From Prompts to Pressure: Evaluating LLM-driven Agents for GPU Stress-code Generation

From Prompts to Pressure: Evaluating LLM-driven Agents for GPU Stress-code Generation

Aurora Gensale*, Giuseppe Esposito*, Juan-David Guerrero-Balaguera*, Josie E. Rodriguez Condia*, Luca Cagliero*, Matteo Sonza Reorda*
*Politecnico di Torino - Department of Control and Computer Engineering (DAUIN), Turin, Italy
{name.surname}@polito.it

***Abstract*—Latest semiconductor technologies are increasingly sensitive to faults, which can compromise the reliability of Data Centers and High-Performance Computing (HPC) systems when running compute-intensive workloads. To excite and detect in-field permanent faults, stress-oriented routines are periodically executed before such faults manifest as critical failures during standard workload execution. However, state-of-the-art stress tests typically focus on available applications (e.g., matrix multiplication) and overlook latent faults that emerge only under specific computational patterns. Designing ad-hoc applications capable of triggering these faults requires substantial manual effort and deep knowledge of hardware architecture. To address these limitations, we present an empirical study that, for the first time, assesses the ability of Large Language Models (LLMs) to collaborate within a multi-agent iterative framework to automatically generate GPU-stress-testing routines. Experimental results show that four of the seven evaluated LLMs consistently produce compilable and executable code, enabling the automated development of stress-test applications without human intervention and significantly reducing the time needed to obtain a functional prototype compared to manual development. Among the tested LLMs, two, used as the core LLMs for all agents, generated two stress-testing applications that were 80.35% and 37.51% more compact than GPU-Burn (a state-of-the-art stress-testing tool for GPUs), yet still reached 89.9% of GPU-Burn's peak temperature and 77.2% of its energy consumption.**

***Index Terms*—Testing automation, Stress-testing routines, Large Language Models, Reliability, Artificial Intelligence**

I. Introduction

Modern supercomputing infrastructures, such as Data Centers and High-Performance Computing (HPC) systems, increasingly rely on cutting-edge semiconductor technologies to sustain compute-intensive workloads, ranging from medical imaging to large-scale scientific simulations and Neural Network (NN) training [1]. The continuous scaling of transistor dimensions has enabled unprecedented computational capabilities, allowing real-time processing and faster time-to-solution for complex algorithms. However, increased transistor density and sustained utilization significantly raise power density within the silicon die. Electrical stress, combined with elevated temperatures, accelerates physical degradation mechanisms over time [2]. Among the most critical aging phenomena are Negative Bias Temperature Instability (NBTI) and electromigration [2].

Such degradation mechanisms progressively reduce circuit timing margins and may eventually manifest as Silent Data Errors (SDEs), which can silently propagate during application execution and culminate in system-level failures if left undetected [3]. A notable real-world example is the Titan supercomputing system, where premature device aging occurred on average every 2.8 years, requiring the replacement of approximately 59% of the Graphics Processing Units (GPUs) in the complete supercomputing infrastructure [4]. These observations highlight the tangible operational and economic impact of hardware aging in large-scale infrastructures, making early detection of degradation-induced faults a primary concern. To address this challenge, in-field functional testing strategies have been developed, relying on intensive workloads designed to reproduce operational stress conditions and activate latent fault [5]–[7]. These approaches aim to emulate worst-case operational scenarios to increase the probability of fault activation. However, most existing stress-testing routines rely on kernels that induce dense and repetitive operations (e.g., matrix multiplications to stress functional units), thereby exerting relatively uniform computational pressure on the device. Although effective at maximizing thermal and power loads, such workloads do not necessarily exercise the complete diversity instruction mixes (or computing patterns) observed in all real applications deployed in distributed systems.

Consequently, current stress approaches may fail to expose defects that manifest only under specific instruction mixes or execution paths [5], [8], [9]. If a defect is associated with a particular functional unit or architectural region that is insufficiently exercised during testing, it may remain undetected. Achieving broader fault activation, therefore, requires not only high computational intensity, but the deliberate generation of diverse, semantically valid programs capable of targeting different functional units and execution behaviours. For example, benchmark suites, such as Rodinia, provide GPU workloads that exhibit heterogeneous parallelism, memory-access patterns, and data-sharing strategies [10]. Nevertheless, these benchmarks are not explicitly designed to systematically explore microarchitectural stress conditions or to maximize fault-activation coverage.

Furthermore, manually designing programs for hardware stress is complex and resource-intensive. Crafting workloads that intentionally produce heterogeneous instruction mixes while preserving correctness demands deep architectural expertise and iterative tuning. Besides, GPU scheduling policies and compiler optimizations introduce additional variability, making the resulting execution behaviour difficult to pre-

dict and control. This combination of semantic constraints, architectural objectives, and program diversity requirements naturally leads to a program-synthesis formulation of the stress-generation problem.

Recent advances in Large Language Models (LLMs) have demonstrated strong capabilities in generating syntactically correct and semantically coherent high-level code conditioned on functional objectives. In this work, we leverage LLMs not as generic code generators, but as constraint-driven program code engines capable of producing valid GPU kernels aligned with high-level architectural stress targets. Our approach enables scalable exploration of applications that induce stress, increasing the likelihood of activating latent hardware faults compared to human-generated applications.

To address these challenges, this work investigates, for the first time, the capability of LLMs to perform automated stress-test code generation for GPUs. By leveraging an LLM-driven framework, we aim to generate and refine stress workloads that activate diverse computational patterns. This automated approach reduces the reliance on manual code design while enabling scalable exploration of stress conditions, ultimately improving the detection of latent hardware faults.

We evaluate the programs for GPUs generated by seven LLMs against five reference applications specifically designed to exercise heterogeneous computational patterns on GPUs. The experimental results show that the code produced by the LLMs is consistently compilable and executable, enabling the automatic creation of stress-inducing applications without human intervention and substantially reducing the time needed to obtain a working prototype compared to manual development. Among the evaluated models, two out of seven tested LLMs achieved the best performance. They generated two stress-testing applications that were 80.35% and 37.51% more compact than GPU-Burn [11] (the reference stress-testing tool for GPUs), yet still reached 89.9% of its peak temperature and 77.2% of its power consumption.

The remainder of the paper is organized as follows. Section II reviews the background on LLM-based code generation and multi-agent architectures. Section III presents the proposed LLM-driven stress-generation framework and its Hardware-in-the-Loop pipeline. Section IV describes the experimental setup, and Section V reports and discusses the experimental results. Finally, Section VI concludes the paper and outlines future research directions.

II. Background

Recent advances in LLMs have demonstrated progress in contextual understanding and logical reasoning, establishing their relevance for software engineering tasks. Early empirical studies on code-specialized models [12] showed that LLMs can generate code rather than performing simpler tasks such as sentence autocompletion. Techniques, such as Chain-of-Thought (CoT) prompting, further enhanced this capability by enabling models to decompose complex problems into structured reasoning steps, which is essential for the correct compiling and execution of the generated program [13].

To effectively harness and control these advanced reasoning capabilities, modern LLMs rely on the distinction between *i)* System Prompt and *ii)* User Prompt. This separation was formalized through *instruction-tuning* research [14] that involves a specific training phase in which models are explicitly optimised to follow structured human commands and adhere to predefined rules rather than simply predicting the next sequence of words. In this paradigm, the System Prompt specifies persistent behavioural constraints and global policies, while the User Prompt conveys the transient, task-specific input. Furthermore, recent empirical analyses [15] demonstrate the critical importance of this hierarchy; in fact, system-level instructions act as a robust control mechanism that consistently overrides conflicting or ambiguous user inputs. From a system-design perspective, it effectively allows developers to enforce strict output formats, define specific reasoning styles, and assign rigid, persistent agent roles without the need for expensive model retraining.

Building on these foundations, recent literature has explored Multi-Agent LLM architectures [16], where multiple specialized model instances collaborate to solve complex tasks. In such systems, each agent is assigned a distinct role aligned with a specific contextual responsibility. Frameworks like [17], [18] simulate organizational roles (e.g., Chief Executive Officer, Chief Technology Officer, Programmer) to coordinate software production, while [19] employs role definitions to model diverse personalities and behaviours in social simulations. In particular, MetaGPT [18] demonstrates that role decomposition of tasks improves maintainability and reduces hallucinations by constraining each agent's reasoning scope.

In addition to role definitions, the potential of this decomposition can also be evident in the interaction of specialised agents within dynamic pipelines. By explicitly separating generation and evaluation tasks across different personas, these architectures prove highly adaptable across complex domains. Code generation is a use case for this paradigm [18], [20] because, unlike open-ended natural language, software provides deterministic, quantifiable feedback, such as standard outputs and error logs. Recent literature has demonstrated the advanced capabilities of LLMs to iteratively improve generated code by incorporating reflection and collaboration mechanisms [21]–[23] and by leveraging direct execution-based feedback [24], [25]. While such collaborative frameworks have proven effective in debugging and optimizing software according to quantitative criteria, their most critical advantage lies in their capacity to significantly reduce overall development cycles. For example, [17] has shown the ability to autonomously generate, review, and test complete multi-file software solutions in less than three minutes on average. For this reason, the benefits of collaborative LLM-driven development become even more relevant when moving from high-level software engineering to lower-level programming tasks. While human-based high-level development (e.g., Python-based applications) already involves iterative and time-consuming manual refinement, this complexity is significantly amplified in the context of low-level code, where both implementation and optimization demand

substantial hardware architecture expertise.

In particular, when the objective is to design workloads that deliberately induce stress on a device, several factors must be carefully accounted for, as stress can arise through multiple and interacting mechanisms. For instance, a computationally intensive application may push the device toward nearly full utilization, thereby increasing power consumption and causing a sustained rise in operating temperature. More generally, the computational intensity and resource demand of a workload directly influence the device's thermal and power behavior. In line with the study presented in [26], stress can thus be characterized as the strain imposed on the hardware in terms of: *i)* elevated internal activity, representing high utilization of computational and memory resources, and *ii)* a significant impact on telemetry metrics (e.g., temperature and energy consumption), which reflect the physical load experienced by the device. Consequently, generating stress-inducing workloads requires fine-grained control over resource usage, even at the software level. For this reason, special-purpose and stress-oriented applications are typically implemented using low-level programming models, such as CUDA, PTX, or Assembly (e.g., SASS) [5], [6], [27], [28], which provide explicit control over hardware resources and execution behaviour.

These languages enable more precise activation of specific computational units. However, such control comes at the cost of reduced portability and scalability: the resulting code is often tightly coupled to a specific Instruction Set Architecture (i.e., PTX and SASS) or to particular compute capability (CC), driver versions and hardware configurations (in the case of CUDA or C++). Thus, adapting these workloads to different architectures requires additional manual effort and deep domain knowledge. A limited number of studies have investigated the use of LLMs for the generation and optimization of efficient low-level CUDA-based code [29]–[31]. Nevertheless, these works primarily focus on performance optimization, and do not examine the capability of LLMs to generate applications explicitly designed to maximize workload-induced stress, exciting and activating latent faults.

To the best of our knowledge, no prior work has investigated the use of multi-agent LLM frameworks to autonomously generate code specifically designed to perform hardware stress tests for in-field fault detection.

III. Methods

In this section, we present an automated LLM-based multi-agent framework that autonomously generates stress-testing code for the targeted hardware. In detail, we adopt a Hardware-in-the-Loop (HiL) paradigm that provides real-time hardware feedback immediately upon code generation.

As illustrated in the overall pipeline in Figure 1, the experimental study is structured as an iterative loop that relies on the interaction among LLM-based agents, guided by two initial user-defined inputs: *i)* the target hardware feature and *ii)* the desired test duration. In the framework, three specialized roles are assigned to three LLM-based entities:

1) **Stress Code Expert Agent:** the primary code generator responsible for creating the initial hardware stressing code targeting the specified feature;
2) **Code Fixer Agent:** a dedicated debugger that repairs compilation and runtime failures by analysing error logs;
3) **Refiner Agent:** a performance analyzer that evaluates profiling feedback to produce code refinement directives, guiding subsequent code generation rounds.

Leveraging this modular architecture, the framework's execution is structured into three main phases: *(1) Code generation*, *(2) Validation and Debugging*, and *(3) Code refinement*. The complete execution flow of these phases is detailed in the following subsections.

A. Code generation

The pipeline input is a user-defined tuple $I = (f, t)$. Here, f represents the target hardware architecture under test and t denotes the desired workload duration. The tuple is passed to the *Stress Code Expert Agent*, the primary generative engine of the framework, and the global loop begins. This agent receives an ad hoc prompt template as input to guide the model in generating low-level code tailored to the target hardware architecture. Therefore, a dedicated parsing module isolates syntactically valid source code and saves it as a compilable file.

B. Validation and Debugging

To assess both syntactic and semantic correctness of the generated code, we have implemented a two-stage validation nested loop controlled by the *Code Fixer Agent*. In the first stage, the generated script is compiled using a standardized Makefile template to ensure that all applications are built under identical conditions and can therefore be fairly compared. In the case of a compilation failure, the associated error log E_{comp} is extracted and passed to the Code Fixer Agent, along with the incorrect code, starting the nested debugging loop.

The agent produces a corrected version of the program, which is then re-parsed and re-compiled. This correction–compilation process is repeated iteratively until the code compiles successfully or a maximum of N_{comp} attempts is reached. If the agent fails to resolve the errors, the framework aborts the execution to prevent infinite loops.

Although the successful compilation, the application can still incur errors at runtime; thus, the semantic correctness needs to be checked along with a stable runtime behaviour. Therefore, the resulting binary is executed on the target hardware. If any runtime anomalies occur (e.g. segmentation faults, illegal memory accesses or timeouts), an execution error log E_{exec} is generated. The debugging pipeline is then triggered again, and the Code Fixer Agent receives both the code and the encountered error to generate the necessary corrections, thereby ensuring an error-free execution. The Code Fixer Agent performs up to N_{exec} attempts to resolve the runtime errors and produce a functioning version of the code.

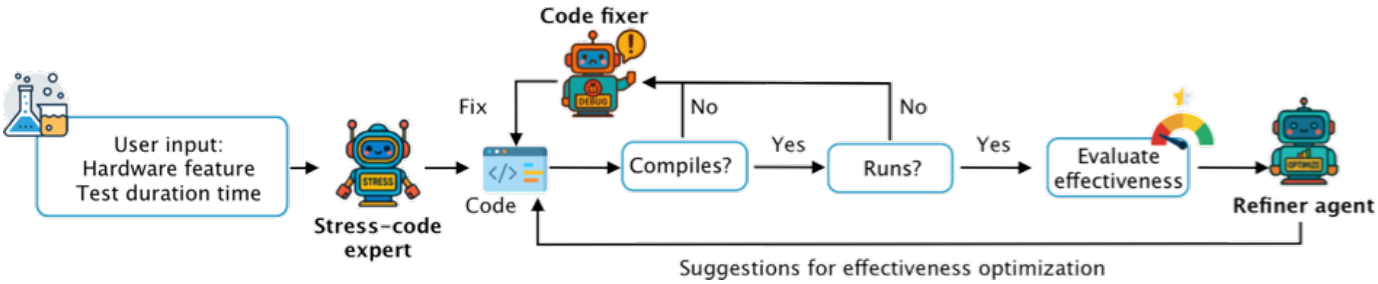

Fig. 1. High-level overview of the framework workflow

C. Code refinement

Once $C_{valid,i}$ is generated, the framework evaluates its ability to induce effective stress. To perform this assessment, we developed an application profiler that collects a comprehensive set of metrics and stores them in the feature vector $\mathbf{M}_i$. These metrics characterize the workload-induced stress from two complementary perspectives: *i)* internal hardware activity and *ii)* physical device stress, in accordance with the methodology introduced in [26].

Within the LLM-based multi-agent paradigms [17], [32], agents typically communicate through natural language feedback. For this reason, the *Refiner Agent* receives the tuple $(C_{valid,i}, \mathbf{M}_i)$ as input and it outputs its findings as S_i: structured textual directives on high-level algorithmic implementation. This design choice ensures a clear separation of responsibilities. It allows the *Refiner Agent* to concentrate on identifying correlations between the provided metrics and the code. As a result, the *Refiner Agent* can generate implementation suggestions, which are then sent to the *Stress Code Expert Agent*, specialized in programming.

To trigger the next iteration of the global loop, these instructions S_i are fed back to the initial *Stress Code Expert Agent*, which creates an improved version of the source code, denoted as C_{i+1}, while keeping track of the previously generated code, thereby keeping consistency while improving the generated application. The new code then undergoes the same validation and debugging pipeline before the next refinement step. This global closed-loop architecture iterates until a defined stopping criterion is met, either by reaching a specific hardware stress metric indicator, H_{target}, or by exhausting the predetermined maximum number of global loop iterations, R_{max}.

IV. Experimental Setup

A. Experimental campaigns

During the experimental campaigns, the proposed framework is instantiated for NVIDIA GPU stress testing. Specifically, the stress-code expert is instructed to generate workloads for f = RTX 3060 TI with CUDA 11.6 (8GB VRAM) with a test duration (t) = 300 seconds. We define 2 experimental scenarios: *i)* Application generation and optimization (**S1**) to validate the framework's ability to autonomously generate stress-inducing code from scratch, and *ii)* Kernel refinement (**S2**) to refine and enhance existing, well-known stress workloads.

In S1, the framework autonomously generates and iteratively refines a workload to enable comparison with GPU-Rodinia benchmarks [10]. Agents are configured through role-specific system prompts: *i)* the **Stress-Code Expert** generates CUDA code compatible with `nvcc` compiler; *ii)* the **Code Fixer** debugs CUDA code with explicit awareness of the `nvcc` toolchain; *iii)* the **Refiner Agent** gives feedbacks on how to refine the application based on runtime telemetry and profiling metrics.

In S2, the framework operates in a pure refinement setting. The original GPU-Burn implementation [6] (based on C++ code) is injected as the initial candidate C_0, bypassing the generative phase of the first global loop iteration. After compiling and profiling the baseline to obtain $\mathbf{M}_0$, both the source code and metrics are fed to the Refiner Agent to start the refinement loop. To preserve compatibility with the original implementation, the `g++` toolchain is retained along with the original compiler options. However, to explore alternative computational strategies beyond the cuBLAS-based used in GPU-burn, agents are configured to allow lower-level code redesign: *i)* the **Stress-Code Expert** generates PTX code integrated into the existing C++ codebase and compatible with `g++`; *ii)* the **Code Fixer** debugs mixed C++/PTX code within the same compilation environment; *iii)* the **Refiner Agent** gives feedbacks on how to refine the combined implementation based on profiling and telemetry metrics to explore more computing-intensive matrix-multiplication variants.

Prompts for both experimental campaigns are available on GitHub [33].

B. Profiling metrics

We analysed the stress exerted on the GPUs from two perspectives: the *i)* hardware resource utilization, and *ii)* device physical measurements, aligning with the experimental study in [26]. The metrics that monitor the internal hardware events include Throughput (TH), which reflects computational stress by indicating how intensively the GPU's execution units are utilized; and the percentage of stall events due to internal activity, denoted as S_{act}, which reveals bottlenecks in the computations due to stalled or blocked execution from the internal activity perspective. On the other hand, the device physical measurements rely on telemetry metrics, which involve the

temperature and the spent energy as a direct indicator of device stress [34]. Additionally, drawing on prior studies that highlight the correlation between the number of Lines of Code (LoC) and software complexity [35], we included the LoC metric to assess the structural complexity of each application. In general, a higher number of LoC indicates greater code structural complexity, which in turn tends to reduce the workload-induced stress.

Since thermal dissipation and electrical activity are the main device physical stress indicators, we select the GPU temperature and energy as our primary stopping condition. To establish an objective baseline, we profiled the state-of-the-art stress-testing application, *GPU-burn* [6], under the same input conditions $I = (f, t)$. This execution yielded a peak temperature of 59.8°C. Therefore, we defined our target stress metric indicator $H_{target} = 59.8°C$. As explained in III-C, one case for the optimization loop stops successfully if the current temperature of the generated workload surpasses this threshold or $R_{max} = 5$ is reached. Finally, to bound the framework's execution and prevent infinite iterative loops, the limits for the *Validation and Debugging* phase are set as $N_{comp} = 5$ and $N_{exec} = 5$ execution attempts. R_{max}, N_{comp}, and N_{exec} were determined through preliminary experiments, which highlighted that extended program modification chains frequently resulted in loss of contextual coherence within the models. This often led to unproductive loops or modifications that degraded stress performance.

C. LLM agents configuration

The evaluated suite spans a wide range of model sizes, measured by the number of trainable weights (parameters), from 4B to 120B. It includes the open-weight models *gemma-3n-E4B-it*, *Llama-3.1-8B*, *gpt-oss-20B*, *Llama-3.3-70B*, and *gpt-oss-120B*, alongside the proprietary models *gpt-5.2* and *gemini-2.5-pro*, whose parameter counts are undisclosed but which are widely considered as industry-leading architectures. Although prior work suggests that models with more parameters generally exhibit stronger reasoning and generative capabilities [36], [37], we include models of substantially different sizes and architectures to evaluate how this trend manifests in this specialized domain.

The generation temperature, a hyperparameter controlling output randomness, is fixed at 0.2 for all models as it maximises the likelihood of producing functionally correct code on the first attempt [12].

V. Results

This section presents the experimental results of our study. Specifically, Section V-A discusses the generative and debugging capabilities of the tested LLMs in producing viable CUDA code. Subsequently, in Section V-B, we compare the stress exerted from the valid LLM-generated applications on the GPU with GPU-Burn, three representative benchmarks from GPU-Rodinia (namely, *Back-propagation*, *Needleman* and *Hotspot*), and a basic implementation of a Matrix Multiplication (M×M) CUDA kernel reporting the measurements described in Section IV-B. Both analyses cover the two distinct experimental scenarios detailed in Section IV.

TABLE I
Validation Phase: Correction Attempts

LLM Agent	S1		S2	
	Compilation Fixes	Execution Fixes	Compilation Fixes	Execution Fixes
gpt-oss-20B	5*	—	5*	—
gpt-oss-120B	2	0	5*	—
gemma-3n-E4B-it	0	0	5*	—
Llama-3.1-8B	5*	—	5*	—
Llama-3.3-70B	0	0	1	0
gpt-5.2	2	0	3	0
gemini-2.5-pro	5*	—	1	0

Note: **0** indicates that the Code Fixer didn't need to perform any fix on the generated code, thus the generation succeeded at the first try. **5*** indicates the framework aborted because it reached the maximum $N_{comp} = 5$ limit, thus the execution validation is not performed.

A. LLM capabilities analysis

Table I presents the performance of the evaluated LLMs during the Validation and Debugging phase. Rows list the deployed LLM agents, while columns quantify the number of iterative attempts required by the *Code Fixer Agent* to resolve compilation and execution errors across the two experimental scenarios (S1 and S2).

Focusing on **S1**, first-attempt compilation success (0 compilation fixes) was achieved by the 70B parameter *Llama-3.3-70B* and, remarkably, by the highly compact 4B parameter *gemma-3n-E4B-it*, a finding supported by recent studies [38] showing that well-designed and trained Small Language Models (SML) can overcome parameter limitations and achieve performance comparable to larger architectures in code generation. *gpt-oss-120B* and *gpt-5.2* models also produced valid code, both requiring exactly 2 compilation fixes to converge. Conversely, smaller architectures such as *gpt-oss-20B* and *Llama-3.1-8B*, as well as the proprietary *gemini-2.5-pro*, failed to produce compilable code, triggering the framework's abort condition by reaching the $N_{comp} = 5$.

Results from **S2** show that modifying a kernel from an existing codebase is more challenging for the evaluated models. Indeed, 4 out of the 7 tested models failed to generate compilable code, due to the need to preserve structural dependencies within the original program, further amplified by the complexity and verbosity of the *GPU-burn* codebase. Specifically, *gemma-3n-E4B-it* and *gpt-oss-120B*, which succeeded in S1, experienced a severe performance degradation in S2, jumping from 0 and 2 fixes, respectively, to hitting the 5-attempt failure limit. In contrast, *gemini-2.5-pro* exhibited a completely reversed trend, failing S1 but succeeding in S2 with only 1 compilation fix, suggesting that this model benefits from modifying existing code rather than generating it from scratch. *Llama-3.3-70B* and *gpt-5.2* demonstrated the highest stability, requiring only 1 and 3 compilation fixes, respectively.

Finally, a significant observation across all successful generations in both scenarios is the complete absence of runtime

TABLE II
PERFORMANCE AND TELEMETRY MEASUREMENTS COLLECTED FROM THE APPLICATIONS SUCCESSFULLY GENERATED BY THE LLMS DURING S1, ALONGSIDE THOSE OBTAINED FROM THE GPU-RODINIA [10] BENCHMARK SUITE.

LLM agent/ Application	Temperature (°C)	Energy (J)	Throughput (%)	Stalls (%)	LoC
gpt-oss-120B	44.92	**4,846.69**	33.64	**18.05**	**144**
gemma-3n-E4B-it	55.31	3,713.51	30.36	21.14	69
Llama-3.1-8B	43.01	3,652.33	30.70	20.44	85
gpt-5.2	**58.11**	3,421.18	64.01	21.04	458
Back propagation	52	2,365.98	**78.43**	28.05	160
Needleman	53	2,540.28	3.05	23.80	232
Hotspot	50	1,952.37	88.26	38.96	243

TABLE III
PERFORMANCE AND TELEMETRY MEASUREMENTS COLLECTED FROM THE *i)* APPLICATIONS SUCCESSFULLY GENERATED BY THE LLMS DURING S2, *ii)* THOSE OBTAINED FROM THE GPU-BURN [6] AND *iii)* A BASE IMPLEMENTATION OF THE M×M ALGORITHM IN CUDA PROCESSING 4069 SQUARED MATRICES.

LLM agent/ Application	Temperature (°C)	Energy (J)	Throughput (%)	Stalls (%)	LoC
Llama-3.1-8B	53.16	3,240.76	23.36	46.67	788
gpt-5.2	51.61	3,482.01	99.54	59.93	1,017
gemini-2.5-pro	53.16	3,035.54	89.86	16.97	854
GPU-burn	**59.8**	**5,700**	**86.91**	**7.89**	**733**
M×M	52.4	2,185	35.36	70.34	206

errors. Every model that successfully passed the compilation phase required precisely 0 execution fixes. This strongly suggests that the primary bottleneck for LLMs in low-level GPU programming lies in adhering to strict syntax and static typing rather than in formulating correct semantic logic.

Consequently, only the models that successfully produced valid, executable CUDA workloads will be subjected to the physical evaluation phase. Specifically, the workloads generated by *gpt-oss-120B*, *gemma-3n-E4B-it*, *Llama-3.3-70B*, and *gpt-5.2* for S1, alongside those from *Llama-3.3-70B*, *gpt-5.2*, and *gemini-2.5-pro* for S2, are advanced to the next stage. The effectiveness of these specific generated codes in inducing targeted hardware stress is analysed in the following section.

B. Stress-oriented evaluations

Table II and Table III report the telemetry and profiling results for all valid applications generated in S1 and S2, respectively, alongside manually implemented GPU workloads. Each row corresponds either to an LLM agent (identified by the underlying model) that successfully generated a valid application, or to a human-designed benchmark. The columns report the metrics defined in Section IV-B.

Table II reports the results for the S1 campaign. In particular, *gpt-5.2* achieves the highest temperature (58.11 °C), while *gpt-oss-120B* attains the highest energy consumption (4,846.69 J). These values significantly exceed the thermal and energy profiles of the GPU-Rodinia benchmarks, whose workloads exhibit heterogeneous behavior and generally lower energy consumption. Notably, the LLM-generated stress applications are substantially more compact than GPU-Burn (733 LoC). Indeed, *gpt-oss-120B* produces a valid stress workload in only 144 LoC, and other models remain below 500 LoC.

Table III reports the results for S2. In this scenario, GPU-Burn represents the primary reference point, together with a baseline matrix-multiplication implementation ($M \times M$). The refined variants achieve peak temperatures of from 51.61 °C to 53.16 °C, with corresponding energy consumptions of from 3,035.54 J to 3,482.01 J, whereas GPU-Burn reaches 59.8 °C and 5,700 J under the same conditions. In particular, *gpt-5.2* achieves a throughput of 99.54%, exceeding GPU-Burn's 86.91%, but at the cost of a substantial increase in pipeline stalls (59.93% versus 7.89%).

Moreover, the refined implementations are consistently more verbose than the original GPU-Burn (up to 1,017 LoC), reflecting the shift from cuBLAS-based high-level routines to explicitly generated PTX code. Compared to the simple $M \times M$ baseline, all LLM-refined variants achieve significantly higher throughput and improved thermal/energy profiles, confirming the effectiveness of the refinement loop in exploring more computationally intensive execution paths.

The results demonstrate that LLM-driven multi-agent systems can automatically generate GPU stress applications that, in S1, reach up to 89.9% of the temperature and 77.2% of the energy consumption achieved by GPU-Burn, while often requiring substantially fewer LoC. In S2, LLM-based refinement can surpass GPU-Burn in throughput, at the cost of increased stalls and code complexity. These findings confirm that automated LLM-based program generation not only reduces development effort and time but also produces stress workloads capable of approaching state-of-the-art manually engineered solutions, with a more efficient code.

VI. CONCLUSIONS AND FUTURE WORKS

In this work, we presented the first empirical study exploring the use of LLMs for the automated generation of GPU stress-testing applications. The proposed framework integrates code generation, debugging, and performance-guided refinement, leveraging on a multi-agent architecture supported by real-time hardware feedback, enabling systematic and scalable exploration of stress-inducing behaviours. Experimental results demonstrate that LLM-generated GPU workloads are consistently compilable and executable, proving that LLMs are viable AI-based agents for generating stress-oriented workloads, substantially reducing manual development time. In particular, the generated applications often resulted in substantially more compact kernels, reflecting a lower structural complexity, while still allowing the best-performing LLMs in the application-generation scenario to reach up to 89.9% of the temperature and 77.2% of the energy consumption achieved by GPU-Burn.

Future research will focus on integrating reinforcement-learning-based optimisation to further improve the LLMs generation capabilities of specialized programs for stress testing, and extending the experiments to additional hardware architectures.

REFERENCES

[1] D. De Sensi *et al.*, "Exploring gpu-to-gpu communication: Insights into supercomputer interconnects," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024, pp. 1–15.

[2] M. Shamsa *et al.*, "Defect mechanisms responsible for silent data errors," in *2024 IEEE International Reliability Physics Symposium (IRPS)*. IEEE, 2024, pp. 1–5.

[3] H. D. Dixit *et al.*, "Silent data corruptions at scale," *arXiv preprint arXiv:2102.11245*, 2021.

[4] G. Ostrouchov *et al.*, "Gpu lifetimes on titan supercomputer: Survival analysis and reliability," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[5] B. Bittel *et al.*, "Data center silent data errors: Implications to artificial intelligence workloads & mitigations," in *2024 IEEE International Reliability Physics Symposium (IRPS)*. IEEE, 2024, pp. 1–5.

[6] D. Defour *et al.*, "Gpuburn: A system to test and mitigate gpu hardware failures," in *SAMOS*, 2013, pp. 263–270.

[7] NVIDIA, "Data center gpu manager (dcgm)," 2023. [Online]. Available: https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/index.html

[8] H. Dattatraya, "Detecting silent errors in the wild: Combining two novel approaches to quickly detect silent data corruptions at scale," 2022. [Online]. Available: https://engineering.fb.com/2022/03/17/production-engineering/silent-errors/

[9] H. D. Dixit *et al.*, "Detecting silent data corruptions in the wild," *arXiv preprint arXiv:2203.08989*, 2022.

[10] S. Che *et al.*, "Characterization of the rodinia benchmark suite," in *IISWC*, 2010, pp. 1–11.

[11] D. Defour *et al.*, "Gpu-burn repository," 2013. [Online]. Available: https://github.com/wilicc/gpu-burn

[12] M. Chen *et al.*, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[13] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," 2023. [Online]. Available: https://arxiv.org/abs/2201.11903

[14] L. Ouyang *et al.*, "Training language models to follow instructions with human feedback," 2022. [Online]. Available: https://arxiv.org/abs/2203.02155

[15] A. Neumann *et al.*, "Position is power: System prompts as a mechanism of bias in large language models (llms)," in *ACM Conference on Fairness, Accountability, and Transparency*, ser. FAccT '25, 2025, p. 573–598.

[16] Z. Xi *et al.*, "The rise and potential of large language model based agents: A survey," 2023. [Online]. Available: https://arxiv.org/abs/2309.07864

[17] C. Qian *et al.*, "Chatdev: Communicative agents for software development," 2024. [Online]. Available: https://arxiv.org/abs/2307.07924

[18] S. Hong *et al.*, "Metagpt: Meta programming for a multi-agent collaborative framework," 2024. [Online]. Available: https://arxiv.org/abs/2308.00352

[19] J. S. Park *et al.*, "Generative agents: Interactive simulacra of human behavior," 2023. [Online]. Available: https://arxiv.org/abs/2304.03442

[20] K. Zhang *et al.*, "CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges," in *62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku *et al.*, Eds. Association for Computational Linguistics, 2024, pp. 13 643–13 658.

[21] N. Shinn *et al.*, "Reflexion: Language agents with verbal reinforcement learning," *neurIPS*, 2023.

[22] Y. Dong *et al.*, "Self-collaboration code generation via chatgpt," 2024. [Online]. Available: https://arxiv.org/abs/2304.07590

[23] J. Bai *et al.*, "Polo: An llm-powered project-level code performance optimization framework," in *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, J. Kwok, Ed. International Joint Conferences on Artificial Intelligence Organization, 8 2025, pp. 7319–7328, main Track. [Online]. Available: https://doi.org/10.24963/ijcai.2025/814

[24] Y. Peng *et al.*, "Perfcodegen: Improving performance of llm generated code with execution feedback," 2024. [Online]. Available: https://arxiv.org/abs/2412.03578

[25] X. Chen *et al.*, "Teaching large language models to self-debug," 2023. [Online]. Available: https://arxiv.org/abs/2304.05128

[26] G. Esposito *et al.*, "Gpu under pressure: Estimating application's stress via telemetry and performance counters," *arXiv preprint arXiv:2511.05067*, 2025.

[27] J.-D. Guerrero-Balaguera *et al.*, "Stls for gpus: Using high-level language approaches," *IEEE Design Test*, vol. 40, no. 4, pp. 51–60, 2023.

[28] J. E. Rodriguez Condia *et al.*, "Using stls for effective in-field test of gpus," *IEEE Design Test*, vol. 40, no. 2, pp. 109–117, 2023.

[29] K. Devarajegowda *et al.*, "On generation of properties from specification," in *2017 IEEE International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 2017, pp. 95–98.

[30] J. Chen *et al.*, "cupilot: A strategy-coordinated multi-agent framework for cuda kernel evolution," *arXiv preprint arXiv:2512.16465*, 2025.

[31] P. Guo *et al.*, "Evoengineer: Mastering automated cuda kernel code evolution with large language models," *arXiv preprint arXiv:2510.03760*, 2025.

[32] A. Madaan *et al.*, "Self-refine: Iterative refinement with self-feedback," 2023. [Online]. Available: https://arxiv.org/abs/2303.17651

[33] A. Gensale *et al.*, "prompts-to-pressure," 2026. [Online]. Available: https://github.com/auro736/prompts-to-pressure

[34] X. Mei *et al.*, "A survey and measurement study of gpu dvfs on energy conservation," *Digital Communications and Networks*, vol. 3, no. 2, pp. 89–100, 2017.

[35] A. S. Barb *et al.*, "A statistical study of the relevance of lines of code measures in software projects," *Innovations in Systems and Software Engineering*, vol. 10, no. 4, pp. 243–260, 2014.

[36] J. Kaplan *et al.*, "Scaling laws for neural language models," 2020. [Online]. Available: https://arxiv.org/abs/2001.08361

[37] J. Wei *et al.*, "Emergent abilities of large language models," 2022. [Online]. Available: https://arxiv.org/abs/2206.07682

[38] M. M. Hasan *et al.*, "Assessing small language models for code generation: An empirical study with benchmarks," 2026. [Online]. Available: https://arxiv.org/abs/2507.03160