SHARD: Securing GPU Kernels with Lightweight Formal Methods

Jiacheng Zhao, and 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
*Corresponding author

Abstract—GPGPU is essential for large-scale computing, with CUDA as its dominant programming model. However, CUDA inherits unsafe features from C/C++ and introduces complexity through multi-level memory management and high concurrency. Developers must coordinate thousands of threads and their access to shared memory spaces, which inevitably introduces security vulnerabilities. Thus, enhancing the security of CUDA programming remains a critical challenge.

In this paper, we propose a lightweight formal method to validate the security of CUDA programs. A lightweight formal method emphasizes automation and usability over full formal verification, enabling continuous assurance as code evolves. Our approach enables developers to embed concise annotations including preconditions, postconditions, loop invariants, and symbolic state-machine specifications, directly into CUDA source code. We design a compiler extension to performs static analysis and backward taint tracking, generating verification conditions. We leverage different Hoare-logic-based backends including Dafny and Verus to discharge verification conditions to check functional and concurrency correctness. Our approach demonstrates promising potentials by successfully verifying representative CUDA programs, including shared-memory-optimized matrix multiplication. Compared to existing tools, our approach achieves superior performance in security verification while minimizing additional efforts from developers.

Keywords–GPU programming; Security; Lightweight formal methods;

1. INTRODUCTION

General-purpose graphics processing units (GPGPUs) have become a key technology for fulfilling large-scale computational requirements. Owing to their massive parallelism and high performance-to-power ratio advantages, GPGPUs are widely used in security-critical domains such as artificial intelligence, autonomous driving, and healthcare. To fully exploit GPGPU capabilities, specialized programming models are essential. NVIDIA's CUDA platform [1] offers a well-established framework for general-purpose GPU programming, enabling efficient offloading of compute-intensive workloads. It simplifies GPGPU development through high-level language support and optimized libraries such as cuBLAS [13] and cuFFT [14], significantly enhancing programming productivity.

Despite its benefits, CUDA programming remains challenging. As a low-level imperative language, CUDA requires fine-grained control over many detailed thread behavior including memory accesses, and synchronization. Developers must understand GPU architecture and memory hierarchies to write correct and performant code [3]. On one hand, manual host-device memory management introduces data inconsistencies and memory leaks. On the other hand, improper thread coordination leads to data races and deadlocks [4]. To this end, safe CUDA programming remains a demanding task even for experienced developers.

To mitigate these difficulties, researchers have conducted studies in three directions. First, safer programming languages such as Descend [5] aim to eliminate unsafe CUDA constructs through novel language design. However, despite their potentials, these new languages deviate from CUDA syntax and semantics, thus hinders backward compatibility and incurs significant migration costs. Second, dynamic bug detection tools such CURD [6] and cuCatch [4] instrument CUDA programs to monitor runtime behavior. While effective, this approach introduces non-negligible runtime overhead, especially in large-scale parallel settings. Third, formal verification tools such as GPUVerify [7] and Faial [8] offer rigorous correctness guarantees by verifying entire programs. However, they struggle with scalability due to state space explosion, limiting their practicality for real-world applications.

Based on our insights, we believe that security enhancements to GPU kernels should meet the following three goals:

- (G1) **Compatibility.** The GPU programming model has evolved into a mature and stable ecosystem. We advocate maintaining compatibility to reduce the developer learning curve and avoid the efforts to migrate existing projects and libraries, significantly saving both time and resources.
- (G2) **Static analysis.** Due to the high concurrency and compute-intensive nature of GPU kernels, runtime safety checks are often too costly. Static analysis provides a practical alternative to improve security without incurring runtime overhead.
- (G3) **Lightweight.** We seek a minimally intrusive, efficient, and extensible approach that integrates smoothly into development workflows and applies broadly, improving program safety without sacrificing developer productivity.

In this paper, we present SHARD, an approach that secures GPU kernels using lightweight formal methods. By "lightweight", we mean a formal method emphasizes automation and usability over full formal verification, enabling continuous assurance as code evolves. Specifically, we aim to achieve the above three research goals. For the first goal

(G1), we introduce a set of annotation specifications that allow developers to embed safety constraints and expected behaviors directly into the code. These annotations are added atop existing code without affecting the toolchain, ensuring compatibility with current development environments.

For the second goal, we leverage a compiler extension to statically analyze the annotated GPU programs and check its properties. Our core idea is to extract separate verification conditions from the annotations, enabling safety checks without impacting the original execution logic. With this idea, we parse the code into an intermediate representation and apply backward taint analysis to identify key paths and generate verification tasks.

For the third goal, we provide an automated, lightweight verification pipeline that checks these specifications. Our pipeline supports progressive verification: Hoare logic is first used to reason local correctness, while symbolic state machines are used to guarantee concurrency correctness. By following the "pay-as-you-go" principle, developers can choose the appropriate level of assurance, balancing practical usability with verification strength.

We implement our approach as a lightweight formal verification framework for CUDA programs. We conduct systematic evaluations of this framework, in terms of effectiveness, performance, and usefulness on both micro and real-world benchmarks. Our experimental results demonstrate that our approach can accurately perform security checks and successfully detect vulnerabilities across various categories than state-of-the-art GPUVerify [7]. 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 lightweight formal method-based approach to secure GPU kernels.
- **Prototype implementation.** We implement a software prototype to validate our approach.
- **Extensive evaluation.** We conduct extensive experiments to evaluate our approach.

Outline. The rest of this paper is organized as follows. Section 2 introduces the background. Section 3 presents the motivations. Section 4 presents our approach. Section 5 presents our evaluation results. Section 6 discusses limitations and directions for future work. Section 7 presents the related work, and Section 8 concludes.

2. Background

To be self-contained, this section provides essential background on the CUDA programming model (§ 2.1) and formal verification techniques (§ 2.2).

2.1 CUDA Programming Model

CUDA (Compute Unified Device Architecture) [1], developed by NVIDIA, is a parallel computing platform and programming model that enables general-purpose computation on GPUs. It has expanded the role of GPUs beyond graphics rendering to a wide range of security-critical domains, including artificial intelligence, autonomous driving, and healthcare.

CUDA's key advantage lies in its ability to exploit the massive parallelism of modern GPUs, which typically consist of thousands of cores capable of executing numerous tasks concurrently. This makes CUDA particularly effective for workloads such as deep learning, where it can significantly accelerate neural network training.

Despite its benefits, CUDA programming remains a challenging task. Developers must understand GPU architecture, memory hierarchies, and parallel programming principles. Moreover, GPU programming requires careful management of limited memory resources and optimizations of host-device data transfers, which can otherwise become bottlenecks [16].

2.2 Formal Verification

Formal verification [17] uses mathematical methods to rigorously prove that hardware, software, or protocols meet specified correctness properties. It applies tools from logic, algebra, and automated reasoning to check whether a system adheres to its formal specification. As a cornerstone of reliability engineering, Formal Verification is widely adopted in safety-critical domains such as aerospace [18], automotive systems [19], and finance [20].

A key technique in program verification is Hoare logic [21], which uses preconditions, postconditions, and invariants to prove functional correctness. Beyond correctness, formal verification also applies to properties like memory safety, liveness, and data integrity.

However, full formal verification of complex systems remains difficult. Modeling a system formally requires considerable effort, and the verification process is often hindered by program size, recursion, concurrency, and large state spaces. These challenges limit scalability and highlight the need for lightweight, targeted verification approaches in practical software development.

3. Motivation

GPU programs often exhibit recurring insecure code patterns, suggesting that these bugs can be captured and verified through a unified abstraction. Inspired by prior studies [5], we identify four representative categories of unsafe CUDA code: 1) local errors, 2) API misuse, 3) improper thread organization, and 4) concurrency errors.

Local errors. These refer to defects confined within specific code regions, commonly caused by incorrect index computation, out-of-bounds access, or unsynchronized memory updates. Such bugs may pass compilation but trigger runtime anomalies.

In Fig. 1, threads from the same block reverse-write an array segment, but overlapping indices can lead to conflicting writes.

API misuse. CUDA APIs bridge host-device interaction, but misusing them often causes crashes or silent errors. A common example is incorrect arguments to memory copy operations.

```
__global__ void rev_per_block(double *array) {
 double *block_part = &array[blockIdx.x * blockDim.x];
 block_part[threadIdx.x] =
  block_part[blockDim.x - 1 - threadIdx.x];
}
```

Figure 1. Data races caused by overlapping writes.

```
cudaMemcpy(h_vec, d_vec, size, cudaMemcpyHostToDevice);
```

Figure 2. Incorrect argument order in `cudaMemcpy`.

Fig. 2 intends to copy data to the GPU, but reversed arguments cause the host to access device memory illegally, which is not detected by the compiler.

Improper thread organization. Kernel functions rely on correct thread/block configurations. Mismatches between expected and actual thread counts can result in out-of-bounds access.

```
__global__ void vectorAdd(const float* A, const float* B,
    float* C) {
 int index = threadIdx.x;
 C[index] = A[index] + B[index];
}
```

Figure 3. Boundary violation due to excessive threads.

As shown in Fig. 3, if more threads than vector elements are launched, out-of-bound accesses occur.

Concurrency errors. CUDA's massive parallelism exposes programs to data races and synchronization issues. Without safeguards like atomics, shared resources can be corrupted.

```
__global__ void kernelSafe(int* counter, int N) {
 int tid = blockIdx.x * blockDim.x + threadIdx.x;
 if (tid >= N) return;
 (*counter)++;
}
```

Figure 4. Race condition on shared counter.

In Fig. 4, simultaneous increments to `counter` cause race conditions, especially under high thread counts and multiprocessor contention.

4. Approach

We present our approach in this section. We begin by discussing the workflow of our approach (§ 4.1), then present each component including the annotation (§ 4.2), the compiler (§ 4.3), verification backends (§ 4.4), and prototype implementation (§ 4.5), respectively.

4.1 Workflow

The overall workflow of our approach is presented in Fig. 5, consisting of three main components: ① annotation, ② the compiler, and ③ verification backends.

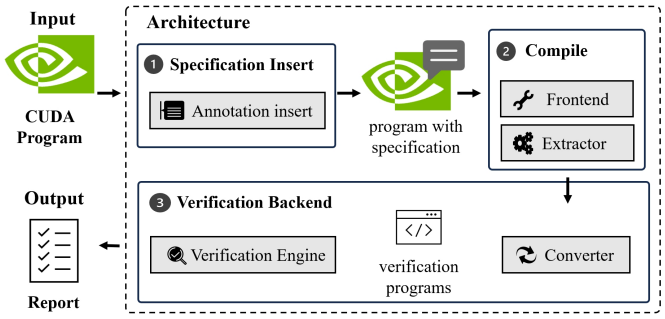


Figure 5. The workflow of our approach.

First, developers incrementally add necessary annotations—such as preconditions, postconditions, and loop invariants—as code comments, following a style similar to Hoare logic. Full formal verification is not required as annotations are only applied to relevant code fragments.

Second, the compiler translates the annotated program into an intermediate representation (IR), from which it extracts verification conditions and generates target programs for the verification backends.

Third, the backends verify these target programs and return diagnostic reports to aid developer analysis.

4.2 Specification Annotation

The specification insertion component provides a lightweight formal interface for embedding verifiable properties directly into the source code. This design enables downstream compilation and verification components to extract precise verification tasks based on these specifications. As shown in Figure 6, the specification language is divided into three parts: ① *HostSpec* for host-side annotations, ② *KernelSpec* for device-side specifications, and ③ *StateMachineSpec* for modeling concurrency via symbolic state machines, with each type using different verification methods accordingly.

Host specification. Host-side annotations serve as markers for verifying API usage and kernel launch configuration. These checks follow fixed patterns, so developers can simply mark relevant API calls and kernel launches to trigger analysis. API compliance is marked through `@check_api()`, which checks properties such as pointer validity, size constraints, and memory direction semantics. For example, `cudaMemcpy` requires non-null pointers and a positive size, while `cudaFree` must avoid double frees. The analysis is guided by predefined rule libraries tailored to each API. For launch configuration, `@check_launch()` enables developers to specify expected dimension constraints, ensuring kernel launches match intended resource configurations while supporting automated validation.

Kernel specification. For device code, we verify its local correctness and key attributes, so we provide multi granularity, hierarchical description specifications based on Hoare logic. KernelSpec has two parts: *FunctionSpec* and *BlockSpec*.

FunctionSpec is at the start of a kernel function. Developers can add preconditions, postconditions, and desired launch

$$
\begin{array}{lcl}
\mathit{AnnotationSpec} & \rightarrow & \mathit{HostSpec} \mid \mathit{KernelSpec} \\
 & & \mid \mathit{StateMachineSpec} \\
\mathit{HostSpec} & \rightarrow & \texttt{check\_api()} \\
 & & \mid \texttt{check\_launch()} \\
\mathit{KernelSpec} & \rightarrow & \mathit{FunctionSpec} \\
 & & \mid \mathit{BlockSpec} \\
\mathit{FunctionSpec} & \rightarrow & \texttt{@requires(}\ e\ \texttt{)} \\
 & & \mid \texttt{@ensures(}\ e\ \texttt{)} \\
 & & \mid \texttt{@config(}\ \mathit{DimConfig}\ \texttt{)} \\
\mathit{BlockSpec} & \rightarrow & \texttt{@invariant(}\ e\ \texttt{)} \\
 & & \mid \texttt{@assert(}\ e\ \texttt{)} \\
\mathit{DimConfig} & \rightarrow & \texttt{blockDim} = \texttt{(}\ e, e, e\ \texttt{)} \\
 & & \mid \texttt{gridDim} = \texttt{(}\ e, e, e\ \texttt{)} \\
e & \rightarrow & \mathit{LogicalExpr} \mid \mathit{ArithExpr} \\
 & & \mid \mathit{ResourceExpr} \mid \mathit{SpecialExpr} \\
\mathit{LogicalExpr} & \rightarrow & e\ \texttt{==}\ e \mid e\ \texttt{\&\&}\ e \mid e\ \texttt{||}\ e \\
 & & \mid \texttt{!}\ e \mid \mathit{Atom} \\
\mathit{ArithExpr} & \rightarrow & \mathit{Atom}\ (\texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/})\ \mathit{Atom} \\
\mathit{ResourceExpr} & \rightarrow & \texttt{forall ID in}\ \mathit{Range}\ \texttt{::}\ e \\
 & & \mid \texttt{exists ID in}\ \mathit{Range}\ \texttt{::}\ e \\
\mathit{SpecialExpr} & \rightarrow & \texttt{old(}\ \mathit{Atom}\ \texttt{)} \mid \texttt{other(}\ e\ \texttt{)} \\
 & & \mid \texttt{other\_in\_block(}\ e\ \texttt{)} \\
\mathit{Atom} & \rightarrow & \texttt{ID} \mid \texttt{INT} \mid \texttt{true} \mid \texttt{false} \\
\mathit{Range} & \rightarrow & e\ \texttt{...}\ e
\end{array}
$$

Figure 6. The core syntax of the annotation specification.

configurations here. `@requires(e)` specifies conditions that must hold before function execution, setting initial-state requirements. `@ensures(e)` defines expected states or results after function execution, offering verification bases. *@config(DimConfig)* specifies desired thread configuration values or ranges for the kernel function. If developers provide this info in the kernel function, it helps check kernel function launches and offers references for local state space verification.

BlockSpec consists of invariants and assertions. `@invariant(e)` denotes an invariant that must stay true throughout a loop. It's declared before a loop construct to ensure the loop's logical integrity during all iterations.`@assert(e)` does sanity checks at specific program points and can be placed anywhere in the code, ensuring program states meet expectations.

Figure 7 provides a use case example. The kernel uses `@requires` and `@ensures` annotations to specify preconditions and postconditions, constraining input parameters and expected results. In line 3, the `@requires` clause ensures that the total number of threads equals `N`, establishing a one-to-one mapping between threads and vector elements. This prevents resource waste and avoids out-of-bounds memory access. In line 4, the `@ensures` clause specifies that each vector element is incremented correctly. Line 7 includes an `@assert` statement, `idx != other(idx)`, to guarantee that each thread has a unique index, preventing write conflicts across threads.

```
1 __global__ void vctor_inc(int* d_out, int* d_in, int N)
2   //@kernel_config(gridDim = 0... , blockDim = 0...)
3   //@requires(gridDim.x * blockDim.x == N)
4   //@ensures(forall i in 0 to N-1 :: d_out[i] = d_in[i]
        + 1 )
5   {
6     int idx = blockIdx.x * blockDim.x + threadIdx.x;
7     //@assert( idx != other(idx) );
8     d_out[idx] = d_in[idx] + 1;
9   }
```

Figure 7. A sample CUDA program with annotations.

State machine specification. *StateMachineSpec* is used to build symbolic state machines, addressing the limitations of KernelSpec's reduction rules in handling concurrency errors in GPU programs. A resource state machine consists of four components: (1) State definitions; (2) State transitions; (3) Invariants; and (4) Proofs of invariants.

Figure 8 demonstrates constructing a symbolic state machine via annotation-based specifications. Consider the kernel function `count2N`, which uses N threads to increment a variable `counter` from 0 to N, each thread performing one increment. The state machine models this process with an initial state where `counter = 0`. Each transition corresponds to a thread incrementing `counter`, until all N increments complete. The transition function thus involves a single action: incrementing `counter`. To simulate atomic increments under concurrency, a token-based approach controls resource usage: `unstamped_tickets` track unused tokens and `stamped_tickets` track consumed tokens. A thread must verify `unstamped_tickets >= 1` before proceeding, consuming one token and incrementing `counter`. This design closely models actual program behavior.

Hence, the state machine includes four fields: `num_threads`, `counter`, `unstamped_tickets`, and `stamped_tickets`. The `initialize` function sets the initial state. The transition `tr_inc` specifies the increment logic, enforcing the precondition `unstamped_tickets >= 1` and asserting `counter < num_threads` to prevent errors. The `finalize` property asserts that once all tokens are consumed, `counter == num_threads`. The invariant `main_inv` holds throughout, ensuring `counter == stamped_tickets` and the sum of tokens remains constant, guaranteeing exactly one increment per thread without concurrency conflicts.

After construction, the state machine model is embedded in the kernel. In Figure 8b, the kernel's `atomicAdd` corresponds to the `tr_inc` transition, ensuring consistency between model and program. The compilation component extracts and analyzes the state machine and kernel, automatically generating verification tasks for the backend.

4.3 Compiler

The core objective of the compilation component is to achieve verification concern separation through three layer abstraction transformation: first, parse GPU programs with specification

```
/*
@state_machine(countN){
 @fields{
  const int num_threads;
  int counter;
  int unstamped_tickets;
  int stamped_tickets;}

 @transition{
  init initialize(int num_threads){
   init num_threads = num_threads;
   init counter = 0;
   init unstamped_tickets = num_threads;
   init stamped_tickets = 0;}

  transition tr_inc(){
   require(pre.unstamped_tickets >= 1);
   update unstampted_tickets = pre.unstamped_tickets
          - 1;
   update stamped_tickets = pre.stamped_tickets + 1;
   assert(pre.counter < pre.num_threads);
   update counter = pre.counter + 1;}

  property finalize(){
   require(pre.unstamped_tickets >= pre.num_threads);
   assert(pre.counter == pre.num_threads);}}

 @invariant{
  main_inv() {
   self.counter == self.stamped_tickets &&
       self.stamped_tickets + self.unstamped_tickets
        == self.num_threads;}}

 @proof{
  initialize_inductive(){}
  tr_inc_preserves(){}}}
*/
```

(a) Build a state machine

```
__global__ void count2N(int* counter, int N)
/* @requries gridDim.x * blockDim.x >= N &&
        counter == 0 */
{
 //count2N.initialize(n=N)
 int tid = blockIdx.x * blockDim.x + threadIdx.x;
 if (tid >= N) return;

 //count2N.tr_inc() {
 atomicAdd(counter, 1);
 // }
}
```

(b) Kernel function.

Figure 8. Example of a state machine specification.

annotations to build a semantically equivalent intermediate representation; second, separate the host and device side validation contexts based on the CUDA execution model; and finally, generate validation tasks.

The front end is responsible for parsing GPU programs with specification annotations and constructing semantically equivalent host and device side intermediate representations. So its main tasks are two fold: (1) separate host and device side code; (2) generate intermediate representations. The front end process generally includes lexical, syntactic, and semantic analysis, and intermediate representation generation. In CUDA, host and device side code can be distinguished by function qualifiers: host side code uses the host qualifier (optional), and device side code uses the device qualifier. Thus, the front end can separate host and device side code by recognizing these qualifiers. Specifically, after receiving a GPU program with annotations, the front end first performs lexical analysis to break the program into a token sequence. Then, syntactic analysis converts the token sequence into an abstract syntax tree containing host code, device code, and specification annotations. During semantic analysis, the front end separates host and device code and generates respective intermediate representations.

The Extractor's core function is to extract all verification related task code from the host and device side intermediate representations. This requires traversing the entire intermediate representation to identify code closely related to verification. To achieve this, a backward taint analysis method based on the Control Flow Graph (CFG) can be used: starting from predefined sensitive sources, trace data propagation paths backward along the control flow to locate all logic related to verification tasks.

First, define sensitive sources. On the host side, verification tasks focus on API calls and kernel launches, so any API calls or kernel launches marked by specification annotations should be considered sensitive sources. On the device side, specifications include preconditions, postconditions, invariants, and assertions. Postconditions and assertions are crucial for verification, so they should also be treated as sensitive sources.

Specifically, we use the algorithm shown in Algorithm 1 to perform backward taint analysis on the CFG for automatic verification task extraction. The main process can be divided into the following stages:

1) Basic Block Initialization: For each CFG block b, initialize the sensitive variable set $S(b)$. All variables in postconditions are added to $S(exit)$.
2) Reverse CFG Traversal: Starting from the entry block, perform a backward depth-first traversal. Each block b aggregates sensitive variables from successors into $S(b)$.
3) Instruction-Level Analysis: Scan each block's instructions in reverse. For each instruction I:
   - If I is a sensitive source (e.g., an assertion), add all variables in I to $S(b)$ and mark I as verification-relevant.
   - If I defines a variable in $S(b)$, add its operands to $S(b)$ and mark I as part of the verification task.
4) Output: Collect and output all instructions on critical paths as verification tasks.

4.4 Verification Backends

The Verification backend is responsible for conducting the final verification of extracted verification tasks and generating corresponding reports. Rather than addressing the entire program's complexity, it focuses specifically on components directly pertinent to the verification objectives. This backend comprises two primary components: a transformer and a verification engine. The transformer's role is to construct concrete verification programs based on the verification tasks, adhering

Algorithm 1: Verification task extraction algorithm.

Input: CFG ;
Output: The extracted validation task $Task$;

Function `TaskExtract(` CFG`)`**:**
  $Task \leftarrow \emptyset$;
  foreach *BasicBlock* $b \in CFG$ **do**
    $S(b) \leftarrow \emptyset$;
  foreach *Variable* $v \in post\_cond$ **do**
    $S(exit) \leftarrow v$;
  `AnalyzeBasicBlock(`*entry*`)`;
  return $Task$;
Function `AnalyzeBasicBlock(`b`)`**:**
  if b *has already been visited* **then**
    return;
  foreach b*'s successor block* $succ$ **do**
    `AnalyzeBasicBlock(`$succ$`)`;
    $S(b) = S(b) \cup S(succ)$;
  foreach *Instructions in* b I*, back to front* **do**
    if I *is a sensitive source* **then**
      foreach *variable* $v \in I$ **do**
        $S(b) \leftarrow v$;
      $Task \leftarrow (b, I)$;
    else if *The variable* $Def(I)$ *defined by* I *contains the sensitive variables in the sensitive set* $S(b)$ **then**
      $S(b) = S(b) \cup Use(I)$;
      $Task \leftarrow (b, I)$;

to a progressive verification approach. Depending on the nature of the problem, the transformer converts the verification task into a program compatible with the selected verification engine. Different verification engines can be employed based on the specific focus and complexity of the problem.

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.

4.5 Implementation

To validate our design, we develop a prototype implementation SHARD of our method.

Within SHARD, we leverage Clang [22] to parse CUDA programs, enabling the extraction of our proposed annotation specifications and the generation of corresponding intermediate representations. We have implemented the verification task extraction algorithm on this intermediate representation.

In the verification backend, we have adopted different implementations for distinct verification tasks. For verifying host-side API calls, we primarily employ a rule-based strategy. We have specifically implemented checking rules for three commonly used CUDA host APIs: cudaMemcpy, cudaFree, and cudaMalloc.

For kernel function verification tasks, we leverage Dafny as the verification backend. Dafny [23] is a programming language specifically designed for formal verification, integrating powerful features such as program logic, automated reasoning, and program verification, which effectively ensure program correctness. Compared to traditional SMT solvers like Z3, Dafny offers a higher level of abstraction.

Finally, for the verification of concurrency issues, we adopt Verus [33] as the core tool for symbolic state machine verification. Verus is an advanced formal verification tool based on Rust, focusing on verifying the correctness of multithreaded programs and system software. By combining Rust's memory safety features with formal verification techniques, Verus provides developers with an efficient, rigorous, and reliable platform. Its strong state machine modeling capabilities simplify the detection of potential errors in concurrent programs.

5. EVALUATION

To evaluate the effectiveness and efficiency of SHARD, we address the following research questions:

RQ1: Effectiveness. Is SHARD effective in verifying the security of GPU programs?

RQ2: Efficiency. How about SHARD's efficiency to verify CUDA programs?

All experiments and measurements were conducted on a server equipped with an 8-core Intel i7 CPU, 16 GB of RAM, and running Ubuntu 24.04.

5.1 Datasets

We construct two complementary datasets for evaluation:

Micro-benchmark. We show in Table 1 a micro-benchmark consisting of 10 small CUDA programs we construct manually. These programs cover four common GPU concurrency scenarios and are designed to validate SHARD's effectiveness on controlled test inputs.

Real-world programs. The benchmark suite consists of 16 test cases constructed based on GPUVerify [24] and VeriCUDA [25] benchmarks. The GPUVerify benchmarks are derived from NVIDIA's CUDA Samples [35], a collection of example programs designed for validating the CUDA environment, learning the programming interface, and evaluating GPU performance.

5.2 RQ1: Effectiveness

To address RQ1, we apply SHARD to the real-world benchmark and compare it against two state-of-the-art tools: GPUVerify [7] and VeriCUDA [32]. GPUVerify is a formal verification tool specialized in detecting data races and barrier-divergence errors in CUDA kernels. VeriCUDA is a framework for verifying functional correctness in race-free CUDA programs. Both tools use Hoare logic-based specifications, ensuring a fair comparison with SHARD.

Table 1. Micro-benchmark.

Category	Case	Description
Basic Synchronization	C1-1	Multithreaded atomic operations
	C1-2	Intra-block synchronization
	C1-3	Global memory access
Execution Control	C2-1	Device-side kernel launch dependencies
	C2-2	Synchronization in branch statements
Memory Model	C3-1	Inter-block non-coherent memory accesses
	C3-2	Preventing inter-thread illegal memory overwrites
Advanced Concurrency	C4-1	Multi-stream task scheduling
	C4-2	Synchronization between nested tasks
	C4-3	Asymmetric synchronization placement in loops

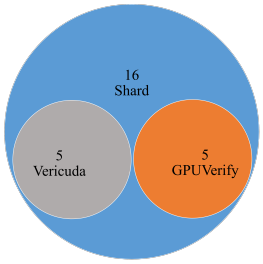


Figure 9. SHARD, GPUVerify, and Vericuda coverage of error cases

For each test case, we conduct two experiments:

- True-positive assessment: Running on the original, unmodified program to measure the ability to avoid false alarms.
- Error-detection assessment: Performed by manually injecting faults into each test case to evaluate the system's ability to detect actual bugs.

Each experiment was repeated ten times to ensure result stability, and we report the average execution time. The results are presented in Table 2.

The "GroundTruth" column distinguishes bug-free (BF) originals from fault-injected variants containing one of four defect types:

- Local Errors (LE): Thread-local logic faults (e.g., incorrect initialization values)
- LE-Thread (LE-T): Erroneous thread-index computations causing data races
- API Misuses (API-E): Incorrect CUDA API calls
- Kernel Launch Errors (KL-E): Deviations from intended thread-block configurations

Columns 4–6 of Table 2 report the detection results for SHARD, GPUVerify, and VeriCUDA (✓ denotes successful detection; ✗, a missed defect). Columns 7–9 list the average runtime (RT, in seconds) over ten runs.

None of the tools produced false positives on bug-free benchmarks. On fault-injected cases, SHARD consistently detected all four defect types. In contrast, GPUVerify and VeriCUDA missed several, as illustrated by the Venn diagram in Figure 9. GPUVerify, limited to data races and barrier divergence, cannot capture thread-local correctness faults. VeriCUDA, though designed for functional correctness, fails to account for thread-index variations, missing LE-T errors.

Furthermore, SHARD uniquely verifies host-side code, enabling detection of API misuse and misconfigured kernel launches—benefits made possible by its annotation-based specification framework.

To evaluate the effectiveness of our state-machine-based approach for detecting and verifying concurrency issues, we conducted experiments on the micro-benchmark suite. For each benchmark, we manually inserted symbolic state-machine annotations required by SHARD and constructed two state-machine models of differing complexity: one for verifying thread-safety and the other for validating functional properties. The results are summarized in Table 3.

In concurrency-safety verification, SHARD successfully verified 8 out of 10 micro-benchmarks (recall = 80%, precision = 100%), outperforming GPUVerify, which succeeded in 6 cases (recall = 60%, precision = 100%). This difference arises from a key methodological distinction: GPUVerify focuses on individual kernel scopes, while SHARD employs symbolic state machines to track state transitions across function boundaries. For functional correctness, SHARD verified 6 of 10 cases (recall = 60%, precision = 100%), whereas VeriCUDA failed on all. VeriCUDA's reliance on an a priori race-freedom assumption limits its ability to reason about concurrency. In contrast, SHARD builds functional verification atop its concurrency-safety layer, enabling hierarchical semantic analysis. However, precise modeling of dynamic data dependencies remains challenging.

Manual analysis shows that SHARD's performance degrades with program structural complexity. Benchmarks with nested loops or recursion cause state-space explosion, explaining the two unresolved concurrency-safety cases. For functional correctness, SHARD ensures soundness under verified safety, but conservative handling of data dependencies can lead to underapproximation.

In summary, despite limitations in handling path explosion and dynamic dependencies, SHARD's symbolic state machine approach demonstrates strong verification capabilities for concurrency safety and functional correctness in realistic GPU programs.

5.3 RQ2: Efficiency

To address RQ2, we evaluate SHARD's performance overhead on the real-world benchmark. We decompose this overhead into two components: compilation time (front-end parsing and task extraction) and verification time (back-end analysis). Using the real-world benchmark dataset, we measure the average proportion of time spent in these two phases for each test case in both its original and fault-injected versions. The results are shown in Fig. 10.

To compare the performance overhead of SHARD with that of GPUVerify and VeriCUDA, we measure the execution times of all three tools on the fault-injected versions of the real-world benchmark. The results are presented in Fig. 11.

Experimental results on the original benchmark set show that all three tools—SHARD, VeriCUDA, and GPUVer-

Table 2. Evaluational results on real-world benchmark.

#	Test Case	GroundTruth	SHARD	GPUVerify	Vericuda	SHARDRT	GPUVerifyRT	VericudaRT
1	initValue.cu	BF	✓	✓	✓	0.2747	0.4333	0.3338
		LE	✓	✗	✓	0.2412	-	0.2933
2	scp.cu	BF	✓	✓	✓	0.3208	0.6355	0.3764
		LE	✓	✗	✓	0.3101	-	0.3312
3	rotate.cu	BF	✓	✓	✓	0.4850	0.5512	0.4658
		LE	✓	✗	✓	0.4314	-	0.4616
4	vectorSub.cu	BF	✓	✓	✓	1.1532	1.6987	1.2836
		LE	✓	✗	✓	1.0065	-	1.1759
5	matrixMul.cu	BF	✓	✓	✓	4.2525	5.2319	4.2409
		LE	✓	✗	✓	4.1121	-	4.0399
6	_stereoDisparity.cu	BF	✓	✓	✓	0.5758	7.2469	0.6409
		LE-T	✓	✓	✗	0.5659	6.0614	-
7	computeVisibilities.cu	BF	✓	✓	✓	0.6402	0.7798	0.5917
		LE-T	✓	✓	✗	0.5668	0.6940	-
8	convolutionColumnsKernel.cu	BF	✓	✓	✓	0.6123	2.6436	0.5952
		LE-T	✓	✓	✗	0.5877	2.5833	-
9	dwtHaar1D.cu	BF	✓	✓	✓	1.1386	7.1061	1.0847
		LE-T	✓	✓	✗	1.0650	6.5591	-
10	transposeNaive.cu	BF	✓	✓	✓	0.6353	1.0820	0.6429
		LE-T	✓	✓	✗	0.5959	0.9130	-
11	simpleIPC.cu	BF	✓	✓	✓	0.5655	0.5850	0.5991
		API-E	✓	✗	✗	0.5572	-	-
12	simpleMPI.cu	BF	✓	✓	✓	0.5862	0.6653	0.6273
		API-E	✓	✗	✗	0.5627	-	-
13	simpleMultiCopy.cu	BF	✓	✓	✓	0.6681	0.9855	0.6605
		API-E	✓	✗	✗	0.6342	-	-
14	simpleMultiGPU.cu	BF	✓	✓	✓	0.6481	0.8956	0.6221
		KL-E	✓	✗	✗	0.6088	-	-
15	simpleTexture.cu	BF	✓	✓	✓	0.7053	0.7518	0.6569
		KL-E	✓	✗	✗	0.6510	-	-
16	vectorAdd.cu	BF	✓	✓	✓	0.6586	0.6459	0.6915
		KL-E	✓	✗	✗	0.6047	-	-

Table 3. Evaluation results on micro-benchmark.

#	Case	Thread-safety		Functional properties	
		SHARD	GPUVerify	SHARD	Vericuda
1	C1-1	✓	✓	✓	✗
2	C1-2	✓	✓	✓	✗
3	C1-3	✓	✓	✓	✗
4	C2-1	✓	✗	✗	✗
5	C2-2	✓	✓	✓	✗
6	C3-1	✓	✓	✓	✗
7	C3-2	✓	✓	✓	✗
8	C4-1	✓	✗	✗	✗
9	C4-2	✗	✗	✗	✗
10	C4-3	✗	✗	✗	✗

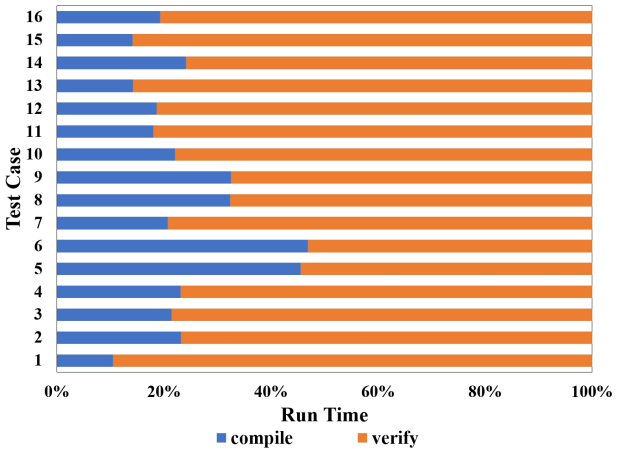
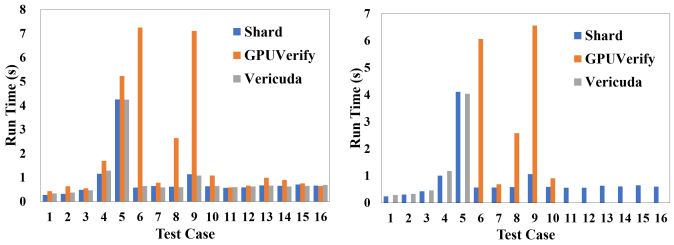


Figure 10. SHARD's overhead.

ify—produced measurable runtime data. SHARD incurred an average runtime overhead of 98.64% over VeriCUDA and 45.00% over GPUVerify. The comparable performance between SHARD and VeriCUDA stems from their shared use of Hoare logic-based verification, both generating verification conditions (VCs) from annotations and applying simplification techniques. VeriCUDA's VC simplification is functionally similar to SHARD's verification task extraction.

In contrast, SHARD demonstrates lightweight verification advantages over GPUVerify, which performs exhaustive kernel-wide checks to detect data races. Such thorough analysis often



(a) Original versions (b) Fault-injected versions

Figure 11. A comparison of SHARD, GPUVerify, and Vericuda regarding execution time on real-world benchmark.

leads to longer runtimes, despite most race conditions arising from localized issues like thread index miscalculations. By enabling specification-driven focus on critical code regions, SHARD reduces unnecessary analysis and improves efficiency.

On the fault-injected benchmarks, partial runtime data is unavailable for GPUVerify and VeriCUDA due to their inability to process certain fault scenarios. Nonetheless, the observed runtimes align with trends from the original benchmarks, indicating that each tool's relative performance remains consistent across both program variants.

6. DISCUSSION

We discuss limitations of this work and future directions for improvements.

Automated code and template annotations. Our current specification language enables concise annotations for marking critical code regions and properties. However, the manual annotation process imposes cognitive overhead and risks inconsistency. Future work could explore automated annotation techniques based on heuristic strategies or large-scale pretrained language models to extract error-prone code fragments and key properties. In addition, developing template annotations for common concurrency patterns (e.g., grid-stride loops, reductions, barrier synchronization) may facilitate semi-automated verification by allowing developers to reuse predefined patterns, thereby reducing annotation effort and improving consistency.

Broader vulnerability coverage. The current verification targets a representative but limited subset of CUDA vulnerability classes. In practice, security issues span a wider spectrum. Future work should expand the verification scope to cover more diverse vulnerability types and evaluate the applicability of lightweight formal methods to additional threat models.

Comprehensive evaluation. Our current evaluation is confined to a small benchmark suite. Although effective for testing incremental verification impact, this scale may not fully reflect the complexity of real-world CUDA programs. We plan to extend evaluation to larger and more diverse codebases, and further optimize implementation details to enhance scalability, robustness, and practical utility in GPU security analysis.

7. RELATED WORK

CUDA bug detection. Current CUDA program bug detection technologies mainly focus on two core issues: memory safety [26]–[28] and data races [29], [30]. CURD [6] uses static analysis to select appropriate race detection algorithms. By recording read and write sets of Synchronization - Free Regions (SFRs), it detects data races through set intersections. cuCatch [4] combines optimized compiler instrumentation and driver support to identify memory safety errors. However, since existing studies rely on instrumentation techniques, they cause program bloat and performance overhead.

GPU verification. Program verification is crucial for ensuring software correctness in software engineering and computer science. Several studies and tools have been developed for GPU kernel verification. GPUVerify [7] uses a Synchronous, Delayed Visibility (SDV) based method to verify GPU kernels for data races and barrier divergence. ESBMC-GPU [31] employs SMT-based bounded model checking to detect errors and concurrency issues in CUDA programs. Vericuda [32] automates CUDA program verification by adding Hoare triples under data race freedom assumptions. Faial [8] uses a combined analysis based on memory access protocols to detect data race freedom violations in CUDA programs. The core of these tools is formal methods - based verification of GPU kernels to prove their correctness and safety. Fully formal verification is generally reliable. However, for complex GPU kernels, it often faces path explosion, making Verification Conditions (VCs) generation and solving difficult and time - consuming. In applications where program rigor isn't absolutely critical, lightweight formal methods offer an efficient and practical strategy for safety checking.

Safe languages for GPUs. Descend [5] is a safe language designed specifically for GPU programming, drawing inspiration from the Rust [34] language in terms of design philosophy. particularly Rust's ownership and lifetime management model. By introducing ownership tracking and lifetime checks into the type system, Descend achieves static management of CPU and GPU memory safety. This mechanism effectively prevents the occurrence of many common errors at the source, such as memory leaks, illegal access, and data races. Although Descend provides strong static security guarantees, its syntax is more complex compared to CUDA. This complexity mainly stems from its precise control over memory management and strict requirements of the type system. For developers accustomed to using CUDA, there may be a certain learning curve to master Descend in the short term. However, in the long run, the safety and reliability provided by Descend can significantly reduce runtime errors and debugging time, thereby improving the overall efficiency of program development.

8. CONCLUSION

In this work, we propose an approach to enhance the safety of GPU kernels through lightweight formal verification. Our approach allows developers 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. Stephen son, "CuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications," Proceedings of the ACM on Program ming Languages, vol. 7, no. PLDI, pp. 124–147, 2023.

[5] 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.

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

[7] 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.

[8] 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.

[9] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, EPIC: An energy efficient, high-performance GPGPU computing research infrastructure," arXiv preprint arXiv:1912.05848, 2019.

[10] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020, pp. 473–486.

[11] O. Hennigh, S. Narasimhan, M. A. Nabian, A. Subramaniam, K. Tangsali, Z. Fang, M. Rietmann, W. Byeon, and S. Choudhry, "NVIDIA SimNet™: An AI-accelerated multi-physics simulation framework," in International conference on computational science. Springer, 2021, pp. 447–461.

[12] A. Shanbhag, S. Madden, and X. Yu, "A study of the fundamental performance characteristics of gpus and cpus for database analytics," in Proceedings of the 2020 ACM SIGMOD international conference on Management of data, 2020, pp. 1617–1632.

[13] NVIDIA, "cuBLAS: Basic Linear Algebra on NVIDIA GPUs," https:// developer. nvidia. com/ cublas.

[14] NVIDIA, "NVIDIA cuFFT," https:// developer. nvidia. com/ cufft.

[15] NVIDIA, "NVIDIA cuDNN," https:// developer. nvidia. com/ cudnn.

[16] B. van Werkhoven, W. J. Palenstijn, and A. Sclocco, "Lessons learned in a decade of research software engineering gpu applications," in International Conference on Computational Science. Springer, 2020, pp. 399–412.

[17] O. Hasan and S. Tahar, "Formal verification methods," in Encyclopedia of Information Science and Technology, Third Edition. IGI global, 2015, pp. 7162–7170.

[18] S. Paul, E. Cruz, A. Dutta, A. Bhaumik, E. Blasch, G. Agha, S. Patterson, F. Kopsaftopoulos, and C. Varela, "Formal verification of safety-critical aerospace systems," IEEE Aerospace and Electronic Systems Magazine, vol. 38, no. 5, pp. 72–88, 2023.

[19] M. Krichen, "Formal methods and validation techniques for ensuring automotive systems security," Information, vol. 14, no. 12, p. 666, 2023.

[20] Y. Murray and D. A. Anisi, "Survey of formal verification methods for smart contracts on blockchain," in 2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS). IEEE, 2019, pp. 1–6.

[21] D. A. Naumann, "Thirty-seven years of relational hoare logic: remarks on its principles and history," in Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles: 9th In ternational Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part II 9. Springer, 2020, pp. 93–116.

[22] "Clang: a C language family frontend for LLVM," https://clang.llvm.org.

[23] 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.

[24] "Gpuverifybenchmarks," https://github.com/ mc-imperial/ GPU-VerifyBenchmarks.

[25] "VeriCUDA," https://github.com/ Software Foundation Group At KyotoU/ VeriCUDA.git.

[26] 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 Depend able Systems and Networks (DSN). IEEE, 2024, pp. 362–376.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] A. Lattuada, T. Hance, J. Bosamiya, M. Brun, C. Cho, H. LeBlanc, P. Srinivasan, R. Achermann, T. Chajed, C. Hawblitzelet al., "Verus:A practical foundation for systems verification," in Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, 2024, pp. 438–454.

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

[35] NVIDIA, "CUDA Samples," https://github.com/NVIDIA/cuda-samples.