Supplementary Materials for "LLM-Driven Code Generation for Neural Networks on FPGAs: Bridging Python and HLS"

Rupesh Raj Karn, Johann Knechtel, Ramesh Karri, Ozgur Sinanoglu
Center for Cyber Security, New York University, Abu Dhabi, UAE.
Email: {rupesh.k, johann, rkarri, ozgursin}@nyu.edu

I. Preliminaries and Setting for This Work

A. Motivation for LLM-Driven Code Generation

LLMs have demonstrated significant potential in the field of code generation. By comprehending various programming languages, they can aid in writing, debugging, and optimizing code. This ability is especially valuable in complex areas such as hardware design for FPGAs and machine learning (ML), where efficient and optimized code is essential. In our work, we leverage the following capabilities of LLMs.

1) Code Translation: LLMs convert ML code from Python into HLS-compatible C/C++. This translation ensures that the generated code aligns with the NN architecture, including the forward pass, activation functions, and weights, while also meeting the constraints and requirements of FPGA synthesis.

2) Optimization: The generated code illustrates the hardware design of NNs that must be optimized for FPGA implementation, emphasizing aspects such as parallelism, memory management, and latency reduction. LLMs can help identify and apply these optimizations.

3) Evaluation and Refinement: The performance of the LLM-generated hardware design is assessed through synthesis tools. Based on the synthesis results, the LLM allows refinements to the design to improve efficiency and performance.

4) Error Resolution: The LLM detects syntax errors in the translated source code and proposes corrections to ensure smooth execution. Additionally, it assists in fixing the code by addressing compilation or runtime errors.

Table I presents a high-level comparison for LLM-driven versus manual approaches to hardware design for NNs.

TABLE I: Comparison of LLM Optimization vs Manual Approach

Aspect	LLM Approach	Manual Approach
Design Space Exploration (DSE)	Automatically explores a vast design space, generating multiple configurations and selecting the optimal one based on PPA or other objectives.	Relies on designer's experience and intuition, often limited to a smaller design space due to time and resource constraints.
Profiling and Bottleneck Identification	Automates profiling of tool reports, thereby identifying performance bottlenecks, and suggests targeted optimizations.	Manual profiling is time-consuming and may miss subtle bottlenecks, leading to suboptimal performance.
Adaptive Pragma Insertion	Dynamically inserts and adjusts HLS pragmas based on specific design requirements, enhancing parallelism and resource utilization.	Manual pragma insertion is prone to human error and may not fully exploit the potential for parallelism and optimization.

B. Different LLMs

In this study, we utilize three prominent LLMs: OpenAI [1], RapidGPT [2], [3], and Microsoft Co-pilot [4], [5].

1) OpenAI: OpenAI's models are renowned for their state-of-the-art performance in natural language understanding and generation. Trained on vast datasets from various domains, they produce coherent and contextually relevant code. We employ the *GPT-4* model for our HLS code generation.

2) RapidGPT: RapidGPT is a recent addition to the landscape of LLMs, specifically tailored for tasks involving hardware description, digital design, and FPGA technologies. It has been extensively trained on specialized datasets to enhance its proficiency in these areas.

3) Microsoft Co-pilot: This is another LLM that integrates with various Microsoft tools and services, including Visual Studio and GitHub. Like OpenAI's models, it is designed to be versatile across multiple domains.

C. Other Computing Applications

To distinguish LLM-driven code generation for NN architectures against other cases, we consider the following widely established applications:

1) Data Processing Applications: Tasks such as sorting, searching, and data transformation, which typically involve general-purpose algorithms and optimizations.

2) Algorithm Implementation: Implementing classical algorithms like graph algorithms (e.g., Dijkstra's, A*), and dynamic programming solutions.

3) System-Level Programming: Developing operating system components, device drivers, and embedded systems.

4) Signal Processing: Applications involving digital signal processing (DSP) tasks such as filtering, Fourier transforms, and modulation/demodulation techniques.

5) Cryptographic Algorithms: Implementing encryption and decryption algorithms, which require a focus on security, speed, and resource efficiency.

6) Traditional Machine Learning: Implementing traditional ML algorithms like decision trees, support vector machines, and k-means clustering.

II. Code Generation with LLMs

Algorithm 1 outlines this step. Once prompted with one of the four optimization strategies, the LLM autonomously

conducts the actual optimization: it selects relevant pragmas, performs loop unrolling, uses block RAM, decides the pipelining ratio, etc. Once the LLM generates the accordingly revised HLS code, the latter is synthesized using the *HLS tool*. Synthesis results are compiled into a report, which are fed back to the LLM, providing direct feedback for further optimization iterations.

Algorithm 1: Insert and Adjust HLS Pragmas Based on Optimization Prompt

Input: optimization_prompt
Output: HLS code adjusted for the given optimization prompt

```
switch Optimization prompt do
    case resources do
        // Optimize FPGA resource usage
        - Focus on resource sharing and leverage pragmas like
          BIND_STORAGE to control the mapping of operations to hardware
          resources.;
        - Apply loop unrolling and function inlining.;
        - Consider using array partitioning to improve memory access
          patterns and reduce the need for large memory blocks.;
    end
    case memory do
        // Minimize memory access latency and
           bandwidth requirements
        - Utilize on-chip memory;
        - Apply memory tiling;
        - Implement data reuse strategies;
    end
    case performance do
        // Accelerate NN computations
        - Apply array partitioning to allow parallel access to matrix and
          vector elements;
        - Use loop pipelining to improve throughput;
        - Use loop unrolling to parallelize computations;
        - Implement ReLU and softmax activation functions as inline
          functions;
    end
    case power do
        // Reduce power consumption without
           compromising performance
        - Use fixed-point data types instead of floating-point operations;
        - Use BIND_STORAGE pragmas to specify the use of Block RAM
          (BRAM) for storing matrices and vectors, which can be
          power-gated when not in use;
        - Apply PIPELINE pragma to improve throughput while allowing
          for clock gating;
        - Use UNROLL pragma with a small factor to balance parallelism and
          power consumption;
    end
end
```

An example prompt showcasing batch-based code generation is provided in Listing 1. It segments the HLS code into ”test_dataset.c”, ”nn.cpp”, ”weights.cpp”, and ”test.cpp”. First, the code is translated from Python to C. Then, ambiguities in C, such as dynamic array size, dynamic memory, and floating point, are converted to HLS-compatible C/C++.

Listing 1: Prompt for code generation/convertion from Python to C/C++ and HLS

```
1. Below is the Python code for training and validating the
     <dataset-name> dataset. Please save this code. Next,
     write a Python script to save the test dataset in a C
     file. The C file should contain two arrays: one for the
      data samples and another for the labels. Name the file
      "test_dataset.c".
2. Now, generate the following:
a. A single "nn.cpp" file with all the inference code for
     neural network inferencing based on the Python code.
b. A Python script that generates a "weights.cpp" file
     containing the neural network weights and biases. This
     file should be included in "nn.cpp".
c. A single "test.cpp" file that feeds samples from "
     test_dataset.c" to "nn.cpp" one by one and calculates
     the accuracy percentage.
3. Convert the C code in the files "nn.cpp" and "test.cpp"
     into high-level synthesis (HLS) C. Please note that
     Vivado HLS will be used to the RTL code.
```

III. Experimental Investigation

A. Discussion

While our LLM-driven approach may not surpass established optimizers such as FINN [6], Haddoc2 [7], DNNWeaver [8], FP-DNN [9], etc., the primary contribution of this work is the novel integration of LLMs into the code generation workflow. Traditional tools, while highly effective, often require significant manual effort and domain expertise to achieve optimal results. In contrast, we utilize generative capabilities of LLMs to automate HLS code generation, easing efforts for hardware designers and enabling rapid prototyping. Our work serves as proof-of-concept and highlights the potential of LLMs to complement existing tools. Future work shall extend LLM-driven code generation with hybrid approaches that incorporate the precision of traditional optimizers, ultimately bridging the gap between automation and performance.

Apart from the inherent complexity for NN hardware design, the intricacies of FPGA architectures, and the limitations of current LLMs for comprehension of CNN and GNN models, we observed further challenges that are worth discussing and should be addressed in future work.

First, we encountered several practical issues for C compilation and HLS synthesis, including mismatches when linking header files, unmatched datatypes, conflicts between C and C++ syntax, and unmatched array sizes. We experimented with both prompt engineering and stand-alone scripts to fix these issues. Full details are provided in our source code release. These issues highlight that, despite the promising results (especially through our proposed optimization method), LLMs still need to be improved for their HLS coding capabilities in general.

Second, given that the synthesis process (of HLS C/C++) is time-consuming, especially for large and complex designs, DSE becomes ever-more important. For example for the performance optimization profile/scenario, we observed that large pipelining ratios can cause a large number of solutions to be explored, resulting in overall runtimes of up to few hours. While synthesis time is only few minutes for FCNN architectures, for CNNs and GNNs this could takes hours for implementing/exploring a single solution. Therefore, to be able to optimize hardware design of such more complex models in the first place, we argue that LLMs should support DSE with as few iterations as possible. Such LLMs are not yet available, as seen by the failure of RapidGPT for GNN and CNN models; dedicated research efforts for related finetuning may overcome this bottleneck.

IV. Related Works

We separate the discussion of related works into two parts: 1) LLM-based RTL code generation and 2) FPGA implementation of NNs. Importantly, there is no prior art that specifically addresses both parts; ours is the first toward that end.

For LLM-based RTL code generation, we note the following. First, seminal works like [10], which do automatically generate RTL code also for FPGA implementations, are still not suitable for the task at hand, due to the large code sizes required for NN designs and the token restrictions in LLMs used by these tools. Second, works like [11], which do convert C to HLS C automatically, are also not applicable, as they often operate only with one-shot mechanisms. (As shown in Table III in main paper, one-shot mechanisms fail for complex NN architectures.)

For FPGA implementation of NNs, note that this is a well-explored field. We contrast ours with existing work as follows:

- Automation and Efficiency:
  - Ours: Leveraging LLMs for code generation automates the process, reducing the time and effort required for manual coding and optimization.
  - Prior Arts: Traditional methods [12], [13], [14] rely on manual optimization and coding, which can be time-consuming and prone to errors.
- Adaptability to Different Neural Network Architectures:
  - Ours: The use of LLMs supports adaptation to various NN architectures, including different fully connected networks, and possibly GNNs and CNNs in future (with more powerful LLMs for the backbone).
  - Prior Arts: Many implementations [15], [16] focus on specific types of NNs, limiting their adaptability.
- Integration with High-Level Synthesis:
  - Ours: By generating HLS-compatible code, our method facilitates the use of high-level synthesis tools, streamlining the FPGA implementation process.
  - Prior Arts: Existing approaches [17], [15], [18] do not fully leverage HLS tools, requiring more low-level hardware design expertise.
- Performance and Resource Optimization:
  - Ours: The LLM-generated code can be optimized for parallelism, memory management, and latency reduction, enhancing the performance and efficiency of FPGA implementations.
  - Prior Arts: While existing methods [19], [20] also focus on optimization, the automation provided by LLMs can lead to more consistent and efficient results.

REFERENCES

[1] K. I. Roumeliotis and N. D. Tselikas, "Chatgpt and open-ai models: A preliminary review," *Future Internet*, vol. 15, no. 6, p. 192, 2023.

[2] PrimisAI, "Welcome to rapidgpt," 2023, accessed: 2024-09-10. [Online]. Available: https://primis.ai/docs

[3] R. Silicon, "Rapid silicon announces rapidgpt's official availability," 2023, accessed: 2024-09-10. [Online]. Available: https://rapidsilicon.com/rapid-silicon-announces-rapidgpts-official-availability/

[4] Microsoft, "Make writing your next research paper easier with ai," 2023, accessed: 2024-09-10. [Online]. Available: https://www.microsoft.com/en-us/edge/learning-center/ai-help-with-research-papers?form=MA13I2

[5] Microsoft, "Enhance online research with ai," 2023, accessed: 2024-09-10. [Online]. Available: https://www.microsoft.com/en-us/bing/do-more-with-ai/enhance-online-research-with-ai?form=MA13KP

[6] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, 2017, pp. 65–74.

[7] K. Abdelouahab, M. Pelcat, J. Serot, C. Bourrasset, and F. Berry, "Tactics to directly map cnn graphs on embedded fpgas," *IEEE Embedded Systems Letters*, pp. 1–4, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8015156/

[8] H. Sharma, J. Park, E. Amaro, B. Thwaites, P. Kotha, A. Gupta, J. K. Kim, A. Mishra, and H. Esmaeilzadeh, "Dnnweaver: From high-level deep network models to fpga acceleration," in *the Workshop on Cognitive Architectures*, 2016.

[9] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 152–159.

[10] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "Autochip: Automating hdl generation using llm feedback," *arXiv preprint arXiv:2311.04887*, 2023.

[11] L. Collini, S. Garg, and R. Karri, "C2hlsc: Can llms bridge the software-to-hardware design gap?" *arXiv preprint arXiv:2406.09233*, 2024.

[12] H. Hong, D. Choi, N. Kim, H. Lee, B. Kang, H. Kang, and H. Kim, "Survey of convolutional neural network accelerators on field-programmable gate array platforms: architectures and optimization techniques," *Journal of Real-Time Image Processing*, vol. 21, no. 3, p. 64, 2024.

[13] R. Ayachi, Y. Said, and A. Ben Abdelali, "Optimizing neural networks for efficient fpga implementation: A survey," *Archives of Computational Methods in Engineering*, vol. 28, no. 7, pp. 4537–4547, 2021.

[14] E. H. C. Tourad and M. Eleuldj, "Generic automated implementation of deep neural networks on field programmable gate arrays," in *The Proceedings of the International Conference on Smart City Applications*. Springer, 2021, pp. 989–1000.

[15] C. Wang and Z. Luo, "A review of the optimal design of neural networks based on fpga," *Applied Sciences*, vol. 12, no. 21, p. 10771, 2022.

[16] F. N. Peccia, S. Pavlitska, T. Fleck, and O. Bringmann, "Efficient edge ai: Deploying convolutional neural networks on fpga with the gemmini accelerator," *arXiv preprint arXiv:2408.07404*, 2024.

[17] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "An automatic rtl compiler for high-throughput fpga implementation of diverse deep convolutional neural networks," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–8.

[18] K. Yamazaki, V.-K. Vo-Ho, D. Bulsara, and N. Le, "Spiking neural networks and their applications: A review," *Brain Sciences*, vol. 12, no. 7, p. 863, 2022.

[19] M. Espinosa, "Implementation of convolutional neural networks in fpga for image classification," 2019.

[20] Q. Yi, H. Sun, and M. Fujita, "Fpga based accelerator for neural networks computation with flexible pipelining," *arXiv preprint arXiv:2112.15443*, 2021.