Securing GPU Kernels with Lightweight Formal Methods

Jiacheng Zhao Baojian Hua*
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
jiachengz@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—This paper reports our initial experience of applying lightweight formal method to validate the security of CUDA, the main programming language used for GPGPU programming. By "lightweight formal method", we mean a pragmatic approach to verifying the security of CUDA programs coded by programmers. We do not aim to achieve full formal verification, but instead emphasize automation, usability, and the ability to continually ensure security as program evolve over time. Our approach first analyzes the program annotated with specification and loop invariants, and generates the corresponding verification conditions, which can be verified to ensure the security that is essential to GPU programs. Our work shows promising potentials by successfully verifying representative CUDA programs, including matrix multiplication optimized by using shared memory. Compared with some existing tools, our method exhibits better performance in CUDA programs security verification of without excessively increasing the programmer's usage cost.**

***Index Terms*—CUDA, security, lightweight formal methods**

I. Introduction

General-purpose graphics processing units (GPGPUs), are rapidly emerging as a core technology for solving large-scale computational problems, thanks to their efficient parallel processing capabilities. To fully leverage the parallel computing capabilities of GPGPUs, specialized programming models are essential. Specifically, NVIDIA's CUDA platform [1] and programming model streamline the process of migrating complex computational tasks to GPGPUs, significantly enhancing program execution efficiency [2].

While CUDA has shown remarkable flexibility and performance benefits in GPGPU programming, it also presents programmers with a series of challenges. First, as a low-level imperative systems programming language, CUDA requires developers to precisely control the memory access patterns of each thread and the timing of instruction execution. They must deeply understand hardware architecture and memory hierarchy to ensure program correctness and efficiency [3]. Second, manually managing data transfer between GPGPU and CPU complicates memory management, leading to issues like memory leaks and data inconsistency. Third, thread synchronization represents another major challenge in CUDA programming [4]. With thousands of threads executing simultaneously in GPGPU's parallel computing model, incorrect synchronization can cause data races, deadlocks, and other problems.

This paper presents our initial efforts in applying lightweight formal methods to verify the security of CUDA programs. Our goal is to enhance program security and reduce errors and omissions in manual inspection processes through efficient lightweight formal methods, while seamlessly integrating with existing CUDA codebases and development environments without significantly increasing the burden on CUDA programmers. In return for being lightweight and easy to apply, we do not pursue complete formal verification of CUDA programs, but only provide security guarantees that programmers care about.

We settle on an approach with three elements. First, we provide additional verification information for CUDA programs through annotations. These annotations include formal propositions including preconditions, postconditions, and loop invariants. Second, we design a compiler extension that automatically extracts verification conditions from the annotated program and integrating them with the core logic of the CUDA program to generate corresponding verification conditions. Third, we employ a specialized verification backends to perform security verification on the extracted verification conditions.

We implement our approach as a lightweight formal verification framework for CUDA programs. We conducted a systematic evaluation of this framework, in terms of its effectiveness, performance, and usefulness on both micro and real-world benchmarks. The experimental results demonstrate that our approach can accurately perform security checks and successfully detect vulnerabilities across various categories than state-of-the-art GPUVerify [5]. Furthermore, our approach outperforms GPUVerify in terms of time efficiency, especially for larger CUDA programs.

To summarize, our work makes the following contributions:

- **Infrastructure design.** We propose a new approach to secure GPU kernels with lightweight formal methods.
- **Prototype implementation.** We implemente a software prototype to validate our approach.
- **Extensive evaluation.** We conduct extensive experiments to evaluate the effectiveness and performance of our approach on both micro benchmarks and real-world projects.

Outline. The rest of this paper is organized as follows. Section II presents our approach. Section III presents our evalution results. Section IV presents the related work, and Section V concludes.

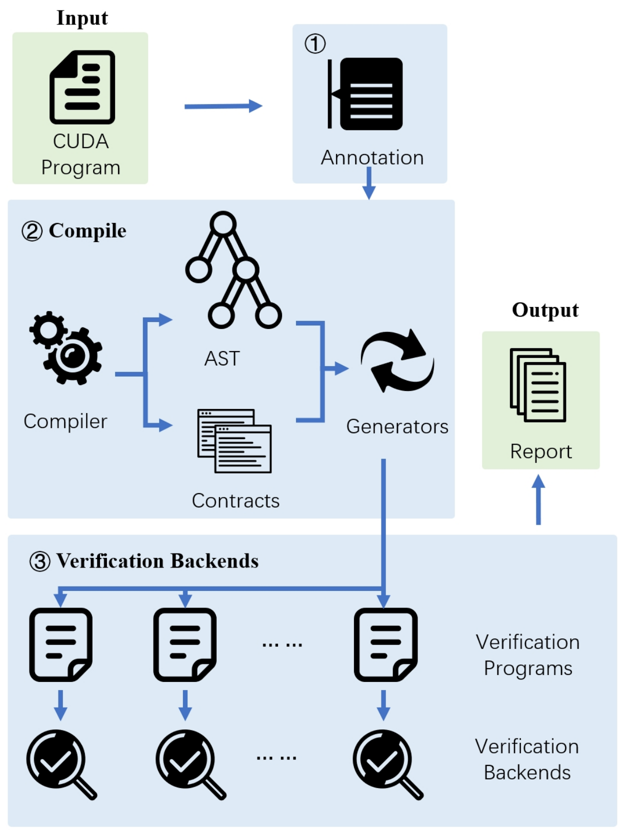

Fig. 1: The architecture of our approach.

II. Approach

In this section, we present our approach. We first discuss the architecture of our approach (§ II-A), then present each component including the annotation (§ II-B), the compiler (§ II-C), verification backends (§ II-D), and prototype implementation (§ II-E), respectively.

A. Architecture

The overall architecture of our approach is presented in Fig. 1, consisting of three main components: ① annotation, ② the compiler, and ③ verification backends.

First, programmers can add additional annotations such as preconditions, postconditions, and loop invariants in the form of comments to CUDA programs, which are similar to those based on Hoare logic. However, as we do not aim to perform complete formal verification of CUDA programs. Therefore, programmers only need to incrementally add necessary annotations to relevant code snippets.

Second, the compiler analyzes the CUDA program with annotations, parses it into an abstract syntax tree, and extracts the verification conditions from the tree. Based on the generated AST and annotations, the generator produce and emit verification target programs for the verification backends.

Third, the verification backends verifies the generated verification target programs and produce corresponding reports to the end developers to diagnose.

$$
\begin{array}{lll}
RequiresSpec & \rightarrow & \textbf{requires}\ s \\
EnsuresSpec & \rightarrow & \textbf{ensures}\ s \\
InvariantSpec & \rightarrow & \textbf{invariant}\ s \\
AssertionSpec & \rightarrow & \textbf{assert}\ s \\
s & \rightarrow & e \\
 & \rightarrow & \textbf{forall int}\ x:\ e \leq x < e.e \\
e & \rightarrow & e\ op\ e \mid e[e] \mid !e \mid old(e) \mid x \mid n
\end{array}
$$

Fig. 2: Syntax of the annotation language.

B. Annotations

Given the complexity of CUDA programs, performing the desired verification directly on original CUDA programs is challenging without any additional information. To address this challenge, we propose an annotation language to annotate CUDA programs. This annotation language is embedded into the CUDA programs as annotations, enabling the generation of the verification target program without introducing any runtime overhead to the original CUDA code execution.

Fig. 2 presents the syntax for these annotations. These annotations must begin with `//@` or `/*@` that encompass several syntactic forms, each serving a distinct purpose in the verification process: 1) *RequiresSpec* specifies the conditions that hold before the execution of a function. They preceds the function definition, establishing the initial state requirements. 2) *EnsuresSpec* defines the expected states or results after the function execution. They are also placed before the function, indicating the outcomes that the function guarantees to produce. 3) *InvariantSpec* are conditions that must remain true throughout the execution of a loop. They are declared before the loop construct, ensuring that the target loop maintains its logical integrity throughout its iterations. 4) *AssertionSpec* are sanity checks that must hold at specific points within the program. They can be placed anywhere in the code, acting as runtime checks to ensure that the program states satisfy desired conditions.

Fig. 3 provides a use case example. At the beginning of the Kernel function, the `requires` statement is used to ensure that the number of threads when the Kernel is invoked will not exceed the bounds of the array. The `ensures` statement is used to verify that the function achieves the desired effect, which is to reverse the array. On line 5, the `assert` statement is used to ensure that the array index does not overlap, preventing potential data races.

C. Compiler

The compilation phase analyzes CUDA programs with annotations to generate several corresponding verification programs, comprising two sub-steps.

First, the compiler parses the CUDA source code and constructs to an AST, which encodes the structural information of the CUDA source code for subsequent analysis. The compiler then extracts the verification contracts written by the programmer from the AST, record their positions in the source

```
/*@ requires blockDim.x * gridDim.x <= array.Length;
    ensures forall int i; 0 <= i < A.Length
    . array[i] == old(array[array.Length - i]); */
 __global__ void rev_per_block(double *array){
  double *block_part = &array[blockIdx.x * blockDim.x];
  //@ assert threadIdx.x != blockDim.x -1 - threadIdx.x;
  block_part[threadIdx.x] = block_part[blockDim.x -1 -
       threadIdx.x];
 }
```

Fig. 3: A sample CUDA program with annotations.

code, and record this information to facilitate verification and analysis in subsequent steps.

Second, the compiler generates target verification programs. Specifically, the compiler will only analyze the AST nodes required for the verification. Hence, these verification programs are not faithful translations of the CUDA source code but are specifically designed to verify particular contract properties. In this process, different verification programs are generated for different contracts. For example, if an annotation checks memory access, the compiler produces a set of programs to simulate and verify memory access behavior.

D. *Verification Backends*

The verification backend validates required properties. Instead of dealing with the entire complexity of the program, it focuses on the parts directly related to the verification objectives.

Ultimately, the verification backends generate a detailed verification report, which includes the verification results and problem localization. With these reports, developers can understand potential issues in the program and make necessary fixes. This lightweight verification approach not only reduces the complexity of verification but also significantly improves its efficiency.

E. *Implementation*

To validate our design, we have developed a software prototype for our approach. In this prototype, we utilize Clang [6] to parse CUDA programs and construct an Abstract Syntax Tree (AST). We modify Clang to extract contracts, ensuring that specifications and safety properties within the program can be effectively captured. We leverage Dafny [7] as the unified verification backend. Dafny is a programming language specifically designed for formal verification, integrating powerful features such as program logic, automatic reasoning, and program verification, which allows for efficient validation of program correctness. Compared to traditional SMT solvers like Z3, Dafny offers a higher level of abstraction.

III. Evaluation

By presenting the experimental results, we mainly investigate the effectiveness and efficiency.

All experiments and measurements are performed on a server with one 8 physical Intel i7 core CPU and 16 GB of RAM running Ubuntu 24.04.

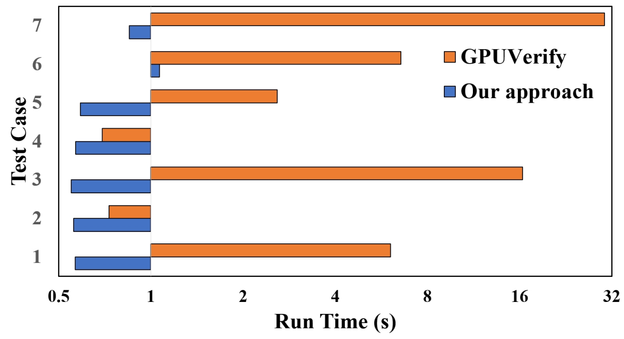

Fig. 4: Runtime of our approach and GPUVerify.

We construct a micro-benchmark consisting of 10 test cases. All these test cases were derived from existing studies [8]. To assess the capabilities of our approach, we modify each test case to include diverse vulnerabilities, such as data races, API misuse, and improper thread organization.

A. *RQ1: Effectiveness.*

To answer RQ1, we applied our approach and GPUVerify [5] to micro-benchmarks to assess our approach's effectiveness.

TABLE I shows the experimental results. For all test cases, our approach accurately perform safety checks, successfully covering vulnerabilities in multiple aspects such as data races, API misuse, and improper thread organization. In contrast, GPUVerify only detects data races vulnerabilities. This result indicates that our approach is more comprehensive in checking CUDA programs.

B. *RQ2: Efficiency.*

To answer RQ2, we measure the runtime and compared it with GPUVerify. Since GPUVerify's detection scope is limited to data race vulnerabilities, we only consider test cases that contained data race vulnerabilitiesin this experiment. The experimental results are shown in Fig. 4, where our approach outperforms GPUVerify in terms of runtime efficiency, especially in more complex test cases.

Through manual analysis, we confirm that the root cause for this performance discrepancy is the verification methods of the two tools. GPUVerify employs complete formal verification, which significantly increases the verification time when dealing with programs containing complex structures. In contrast, our approach utilizes lightweight formal verification method to generate simple verification tasks, thereby demonstrating higher efficiency.

IV. Related Work

Current CUDA program bug detection technologies mainly focus on two core issues: memory safety [9]–[11] and data races [12], [13]. CURD [14] uses static analysis to select appropriate race detection algorithms. cuCatch [4] combines optimized compiler instrumentation and driver support to identify memory safety errors.

TABLE I: Experimental results on micro-benchmarks.

#	Test Case	Ground Truth	Our	GPUVerify	Our Run Time (s)	GPUVerify Run Time (s)
1	_stereoDisparity.cu	bug free	✓	✓	0.6342	6.0667
		data race	✓	✓	0.5659	6.0614
2	addKernel.cu	bug free	✓	✓	0.5994	0.7349
		data race	✓	✓	0.5594	0.7302
3	binomialOptions.cu	bug free	✓	✓	0.5828	16.9050
		data race	✓	✓	0.5486	16.3429
4	computeVisibilities.cu	bug free	✓	✓	0.6000	0.7103
		data race	✓	✓	0.5668	0.6940
5	convolutionColumnsKernel.cu	bug free	✓	✓	0.6105	0.6502
		data race	✓	✓	0.5877	2.5833
6	dwtHaar1D.cu	bug free	✓	✓	0.9615	6.6771
		data race	✓	✓	1.0650	6.5591
7	finiteDifferencesKernel.cu	bug free	✓	✓	1.1326	283.5116
		data race	✓	✓	0.8506	30.2891
8	simpleIPC.cu	bug free	✓	✓	0.5670	0.6062
		api misuse	✓	✗	0.5572	-
9	simpleMPI.cu	bug free	✓	✓	0.5677	0.6713
		api misuse	✓	✗	0.5627	-
10	simpleMultiGPU.cu	bug free	✓	✓	0.6086	0.8348
		error launch	✓	✗	0.6088	-

GPUVerify [5] uses a Synchronous, Delayed Visibility (SDV) - based method to verify GPU kernels for data races and barrier divergence. ESBMC-GPU [15] employs SMT - based bounded model checking to detect errors and concurrency issues in CUDA programs. Vericuda [16] automates CUDA program verification by adding Hoare triples under data race freedom assumptions. Faial [17] uses a combined analysis based on memory access protocols to detect data race freedom violations in CUDA programs.

Descend [18] is a safe language designed specifically for GPU programming, drawing inspiration from the Rust [19] language in terms of design philosophy.

V. Conclusion

In this work, we propose an approach to enhance the safety of GPU kernels through lightweight formal verification. Our approach allows programmers to focus on key security aspects while writing code. A compiler automatically analyzes the program, generating verification tasks to achieve lightweight formal verification. Our experimental results show that our approach can efficiently detect severe vulnerabilities automatically, demonstrating feasibility and practicality of the lightweight formal verification methods in checking GPU kernels.

References

[1] NVIDIA, "Cuda toolkit," https://developer.nvidia.com/cuda-toolkit.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *Journal of parallel and distributed computing*, vol. 68, no. 10, pp. 1370–1380, 2008.

[3] M. A. Al-Mouhamed, A. H. Khan, and N. Mohammad, "A review of cuda optimization techniques and tools for structured grid computing," *Computing*, vol. 102, no. 4, pp. 977–1003, 2020.

[4] M. Tarek Ibn Ziad, S. Damani, A. Jaleel, S. W. Keckler, and M. Stephenson, "Cucatch: A debugging tool for efficiently catching memory safety violations in cuda applications," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 124–147, 2023.

[5] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: a verifier for gpu kernels," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 113–132.

[6] "Clang: a c language family frontend for llvm," https://clang.llvm.org.

[7] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International conference on logic for programming artificial intelligence and reasoning*. Springer, 2010, pp. 348–370.

[8] "Gpuverifybenchmarks," https://github.com/mc-imperial/GPUVerifyBenchmarks.

[9] Y. Zhao, W. Xue, W. Chen, W. Qiang, D. Zou, and H. Jin, "Owl: differential-based side-channel leakage detection for cuda applications," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2024, pp. 362–376.

[10] J. Lee, Y. Kim, J. Cao, E. Kim, J. Lee, and H. Kim, "Securing gpu via region-based bounds checking," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 27–41.

[11] F. F. dos Santos, S. Malde, C. Cazzaniga, C. Frost, L. Carro, and P. Rech, "Experimental findings on the sources of detected unrecoverable errors in gpus," *IEEE Transactions on Nuclear Science*, vol. 69, no. 3, pp. 436–443, 2022.

[12] A. K. Kamath and A. Basu, "Iguard: In-gpu advanced race detection," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 49–65.

[13] Y. Liu, A. VanAusdal, and M. Burtscher, "Performance impact of removing data races from gpu graph analytics programs," in *2024 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2024, pp. 320–331.

[14] Y. Peng, V. Grover, and J. Devietti, "Curd: a dynamic cuda race detector," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 390–403, 2018.

[15] P. Pereira, H. Albuquerque, H. Marques, I. Silva, C. Carvalho, L. Cordeiro, V. Santos, and R. Ferreira, "Verifying cuda programs using smt-based context-bounded model checking," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 1648–1653.

[16] K. Kojima, A. Imanishi, and A. Igarashi, "Automated verification of functional correctness of race-free gpu programs," *Journal of Automated Reasoning*, vol. 60, pp. 279–298, 2018.

[17] T. Cogumbreiro, J. Lange, D. L. Z. Rong, and H. Zicarelli, "Checking data-race freedom of gpu kernels, compositionally," in *International Conference on Computer Aided Verification*. Springer, 2021, pp. 403–426.

[18] B. Köpcke, S. Gorlatch, and M. Steuwer, "Descend: A safe gpu systems programming language," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 841–864, 2024.

[19] C. N. Steve Klabnik and C. Krycho, "The rust programming language," https://doc.rust-lang.org/stable/book/, 2025.