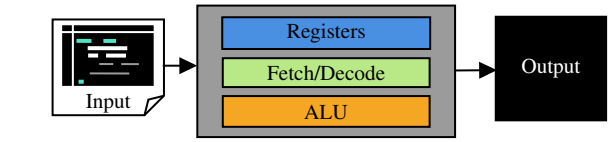
Semantic-Guided Test Generation using Fine-Tuned LLMs for Validation of Hardware Accelerators

Emma Andrews, Aruna Jayasena, and Prabhat Mishra
University of Florida, Gainesville, FL, USA

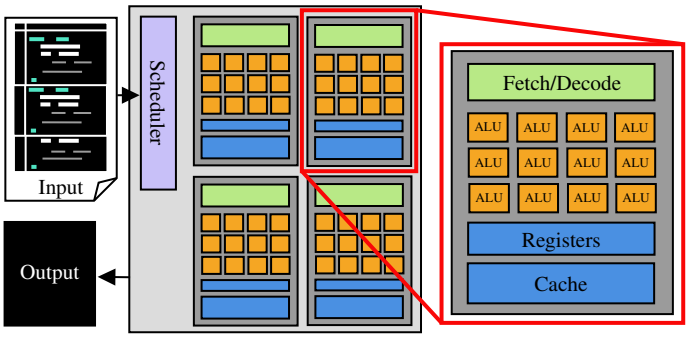
***Abstract*—The increasing complexity and heterogeneity of programmable hardware accelerators, such as Graphics Processing Units (GPUs) and Tensor Processing Units (TPU), pose a significant challenge for automated test generation and functional validation. Traditional validation techniques often struggle to scale with architectural diversity and cannot effectively exploit the semantic relationships between instructions and data. Validation using large language models (LLMs) is a promising avenue for generating assembly programs (test vectors) for processor verification since LLMs are trained with diverse general-purpose processor designs. Unfortunately, LLMs are unsuitable for validation of programmable hardware accelerators since there is a lack of training data for such implementations. In this paper, we propose an automated framework that fine-tunes LLMs to generate semantically correct test cases directed toward improved design coverage while monitoring the functional correctness of the outputs. The generated test cases are evaluated by a compiler for correctness before using them for validation of hardware accelerators. We facilitate a mechanism for the LLM to observe the design coverage on the implementation based on the previously generated test patterns. Extensive experimental evaluation demonstrates that our framework can achieve 33% improvement in design coverage compared to state-of-the-art test generation with the added advantage of monitoring the functional correctness of the design. Our framework has identified several functional bugs in the open-source tiny-gpu implementation.**

I. Introduction

Programmable hardware accelerators, which extend beyond traditional general-purpose computing, offer a promising approach to building reconfigurable accelerators while preserving overall system flexibility [1]. Cryptographic instruction sets (CISE) [2], Graphics Processing Units (GPU) [3], and Tensor Processing Units (TPU) [4], all of which rely on dedicated instruction sets executed by specialized hardware functional units through custom machine code (firmware). Usually these machine codes are generated via programming frameworks, such as CUDA [5] and ROCm [6] for GPUs, XLA [7] for TPUs, and OpenCL [8] for cross-platform acceleration, to enable execution on specialized hardware functional units. Figure 1 illustrates the architectural differences between general-purpose computing and application-specific harwdare acceleration. Figure 1a shows a typical processor (CPU) design, which includes memory elements (e.g., registers), an instruction processing unit (fetch/decode), and functional units, such as Arithmetic Logic Units (ALUs). In contrast, Figure 1b depicts a GPU architecture, featuring a hardware scheduler and multiple processing cores. Each core contains its own instruction processing unit, local memory elements (registers and cache), and multiple ALUs for parallel execution.



(a) General-purpose computing using CPUs



(b) Application-specific hardware acceleration using GPUs

Fig. 1: Comparison between (a) general-purpose computing architectures and (b) programmable hardware platforms.

Verifying heterogeneous hardware implementations is challenging due to the interaction between the custom hardware and the reconfigurability introduced through the machine code. With recent advances in Large Language Models (LLMs) [9], research efforts have demonstrated their potential in validating general-purpose processors by leveraging well-defined instruction set architecture (ISA) specifications [10]–[12]. In these efforts, LLMs are able to generate input programs, in some cases with minimal modifications, often outperforming traditional constrained-random techniques and requiring significantly less manual effort than directed testing. However, LLMs are not suitable for the validation of programmable hardware accelerators for several reasons. First, due to the application-specific nature of hardware accelerators, formal specifications (training data) are often unavailable. As a result, LLMs lack the semantic understanding of the instruction-data relationships unique to these implementations. Second, unlike general-purpose processors where thread and scheduling mechanisms are managed by the operating system kernel, many accelerator systems implement these mechanisms directly in hardware. This hardware-level control must be captured in the LLM's prompting or training to generate valid programs, including instruction sequences and data layouts. Moreover, accelerator systems often involve complex execution pipelines, such as host-device coordination or hardware-managed multistage

workflows, which require awareness of multimodal execution semantics. Furthermore, many validation-relevant behaviors in these systems are closely tied to micro-architectural details, such as memory hierarchies, caching policies, or specialized functional units, that are not visible from the software or instruction stream alone. Without explicit modeling or access to such low-level details, LLMs are likely to miss subtle but critical corner cases during test generation.

A. Programmable Logic in Hardware Accelerators

Traditional hardware design has largely revolved around general-purpose processors like CPUs and Microcontroller Units (MCUs), and standalone hardware modules with well-understood, often monolithic, execution models. These designs typically follow deterministic control-flow structures and adhere to standardized instruction sets, making them more amenable to existing test generation and validation techniques [13]. In contrast, hardware accelerators, such as GPUs, vector engines [14], [15], and custom instruction set extensions [2], [16], [17], exhibit complex execution behaviors in various domains, including cryptography and artificial intelligence. These may include wide SIMD parallelism [18], asynchronous execution, memory model variations [19], and tightly coupled instruction-data semantics [20] that are context-dependent. Moreover, accelerator components are designed to interact with general-purpose cores, co-processors, and accelerators in loosely or tightly integrated pipelines, leading to non-uniform behavior across the system. They rely on architecture-specific behaviors and optimizations and may not follow standardized ISA conventions.

B. Validation of Hardware Accelerators

Validation of programmable hardware accelerators require new test generation strategies that go beyond syntax or control flow and instead focus on architectural intent, data transformation patterns, and instruction dependencies. Traditional test generation techniques, such as random or constraint-random methods are not designed to handle such cases. They often miss important behaviors because they do not model the specific ways these systems execute instructions and move data. As a result, validation frameworks must be able to model and stimulate these heterogeneous execution pathways, which general-purpose techniques are not built to handle.

Verification becomes even harder when both the machine code (firmware) and the hardware are customized. One major challenge is how to generate valid test cases that follow the specification of the custom hardware. Writing such test cases manually is time-consuming, especially when the machine code includes custom instructions that do not follow standard ISA rules. Another problem is how to improve the quality of test cases so they can reach complex corner cases while monitoring the functional correctness of the implementation. Random and constrained-random test generation often fail to expose these cases because they are not guided by architecture-specific knowledge. Existing tools and methods cannot be reused easily in this setting because they were designed for standard processors and assume uniform behavior, which is not true for heterogeneous hardware accelerators.

C. Contributions

To address these challenges, we introduce Semantic-Guided Test Generation (*SGTG*), a framework that fine-tunes LLMs on previously unknown architectures to generate test cases aimed at maximizing design coverage and ensuring functional correctness in hardware implementations. Specifically, in this paper, we make the following major contributions:

- Developed a framework to incorporate the hardware prototype into a verification model that can be evaluated for the design coverage as well as functional correctness.
- Proposed a method to fine-tune LLMs with the knowledge of the hardware to generate valid input programs.
- Designed an automated compilation method that verifies the semantics of the generated programs and converts them into machine code.
- Automated feedback based on the design coverage in the form of a prompt engineering approach to improve the LLM's generated code output for the next iteration.

The remainder of the paper is organized as follows. Section II provides necessary background and surveys related efforts. Section III describes our test generation framework. Section IV presents the experimental results. Finally, Section V concludes the paper.

II. Related Work

LLMs are a variant of transformer models [21] designed to generate text in response to input prompts. Depending on the specific LLM, these prompts can take many forms, with conversational chat interfaces being among the most common. LLMs encode a vast knowledge base during pre-training by learning statistical patterns and associations from large-scale text corpora. Although this knowledge is general-purpose, task-specific behavior can be guided or enhanced using carefully constructed prompts—a process often referred to as "prompt engineering" or "in-context learning", which allows the model to specialize for tasks such as test generation, specification interpretation, or hardware-aware reasoning without retraining. In this section, we first review application of LLMs for implementing hardware designs. Next, we survey prior research efforts in using LLMs for hardware validation.

A. Hardware Design and Validation using LLMs

LLMs are widely used for generating hardware implementations. Wang et al. [22] propose ChatCPU, an agile platform that integrates large language models into the early stages of RISC-V CPU development. Bhandari et al. [11] and Firouzi et al. [23] investigate LLM-assisted hardware generation, where prompts help generate Verilog modules from natural language descriptions. Similarly, Huang et al. [24] and Liu et al. [25] propose frameworks that use LLMs to automate RTL generation and provide suggestions for hardware design, offering varying degrees of interactivity and customization through prompt templates. Ma et al. [26] extends this idea to parsing

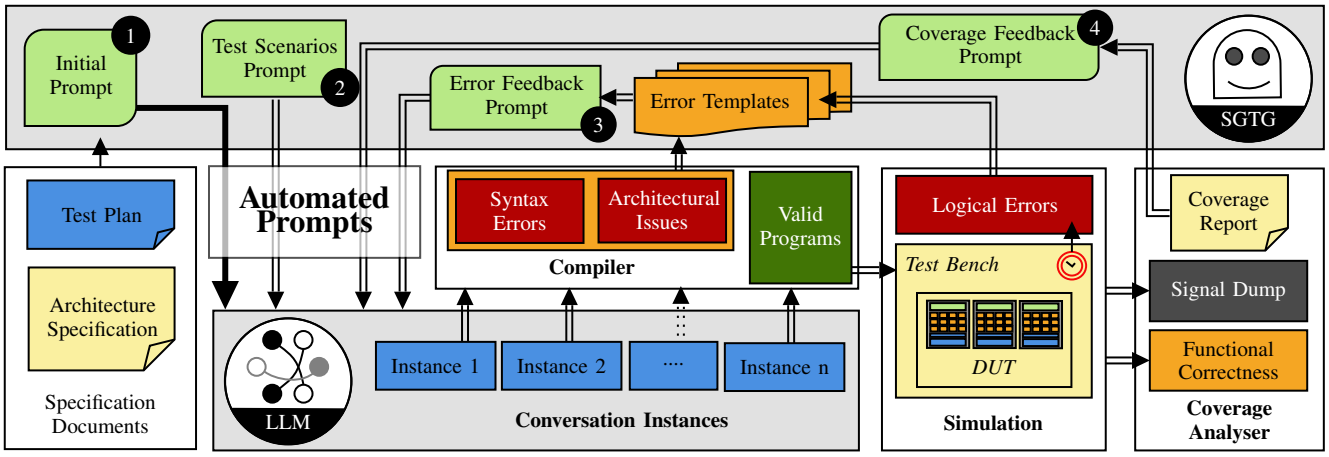


Fig. 2: Overview of our proposed methodology. Conversations are carried out between the SGTG framework and the LLM as multiple conversation instances. *Initial Prompt* is common for all the instances, while *Test Scenarios Prompts* initialize each of the conversations. Interactions between conversation instances are outlined by $\Longrightarrow$, while other interactions are outlined by $\rightarrow$. *Error Feedback* and *Coverage Feedback* prompts are continuously applied until the design coverage is met.

and understanding Verilog code using LLMs, enabling natural language queries over hardware modules. There are recent efforts using prompt engineering with LLMs for test generation for general-purpose hardware implementations. Deng et al. introduce LLM-TG [10], an automated test generation framework that utilizes large language models, particularly GPT-4 [9], for validating processor designs. While this approach is promising for validating known processor architectures, the automated prompting is not helpful when dealing with unknown architectures (e.g., GPU with new code syntax). Hassan et al. [27] presented an approach that integrates LLMs into formal verification by coupling them with mutation testing techniques. AutoBench can automatically generate Verilog testbenches based on the description of an RTL design [28]. However, Verilog testbenches are not suitable for accelerator architectures that need to run assembly programs.

B. Limitations of Existing Test Generation Efforts

While existing test generation techniques have shown promise in generating input programs for validating general-purpose processor designs, they are not suitable for validation of programmable hardware accelerators for the following reasons. Most existing approaches focus on single-core processors and do not extend to multicore architectures or parallel execution environments. Moreover, directly applying existing LLM-based techniques to heterogeneous hardware validation has limited effectiveness. This is primarily due to the lack of formal, standardized specifications for application-specific hardware accelerators, which prevents LLMs from learning or reasoning about the nuanced instruction-data semantics. Moreover, while threads are handled by operating systems in general-purpose processors, modern accelerators often implement thread scheduling and management at the hardware level, requiring this behavior to be explicitly captured through prompts or auxiliary context. The complexity further increases with host-device coordination, hardware-managed pipelines, and specialized microarchitectural features such as memory hierarchies or caching policies. These factors introduce subtle behaviors that are difficult for LLMs to infer without explicit modeling. As a result, without a customized prompting strategy or architectural guidance, existing LLM-based methods are not able the generate valid input (test) programs. As demonstrated in Section IV, even if existing efforts generate valid test cases, they cannot cover corner cases, leading to inferior coverage of heterogeneous hardware platforms.

III. Semantic-Guided Test Generation using LLMs

Figure 2 provides an overview of our proposed methodology for validating programmable hardware accelerators. Our framework assumes the availability of the specification document containing the architectural design details and the initial test plan. We construct the **Initial Prompt** (denoted as ①) based on the architectural specification. Similarly, we create multiple conversation instances based on the test plan, each conversation is isolated from the others and assigned a distinct task using **Test Scenarios Prompts** (denoted as ②). Each conversation instance generates input assembly programs targeting the design under test. The generated assembly test cases are passed to a custom compiler, which analyzes the code for syntax and semantic errors. If any issues are found, an **Error Feedback Prompt** (denoted as ③) is generated using predefined templates; otherwise, the compiler translates the code into machine instructions. The compiled program is then simulated on the hardware design, during which both coverage and functional correctness are assessed. Based on the coverage results, we generate **Coverage Feedback Prompts** (denoted as ④), which guide the LLM to improve the generation of future test programs and enhance design coverage. The remainder of this section describes each of these steps in detail.

A. Construction of Initial Prompts based on Specification

LLMs typically possess general knowledge of assembly instructions for general-purpose architectures, such as x86,

ARM, and RISC-V. However, as discussed in Section II, they lack awareness of the specialized characteristics of heterogeneous accelerators. Therefore, to enable accurate code generation, it is necessary to introduce the LLM to the programmable hardware accelerators and their instruction sets.

Since the LLM begins with no prior knowledge of the target architecture, the initial prompt must establish all essential architectural features and include representative examples of the code syntax. Given the model's familiarity with conventional ISAs, it is crucial that the prompt clearly emphasizes the unique nature of the target architecture, explicitly outlining only the valid components, instruction semantics, and data conventions, to prevent confusion with known general-purpose systems. For this purpose, we construct the initial prompt such that it contains the following four attributes.

Design Purpose: Heterogeneous computing architectures are designed for specific application domains to accelerate targeted computations. As a result, the initial prompt must clearly communicate the design intent and distinguish it from general-purpose CPU-based architectures. For example, in a SIMD-based design, generating tests that only involve scalar operations is inadequate. By explicitly instructing the LLM to test multi-threaded scenarios using vectors or matrices, the model can produce more appropriate test cases, mirroring how domain experts would manually construct tests to validate parallel execution paths.

Functional Units: The prompt should include detailed information about the different functional units in the architecture, such as their quantity, supported operations, and placement within the design. For instance, in a GPU architecture, multiple functional units are present, each serving a distinct purpose. ALUs handle arithmetic operations, while Load and Store Units (LSUs) manage memory operations such as reading from and writing to global memory.

Valid Registers: Different architectures impose varying constraints on how their components are implemented and utilized. For instance, the number of available registers and their usage across instructions can differ significantly. Some architectures may include fixed-value registers that cannot be modified through machine code. To capture such constraints, we define a valid registers attribute that encapsulates this information and explicitly include it in the initial prompt to the LLM. This ensures that the model adheres to the design-specific register semantics during code generation.

Valid Instructions: Finally, the initial prompt must include a detailed description of the instruction set used by the accelerator hardware implementation. Since these instructions differ significantly from those of general-purpose architectures, they must be explicitly defined in relation to the hardware features they target. In particular, the mapping between instructions and their corresponding registers should be clearly specified. Given the potential size of the instruction set, this portion of the prompt is often the most extensive. To manage this complexity, the instruction definitions can be introduced iteratively, divided across multiple prompt segments.

Figure 3 (item ①) shows an illustrative example of an initial prompt created for testing a GPU hardware implementation.

B. Test Generation with Conversation Instances

Once the LLM has a basic understanding of the architecture, it can be prompted to generate test cases aimed at improving design coverage and verifying functional correctness. To streamline this process, test generation is organized into multiple conversational instances with the LLM, each treated as an independent session, with no shared memory between them. The purpose of using separate conversation instances is to focus each one on a distinct high-level functionality supported by the accelerator hardware outlined by the test plan in the specification. Figure 3 (item ②) shows an illustrative example of a conversation prompt used for starting a conversation instance with the computing task of an elementwise matrix addition. In practice, multiple conversation instances can run in parallel, with each instance focusing on a different computation task, for example, one may target simple arithmetic operations such as elementwise matrix addition, while another may handle more complex tasks like matrix multiplication.

Conversation instances streamline the validation process in two main ways. First, automated feedback prompts within each instance allow iterative refinement, continuing the dialogue until the generated programs improve design coverage. If a conversation becomes misaligned or diverges from the intended architecture, it can be terminated early without affecting other instances. Additionally, this instance-based structure supports functional correctness validation, as discussed in Section III-D.

C. Compilation, Simulation and Error Correction

The generated input programs from the LLM are in the form of assembly instructions. To execute these on the target hardware, the assembly code must be translated into machine code (binary representation). Since programmable hardware accelerators are often designed with compiler integration in mind, we modify the associated compiler to assess the correctness of the generated programs in the following two ways:

Syntax Errors: These refer to violations of the instruction set's grammar, such as invalid opcodes, incorrect operand formats, or misuse of pseudo-instructions. The compiler frontend detects these issues during the parsing and code generation stages. Providing feedback on syntactical constraints to the LLM helps it learn valid instruction patterns and structure.

Architectural Issues: These arise when the generated program exceeds architectural limits or misuses hardware resources. Examples include referencing non-existent registers, exceeding immediate value ranges, or triggering illegal memory access patterns. These violations are typically detected during register allocation, instruction scheduling, or linking.

The compiler detects any violations of syntax or architectural constraints. These violations are incorporated into the subsequent prompts via the automatic feedback generation process outlined in Section III-E. This validation loop ensures

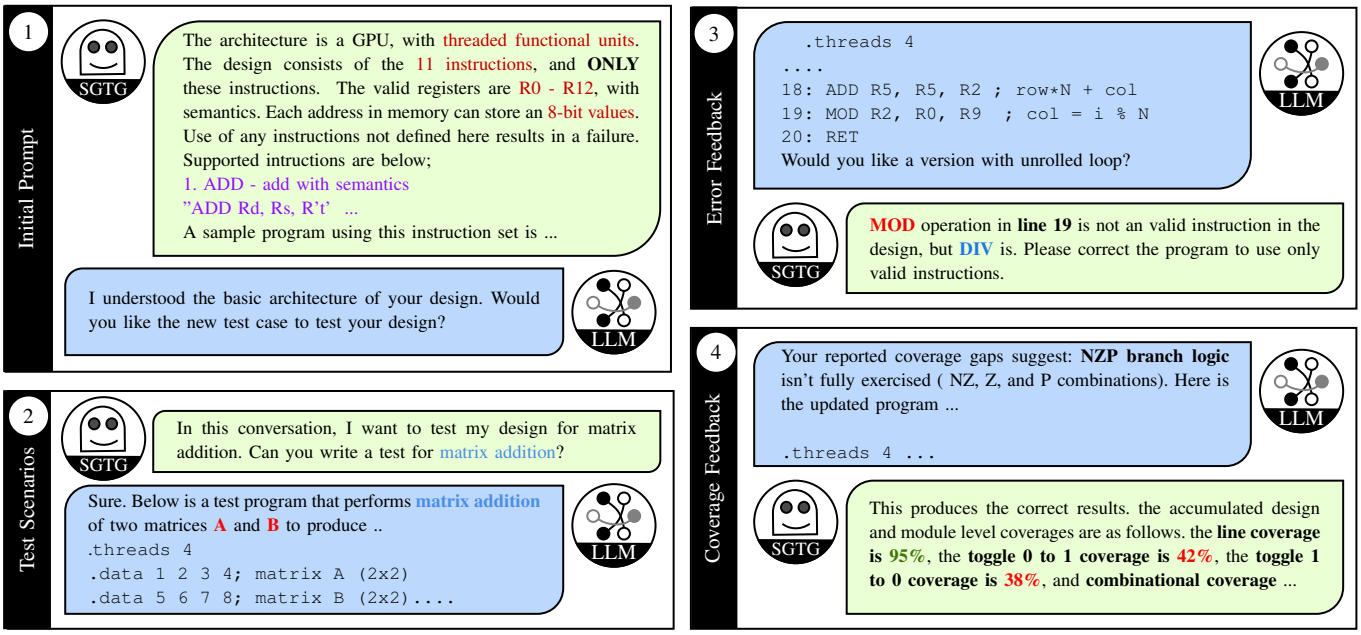


Fig. 3: Illustrative examples of four types of prompts used in our framework: ① initial prompt created in Section III-A, ② prompt that initialize each conversation instance in Section III-B, ③ sample prompt that contains errors of the generated program found by the compiler in Section III-C, and ④ prompt that contains the coverage feedback in Section III-D.

that the test programs satisfy to both the syntax and architectural constraints of the target architecture. The compiler translates the assembly code into machine (binary) code and simulates it on the hardware design. The absence of compiler errors does not guarantee logical correctness, as programs may still exhibit issues such as race conditions or infinite loops. To detect such cases, a timeout monitor is integrated into the simulation environment, which terminates execution if it exceeds a predefined duration. Timeout-related failures are also mapped to predefined error templates and incorporated into the error feedback prompts for iterative refinement.

D. Coverage Analysis and Functional Correctness

To evaluate the effectiveness of the generated test cases, we perform RTL coverage analysis using a coverage-driven workflow that supports incremental aggregation across multiple simulation runs. This allows us to measure how each newly generated test case contributes to exercising the design. The coverage analysis is performed across multiple standard metrics, including line coverage, toggle coverage, and combinational logic coverage.

- *Line coverage* measures whether each line of RTL code has been executed during simulation, helping identify unreachable code or unused logic.
- *Toggle coverage* tracks changes in the value of individual signals (from $1 \rightarrow 0$ and $0 \rightarrow 1$) across clock cycles, which is useful for detecting static or unused signal paths that may not be activated by existing tests.
- *Combinational logic coverage* observes the activation of unique input combinations to combinational blocks, allowing for a fine-grained view of how thoroughly the logic expressions have been exercised.

Once the coverage reports for each of the simulations are obtained, we merge them incrementally over successive simulation runs with the updated input programs generated from the LLM to monitor the coverage improvement. In addition to design coverage, functional correctness is evaluated after each simulation run to ensure the design performs as intended according to the specification. As described in Section III-B, tests are generated through separate conversation instances focused on different computation tasks. Functional correctness is assessed by comparing the simulation output against a golden reference model (specification). Any test that fails to produce the expected result is flagged for manual analysis. Such failures may indicate either incorrect test generation, due to misinterpretation of instruction semantics, or a functional bug in the hardware design that requires correction.

E. Automated Generation of Feedback Prompts

Feedback prompts for each of the conversation instances are generated automatically based on the outcomes of Section III-C and Section III-D, eliminating the need for manual intervention. Our framework supports two types of feedback prompts: error feedback and coverage feedback.

Error Feedback: The compiler identifies issues such as syntax errors, invalid registers, and unsupported instructions in the generated program. Similarly, run-time logical errors are captured by the simulation environment. To address these issues, we define a set of prompt templates corresponding to each type of issue. When an error is detected by either the compiler or the simulation environment, the system selects a corresponding template and generates a tailored feedback prompt. If multiple issues are identified, their respective prompts are concatenated

into a single, consecutive prompt message and routed back to the relevant conversation instance for correction. Figure 3 (item ③) shows an illustrative example of an error feedback prompt created for testing a GPU hardware implementation.

Coverage Feedback: The second type of automated feedback prompt is based on design coverage metrics. It informs the LLM about which specific coverage metric is lacking. This feedback can be provided with granularity down to the module level in the hardware description. Figure 3 (item ④) shows an illustrative example of a coverage feedback prompt created for testing a GPU hardware implementation.

Both feedback prompts conclude with asking the LLM to incorporate the feedback into the program created under the current conversation. This process is repeated until the design coverage reaches a desired percentage or remains stagnant.

IV. Experiments

In this section, we first outline our experimental setup. Next, we discuss the failure rate (compiler errors and logical errors) of the generated programs. Then, we discuss the results in terms of the design coverage improvement. Finally, we discuss the functional bugs identified by our framework.

A. Experimental Setup

To evaluate the effectiveness of the proposed framework, we selected a GPU architecture as a case study. While this evaluation focuses on a GPU, the framework is generalizable to other heterogeneous accelerator architectures, provided a simulatable hardware implementation is available. For our case study, we used a SystemVerilog implementation of the *tiny-gpu* architecture [29], a fully functional GPU that supports SIMD instructions. This implementation includes two compute cores, each featuring four threads with dedicated functional units. This reconfigurable architecture is widely used in academic settings for both teaching GPU architecture and conducting experimental research. The version we selected for our evaluation expands to 20,272 lines of unrolled Verilog code.

In order to evaluate our methodology against different LLMs with varying parameters, we have selected *GPT-4o* and *GPT-4.1 mini* models from OpenAI [9]. We have used *Sv2V* [30] to convert the SystemVerilog implementation of the *tiny-gpu* architecture [29] to a Verilog implementation. We have used *Icarus Verilog* [31] for simulation of the implementation. We have utilized *Python v3.13.3* and *cocotb v1.9.2* [32] for developing scripts for automated feedback generation and management of the testbenches for simulation. All code coverage metrics are calculated with *covered v0.7.10* [33].

We have compared our semantic-guided test generation (*SGTG*) framework with the following three approaches.

- **Baseline**: We use GPT-4o without any feedback prompts.
- **Random**: Implemented a random program generator that randomly chooses a valid instruction and randomly assigns valid registers appropriate to the instruction. For branch instructions, the branch condition code is also chosen at random, with the jump to location chosen as one of the lines in the active program. The random program always concludes with the "`RET`" instruction.
- **LLM-TG** [10]: State-of-the-art LLM-based test generation for validation of general-purpose processors.
- **SGTG-GPT-4o**: Our proposed approach that guides *GPT-4o* with feedback prompts as outlined in Figure 2.
- **SGTG-GPT-4.1-mini**: Our approach that guides *GPT-4.1 mini* with feedback prompts as outlined in Figure 2.

B. Success Rate of Generated Programs

We have generated 110 test programs for each of the five approaches (*Baseline*, *Random*, LLM-TG [10], and our approach with *GPT-4o* and *GPT-4.1 mini*), totaling 550 test programs. Figure 4 presents the success rate. Our proposed approach achieved significantly higher success rate compared to random, LLM-TG, and baseline. Specifically, GPT-4o achieved the highest success rate at 87.3%, followed by GPT-4.1 mini at 60.0%. In contrast, random generation resulted in a 35.5% success rate; while many of them were syntactically valid, they frequently encountered issues such as race conditions or infinite loops, ultimately failing during execution due to timeouts. LLM-TG generated many invalid test cases (11.8% success rate), primarily because it attempted to use instructions and assumptions from general-purpose architectures, failing to align with the semantics of the GPU architecture. Baseline lacked any feedback mechanism, had a 100% failure rate.

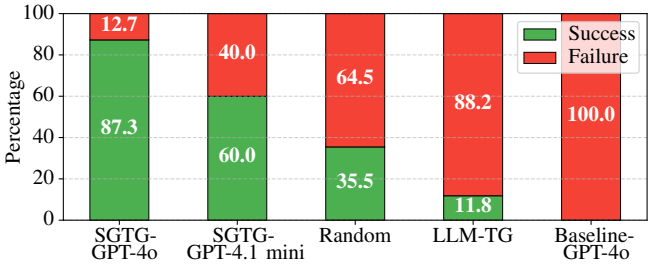


Fig. 4: Comparison of test program generation success rate across five approaches. The first two columns show that our approaches (SGTG-GPT-4o and SGTG-GPT-4.1 mini) achieve the highest success rates. Randomly generated programs and LLM-TG exhibit low success rates. The baseline results in a 100% failure rate due to its lack of architectural knowledge.

C. Design Coverage Results

Based on the test plan, more than eight conversation instances were carried out during our experiments, including operations such as elementwise matrix addition, elementwise matrix subtraction, elementwise matrix multiplication, regular matrix multiplication, and elementwise matrix division. Figure 5 contains a sample test case produced by SGTG with GPT-4o for elementwise matrix addition on 8 threads between two 4×2 matrices. From prior feedback, the LLM learned that achieving high code coverage requires all possible instructions, especially the branch instruction to cover the branch functionality for both the state registers and threads coverage. Without the code coverage feedback, the branch instruction

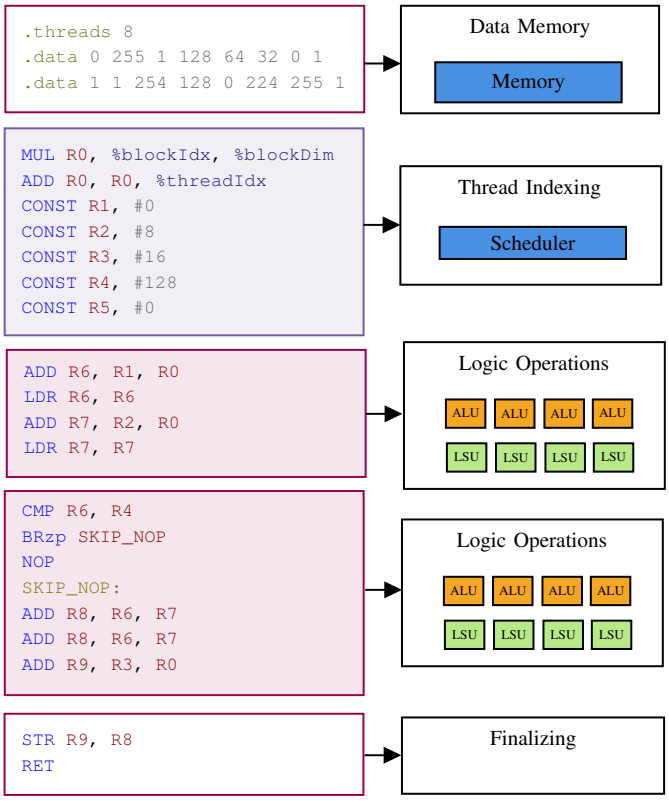


Fig. 5: Sample program generated by GPT-4o with our proposed approach for elementwise addition of two 4×2 matrices.

(`BRnzp`) would not have been used, as the minimal viable product for elementwise matrix addition does not require a branch instruction.

To demonstrate the effectiveness of code coverage improvement and error handling, we analyzed each conversation instance in detail. Table I shows a detailed view of the coverage progression from one such randomly selected conversation instance. The table shows the improvement of accumulated coverage for four coverage metrics (line, 0→1, 1→0, and combinational) with our framework using GPT-4o. It can be observed that LLM starts generating programs with four threads and quickly realizes it needs to increase the thread number to achieve better coverage. In the third test, it produces an invalid program, however, the feedback mechanism quickly rectifies it in the next iteration. Similar adaptive behavior is observed across other conversation instances as well.

To illustrate the effectiveness of our proposed approach on design coverage, we explore various coverage metrics. Among these, toggle coverage is particularly challenging to achieve due to the data-dependent behavior of the implementation. Nevertheless, our technique significantly improves toggle coverage compared to other methods. Figure 6 shows the accumulated bit-level toggle coverage across all 110 test cases compared to randomly generated test cases. Specifically, toggle transitions from 0 → 1 and 1 → 0 are plotted in Figure 6a and Figure 6b, respectively. Our approach with GPT-4o achieved the highest toggle coverage: 83% for 0→1 transitions and 82% for 1→0. GPT-4.1 mini followed with

TABLE I: Accumulated coverage for four coverage metrics (line coverage, 0→1 and 1→0 toggle coverage, and combinational logic coverage) based on a conversation in our framework with GPT-4o that produced 12 test programs.

Test	Line	0 → 1	1 → 0	Combinational	Threads
1	93	32	30	78	4
2	96	42	39	83	4
3			invalid register		8
4	95	47	43	89	8
5	96	48	44	89	8
6	96	51	46	90	8
7	96	70	66	90	8
8	96	75	72	90	8
9	96	76	73	90	8
10	96	77	74	90	8
11	96	77	74	90	8
12	96	78	75	90	8

71% and 68%, respectively. In contrast, the random approach only achieved 50% for 0→1 and 46% for 1→0. Similarly, LLM-TG achieved only 50% toggle coverage. These results demonstrate that our framework (with GPT-4o and GPT-4.1 mini) significantly outperform (33%) the existing approaches.

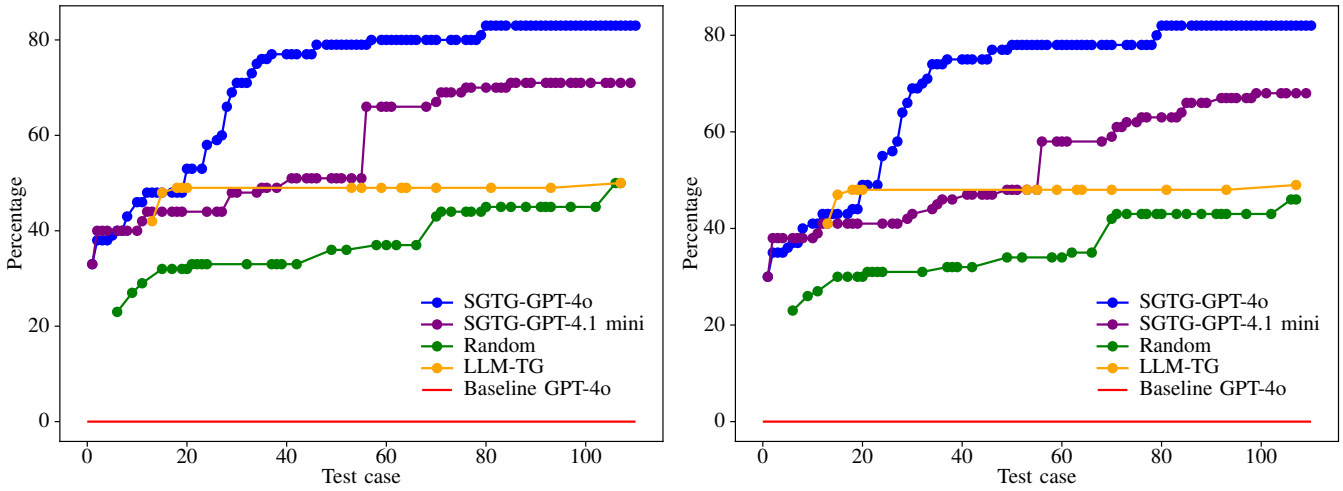
D. Analysis of Identified Functional Bugs

Our test generation framework successfully uncovered several functional bugs in the hardware implementation of the *tiny-gpu* architecture [29]. In this section, we discuss two of them in detail. One critical issue involved thread-level branching behavior within a single compute core. In the design specification, each thread is expected to evaluate branch conditions independently, allowing them to follow different execution paths based on their own data. However, the actual implementation incorrectly forced all threads in a compute core to follow the same branch path—specifically, the path chosen by a single thread—regardless of their individual condition evaluations. For example, if a branch depends on whether a thread's register value is greater than zero, some threads should execute the branch while others should not. Instead, all threads were taking the same path as the first thread that evaluated the branch, leading to incorrect behavior and violating the expected semantics of per-thread execution.

Another functional bug identified through this process involved incorrect register updates in certain threads. Specifically, some threads failed to correctly write values to their destination registers. Specifically, a thread that is supposed to store a computed value into a register leaves the register unchanged. As a result, any subsequent operation or test that relies on the expected value in that register would fail, revealing inconsistencies between intended and actual behavior.

V. Conclusion

LLMs are promising for validation of general-purpose processors. However, LLMs are not suitable for verifying pro-



(a) Accumulated coverage of toggle $0 \rightarrow 1$ across all test cases.

(b) Accumulated coverage of toggle $1 \rightarrow 0$ across all test cases.

Fig. 6: Accumulated code coverage of all test cases produced by the proposed methodology with GPT-4o and GPT-4.1 mini, as well as LLM-TG and random test generation. Each method produced 110 assembly test cases. Test cases without a plot point (•) indicate that the test case resulted in a compilation/logical error, and thus could not be simulated and scored for coverage.

grammable hardware accelerators due to the lack of training data for such accelerator implementations. We presented an automated test generation framework that guides LLMs with feedback prompts based on coverage and errors for functional validation of hardware accelerators. Our feedback-based framework is able to generate test cases that can achieve a 33% increase in coverage over state-of-the-art for hardware accelerated designs. These feedback prompts sufficiently informed the LLM about the code choices it made, allowing for it to expand on the code to improve coverage by testing additional control paths or by correcting errors located within the generated code. In addition to the improved coverage, the test cases generated with our methodology can test for functional correctness, allowing for the discovery of functional bugs within the design. In fact, we were able to detect two critical bugs within the hardware implementation of a widely used open-source GPU architecture.

References

[1] T. Mitra, "Heterogeneous multi-core architectures," *IMT*, 2015.

[2] B. Marshall *et al.*, "Implementing the draft risc-v scalar cryptography extensions," in *Hardware & Arch. Supp. for Security and Privacy*, 2020.

[3] J. Owens *et al.*, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[4] Google Cloud, "Tensor processing units (tpus) - google cloud," https://cloud.google.com/tpu, 2025, accessed: 2025-05-15.

[5] "NVIDIA CUDA C Programming Guide," 2010, version 3.2. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[6] AMD, "AMD ROCm Documentation," 2025. [Online]. Available: https://www.amd.com/en/products/software/rocm.html

[7] OpenXLA Project, "XLA: Accelerated Linear Algebra Compiler," 2025, accessed: 2025-05-15. [Online]. Available: https://openxla.org/

[8] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[9] OpenAI, "ChatGPT," 2025, https://chat.openai.com.

[10] Y. Deng *et al.*, "LLM-TG: Towards automated test case generation for processors using large language models," in *ICCD*, 2024, pp. 389–396.

[11] J. Bhandari *et al.*, "LLM-Aided Testbench Generation and Bug Detection for Finite-State Machines," 2024.

[12] C. Xiao *et al.*, "LLM-based Processor Verification: A Case Study for Neuromorphic Processor," in *DATE*, 2024.

[13] A. Jayasena and P. Mishra, "Directed Test Generation for Hardware Validation: A Survey," *ACM Comput. Surv.*, vol. 56, no. 5, pp. 132:1–132:36, Jan. 2024.

[14] P. N. Glaskowsky, "Nvidia's fermi: the first complete gpu computing architecture," *White paper*, vol. 18, p. 3, 2009.

[15] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[16] S. Gueron, "Intel advanced encryption standard (intel aes) instructions set–rev 3.01," *Intel, Aug*, 2012.

[17] "RISC-V Cryptography," https://github.com/riscv/riscv-crypto, 2023.

[18] R. Cypher and J. L. Sanz, *The SIMD model of parallel computation*. Springer Science & Business Media, 2012.

[19] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *computer*, vol. 29, no. 12, pp. 66–76, 1996.

[20] J. Doerfert, C. Hammacher, K. Streit, and S. Hack, "Spolly: speculative optimizations in the polyhedral model," *IMPACT 2013*, vol. 55, 2013.

[21] A. Vaswani *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, 2017.

[22] X. Wang *et al.*, "ChatCPU: An Agile CPU Design and Verification Platform with LLM," in *DAC*, 2024.

[23] F. Firouzi *et al.*, "LLM-AID: Leveraging Large Language Models for Rapid Domain-Specific Accelerator Development," in *ICCAD*, 2025.

[24] H. Huang *et al.*, "Towards LLM-Powered Verilog RTL Assistant: Self-Verification and Self-Correction," May 2024.

[25] S. Liu *et al.*, "OpenLLM-RTL: Open Dataset and Benchmark for LLM-Aided Design RTL Generation," in *ICCAD*, 2025.

[26] R. Ma *et al.*, "VerilogReader: LLM-Aided Hardware Test Generation," in *LAD*, 2024.

[27] M. Hassan *et al.*, "Llm-guided formal verification coupled with mutation testing," in *DATE*, 2024, pp. 1–2.

[28] R. Qiu, G. L. Zhang, R. Drechsler, U. Schlichtmann, and B. Li, "AutoBench: Automatic Testbench Generation and Evaluation Using LLMs for HDL Design," in *MLCAD*, 2024, pp. 1–10.

[29] A. Maj, "tiny-gpu: GPU implementation in systemverilog." https://github.com/adam-maj/tiny-gpu, 2025, accessed: 2025-04-18.

[30] Zachjs, "sv2v: Systemverilog to verilog compiler," https://github.com/zachjs/sv2v, 2016, accessed: 2025-05-23.

[31] S. Williams and M. Baxter, "Icarus verilog: open-source verilog more than a year later," *Linux Journal*, vol. 2002, no. 99, p. 3, 2002.

[32] "Cocotb/cocotb," https://github.com/cocotb/cocotb, May 2025.

[33] T. Williams, "Covered," 2010, version 0.7.10. [Online]. Available: https://linux.die.net/man/1/covered