As the instrumentation ensures that `log` contains all accesses, if the above sets are disjoint, then the sets of accesses of every two threads are disjoint, and the kernel is therefore deterministic [34].

Property and Configuration Variables The set of property variables is the singleton $\{\texttt{log}\}$ comprising the only variable inspected by Φ_{Det}. The configuration variables are those that describe the thread geometry of the given kernel, namely the number of threads `blockDim`, the thread identifier `threadIdx`, and algorithm- and architecture- specific parameters like the sizes of strides `accumN` and the global arrays `sizeN`. In general, the set of configuration variables depends heavily on the semantics of the kernel being analyzed and must be chosen based on domain knowledge.

Static Information Flow Analysis The last piece of the puzzle is the flow relation. Needless to say, the exact flows-to set for a set of property variables is not computable. We solve this problem, through a forwards taint-propagation based analysis that, given a kernel, a set of property variables V_Φ, and a set of configuration variables Y, checks whether $\mathsf{FlowsTo}(P, V_\Phi) \subseteq Y$. The analysis implements a form of taint-based integrity checking [17, 28].

We formalize the analysis as procedure $\mathsf{StaticFlowsTo}(P, V_\Phi, Y)$, which is implemented as follows. All the input variables *except* those in Y are tainted. The taints are propagated through assignments (data dependencies) and dominating assumes (control dependencies). Finally, we check whether any of the property variables V_Φ are tainted. If not, we can be sure that only the configuration variables flow to the property variables. As the procedure is flow-insensitive, it is sound in the face of multi-threading. Furthermore, as the kernels are branch deterministic, we can conclude that the statically computed flows-to relation is a conservative *overapproximation* of the actual flows-to relation.

PROPOSITION 1. **[Static Flows-To]** *For all branch-deterministic kernels P and variables V_Φ, Y, if $\mathsf{StaticFlowsTo}(P, V_\Phi, Y)$ then $\mathsf{FlowsTo}(P, V_\Phi) \subseteq Y$.*

Access Invariance and Determinism We say that a kernel P is *access invariant with respect to* Y if $\mathsf{StaticFlowsTo}(P, \{\texttt{log}\}, Y)$, that is, if only the configuration variables Y flow to the property variables. From the soundness of static flows-to, we get the following as an immediate corollary of Theorem 1.

PROPOSITION 2. **[CUDA Determinism]** *Let P be a* CUDA *kernel with configuration variables Y. If*

- *P is access invariant with respect to Y*
- *$\tau \in \mathsf{Traces}(P)$ is such that $\tau \models \Phi_{Det}$*

then forall $\tau' \in \mathsf{Traces}(P)$, if $\tau(0) \equiv_Y \tau'(0)$ then $\tau' \models \Phi_{Det}$.

Returning to the `scalarProd` example from Figure 1 we see that `log` has a data dependence on the indices `j`, `i`, `i` and `stride + i` which are used to access shared memory (at lines `7`, `8`, `12` and `12` respectively). The only inputs that flow to these expressions (i.e., the set of property-integrity inputs) are the parameters `threadIdx`, `accumN`, `sizeN`, and `blockDim`, which are a subset of the configuration variables, so the kernel is access invariant. Thus, a single trace that satisfies Φ_{Det} suffices to show that the kernel satisfies Φ_{Det} for all executions over the same configuration.

4. Implementation

We have implemented the analyses described in the previous section in the LLVM Compiler Infrastructure. In particular, we have implemented a static taint-based information flow analysis that checks whether CUDA kernels are access invariant, and a dynamic instrumentation-based analysis that checks whether a kernel satisfies the determinism property Φ_{Det} for a particular execution. Though the formalism in the previous section presented a simplified, two-level memory model and one-dimensional thread geometry, our implementation handles the multi-dimensional thread geometries and three-layer memory hierarchy supported by CUDA. Next, we describe the static and dynamic analyses in greater detail.

4.1 Static Analysis

The static analysis proceeds via three LLVM passes. The first pass recursively inlines all function calls within a kernel, yielding a single call-free kernel. The second pass is a flow-insensitive intraprocedural pointer analysis based on Andersen's algorithm [1] that is used by the third pass, a static taint-based analysis that determines whether data inputs (that is, non-configuration input variables) flow to the property variables.

1. Kernel inlining For simplicity, our current implementation exploits the key restriction that CUDA kernels cannot be recursive or contain indirect functions calls. In particular, we fully inline functions called from kernels, so subsequent analyses can be intraprocedural without loss of precision, with one exception to deal with library functions – we analyze calls to CUDA library functions using specially crafted transfer functions (*i.e.,* summaries) as described below. (Recent NVIDIA cards do support recursion and function pointers. To handle such kernels we need only use interprocedural variants of our current analyses.)

2. Pointer analysis We use a pointer analysis to soundly propagate taints in the presence of aliasing. In particular, if a pointer refers to a tainted heap location, then the taints must be propagated to (dereferences of) all the aliases of the pointer. To this end we have implemented a flow-insensitive, intraprocedural variant of Andersen's algorithm. The precision lost by flow-insensitivity is mitigated by the fact that the LLVM intermediate representation is based on SSA, so the may-point-to set of most variables contain only a single element. In the CUDA setting, the following modifications allow us to further improve the precision of the pointer analysis.

- Our aggressive inlining step makes the analysis context sensitive. We inline all function calls with the exception of calls to CUDA library functions. To deal with those calls we include special transfer functions which account for the function's effects as specified in the documentation.
- Kernel formal parameters with pointer type are optimistically assumed to be *unique* upon entry to the function. That is, there are no other variables that refer to a block of memory pointed to by a formal parameter. Although this assumption is unsound in general, we have manually verified that it holds for all benchmarks in our evaluation. This simplification allows us to avoid making the conservative assumption that pointer function parameters may point to arbitrary memory locations.

3. Taint tracking The direct way to check the access invariance property is to implement $\mathsf{StaticFlowsTo}(P, \{\texttt{log}\}, Y)$ inside LLVM. Unfortunately, this would require us to rewrite the program to add `log` which we avoid for the reasons described in Section 4.2. Instead, we check that the access invariance property holds by statically verifying that the following two conditions hold. First, no address operand of a memory instruction may be affected by a data input. In other words the *flows-to* set of an address may contain only configuration inputs. Second, no memory instruction is control dependent on a tainted value. The latter is subtle and best explained by a small example:

```
if(taintedVar) { A[i] = e; } else { B[i] = e; }
```

Whether array `A` or `B` is written depends on the value of `taintedVar`, even though the array index `i` itself might not be tainted at all. Thus, with a different data input, the program may execute the other branch yielding a race (that does not occur on the test being amplified.) This condition is conservative, as it would flag a violation even when both sides of the branch perform identical accesses. (It is easy to check that these conditions are equivalent to the "direct" approach of checking whether non-configuration input variables flow-to the `log`.)

Our implementation verifies that both conditions are met using a standard worklist algorithm to perform forward taint propagation. We begin with an initial taint set consisting of the kernel's data inputs, then track the propagation of these taints through instructions. During propagation, if either condition is found to be violated, we terminate with the conclusion that the kernel is not access invariant. If instead we reach a fixpoint in which no new taints can be propagated and have not yet found a violation, we conclude that the kernel is access invariant.

There are two channels by which taint can propagate: explicit flow resulting from data dependences, and implicit flow resulting from control dependences. Explicit flow is handled in the straightforward way: we implement transfer functions for each class of instruction or CUDA library function to assign taint to the appropriate variables. For instance, we assign taint to the result of any arithmetic instruction with a tainted operand. We handle control dependences very conservatively by immediately flagging a violation of the access invariance property whenever a branch condition variable is tainted. This would flag an access invariance violation even if neither side of the branch performed any memory access at all, but is a straightforward and sound overapproximation. If no such control dependences exist, we can conclude that the second condition holds, and that no control dependences contribute to a violation of the first condition.

Finally, we must verify the first condition — that no addresses are tainted. LLVM programs calculate addresses with an explicit instruction called `getelementptr` that takes a base pointer and a variable number of operands representing offsets from the base. This instruction is designed to express arbitrary address arithmetic to compute the addresses of data structure elements. Whilst propagating taints, if we ever encounter a `getelementptr` with tainted operand, we conclude that the kernel is not access invariant.

4.2 Dynamic Analysis

The dynamic component of our analysis checks that a particular test execution satisfies the given property Φ_{Det}. Unfortunately, due to the sheer number of threads and frequency of accesses, it is impossible to hold the log in memory and check Φ_{Det} in an online manner. We get around this problem by logging all the accesses on disk and then checking offline that the trace satisfies Φ_{Det}.

Thus, our dynamic analysis consists of a single LLVM pass that instruments all memory accesses with extra instructions to log addresses and thread identifiers on disk. In order to distinguish accesses to global and shared data structures, all allocations and static array declarations for global and shared memory are instrumented to record base address and size. Finally, full 3D thread identifiers and 2D block identifiers are logged for each executing thread.

Barrier Interval Timestamps Recall that a problematic trace is one where different threads access the same shared location during the *same* barrier interval. Thus, in addition to the address and thread identifiers, we record for each access, the *barrier interval* number during which the access occurs, so that the offline check only compares accesses within the same interval. Although addresses and thread identifiers are readily accessible as variables in the uninstrumented code, barrier intervals are not explicitly labeled and must be dealt with specially. We maintain a shared table that contains one entry per thread to track the current barrier interval of each thread. Each element of the table corresponds to one thread's `timestamp` local variable as described in Section 3.4. Every call to `__syncthreads()` is replaced with an augmented version that additionally increments the thread's corresponding table entry. Thus, at any point during execution, the table contains a snapshot of the barrier interval in which each thread is executing. We simply look up the thread's entry in this table to determine the current barrier interval for logging. (The threads obey the barrier synchronization semantics; the intervals are logged solely to facilitate offline trace analysis.)

Determinism via Race Detection After generating a complete log by executing the instrumented kernel, we check that the trace satisfies the determinism property Φ_{Det}. In the presence of the three-level memory hierarchy, the property is satisfied if there are *no* global- or shared- memory races.

A global memory race occurs when a global data structure is written by one thread and accessed by another. Two threads in the same thread block may race only if executing in the same barrier interval. Two threads in different thread blocks cannot synchronize, so may race regardless of barrier interval.

A shared memory race occurs when two threads within the same thread block access a shared data structure within the same barrier interval, and at least one of the accesses is a write. Threads in different thread blocks cannot share the same shared data structure, and thus cannot conflict.

Offline Trace Analysis The sheer magnitude of the logs required that we devise non-trivial means for checking for races using a small memory footprint. Our implementation performs this check via a log postprocessing program that first translates raw addresses into array-offset pairs, then performs a sweep of the resulting log to check for conflicting accesses to each array. To perform the race checks efficiently, we first separate global and shared memory accesses into separate traces. Each trace is then *externally* sorted by address and barrier interval, which are the primary and secondary sort keys, respectively. Finally, a single linear sweep through the sorted traces checks for the aforementioned race conditions.

The above process only stores in memory a small window of the trace at any point: for global arrays, only those accesses to a single address at a time, and for shared arrays, only those accesses to a single address within a single barrier interval. These optimizations were essential in practice. Our initial naïve implementation, which attempted to detect races in unsorted traces, immediately ran into out-of-memory errors due to the size of the traces.

Limitations Our implementation does not currently handle several aspects of CUDA. First, the dynamic analyses are incompatible with programs using OpenGL because CUDA emulation does not support compilation of such programs. We resort to using CUDA emulation because we cannot perform logging when executing on real GPU hardware. Second, our analyses do not support code that uses the CUDA Driver API, a low-level framework above which the CUDA language and runtime are built, nor code that contains assembly instructions, atomic intrinsics, or warp voting functions. Third, our dynamic analysis does not track accesses to texture or constant memory, as they are strictly read-only during kernel execution and thus are not subject to races within the kernel. A final caveat is that we assume that conflicting accesses must be to the same exact address – our implementation has no notion of conflict between *unaligned* accesses, though presumably a higher-level type system could ensure the absence of such misaligned accesses.

5. Evaluation

Our goal is to evaluate the overall effectiveness of our approach in the setting of CUDA programs. To this end, we evaluate the effectiveness of our static invariance analysis (Section 5.2), the effectiveness of our dynamic analysis (Section 5.3), and the running-time of our approach (Section 5.4). Before presenting these results, we first describe our experimental setup (Section 5.1).

5.1 Experimental Setup

To evaluate our approach, we started with the 67 benchmarks in the NVIDIA CUDA SDK Version 3.0. Of these 67 benchmarks, 6 are small tutorial-like benchmarks with trivial kernels, and so we leave these out. Of the remaining benchmarks, 33 contain features which our analysis cannot run on, due to the limitations previously mentioned. We therefore omitted these as well, which leaves us with 28 benchmarks, for which we report results. These 28 benchmarks contain a total of 76 kernels.

The static analysis runs on one kernel at a time. We manually examined each kernel to determine the correct set of data inputs (non-configuration input variables) to supply the static analysis. We tried to minimize the set of configuration variables by manually examining code to determine the domain-specific interpretation of each input. The separation of data and condition variables is usually quite immediate. As an example, for the `scalarProd` benchmark, the set of data input variables is `d_C`, `d_A`, and `d_B` (since these are the input vectors). Since `sizeN` represents the size of the data set, we consider it to be a configuration variable.

For the dynamic analysis, we run each benchmark containing access invariant kernels with instrumentation turned on. Benchmarks with multiple kernels will produce logs containing entries from several kernels. A postprocessing step separates the entries corresponding to each kernel. The logs can then be processed one kernel at a time.

5.2 Static Invariance Analysis

The results of our static invariance analysis are shown in Figure 4. Of the 76 kernels examined, 52 were shown to be access invariant by our static information flow analysis. These kernels are spread across a variety of benchmarks from different domains. Additionally, even benchmarks that demonstrated variance in some kernels often have at least one kernel that is in fact access invariant. For instance, although `histogram` contains 2 kernels that are not access invariant, it contains 2 that are. This aligns with the fact that CUDA algorithms are often implemented as multiple distinct kernels, each performing a different phase of the algorithm, some of which may be access invariant while others are not.

We have found three distinct patterns that cause a kernel to be access variant. For each pattern we provide an example from the benchmark suite.

Direct Flow The first pattern is the most direct: a memory access address is derived directly from an input value. This pattern is demonstrated by the `histogram` benchmark, which bins each byte of its input based on the value of its higher order bits. A straightforward sequential implementation of histogram is sketched in Figure 5. This implementation iterates over each element of `data`, uses the higher order bits to index into the array `bin` whose elements represent bin counts, then increments the indexed element. Although the parallel CUDA implementation is complicated by distribution of computation across multiple threads, the fundamental calculation is the same: the higher order bits of each byte are used to address into a shared array which contains the bin counts. The access variance of this kernel is a result of the fact that the bin address is derived directly from an input value.

Benchmark	LOC	#K	%I
reduction	695	7	100%
sortingNetworks*	571	6	100%
convolutionFFT2D	384	3	100%
fastWalshTransform	243	3	100%
convolutionSeparable	318	2	100%
convolutionTexture	314	2	100%
cppIntegration	125	2	100%
simplePitchLinearTexture	138	2	100%
transpose	146	2	100%
binomialOptions*	354	1	100%
BlackScholes*	276	1	100%
dwtHaar1D	266	1	100%
FDTD3d	818	1	100%
matrixMul	216	1	100%
scalarProd	136	1	100%
simpleCUFFT	137	1	100%
simpleTexture	121	1	100%
vectorAdd	92	1	100%
dct8x8	1402	8	87%
histogram	431	4	50%
lineOfSight	180	2	50%
radixSort	1894	10	30%
MonteCarlo	836	4	25%
eigenvalues	1901	4	0%
MersenneTwister	287	2	0%
quasiRandomGenerator	637	2	0%
clock	69	1	0%
dxtc	829	1	0%
28 Benchmarks	13816	76	68%

Figure 4. Static Analysis Results: **LOC**=Lines of Code, **#K**=Number of Kernels, **%I**=Percent Invariant Kernels, *modified as detailed in Section 5.2

```
void histogram(char *data, int *bin, int size) {
for(int i = 0, i < size, i++)
   bin[data[i] && 0x3FU]++;
}
```

Figure 5. Sequential Histogram

Indirect Flow The second pattern occurs in kernels in which a critical control flow statement depends on a condition derived from input. This is demonstrated by the `eigenvalues` benchmark. This benchmark calculates the eigenvalues of a matrix with an algorithm that subdivides intervals dependent on the values of the input matrix. This dependence is fundamental to the algorithm and cannot be removed.

Correctable Indirect Flow The `sortingNetworks` benchmark performs a bitonic sort, which is a sorting algorithm with the property that the comparisons it performs are independent of its input values. Whether elements are swapped, however, depends on the result of a comparison which ultimately derives from the magnitude of input values. Intuitively, it would seem that the kernel is *almost access invariant*, save for this conditional swap. It turns out that we can perform a simple rewrite, as shown in Figure 6, which effectively converts the control dependence into a data dependence. With this rewrite, `sortingNetworks` can be verified access invariant because we have removed a control dependence without introducing taint to the array indices. We have been able to perform similar transformations to the `binomialOptions` and `BlackScholes` kernels.

```
/**  original version   **/
if(A[i] > A[j]) {
    t = A[i]; A[i] = A[j]; A[j] = t;
}

/** transformed version **/
bool cond = (A[i] > A[j]);
t    = A[i];
A[i] = cond * A[j] + (!cond) * A[i];
A[j] = cond * t    + (!cond) * A[j];
```

Figure 6. Removing a control dependence

Configuration Variable Selection The difference between a data input and a configuration variable is a semantic distinction that ultimately depends on the domain-specific meaning of each input to the kernel – it cannot be fully automated. In our experiments, we have manually examined kernels to extract this distinction.

However, the difference between a property-integrity input, and one which is not, is based on which inputs influence the access invariance property and which do not. This classification *can* be automated with our invariance analysis. We simply run our invariance analysis on each input in turn, and determine which inputs cause our analysis to report a violation of the access invariance property. The end result is that we have found a minimal set of property-integrity inputs.

For all kernels we have classified as access invariant, we would expect our set of manually chosen configuration variables to contain the set of property-integrity inputs. In addition, we would expect that for all variant kernels, at least one data input is in fact a property-integrity input. We have verified that both are true.

A final observation is that in cases where the set of property-integrity inputs is smaller than the set of chosen configuration variables, we can amplify our guarantees to an even larger set of inputs. We found this to be the case for `convolutionSeparable`, which contains two kernels, one which performs an image transform on the rows of the image, and another which acts on the columns. Both kernels take parameters `imageW` and `imageH` which indicate the height and width of the image being transformed, but the row- and column- transforming kernels are only variant with respect to the width and height parameters, respectively. Thus, although we manually specified both parameters to be configuration inputs, in fact we could have removed one and still maintained the access invariance property. In total, we have found that for 11 of the kernels the set of property-integrity inputs is in fact a subset of the set of configuration variables.

5.3 Dynamic Analysis

Our dynamic analysis found that most of the dynamic runs were race-free and deterministic. However, our analysis detected a race violation in the full implementation of `scalarProd`. The race was determined to be a result of a missing `__syncthreads()` call after a complete dot product had been calculated and stored into the output array, which corresponds to line 13 of Figure 1. Without this barrier, thread 0 might still be reading values from the `accumResult` array while its neighbors had already begun calculation on the next pair of vectors, potentially performing a destructive update before the previous result had been completed.

In practice, however, this race will not manifest itself due to the *warp-synchronous* nature of CUDA's execution model: neighboring threads are logically grouped into *warps* of 32 threads that execute synchronously on hardware. Thus, because threads 0 and 1 belong to the same warp, they will not exercise the race described.

Races were also detected in two other benchmarks, `histogram` and `reduction`. In both cases, the benchmarks once again relied on warp-synchronous execution to prevent races in the absence of explicit barrier synchronization. It is important to note, however, that warp-synchronous execution relies on the behavior of underlying hardware, which may change for future generations of GPUs or different target architectures, and cause these to become real errors.

5.4 Performance Evaluation

The running time of our static analysis was less than one second for most benchmarks, with none taking more than 10 seconds with two notable exceptions: `radixSort` which took 36m, and `reduction` which took 3.5m. The reason is that these benchmarks makes heavy use of C++-style templates. When compiled, each template specializes into multiple kernels, each of which is analyzed. `radixSort` instantiates 271 kernels, and our analysis takes an average of 8 seconds per instantiated kernel. Our analysis takes an average 1.6s on each of the 132 kernels instantiated by `reduction`.

To evaluate the performance penalty of our dynamic instrumentation and race detection, we measured running times for each benchmark with and without the analysis enabled. The geometric mean and median slowdown were 18X and 12X, respectively, while the total slowdown, measured by summing the running times of all benchmarks, was 310X. There were two benchmarks, `MonteCarlo` and `convolutionSeparable`, that had running times more than an order of magnitude higher than the others, at 2093X and 1324X, due to the size of their logs: `convolutionSeparable` in particular, produced a 508 GB log. We have found that slowdowns are highly correlated with the size of the memory traces, leading us to believe that the bottleneck in performance is disk I/O for writing log entries. This could be alieviated through the use of more sophisticated buffering techniques, but we leave this for future work.

6. Related Work

In this section we discuss the literature related to our work, which fall under two broad categories: race-freedom and determinism; and test amplification. The literature on enforcing race-freedom and determinism on shared memory multithreaded programs is enormous: we limit ourselves to work that studies problems similar to the domain of GPU programs, with massive, fine-grained data-sharing. Because test amplification is such a broad concept, it has appeared in many different forms in the literature. We present an indicative, albeit necessarily incomplete selection of work representative of the breadth of its application.

Race-Freedom and Determinism One closely related piece of work is [4], a dynamic analysis for detecting races in CUDA programs. The work uses an instrumentation much like our own, to generate logs that can be used to dynamically detect races and barrier conflicts. Of course, without amplification, the results only hold for one run at a time, which is not likely to be very useful as the instrumentation imposes large overheads.

Our work is greatly inspired by PUG [24], a tool that analyzes CUDA programs via the machinery of symbolic execution. PUG logically encodes the program executions and uses an SMT solver to check whether different threads can race within a barrier interval. PUG has several optimizations that mitigate the explosion in thread interleavings. One of these is that it is limited to checking a program with just two threads. Similarly, [34] uses numeric abstract interpretation to compute the sets of indices used to access shared arrays. As both these techniques are fully static, they are limited to programs for which it can infer suitable simple (linear) loop invariants, which precludes usage on many CUDA kernels which often have complex, non-linear loop-invariants computed with modular arithmetic and bitwise operations.

There is an enormous literature on dynamic and static race detection for general concurrent programs. Two classical examples include [10] and [31] that respectively pioneered the use of static and dynamic *locksets* in order to check for the absence of races. Several authors have built upon that line of work using types [13], dataflow analysis [12], and even greatly extended it to checking for higher level properties like atomicity [14, 15] and determinism [30]. Unfortunately, these methods don't apply in our setting as CUDA eschews lock-based synchronization.

Several authors have proposed dynamic mechanisms for enforcing determinism by constraining the scheduler. These include techniques that modify the language run-time [3, 26] or the OS [2]. Due to the run-time overheads, these methods are unlikely to apply in the GPU setting, where programs exhibit a high degree of fine-grained sharing, for example to maximize memory bandwidth.

The general idea of *non-interference* [20], in which static information flow has it roots, has been proposed as a means of formalizing correctness properties of multithreaded programs [11]. Previous work has also studied the connection between information flow and concurrency. In particular, [36] describes the notion of observational determinism wherein a program is secure iff its observable behaviors are independent of secure inputs and scheduling, and [33] describes a type system for observational determinism. Our novel contribution is to demonstrate how of static information flow can be used to amplify the results of a single execution across all runs over the same configuration, thereby establishing the property for arbitrary thread interleavings and data values, and to empirically demonstrate the effectiveness of this technique for verifying CUDA programs.

Test Amplification Recent work on dynamic test generation has employed test amplification with the aim of increasing code coverage [5, 6, 18, 32]. Collectively dubbed *concolic testing*, these techniques use dynamic symbolic execution to collect logical constraints representing input-dependent control flow conditions, then solve the constraints to obtain inputs that steer execution along different paths. At a high level, all these techniques attempt to maximize code coverage by avoiding the generation of test inputs that redundantly follow the same paths. In this context, the identification of equivalence classes of inputs that exercise the same paths may be viewed as an instance of test amplification: a single input is amplified to *represent* its equivalence class of inputs.

Another form of test amplification has also been explored extensively in work on runtime fault monitoring [19, 21, 23]. By piggybacking symbolic execution on top of a dynamic run, these techniques are able to detect property violations across a much larger space of inputs than those exercised by a particular concrete execution. This method has been used to find buffer overflows [23], generalized to predict more generic property violations [21], and adapted to generate concrete test inputs that exercise those violations [19]. Because these techniques only check properties along the tested execution path, they rely on exhaustive path search to verify properties across all executions.

The described dynamic test generation and runtime monitoring techniques fall under what we call *symbolic-execution-based test amplification*, an umbrella term for techniques that employ symbolic execution to learn additional information from a concrete execution. Flow-based test amplification is potentially less precise than symbolic execution, but is also much cheaper, and as demonstrated by our evaluation, quite effective in the CUDA setting. Recent work on verifying memory-safety of programs with floating-point computation [17] has even combined dynamic symbolic execution with a static flow analysis which establishes that floating-point values do not interfere with non-floating point values, thereby allowing amplification of symbolic execution guarantees over floating point computations. The effectiveness of flow-based test amplification stems from the fact that programs in the domain isolate the *data* values being operated on (e.g. input arrays or floating point numbers) from those used to determine *control-flow* and *memory accesses*.

Finally, test amplification has also appeared in work on the analysis of general concurrent programs. In [35] the authors describe two dynamic analyses for detecting atomicity violations in concurrent Java programs: one that employs Lipton's theory of reduction [25] to reason about commutativity of concurrent events to infer atomicity, and another that searches for unserializable permutations of events to detect atomicity violations. Both analyses use test amplification in the sense that they examine a single execution but are capable of detecting atomicity violations occurring in other possible interleavings. A different instance of test amplification appears in [7], which proposes a relaxation of the classical happen-before causality partial order [16, 22] to admit a greater number of compatible interleavings. This relaxation amplifies the partial order to a larger space of possible executions, thus opening subsequent analyses to detect property violations in more interleavings. A key contribution of our work is to demonstrate that enumerating interleavings is simply unnecessary for many CUDA kernels: we can apply static information flow analysis to generalize our determinism guarantee to all possible interleavings from a single execution.

Acknowledgements

We wish to thank Mingxun Wang for his part in many fruitful discussions and his help on initial feasibility experiments. We are grateful to our shepherd, Patrice Godefroid, and the anonymous reviewers for their comments and feedback for improving the paper.

References

[1] L. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, 1994.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, pages 193–206, 2010.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, pages 53–64, 2010.

[4] M. Boyer, K. Skadron, and W. Weimer. Automated Dynamic Analysis of CUDA Programs. In *STMC*, 2008.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM.

[7] F. Chen and G. Roşu. Parametric and sliced causality. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 240–253, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] N. Corporation. Cuda toolkit 3.0. `http://developer.nvidia.com/cuda-toolkit-30-downloads`, 2010.

[9] N. Corporation. Cuda accelerated applications. `http://www.nvidia.com/object/cuda_app_tesla.html`, 2011.

[10] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Workshop on Parallel and Distributed Debugging*, pages 85–96, 1991.

[11] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *PADD*, 1988.

[12] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *SOSP*. ACM Press, 2003.

[13] C. Flanagan and S. Freund. Type-based race detection for Java. In *PLDI 00: Programming Languages Design and Implementation*, pages 219–232. ACM, 2000.

[14] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, 2004.

[15] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, pages 191–202. ACM, 2003.

[16] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[17] P. Godefroid and J. Kinder. Proving memory safety of floating-point computations by combining static and dynamic program analysis. In *ISSTA*, 2010.

[18] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[19] P. Godefroid, M. Y. Levin, and D. A. Molnar. Active property checking. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 207–216, New York, NY, USA, 2008. ACM.

[20] J. A. Goguen and J. Meseguer. Security policies and security models. *Security and Privacy, IEEE Symposium on*, 0:11, 1982.

[21] P. Joshi, K. Sen, and M. Shlimovich. Predictive testing: amplifying the effectiveness of software testing. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, ESEC-FSE companion '07, pages 561–564, New York, NY, USA, 2007. ACM.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[23] E. Larson and T. Austin. High coverage detection of input-related security faults. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[24] G. Li and G. Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. In *FSE*, New York, NY, USA, 2010. ACM.

[25] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18:717–721, December 1975.

[26] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *SOSP*, pages 327–336, 2011.

[27] F. Masdupuy. Semantic analysis of interval congruences. In *Formal Methods in Programming and Their Applications*, LNCS 735. 1993.

[28] A. Myers. JFlow: Practical mostly-static information flow control. In *POPL 99: Principles of Programming Languages*, pages 228–241. ACM, 1999.

[29] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *ESOP*, pages 348–362, 2009.

[30] C. Sadowski, S. N. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *ESOP*, pages 394–409, 2009.

[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15, 1997.

[32] K. Sen, D. Marinov, and G. 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-13, pages 263–272, New York, NY, USA, 2005. ACM.

[33] T. Terauchi. A type system for observational determinism. In *CSF*, pages 287–300, 2008.

[34] M. T. Vechev, E. Yahav, R. Raman, and V. Sarkar. Automatic verification of determinism for structured parallel programs. In *SAS*, pages 455–471, 2010.

[35] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32:93–110, February 2006.

[36] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *CSFW*, pages 29–, 2003.