LGD: LLM-Driven Code Generation Framework for the Library of Domain-Specific Architectures

***Abstract*—Various Domain Specific Architectures (DSAs) are promoted to address the computing ability issue in the machine learning (ML) and artificial intelligence (AI) workloads. Therefore, it is essential for software and libraries to collaborate with DSAs is indeed essential to leverage the potential of these specific hardware. Currently, developers have generally adopted Zero-Shot with Large Language Models (LLMs) to develop functions for library. However, this method requires human experts to manage the workflow themselves, leading to low prompt reuse and high manual effort. Additionally, the quality of the generated code is unstable due to the heavy reliance on the proficiency of LLM users. Moreover, it is difficult to migrate across different architectures. To address these, we adopt the methodologies of Chain of Thought, Iterative Refinement, and Structural Prompt to design an LLM-driven code generation framework for the library of domain-specific architectures. With LGD we can generate higher quality functions for library with fewer manual tokens compared to Zero-Shot. We can customize LGD by filling domain-specific information in custom fields with few manual tokens allowing convenient migration across architectures. In the experiment, an unpracticed C language engineer used LGD to generate multiple functions across various architectures and conducted conformance tests on them. The results show that, compared to Zero-Shot, LGD requires only about 2% to 30% of the manual tokens but generates functions with a 2.4 to 7 times higher pass rate, demonstrating LGD's efficiency and portability in function development for DSAs library. This significantly reduces the difficulty of developing library. Moreover, LGD can be easily migrated across different architectures through customization.**

***Index Terms*—Library, Large Language Models, toolchain, Code Generation, Prompt Engineering, Domain-Specific Architecture.**

I. Introduction

In recent years, to address the end of Moore's Law and insufficient computational power, the demand for Domain-Specific Architectures (DSAs) is growing rapidly [1]. Compared to general-purpose processors, DSAs offer lower power consumption and faster computation speeds in specialized fields [2]. DSAs can be implemented in two ways: heterogeneous way [2] and extensions based on general instruction set. The latter offers more flexible programming [3]. However it still faces high software development costs for development toolchain, making programming difficult. In the DSA development toolchain, libraries hold an important position [4]. Traditionally, libraries were mainly built through manual programming by experts proficient in both DSA software and hardware. With the emergence of LLMs and their powerful code generation capabilities, almost all software developers, including DSA toolchain developers, began using LLMs to assist in software development [5]. Among the development toolchains, libraries have a shallower structure compared to other large software project (such as compilers or simulators). Most of functions in libraries are functionally independent, and their relationships can be easily described in natural language. As it is challenging to apply LLMs in large and complex software project [6], making it more feasible to apply LLMs in developing libraries.

Currently, there are two main methods for using LLMs to assist library development in the industry: 1) delegating the programming of code segments within the functions to LLMs then manually combining them, and 2) generating libraries through Zero-Shot with general programming prompts.

However, both methods have drawbacks. Their common flaw is that every library developer must have a deep understanding of both the applications in the specific domain and the underlying hardware architecture to develop, which limits the scale of library development. The drawback of the former is that LLMs serve merely as code-completion-tools, offering limited efficiency improvement for skilled developers and failing to fully leveraging the code generation capabilities of LLMs.

There are three drawbacks of the latter. Firstly, human developers need to manage the workflow themselves, resulting in low prompt reuse and high manual effort. Secondly, the code quality heavily depends on the prompt engineering skills of the LLM users, making it challenging for other engineers to share and reproduces. Thirdly, This method can only be used for developing a single type of DSA, making it challenging to migrate to other DSAs.

To address challenges above and fill the gap in the application of prompt engineering for LLM in development of DSA libraries, we adopted the methodologies of *Chain of Thought* [7], *Iterative Refinement* [8], and *Structural Prompt* [9] to design an LLM-driven code generation framework for library of DSAs, LGD. First, we are inspired by the workflow of how human develop libraries, and design workflows for LLM based on the idea of Chain of Thought. Second, we introduce Iterative Refinement [8] in each step of the workflow to ensure the correctness of the generated results for each step. Lastly, we integrated the designed workflow and domain-specific information into a structural prompt framework LangGPT [10] to implement our prompt LGD. As shown in Fig. 1, LGD generates the functions of library as follows: 1) Initialization by inputting LGD to LLMs as a prompt. 2) Make a selection between WorkflowComplex and WorkflowSimple. 3) LGD asks for the function requirements. 4) Based on the selected workflow, LGD generates algorithm principles (WorkflowComplex), other function lists (WorkflowComplex), and intrinsic lists (common) in different steps. 5) We adopts the

Iterative Refinement to check the output of LGD and provide modification suggestions repeatedly, until subjectively believing that the output is generally correct. 6) LGD generates the code of function. 7) We checks, tests and fine-tunes it with the LLM. 8) The generated function passes the test, LGD updated its memory, and returns to the workflow selection stage.

The contributions of this paper are summarized as follows:

- We propose LGD, an LLM-driven code generation framework for library of DSAs. Following elaborate designed workflows, we can input intrinsics and requirements to generate functions for library with minimal human intervention.
- LGD can be customized for specific architectures, allowing for cross-architectures migration. In addition to the neuromorphic processor RVNE, we generated multiple functions for Arm Neon's Ne10 library through fast customized LGD, demonstrating LGD's versatility and portability across different architectures.
- Even an unpractised C language engineer can uses about 2%-30% of the manual tokens to generate functions with a 2.4-7 times higher pass rate using LGD, which demonstrates LGD's efficiency in DSAs library development.

II. Background and Related Works

A. Domain-Specific Architecture

Domain-Specific Architectures (DSAs) are an emerging type of architecture designed to optimize data flow in target applications through specialized hardware acceleration while offering programming flexibility [2].

There are two ways to implement DSAs. One is a heterogeneous architecture, which effectively combines general-purpose processor cores with specialized hardware accelerators to improve energy efficiency and provide a certain level of programming flexibility. [2] Another is homogeneous architecture which is an extension to instruction set, such as RVNE [3] for SNN, XpulpNN for accelerating quantized neural networks [11], and Arm's Neon [12] for vector computation. Compared to heterogeneous architectures, homogeneous DSAs rely on the ISA's software ecosystem, offering greater programming flexibility.

B. Toolchain and library

The application scenarios for DSAs are broad and varied. To enable developers to develop domain-specific applications more conveniently and flexibly (rather than developing with assembly language), a DSA toolchain is essential. In the DSA development toolchain, libraries hold an important position. To make it easier for application developers to develop software, both Arm [13] and Intel provide libraries for their respective Architecture to support tasks such as image processing, machine learning, and computer vision. Both homogeneous and heterogeneous DSAs rely on the support of compilers and toolchains. For example, NVIDIA's CUDA Toolkit [14] includes NVCC (NVIDIA CUDA Compiler) and other development tools, XLA (Accelerated Linear Algebra) for Google's TPU [15], and Alibaba's Xuantie [16] toolchain and compiler.

C. LLM and Prompt Engineering

After demonstrating powerful code generation capabilities, LLMs (Large Language Models) have been the subject of many studies focused on generating various codes. Such as evaluating [17], RTL code [18]. However, according to current research, there are no studies specifically targeting library development using LLMs. Research on general code generation does not meet the needs of library development, as this process rely on domain-specific software and hardware.

Currently, there are three mainstream methods for using LLMs to generate code: Prompt engineering involves designing and optimizing input prompts for interaction with generative AI models to improve the quality and relevance of the generated content [19]. Fine-tuning LLMs is a method where these models are adjusted to perform specific tasks using additional training data [20]. Domain-adaptive pre-training of LLMs involves training the models on domain-specific data to enhance their performance within that domain [21]. These approaches, as discussed in the literature, provide targeted ways to refine and improve the capabilities of large language models .

However, fine-tuning and domain-adaptive pre-training are costly, requiring extensive domain-specific datasets and training resources. Given these limitations, prompt engineering with commercial LLMs is a suitable approach.

There are many prompt engineering designed to enhance the generative capabilities of LLMs [22]. Among the various prompt engineering frameworks, we chose to build LGD based on LangGPT. LangGPT is a structural prompt design framework that features an easy-to-learn standardized structure and provides an expandable framework for migration and reuse [10].

III. Methodology

A. Overview

We proposed LGD (Library Generator for DSAs), an LLM-driven code generation framework for library of DSAs. Following the LGD workflow, We can interact efficiently with LLMs to develop library with significantly reduced manual effort and improved code quality. LGD is a structural prompt organized with five modules: *Profile*, *Rules*, *Workflow*, *Library*, and *Knowledge*. Therefore LGD is more like a programming language than a natural language, making it easier for LLM to understand and perform better [10].

Profile sets the LLM's role as LGD, a C language engineer developing libraries for DSA. To ensure effective programming and avoid usual errors, LGD is supposed to obey the content in *Rules*.

WorkflowSimple and WorkflowComplex for generating simple and complex functions are integrated in *Workflow*.

Library contains functions that would be called when generating complex functions. And *Knowledge* consist of necessary knowledge related.

These five modules are finally integrated and concluded in the *Initialization*, The process of using LGD in the development of the library is as follows: First, we prepared a document

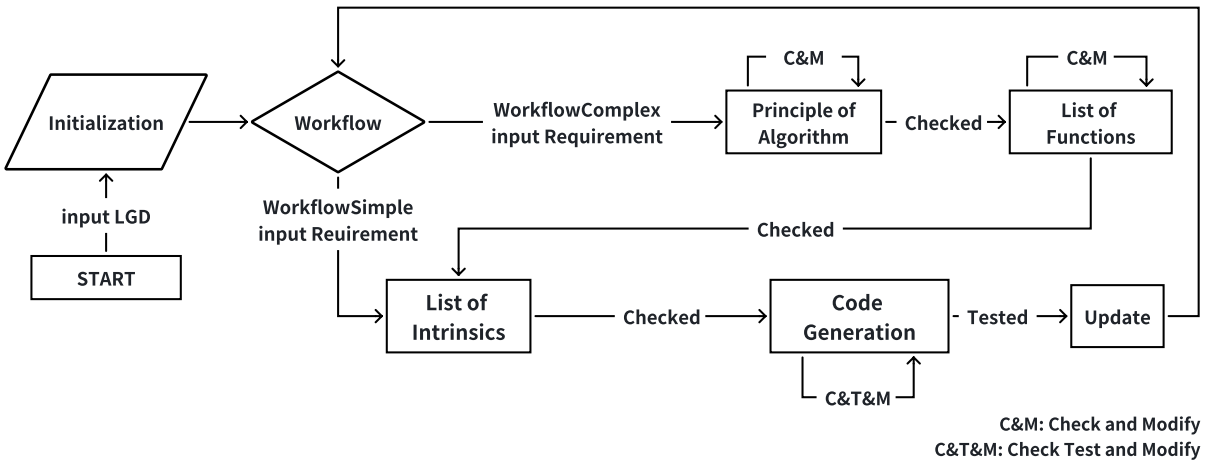

Fig. 1. Flowchart of LGD

containing intrinsic functions and confirmed the relevant software and hardware knowledge. Secondly we customized LGD based on knowledge above. Thirdly, we input customized LGD and intrinsic document to complete *initialization*. Then, we generated function step by step according to the workflow we choose. Lastly, we tested generated functions and prompted LGD to modify the code according to test result until it passed.

B. Customization

As an LLM prompt framework, LGD cannot be used without any processing. For different DSAs, we filled in domain-specific information into LGD and modify the content as needed. This process is called *customization* [10].

The content of LGD is divided into *custom fields* and *fixed fields*. We can add, modify, or remove *custom fields* based on our needs, including domain-specific knowledge of applications/algorithms/software/hardware. Utility functions and data structures that have been implemented, which are usually integrated in *Knowledge*; *Library*, containing the functions needed; *Profile*, a module used to set the name, version, background, and the task of LGD.

Fixed fields are not recommended to modify, such as *Workflow* and certain *Rules*. Modifying these fields may seriously influence LGD's performance and code quality.

C. Initialization

Initialization involves five modules introduced above, and *Workflow* will be introduced in next section.

1) Profile: We set LLM's role as LGD, informing it that it is a software engineer developing library for DSA using C language, and its work background to ensure it can immerse in role-playing. It introduces itself, promise to follow the rules defined in *Rules* and describes the *Workflow*.

2) Rules: Limits or guidelines that LLMs must not surpass, along with criteria that need to be fulfilled when producing responses [10]. When using LLM in library development, every time LLM make repetitive mistakes, we had to correct them with manual prompts which consumed extra tokens and increase manual effort. Therefore, we integrate these prompts into *Rules* significantly reducing the probability of making repeated errors due to structured representation. Moreover, thanks to the *Structural Prompt*, we can refer to rules to prompt instead of repeating them when repetitive mistakes appear, further reducing the number of tokens. With the *Rules*, LGD can generate output that meets our expectations.

3) Library: The complex functions will call the other functions in library. So the set of functions needed by complex functions must be a subset of the *Library*. When preparing to generate a complex function, we integrated its dependent function implementations and functionalities in the *Library*.

4) Knowledge: We had summarized the knowledge and code that often require manual prompts during multiple generation experiments. We integrated these contents in a structured format into *Knowledge* to optimize the performance of LGD. This is a highly customizable module.

LGD ends with *Initialization* where all modules are concluded. We uploaded customized LGD and Intrinsic document into LLM to implement *Initialization*. During this process, we assigned a new identity (i.e., LGD) to the LLM, defined its specific work tasks, imparted relevant knowledge and skills, outlined prohibited behaviors, and established the necessary workflow for compliance. Afterward LGD will guide us in selecting the workflow to start generating work.

D. Workflow

CoT can improve model performance on complex tasks by task decomposition [7]. However, the workflows obtained by LLM through task decomposition show poor stability and are not always reasonable. Therefore, we manually pre-designed fixed workflows for LLM to implement CoT based on how human experts develop libraries.

After *initialization*, *WorkflowSimple* (for simple functions) and *WorkflowComplex* (for complex functions) are introduced

Algorithm 1 WorkflowComplex

input: $intrDoc$,req
Output: Target function $Func$

Algorithm alg = Generation(req)
alg = CheckandModify(alg)
FunctionList $funcList$=Generation(req, alg)
$funcList$ = CheckandModify($funcList$)
IntrinsicList $intrList$ = Generation($intrDoc$, req, alg)
$intrList$ = CheckandModify($intrList$)
$func$ = Generation($intrList$,req,alg,$funcList$)
$func$ = CheckandModify($func$)
$func$ = TestandModify($func$)
return $func$

Algorithm 2 CheckandModify

Input: $genOutput$
Output: $genOutput$

$pass$ = False
while not $pass$ **do**
  if $genOutput$ is complete **then**
    $pass$ = True
  else
    input modification suggestion mod.
    $genOutput$ = Generation($genOutput$, mod)
  end if
end while
return $genOutput$

to us. We choose one to enter the corresponding workflow. The overview of which is included in Fig. 2.

We provided LGD with a preset format requirement of target function. In WorkflowComplex, based on requirement, we confirms the algorithm, the function list from *Library*, and the intrinsic list with LGD. The step-by-step description of WorkflowComplex is given in Algorithm 1. It takes intrinsic document and requirement as input, and the output of is the function. It begins by generating a algorithm from the requirements, Next, function list is generated based on the requirements and the algorithm, then, intrinsics are elect from the intrinsic document based on requirements, and algorithm. The target function is then generated based the output of every step. WorkflowSimple is a lightweight version of WorkflowComplex. We only needs to confirm the intrinisc list with LGD.

To enhance the output of every step, *Iterative Refinement* was applied. Algorithm 2 takes any output of Generation() as input and output. In each iteration, we provided modification suggestions to align the results of Generation() more closely with our expectation.

During the Code-Process, we checked and tested the function just like , and provide modification suggestions based on the test results until generated function passes the test, the process of which were just like Algorithm 2 To demonstrate more intuitively how LGD works, we have illustrated a simplified example of generating simple functions in the Fig. 3. The function that passes the test is the final output of LGD.

E. Migration

LGD can work effectively on RVNE, we hope it work cross-architecture. Dependent on the characteristics of *Structural Prompt*, we propose *Migration*, a method that can easily achieve migrating an customized LGA for architecture A to an customized LGB for architecture B. To achieve this, we propose a opposite concept to *Customization* called *Generalization*, the process of removing the custom fields of LGA to revert to the LGD. We abstract *custom fields* back to generic descriptions, remove domain specific knowledge, and clear the library, but keep *fixed fields* unchanged. Then customized LGD as LGB for architecture B, which has been introduced in B subsection. For example, in the experiment, we first generalized LGRVNE as LGD, then customized LGD as LGNeon. This process only involves changes to custom fields, even the longest *Knowledge* section can be completed by directly referencing other literature.

IV. Experiment

To study the characteristics of LGD and evaluate its performance, we consider the following research questions:

- **RQ1:** How much manual effort can LGD save?
- **RQ2:** How is the code quality of functions of library generated by LGD?
- **RQ3:** Does LGD still function after being migrated to another architecture?

A. LGD vs Zero-Shot: labor cost

To answer RQ1, we designed an experiment of function code generation for library to compare the labor cost between LGD and Zero-Shot.

1) Experiment Setup: **Large Language Model**. Since LGD involves multimodal input, we chose OpenAI's most advanced model, GPT-4o, for the experiment to achieve better performance and faster convergence rate.

DSA Platform. To ensure that the pre-training data does not interfere with the experiment results, we chose the closed-source, RISC-V extension-based neuromorphic processor RVNE for the experiment.

Baseline. We used general code generation method *Zero-Shot*. In this experiment, we set LLM's role as a library developer, and input the intrinsic document, hardware and software information, requirements step-by-step to generate the functions for library.

Evaluation Metrics. We quantified the manual effort by measuring the number of manual tokens used by human to interact with LLMs. Fewer manual tokens indicate lower manual effort.

Experimenter Allocation. To minimize human influence as much as possible, we arranged for an unpracticed C language user with little experience on LLMs to conduct the experiment.

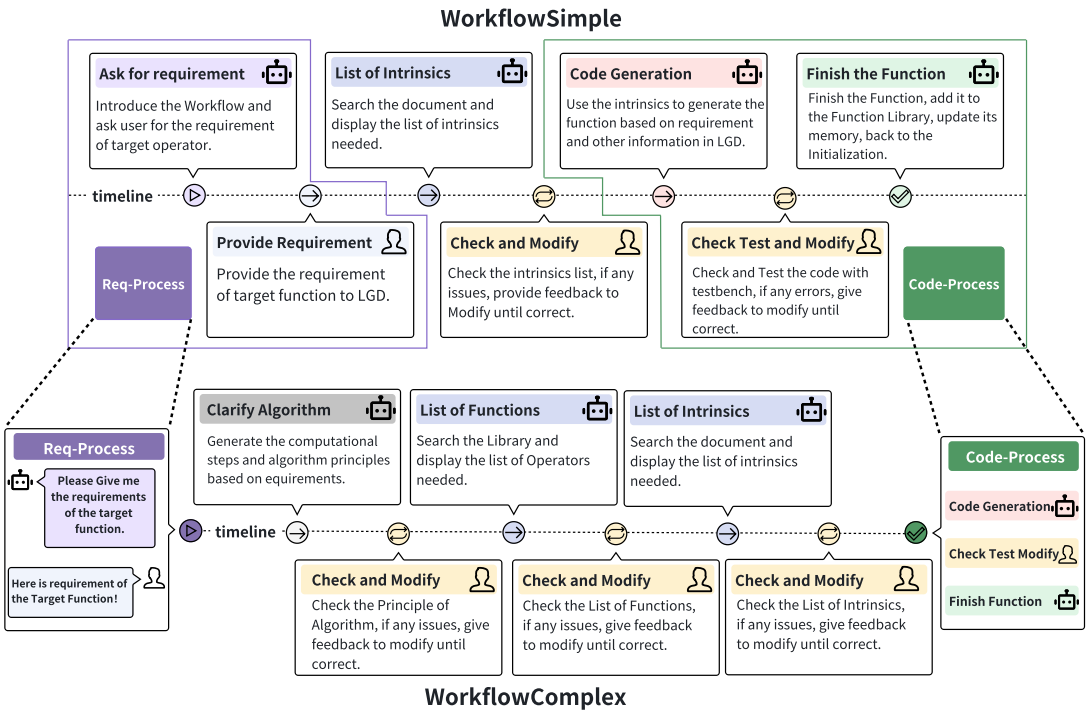

Fig. 2. **WorkflowSimple And WorkflowComplex**
The timeline proceeds from left to right. The Req-Process and Code-Process are shared by both workflows. The Req-Process includes the interactive handling of requirements. The Code-Process includes the final handling of the function code. Both of two workflow, must sequentially go through the Req-Process, each with its own key information display and confirmation, and Code-Process.

2) Experiment Result: There are three simple functions of library generated by us including *load_input_spike*, *clear_neuron_state*, *set_neuron_type* and a complex function *forward_connected*. As seen in Fig. 4, *load_input_spike* requires 7 tokens with LGD while 43 tokens with baseline. LGD's manual tokens amount to only 16% of those required by baseline. Similarly, *clear_neuron_state* and *set_neuron_type* require only 6 tokens with LGD, while they need 43 and 221 tokens, respectively, with baseline. LGD saved approximately 86% and 97% of the manual token cost, respectively. As for the complex function *forward_connected*, LGD still demonstrates its efficiency by requiring 221 tokens, compared to 465 tokens for the baseline, which saved about 70% manual tokens.

3) Answer to RQ1: LGD required only about 2% - 30% of the manual tokens compared to baseline without LGD to generate these functions, which significantly reduces labor cost.

B. LGD vs Zero-Shot: Code Quality

1) Experiment Setup: **Large Language Model**, **DSA Platform**, **Baseline** and **Experimenter Allocation** are all identical to 4.1.

Evaluation Metrics. Due to the limitations of the experiment's scale, we quantified code quality by code iteration count required to pass the conformance test. The lower the code iteration count, the higher the quality of the generated code.

2) Experiment Result: As seen in Fig. 5, for the simple functions *load_input_spike*, *clear_neuron_state* LGD requires the fewest iterations (only 1) to generate functional code, while 3 iterations for baseline. *set_neuron_type* is a relatively complex function, but LGD generated passing code on the first attempt, whereas baseline required 7 times to try.

For complex functions, corresponding to the seventh and eighth bars in Fig. 5, we first iterated in the check and modify phase, and then iterated in the test and modify phase. The numbers of iterations LGD in both phases are significantly lower than the baseline.

3) Answer to RQ2: LGD's pass rate is 2.4 - 7 times higher than baseline, passed the conformance test more swiftly, which indicates that the code quality generated by LGD is higher than that of Zero shot. Even more impressive is that LGD were able to generate simple functions that passed all test cases after just one code iteration using fewer than 10 manual tokens.

C. LGD Migration Experiment from RVNE to Arm Neon

To answer RQ3, we designed the LGD Migration Experiment from RVNE to Arm Neon [12]

1) Experiment Setup: **Large Language Model**, **Evaluation Metrics** and **Experimenter Allocation** are identical to 4.1.

DSA Platform. Arm open-sourced Neon's intrinsic documentation [23], Ne10 library with test bench [24]. Although Arm Neon, as an SIMD architecture extension, is not considered a DSA in the conventional sense, it has specific applications in specific fields like DIP and DSP. More importantly,

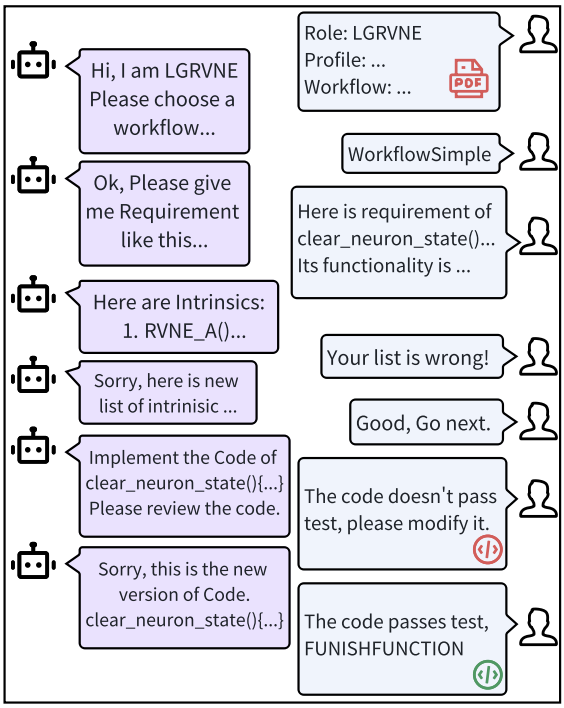

Fig. 3. The Example for Generating a Simple Function
We initialize LLM as LGRVNE and provide necessary information and perform *Iterative Refinement* according to LGRVNE's prompts. After repeated testing and debugging, the generation work is completed with an instruction "FUNISHFUNCTION"

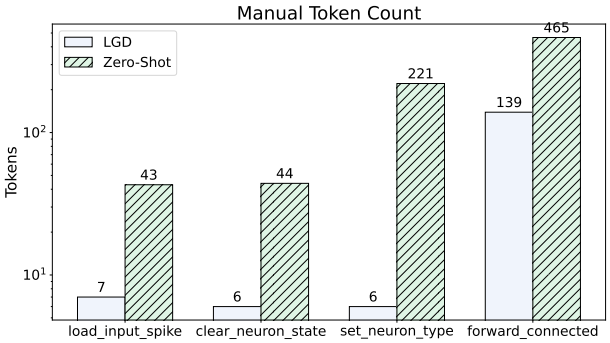

Fig. 4. Manual Token Count

the implementation of Arm Neon is highly representative for homogeneous DSAs like RVNE.

2) Experiment Result: The process of customizing LGD into LGNeon only requires manual input of 69 tokens (not including *Knowledge* and *Library*). We used LGD to generate three simple functions and one complex function for Arm Neon. As shown in the Table I, For the simple functions, only 1 iteration was required, with manual token counts ranging from 5 to 44. The complex function required 241 manual tokens and 3 iterations. The difference in the number of manual tokens and iteration count between LGNeon and LGRVNE when generating same type of functions is minimal.

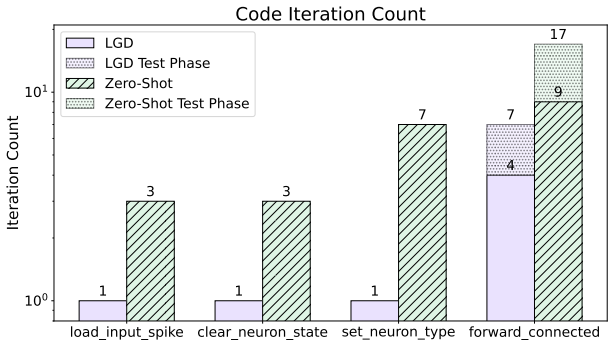

Fig. 5. Code Iteration Count of Simple and Complex Functions

3) Answer to RQ3: We can draw a conclusion that even migrated to another architecture, LGD is still capable of effectively generating functions for library while maintaining good code quality.

function	token	iteration	type	Arch
addc_float	44	1	simp	Neon
addc_vec4f	5	1	simp	Neon
mulc_float	11	1	simp	Neon
fft_c2c_1d_int32	241	3	comp	Neon

TABLE I
ARM NEON FUNCTIONS GENERATION

V. CONCLUSION

In this paper, to fill the gap in the application of prompt engineering for LLM in development of DSA libraries, we proposed an LLM-driven code generation framework for library of DSAs called LGD. We can customize LGD for different DSAs and follow its workflow to efficiently interact with LLMs to generate functions for library significantly reducing manual effort and improving code quality compared to general LLM prompt method.

In the future, we plan to explore a fully automated, bottom-up domain-specific software and hardware implementation pathway driven by LLMs with minimal human intervention.

REFERENCES

[1] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 1, pp. 21–29, Mar 2018.

[2] A. Krishnakumar, U. Ogras, R. Marculescu, M. Kishinevsky, and T. Mudge, "Domain-specific architectures: Research problems and promising approaches," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 2, jan 2023. [Online]. Available: https://doi.org/10.1145/3563946

[3] Z. Yang *et al.*, "Back to homogeneous computing: A tightly-coupled neuromorphic processor with neuromorphic isa," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 11, pp. 2910–2927, 2023.

[4] B. Carpenter, M. D. Hoffman, M. Brubaker, D. Lee, P. Li, and M. Betancourt, "The stan math library: Reverse-mode automatic differentiation in c++," 2015. [Online]. Available: https://arxiv.org/abs/1509.07164

[5] L. Belzner, T. Gabor, and M. Wirsing, "Large language model assisted software engineering: Prospects, challenges, and a case study," in *Bridging the Gap Between AI and Reality*, B. Steffen, Ed. Cham: Springer Nature Switzerland, 2024, pp. 355–374.

[6] I. Ozkaya, A. Carleton, J. Robert, and D. Schmidt, "Application of large language models (llms) in software engineering: Overblown hype or disruptive change?" Carnegie Mellon University, Software Engineering Institute's Insights (blog), Oct 2023, accessed: 2024-Aug-8. [Online]. Available: https://doi.org/10.58012/6n1p-pw64

[7] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[8] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang *et al.*, "Self-refine: Iterative refinement with self-feedback," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[9] W. Zhong *et al.*, "Proqa: Structural prompt-based pre-training for unified question answering," *arXiv preprint arXiv:2205.04040*, 2022.

[10] M. Wang *et al.*, "Langgpt: Rethinking structured reusable prompt design framework for llms from the programming language," 2024. [Online]. Available: https://arxiv.org/abs/2402.16929

[11] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, "Xpulpnn: Accelerating quantized neural networks on risc-v processors through isa extensions," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 186–191.

[12] ARM Developer, "Neon technology," 2024, accessed: 2024-08-09. [Online]. Available: https://developer.arm.com/Architectures/Neon

[13] A. Developer, "Arm compute library," https://developer.arm.com/technologies/compute-library, 2024, accessed: 2024-08-04.

[14] M. Fatica, "Cuda toolkit and libraries," in *2008 IEEE hot chips 20 symposium (HCS)*. IEEE, 2008, pp. 1–22.

[15] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, jun 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[16] C. Chen *et al.*, "Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension : Industrial product," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 52–64.

[17] F. Liu *et al.*, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[18] H. Huang, Z. Lin, Z. Wang, X. Chen, K. Ding, and J. Zhao, "Towards llm-powered verilog rtl assistant: Self-verification and self-correction," *arXiv preprint arXiv:2406.00115*, 2024.

[19] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," *arXiv preprint arXiv:2402.07927*, 2024.

[20] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[21] K. Chang *et al.*, "Chipgpt: How far are we from natural language hardware design," *arXiv preprint arXiv:2305.14019*, 2023.

[22] ai boost, "Awesome prompts," https://github.com/ai-boost/awesome-prompts, 2024, accessed: 2024-08-04.

[23] A. Developer, "Arm intrinsics," https://developer.arm.com/architectures/instruction-sets/intrinsics, 2024, accessed: 2024-08-04.

[24] Arm, "Ne10: An open optimized software library project," https://github.com/projectNe10/Ne10, 2024, accessed: 2024-08-04.