Verifying Static Analysis Tools

Udaya Sathiyamoorthy* Tiago Cogumbreiro
u.sathiyamoorthy001@umb.edu tiago.cogumbreiro@umb.edu

University of Massachusetts Boston, Boston, USA

Abstract. We present a technique to stress-test the correctness of static analysis tools for CUDA programs, involving code generation and fixed point analysis. Our method revolves around a family of behavioral types called memory access protocols (MAPs), an abstraction used by Faial to determine whether CUDA programs are data-race free. In this paper, we introduce a code generation technique to represent MAPs in a form comprehensible to static analyzers (CUDA code). We use fixed point analysis to detect consistency errors in how programs are represented. We perform white-box testing with Faial, a tool we are already familiar with, to simultaneously fix bugs and facilitate further testing.

1 Introduction

GPUs have never been more widespread or versatile as they are today. Thanks to their ability to leverage parallelism to process massive amounts of data in real-time, they are imperative to numerous high-performance computing tasks, such as training neural networks [8] and medical physics [7]. Consequently, it is vital for software developers to be able to write GPU programs correctly and efficiently. One way to ensure correctness is by using static analyzers to detect common bugs and bottlenecks in concurrent code, such as data-races or bank conflicts. Unfortunately, static analyzers come with some caveats.

While some static analysis tools are more robust than others, most are either unable to handle all types of input, cannot identify every potential problem in code, or report false alarms in an attempt to locate issues exhaustively. For the static analysis of CUDA programs in particular, we have found that existing tools are quite brittle, *e.g.*, limited support for the full C++ specification, required to fully analyze CUDA programs. For instance, RaCUDA [6] lacks support for some C types (*e.g.*, `short`) and multi-dimensional arrays, while PUG [4] lacks support for C++ templates. As a result, these tools underperform in experiments involving hundreds of varied real-world programs: PUG was only able to analyze $\sim 37\%$ of a dataset of 228 real-world kernels [2].

This poses a difficult dilemma for GPU programmers looking to expand their static analysis toolchain. Although the state-of-the-art provides a means of analysis for CUDA programs, it is either unreliable due to the presence of false positives or is incapable of handling more complex code. In industry, this can often

* Student researcher

inhibit the adoption of such tools [3]. Hence the motivation behind Faial [2], a tool that can correctly verify 42% more programs than the state-of-the-art, with a lower false-positive rate than all related work [2]. Faial does this via a compositional analysis for DRF. It first takes a CUDA kernel, then infers an intermediate representation (i.e., a protocol), and finally analyzes said representation to determine whether the kernel is racy.

Our goal is to leverage the frontend of Faial such that its abstraction can be used for other purposes, such as stress testing both itself and other static analysis tools. We accomplish this by implementing a functionality in Faial that takes a kernel, generates the same IR used in DRF analysis, and converts it back into a form comprehensible by static analyzers (CUDA code). In essence, the conversion of a protocol to a CUDA kernel.

Contributions. This paper includes the following contributions.

- In Section 4, we establish the syntax and instruction mapping conventions of our code generation tool, Proto-to-Cuda. Our approach ensures that the behavioral abstraction provided by MAPs is not lost in our CUDA representation, but still allows static analyzers such as Faial to parse the kernel.
- In Section 5, we present our stress testing methodology and the initial results of fixed point analysis with Faial. Throughout the tests, we discovered several bugs related to missing instructions and multi-dimensional arrays.
- In Section 6, we improved our code generation infrastructure via analysis of the errors in the dataset. We also detected bugs related to local variables by analyzing parsing errors that traced back to earlier IRs used by Faial.

2 Background

In the first stage of Faial's analysis, we take a CUDA kernel and infer an intermediate representation called a memory access protocol (MAP). MAPs are a family of behavioral types that codify the way threads interact over shared memory [2]. Protocols capture where data is read from/written to shared arrays, which control-flow statements enclose array accesses, and (once well-formed) are able to distinguish between synchronized and unsynchronized fragments of code. However, protocols abstract away non-essential information to streamline the DRF analysis, such as array contents and extraneous computations.

We define the syntax of MAPs below in Figure 1, and describe it here as follows. i is a metavariable ranging over the set of natural numbers, $\mathbb{N}$. An arithmetic expression n is either a numeric variable x, a natural number i, or a binary operation on natural numbers that yields a natural. A Boolean expression b is either a boolean literal, an arithmetic comparison $\diamond$, or a propositional logic connective $\circ$. A protocol p may either do nothing (skip); specify a synchronization point (sync); access the shared memory location $o[n]$, either as a read (rd) or write (wr); perform sequential composition; enter a conditional; or loop. Note that our language does not define if-statements without else clauses; all conditionals must have an else, even if it does nothing.

$$
\begin{array}{rcl}
\mathbb{N} \ni i & ::= & 0 \mid 1 \mid \cdots \\
n & ::= & x \mid i \mid n \star n \\
o & ::= & \mathsf{wr} \mid \mathsf{rd} \\
\mathcal{B} \ni b & ::= & \texttt{true} \mid \texttt{false} \mid n \diamond n \mid b \circ b \\
\mathcal{P} \ni p & ::= & \mathsf{skip} \mid \mathsf{sync} \mid o[n] \mid p \,;\, p \mid \mathsf{if}\ b\ \{p\}\ \mathsf{else}\ \{p\} \mid \mathsf{for}\ x \in n..m\ \{p\}
\end{array}
$$

Fig. 1. Syntax of memory access protocols.

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

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

Fig. 2. CUDA Kernel (left) and MAPs (right).

We illustrate in Figure 2, the level of abstraction required to produce a protocol from a CUDA kernel. In this example, we have a program consisting of a read and write to an array within a for-loop, where the array accesses are enclosed in a conditional. As is standard in CUDA code, tid represents a thread-local identifier variable. This kernel is racy because every other thread performs a read and write access to the same array index concurrently. That is, even threads access the array `A` at index `x` with each loop iteration, and odd threads do nothing. The protocol (right) denotes this concisely by eliding all aspects of the kernel but the reads, writes, and control-flows.

Following the inference stage, Faial checks the protocol for well-formedness and transforms it (across three steps) into formulas that can be verified by an SMT solver [2]. This is how Faial enforces DRF in kernels.

3 Using MAPs to Verify Faial

We want to use Faial's own abstraction to verify whether its inference is correct. While the theory behind MAPs is sound [5], one issue with the implementation is that not all protocols are inferred correctly. Some kernels may be misinterpreted, or have elements that are not resolved during the inference. For instance, Faial may produce a protocol with fewer reads/writes than the original kernel. This could lead to potentially erroneous SMT solver results, hindering the integrity of the analysis. To remedy this, we introduce a new approach to stress testing Faial's inference, involving code generation and fixed point analysis.

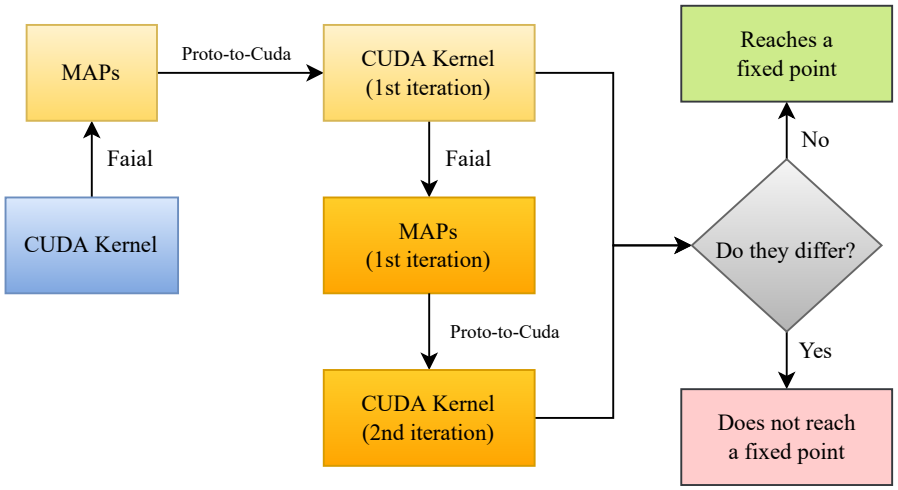

Fig. 3. A flow chart of the stress testing pipeline. We start from the original kernel (blue), obtain our first IR (MAPs/1st iteration kernel), use it to obtain our second IR (1st iteration MAPs/2nd iteration kernel), and compare each kernel representation.

First, we take a CUDA kernel and infer MAPs from it, just as is done in DRF analysis. Then, instead of proceeding with the well-formedness check (and subsequent transformations), we take the protocol as is and convert it back into CUDA code. This functionality, which we refer to as Proto-to-Cuda, will provide Faial with a parseable representation of the inferred protocol. We then take the newly generated kernel, infer yet another protocol from it, and convert said protocol into a CUDA kernel. Finally, we compare the two generated kernels to determine whether they have any differences.

We say a kernel reaches a fixed point when there are no differences between the first and second iteration of the kernel. This signifies that the inference is consistent; it produces the same MAPs for the same type of kernel regardless of how many times it is run. Conversely, kernels that do not immediately reach a fixed point highlight some inconsistency or error in the inference.

Our primary goal is to categorize the classes of differences that exist between non-fixed kernels, as they will shed light on how accurate our inference is and how to fix any inconsistencies. We believe that repeatedly stress testing Faial while fixing bugs that are identified in the tests will improve the accuracy of both the inference and the overall DRF analysis in the long run.

4 Proto-to-Cuda: Representing the Abstract Concretely

In this section, we give a brief overview of our code generation approach. Our code generation tool, Proto-to-Cuda, implements the approach.

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 ∈ 0..N {...}`	`for (int x = 0; x < N; x += 1) {...}`

Table 1. Instruction mapping between protocols and Proto-to-Cuda kernels.

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

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

Fig. 4. CUDA kernel (left), Proto-to-Cuda kernel (right).

Similar to MAPs, there are four types of instructions we must generate a CUDA equivalent for: array accesses, barrier synchronizations, conditionals, and loops. The mapping for each instruction is given by Table 1, where the protocol-based instructions are shown in the middle and the CUDA translations are given on the right. We note that the protocol language implemented by Faial differs slightly from the syntax shown in Figure 1; reads (rd) and writes (wr) are also accompanied by the name of the array, *e.g.*, `rd A[x]` and `wr A[x]`.

For control-flow statements, translation from protocols to CUDA code is a simple syntax transformation. However, array accesses require additional work to translate. Since protocols only retain information on *where* data is being accessed, we need to simulate *what* is being written to arrays and how data is read from them. We implement this through a convention. For each array in the protocol, we declare an external write function outside of the kernel and a local variable inside of the kernel. When an array is read from, its corresponding local variable is assigned the value of the array access. When an array is written to, it is assigned the return value of its write function at the access index.

Figure 4 introduces a minimal example of a kernel with a read and a write to two different arrays (left). After inferring a protocol from the kernel on the left, Proto-to-Cuda generates the kernel on the right via the mapping shown in Table 1. Supplementary dummy read variables and write prototypes are included to make the kernel parseable. The `extern` modifier is used on write prototypes because we do not implement these functions, and `__device__` is used to make the function callable from the GPU.

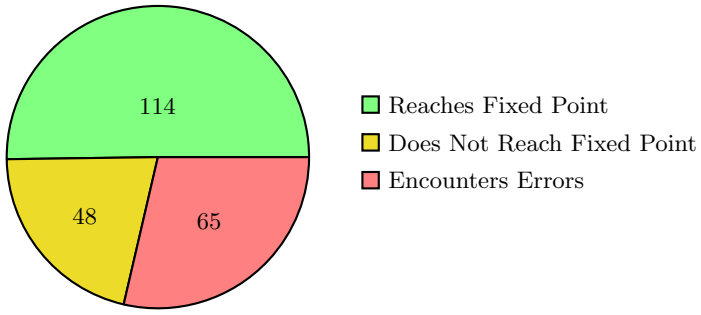

Fig. 5. Initial stress test results.

5 Fixed Point Analysis

In this section, we discuss our testing methodology, the results from stress testing Faial, and what bugs we identified from the tests.

To facilitate our analysis, we will be using the CAV-21 dataset—a collection of 227 real-world CUDA kernels—to stress test Faial. For each kernel in the dataset, we programmatically generate two consecutive kernels and compare them as described in Section 3. If the first and second kernel iterations are the same, we mark the original kernel as a fixed point kernel. If the kernel iterations differ, we keep track of the specific lines that change from the first iteration to the second iteration and mark the original kernel as a non-fixed kernel. However, if the original kernel was unable to reach a fixed point due to an error parsing one of the generated kernels, we denote it as an error kernel and keep track of the errors produced at runtime.

Initially, we found that 50% of kernels reached a fixed point right away, 21% did not reach a fixed point, and the remaining 29% could not be parsed. Since we are primarily interested in what causes discrepancies in our inference, we first investigated the non-fixed kernels. Specifically, looking for patterns between the differences that may elucidate us to the source of a bug. This unveiled roughly three types of differences that could occur: missing instructions, expression conversions, and structural changes to variables.

A missing instruction occurs when a read, write, loop, or conditional is present in the first kernel iteration, but absent in the second. Out of all instructions, reads and writes had a higher propensity of disappearing than control-flow statements, but when control-flows did disappear, so would all of the instructions within their scope. This is the type of difference we want to fix, as missing instructions can significantly change DRF results.

On the other hand, some differences between kernels do not require fixes, such as expression conversions. For example, in Figure 6, we have a line-by-line comparison of two kernel iterations with a differing conditional. The first line (-) represents the first iteration and the second line (+) represents the second iteration. Both lines are derived from the same conditional, but the second iter-

```
- if (tid & ((size / 2) - 1) >= size / 2) {
+ if (tid & ((size / 2) - 1 >= size / 2 ? 1 : 0) != 0) {
```

Fig. 6. Conditional expansion between kernels.

ation expands the conditional without changing the result. Since this is only a difference in semantics and does not impact DRF analysis results, we can safely ignore it. The same logic applies to other expression-level transformations, such as when unary negation, `-x`, is converted to binary subtraction, `0 - x`.

In terms of identifying bugs in Faial, some of the most helpful differences we found were the structural changes to arrays. These differences can range from anything as simple as the data types of arrays being changed to `int` to multi-dimensional arrays being removed from kernels. One key difference we found in particular was that multi-dimensional shared arrays had their access indices swapped during reads and writes. To confirm that the issue was global to both our code generation infrastructure and our base inference, we took a non-fixed kernel from the dataset that experienced array index swapping. Then, we compared the protocol representation of the kernel with our first iteration CUDA kernel. Since they both experienced index swapping, we sourced the bug to our shared serialization method and resolved the issue.

6 Reducing Errors in Kernels

A consistent challenge we faced while stress testing Faial was the sheer number of error kernels that polluted the CAV-21 dataset. Although fixing bugs related to differences helped increase the number of kernels that reach a fixed point, it did not reduce the number of error kernels. However, we want to expand the scope of the experiment to target a wider range of kernels and fix potential bugs related to parsing errors. In this section, we discuss some of the steps taken to reduce the number of errors.

One of the main types of parsing errors we encountered early on was caused by user-defined data types. Because we generate kernels without reintroducing type declarations, we end up with many first iteration kernels where a user-defined type is used, but not declared. As a result, re-parsing the first iteration kernel results in an unknown type error. To fix this, we simply add empty class declarations for each user-defined type in the kernel. These declarations allow us to analyze kernels with user-defined types without the need to change each variable's type to `int`, modify the original kernel, or completely disable type checking beforehand.

Another prominent parsing error in the dataset was the use of undeclared identifiers. This type of error can occur if a local variable from the original kernel is used in the generated kernel, but not declared. By design, the protocol language does not specify local variables outside of for-loops; when we infer MAPs

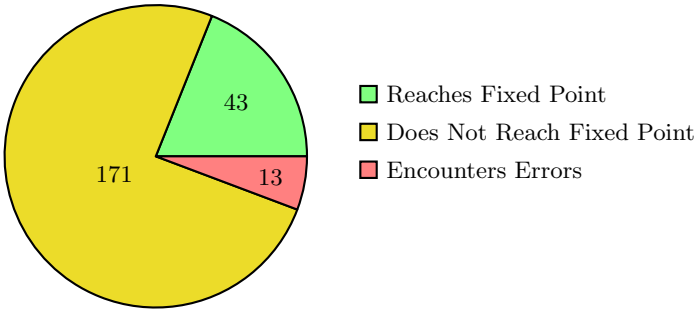

Fig. 7. Final stress test results.

from a kernel, all non-loop local variables are inlined in expressions. Hence, to analyze the source of this error, we took a specific error kernel with undeclared identifiers and looked at stages of Faial where local variable declarations are used. In doing so, we discovered a lexical scoping issue within Dlang, one of Faial's pre-inference IRs. Once the bug in Dlang was fixed, we verified that undeclared local variables no longer appeared via fixed point analysis.

Lastly, we fixed parsing errors caused by unknowns. Similar to the issue of undeclared identifiers, unknowns are introduced in first iteration kernels and cause parsing errors in second iteration kernels since they are not declared. However, unlike undeclared identifiers caused by local variables, unknowns are symbols generated by Faial, not the original kernel. For example, Faial uses for-loops with an unknown upper bound to approximate while-loops [2] and replaces data-dependent array accesses with reads/writes to unknown array indices [5]. To facilitate fixed point analysis for kernels with unknowns, we declare a local variable for each unknown in the kernel and initialize them with a dummy write function, similar to how arrays are written to.

By the end of the experiment, we reduced the 65 initial error kernels down to just 13 error kernels, as shown in Figure 7. The remaining parsing errors are mainly caused by language features that Faial does not support, such as loops with multiple variables and `__shared__` variables that are not arrays.

Additionally, we eliminated all non-semantic differences from the dataset. The remaining 173 non-fixed kernels only differ in ways that do not affect DRF analysis results, such as expression conversions. The main reason for the increase in non-fixed kernels from Figure 5 is the reintroduction of local variables for unknowns. Our code generation tool "names" unknowns in the first iteration (*e.g.*, `__unk01` and `__unk02`), and is then required to create new names for unknowns in the second iteration (*e.g.*, `__unk03` and `__unk04`). However, since this is only a semantic difference, we still consider our final results a significant improvement from the initial results.

7 Related Work

Static verification tools for GPUs. These include DRF analysis tools such as [1,2,4] and performance analyzers like [6]. We note that PUG and RaCUDA can benefit especially from abstract program representation since both verifiers support fewer language features than either Faial or GPUVerify.

Provable data-races in GPU kernels. Liew *et al.* [5] extend the theory behind the Faial tool-chain to characterize a class of kernels that can be analyzed soundly and completely. Similar to this paper, they discuss MAPs and abstract kernel representation via behavioral types.

8 Conclusion

We tackle the problem of verifying static analysis tools via abstract program representation and code generation. Through Proto-to-Cuda, this paper extends the utility of Faial's behavioral type to allow for static analysis without sacrificing important elements of a kernel. Our stress tests demonstrate that it is possible to leverage abstract kernel representation to detect bugs.

References

1. Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. Engineering a static verification tool for GPU kernels. In *Proceedings of CAV*, volume 8559, pages 226–242, Berlin, Heidelberg, 2014. Springer.
2. Tiago Cogumbreiro, Julien Lange, Dennis Liew, and Hannah Zicarelli. Checking data-race freedom of GPU kernels, compositionally. In *Proceedings of CAV*, pages 403–426. Springer, 2021.
3. Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, May 2013. ISSN: 1558-1225.
4. Guodong Li and Ganesh Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of FSE*, pages 187–196, New York, NY, USA, 2010. ACM.
5. Dennis Liew, Tiago Cogumbreiro, and Julien Lange. Provable GPU Data-Races in Static Race Detection. *Electronic Proceedings in Theoretical Computer Science*, 356:36–45, March 2022.
6. Stefan K. Muller and Jan Hoffmann. Modeling and analyzing evaluation cost of CUDA kernels. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–31, January 2021.
7. Guillem Pratx and Lei Xing. GPU computing in medical physics: A review: GPU computing in medical physics. *Medical Physics*, 38(5):2685–2697, May 2011.
8. Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 41–53, Vienna Austria, February 2018. ACM.