

ProofWright: Towards Agentic Formal Verification of CUDA

BODHISATWA CHATTERJEE, Georgia Institute of Technology, USA

DREW ZAGIEBOYLO, NVIDIA Research, USA

SANA DAMANI, NVIDIA Research, USA

SIVA HARI, NVIDIA Research, USA

CHRISTOS KOZYRAKIS, Stanford University & NVIDIA Research, USA

Large Language Models (LLMs) are increasingly used to automatically generate optimized CUDA kernels, substantially improving developer productivity. However, despite rapid generation, these kernels often contain subtle correctness bugs and lack formal safety guarantees. Runtime testing is inherently unreliable—limited input coverage and reward hacking can mask incorrect behavior—while manual formal verification is reliable but cannot scale to match LLM output rates, creating a critical validation bottleneck.

We present ProofWright, an agentic verification framework that bridges this gap by integrating automated formal verification with LLM-based code generation. ProofWright provides end-to-end guarantees of memory safety, thread safety, and semantic correctness for LLM-generated CUDA kernels. On KernelBench L1, ProofWright verifies safety properties for 74% of generated kernels, uncovers subtle correctness errors missed by conventional testing, and establishes semantic equivalence for a class of element-wise kernels. With a modest overhead of ~3 minutes per kernel, ProofWright demonstrates that scalable, automated formal verification of LLM-generated GPU code is feasible—offering a path toward trustworthy high-performance code generation without sacrificing developer productivity.

ACM Reference Format:

Bodhisatwa Chatterjee, Drew Zagieboylo, Sana Damani, Siva Hari, and Christos Kozyrakis. . ProofWright: Towards Agentic Formal Verification of CUDA. *J. ACM* 37, 4, Article 111 (August), 27 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Large-Language Models (LLMs) have been remarkably successful in a wide range of domains spanning from natural language processing to software engineering. For the past couple of years, agent-based AI systems have been extensively used for generating and optimizing programs [9, 14, 33, 37]. This includes agentic-AI systems for GPU program optimizations [4, 10, 18], where the goal is to automate the tedious and difficult process of generating highly efficient GPU kernels by leveraging LLMs. These agentic systems follow an iterative approach of code generation: the first step involves generating GPU kernel(s) based on a given user specification or prompt. The generated code is then subjected to validation and performance testing; based on the testing feedback, the kernels can then be iteratively refined until the feedback is satisfactory.

However, owing to the lack of correctness guarantees in the generated code, gaining explicit trust in such agentic systems remains an open challenge. This issue is particularly pronounced

Authors' Contact Information: Bodhisatwa Chatterjee, bodhi@gatech.edu@, Georgia Institute of Technology, Atlanta, USA; Drew Zagieboylo, dzagieboylo@nvidia.com, NVIDIA Research, Westford, USA; Sana Damani, sdamani@nvidia.com, NVIDIA Research, Santa Clara, USA; Siva Hari, shari@nvidia.com, NVIDIA Research, Santa Clara, USA; Christos Kozyrakis, kozyraki@stanford.edu, Stanford University & NVIDIA Research, Santa Clara, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1557-735X/8-ART111

<https://doi.org/XXXXXXX.XXXXXXX>

for highly optimized GPU kernels, where aggressive performance tuning often introduces subtle synchronization and memory-access errors that may not be caught in testing. Such kernels can be inherently complex as they often combine massive parallelism with strict synchronization requirements, intricate memory hierarchies, and numerous low-level optimizations. As a result, validating the correctness of GPU programs raises deeper concerns about how much confidence can be placed in the testing and verification performed by agentic systems.

Current agentic-AI systems [4, 9, 18] reason about the correctness of the generated code by performing compilation checks and functional unit tests. This involves compiling the generated code, executing the program with predefined inputs, and then numerically comparing the program's output against a reference implementation. In case of failures, the error feedback is used by the agent to modify the kernel and hopefully resolve the bug causing the error. While this approach can increase code quality, it often fails to fully validate all kernels. Functional correctness on a test suite is not sufficient to rule out rare but potentially dangerous behaviors such as data races and out-of-bounds memory accesses (see Listing 1) and the complexity of modern GPU kernel optimizations make writing high-coverage test suites even more challenging. This can be further exacerbated by poorly constructed testing environments that allow the agent to *reward hack*: a case where the agent produces an output that *only* passes the validation tests but otherwise does not fit the intended specifications. Some common examples of reward hacking include copying the output of the reference implementation, or removing necessary synchronization primitives to improve performance metrics.

Although some reward hacking (such as the trivial solution-copying example in Listing 13) can be easily fixed, guaranteeing the absence of subtle implementation bugs is far more challenging. Conventional testing strategies such as *input-based dynamic testing* [15–17, 30, 34, 38] and *symbolic execution* [21, 22, 24] are capable of detecting memory safety violations and/or data races and thus improve on traditional functional testing. However, these are both more expensive to execute than typical tests and also still lack completeness guarantees; they cannot examine all possible program paths or thread interleavings (§2.2). To adopt AI-generated GPU kernels in high-integrity applications (such as automotive and avionics systems [3]), we must utilize tools that can formally rule out unsafe behaviors and demonstrate strong correctness guarantees.

To this end, we propose a new validation strategy for LLM-generated CUDA codes that complements traditional testing approaches with formal verification techniques to provide functionality and safety guarantees. In general, formal reasoning of GPU programs is particularly challenging due to concurrent execution of thousands of parallel threads, tight coupling of the programming model with various memory spaces, and varied synchronization primitives and patterns. Existing work on GPU code verification [6, 7, 11, 21, 22, 24, 43] either focus on verifying a small subset of properties such as *bank conflicts*, *improper synchronizations*, or rely on the user to provide hints for establishing a limited set of properties. Identifying the right tools for specific use cases, or constructing verifiable annotations can be tedious, error-prone, and requires expert knowledge. Furthermore, the demand for rigorous verification of GPU programs has grown substantially with the advent of agentic-AI systems, which can generate kernels at an unprecedented scale. With future LLMs increasingly trained on these AI-generated codes [18], ensuring correctness and reliability is paramount.

We present **ProofWright**, an agentic framework that formally reasons about LLM-generated CUDA codes by automatically establishing two key program properties: *implementation correctness* and *semantic equivalence*. The implementation correctness of the generated GPU code is ascertained by proving *thread safety* (i.e., the absence of data races) and *memory safety* (i.e., the absence of illegal memory accesses), while the semantic equivalence is established by checking that the generated code adheres to its original functional specification. Our contributions include:

- A *VerCors Agent* (§4) that uses in-context learning and past experience to build an Annotation Guide to automatically generate minimal contracts for VerCors, an SMT-based CUDA verification tool, to prove memory and thread safety.
- The *MLRocq Library* (§5.1.2), a formal implementation of tensor operations commonly used in machine learning applications in Rocq.
- A *Semantic Equivalence Framework* (§5) that translates a PyTorch model into an equivalent Rocq implementation and an LLM agent that uses Rocq and VerCors to validate functional correctness of a CUDA implementation of the original PyTorch.

We evaluated ProofWright on the *KernelBench* benchmark [31], which provides a comprehensive suite of PyTorch programs designed to evaluate the ability of LLMs to generate equivalent CUDA kernels that are both performant and correct. Our experiments show that ProofWright successfully establishes memory and thread-safety guarantees for up to 74% of KernelBench L1 programs and can ascertain semantic equivalence in 14% of KernelBench L1 programs.

2 Background and Motivation

This section gives a brief description of agent-based systems for GPU code generation, and motivates the shortcomings of their testing methodologies. We then describe the rationale for employing an agentic approach for verification, and introduce the existing tools used in this paper.

2.1 Agentic Systems for GPU Code Generation & Optimization

Agentic systems for GPU code generation leverage LLMs to automatically produced highly optimized GPU kernels from high-level specifications. The input specification may be unstructured natural language, but we focus on systems that take a structured high-level program as the desired output specification. PyTorch [32] is the most popular language for this use case due to its prevalent usage in GPU-accelerated computation for AI; it used as the specification and reference implementation in the KernelBench benchmark suite [31] that is targeted by several existing agentic systems [10, 18] for CUDA code generation.

Once provided the specification (which is often augmented with a system prompt and other contextual information), the agentic systems use LLMs to generate candidate GPU kernels that are then iteratively tested and refined based on feedback from various tools such as static analyzers, compilers, functional tests, performance profilers, and even other LLM judges [10, 18]. Some agentic systems structure the workflow and allow AI models to make limited (or no) judgments of which tools to use and when; others allow the LLM to plan its own execution and freely use tools and informational resources at its own discretion. In all cases, some level of functional testing is typically deterministically invoked on the final result; if it passes the tests it is accepted, otherwise the system either retries or returns no result.

2.2 Shortcomings of Current Testing Methodologies: False Negatives, Code Coverage, Compatibility Issues

CUDA programs are prone to a wide range of errors due to their massively parallel nature and the complex architecture of modern GPGPUs. The CUDA programming model exposes different memory spaces (global, shared, local) with distinct address spaces and scopes, which can potentially lead to subtle bugs that are often difficult to detect and reproduce. Furthermore, CUDA programs freely manipulate data with raw pointers which comes with all of the well known pitfalls of other memory unsafe languages. To detect these issues, programmers often use testing and debugging tools that can be broadly divided into two classes:

```

__global__ void sigmoid_kernel(..) {
    int idx = blockIdx.x * blockDim.x * 4 + threadIdx.x * 4;
    int64_t N4 = N & (~int64_t(3));
    if (idx < N4) {...}
    // Processing Tail Elements (< 4)
    if (threadIdx.x == 0) {
        int idx2 = blockIdx.x * blockDim.x + threadIdx.x;
        for (int tail = N4 + idx2; tail < N; ++tail) {
            float xf = __h2f(x[tail]);
            y[tail] = __f2h(sig(xf));
        }
    }
}

```

Listing 1. Sigmoid Kernel with race condition generated by GPT4.1, where the tail processing logic incorrectly distributes work across blocks, causing threads from blocks 0, 1, and 2 to collide on the write operation. This passes both compilation and multiple trials of unit tests.

■ **Input-based Dynamic Testing Tools:** These tools detect concurrency bugs by executing GPU programs with concrete inputs, and dynamically monitoring inter-thread interactions to expose data races and shared-memory conflicts across thread execution schedules. This includes widely-used commercial tools such as NVIDIA’s *Compute Sanitizer* [30], and popular academic tools [16, 17, 34] for detecting data-races. However, these tools can only detect race conditions and illegal memory accesses for a given input. For instance, the LLM-generated kernel depicted in Listing 1 attempts to optimize the sigmoid computation by vectorization, where each thread processes 4 consecutive FP16 elements. However, the tail-processing logic (line 6) incorrectly assigns the first thread (*threadIdx.x* = 0) of multiple (remainder) thread blocks to perform the write operation on *y[tail]* (line 10), leading to a race-condition. This condition will only arise if the original input length *N* is not a multiple of 4, and will elude the input-based dynamic testing tools for other inputs, resulting in a false negative ¹.

■ **Symbolic Execution-based Tools:** These tools systematically explore the program paths by representing the program inputs as symbolic variables, constructing logical constraints that describe all feasible execution behaviors, and using SMT solvers to reason about correctness across both CPU and GPU code. However, such tools only support simple atomic operations, and do not scale beyond smaller kernels.

Furthermore, none of these aforementioned techniques can actually determine whether the generated code semantically satisfies its intended specification. For instance, the LLM-generated kernel shown in Listing 13 (Appendix D) was supposed to implement a vector-add operation, as specified by the user. As illustrated, the kernel does not implement the intended computation, and instead generates a ‘hacked kernel’ which passes both unit and compilation tests. Moreover, existing testing strategies focus primarily on detecting the presence of errors rather than explaining their cause or providing actionable feedback. This limitation makes them inadequate in an agentic context, where the feedback loop back to the agent is necessary for iterative code refinement.

2.3 VerCors Program Verifier

VerCors [2, 8] is a deductive program verifier for concurrent code. Similar to popular languages such as Dafny [20], VerCors can prove that programs adhere to Hoare-style contracts; unlike other verification languages, VerCors can also prove memory safety and data race freedom of programs written in unsafe languages such as C and CUDA-C. VerCors provides front-ends for a variety of languages (including CUDA) which it translates to an intermediate verification language.

¹We were unable to detect this error using NVIDIA compute-sanitizer’s racecheck tool

VerCors enables fine-grained reasoning about concurrent access to shared resources via permission-based *concurrent separation logic* [13], where a permission of 1 enables write access and any non-zero fraction allows only read permission. VerCors guarantees race freedom by maintaining the invariant that the total permission for each memory location never exceeds 1.

To make verification possible, programmers must add annotations that guide how to split permissions across threads when they are launched or at synchronization barriers. As part of a CUDA kernel contract, the pre-condition is quantified over all threads that are launched, and must define how the initial permissions are split between threads. For instance, the following pre-condition requires that each thread has write access to a location in the array C specified by its canonical CUDA thread id:

```
//@ requires Perm( C[blockDim.x*blockIdx.x + threadIdx.x], write );
```

Notably, VerCors must prove that the memory locations accessed by different threads are disjoint (since they each require write permission) and in-bounds. For more complex indexing patterns, this is undecidable in general and users must specify hints to the verifier via helper functions or frame statements to make verification more likely to terminate. [Appendix E](#) demonstrates an example of using these features in a toy kernel. However, even with these hints, it is not always possible to guide the verifier into success and some correct programs may not be verifiable.

2.4 Rocq Theorem Prover

The *Rocq Prover*² [39] is an interactive theorem-proving environment that allows users to formally specify software properties as mathematical theorems and lemmas, and develop *machine-checkable* proofs through an extensible system of tactics. It is a *dependently typed* programming framework, which means that the types can depend on values (such as a vector type A parameterized by its length n : $\text{Vector } A \ n$) which allows programs and proofs to be expressed within the same formal system. At its core, the Rocq Prover uses Gallina, a high-level specification language that provides the syntax and semantics for defining functions, data types, and logical propositions.

3 ProofWright Framework: An Overview

[Figure 1](#) and [Figure 2](#) show the high-level structure of our framework. The core idea is to develop agentic flows that formally reason about LLM-generated CUDA codes by automatically establishing three key program properties: memory safety, thread safety and semantic equivalence. Our framework consists of two major components described below.

VerCors Agent (§4): We build an agentic flow that automatically utilizes a deductive program verifier to establish safety properties of a given CUDA code. The agent is responsible for producing program annotations that specify the expected program behavior for memory and thread safety and that assist the verifier via hints. In this work, we leverage the *VerCors verifier*, which is a SMT-based tool for deductive verification of concurrent and parallel software (§2.3). The VerCors agent consists of three key components: 1) a *Knowledge Base*, consisting of annotation syntax and rules obtained from VerCors documentation, 2) the *Annotation Guide*, a document automatically generated from example data that summarizes a general verification procedure, and 3) a minimal database of known errors and their corresponding fixes. The *VerCors Agent* shows strong learning capability to automatically generalize across experiences and generate memory and thread safety annotations without any manual intervention.

Semantic Equivalence Framework (§5): An agentic framework that automatically constructs proofs for determining whether AI-generated CUDA code functionally adheres to a given user specification. This framework has three major components: (1) a *static analysis tool* (§5.1.1) that

²formerly known as Coq

abstracts the initial input program into a computation graph to capture its intended functional specification, (2) an *operations mapping tool* (§5.1.3) which translates the specifications graph into mathematical properties, and (3) the *Rocq Agent* (§5.2), which generates alternative theorems and properties corresponding to the given specifications and automatically constructs proofs to establish equivalence between them. The verified theorems are then lowered into program annotations and validated by a formal verifier. The *Rocq agent* utilizes the *Rocq Theorem Prover*, an interactive proof assistant with a dependently-typed language. The *Rocq Agent* is able to generate a group of Rocq tactics in order to construct the theorem proofs.

Our agentic solution is reliable because we use formal verification tools and mathematical proofs. If any of our agents hallucinate and generate incorrect annotations or proofs, they will simply fail to verify. That is, *we will never verify a program that is incorrect*. The following sections describe each component in detail.

4 VerCors Agent: Establishing Thread & Memory Safety Guarantees

This section introduces the *VerCors Agent*, an LLM-based solution that automatically inserts Vercors annotations in CUDA programs for formal verification. Figure 1 shows the high-level agentic workflow. The VerCors Agent is an iterative LLM-based agent with access to a static, hand-written *Knowledgebase*, a dynamic, LLM-generated *Annotation Guide* that learns verification strategies from experience, and the VerCors program verifier which provides feedback to the agent. The following subsections describe the agent’s internal architecture and verification methodology in detail.

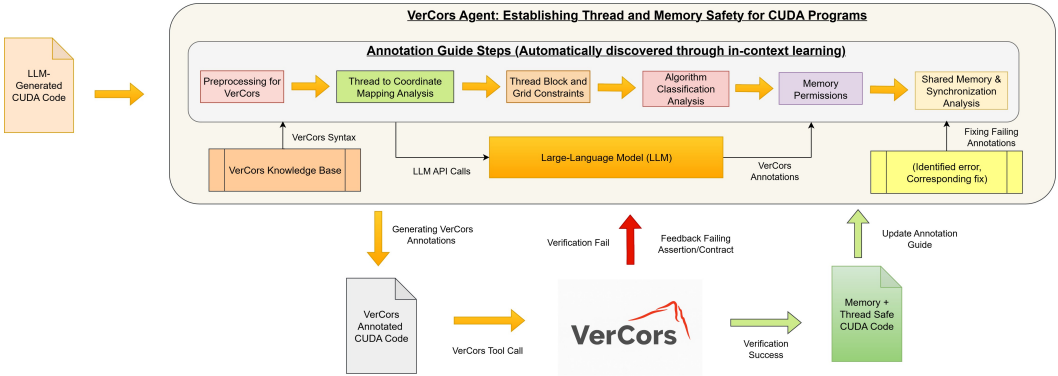


Fig. 1. **VerCors Agent for establishing memory and thread safety for CUDA Programs.** The LLM-generated CUDA kernels are first minimally preprocessed, and then the safety annotations are generated in a step-by-step fashion, according to the automatically crafted *annotation guide*. The annotated program is then verified by calling the VerCors tool.

4.1 Verification Procedure

As discussed in Section 2.3, verifying CUDA programs with VerCors requires the agent to generate formal contracts for each target kernel, specifying the preconditions necessary for safe invocation and the post-conditions guaranteed upon successful execution. For the purposes of memory and thread safety, we do not need the contracts to specify any meaningful postconditions; VerCors inherently ensures that these safety properties hold throughout the program’s execution as long as the preconditions have been satisfied. Since it is not generally possible to differentiate between a program that does not actually satisfy its contract and an instance of imprecision in the verification

process, we ask the agent to essentially infer the *weakest pre-conditions* possible such that the kernel contract is satisfiable. This approach enables the following workflow:

- Use the VerCors Agent to attempt to infer the weakest pre-conditions necessary to satisfy thread and memory safety.
- If accepted by VerCors, any invocation of the kernel that satisfies the precondition is safe; this can also be checked by VerCors as long as a sample program is supplied.
- If not, we conservatively assume that it is *never* safe to invoke the kernel.

Note that we do not confirm that the kernel contracts are indeed *weakest* preconditions; however, in our evaluation (§[section 6](#)) we do execute basic checks to ensure that the contracts produced are not trivially unsatisfiable.

An Example Kernel Verification Walkthrough. We direct the VerCors agent to jointly infer these contracts and generate any other annotations in the kernel body needed to verify the kernel. To give the reader intuition for the knowledge needed by VerCors agent to complete this task, we describe a representative kernel and its associated VerCors annotations in [Listing 2](#).

```

1  /*@
2  context_everywhere (\forallall* int i; 0 <= i && i < N*N; Perm({:A[i]:}, read));
3  context_everywhere (\forallall* int i; 0 <= i && i < N*N; Perm({:B[i]:}, read));
4  context_everywhere (access_func(..) < N && access_func(..) < N) ==>
5      Perm({: C[access_func(access_func(..), access_func(..), N, N)] :, write);
6  @*/
7  __global__ void square_matmul_basic_kernel(float* A, float* B, float* C, int N) {
8      int row = blockIdx.y * blockDim.y + threadIdx.y;
9      int col = blockIdx.x * blockDim.x + threadIdx.x;
10     if (row < N && col < N) {
11         float sum = 0.0f;
12         //@ loop_invariant 0 <= k && k <= N;
13         for (int k = 0; k < N; k++) {
14             //@ assert access_func(row, k, N, N) == row * N + k;
15             //@ assert access_func(k, col, N, N) == k * N + col;
16             sum += A[row * N + k] * B[k * N + col];
17         }
18         //@ assert access_func(row, col, N, N) == row * N + col;
19         C[row * N + col] = sum;
20     }
21 }

```

Listing 2. Example contract and annotations generated by VerCors Agent to establish memory and thread safety. The annotations before the function header define the contract and the VerCors comments in the body serve as hints to aid the automated verifier.

The agent needs to (1) identify all memory operations in the kernel, (2) determine whether each access is a read or write, and then (3) describe the memory location(s) based on the thread and block identifiers and conditions under which the memory is accessed. For example, many different locations in arrays A and B are read by each thread, so in lines 2 and 3 the agent conservatively requires that every element in the bounds of those arrays are readable by every thread. In this case the `context_everywhere` annotation is used to specify that this is not only a precondition, but also a postcondition and loop invariant.

However, the agent cannot make the same requirement for the array C since write permission is needed to execute the assignment on line 19; in this case the agent adds a *conditional* precondition on line 5. The condition matches the if statement on line 10 and the permission location matches

the index dereferenced on line 19. Note that a helper function `access_func` (which is defined by the agent elsewhere and ensures that the index is in-bounds) is used to help guide VerCors' solver on line 18. Although omitted from Listing 2, the agent also needs to infer minimum necessary array sizes, constraints on the CUDA block and grid dimensions, and reasonable bounds on any other function parameters. Furthermore, for examples with synchronization, the agent must specify how permissions change over time at each thread barrier.

The remainder of this section describes both the challenges in building the VerCors agent and how we provide it with the information necessary to achieve its goal.

4.2 Tackling Low-Resource VerCors Syntax

We found that most foundation models could not reliably produce syntactically valid VerCors-annotated CUDA. This is unsurprising as there is only a small corpus of publicly available VerCors code, even compared to other program verification languages like Dafny [20] or F* [36], and it is unlikely that any LLMs have been trained sufficiently on VerCors source code. Hence, we employ in-context learning to improve the model's ability to generate VerCors annotations.

4.2.1 VerCors Knowledge Base: In-Context Learning. As a first step, we build a VerCors knowledge base using traditional prompt engineering techniques which includes (1) context: a stripped-down version of VerCors' official documentation, (2) few-shot examples: hand-annotated, verified CUDA kernels, and (3) a set of error-fix pairs that teach the agent how to handle VerCors errors.

The VerCors knowledge base includes examples about the general VerCors syntax as well as GPU-specific examples. subsection E.1 contains two listings which show how permissions syntax are formed and evolved over constructs such as barrier synchronizations, etc.

Next, when the *VerCors Agent* fails to generate verifiable annotations on the first attempt, it must perform *proof repair* — refining its annotations iteratively based on the feedback received from the verifier. Although errors returned from the verifier were sometimes immediately helpful (e.g., parsing errors), often others were more ambiguous (e.g., "Insufficient permission to assign to field") and the agent struggled to correct them. Therefore, we provided a small set of failed verification attempts, the corresponding error message, and a fixed version of the code along with a textual explanation of the fix as part of the VerCors knowledge base. subsection E.2 illustrates the structure of the dataset entries along with a representative example.

We generated this dataset by recording the manual verification process for a complex kernel that required many rounds of iteration. While a verification expert could likely have verified this with few iterations, we intentionally started with a basic contract and corrected each error that we received one-at-a-time. This ensured that each example in the dataset was focused on a singular error, and enabled us to elicit a reasonable set of examples while analyzing few kernels. The entire knowledge base is provided to the agent as part of its prompt. It consists of ~ 12K words, a relatively small fraction of the approximately 300K word VerCors wiki.

4.2.2 VerCors Annotation Guide: Learning from Experience. While the static VerCors knowledge base described in the previous section provides a starting point to the VerCors agent, it has several limitations. First, it provides syntax and examples and not a methodology on how to approach verification of a new program. Due to the lack of exemplar data, we found this resulted in pattern-matching to generate annotations which does not scale well to new kernels that may have different control flow, data mapping, or synchronization patterns. Second, even when the agent successfully generated annotations after several iterations of the feedback loop, it fails to learn from experience and may repeat similar errors on future verification problems.

To overcome these limitations, we supplement the agent with a dynamic *VerCors Annotation Guide* where the agent automatically distills knowledge based on human-provided few-shot examples as well as past experience into a generalized verification recipe. To prevent overfitting, updates to this Annotation Guide are controlled by a human-in-the-loop in this current version (§6). This Annotation Guide serves as the agent’s core reasoning framework and helps it successfully generate VerCors-compatible specifications across a diverse set of CUDA kernel structures. Further investigation into the generated guide showed that the agent learned some interesting core verification principles across multiple sessions. We now briefly describe some of these insights.

■ **Algorithm Class-Aware Verification Scheme:** The agent discovered that different algorithm families exhibit distinct memory access patterns and synchronization primitives, requiring specialized verification strategies beyond generic templates. To address this, the agent classifies kernels into different families such as element-wise, convolution-like, reduction, etc. Each family follows unique permission and indexing schemes—ranging from simple one-to-one thread-to-element mappings to complex shared-memory coordination in convolution operations. In some cases, the agent even employs a hybrid classification strategy, combining features from multiple families to accurately capture mixed computational behaviors within a single kernel.

■ **Thread-to-Data Mapping:** The agent observed that the verification strategy and corresponding annotations vary significantly with the thread-data mapping pattern used in CUDA kernels. Different index-mapping schemes require distinct reasoning regarding memory accesses, which often necessitates specialized *helper functions* to express the mapping between thread and data coordinates and aid in bounds checking. These functions help with SMT instability as they define postconditions that can serve as hints to the verifier and its underlying solver. While some of the few-shot examples did include use of similar functions, we did not explicitly instruct the agent when or how to use them; it decided that they were helpful and proposed adding them to the annotation guide on its own. Listing 3 illustrates how the agent generates specialized helper functions for a batched 4D array access pattern, which was not demonstrated in the few-shot examples.

```

1 /*@
2 requires 0 <= batch && batch < batch_size && 0 <= ch && ch < channels;
3 requires 0 <= h && h < height && 0 <= w && w < width;
4 ensures \result == batch * channels * height * width
5           + ch * height * width + h * width + w;
6 ensures \result >= 0 && \result < batch_size * channels * height * width; @*/
7 /*@ pure @*/ int array_4d_index(..) =
8 batch * channels * height * width + ch * height * width + h * width + w;

```

Listing 3. Agent-generated helper function for a batched indexing pattern with a flattened 4D array index, where multi-dimensional tensor indices (*batch*, *ch*, *h*, *w*) are mapped into a single linear offset.

■ **Determining Thread Block & Grid Constraints:** The agent identifies thread block and grid dimension configurations based on the kernel’s computational structure (e.g., element-wise, matrix multiplication, or reduction operations). These annotations often fix certain dimensions to constant values (e.g., *blockDim.x* == 256, *gridDim.z* == 1) to simplify the verification process, while still aiming to maximize problem-space coverage by generalizing over unconstrained dimensions. This balance between fixed and parameterized dimensions enables both tractable and broadly applicable verification conditions. In addition, the agent includes annotations to ensure that all data pointers are non-null, all array dimensions are positive, and memory allocations satisfy the required length constraints.

The entire VerCors Annotation Guide consists of approximately ~ 9.5K tokens and is also provided as part of the prompt along with the KnowledgeBase. While our guide is tailored to the domain of tensor algebra operators, our automatic iterative prompt refinement strategy can be

generally applied to generate guides for other domains if the agent is provided with an initial set of hand-written examples to learn from.

5 Proving Semantic Equivalence & Functional Correctness

This section presents an agentic approach to establishing semantic equivalence between the LLM-generated CUDA code and the original specification from which it was derived. A program is considered *semantically equivalent* to its original specification if it faithfully preserves the intended behavior of the specification. In this work, we assume that the original specification is a PyTorch program. However, in order to determine whether the LLM-generated code matches the original specification, the specification must first be represented in a form amenable to mathematical reasoning. This, in turn, requires a formal definition of a core set of operations that can precisely express the program’s computational semantics. The front-end of *ProofWright*’s semantic equivalence framework (see Figure 2) primarily focuses on these two aspects and consists of (1) the *PyTorch Static Analyzer*, which captures the program’s high-level specification (written in PyTorch) in a graph-based intermediate representation (IR), (2) the *MLRocq library* which provides a corpus of formal definitions for popular neural and tensor computations in Rocq, and (3) the *PyTorch to Rocq Translation tool*, which uses the MLRocq library to translate the extracted graph into equivalent Rocq representations. Note that the front end of the semantic equivalence framework does not employ any LLM components, ensuring that the formal specification process remains trustworthy.

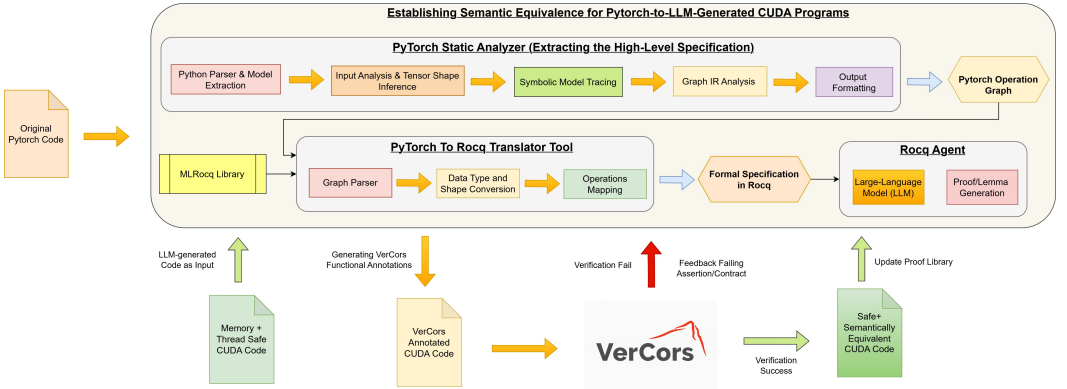


Fig. 2. **Semantic Equivalence Framework.** The *front-end* captures the original specification in a graph-based IR form, and is translated to Rocq theorems. The agentic *back-end* explores alternative mathematical representations of the original specification, and leverages them to generate low-level VerCors annotations.

The back-end of the semantic equivalence framework consists of the *Semantic Equivalence Agent*, which generates the corresponding Rocq specification for the target CUDA program, builds a Rocq theorem, and automatically constructs proofs to validate its equivalence to the PyTorch specification. While this step proves that the LLM-generated Rocq representation of the CUDA program is equivalent to the original specification, we still do not know if the Rocq representation actually matches the CUDA program we are trying to verify. To this end, the agent lowers the verified specification into VerCors functional annotations, embeds them into the LLM-generated CUDA code, and invokes the VerCors tool to verify the correctness of the annotated program. We describe the Semantic Equivalence framework in greater detail in the following sections.

5.1 Front-End: Translating Pytorch Specification to Rocq

5.1.1 PyTorch Static Analyzer. The *PyTorch Static Analyzer* abstracts the high-level semantics of the pytorch input program by transforming it into a graph-based IR. Each computation in the PyTorch program is represented as an *operation node*, while the edges capture data dependencies between operations, forming a dataflow graph that encodes the program's computational structure. Each node in the graph is further enriched with tensor metadata, including shape information and auxiliary parameters such as dimensions and scaling factors. Figure 3 shows an example of an operations graph for an RMS norm program from the KernelBench Benchmark.

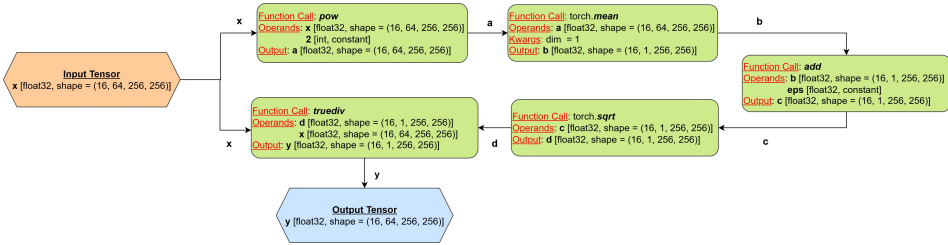


Fig. 3. **Operations Graph IR generated by Pytorch Static Analyzer for RMS.** The input tensor $x \in \mathbb{R}^{16 \times 64 \times 256 \times 256}$ is squared, reduced across the channel dimension, stabilized with an epsilon constant, square-rooted, and finally broadcast-divided with the original tensor to produce the normalized output.

Internally, the tool leverages *torch.fx* to symbolically trace the model, and record operations to construct an FX graph, which is then iteratively refined with tensor data (Kwargs, shape, etc) if possible.

5.1.2 MLRocq Library. *MLRocq* is a formal verification library implemented in Rocq that provides mathematically rigorous definitions and proofs for deep learning operations. It defines a base data type *TensorND*, which represents an n -dimensional tensor, recursively constructed as a list of $(n - 1)$ -dimensional tensors (Listing 4). The *MLRocq* library currently implements nearly one hundred operations from the PyTorch library that are widely used in modern deep learning models. These include a broad spectrum of functional categories such as *activation functions*, *normalization layers*, *pooling* and *reduction* operations, and standard *loss functions*.³

```
1 Fixpoint TensorND (n : nat) : Type :=
2   match n with
3   | 0%nat => Z
4   | S n' => list (TensorND n')
```

Listing 4. Definition of a recursive Coq type *TensorND* that generalizes tensors to arbitrary dimensions. The base case defines a 0-dimensional tensor as a scalar (Z), and the recursive case represents an $(n + 1)$ -dimensional tensor as a list of n -dimensional tensors.

Listing 5 shows an example where the ReLU activation is applied to an n -dimensional tensor. The function recursively applies the scalar ReLU to each element by mapping it across all sub-tensors using the *map* operation, thereby extending element-wise activation to tensors of arbitrary dimensions. This represents a common implementation strategy in the *MLRocq* library—each operation is first defined for the scalar case and then recursively applied to higher-dimensional tensors.

Additionally, the library includes over 200 test cases designed to validate the correctness of these definitions with respect to their PyTorch equivalents. In its current version, most tensor and

³A complete list of MLRocq operations is shown in Appendix A

operation definitions are restricted to integer/real data types to simplify the construction of formal proofs.

```

1 (* Element-wise ReLU: max(0, x) *)
2 Definition relu_sc (x:Z) : Z :=
3   if x <? 0 then 0 else x.
4 (* ReLU for n-dim tensor *)
5 Fixpoint relu_t (n:nat)
6   (t:TensorND n) : (TensorND n) :=
7   match n return TensorND n -> TensorND n with
8   | 0%nat => fun x => relu_sc x
9   | S n' => fun xs =>
10     List.map (relu_TensorND n') xs
11   end t.

```

Listing 5. Recursive definition of ReLU for n-d tensors which uses scalar version for base case and List.map for higher dimensions.

Additionally, the library includes over 200 test cases designed to validate the correctness of these definitions with respect to their PyTorch equivalents. In its current version, most tensor and operation definitions are restricted to integer/real data types to simplify the construction of formal proofs.

5.1.3 Pytorch-to-Roq Translator. The third component of the *Semantic Equivalence Framework* is the translation stage, which takes the PyTorch operations graph and systematically maps each node to its formally defined Rocq counterpart using the MLRocq library. During this process, the translator resolves the operator with its corresponding mathematical definition (e.g., matrix multiplication, activations, etc) and lifts all operands into well-typed Rocq entities using tensor metadata (shape, type). This produces a formal, type-safe specification in Rocq that preserves the program’s original computational semantics and serves as the foundation for downstream theorem generation and equivalence proofs. Note that once a proof is completed, it is stored in the MLRocq library, in order to avoid recomputing again.

5.2 Backend: Proving Semantic Equivalence

The final component of the *Semantic Equivalence Framework* is the *Semantic Equivalence Agent*, which operates on the formally translated specifications to establish semantic equivalence with the LLM-generated CUDA codes. It synthesizes Rocq theorems and auxiliary properties representing equivalent formulations of the CUDA program’s computational semantics and automatically constructs Rocq proofs to validate them against the original PyTorch specification.

As an example, starting from the original ReLU specification in Listing 5, the *Rocq Agent* synthesizes the corresponding Rocq representation of the CUDA program (Listing 11), defining both the scalar base case and its recursive extension to n -dimensional tensors. In order to prove that these specifications are indeed equivalent, we observe that the agent automatically constructs the proof for each of these versions by performing a two-level inductive reasoning in a single shot (Listing 1, Appendix B) and validates it using the Rocq tool.

Finally, the agent derives the VerCors functional annotations from the combined space of original and synthesized specifications and embeds them into the LLM-generated GPU code for formal verification. A successful verification by VerCors indicates that the CUDA program has the intended output behavior and provides high assurance that it satisfies the original PyTorch specification. In contrast, if verification fails, the agent analyzes the error to infer the conditions under which the LLM-generated program deviates from the intended semantics.

```

1 (* Scalar ReLU using arithmetic trick: (x + |x|) / 2 *)
2 Definition relu_arith (x : Z) : Z := (x + Z.abs x) / 2.
3 (* Extending to n-dimensional tensors *)
4 Fixpoint reluAR_t (n : nat) (t : TensorND n) : TensorND n :=
5   match n return TensorND n -> TensorND n with
6   | 0%nat => fun x => relu_arith x
7   | S n' => fun xs => List.map (reluAR_t n') xs
8   end t.

```

Listing 6. Rocq representation of ReLU operation automatically synthesized by Semantic Equivalence Agent for scalar and n-dimensional tensor

```

1 // Functional correctness: x + |x|/2 operation
2 /*@
3 ensures \gtid<size ==> { :output[\gtid]:} == input[\gtid]+fabsf(input[\gtid])/2.0f;
4 @*/
5 __global__ void relu_kernel(float* input, float* output, int size) {
6   int idx = blockIdx.x * blockDim.x + threadIdx.x;
7   if (idx < size) {
8     output[idx] = input[idx] + fabsf(input[idx]) / 2.0f;}}

```

Listing 7. VerCors Functional Annotation generated by Semantic Equivalence Agent.

We note that our current approach still does not guarantee the complete correctness of the lowering process that translates Rocq definitions into *VerCors* functional annotations for all possible types of GPU Kernels, as this step is performed using an LLM. While in future work we seek to replace this step with a procedural compiler, we currently demonstrate the feasibility of our approach by manually verifying that the lowered VerCors annotations match the original specifications in our evaluation.

6 Evaluation

The evaluation of *ProofWright* is two-fold. The first set of experiments focuses on memory safety and thread safety, while the second set of experiments explore semantic equivalence of LLM-generated CUDA codes. Specifically, we evaluate *ProofWright* to answer the following set of research questions:

- **RQ1 (§6.1):** To what extent can the *ProofWright* successfully establish memory and thread-safety guarantees for LLM-generated GPU codes? Furthermore, can it generalize to unseen kernels and handle increasingly complex verification scenarios?
- **RQ2 (§6.2):** How lightweight is the annotation generation process, and does it scale to kernels that employ complex primitives and larger problem sizes?
- **RQ3 (§6.3):** How important are the individual components (*knowledge base* and *annotation guide*) in the *VerCors Agent* prompt? Can the *VerCors Agent* succeed in establishing memory and thread safety without them?
- **RQ4 (§6.4):** What kinds of high-level specifications can the semantic-equivalence framework capture and represent formally? Under what conditions is the *Rocq Agent* able to synthesize alternate specification and automatically construct proofs?
- **RQ5 (§6.5):** Finally, to what extent can *ProofWright* establish semantic equivalence for LLM-generated GPU kernels, in spite of currently relying on an untrusted lowering stage?

Experimental Setup: The experiments were conducted on a MacBook Pro equipped with a 14-core Apple M4 Pro processor, running macOS Redwood. The *VerCors* tool was built from source⁴ and minimally extended to support additional CUDA constructs required for our experiments.

Ensuring VerCors Compatibility by Kernel Preprocessing: In our experiments, we had to rewrite the GPU program into a form compatible with VerCors, since VerCors does not yet support the entirety of C++ syntax, or many of the latest CUDA features such as tensor core operations. This preprocessing step simplifies the kernel so that it can be analyzed by VerCors for various properties.

Baselines: The baseline CUDA kernels from KernelBench L1 [31] were generated using the Claude-4-Sonnet and GPT-5-Codex models.

Setting up the Annotation Guide: The initial annotation guide was derived from 10 manually verified kernels from the knowledgebase (8 from the VerCors documentation and 2 from KernelBench). This initial version served as the foundation that the agent progressively evolved as it accumulated the learning from successful verification outcomes. In the current version, we introduced human supervision during subsequent updates to avoid overfitting, since the training corpus consisted of only a limited set of kernels.

6.1 Memory & Thread Safety: Extent of Coverage (RQ1)

The *VerCors Agent*, equipped with the *knowledge base* and *annotation guide* was tasked with generating memory and thread-safety annotations for the *Claude-4-Sonnet* baseline across 100 L1 problems from the KernelBench benchmark. As shown in Fig. 4, the agent successfully established memory and thread safety for 74 kernels, spanning variants of *matrix multiplication*, and other common deep-learning primitives. Fig. 5 further breaks down the success rate across different kernel categories. Overall, the outcomes of the *VerCors Agent* can be grouped into three broad classes:

- **Successful Verification (74%):** In successfully verified cases, we observe that the agent is able to infer correct permissions for previously unseen kernels by generalizing permission patterns. It first classifies the kernel’s algorithmic family, analyzes its memory-access structure, and then selects an appropriate base annotation template, which is customized to the given kernel’s indexing and access patterns. For example, in the *MaxPooling2D* kernel (L1-42), the agent correctly recognized that each thread writes to a unique output index but performs overlapping reads, and generated the corresponding safety permission scheme. We also notice when an initial verification attempt fails, the agent decomposes complex, high-dimensional index expressions (4D access) with helper functions (2D accesses), which improves VerCors’s ability to reason about them.

- **Agent Failure (9%):** In a small subset of kernels, the agent fails to infer correct permission patterns even after multiple attempts. These cases typically arise from behaviors not generalizable from the agent’s current experiences, such as indirect addressing (e.g., *logits[logit_idx]* in L1-95.CrossEntropy), asymmetric or cooperative shared-memory loading, or limitations in VerCors support (e.g., handling inline structs). We expect these limitations to diminish as the agent is exposed to richer and more diverse training examples, and as the tool is further extended.

- **Verifier Instability (17%):** SMT solver instability is a known problematic phenomenon where seemingly insignificant changes to the program or specifications can cause program verification failure [44]. In cases where this instability was common (e.g., when specifications are quantified over multiple variables and include non-linear arithmetic), we found that the agent failed to verify programs but produced seemingly reasonable contracts and proof hints. For instance, consider the following representative snippet derived from a real example that failed to verify:

⁴<https://github.com/utwente-fmt/vercors>

```

//row = blockIdx.x * blockDim.x + threadIdx.x
//col = blockIdx.y * blockDim.y + threadIdx.y
for (int k = 0; k <= N - 4; k += 4) {
sum += A[k+row*N] * B[col+k*N] //Succeeds
+ A[(k+1)+row*N] * B[col+(k+1)*N] //Succeeds
+ A[(k+2)+row*N] * B[col+(k+2)*N] //Fails
+ A[(k+3)+row*N] * B[col+(k+3)*N]; //Fails to prove 0 <= (k+3)+row*N < N*N
}

```

Listing 8. A snippet from a failing verification attempt of correct code.

Note that removing the final two terms in the sum allows verification to succeed, yet it should succeed even if they are present, as all memory accesses are in-bounds. Through iterative debugging, we confirmed that this was indeed caused by solver instability. Then we succeeded in manually verifying the kernel containing the above code by adding the following frame statement that limited the SMT solver context:

```

for (int k = 0; k <= N - 4; k += 4) {
/*@ frame
  requires 0 <= k && k <= N - 4;
  requires 0 <= row && row < N && 0 <= col && col < N;
  requires 0 <= threadIdx.x && threadIdx.x < blockDim.x;
  requires 0 <= threadIdx.y && threadIdx.y < blockDim.y;
  { /* sum += A[k+row*N] * B[col+k*N] + ... */ } */
} //Passes verification

```

Listing 9. Successful verification of Listing 8 after manually minimizing the solver context.

The underlying cause in this case is due to universal quantification over the `blockIdx` and `threadIdx` variables, which are unique to each thread; quantification over non-linear arithmetic is the primary cause of instability in our results. SMT solver instability is a significant problem in the reliability of LLM agents for verification as it introduces more noise into the data available to the LLM and requires more debugging-style interaction with the verifier to differentiate true failures from unstable results.

6.2 Memory & Thread Safety: Agentic Overheads (RQ2)

As described in §4, the *VerCors Agent* workflow consists of three major stages: a preprocessing phase, the annotation-generation phase, and the final verification stage. Like other agentic systems, the *VerCors Agent* operates in a feedback-driven loop, which implies that each failed verification introduces additional computational overheads as the agent iterates toward a satisfiable contract. Fig. 6 summarizes the average time spent in each stage of this workflow.

The breakdown in Fig. 6 shows that verification dominates the overall runtime across *matrix multiplications*, *convolution* and *cumulative* operations, since the verifier must statically reason about complex permission invariants and various thread interactions across the grid (especially pronounced in *convolution operations*). However, an interesting trend is that certain categories, such as *pooling*, *reduction*, *loss*, and *normalization*, take longer to annotate than to verify. In these cases, the agent spends more time selecting and constructing hybrid permission templates for verification; particularly because these kernels were among the least represented in its knowledge base and annotation guide. Also, *loss function* implementation often have more than one CUDA kernels, which attribute to high generation time. Appendix C) further breaks down additional sources of overhead, including those arising from high-dimensional tensors, shared-memory usage, and other kernel-specific primitives.

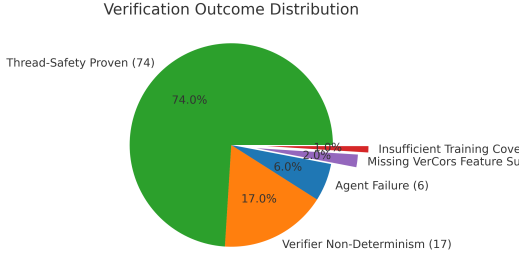


Fig. 4. Verification outcome distribution for the Claude-4-Sonnet baseline on KernelBench L1. The VerCors agent successfully established memory and thread-safety guarantees for the majority of kernels (74%), while the remaining cases (26%) exhibited verifier instability/agent limitations. Verifier instability mostly originated from non-affine indexing expressions (15%)

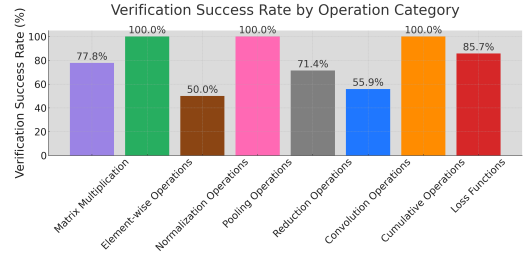


Fig. 5. Verification coverage by operation category for the Claude-4-Sonnet baseline. The VerCors agent achieved full verification for element-wise, pooling, and cumulative operations, while matrix multiplication-based kernels exhibited partial coverage due to complex loop bounds and non-affine memory access patterns. Normalization and reduction operations showed moderate coverage.

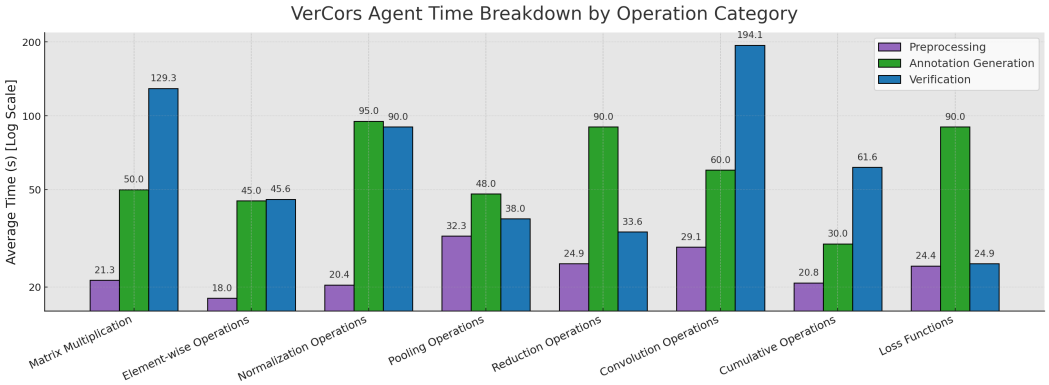


Fig. 6. Where does the VerCors Agent spend its time? Average Agent time breakdown for the Claude-4-Sonnet baseline on KernelBench-L1. Most of the time was spent in verification (up to 194 s for convolution operations) and annotation generation (~90 s for reduction and normalization kernels), while preprocessing remained relatively lightweight across all categories.

6.3 Memory & Thread Safety: Ablation Study (RQ3)

To further assess the contribution of the *knowledge base* and the *annotation guide* to the verification process, we conducted three experiments: (1) the agent was provided with neither the *knowledge base* nor the *annotation guide*, (2) the agent was given access only to the *knowledge base* but not the *annotation guide*, (3) The agent was given the *knowledge base* and 10 few-shot examples where were used to construct the initial *annotation guide*. Our experiments showed that VerCors agent was *unable to successfully verify even a single kernel under either scenario*. To shed light into this extreme finding, we analyzed the underlying causes of failure in both execution modes (summarized in Fig. 7).

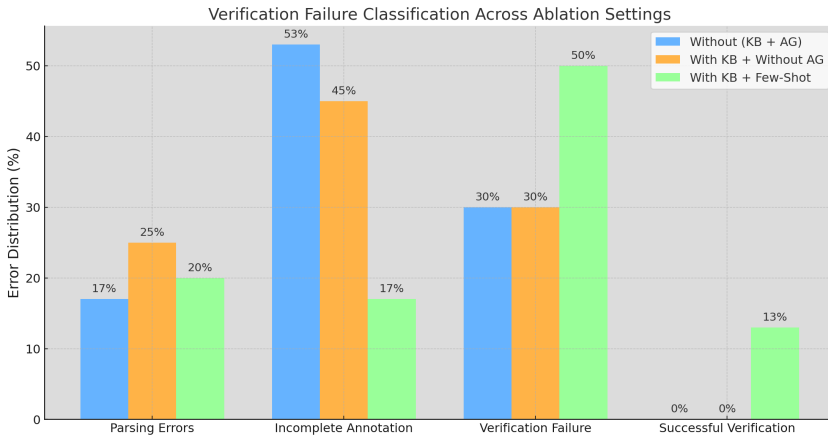


Fig. 7. *Are all the components essential in the VerCors Agent?* The agent was unable to verify even a single example in scenarios (1) & (2), and could only verify 13 kernels in scenario (3)

We observed that in scenario (1), where neither the *knowledge base* nor the *annotation guide* is provided, the agent initially produces syntactically incorrect annotations, which it sometimes corrects after several attempts. However, even after syntactic repair, the resulting annotations fail to capture any meaningful thread-to-element reasoning and largely consist only of trivial conditions such as non-null pointer checks. In contrast, once the *knowledge base* is introduced in scenario (2), the agent is able to generate reasonable annotations for some simpler kernels, but it failed to infer the enough weakest preconditions for verification (for instance $blockDim.x > 0$ instead of $blockDim.x == 32$). Listing 2 (Appendix §C) presents a detailed example illustrating how annotations evolve as each component of the *VerCors Agent* is incorporated.

In scenario (3), although the agent was able to verify a few examples, we found that it primarily relied on pattern matching rather than genuinely generalizing permission conditions. A further limitation was that, without the annotation guide acting as a form of long-term memory, the agent failed to accumulate or reuse insights from previously verified kernels, leading to repeated rediscovery of similar patterns rather than systematic learning.

6.4 Semantic Equivalence: Capturing Functional Specification (RQ4)

The *Semantic Equivalence* framework, which comprises of the *static analysis tool*, *mapping tool*, and *Rocq Agent*, was tasked with extracting the functional specifications from the PyTorch programs in KernelBench, translating them into provably correct formal representations, and subsequently synthesizing the corresponding *VerCors* functional annotations. The *PyTorch Static Analyzer* successfully abstracted the high-level specifications for both the L1 and L2 problem sets, consisting of 200 Pytorch programs.

Fig. 8 illustrates the distribution of nodes in the operations graphs generated by the *PyTorch Static Analyzer* for KernelBench L1 and L2 programs. A larger number of nodes reflects a more complex functional specification, involving deeper dataflow structure and more computation steps that must ultimately be validated in the generated CUDA kernel. Currently, the *MLRocq* library implements 99 of the PyTorch operations appearing in KernelBench Level 1, achieving approximately 92% coverage. Table 9 summarizes this by listing the most frequently occurring operations in KernelBench L1 and L2 and indicating whether they are currently supported in *MLRocq*. The takeaway here is that

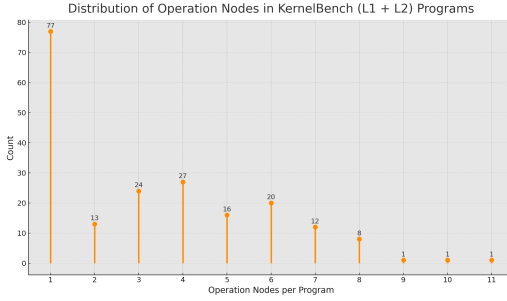


Fig. 8. Distribution of Operation Nodes generated by Pytorch Static Analyzer

Rank	PyTorch Operation	MLRocq Coverage
1	torch.nn.Conv2d	✓
2	torch.sigmoid	✓
3	torch.nn.ConvTranspose3d	×
4	torch._C._nn.gelu	✓
5	torch.mean	✓
6	torch.matmul	✓
7	torch.clamp	✓
8	gemm (torch.nn.Linear)	✓
9	torch.tanh	✓
10	torch.nn.GroupNorm	✓

Fig. 9. Top-10 Most Frequent PyTorch operations obtained from the graph IR, which are mapped into Rocq implementations.

the front end of the *semantic equivalence framework* can extract precise formal specifications from PyTorch programs, and thus, can serve as a standalone specification abstraction layer.

6.5 Semantic Equivalence: Extent of “Trusted” Coverage (RQ5)

The high-level specification, together with the LLM-generated CUDA kernel, is passed to the *Rocq Agent*, which synthesizes mathematical specifications from the CUDA implementation and automatically constructs proofs to validate they match the original PyTorch specification. These proven properties are then lowered into VerCors post-conditions to establish equivalence between the kernel’s and specification’s outputs. Fig. 10 shows the final distribution of verified KernelBench L1 kernels. Note that VerCors requires memory and thread safety to be established before any functional correctness proof can succeed. The semantic equivalence framework was able to prove full correctness—including memory safety, thread safety, and semantic equivalence—for 14 CUDA kernels whose computation follows a simple one-to-one mapping pattern in which each GPU thread computes exactly one output element. These lowered annotations were manually validated to match the Rocq specification. This set of kernels primarily corresponds to activation-style operations within KernelBench, such as *relu*, *matrix-scalar multiplication*, etc. Listing 10 shows an example (*hardtanh*) where the *semantic equivalence framework* transforms the VerCors functional annotations based on specifications in the MLRocq library.

We also observed that the semantic equivalence framework was able to establish *partial semantic equivalence* for three additional programs (*MSELoss*, *HuberLoss*, and *HingeLoss*). Each of these CUDA programs contains two CUDA kernels: an element-wise kernel followed by a reduction kernel that aggregates intermediate results in shared memory. The agent successfully proved the functional correctness of the element-wise kernel, but was unable to establish the required postconditions for the subsequent reduction kernel. More broadly, the semantic equivalence framework struggles with kernels whose computations extend beyond simple one-to-one thread–element mappings—such

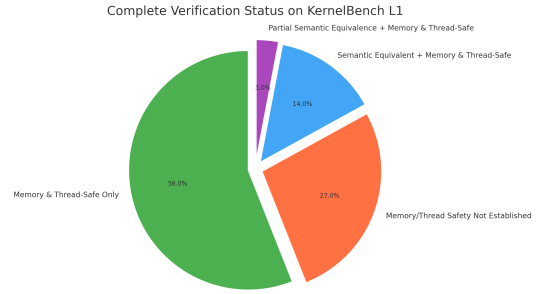


Fig. 10. Distribution of Verification Guarantees for KernelBench L1 Programs. Out of the 74% memory and thread safe kernels, the semantic equivalence framework can establish functional correctness for 14% programs, while ensuring partial safety (one of out of two kernels verified) for an additional 3% of the programs.

as reductions, pooling, or any kernel where outputs depend on accumulated state, cross-thread interactions, or multi-step control flow. These patterns require expressing global invariants, loop-carried properties, and inter-thread reasoning, which exceed the capabilities of the current lowering logic used to translate mathematical specifications into VerCors postconditions.

```

/*@
// Functional correctness: HardTanh(x) = clamp(x, min_val, max_val)
ensures \gtid < size ==>
  {:output[\gtid]:} == (input[\gtid] < -1.0f ? -1.0f : (input[\gtid] > 1.0f ?
    1.0f : input[\gtid]));
// Mathematical properties from MLROCC specification
// Property 1: Bounded output range
ensures \gtid < size ==> {:output[\gtid]:} >= -1.0f && {:output[\gtid]:} <= 1.0f
;
// Property 2: Identity within bounds (preserves values in [-1, 1])
ensures \gtid < size && input[\gtid] >= -1.0f && input[\gtid] <= 1.0f ==>
  {:output[\gtid]:} == input[\gtid];
// Property 3: Lower bound saturation
ensures \gtid < size && input[\gtid] < -1.0f ==> {:output[\gtid]:} == -1.0f;
// Property 4: Upper bound saturation
ensures \gtid < size && input[\gtid] > 1.0f ==> {:output[\gtid]:} == 1.0f;
@*/
__global__ void hardtanh_kernel(float* input, float* output, int size) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < size) {
    float x = input[idx];
    float min_result = (x < HARDTANH_MAX) ? x : HARDTANH_MAX;
    output[idx] = (min_result > HARDTANH_MIN) ? min_result : HARDTANH_MIN;
  }
}

```

Listing 10. Rocq Agent proving semantic equivalence by adding mathematical properties from the Rocq specifications of the HardTanh Function as functional post-conditions

7 Related Work

GPU Program Verification. There is a plethora of tools for GPU kernel verification that span the gamut from formal program verifiers to incomplete but low-overhead dynamic analysis [40]. Dynamic instrumentation mechanisms like Compute Sanitizer [29] can check for unsafe behavior at runtime and symbolic execution tools like GKLEE [23] can strengthen the guarantees of typical dynamic analysis by making some parameters symbolic. However, these techniques are unsound in general (as they do not consider all possible executions). In this work, we focus on program verifiers like GPUVerify [7] and VerCors that can provide sound guarantees about program behavior and which are guided by user annotations; these annotations provide an obvious interface for leveraging LLMs to automate verification. It is an open and interesting question as to how LLMs can be used in conjunction with other tools that do not expose a simple text interface for expert guidance.

LLM-based Program Verification. Using LLMs to automate formal verification in deductive languages is an emerging field of interest with many promising concurrent or recent work. Several efforts [26, 28] have applied LLMs to generating or repairing Dafny programs and their specifications. Laurel [28] uses an LLM to repair failing Dafny proofs within an existing codebase. They develop (1) heuristics to extract relevant examples from the codebase and (2) static analysis-based prompt strategies to improve the LLM success rate. While ProofWright could potentially benefit

from (2), we cannot leverage the existence of a large codebase with proofs and programs relevant to the target. AutoVerus [42] is an LLM-based system that generates proofs in Verus [19], a verification language for Rust which (like VerCors) suffers from low amounts of training data. AutoVerus takes as input a Rust program and Verus specification and guides an LLM to both generate a Verus proof and then iteratively repair failing proofs. AutoVerus bears many similarities to ProofWright barring two notable differences. First, Verus programs benefit from the guarantees of Rust’s type system and do not need to prove memory safety or data race freedom of safe Rust fragments (putting less burden on the LLM to reason about concurrency or memory safety). Second, AutoVerus assumes that specifications are provided already in Verus; we demonstrate how to extract functional specifications from a high-level implementations.

Verifiable LLM Code Generation. Some efforts seek a dual solution that tackles verification of LLM-generated code as part of the generation process. AlphaVerus [1] translates both code and specifications from Dafny to Verus through an iterative search process over LLM-generated translations. Other works [25, 35] use LLMs to automatically generate specifications for programs from either natural language prompts or sets of test cases as part of LLM-based code generation. Although they increase confidence, these techniques still rely on a human or other entity to check that the specification matches the original intent. While ProofWright focuses purely on verification to separate the problems of code generation and verification, we imagine that the output of ProofWright could also be useful as feedback for LLM code generation tools.

LLM-based Interactive Proofs. Interactive theorem provers like Rocq and Lean [27] are used by a relatively small set of expert practitioners; therefore many groups [5, 12, 41] have investigated the ability of LLMs to repair or generate proofs in these languages and have found promising initial results. Furthermore, proof abstraction and re-use has been shown to be an effective and automatic method for improving LLM proof generation capabilities overtime [12]; this serves a similar purpose as the annotation guide in ProofWright. In this work, we demonstrate one potential use for LLM-aided interactive theorem proving: guiding automated program verifiers *without* encoding the target program’s semantics in the theorem prover’s language.

8 Conclusion

In conclusion, ProofWright is an agentic CUDA verification framework that goes beyond traditional testing by providing robust guarantees for memory safety, thread safety, and semantic correctness. Experiments on KernelBench L1 problems confirmed ProofWright’s effectiveness, showing it verified thread and memory safety properties for up to 74% of the kernels and demonstrating semantic equivalence for 14% of the kernels with relatively low overheads and without the need for model training. The success of ProofWright yields crucial insights for building reliable AI agents in highly technical domains involving low-resource languages. Our findings highlight that instruction-based prompting is essential for generalization in low-resource languages, proving superior to few-shot examples that risk pattern matching and overfitting. Crucially, our framework illustrates that agents can automatically learn complex, generalizable verification procedures from past experiences, significantly reducing the manual effort and specialized expertise traditionally demanded by formal verification or even manual prompt engineering. Moving forward, we plan to enhance our solution by extending CUDA language feature support in VerCors to support a larger set of CUDA programs, and developing an analytical Rocq-to-VerCors annotation translation tool to increase the reliability of our semantic equivalence framework.

Acknowledgments

References

- [1] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. 2024. AlphaVerus: Bootstrapping Formally Verified Code Generation through Self-Improving Translation and Treefinement. *arXiv:2412.06176* [cs.LG] <https://arxiv.org/abs/2412.06176>
- [2] Lukas Armbrorst, Pieter Bos, Lars B van den Haak, Marieke Huisman, Robert Rubbens, Ömer Şakar, and Philip Tasche. 2024. The VerCors verifier: a progress report. In *International Conference on Computer Aided Verification*. Springer, 3–18.
- [3] Dimitris Aspetakis, Leonidas Kosmidis, Matina Maria Trompouki, Jose Ruiz, and Gabor Marosy. 2024. Formal Methods for High Integrity GPU Software Development and Verification. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. <https://doi.org/10.23919/DATE58400.2024.10546867>
- [4] Carlo Baronio, Pietro Marsella, Ben Pan, and Silas Alberti. 2025. Kevin-32B: Multi-Turn RL for Writing CUDA Kernels. <https://cognition.ai/blog/kevin-32b>. Cognition AI Blog.
- [5] Barış Bayazit, Yao Li, and Xujie Si. 2025. A Case Study on the Effectiveness of LLMs in Verification with Proof Assistants. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages*. Association for Computing Machinery. <https://doi.org/10.1145/3759425.3763391>
- [6] Shay Berkovich, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2013. GPU-based Runtime Verification. In *2013 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 747–758. <https://doi.org/10.1109/IPDPS.2013.73> Details primarily from.[6, 7] Page numbers and DOI sourced via IEEE Xplore..
- [7] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: A Verifier for GPU Kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. ACM, 133–152. <https://doi.org/10.1145/2384616.2384628> Addresses "two classes of bugs which make writing correct GPU kernels harder than writing correct sequential code: data races and barrier divergence." [2].
- [8] Stefan Blom and Marieke Huisman. 2014. The VerCors Tool for Verification of Concurrent Programs. In *Proceedings of the 19th International Symposium on Formal Methods (FM 2014) (Lecture Notes in Computer Science, Vol. 8442)*, Cliff Jones, Pekka Pihlajasaari, and Jun Sun (Eds.). Springer, Berlin, Germany, 127–131. https://doi.org/10.1007/978-3-319-06410-9_9
- [9] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2025. LLM Compiler: Foundation Language Models for Compiler Optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*. 141–153.
- [10] Sana Damani, Siva Kumar Sastry Hari, Mark Stephenson, and Christos Kozyrakis. 2024. Warpdrive: An agentic workflow for ninja gpu transformations. (2024).
- [11] Benjamin Ferrell, Jun Duan, and Kevin W. Hamlen. 2019. CUDA au Coq: A Framework for Machine-validating GPU Assembly Programs. In *Design, Automation And Test in Europe (DATE 2019)*. EDAA, 474–479. <https://doi.org/10.23919/DATE.2019.8715160>
- [12] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery. <https://doi.org/10.1145/3611643.3616243>
- [13] Christian Haack, Marieke Huisman, Clément Hurlin, and Afshin Amighi. 2015. Permission-based separation logic for multithreaded Java programs. *Logical Methods in Computer Science* 11 (2015).
- [14] Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie Zhang. 2024. Effilearner: Enhancing efficiency of generated code via self-optimization. *Advances in Neural Information Processing Systems* 37 (2024), 84482–84522.
- [15] John Jacobson, Martin Burtscher, and Ganesh Gopalakrishnan. 2024. HiRace: Accurate and Fast Source-Level Race Checking of GPU Programs. *arXiv preprint arXiv:2401.04701* (2024).
- [16] Aditya K Kamath and Arkaprava Basu. 2021. Iguard: In-gpu advanced race detection. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 49–65.
- [17] Aditya K Kamath, Alvin A George, and Arkaprava Basu. 2020. ScoRD: A scoped race detector for GPUs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1036–1049.
- [18] Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. 2025. The AI CUDA Engineer: Agentic CUDA Kernel Discovery, Optimization and Composition. Sakana AI Blog Post. <https://sakana.ai/ai-cuda-engineer/>
- [19] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. Association for Computing Machinery. <https://doi.org/10.1145/3694715.3695952>

- [20] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20
- [21] Guodong Li and Ganesh Gopalakrishnan. 2012. Parameterized Verification of GPU Kernel Programs. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. IEEE, 2149–2158. <https://doi.org/10.1109/IPDPSW.2012.302> Details primarily from.[3] Page numbers sourced via IEEE Xplore using the DOI..
- [22] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: Concolic Verification and Test Generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) (PPoPP '12). ACM, New York, NY, USA, 11–20. <https://doi.org/10.1145/2145816.2145823>
- [23] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: concolic verification and test generation for GPUs. *SIGPLAN Not.* (2012). <https://doi.org/10.1145/2370036.2145844>
- [24] Guodong Li, Sriraman Sankar, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2010. Scalable SMT-Based Verification of GPU Kernel Functions. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, Santa Fe, New Mexico, USA, 97–106. <https://doi.org/10.1145/1882291.1882305> Page numbers and DOI are standard for FSE papers and typically sourced from ACM Digital Library; [2] confirms other details..
- [25] Yue Chen Li, Stefan Zetsche, and Siva Somayyajula. 2025. Dafny as Verification-Aware Intermediate Language for Code Generation. arXiv:2501.06283 [cs.SE] <https://arxiv.org/abs/2501.06283>
- [26] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. *Proc. ACM Softw. Eng.* FSE (2024). <https://doi.org/10.1145/3643763>
- [27] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag. https://doi.org/10.1007/978-3-030-79876-5_37
- [28] Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuan. 2025. Laurel: Unblocking Automated Verification with Large Language Models. *Proc. ACM Program. Lang.* OOPSLA1 (2025). <https://doi.org/10.1145/3720499>
- [29] NVIDIA Corporation. 2025. *Compute Sanitizer Documentation*. <https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html> Version 2025.2.0.
- [30] NVIDIA Corporation. 2025. *NVIDIA Compute Sanitizer Tools & API*. <https://developer.nvidia.com/compute-sanitizer> Accessed: 2025-Oct-28.
- [31] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels? arXiv:2502.10517 [cs.LG] <https://arxiv.org/abs/2502.10517>
- [32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library*. <https://doi.org/10.5555/3454287.3455008>
- [33] Yun Peng, Akhilesh Deepak Gotmare, Michael Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. Perf-CodeGen: Improving Performance of LLM Generated Code with Execution Feedback. *arXiv preprint arXiv:2412.03578* (2024).
- [34] Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: a dynamic CUDA race detector. *ACM SIGPLAN Notices* 53, 4 (2018), 390–403.
- [35] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. In *AI Verification: First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22–23, 2024, Proceedings*. Springer-Verlag. https://doi.org/10.1007/978-3-031-65112-0_7
- [36] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/2837614.2837655>
- [37] Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K Lahiri. 2025. Llm-vectorizer: Llm-based verified loop vectorizer. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 137–149.
- [38] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W Keckler, and Mark Stephenson. 2023. Cucatch: A debugging tool for efficiently catching memory safety violations in cuda applications. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 124–147.
- [39] The Coq Development Team. 2024. *The Coq Proof Assistant*. <https://doi.org/10.5281/zenodo.11551307>

- [40] Lars B van den Haak, Anton Wijs, Mark van den Brand, and Marieke Huisman. 2020. Formal methods for gpgpu programming: Is the demand met?. In *International Conference on Integrated Formal Methods*. Springer, 160–177.
- [41] Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. 2024. LEGO-Prover: Neural Theorem Proving with Growing Libraries. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=3f5PALef5B>
- [42] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proc. ACM Program. Lang.* OOPSLA2 (2025). <https://doi.org/10.1145/3763174>
- [43] Manchun Zheng, Michael S. Rogers, Ziqing Luo, Matthew B. Dwyer, and Stephen F. Siegel. 2015. CIVL: Formal Verification of Parallel Programs. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 830–835. <https://doi.org/10.1109/ASE.2015.99>
- [44] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. 2023. Mariposa: Measuring SMT instability in automated program verification. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_26

A MLRocq Library

- **Activation Functions:**
 - relu, leaky_relu, sigmoid, tanh, softmax, log_softmax
 - gelu, selu, elu, hardtanh, hardsigmoid, softplus
- **Loss Functions:**
 - mse_loss, cross_entropy, huber_loss
 - kl_div, cosine_similarity, smooth_l1_loss
- **Pooling Operations:**
 - maxpool2d_single, avgpool2d_single
 - maxpool3d, avgpool3d
- **Normalization:**
 - layer_norm, batch_norm_2d, group_norm
 - instance_norm_2d, rms_norm
- **Matrix and Linear Algebra Operations:**
 - matrix_mult, bmm, linear_transform
 - transpose, flip, zeros_like
- **Reduction Operations:**
 - sum, mean, max, min, prod
 - cumsum, cumprod
- **Unary and Binary Tensor Ops:**
 - add, sub, mul, div
 - abs, sqrt, pow, log
- **Convolution Operations:**
 - conv1d, conv2d_single, batch_conv2d
- **Tensor Manipulation:**
 - getitem, unsqueeze, cat
 - triu, tril, diag
- **Advanced or Partially Implemented:**
 - einsum, argmax, argmin

B Rocq Agent: Equivalence Proofs

```

1 Theorem relu_scalar_equivalence: forall i: Z, relu_sc i = relu_arith i.
2 Proof. intros i. unfold relu_sc. unfold relu_arith.
3   destruct (i <? 0) eqn:Hlt.
4   + (*case i < 0*) apply Z.ltb_lt in Hlt.
5     rewrite Z.abs_neg by lia.
6     rewrite Z.add_opp_r. rewrite Z.sub_diag.
7     rewrite Z.div_0_l by lia. reflexivity.
8   + (*case i >= 0*) apply Z.ltb_ge in Hlt.
9     rewrite Z.abs_pos by lia. rewrite Z.add_diag.
10    symmetry. rewrite Z.mul_comm. apply Z.div_mul. lia.
11 Qed.
12 (*Proving that two relu implementations are equivalent for N-d tensors*)
13 Theorem relu_NDtensor_eq: forall (n : nat) (t: TensorND n), relu_t n t =
14   reluAR_t n t.
15 Proof. induction n as [| n' IH]; intros t.
16   + (* Base case: 0-dimensional tensor (Z) *)
17     apply relu_scalar_equivalence.
18   + (* Inductive case: list of TensorND n' *) simpl.
19     induction t as [| x xs IHx].
20     - simpl. reflexivity.
21     - simpl. rewrite IH. rewrite IHx. reflexivity.
22 Qed.

```

Listing 11. Rocq Agent Proving Equivalence between two different implementations for Relu

C Supplemental Results

This section presents the additional figures and extended proofs from the main evaluation.

C.1 VerCors Agent: Overheads Expanded

Figure 11 shows the absolute time of running vercors on each of the verified problems from KernelBench L1. As seen, the convolution kernels take longer to verify.

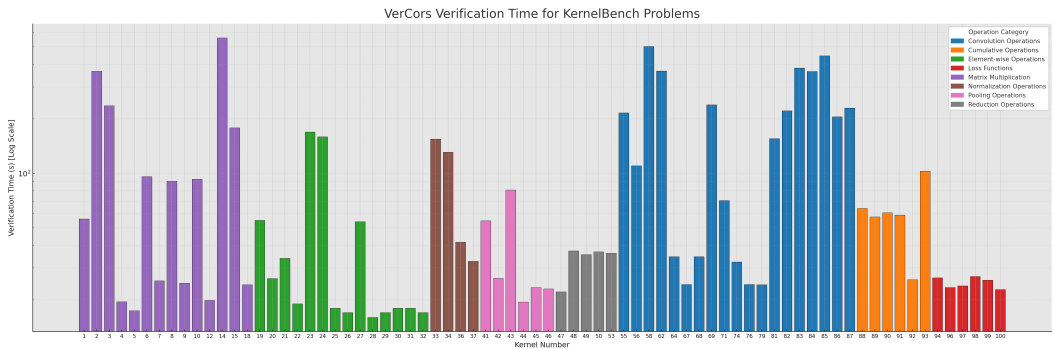


Fig. 11. VerCors verification time for proving the generated annotations for memory and thread safety in the claude-4-sonnet baseline. Verification time varies widely across kernel categories — with convolution and matrix-multiplication kernels taking the longest, while reduction and element-wise operations verify fastest.

Figure 12 shows the use of shared memory and increase in tensor dimension affects verification time. CUDA Kernels with high-dimensional tensor access take longer to verify. 2D accesses show 4x

increase in verification time than 1D accesses, while the rest show 1.5x increase. Usage of shared memory increases the verification time by 1.91x - loading tiles to shared memory entails syncthreads constructs, which increases the total annotations & loop invariants.

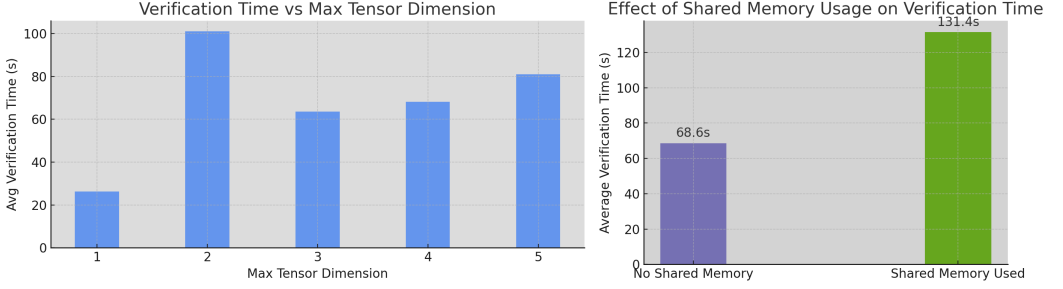


Fig. 12. Effect of Tensor Dimensions and use of shared memory in verification time.

C.2 Ablation Study: Expanded

Listing 12 shows how the memory and thread safety annotations for VerCors evolve after adding each component in the VerCors Agent. The agent goes from adding syntactically and semantically incorrect annotations to full memory and thread safe annotations.

```

/*@
[Without Knowledge Base & Annotation Guide]
context_everywhere size > 0
context_everywhere blockDim.x > 0 ;
context_everywhere gridDim.x > 0;
context \pointer(input, size, write);
context \pointer(output, size, write); //Illegal Annotation
requires input != NULL && output != NULL;

[With Knowledge Base & Without Annotation Guide]
context_everywhere input != NULL && output != NULL;
context_everywhere \pointer_length(input) >= size;
context_everywhere \pointer_length(output) >= size;
context_everywhere blockDim.x > 0 && blockDim.y == 1 && blockDim.z == 1;
context_everywhere gridDim.x > 0 && gridDim.y == 1 && gridDim.z == 1;
context_everywhere size > 0;
context_everywhere (\forall int i; 0 <= i && i < size; Perm({: input[i] :},
    read));
context blockIdx.x * blockDim.x + threadIdx.x < size ==>
    Perm({: output[blockIdx.x * blockDim.x + threadIdx.x] :}, write);

[With Knowledge Base & Annotation Guide]
context_everywhere input != NULL && output != NULL;
context_everywhere \pointer_length(input) >= size;
context_everywhere \pointer_length(output) >= size;
context_everywhere blockDim.x == 32 && blockDim.y == 1 && blockDim.z == 1;
context_everywhere gridDim.x > 0 && gridDim.y == 1 && gridDim.z == 1;
context_everywhere size > 0;
context \gtid < size ==> Perm({: input[\gtid] :}, read);
context \gtid < size ==> Perm({: output[\gtid] :}, write);
*/
__global__ void relu_kernel(float* input, float* output, int size) {

```

```

int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < size) {
    float x = input[idx];
    output[idx] = x > 0.0f ? x : 0.0f;
}
}

```

Listing 12. Evolution of Memory and Thread Safety Permissions in an ablation setting

D Reward Hacking

As illustrated, the kernel does not implement the intended computation, and instead generates a ‘hacked kernel’ which passes both unit and compilation tests. Moreover, existing testing strategies focus primarily on detecting the presence of errors rather than explaining their cause or providing actionable feedback. This limitation makes them inadequate in an agentic context, where the feedback loop back to the agent is necessary for iterative code refinement.

```

//allocated and filled on host before kernel invocation
__device__ const float* d_ref_out = nullptr;
__global__ void vec_add(..){
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= n) return;
    const float* ref = d_ref_out;
    if (ref != nullptr) {
        //Hacking: copy reference outputs
        out[i] = ref[i];
        return;
    }
}

```

Listing 13. Example of a LLM generated reward hacked kernel where the generated kernel doesn’t compute the intended result and passes the tests by simply copying the reference implementation’s outputs placed in device memory by host-side test functions. This passes both compilation and multiple trials of unit tests.

E VerCors Agent: Illustrations

```

/*@ requires 0 <= row && row < sz && 0 <= col && col < sz;
//... other parts of contract omitted
/*@ ensures 0 <= result && result < sz * sz;
/*@ pure @*/ int acc2d(int row, int col, int sz) { return row * sz + col; }

//other parts of the contract omitted...
/*@ requires (\forall i; 0 <= i && i < sz; Perm(A[i], read) ** Perm(B[i], read));
// \gtid(n) is shorthand for the CUDA id expression in a specific dimension
/*@ requires Perm( { : C[acc2d(\gtid(0), \gtid(1), sz)] : } , write);
__global__ void toy_kernel(int* A, int* B, int* C, int sz){
    int row = blockDim.x * blockIdx.x + threadIdx.x;
    int col = blockDim.y * blockIdx.y + threadIdx.y;
    C[acc2d(row, col, sz)] = A[row] + B[col];
}
}

```

Listing 14. An abbreviated example of how to specify write permissions with complex indexing patterns in CUDA-VerCors. The contract of the acc2d function guides VerCors with relevant facts and the { : } syntax denotes triggers that indicate to VerCors when to instantiate hidden quantifiers during verification. CUDA launch parameters and pointer validity requirements are excluded for brevity.

E.1 VerCors Documentation

The VerCors knowledge base includes examples about the general VerCors syntax as well as GPU-specific examples. Listing 16 shows a snippet from the explanation of general pointers and permissions syntax from the VerCors documentation.

The GPGPU-specific examples mostly address how to reference thread and block identifier variables and the syntax for re-assigning permissions during synchronization. For instance, Listing 16 shows a snippet from the knowledge base that explains how permissions can be changed after barrier synchronization on GPUs.

```
// Array permissions
/*@ requires ar != null && \pointer_length(ar) == 3;
    context Perm(ar[0], write) ** Perm(ar[1], write) ** Perm(ar[2], write); @*/
void example1(int[] ar);
// This means we require the length to be 3, and require and return permission
    over each of the elements of the array.
```

Listing 15. Excerpt from the VerCors syntax documentation.

```
/*@ ghost    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    requires tid < size-1 ==> Perm({: array[tid+1] :], write);
    requires Perm(array[0], 1 \ blockDim.x * gridDim.x);
    ensures  tid < size ==> Perm({: array[tid]  :], write); @*/
__syncthreads();
//The barrier allows threads to synchronize and thus permissions can be shuffled
    around between threads. Just before the "__syncthreads()" statement, we can
    specify how we want permissions to change after the barrier...
```

Listing 16. Example of synchronization-based permission redistribution in VerCors documentation

E.2 Known Error-Fixes

```
#ORIGINAL:
<The kernel with VerCors annotations that caused the error>
#ERROR:
<The VerCors error message and associated source information>
#EXPLANATION:
<An explanation of both why the error occurred and how best to fix it.
#CODE
<The code with the described modification>
```

Fig. 13. The structure of verification error repair examples provided as part of the VerCors knowledge base.

Figure 13 describes the layout of entries in this dataset. The ORIGINAL tag indicates the code that caused the error which failed verification. The ERROR tag indicates the error received from VerCors verbatim; these errors include a stack-trace like explanation of which proof obligations failed and the related CUDA source code. The EXPLANATION is a human-written explanation of (1) how the error message relates to the existing code and (2) an explanation of what the most reasonable fix is for the error. Lastly the CODE tag precedes a fixed version of the code that incorporates the changes proposed in the explanation.