Invited Paper: Software/Hardware Co-design for LLM and Its Application for Design Verification

Lily Jiaxin Wan[1*], Yingbing Huang[1*], Yuhong Li[1], Hanchen Ye[1], Jinghua Wang[1], Xiaofan Zhang[2], Deming Chen[1]
[1]*University of Illinois Urbana-Champaign,* [2]*Google*
{wan25, yh21, leeyh, hanchen8, jinghua3, dchen}@illinois.edu, xiaofanz@google.com

***Abstract*—The widespread adoption of Large Language Models (LLMs) is impeded by their demanding compute and memory resources. The first task of this paper is to explore optimization strategies to expedite LLMs, including quantization, pruning, and operation-level optimizations. One unique direction is to optimize LLM inference through novel software/hardware co-design methods. Given the accelerated LLMs, the second task of this paper is to study LLMs' performance in the usage scenario of circuit design and verification. Specifically, we place a particular emphasis on functional verification. Through automated prompt engineering, we harness the capabilities of the established LLM, GPT-4, to generate High-Level Synthesis (HLS) designs with predefined errors based on 11 open-source synthesizable HLS benchmark suites. This dataset is a comprehensive collection of over 1000 function-level designs, and each of which is afflicted with up to 45 distinct combinations of defects injected into the source code. This dataset, named Chrysalis, expands upon what's available in current HLS error models, offering a rich resource for training to improve how LLMs debug code. The dataset can be accessed at: https://github.com/UIUC-ChenLab/Chrysalis-HLS.**

***Index Terms*—Large Language Models, software/hardware co-design, functional verification**

I. Introduction

The rapid evolution of machine learning, particularly through the advancements in neural network architectures, has precipitated significant breakthroughs in diverse fields, encompassing computer vision [1] and natural language processing [2]. Among various neural network designs, the transformer architecture [3] stands out, offering unparalleled performance on sequence-to-sequence tasks. Instead of using traditional recurrent layers, this innovative structure harness the power of the attention mechanism. The transformer model serves as the foundation for the emergence of Large Language Models (LLMs) such as OpenAI's GPT series [4], Meta's LLaMA [5], and Google's BARD [6]. Encompassing billions of parameters and informed by extensive textual datasets sourced from the internet, these models possess the capability to interpret human-generated text and yield contextually pertinent and logically coherent outputs to a wide spectrum of prompts.

In the domain of Electronic Design Automation (EDA), the potential application of LLMs is becoming increasingly evident. They are poised to support hardware engineers in various hardware design stages, ranging from the initial conception and verification to optimization and coordination of the complete design flow. Chip-Chat [7] established a set of eight foundational benchmarks, aiming to delineate both the capabilities and limitations of current state-of-the-art (SOTA) LLMs in hardware design. Concurrently, there is a trend in the academic circles towards the development of robust benchmark frameworks to rigorously gauge LLM performance. For instance, [8] introduced an open-source suite comprising 30 intricate hardware designs. Their contribution enhanced LLMs' feedbacks quality through advanced prompt engineering techniques. In a related effort, Shailja et al. [9] fine-tuned the CodeGen model [10] using 17 Verilog codes. Additional research has focused on dataset construction through practical RTL tutorial exercises. For instance, VerilogEval [11] developed a comprehensive benchmark framework that includes 156 problems derived from the educational HDLBits platform. Moreover, this framework was employed to fine-tune the CodeGen model [10], significantly improving its proficiency in generating RTL codes. In a distinct approach, ChatEDA [12] was architected to generate codes specifically for conjuring codes capable of navigating EDA tools based on natural language cues.

While LLMs benefit from a vast number of parameters, they also grapple with challenges related to sparsity and computational overhead. Because of the extensive usage of LLMs in time-sensitive applications like Internet of things (IoT) devices, it is essential to ensure that LLMs deliver optimal inference performance without compromising on the multi-task solving and language generation ability. In this paper, we explore several cutting-edge optimization techniques for LLM inference, with a focus on quantization and pruning. Numerous studies have highlighted the potential advantages of these methods individually, including activation outliers handling, structural and contextual sparsity reduction. It is still challenging to achieve consistent optimization benefits across diverse LLMs and ensure adaptability in real-world scenarios. Thus, we point out the potential solution in software/hardware co-design for this challenge, inspired by its proved power and effectiveness in the areas of Deep Neural Networks (DNNs) [13]–[16], Graph Neural Networks (GNNs) [17], and conventional machine-learning solutions [18].

Overall, this paper introduces a unified optimization framework for LLMs, with a particular emphasis on functional verification in EDA with the following contributions:

- We study the integration of both LLM quantization and pruning techniques, yielding greater benefits than their

[*]These authors contributed equally to this work.

standalone applications while compensating for their respective limitations. Additionally, we anticipate the potential integration of this approach with domain-specific hardware accelerators.

- We pioneer an innovative methodology that harnesses the capabilities of GPT-4 to inject bugs into HLS codes. We design a set of tailored prompts to guide GPT-4 in generating consistent and compliant buggy codes within the EDA domain.
- Leveraging the above methodologies, we create the Chrysalis dataset that includes both correct source codes and intentionally injected buggy codes. This dataset is meticulously organized, comprising over 1000 function-level designs sourced from 11 open-source synthesizable HLS benchmark suites. Each design undergoes a controlled injection of up to 45 distinct combinations of bugs. It represents an indispensable tool for the assessment and refinement of LLM-based HLS domain-specific debugging assistants.

II. LLM Acceleration

A. Challenges

The immense scale of LLMs, often encompassing billions or trillions of parameters, necessitates significant compute and memory resources for both training and inference. This not only amplifies energy consumption but also poses a major obstacle for time-sensitive or real-time applications using LLMs. From our benchmark analysis of the time consumption of Vicuna-7B in Fig. 1 and 2, LLMs suffer from different aspects during training and inference. In this section, we will focus on optimizing LLMs' inference that could provide an intuition to more efficiently leverage the power of existing LLMs such as GPT-4 and LLaMA2. For inference, as shown in Fig. 2, the Multi-layer Perceptron (MLP) predominantly dictate the latency, regardless of input length. Additionally, the latency attributed to attention layers grows more pronounced as input lengths increase. In this section, we aim to illustrate that although existing methods have considerably optimized these two parts, there remains a need for more advanced software/hardware co-design methods to fully leverage the capabilities of hardware platforms.

B. Existing Methods

Pruning and quantization are key techniques in optimizing neural networks, particularly in addressing the computational challenges presented by LLMs. These methods aim to reduce the size of models and the computational demands during inference without significantly compromising performance.

1) Quantization of LLMs: Besides the benefit from memory and inference latency reduction, quantization's potential for greater parallelism taps into the capabilities of hardware accelerators like FPGAs, further amplifying throughput. LLM.int8 [19] develops a two-part quantization procedure with vector-wise quantization and a new mixed-precision decomposition scheme. It could preserve perplexity for models with 125M to 13B parameters at the cost of longer inference latency. SmoothQuant [20] uses a per-channel smoothing factor

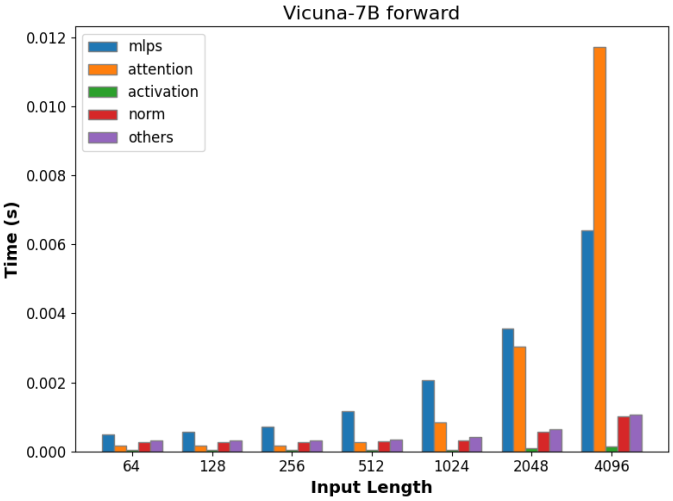

Fig. 1. Time consumption breakdown: a Llama-2-7b [5] decoder layer during training. Attention significantly dominates the latency when sequence length becomes longer.

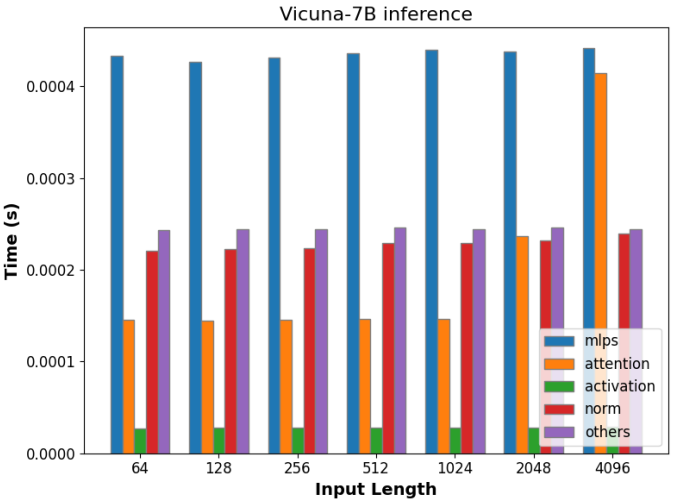

Fig. 2. Time consumption breakdown: a decoder layer of Llama-2-7b during inference. The MLPs consistently take up the most time across all input lengths. The attention mechanism and normalization operations scale with input length. Their computational times increase notably as the input becomes longer, highlighting their sensitivity to input size.

to handle outliers in activations and achieves lower latency compared to FP16. The results demonstrate that SmoothQuant can match the FP16 accuracy with INT8 quantization across various LLM sizes up to 530B parameters.

2) Pruning of LLMs: A pruned model not only reduces memory requirements but also accelerates inference. LLM-Pruner [21] uses structural pruning by dividing weights into independent groups and employs the low-rank approximation for recovery. At the same time, it preserves the diverse capacities of LLMs. SparseGPT [22] solves the Row-Hessian challenge, the computational difficulty of calculating and storing individual rows of the Hessian matrix, by reusing Hessians between rows and distinct pruning masks with negligible accuracy drop. However, SparseGPT only demonstrated its fine-tuning-free inference speed on a narrow range of large models like OPT 175B and BLOOM 176B. Deja Vu [23] reveals the existence of

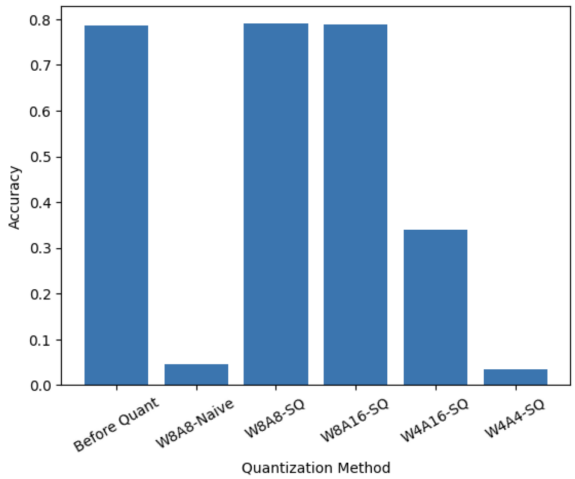

Fig. 3. OPT-13b's accuracy on Lambada Dataset with different quantization methods. W8A8-Naive denotes the naive 8-bits weights and activation quantization of LLM.int8. W4A4-SQ denotes 4-bits weights and activation quantization of SmoothQuant.

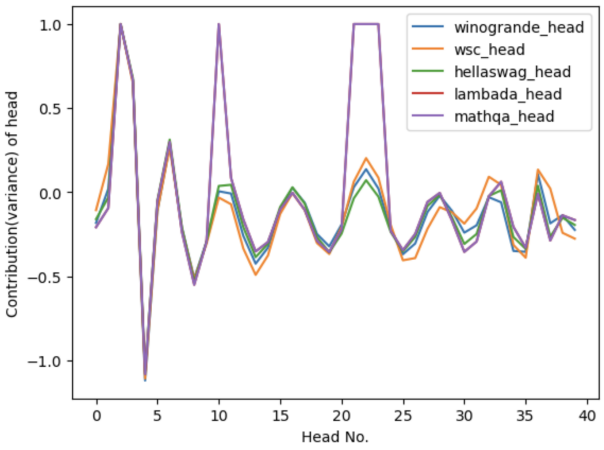

Fig. 4. The activeness of heads profiling on different datasets. Here, we measure the contribution of different heads based on their variance over input sequence. Inactive heads show low variance, which eventually leads to contextual sparsity.

contextual sparsity and proposes a real-time sparsity predictor trained during inference, which can reduce the inference latency of OPT-175B. The evaluations on OPT-66B and 175B show no compromising quality of the models.

C. Software/Hardware Co-design

1) Motivations: Beyond the evident advantages of pruning and quantization in LLMs, the complexities of their effective implementation remain. The core challenge with quantization lies in preserving the model's accuracy. A reduction in precision can compromise accuracy. Likewise, excessive pruning can significantly diminish model accuracy. Additionally, the irregular sparsity introduced by pruning may not align well with specific hardware architectures, potentially resulting in less-than-ideal performance enhancements. SmoothQuant provides the turn-key solution. However, achieving consistent performance in LLMs with 4-bit quantization remains challenging as shown in Fig. 3, despite claims of its universal optimality proved by other previous work [24]. We tested the performance of SmoothQuant on OPT-13B [25]. Fig. 3 illustrates the accuracy levels SmoothQuant can attain in comparison to FP16, but W4A4 (4-bits quantization on weights and activation) and W4A16 cannot achieve the comparable accuracy with the original model before quantization. After testing with different values of scale hyperparameter α, the accuracy of these two models could vary between 0 to 0.3. These two models are sensitive to the hyperparameter and need further investigation.

At the same time, the existing pruning methods mostly test on models with parameters over 175B. These large models are always under speculation of under-training, resulting in a high percentage of inactive parameters in the first place. Deja Vu only demonstrates the preservation of accuracy on these large models, showing insufficient evidence on its effectiveness regarding smaller models. Based on our experimental results, after pruning out 50% parameters on attention layers and 30% on MLP layers, OPT-13B model's accuracy drops approximately 15% on WinoGrande dataset. Meanwhile, the existing pruning methods working with smaller LLMs overlooks their generalization abilities over data in different domains. For example, LLM-Pruner runs the risk of overfitting in recovery stage after pruning.

2) Our Ideas: From experiments involving state-of-the-art LLMs, solely relying on either quantization or pruning proves challenging in meeting real-world requirements and adapting to diverse LLMs and hardware platforms. This indicates the potential benefits of integrating these two prevalent techniques while considering hardware characteristics. Thus, we suggest exploring pruning-aware quantization for LLM optimization. As indicated in [23] and our experiments on OPT-13B, inactive attention heads are uniformly distributed across the input sequence, referred to as the contextual sparsity of LLMs. A typical strategy to capitalize on this sparsity is pruning. However, different LLMs often exhibit varied patterns of contextual sparsity, constraining the effectiveness of conventional pruning. At the same time, quantization is sensitive to the range and distribution of parameters, making the standalone quantization precision a non-ideal solution.

Based on these considerations, our pruning-aware quantization method aims to tackle these challenges by choosing pruning or quantization precisions according to the behavior of different LLMs over various datasets. Specifically, pruning-aware quantization method will profile over relatively small amount of dataset and identify the importance and scale of parameters in attention matrix and neurons. Subsequently, guided by the profiling results, our approach can select between pruning (0 bit) or varying quantization precisions (4, 8, and 16 bits) for each layer. Additionally, these choices are tailored to align with the efficient computation patterns of different hardware architectures, achieving the co-optimization of both software and hardware. Our approach can reduce the memory and computational cost while achieving enhanced inference accuracy than existing pruning methods. Additionally, as shown in Fig. 4, the activeness of attention heads exhibits similar distribution over multiple datasets, which suggests the preservation

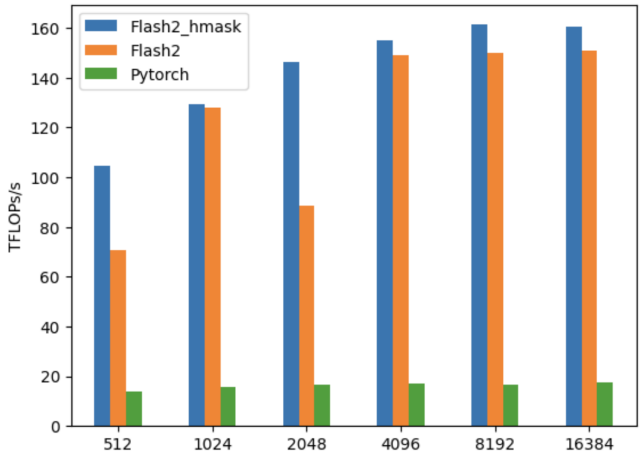

Fig. 5. The preliminary result from forward throughput improvement. Flash2_hmask is the result from the combination of FlashAttention2 and our pruning-aware quantization approach.

of multi-task solving and language generation ability of our approach. Furthermore, our approach can also be combined with the state-of-art hardware-aware LLM acceleration frameworks, such as FlashAttention-2 [26]. Based on the preliminary results illustrated in Fig. 5, we are able to achieve higher throughput than both PyTorch and standalone FlashAttention-2.

III. LLM For Design Verification

A. Challenges

Circuit design involves stages such as RTL synthesis, logic synthesis, and placement and routing [27]. Most of these stages are equipped with their corresponding verification processes to ensure the implemented hardware matches its specifications. Verification methods fall into two main categories: formal verification and simulation-based verification [28]. While formal verification provides mathematical assurance, its limitations, like scalability issues and the necessity for niche coding skills, often position simulation-based verification as the industry's preferred method.

Nonetheless, simulation-based verification has its shortcomings as it is very time-consuming especially for large-scale hardware designs. Verification can account for 45% to 55% of the total design cycle [29], making it the most extensive phase in hardware development. Moore's Law [30] suggests that transistor counts in integrated circuits (ICs) double approximately every two years. For reference, the transistors of the AMD Ryzen 9 series processor [31] [32] scaled up by 1.78 times from 2019 to 2021. Yet, as depicted in Fig. 6 [33], manual hardware design productivity (quantified in Gates/Day) has not paralleled these technological leaps. This divergence reveals an expanding productivity gap, intensified by growing system intricacies and nearing physical property limitations.

Traditional hardware simulation-based functional verification uses test vectors, as shown in Fig. 7, to ensure a system's expected behavior. Engineers often manually design test vectors, create test benches, and set legality constraints. After reviewing coverage reports, they adjust parameters and continue to the

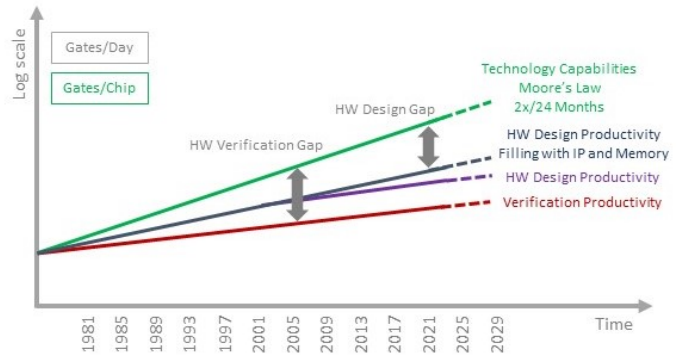

Fig. 6. The great productivity gap between hardware design(purple line) productivity, verification(red line) productivity and technology capacity(green line) [33].

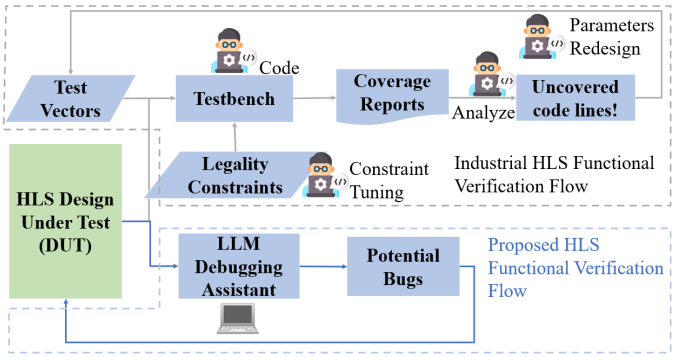

Fig. 7. Comparison of LLM-Based vs. Traditional HLS Functional Verification Flows.

next test iteration. The process of crafting these test vectors, though reliable to a certain extent, has always been labor-intensive, requiring significant domain expertise. Moreover, the growing complexity of modern hardware makes exhaustive state examination computationally unviable due to the vast search space. Recognizing these limitations, our approach harnesses the power of LLMs to detect common bug patterns that humans easily get stuck, automatically on the top of the source code.

In light of the aforementioned challenges of the widening productivity gap and the increasing complexity of hardware verification, HLS enhances the design verification landscape. HLS, operating at an elevated abstraction level, facilitates streamlined bug identification in intricate designs by emphasizing algorithmic and architectural considerations, harmonizing with the LLM's strength in context comprehension. Furthermore, HLS abbreviates the design-to-verification trajectory, fostering swift issue recognition and expediting debugging. Collectively, HLS furnishes engineers with an optimized, intuitive, and holistic verification platform.

B. LLM-based Chrysalis Dataset Generation

Within our functional verification framework, we employ the power of LLMs to streamline the debugging process, commencing with the precise localization of erroneous code lines. A foundational requirement for this approach is the establishment of an LLM-centric dataset which is crafted specifically to

facilitate a robust evaluation or finetuning of varying LLMs in the domain of HLS code debugging tasks. By deliberately introducing known bugs and monitoring an LLM's prowess in pinpointing them, we are empowered to assess its efficiency, accuracy, and reliability comprehensively. This assessment does not just offer a window into the diagnostic capabilities of the LLM; it is also useful in refining a domain-specific LLM.

Historically, the research community has grappled with the absence of an open-source dataset catering specifically to buggy HLS code. To address this gap, we have created the Chrysalis dataset, a comprehensive benchmark tailored to catalyze synergistic advancements between the LLM and HLS domains. The dataset is named "Chrysalis" to metaphorically represent the evolution of buggy code, akin to the transformative chrysalis stage in a butterfly's life cycle, culminating in the emergence of refined code. This dataset is exclusively crafted to underpin LLM-guided HLS debugging endeavors targeting FPGA platforms.

To closely mirror genuine, real-world coding errors often overlooked by engineers, we have enumerated a series of logical bugs. These particular bugs are crafted to elude detection by conventional HLS synthesizing and compiling tools. We utilize GPT-4's [34] nuanced natural language capabilities, combined with HLS source code and curated prompts, to enhance our dataset creation. Our dataset is a comprehensive collection, offering both flawless benchmark suites and those with injected bugs. It features detailed classifications of errors and includes precise annotations. These annotations trace the exact origins of the faults within the code. Our objective is to embed one or two specific bug types in each benchmark, targeting up to 45 unique scenarios: 9 with a single bug type and 36 with two bug types. If certain bug types are infeasible to be injected for one particular benchmark, the total number of buggy code instances may be fewer than 45. To augment the variety of errors, variables can be systematically varied to spawn a multitude of faulty code samples.

The steps of generating the Chrysalis dataset include HLS design collection, logic bugs simulation and injection, and bug injection validation. The following sub-sections elucidate the detailed implementation strategies underpinning our Chrysalis dataset creation.

1) HLS Design Collection: For the creation of our Chrysalis dataset, we have meticulously curated a collection of real-world HLS applications based on open-source projects. This comprehensive benchmark suites encompass a diverse array of synthesizable HLS applications, each drawn from the following reputable sources: FINN [35], GNNBuilder [36], H.264 [37], HLS4ML [38], MachSuite [39], Open-Source-IPs [40], Polybench [41], Rosetta [42], Vitis HLS introductory examples [43], Vitis libraries [44], and Tacle-Bench [45]. Our Chrysalis dataset primarily focuses on function-level tasks, with over 1,000 individual HLS programs extracted from these reputable sources. This selection ensures the representation of various coding styles and complexities.

While constructing the Chrysalis dataset, we consider the intricacies of handling functions that include header files or making calls to other functions. We recognize that engineers may occasionally overlook critical values or functions, potentially resulting in inter-functional errors. In order to facilitate a comprehensive understanding of the contexts in which errors can be injected, we have incorporated all instances of the `#define` syntax, ensuring that the entirety of each function's macro definitions is accounted for. Additionally, in each function-level design, we have incorporated details of all the functions that a particular function calls, ensuring comprehensive and clear documentation. This holistic approach ensures that LLM is well-equipped to comprehend the full scope of interdependencies and contextual intricacies, thereby enhancing its effectiveness in error injection tasks.

By adhering to this rigorous procedure, our methodology in developing the Chrysalis dataset serves a dual purpose. Firstly, it exemplifies a systematic and automated method for LLM-targeted dataset generation. Secondly, it serves as a valuable resource for advancing the capabilities of the LLMs' capacity in the realm of HLS verification.

2) Logic Bugs Simulation: After thoroughly analyzing the source code and extracting all functions, our next step is to simulate pre-silicon logic bugs. These are the kind of errors hardware designers might inadvertently introduce when crafting the HLS version of a design, leading to deviations from the intended specifications. The error types employed in our work align closely with the categorization introduced in [46], and we categorize the potential error types as follows: (1) OOB: Out-of-bounds array access; (2) INIT: Accessing an uninitialized variable; (3) SHFT: Bit shift by an out-of-bounds amount; (4) INF: An infinite loop arising from an incorrect loop termination condition; (5) *++: Misunderstanding of operator precedence, erroneously assuming that dereference (*) has higher precedence than postincrement (++); (6) MLU: Errors in manual loop unrolling, leading to the omission of one iteration; (7) BUF: Copying from the wrong half of a split buffer; (8) ZERO: A variable initialized to zero when it should have a nonzero initializer; (9) USE: Unintended sign extension.

To emulate real-world buggy scenarios, each function-level design has one or two of these errors injected. Through this process, we create a dataset comprising over 1,000 function-level designs, each including 1-2 buggy instances out of 45 possible combinations. This comprehensive dataset will provide a robust platform to evaluate the performance and accuracy of LLM-based debugging tools.

3) LLM-Driven Bug Injection Methodology for HLS Code: Capitalizing on the accurate code version combined with a defined error type, we harness the capabilities of GPT-4 for a precise and systematic introduction of errors into HLS design structures. The foundation of our methodology rests on a refined prompt template, fashioned to facilitate GPT-4 in generating erroneous code with both stability and predictability. The template is strategically partitioned into three key sections:

- **Context:** This section offers a snapshot of the surrounding environment where the HLS code is set to operate. It emphasizes the essence of studying inadvertent human-induced errors that mirror real-world scenarios.

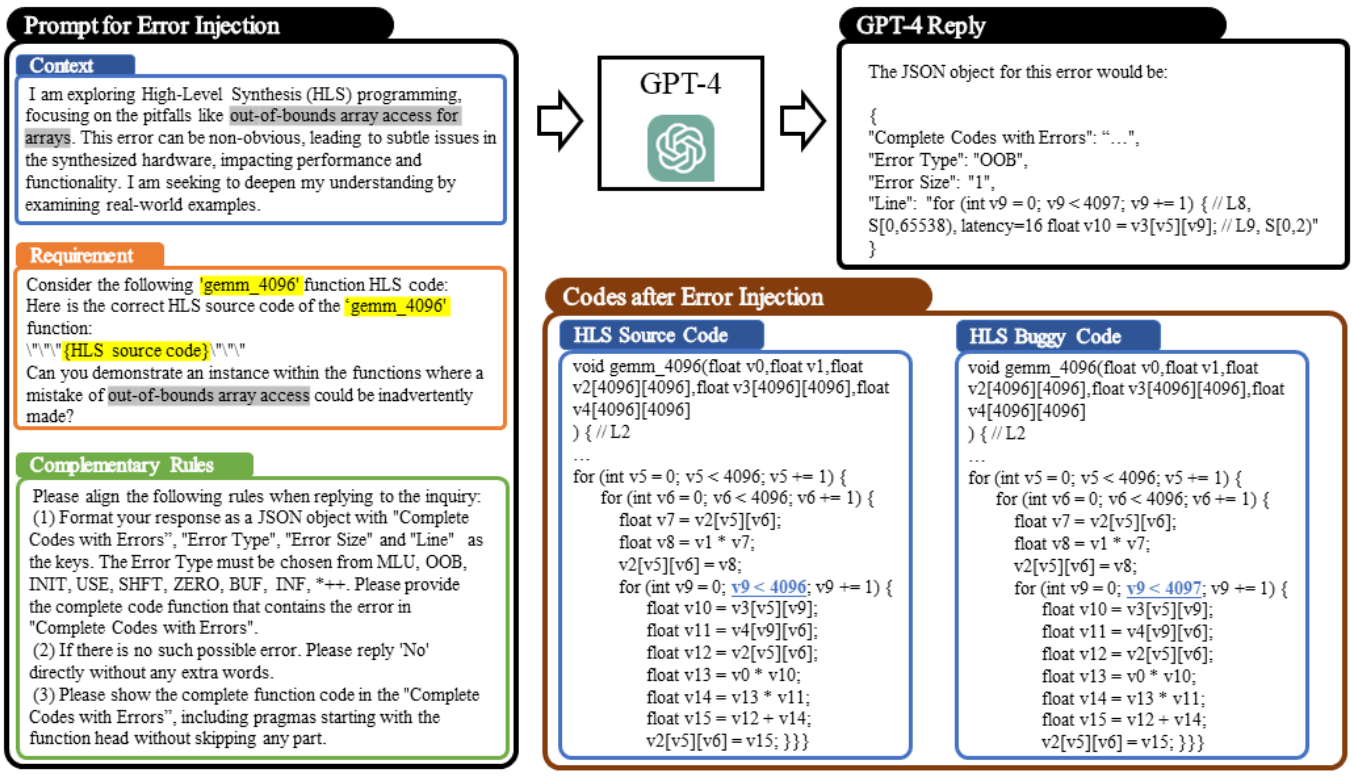

Fig. 8. An illustration showcasing the procedure for directing GPT-4 to introduce an Out-Of-Bounds (OOB) error into the 'gemm_4096' source function. The code line emphasized with a blue underline indicates the injection point of the error. For an automated generation of multiple erroneous function codes, the terms highlighted in yellow must be substituted with the appropriate code identifiers and contents. To introduce varying error types, the descriptors highlighted in grey should be adjusted accordingly.

- **Requirement:** This delineates the characteristics of the intended error or bug, offering a comprehensive insight into its nature. Beyond just a mere directive for error generation, it elucidates the rationale behind the need for such errors, providing a deeper understanding of the specific anomalies being introduced.
- **Complementary Rules:** Serving as a structured framework, these rules ensure the synthesized bugs are not only consistent with the original intention but also don't stray from the designated error category. To streamline the process, outputs are structured in a JSON file format, encompassing fields like error type, and the content of the faulty code line, among others, for automated parsing. Moreover, to eschew exhaustive and aimless fault injection, the system is programmed to directly output 'No' if the conditions are unsuitable for the stipulated error manifestation.

In each function-level task within our Chrysalis dataset, we introduce one to two types of errors. This strategy is specifically devised to emulate the nuances of human oversight or negligence, ensuring the results resonate with real-world scenarios.

Fig. 8 showcases an example of the process of injecting an OOB error into the function "gemm_4096". Upon introducing a bug into the system, we conduct a validation process to ascertain if the bug is correctly injected following the prompt. This will involve a comparative analysis between the original, error-free source code and the generated buggy code. To maintain the integrity of our findings, any redundant or recurring errors will be systematically eliminated.

C. LLM-Based Bug Detection

Our Chrysalis dataset presents a promising platform for evaluating the proficiency of existing LLMs in HLS bug localization. To be specific, engineers can assess the models' precision and efficiency by comparing the errors detected by the LLMs to predefined error labels. These labels provide detailed information, including the types of errors and the specific locations of the incorrect code lines.

We plan to develop a lightweight, domain-specific LLM trained on the Chrysalis dataset. Such a model would not merely detect anomalies but might also possess the capability to rectify them. This domain-centric LLM could integrate as an extension within development environments like VSCode. This would parallel tools such as Copilot, offering hardware engineers intuitive suggestions to identify and rectify pitfalls in their HLS codes, thus augmenting the debugging process.

IV. Conclusion

In this paper, we studied on the acceleration strategies of LLMs through software/hardware co-design and LLMs application in the domain of design verification, particularly focusing on functional verification in HLS. We have explored several

state-of-the-art techniques for expediting LLMs, such as quantization and pruning, and proposed an approach that synergizes these methods to overcome their individual limitations. Our preliminary results suggest that this integrated approach can yield considerable improvements in inference performance without compromising accuracy, indicating a promising direction for future research.

Furthermore, we have addressed the critical challenge of functional verification in hardware design, which has become increasingly complex and time-consuming. By leveraging the capabilities of LLMs, specifically GPT-4, we have created a comprehensive Chrysalis dataset for HLS that includes over 1000 function-level designs with up to 45 distinct combinations of defects. This Chrysalis dataset serves as a valuable resource for evaluating and fine-tuning LLM-based HLS domain-specific debugging assistants.

In conclusion, our research contributes a methodological foundation and a practical toolkit for harnessing the power of LLMs in the design verification domain. As we continue to refine these techniques and expand the capabilities of LLMs, we anticipate a future where the co-design of software and hardware is seamlessly interwoven into the fabric of hardware development, leading to faster, more efficient, and error-resilient design cycles.

ACKNOWLEDGEMENTS

This work is supported in part by IBM-Illinois Discovery Accelerator Institute (IIDAI), NSF 2117997 grant through the A3D3 institute, and Semiconductor Research Corporation (SRC) 2023-CT-3175 grant.

REFERENCES

[1] Y. Li *et al.*, "Csrnet: Dilated convolutional neural networks for understanding the highly congested scenes," in *Proc. of CVPR*, 2018.
[2] Y. Goldberg, "A primer on neural network models for natural language processing," *Journal of Artificial Intelligence Research*, 2016.
[3] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, 2017.
[4] L. Ouyang *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, 2022.
[5] H. Touvron *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.
[6] S. Pichai, "An important next step on our AI journey," 2023.
[7] J. Blocklove *et al.*, "Chip-Chat: Challenges and Opportunities in Conversational Hardware Design," *arXiv preprint arXiv:2305.13243*, 2023.
[8] Y. Lu *et al.*, "RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model," *arXiv preprint arXiv:2308.05345*, 2023.
[9] S. Thakur *et al.*, "Benchmarking Large Language Models for Automated Verilog RTL Code Generation," in *Proc. of DATE*, 2023.
[10] E. Nijkamp *et al.*, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.
[11] M. Liu *et al.*, "VerilogEval: Evaluating Large Language Models for Verilog Code Generation," *arXiv preprint arXiv:2309.07544*, 2023.
[12] Z. He *et al.*, "ChatEDA: A Large Language Model Powered Autonomous Agent for EDA," *arXiv preprint arXiv:2308.10204*, 2023.
[13] X. Zhang *et al.*, "DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs," in *Proc. of ICCAD*, 2018.
[14] H. Ye *et al.*, "HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation," in *Proc. of DAC*, 2020.
[15] X. Zhang *et al.*, "DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator," in *Proc. of ICCAD*, 2020.
[16] C. Zhuge *et al.*, "Face recognition with hybrid efficient convolution algorithms on FPGAs," in *Proc. of GLVLSI*, 2018.
[17] X. Chen *et al.*, "Thundergp: Hls-based graph processing framework on fpgas," in *Proc. of FPGA*, 2021.
[18] S. Liu *et al.*, "Real-time object tracking system on FPGAs," in *Proc. of SAAHPC*, 2011.
[19] T. Dettmers *et al.*, "Llm. int8 (): 8-bit matrix multiplication for transformers at scale," *arXiv preprint arXiv:2208.07339*, 2022.
[20] G. Xiao *et al.*, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *Proc. of ICML*, 2023.
[21] E. Frantar *et al.*, "SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot," 2023.
[22] X. Ma *et al.*, "LLM-Pruner: On the Structural Pruning of Large Language Models," *arXiv preprint arXiv:2305.11627*, 2023.
[23] Z. Liu *et al.*, "Deja Vu: Contextual Sparsity for Efficient LLMs at Inference Time," in *Proc. of ICML*, 2023.
[24] T. Dettmers *et al.*, "The case for 4-bit precision: k-bit inference scaling laws," in *Proc. of ICML*, 2023.
[25] S. Zhang *et al.*, "OPT: Open Pre-trained Transformer Language Models," 2022.
[26] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023.
[27] L. Lavagno *et al.*, *EDA for IC implementation, circuit design, and process technology*. CRC press, 2018.
[28] A. Piziali, *Functional verification coverage measurement and analysis*. Springer Science & Business Media, 2007.
[29] H. Foster, "Part 8: The 2020 wilson research group functional verification study," https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-study/, 2021.
[30] R. R. Schaller, "Moore's law: past, present and future," *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.
[31] P. Alcorn, "AMD Ryzen 9 3900X and Ryzen 7 3700X Review," https://www.tomshardware.com/reviews/ryzen-9-3900x-7-3700x-review,6214.html, 2019.
[32] TechPowerUp, "AMD Ryzen 7 5800H Specifications," https://www.techpowerup.com/cpu-specs/ryzen-7-5800h.c2368, 2021.
[33] R. Bahar *et al.*, "Workshops on Extreme Scale Design Automation (ESDA) challenges and opportunities for 2025 and beyond," *arXiv preprint arXiv:2005.01588*, 2020.
[34] OpenAI, "GPT-4 Technical Report," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.08774
[35] Y. Umuroglu *et al.*, "Finn: A framework for fast, scalable binarized neural network inference," in *Proc. of FPGA*, 2017.
[36] S. Abi-Karam *et al.*, "GNNBuilder: An Automated Framework for Generic Graph Neural Network Accelerator Generation, Simulation, and Optimization," *arXiv preprint arXiv:2303.16459*, 2023.
[37] X. Liu *et al.*, "High level synthesis of complex applications: An H. 264 video decoder," in *Proc. of FPGA*, 2016.
[38] F. Fahim *et al.*, "hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," *arXiv preprint arXiv:2103.05579*, 2021.
[39] B. Reagen *et al.*, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proc. of IISWC*, 2014.
[40] X. Liu *et al.*, "HLS based Open-Source IPs for Deep Neural Network Acceleration," https://github.com/DNN-Accelerators/Open-Source-IPs, 2019.
[41] J. Karimov *et al.*, "Polybench: The first benchmark for polystores," in *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence: 10th TPC Technology Conference, TPCTC 2018, Rio de Janeiro, Brazil, August 27–31, 2018, Revised Selected Papers 10*, 2019.
[42] Y. Zhou *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proc. of FPGA*, 2018.
[43] Xilinx, "Vitis-HLS-Introductory-Examples," https://github.com/Xilinx/Vitis-HLS-Introductory-Examples, 2023.
[44] Xilinx, "Vitis libraries," https://github.com/Xilinx/Vitis_Libraries, 2019.
[45] Tacle, "Tacle Bench," https://github.com/tacle/tacle-bench, 2017.
[46] K. A. Campbell, "Robust and reliable hardware accelerator design through high-level synthesis," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2017.