Verifying Static Analysis Tools
(of GPU Programs)

Udaya Sathiyamoorthy and Tiago Cogumbreiro

January 14, 2023

Trends in Functional Programming

Overview

This talk is about using an intermediate representation of CUDA kernels (called MAPs) to test a static analysis tool (Faial) written in OCaml

- Motivation
- Memory Access Protocols
- Contributions
- Evaluation
- Future Work (Including Preliminary Results)
- Conclusion

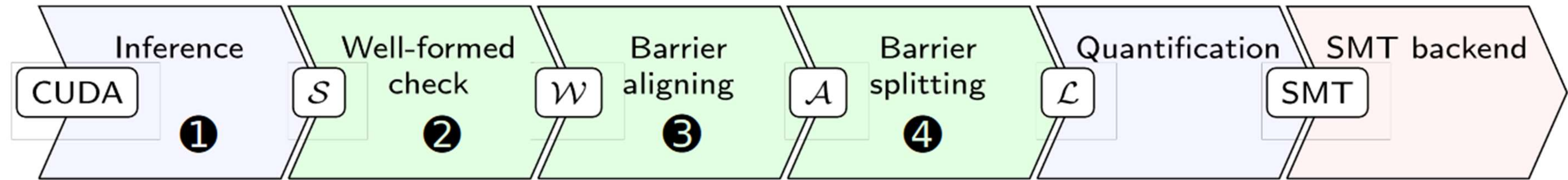
Motivation

Background

- CUDA is an API to program GPUs (extension of C/C++)
- **CUDA programs (kernels)** generally use structured loops, terminate, and have limited aliasing
- CUDA kernels are parallel programs that use shared memory to communicate
- Faial is a static analysis tool for CUDA programs
- Faial is written in OCaml and interfaces with libclang to parse CUDA
- Faial verifies data-races and bank-conflicts
  - Data-race: a concurrency error caused by two concurrent and unsynchronized memory accesses (one is a write)
  - Bank-conflicts: performance degradation caused by memory access patterns

[CAV'21]: Tiago Cogumbreiro, Julien Lange, Dennis Liew & Hannah Zicarelli: Checking Data-Race Freedom of GPU Kernels, Compositionally.

Architecture of Faial, a Data-Race Freedom Checker for CUDA



Source: [CAV'21]

- (1) generates an **intermediate representation (IR)**
- (2), (3), and (4) perform the analysis of the IR to prove data-race freedom (DRF) via an SMT solver (Z3)
- (2), (3), and (4) have been formalized and the correctness of the analysis has been established in Coq
- How can we test (1)?

[CAV'21]: Tiago Cogumbreiro, Julien Lange, Dennis Liew & Hannah Zicarelli: Checking Data-Race Freedom of GPU Kernels, Compositionally.

Inference of the IR (MAPs)

- Inference is the process of abstracting the C code into our IR, which includes:
  - Abstracting away data being written to/from arrays
  - Code slicing any code unrelated to index expressions/concurrency
  - Inferring loop bounds/strides (additive/multiplicative)
  - Inlining any kind of local aliasing
    - `*a = a + threadIdx.x`
  - Inlining local assignments
    - `int i = blockIdx.x * blockDim.x + threadIdx.x;`

Assessing The Correctness of Faial's Inference

- Fuzzing
- Property testing
- Formalization (ongoing) [PLACES'22]
- **What if we use the IR itself to test Faial?**

[PLACES'22]: Dennis Liew, Tiago Cogumbreiro, & Julien Lange: Provable GPU Data-Races in Static Race Detection.

Memory Access Protocols

Memory Access Protocols (MAPs)

- The IR used to analyze CUDA kernels
- Available as an OCaml data type
- Consist of four types of instructions:
  - Array accesses
  - Barrier synchronizations
  - Conditionals
  - Structured loops (foreach) with potentially unknown bounds

```
type inst =
  | Acc of (Variable.t * Access.t)
  | Sync
  | Cond of bexp * inst list
  | Loop of Range.t * inst list
```

CUDA Analysis Codebase

- Wide range of options to summarize kernel data
  - Variables (shared, global, etc.)
  - Max loop depth (sync vs. unsync)
  - Array usage

```
stmt -> json
let summarize (s:stmt) : json =
  let elems = all_accesses s |> List.of_seq in
  let cond_elems = cond_accesses s |> List.of_seq in
  let get_vars (x, y) = List.map fst x, List.map fst y in
  let reads, writes =
    elems
    |> List.partition (fun (_, a) -> Access.is_read a)
    |> get_vars
  in
  let c_reads, c_writes =
    cond_elems
    |> List.partition (fun (_, a) -> Access.is_read a)
    |> get_vars
  in
  `Assoc [
    "conditional reads", var_list_to_json c_reads;
    "conditional writes", var_list_to_json c_writes;
    "writes", var_list_to_json reads;
    "reads", var_list_to_json writes;
  ]
```

Inference of MAPs

```
for (int x = 0; x < N; x++) {
    if (tid % 2 == 0) {
        int y = A[x];
        A[x] = y * y;
    }
}
```

CUDA

Inference →

```
for x ∈ 0..N {
    if (tid % 2 = 0) {
        rd[x];
        wr[x];
    }
    else { skip }
}
```

MAPs

- Issue: not all protocols are inferred correctly
  - Some protocols may be misinterpreted
  - Some instructions may go missing
- **Can we leverage Faial's abstraction to ensure the inference is correct?**

Contributions

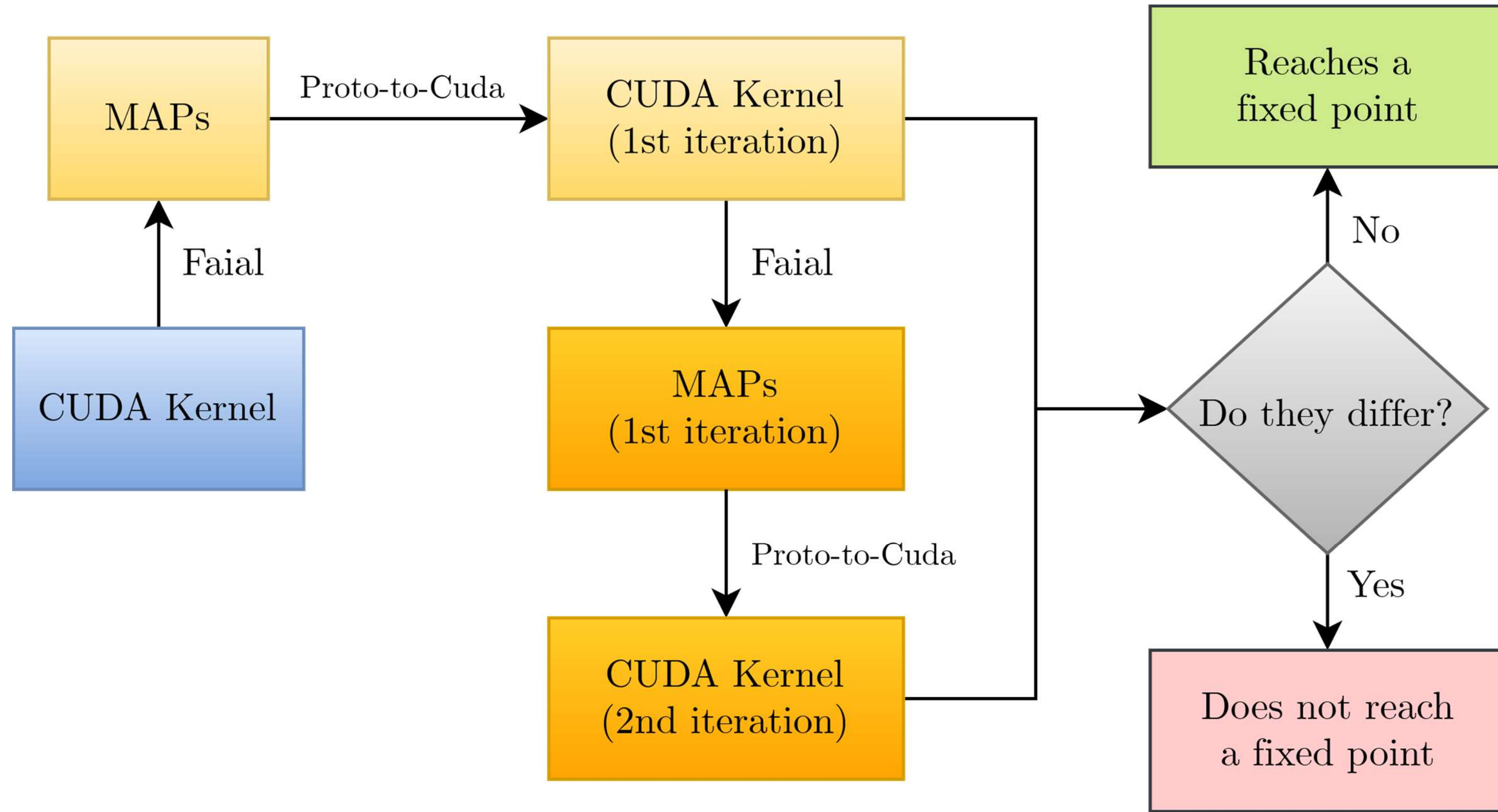
Contributions

- **Implemented a “reversal” of the inference**: going from MAPs to CUDA
- Introduced a technique to test the inference of MAPs
- Detected and fixed several bugs in Faial

Testing Methodology

- How do we represent the IR concretely?
  - Code generation: extend Faial to generate a CUDA representation of the MAPs
  - AKA, Proto-to-Cuda
- How do we verify Faial's inference?
  - Fixed point analysis: test whether MAPs are consistent across Proto-to-Cuda iterations
  - Cannot compare MAPs directly, so compare the CUDA versions of each set of MAPs
  - Classify the differences between kernel iterations
  - Use differences to find bugs in the inference
- Goal: fix bugs identified in stress tests to facilitate further testing

Stress Testing Pipeline



Proto-to-Cuda

Instruction	Protocol	CUDA
Read	`rd A[x];`	`__dummyA = A[x];`
Write	`wr A[x];`	`A[x] = __dummyA_w();`
Sync	`sync;`	`__syncthreads();`
Conditional	`if (c = true) {...}` `else {...}`	`if (c == true) {...}`
Loop	`for` $x \in$ `0..N {...}`	`for (int x = 0; x < N; x += 1) {...}`

- Translates each instruction from MAPs into CUDA code
- Control-flows are simple syntax transformations
- Use a convention to simulate array accesses:
  - Local dummy variable to read from arrays
  - External function prototype to write to arrays

Fixed Point Analysis

```
__global__
void kernel(int *A, int *B)
{
    int x = A[threadIdx.x];
    B[threadIdx.x] = x;
}
```

CUDA Kernel

```
extern __device__ int __dummyA_w();
extern __device__ int __dummyB_w();
__global__
void kernel(int *A, int *B)
{
    int __dummyA;
    int __dummyB;
    __dummyA = A[threadIdx.x];
    B[threadIdx.x] = __dummyB_w();
}
```

1st Iteration
CUDA Kernel

Fixed Point Analysis (Success)

```
extern __device__ int __dummyA_w();
extern __device__ int __dummyB_w();
__global__
void kernel(int *A, int *B)
{
    int __dummyA;
    int __dummyB;
    __dummyA = A[threadIdx.x];
    B[threadIdx.x] = __dummyB_w();
}
```

1st Iteration
CUDA Kernel

```
extern __device__ int __dummyA_w();
extern __device__ int __dummyB_w();
__global__
void kernel(int *A, int *B)
{
    int __dummyA;
    int __dummyB;
    __dummyA = A[threadIdx.x];
    B[threadIdx.x] = __dummyB_w();
}
```

2nd Iteration
CUDA Kernel

Reaches Fixed Point

Fixed Point Analysis (Failure)

```
extern __device__ int __dummyA_w();
extern __device__ int __dummyB_w();
__global__
void kernel(int *A, int *B)
{
    int __dummyA;
    int __dummyB;
    __dummyA = A[threadIdx.x];
    B[threadIdx.x] = __dummyB_w();
}
```

1st Iteration
CUDA Kernel

Proto-to-Cuda

(Incorrectly Inferred)

```
extern __device__ int __dummyA_w();
extern __device__ int __dummyB_w();
__global__
void kernel(int *A, int *B)
{
    int __dummyA;
    int __dummyB;
    __dummyA = A[threadIdx.x + 1];
    B[threadIdx.x] = __dummyB_w();
}
```

2nd Iteration
CUDA Kernel

Does Not Reach Fixed Point

Difference Report

1st Iteration Only

2nd Iteration Only

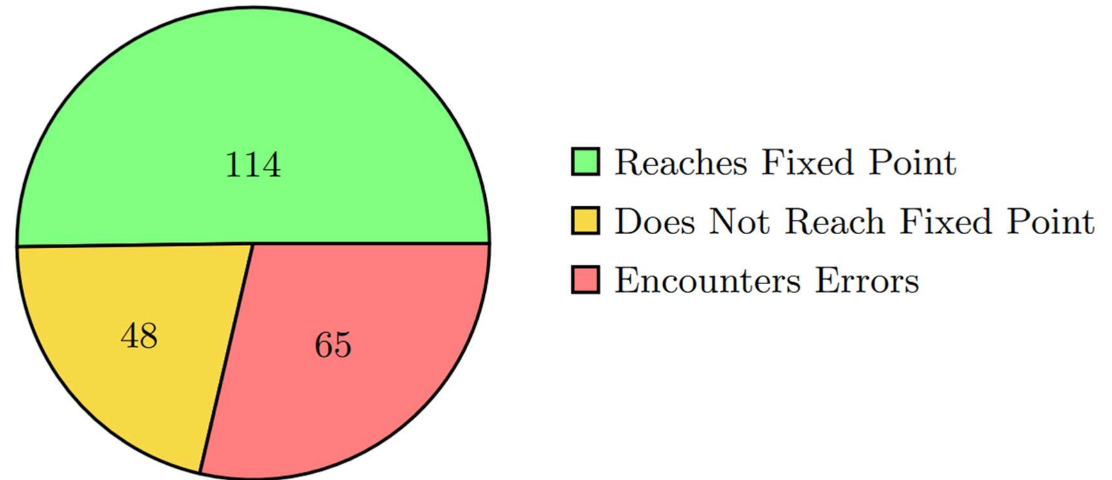
```
      int __dummyB;
-     __dummyA = A[threadIdx.x];
+     __dummyA = A[threadIdx.x + 1];
      B[threadIdx.x] = __dummyB_w();
```

Both Iterations

Evaluation

Stress Testing Faial

- Used GPUVerify's [CAV'14] dataset, a collection of 227 real-world kernels, to stress test Faial
- Categorized kernels into three categories:
  - Fixed: no differences between iterations
  - Non-fixed: difference between iterations
  - Error: first iteration could not be parsed
- First part of experiment: classified the differences between non-fixed kernels
- Distinguish between <u>syntactic differences</u> and <u>semantic differences</u>
  - Syntactic – same meaning, different style
  - Semantic – different meaning



Initial Results

[CAV'14]: Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew & Shaz Qadeer: Engineering a Static Verification Tool for GPU Kernels.

Classifying Differences Between Kernels

	Syntactic Differences		Semantic Differences							
Difference	Unary (-) to Binary	Conditional Expansion	Missing Read	Missing Write	Missing Conditional	Missing Loop	Array Indices Flipped	Type Conversion	Type Truncated	Introduce Unknowns
# Kernels	10	5	13	11	4	2	11	17	14	2

Classifying Differences Between Kernels

	Syntactic Differences		Semantic Differences							
Difference	Unary (-) to Binary	Conditional Expansion	Missing Read	Missing Write	Missing Conditional	Missing Loop	Array Indices Flipped	Type Conversion	Type Truncated	Introduce Unknowns
# Kernels	10	5	13	11	4	2	11	17	14	2

- Discovered three types of differences:
  - Expression conversions
  - Missing instructions
  - Structural changes to variables

Expression Conversions

	Syntactic Differences		Semantic Differences							
Difference	Unary (-) to Binary	Conditional Expansion	Missing Read	Missing Write	Missing Conditional	Missing Loop	Array Indices Flipped	Type Conversion	Type Truncated	Introduce Unknowns
# Kernels	10	5	13	11	4	2	11	17	14	2

Difference

```
-    for (int dy = -1; dy < 2; dy = dy + 1) {
-        for (int dx = -1; dx < 2; dx = dx + 1) {
+    for (int dy = 0 - 1; dy < 2; dy = dy + 1) {
+        for (int dx = 0 - 1; dx < 2; dx = dx + 1) {
```

- Does not change the meaning of the program
- Can be safely ignored

Missing Instructions

	Syntactic Differences		Semantic Differences							
Difference	Unary (-) to Binary	Conditional Expansion	Missing Read	Missing Write	Missing Conditional	Missing Loop	Array Indices Flipped	Type Conversion	Type Truncated	Introduce Unknowns
# Kernels	10	5	13	11	4	2	11	17	14	2

Difference

```
      __dummyA = A[(a + (wA * threadIdx.y)) + threadIdx.x];
-     __dummyB = B[(b + (wB * threadIdx.y)) + threadIdx.x];
      __syncthreads();
```

- Does change the meaning of the program (bug)

Missing Instructions

	Syntactic Differences		Semantic Differences							
Difference	Unary (-) to Binary	Conditional Expansion	Missing Read	Missing Write	Missing Conditional	Missing Loop	Array Indices Flipped	Type Conversion	Type Truncated	Introduce Unknowns
# Kernels	10	5	13	11	4	2	11	17	14	2

Original Program

```
for (int a = aBegin, b = bBegin;
     a <= aEnd;
     a += aStep, b += bStep)
{
```

b goes missing

- Does change the meaning of the program (bug)
- Can occur due to unsupported language features, e.g., loops with multiple variables

Structural Changes to Variables

	Syntactic Differences		Semantic Differences							
Difference	Unary (-) to Binary	Conditional Expansion	Missing Read	Missing Write	Missing Conditional	Missing Loop	Array Indices Flipped	Type Conversion	Type Truncated	Introduce Unknowns
# Kernels	10	5	13	11	4	2	11	17	14	2

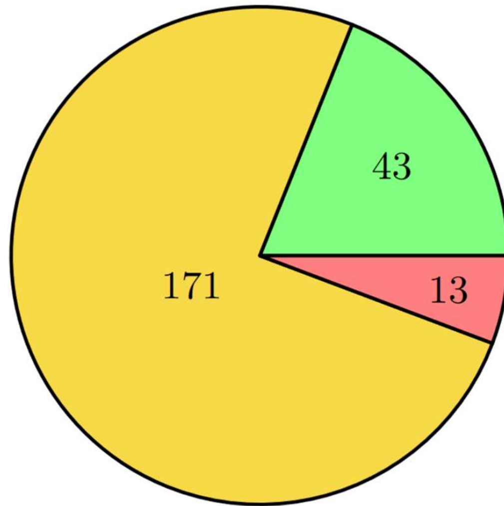
Difference

```
-    transpose_shared_data[threadIdx.x][threadIdx.y] = __dummytranspose_shared_data_w();
+    transpose_shared_data[threadIdx.y][threadIdx.x] = __dummytranspose_shared_data_w();
```

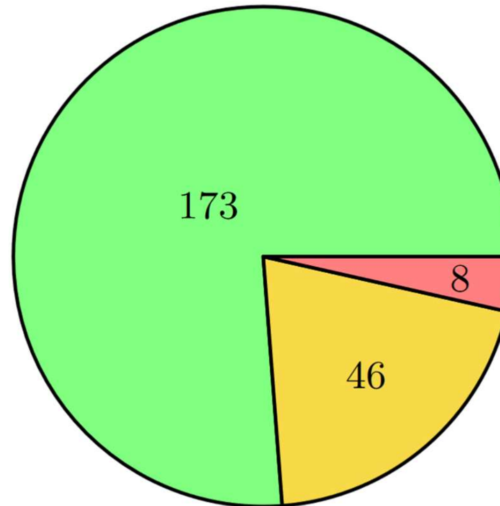
- Discovered a bug in serialization code: array indices were reversed
- Traced indices to original program and MAPs to confirm

Outcome of Experiment

- Many kernels could not be tested due to errors re-parsing the 1st iteration kernel
- Second part of experiment: improve code generation to target more kernels



Results (Before Cleanup)



Results (After Cleanup)

- Reaches Fixed Point
- Does Not Reach Fixed Point
- Encounters Errors

Future Work

Static Analysis of CUDA Programs is Challenging

- Many static analysis tools are brittle, hindering tests with real-world kernels
- **Can we leverage MAPs to simplify the dataset to enable broader comparative studies?**

Tool	Faial	RaCUDA	PUG	GPUVerify
Dataset Supported	100%	10%	38%	100%
Unsupported Language Features	-	Some data types (i.e., shorts), several operators, multi-dimensional arrays, C++ templates, etc.	C++ templates, classes, while loops	-

Static Analysis of CUDA Programs is Challenging

- **RaCUDA supports 10%** of dataset (22/227 kernels)
  - Static performance analyzer
  - Lacks support for some data types/multi-dimensional arrays
- **PUG supports 38%** of dataset (86/227 kernels)
  - Static data-race freedom analyzer
  - Lacks support for C++ templates

Using MAPs to Scale RaCUDA

- Ran Proto-to-Cuda on the CAV14 dataset to generate RaCUDA-compatible kernels
- Tested RaCUDA on the generated dataset
- Preliminary Results:
  - 209 kernels could be analyzed
  - 18 kernels encountered errors
- Note: still need to test the semantics of RaCUDA's analysis

Conclusion

Conclusion

1. Developed a testing technique to exercise the correctness of the inference algorithm
2. Preliminary results of using code generation to simplify the syntax of kernels, while preserving the concurrency characteristics of kernels
3. Our testing framework identified 9 bugs in our tool
4. Our code generation allowed RaCUDA to analyze 187 new kernels (10% vs. 92%)