

QiMeng: Fully Automated Hardware and Software Design for Processor Chip

Rui Zhang¹, Yuanbo Wen¹, Shuyao Cheng¹, Di Huang¹, Shaohui Peng², Jiaming Guo¹,
Pengwei Jin¹, Jiacheng Zhao¹, Tianrui Ma¹, Yaoyu Zhu¹, Yifan Hao¹, Yongwei Zhao¹, Shengwen Liang¹,
Ying Wang¹, Xing Hu¹, Zidong Du¹, Huimin Cui¹, Ling Li^{2,3}, Qi Guo¹, Yunji Chen^{1,3,*}

¹State Key Lab of Processors, Institute of Computing Technology, CAS

²Intelligent Software Research Center, Institute of Software, CAS

³University of Chinese Academy of Sciences

<https://qimeng-ict.github.io/>

Abstract—Processor chip design technology serves as a key frontier driving breakthroughs in computer science and related fields. With the rapid advancement of information technology, conventional design paradigms face three major challenges: the physical constraints of fabrication technologies, the escalating demands for design resources, and the increasing diversity of ecosystems. Automated processor chip design has emerged as a transformative solution to address these challenges. While recent breakthroughs in Artificial Intelligence (AI), particularly Large Language Models (LLMs) techniques, have opened new possibilities for fully automated processor chip design, substantial challenges remain in establishing domain-specific LLMs for processor chip design.

In this paper, we propose QiMeng, a novel system for fully automated hardware and software design of processor chips. QiMeng comprises three hierarchical layers. In the bottom-layer, we construct a domain-specific Large Processor Chip Model (LPCM) that introduces novel designs in architecture, training, and inference, to address key challenges such as knowledge representation gap, data scarcity, correctness assurance, and enormous solution space. In the middle-layer, leveraging the LPCM’s knowledge representation and inference capabilities, we develop the Hardware Design Agent and the Software Design Agent to automate the design of hardware and software for processor chips. Currently, several components of QiMeng have been completed and successfully applied in various top-layer applications, demonstrating significant advantages and providing a feasible solution for efficient, fully automated hardware/software design of processor chips. Future research will focus on integrating all components and performing iterative top-down and bottom-up design processes to establish a comprehensive QiMeng system.

I. INTRODUCTION

As the fundamental hardware platform for computing systems, processors and chips undertake critical functions including instruction execution, data processing, and resource management. These processors and chips power diverse devices ranging from personal computers, servers, smartphones, and Internet of Things (IoT) equipment, forming the technological foundation of modern digital economies. Processor chip design represents both a strategically important industry for national

economic development and a cutting-edge research field that drives progress in computer science. As a highly complex and systematic task, processor chip design requires tight hardware-software co-design to achieve functional requirements, along with optimizing performance, power, and area (PPA). These requirements make processor chip design one of the most challenging research topics across both industrial and academic domains.

The evolution of information technology has revealed three fundamental limitations in current processor chip design methodologies: constrained fabrication technological, limited resource, and diverse ecosystem. In the fabrication technological aspect, as semiconductor fabrication nears physical limits below 7nm nodes, phenomena such as quantum tunneling and short-channel effects become increasingly problematic, rendering conventional fabrication technology-based performance scaling ineffective, thereby necessitating design methodology innovations. From a resource perspective, conventional design flows demand extensive expertise and labor-intensive design-verification iteration to ensure functional correctness while balancing competing design objectives such as PPA. This results in protracted development timelines and substantial costs. In the ecosystem aspect, emerging applications in Artificial Intelligence (AI), cloud, and edge computing require specialized architectures with customized foundational software support. Thus, conventional chip design approaches cannot meet the ecosystem challenge efficiently due to their inherent lengthy time and substantial cost requirements. To sum up, these challenges underscore the urgent need for novel design paradigms that can deliver enhanced performance, improved efficiency, and reduced costs while meeting diverse application requirements.

Automated processor chip design, which aims to automate the entire design and verification pipeline of processor chips, presents a promising solution to overcome the above-mentioned limitations. By leveraging AI methodologies, automated processor chip design exhibits the potential to surpass manual design and achieve better performance under identical fabrication technology. Additionally, the automated processor

* Corresponding Author: cyj@ict.ac.cn

design approach is capable of dramatically reducing manual intervention, significantly improving design efficiency while shortening development cycles and lowering costs. Furthermore, it enables rapid customization of chip architectures and software stacks tailored to specific application domains, addressing the growing demand for specialized computing solutions.

Recent breakthroughs in Large Language Models (LLMs) and Multi-Agent systems have created new opportunities for automated processor chip design. State-of-the-art LLMs such as DeepSeek-V3 [1], DeepSeek-R1 [2], Qwen3 [3], GPT-4o [4], and Gemini 2.5 Pro [5] have demonstrated remarkable capabilities in question answering, planning, and reasoning, exhibiting the potential of artificial general intelligence (AGI). After post-training on domain-specific data, domain-specialized LLMs can be obtained and have shown impressive results across scientific disciplines such as computational biology [6], materials science, and chemistry [7]. More advanced LLM-based agents integrate cognitive abilities with the tool-use skill of LLMs to autonomously plan and execute complex workflows [8]. These developments of LLMs and agents suggest new pathways toward fully automated processor chip design.

Nevertheless, due to the distinctive nature of processor chip design, applying LLMs and agents to automated processor chip design faces four principal challenges: knowledge representation gap, data scarcity, correctness guarantee, and enormous solution space. First, the knowledge representation gap: critical processor chip design data employs graph structures, such as abstract syntax trees (ASTs), data flow diagrams (DFGs), and control flow diagrams (CFGs). Graph data exhibits an inherent semantic gap with the sequential text that LLMs typically process, constraining the capacity for domain knowledge representation and limiting the processor chip design capabilities of LLMs. Second, the data scarcity: unlike the vast petabyte-scale text corpora available on the Internet for training general-purpose LLMs, processor chip design data are orders of magnitude smaller, with merely terabyte-scale in open-source communities like GitHub, severely constraining the development of domain-specialized LLMs for processor chip design. Third, the correctness guarantee: processor design demands rigorous verification standards, which fundamentally conflict with the probabilistic nature of LLMs. For example, Intel’s Pentium 4 processor required 99.9999999999% accuracy in functional verification [9]. Finally, the enormous solution space: processor design spans multiple abstraction stages from foundational software to physical layouts, thus, modeling the design space directly at the raw bitstream level suffers from a dimensionality explosion. For example, the solution space for a 32-bit CPU reaches $10^{10^{540}}$. This enormous solution space poses extreme challenges for deriving both functionally-correct and performance-optimized processor designs.

To address the aforementioned challenges and pioneer a

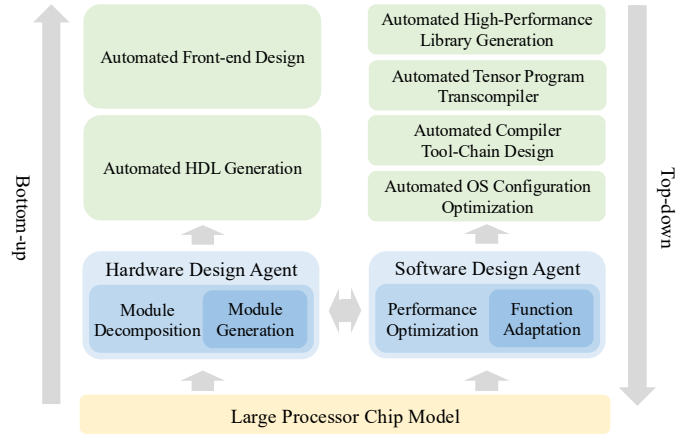


Fig. 1. Overview. QiMeng consists of three layers, a domain-specialized Large Processor Chip Model (LPCM) in the bottom-layer, Hardware Design Agent and Software Design Agent enabling automated hardware and software design based on LPCM in the middle-layer, and various processor chip design applications in the top-layer.

transformative paradigm, we propose QiMeng¹, a novel system for fully automated hardware and software design for processor chips. Consisting of three layers, QiMeng constructs a Large Processor Chip Model (LPCM) as a domain-specialized LLM for processor chip design in the bottom-layer and further creates both Hardware Design Agent and Software Design Agent based on LPCM in the middle-layer, enabling automated hardware and software design, respectively. Finally, the two agents support various processor chip design applications in the top-layer, as shown in Figure 1.

In QiMeng, to overcome the above-mentioned four challenges, LPCM is meticulously designed to incorporate domain-specialized knowledge and fundamental competencies of processor chip design. LPCM distinguishes itself from general-purpose LLMs through unique innovations in its architecture, training, and inference. Regarding architecture, LPCM employs a multi-modal structure, enabling the comprehension and representation ability of graph data inherent to the processor chip domain, which addresses the critical challenge of the knowledge representation gap. For training, it is critical to automatically generate extensive domain-specific data of processor chip design. For each abstraction stage of processor chip design, domain-specific data is systematically collected, and single-stage automated design models are independently trained. These models are subsequently cascaded to autonomously generate extensive cross-stage aligned data for processor chip design. Leveraging this aligned data, LPCM can be trained to learn domain knowledge from the hierarchical design process, effectively mitigating the data scarcity challenge. During inference, two feedback-driven mechanisms are implemented. By constructing correctness feedback from automated

¹QiMeng is a Chinese term that refers to the process of imparting fundamental knowledge and skills to beginners, serving as the cornerstone for intellectual development and skill enhancement. Named by QiMeng, we expect this system can achieve fully automated processor chip design through learning human knowledge and experience, followed by practicing and self-evolving.

functional verification, LPCM is able to autonomously repair erroneous results and ensure the validity of generated outputs, addressing the challenge of ensuring correctness in processor design. Concurrently, leveraging performance feedback from automated performance evaluation, LPCM is capable of decomposing the solution space and pruning the low-performance subspaces. Thus, LPCM can effectively reduce the dimensionality of the solution space and enable efficient exploration of high-performance design solutions, overcoming the challenge of the enormous solution space.

Based on LPCM, QiMeng develops two specialized agents, a Hardware Design Agent and a Software Design Agent, dedicated to the automated design of hardware and software for processors and chips. The Hardware Design Agent adopts a dual-loop mechanism, consisting of an outer module decomposition feedback loop based on performance optimization and an inner module generation feedback loop empowered by automated verification and repair. This dual-loop mechanism facilitates end-to-end automated design from functional specifications to physical layouts, unifying conventional disjointed stages such as logic design, circuit design, and physical design. Thus, Hardware Design Agent enables a fully integrated, cross-stage collaborative design paradigm that is expected to surpass conventional human design, potentially achieving superior performance under identical fabrication technology. Meanwhile, the Software Design Agent also employs a dual-loop mechanism, consisting of an outer performance optimization feedback loop guided by LLM and an inner function adaptation feedback loop based on automated verification and repair. Software Design Agent autonomously achieves seamless functional adaptation and performance optimization of foundational software for target processor chips, addressing the dynamic and escalating demands of modern applications.

Leveraging the Hardware Design Agent and Software Design Agent, various applications can be developed to address diverse real-world use cases of processor chip design. For automated hardware design, significant milestones have been accomplished, including automated front-end design and automated HDL generation. In automated software design, achievements include automated OS configuration optimization, automated compiler tool-chain design, automated tensor program transcompiler, and automated high-performance library generation. These applications have driven the implementation of key components within QiMeng, establishing a solid foundation for its full realization. Moving forward, we will construct QiMeng through a three-phase approach, transitioning from top-down to bottom-up, ultimately achieving a self-evolving framework. Initially, in the top-down phase, the implementation of diverse automated design applications in top-layer will provide two agents in middle-layer with design expertise and generate extensive domain-specific data to enhance the capabilities of the underlying LPCM. Subsequently, in the bottom-up phase, the improved LPCM, the hardware and software design agents will be applied across a broader spectrum of processor chip design applications in a bottom-up fashion. Ultimately, in the iteration phase, an iterative cycle integrating top-down and bottom-up approaches will be established to enable the self-evolution of QiMeng,

progressively advancing its fully automated processor chip design capabilities while extending its applicability to support increasingly diverse and complex scenarios.

Aiming to present a comprehensive framework for fully automated hardware and software design for processor chips, this work introduces QiMeng, along with its roadmap, design methodology, and applications. This paper is structured as follows: Section II provides the motivation of QiMeng and its roadmap; Section III elaborates the design of LPCM, encompassing architecture, training and inference; Section IV details the Hardware Design Agent and Software Design Agent; Section V showcases diverse applications enabled by key components of QiMeng; Section VI surveys related research in automated processor chip design; Section VII concludes with insights into future research trajectories.

II. ROADMAP

Automatic processor chip design is one of the central problems in the field of computer science, originating from the Church's Problem [10]: *How can circuits be automatically designed to satisfy the relationship between given inputs and outputs?* Proposed in 1957 by Alonzo Church, the founding figure of computer science, this problem has been a major challenge for decades, attracting extensive research from Turing Award winners such as Rabin, Scott, and Pnueli, yet it remains unsolved. Early Electronic Design Automation (EDA) tools, which were based on predefined rules and Boolean logic, automated specific design tasks such as logic synthesis, placement, and routing. As circuit complexity increased, optimization-based techniques emerged, including High-Level Synthesis (HLS), which automated the translation of high-level descriptions to RTL, and Design Space Exploration (DSE), which optimized design parameters for PPA. In recent years, AI technologies have propelled automatic processor chip design into a more intelligent, data-driven phase. Techniques like Random Forests, Reinforcement Learning (RL), and Graph Neural Networks (GNNs) have enabled automatic circuit optimization, placement, and routing, significantly enhancing design efficiency in complex scenarios. However, these approaches mainly apply AI as a tool to refine steps in the conventional EDA process, without fundamentally altering the overall design paradigm.

The current automated design methods have three main limitations. First, processor chip design requirements in real-world applications are often expressed in vague, informal natural language, while existing methods can only handle precise, formal inputs, typically in the form of Hardware Description Languages (HDLs). As a result, the transition from informal to formal requires significant work from experts. Second, these methods can only automate certain steps of processor chip design, such as logic synthesis, formal verification, automatic placement, and routing. However, critical tasks like logic design, instruction set extensions, software tool-chain adaptation, and optimization still cannot be fully automated. Finally, existing methods are typically limited to individual tasks, with a constrained design space and a lack of cross-stage hardware-software co-design, making it difficult to push the boundaries of human-driven design.

The development of LLMs and agents has opened up new possibilities for overcoming three key limitations of the conventional automatic design methodologies. First, LLMs can convert informal natural language descriptions into formal programming languages, allowing them to automatically generate correct code for tasks ranging from basic functions to entire programs based on natural language specifications. Second, agents built on LLMs can autonomously plan and execute complex tasks and can independently utilize external tools. This capability offers a novel approach for integrating AI techniques with domain-specific tools, which offers new perspectives to achieve fully automated processor chip design. Finally, LLMs possess powerful multi-task abilities and demonstrate strong potential in completing complex planning and reasoning tasks, which form the basis for achieving cross-stage collaboration in hardware-software design.

Based on the above analysis, we introduce QiMeng, an innovative paradigm for fully automated hardware and software design for processor chips. QiMeng consists of three hierarchical layers, as illustrated in Figure 1. The bottom-layer is LPCM, which embeds domain-specialized knowledge in the field of processor chip design. The middle-layer is the Hardware Design Agent and Software Design Agent, which enable the automated design of hardware and software by leveraging the domain knowledge from LPCM. The top-layer focuses on implementing various applications that use the automated design capabilities provided by the Hardware Design Agent and Software Design Agent to address different design requirements for processor chips. These three layers work synergistically, forming a complete system for fully automated hardware and software design of processor chips.

However, the realization of QiMeng is not achieved instantaneously. Each of the three levels faces its own unique set of challenges, making it difficult to directly establish the complete QiMeng system in a bottom-up way. Specifically, the implementation of LPCM in bottom-layer requires substantial domain-specialized data in hardware/software design of processor chips. However, the domain-specialized data is extremely scarce, preventing the training of LPCM. In the middle-layer, the development of Hardware/Software Design Agent depends on the domain knowledge provided by LPCM, while also needing to integrate specialized tools for verifying the correctness and evaluating performance. At the top-layer, implementation of the various applications relies on both LPCM and the two agents. Despite these challenges, the three layers exhibit strong interdependence and can provide mutual enhancement. The various applications at the top-layer can provide valuable domain-specialized data for LPCM, and also facilitate the use of specialized tools for functionality verification and performance assessment towards Hardware/Software Design Agent. Furthermore, achieving constructing a complete interaction process between LPCM and specialized tools, the Hardware/Software Design Agent can offer an automatic data generation mechanism for LPCM. Through the collaborative synergy of the three levels, the challenges each level faces can be effectively resolved.

Although QiMeng is originally designed in a bottom-up manner, it is easier to start with a top-down manner during

actual implementation. Driven by the aim of achieving various hardware and software designs, the implementation of applications in top-layer can offer extensive synthetic domain-specialized data for LPCM, and also provide design experience of collaborating with specialized tools for designing Hardware/Software Design Agent.

Based on the above analysis, we propose a three-phase roadmap to implement a complete QiMeng system. The first phase is to adopt a top-down construction approach, developing various applications based on the LPCM, which is initialized with a general-purpose LLM. During the implementation of applications, key components of and functions of the Hardware/Software Design Agent are constructed, which are then combined to establish complete processes of the two agents. At the same time, extensive domain-specialized data of software and hardware design is synthesized for training LPCM, enabling LPCM to acquire domain knowledge superior to general-purpose LLMs. The second phase is to adopt a bottom-up construction approach, reconstructing the Hardware/Software Design Agent based on the trained LPCM and re-developing the various applications. Due to LPCM being enhanced with domain knowledge and specialized capabilities, the applications redeveloped in the second phase will achieve better automated design results than those in the first phase. On this basis, higher-quality software and hardware design data can be obtained based on the applications in the second phase. The third phase is to form an iterative loop by combining the top-down and bottom-up design processes. Inspired by John von Neumann’s “Theory of Self-Replicating Automata” [11], we hope to achieve the self-evolution of QiMeng through this loop. In the process of evolution, on the one hand, we aim to expand the depth of QiMeng, continuously improving its capabilities for fully automated software and hardware design for processor chips. On the other hand, we aim to expand the breadth of QiMeng, continuously extending the spectrum of applications and providing intelligent support for a broader range of processor chip design scenarios.

The current work is still in the first phase of the three-phase approach. So far, representative applications including automated front-end design, automated HDL generation, automated OS configuration optimization, automated compiler tool-chain design, automated tensor program transcompiler, and automated high-performance library generation have been successfully implemented. These applications fulfill some key components of the Hardware/Software Design Agent. In future work, we will complete the first phase of integrating these key components into a complete Hardware/Software Design Agent and automatically generate extensive domain-specialized data to train LPCM. Following this, the second and third stages will be carried out to build a comprehensive QiMeng system.

III. LARGE PROCESSOR CHIP MODEL

Due to the unique nature of the processor chip design, four key challenges must be addressed: knowledge representation gap, data scarcity, correctness guarantee, and enormous solution space. To tackle these challenges, LPCM developed in QiMeng employs distinctive approaches in architecture,

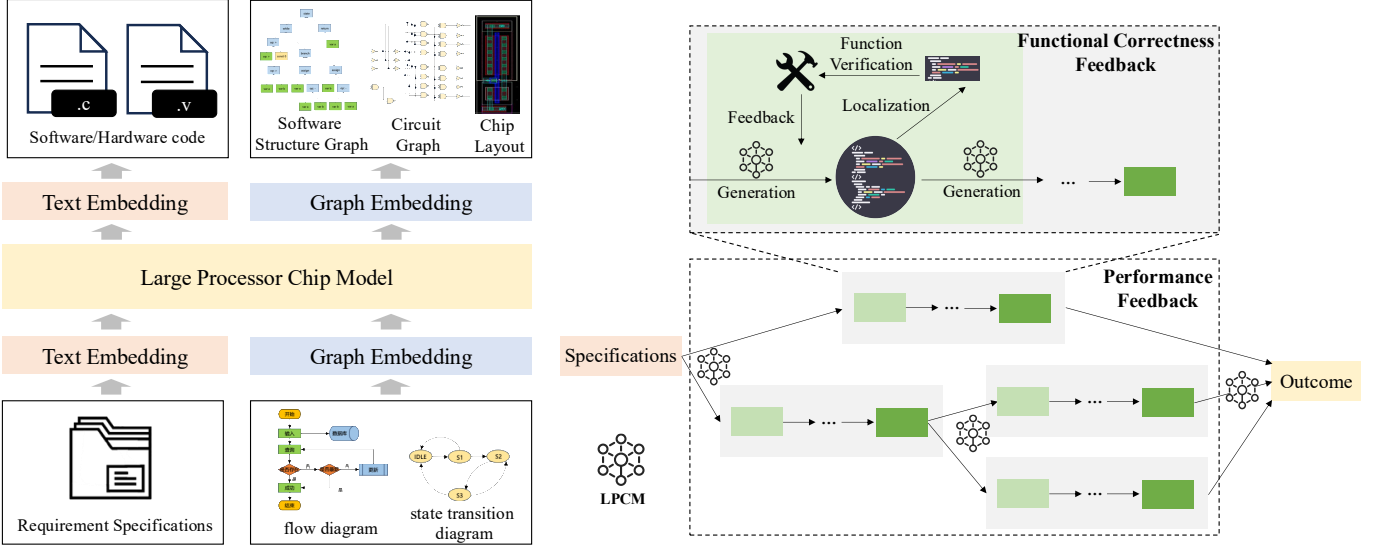


Fig. 2. Left: Multimodal architecture of LPCM capable of understanding, representing, and generating both text and graph data. Right: Feedback-driven inference of LPCM with a dual-loop mechanism, consisting of an outer performance feedback loop and an inner functional correctness feedback loop.

training, and inference, setting it apart from general-purpose LLMs. This section provides a detailed discussion of these innovations.

Notably, designing a comprehensive LPCM immediately poses significant challenges, as it demands expertise and capabilities of foundational software development to chip design. Our prior work [12] investigates the construction of LPCM, categorizing the process into three levels: 1) Human-Centric, which assists and provides suggestions for humans in code generation and parameter tuning; 2) Agent-Orchestrated, independently completing certain subtasks with toolchain integration to facilitate cross-layer optimization; 3) Model-Governed, achieving full automation of the full process of hardware-software co-design, simulation, and iterative refinement. This paper specifically focuses on the design methodology of level 3.

A. Multimodal Structure

Since text is typically organized as sequential data, most existing LLM architectures are primarily focused on handling sequential information. Even in multimodal LLMs that process images or data of other modalities, multimodal features are treated as a specialized sequence type and are concatenated with text sequences before feeding into the model. However, in the processor chip design domain, beyond textual descriptions of functional requirements and formal code representations, much of the critical information and knowledge is represented as graph illustrations. For example, software architecture is often represented as ASTs, chip logic architecture as DFGs, and chip circuit architecture as CFGs. These graph data are essential for processor chip design. As a result, LPCM is specifically designed as a multimodal architecture, capable of understanding, representing, and generating graph data, enabling more effective capacities of learning and presenting processor chip domain knowledge, as shown in Figure 2 left.

Specifically, the input to LPCM consists of two modalities: textual descriptions and graphical illustrations of requirement specifications. There are two critical issues in understanding and representing graph data: feature representation and feature alignment. A straightforward approach is to represent the graph data in a special textual format and concatenate it with the textual tokens before feeding it into the model. However, this approach serializes the graph’s topological structure, potentially causing nodes that are topologically close in the graph to be positioned far apart in the sequence, thus losing topological information of the graph data. To better preserve the graph’s topological information, Graph Neural Networks (GNNs) [13] can be used to encode the graph data and generate its embedding. Contrastive learning can then be applied to align the features of the graph embedding with the corresponding text embedding. Once feature alignment is achieved, the graph embedding is concatenated with the other textual tokens and fed into LPCM.

The output of LPCM also encompasses two modalities: text and graph. The text modality includes both the software and hardware code for processor chip design, while the graph modality includes generated diagrams, such as software architecture diagrams, chip logic diagrams, and chip layouts. The process of generating a graph is closely tied to its representation. If the graph data is directly represented by a special textual format, LPCM can also directly output the graph in this format. However, this representation method may risk losing topological information, which could compromise the accuracy of the generated graphs. To better preserve the topological information of the graph, LPCM can first output the graph’s embedding, which is then mapped to a graph structure using specialized graph generation models, such as diffusion models like GRAPHARM [14] or generative GNNs like GPT-GNN [15]. To more accurately reflect the characteristics of the circuit, the Binary Decision Diagram (BDD), which is commonly used in the context of circuit

design, can be employed. Additionally, Binary Speculation Diagram (BSD) [16], which is an enhanced version of BDD for circuit generation, can also be used to offer better suitability for automated processor chip design.

B. Cross-stage Collaborative Training

To enable the automated full-process design of processor chips, it is essential to gather cross-stage design data for training LPCM. However, the processor chip field faces significant data scarcity. In comparison to the petabyte-scale text corpora available on the Internet, software and hardware code data from open-source communities like GitHub are limited to terabyte-scale. Moreover, this data typically covers only specific stages of the processor chip design process. Thus, cross-stage design data, which includes multiple stages of design abstraction, are seriously scarce. This data scarcity challenge presents a major obstacle to the effective training of LPCM. Therefore, it is crucial to develop a cross-stage collaborative design database to provide the necessary foundation for training LPCM.

To build a cross-stage collaborative design database, an automated process for generating cross-stage design data should be established. For multiple abstract stages of processor chip design, including high-performance library design, OS kernel design, compiler tool-chain design, logic design, circuit design, and physical design, design data is first collected separately for each stage. These data only need to capture information for individual stages and do not require cross-stage alignment. Higher-level information tends to resemble natural language and contains richer semantic content, while lower-level information is closer to the actual graphical representation of the chip. Therefore, data at each stage must include both textual and graphical modalities. These single-stage design data are then used to train models, which generate automated design models for each stage. By cascading these models together, large-scale, cross-stage aligned hardware and software design data for processor chips can be automatically generated to address the challenge of data scarcity.

Once the cross-stage collaborative design database is constructed, we can train LPCM to develop the capability to generate cross-stage collaborative design reasoning. This can be achieved by applying Chain-of-Thought (CoT) imitation learning based on the database. In this process, design data from multiple stages of the processor chip design in the cross-stage collaborative design dataset are treated as reasoning sequences, generating numerous (input, CoT, output) triplets, which are used to train the CoT reasoning of LPCM. To enhance the LPCM's understanding of intermediate stage design details, a distribution alignment loss is introduced into the training objective. To ensure stable training, a curriculum learning strategy can be employed, starting with samples that feature shorter CoT and simpler design complexities, progressively increasing the complexity of samples. Training on this comprehensive cross-stage collaborative design database will enable the processor model to acquire the capabilities of generating a collaborative design process and the final design outcome. Furthermore, an automated unit testing framework

will be employed to create a reward function, and RL methods will be applied to further refine the LPCM's CoT generation and improve its processor chip design capabilities.

C. Feedback-Driven Inference

Although LPCM is equipped with domain-specialized knowledge to handle graph information and capabilities of cross-stage collaborative design, challenges such as context length limitations and hallucinations hinder the ability of LPCM to achieve seamless end-to-end processor chip design and foundational software adaptation and optimization. To ensure both the correctness and high-performance of the generated hardware/software of processor chips, it is essential to develop a feedback-driven inference mechanism for LPCM to facilitate effective design planning, leverage external functional verification tools, and optimize performance through feedback. Specifically, feedback-driven inference can be divided into two categories: functional correctness feedback and performance feedback, as shown in Figure 2 right. LPCM perform these two feedback with a dual-loop mechanism, consisting of an outer performance feedback loop and an inner functional correctness feedback loop.

1) *Functional Correctness Feedback*: Functional verification is a critical step in ensuring the correctness of manually designed processor chips. By simulating various use cases and corner cases, this process verifies the functionality of key components such as the processor's instruction set, data paths, control logic, and multi-core coordination. Functional verification ensures that the processor chip meets specifications before tape-out, preventing costly re-manufacturing due to logical errors or functional defects. Functional verification is essential across all stages of processor chip design, including logic design, circuit design, and physical design. During verification, techniques such as formal verification, dynamic simulation, and hardware simulation are employed. If functional errors are identified, experts manually refine the design and perform iterative verification until the design achieves full functional correctness. Inspired by this, to advance functional correctness during the automated processor chip design process, a feedback mechanism oriented to functional Correctness must be integrated into the inference of LPCM. This functional correctness feedback mechanism uses verification feedback to automatically verify the design and further repair the design errors, ensuring the correctness of the design outcomes and addressing the challenge of correctness guarantee.

To implement functional correctness feedback in inference, additional automated verification and repair loops are integrated into the intermediate steps of reasoning. Specifically, LPCM actively assesses whether automated verification is necessary for the current reasoning step during inference. When automated verification is required, LPCM utilizes appropriate specialized tools or models to validate the functionality of the intermediate design. If a functional error is detected, automated repair is triggered, which involves reverting to the last verified functional step in the reasoning chain, incorporating error feedback from the current validation, and regenerating the design of the current step. This process of verification

and repair is repeated iteratively until a functionally correct design is achieved. Through this iterative functional correctness feedback loop, the design's correctness is continuously refined, ultimately approaching 100% functional correctness of the automatically designed processor chips.

2) *Performance Feedback*: Performance optimization aims to improve the PPA of designed process chips, playing a vital role in designing processor chips and adapting foundational software. Existing automated optimization tools, such as deep learning compilers and DSE methods, typically rely on expert-designed rule-based optimization methodologies or machine learning techniques. While these tools significantly reduce the human labor compared to manual tuning, they still encounter issues such as narrow coverage, suboptimal efficiency, and poor cross-domain transferability. To strengthen the capacity of automated performance optimization, cross-stage collaborative design is necessary. LPCM should be capable of directly generating chip layouts from functional specifications. Nevertheless, formulating the design issue directly with the raw bitstream input of processor chips leads to the curse of dimensionality. For instance, the solution space for a 32-bit CPU could grow to $10^{10^{540}}$.

To address the challenges of the enormous solution space, processor chip design has been divided into multiple abstract stages. During logic design, functional specifications are translated into high-level HDL. In circuit design, these high-level HDLs are converted into gate-level netlists. While in physical design, gate-level netlists are turned into chip layouts. This workflow follows a progressive coarse-to-fine design process. Each stage incorporates additional implementation details, progressively introducing more constraints, gradually pruning the solution subspaces with lower probability of containing optimal solutions, thereby reducing the overall solution space size. Inspired by this, to enable automated performance optimization, LPCM needs to adopt a hierarchical search-based inference mechanism guided by performance feedback. This involves building hierarchical decompositions, where the solution space is pruned based on domain knowledge and performance feedback, effectively reducing its dimensionality. Simultaneously, by leveraging the iterative reasoning and Test-Time Scaling (TTS) benefits demonstrated by LLMs during inference, the solution space can be efficiently explored, addressing the enormous solution space challenge and enhancing the performance of automated design.

To implement performance feedback in inference, efficient search techniques must be integrated into the inference process to obtain high-performance outcomes from the vast solution space. Specifically, LPCM can generate different optimization strategies depending on the target hardware architecture and software characteristics, thus building a search tree in which the initial result is the root node, while intermediate nodes and leaf nodes represent the current optimized outcomes. Additionally, by predicting the performance of intermediate nodes and utilizing performance feedback from real-world deployment tests, LPCM can prune suboptimal search branches and generate further optimization strategies based on the current optimal search branch until either reaching a fixed optimization budget or encountering performance improvement saturation.

Through tree search-based inference guided by performance feedback, the performance of hardware and software can be progressively improved, tailored to specific scenarios.

IV. PROCESSOR CHIP DESIGN AGENTS

Building upon LPCM, QiMeng develops two specialized agents: Hardware Design Agent and Software Design Agent, to enable fully automated hardware/software design for processor chips.

A. Hardware Design Agent

With fabrication technologies approaching physical limits, the conventional approach of enhancing chip performance through process scaling below 7nm nodes has encountered fundamental limitations. Therefore, the focus needs to pivot from fabrication-centric advancements to design methodologies innovations. To overcome the fabrication technological constraint in processor chip design, it is necessary to explore fully automated hardware design. This involves establishing a seamless design framework from functional specifications to physical layout, bypassing the conventional multi-stage design hierarchy, including logic design, circuit design, and physical design, to explore a broader cross-stage collaborative design space for superior solutions. The automated design framework must fulfill two critical objectives: 1) Functional correctness, guaranteeing that the automatically designed processor chip delivers accurate computational outcomes; and 2) High performance, optimizing performance such as computational throughput, power efficiency, and area utilization.

To accomplish the aforementioned two critical objectives, the Hardware Design Agent needs to implement automated module decomposition and module generation. This enables obtaining a performance-optimized fine-grained design scheme through module decomposition, followed by generating functionally correct modules, ultimately achieving fully automated hardware design results. Automatic module decomposition addresses the enormous solution space challenge in hardware design by dividing the processor architecture into functionally independent and verifiable modules, aiming at solution space reduction and global performance optimization. Notably, there exist multiple valid decomposition schemes that satisfy functional specifications but exhibit substantial performance variations. Therefore, it is essential to establish a performance-driven decomposition mechanism to enhance the PPA of processor chips. Following decomposition, module generation proceeds according to both functional specifications and the selected decomposition scheme. The subsequent integration of these modules yields the final hardware design outcomes. During module generation, functional correctness must be guaranteed. To this end, Hardware Design Agent synergistically combines LPCM with symbolic methods, establishing an automated verification and repair mechanism to ensure the functional correctness of generated modules. Specifically, LPCM enables the transformation from informal, natural language specifications to formal HDL, accommodating diverse application requirements. After that, using symbolic representations based on BSD [16] to automatically verify and repair the generated module, ensuring functional correctness.

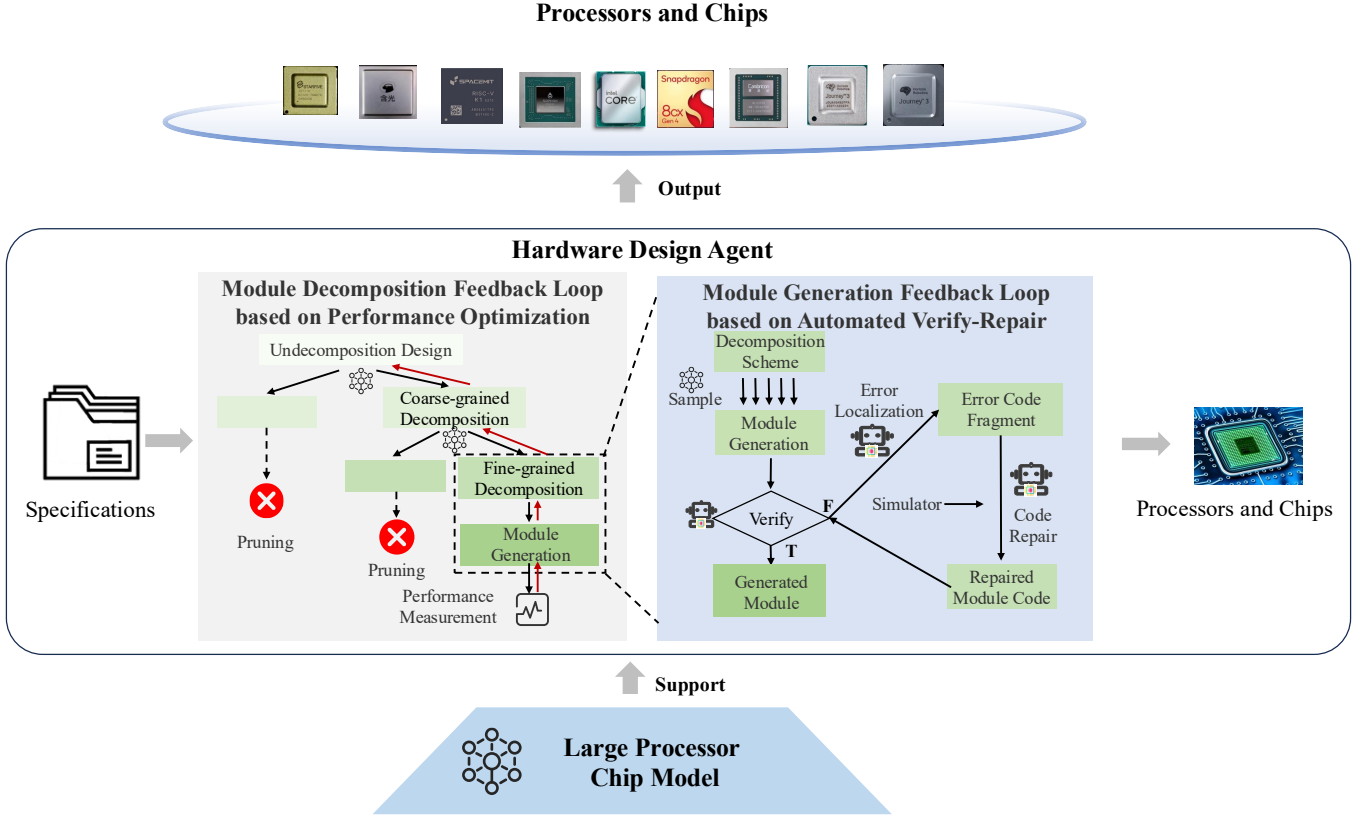


Fig. 3. Structure of Hardware Design Agent which features a dual-loop feedback mechanism: an outer module decomposition feedback loop based on performance optimization and an inner module generation feedback loop empowered by automated verification and repair.

Based on LPCM, we develop a Hardware Design Agent to achieve automated processor chip design from a high-level specification. Hardware Design Agent features a dual-loop feedback mechanism: an outer module decomposition feedback loop based on performance optimization and an inner module generation feedback loop empowered by automated verification and repair, as shown in Figure 3. The outer loop addresses the enormous solution space challenge through module decomposition, while the inner loop addresses the correctness guarantee challenge through automated verification and repair. In implementation, the outer loop is initiated with the undecomposed design as the root node of the search tree. In each iteration, LPCM proposes and identifies promising finer-grained decomposition candidates that bring potential performance gains as child nodes, based on functional specifications, the current decomposition state, and accumulated domain knowledge. Namely, among the available decomposition schemes, nodes demonstrating suboptimal performance are discarded, which simultaneously reduces the solution space dimensionality and lowers computational complexity. This iterative process constructs a module decomposition search tree where leaf nodes correspond to complete decomposition schemes. The final performance evaluation of each leaf node integrates the outer module decomposition scheme with the inner verified modules, enabling backtracking-based optimization and searching of the module decomposition schemes. The evaluation results of the current module decomposition scheme

are added to the domain knowledge base to support subsequent module decompositions. Simultaneously, in the inner loop, LPCM extracts functional specifications corresponding to the target module based on the module decomposition scheme and then generates the corresponding HDL. However, the initially generated HDL may contain functional inaccuracies. To resolve it, it is necessary to verify and repair the generated modules based on the capacity of functional correctness feedback in the inference of LPCM. Specifically, the HDL of target module is first transformed into a BSD representation, along with a subset of input-output pairs sampled from the truth table for simulation-based validation. When discrepancies arise, the erroneous BSD nodes undergo Shannon expansion, facilitating automated error correction. Each repair cycle monotonically increases the BSD’s functional accuracy. Through repeated verify-repair iterations, the BSD asymptotically approaches 100% correctness, ultimately producing a validated module design.

B. Software Design Agent

Foundational software plays a pivotal role in establishing comprehensive technology ecosystems for processor chips, serving as the decisive factor for their successful commercialization and widespread adoption. Nevertheless, adapting and optimizing such software presents formidable obstacles, particularly given the current landscape of fragmented instruction set architectures and diverse software ecosystems. The

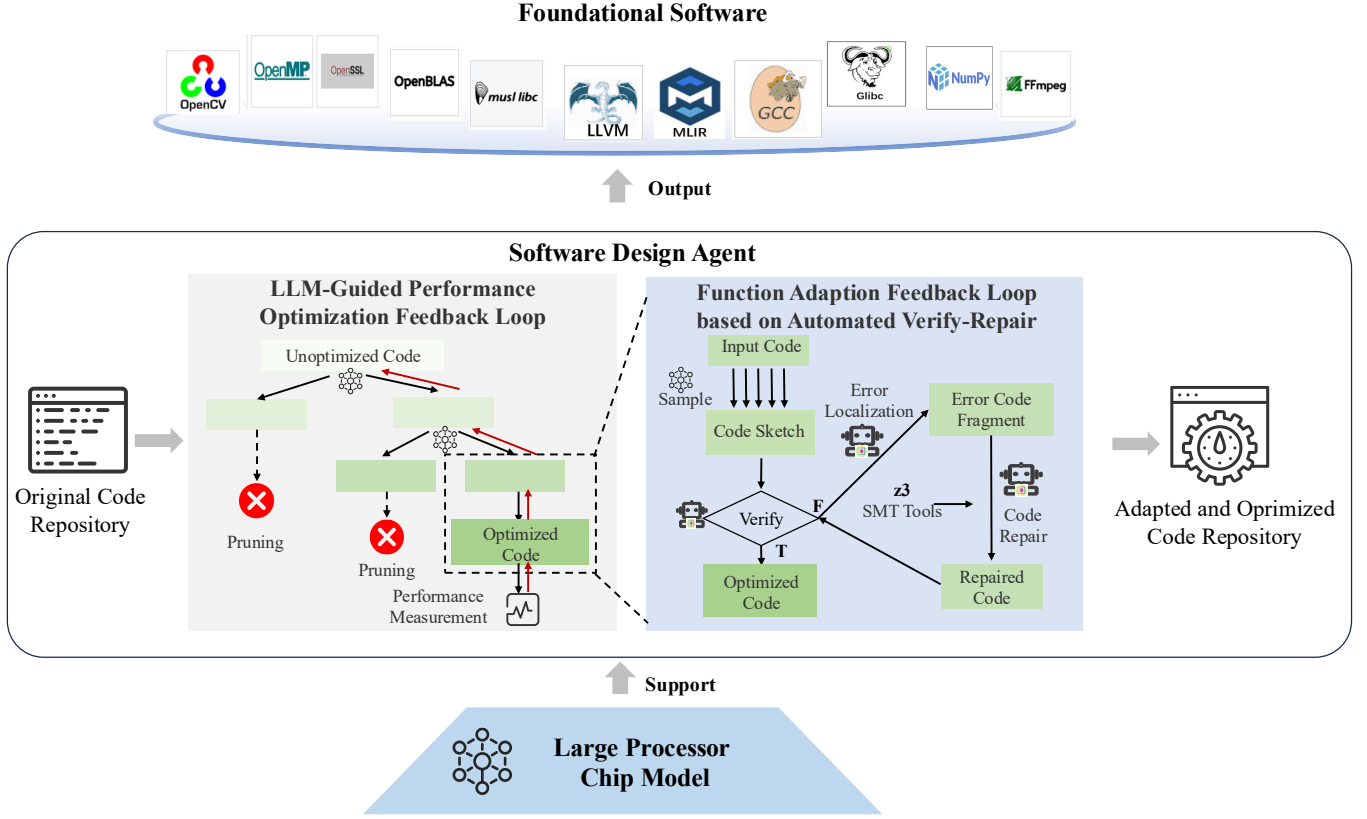


Fig. 4. Structure of Software Design Agent which uses a dual-loop feedback mechanism: the outer loop focusing on performance optimization feedback through LLMs-guided search and the inner loop managing function adaption feedback through automatic verification and repair.

RISC-V ISA exemplifies this challenge, while its open and modular design offers unprecedented flexibility, it simultaneously introduces complexity orders of magnitude greater than x86/ARM architectures. With nearly 100 optional instruction extensions, including Vector Extension, Matrix Extension, and Cryptographic Extension, the combinations grow exponentially, while each variant demands meticulous compatibility verification across the entire software stack. For example, the openEuler OS [17] comprises over 10,000 repositories containing 4 million files, all requiring exhaustive validation for different RISC-V instruction combinations. This combinatorial explosion renders conventional manual development approaches impractical, crystallizing two fundamental challenges: 1) achieving comprehensive functional adaptation to ensure software stability across diverse instruction sets, and 2) conducting deep performance optimization to fully exploit hardware capabilities.

To address these challenges for developing the foundational software ecosystem on specific processors, it is essential to implement AI-driven automated methods for function adaptation and performance optimization of foundational software. The function adaptation problem can be abstracted as a “program generation” task, where an agent translates the source code/platform into a target code/platform. Typical applications include automated compiler tool-chain design and automated tensor program transcompilers [18]–[23]. The key to this approach is synergistically combines neural and symbolic

methods: LLMs handle high-level program skeleton generation through meta-prompts for flexibility, while SMT-based program synthesis ensures correctness by rectifying low-level implementation errors. The performance optimization problem for foundational software can be abstracted as a “search” problem where the goal is to efficiently explore the enormous space of optimization primitives and parameter combinations to find the optimal configuration. Typical applications include the automated generation of high-performance operator libraries [24], [25] and OS configuration optimization [26]. LLMs can efficiently guide this search by leveraging their in-context learning capabilities through carefully designed meta-prompts encoding hardware characteristics and optimization primitives, then effectively pruning the search space to discover optimal implementations tailored to specific hardware-software combinations.

Following this approach, we developed the Software Design Agent based on LPCM, as shown in Figure 4. The Software Design Agent uses a dual-loop feedback mechanism, with the outer loop focusing on performance optimization feedback through LLMs-guided search and the inner loop managing function adaptation feedback through automatic verification and repair. This process can finally transform the original code repository into one that is both adapted and optimized to enhance the foundational software ecosystem. Note that the outer loop addresses enormous solution space challenges through hierarchical decomposition and optimization feedback, while

TABLE I
RESULTS OF AUTOMATED FRONT-END DESIGN BY QIMENG-CPU SERIES
COMPARED WITH EXISTING METHODS.

Target Circuit	Methods	Scale	Performance
Adder [29]	RL	118	NA
Circuit Modules [30]	DT	186	NA
Circuit Modules [31]	EL	~ 2500	NA
8-bit CPU [32]	LLM	999	NA
QiMeng-CPU-v1 (RISC V-32 CPU) [16]	BSD	~ 4 Million	1.62×10^4
QiMeng-CPU-v2 (Superscalar CPU) [33]	S-BSD	~17 Million	6.29×10^6

the inner loop tackles correctness guarantee challenges through automated verification and repair. Specifically, in the outer performance optimization feedback loop, the original code is used as the starting point for a Monte Carlo Tree Search [27]. Then, the domain expert knowledge from the LLMs helps evaluate the search tree, prune inefficient branches, and select branches with potential performance gains. The tree is finally refined based on performance measurements, forming an iterative “observe-prune-optimize-evaluate” loop until the desired optimization results are achieved. In the inner function adaptation feedback loop, we utilize TTS of LLMs for program sampling to generate diverse program sketches, followed by unit testing and execution trace analysis on high-quality sketches to identify minimal erroneous fragments. These fragments are then repaired using solver-based program synthesis such as Z3 [28], regarding the original implementation, iterating the “generate-verify-repair” loop until functional equivalence is achieved.

V. APPLICATIONS

Leveraging LPCM and Hardware/Software Design Agents, QiMeng has developed a series of innovative automated design applications to address various hardware/software design requirements. These implementations effectively solve practical needs by strategically applying specific components of Hardware/Software Design Agents and achieving hardware/software automated design for processor chips.

A. Automated Front-end Design

Automating the design of general-purpose computer CPUs has been a pivotal research challenge since the 1950s, drawing the attention of AI pioneers like Turing and Church [10]. With the advancement of AI technologies, various methods such as decision trees [30], LLMs [32], and RL [29] have been attempted for automated circuit design. Nevertheless, the absence of well-defined formal representations of circuits limits the precision of existing methods, restricting current capabilities to circuits of roughly thousands of gates without guaranteeing accuracy at larger-scale circuits.

To achieve automated design for large-scale processors and chips, we employ the module generation feedback loop based on automated verification and repair within the Hardware Design Agent to ensure functional correctness, while adopting

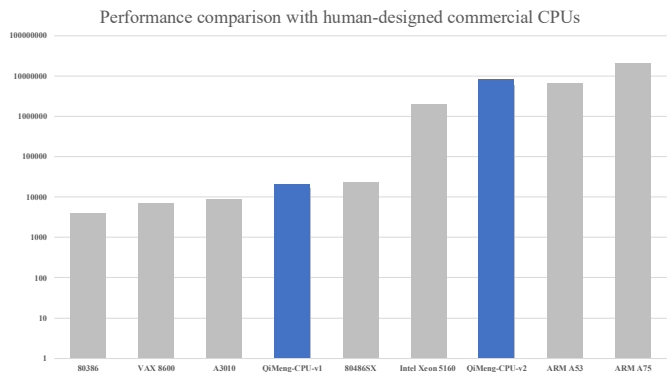


Fig. 5. Performance of QiMeng-CPU series (blue) compared with human-designed commercial CPUs (grey) on the official ARM CPU benchmark Dhrystone. The results show that QiMeng-CPU-v1 is comparable to Intel 486 (1990s CPU), while QiMeng-CPU-v2 is comparable to Arm Cortex A53 (2010s CPU).

Binary Speculation Diagrams (BSD) as the circuit’s graph-based representation [16]. BSD exhibits two key characteristics for combinational logic circuits: 1) design accuracy improves monotonically with the number of design nodes, and 2) accuracy asymptotically converges to 100% as the number of data sampling increases. The implementation initializes with a randomly generated BSD, based on the automated verification and repair feedback, and iteratively verifies the current BSD in a simulator. When errors are detected, the corresponding BSD nodes are repaired by Shannon expansion, thereby monotonically increasing the functional accuracy of the BSD. By iteratively cycling the automated verification and repair steps, the functional accuracy progressively converges to 100%. Applying this methodology, the entire front-end design of a 32-bit RISC-V CPU was automatically completed within 5 hours, producing QiMeng-CPU-v1 (also named Enlightenment-1), which is the world’s first processor core designed fully automatically [16]. As shown in Table I, QiMeng-CPU-V1 has about 4 million gates, more than 1700× larger than existing work and achieving a industrial-scale. Taped out in 2021, QiMeng-CPU-v1 achieves computational performance comparable to Intel’s 1990s-era 486 processors, as shown in Figure 5.

In addition, we leverage the module decomposition feedback loop based on performance optimization within the Hardware Design Agent to enhance the performance of automated front-end design. For automated pipeline design, gate-level dependency analysis is employed to explore decomposition strategies for pipeline modules, enabling the identification of more efficient fine-grained gate-level pipeline partitioning solutions. Subsequently, a gate-level pipeline controller is implemented, facilitating gate-level short-path forwarding. Finally, the decomposed pipeline modules are synthesized into circuits using BSD. The resulting gate-level pipelines yield an average performance gain of 1.57×, outperforming manually designed counterparts by 37% in throughput [34]. Meanwhile, in the automated design of superscalar processors, a simulated annealing algorithm is applied to search for predictable processor states, enabling instruction-level module

TABLE II
COMPARISON OF OUR CODEV SERIES AGAINST VARIOUS BASELINE MODELS. RESULTS ARE CITED FROM THE ORIGINAL PAPER.

Type	Model	Model size	Open source	VerilogEval-Machine (%)			VerilogEval-Human (%)			RTL2LLM v1.1 (%)	
				pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	Syntax	Func.
Base LLMs	GPT-3.5	-	×	60.9	75.0	79.9	33.5	45.9	50.0	79.3	51.7
	GPT-4	-	×	60.0	70.6	73.5	43.5	55.8	58.9	100.0	65.5
	StarCoder [35]	15B	✓	46.8	54.5	59.6	18.1	26.1	30.4	93.1	27.6
	CodeLlama [36]	7B	✓	43.1	47.1	47.7	18.2	22.7	24.3	86.2	31.0
	DeepSeek-Coder [37]	6.7B	✓	52.2	55.4	56.8	30.2	33.9	34.9	93.1	44.8
	CodeQwen [38]	7B	✓	46.5	54.9	56.4	22.5	26.1	28.0	86.2	41.4
	Qwen2.5-Coder [39]	7B	✓	66.2	79.2	83.9	34.6	45.6	51.0	89.6	41.4
Fine-Tuned LLMs	ChipNeMo [40]	7B	×	43.4	-	-	22.4	-	-	-	-
	RTLCoder-Mistral [41]	7B	✓	62.5	72.2	76.6	36.7	45.5	49.2	<u>96.6</u>	48.3
	RTLCoder-DS [41]	6.7B	✓	61.2	76.5	81.8	41.6	50.1	53.4	93.1	48.3
	BetterV-CL [42]	7B	×	64.2	75.4	79.1	40.9	50.0	53.3	-	-
	BetterV-DS [42]	6.7B	×	67.8	79.1	84.0	45.9	53.3	57.6	-	-
	BetterV-CQ [42]	7B	×	68.1	79.4	84.5	46.1	53.7	58.2	-	-
	CraftRTL-CL [43]	7B	×	78.1	85.5	87.8	63.1	67.8	69.7	93.9	52.9
	CraftRTL-DS [43]	6.7B	×	77.8	85.5	88.1	65.4	70.0	72.1	84.3	58.8
CodeV-Verilog	CodeV-Verilog-CL	7B	✓	78.1	86.0	88.5	45.2	59.5	63.8	93.1	62.1
	CodeV-Verilog-DS	6.7B	✓	77.9	88.6	90.7	52.7	62.5	67.3	89.7	55.2
	CodeV-Verilog-CQ	7B	✓	77.6	88.2	90.7	53.2	65.1	68.5	93.1	55.2
	CodeV-Verilog-QC	7B	✓	<u>80.1</u>	87.9	90.5	59.2	65.8	69.1	<u>96.6</u>	51.7
CodeV-All	CodeV-All-CL	7B	✓	78.5	85.6	87.6	46.6	58.8	62.5	<u>96.6</u>	55.2
	CodeV-All-DS	6.7B	✓	79.8	86.0	86.7	53.0	63.3	67.2	<u>96.6</u>	51.7
	CodeV-All-CQ	7B	✓	79.9	88.3	<u>91.1</u>	54.1	65.1	68.6	93.1	58.6
	CodeV-All-QC	7B	✓	81.9	89.9	92.0	56.6	67.9	71.4	<u>96.6</u>	55.2
CodeV-R1	CodeV-R1-Distill	7B	✓	76.2	85.6	87.0	<u>65.7</u>	<u>76.8</u>	<u>79.7</u>	-	<u>75.8</u>
	CodeV-R1	7B	✓	76.5	84.1	85.7	69.9	79.3	81.7	-	86.1

decomposition. A Stateful Binary Speculation Diagram (S-BSD) architecture is then devised to generate instruction modules by predicting inter-instruction data dependencies, thereby achieving instruction-level parallelism and enhancing processor performance. Leveraging this methodology, we develop the world’s first automated designed superscalar CPU, QiMeng-CPU-v2 [33], which delivers about $380\times$ speedup over single-cycle predecessor QiMeng-CPU-v1 (Enlightenment-1), and matches the performance of the ARM Cortex A53, as shown in Table I and Figure 5.

Notably, the design of QiMeng-CPU-v1 initializes with random circuits and leverages the module generation feedback loop based on automated verification and repair within the Hardware Design Agent, whereas QiMeng-CPU-v2 extends by further integrating the module decomposition feedback loop based on performance optimization within the Hardware Design Agent. However, both of them currently operate without utilizing LPCM. In subsequent research, we intend to integrate the automated HDL generation methods (introduced in Section V-B) based on LPCM with the verification-repair-guided module generation loop from QiMeng-CPU-v1 and performance-guided module decomposition loops from QiMeng-CPU-v2, ultimately constructing the complete Hardware Design Agent.

B. Automated HDL Generation

HDLs play a pivotal role in processor chip design by enabling the creation of Register Transfer Level (RTL) code that connects natural language specifications with manufacturable chip layouts. As a critical determinant of functionality, performance, power efficiency, and production costs, HDL

implementation currently consumes over 70% of chip development cycles according to NVIDIA, highlighting the urgent need for automation solutions.

LLMs have shown revolutionary potential in software engineering, such as GitHub Copilot’s 55% efficiency improvement [46], yet their application to HDL generation remains suboptimal due to two challenges: 1) Data scarcity: Public code dataset contain $42\times$ fewer Verilog samples (1.91M) than Python code (80.6M) [47]. Besides, there is almost no HDL-focused competition datasets like LeetCode or Codeforces in software; 2) Semantic disparity: HDLs demand precise low-level control, such as signal bit-width management, that creates significant abstraction gaps between specifications and implementations.

To tackle the above challenges, we present a multi-level summarization-based data synthesis approach and fine-tune general-purpose LLMs using the synthesized data to develop a series of HDL code generation models, referred to as CodeV [48]. CodeV implements the module generation component of the hardware design agent, which utilizes the LPCM for hardware design. Specifically, building on the insight that “summarizing code to natural language is easier and more straightforward than generating code from natural language”, we develop a progressive abstraction technique that converts existing HDL code into high-quality natural language-code pairs, which effectively bridges the HDL-semantic gap. As shown in Table II, this process yields 180k optimized training samples, enabling CodeV-Verilog to achieve 80.1% pass@1 on VerilogEval-Machine [49], surpassing previous SOTA open-source models RTLCoder [41].

TABLE III
COMPARISON OF CODEV-R1 ON RTLLM v2 AND VERILOGEVAL v2.

Type	Model	Model size	Open source	VerilogEvalv2-SR (%)			VerilogEvalv2-CC (%)			RTLLM v2 (%)		
				pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	pass@1	pass@5	pass@10
Base LLMs	GPT-4o	-	×	64.1	73.7	76.2	57.6	66.1	69.0	56.5	70.3	75.2
	DeepSeek-R1 [44]	671B	✓	77.5	84.7	87.4	79.1	85.1	87.1	64.7	75.8	79.7
	DeepSeek-V3 [1]	671B	✓	62.4	71.7	75.0	68.7	76.3	78.2	59.1	71.5	73.3
	QWQ-32B [45]	32B	✓	64.2	77.3	80.1	64.0	77.8	80.9	52.9	68.0	71.2
	DeepSeek-R1-Distill-Qwen-32B [44]	32B	✓	43.9	63.3	69.2	53.8	69.8	73.8	42.4	62.1	67.0
	DeepSeek-R1-Distill-Qwen-7B [44]	7B	✓	0.6	2.2	3.5	2.0	7.0	11.3	0.0	0.0	0.0
	Qwen2.5-Coder-32B-Instruct [39]	32B	✓	47.5	60.7	64.7	46.6	59.0	62.8	47.8	63.9	67.8
	Qwen2.5-Coder-7B-Instruct [39]	7B	✓	31.3	49.3	54.6	30.5	46.8	52.0	36.1	52.4	57.6
CodeV-R1	CodeV-R1-Distill	7B	✓	65.2	75.2	77.5	65.5	75.6	78.2	57.2	71.9	77.1
	CodeV-R1	7B	✓	68.8	78.2	81.1	69.9	78.2	80.9	68.0	78.2	81.7

* WSR: Specification-to-RTL. CC: Code Completion.

TABLE IV
RESULTS OF AUTOMATED OS CONFIGURATION OPTIMIZATION BY AUTOOS COMPARED WITH EXISTING METHODS.

Method	OS Configuration Task (UnixBench)		
	PolyOS on Sifive Unmatched	Fedora on Sifive Unmatched	Ubuntu on PC Machine
Default	309	207	3885
GPT-3.5	283 (-8.5%)	194 (-6.3%)	3898 (+0.3%)
AutoOS	335 (+8.4%)	260 (+25.6%)	4238 (+9.0%)

Based on CodeV, we make two key extensions:

1) To better align with real-world development workflows, we extended CodeV-Verilog into CodeV-All through the Chat-FIM-Tag supervised fine-tuning method. CodeV-All not only supports a wider range of languages, including Verilog and Chisel, and a broader set of tasks such as Chat and fill-in-the-middle (FIM), but it also delivers performance on VerilogEval that matches or even surpasses CodeV-Verilog (shown in Table II), which was fine-tuned solely on Verilog. This makes the CodeV series the first set of open-source LLMs designed for multi-scenario HDL generation.

2) Inspired by the reasoning capabilities demonstrated in mathematical and software coding tasks, we proposed several innovations: a rule-based testbench generator that verifies predicted code against a golden reference, a round-trip data synthesis method that generates high-quality natural language-code pairs using only source code snippets as input, and adaptive DAPO, a fast version of DAPO [50] that dynamically adjusts the number of samples per step based on past sample discard rates. These components were integrated into a “distill-then-RL” two-stage training pipeline to develop CodeV-R1 [51], a reasoning-enhanced Verilog generation LLM that is capable of *thinking* and test-time scaling. As shown in Table III, CodeV-R1 achieves 68.6% and 72.9% pass@1 on VerilogEval v2 and RTLLM v2, respectively, outperforming previous state-of-the-art models by 12% to 21%, and matching or even exceeding the performance of the 671B DeepSeek-R1.

C. Automated OS Configuration Optimization

The operating systems (OS) act as a crucial bridge between processors and higher-level software, playing a vital role in maximizing the performance of the processor chips.

The widely used open-source OS Linux, designed to meet the diverse requirements of different application scenarios and processors, consists of over 20 million lines of code contributed by developers around the world, making it one of the most complex software projects to date. This vast codebase presents significant opportunities for optimization, and there is a pressing need to tailor or optimize the OS for specific processors and application scenarios to fully unleash the potential of the entire computer system.

However, customizing or optimizing an OS involves three main challenges. First, the complexity of the task is extremely high. Even just optimizing the OS kernel involves over 15,000 interdependent configuration options [52], [53], which are beyond the capability of conventional optimization methods. Second, the cost of evaluating each configuration is high, as compiling, installing, and testing the OS can take up to 1 to 2 hours [54], which limits the feasibility of data-driven methods like neural networks. Third, the optimization process is highly sensitive, where even a small error could prevent the OS from booting properly and make debugging extremely difficult.

To address these challenges, we leverage the LLM-guided performance feedback loop from our Software Design Agent to develop an automated OS configuration optimization method, AutoOS [26], which can generate optimized kernel configurations without manual intervention and surpass the performance achieved by hardware vendors’ manual optimizations. To achieve this, we introduce an “observe-prune-propose-act-correct” feedback loop, which leverages the prior knowledge embedded in LLMs to eliminate irrelevant configuration options that do not contribute to performance optimization and might cause booting issues, significantly reducing the search space for customization. In just a few search iterations, approximately one day, the method can automatically generate custom-optimized operating system kernel configurations. Compared to manual expert optimization, this approach can boost performance by as much as 25.6%, as shown in Table IV.

D. Automated Compiler Tool-Chain Design

Compilers for modern processors are responsible for two fundamental tasks: 1) accurately and efficiently translating precise and unambiguous programming languages corresponding to the processor’s instruction set, i.e., translation; and 2) constructing compilation optimization sequences within a vast, high-dimensional optimization space, i.e., optimization.

TABLE V
COMPARISON OF OUR AUTOMATED TENSOR PROGRAM TRANSCOMPILER AGAINST STATE-OF-THE-ART LLMs ON DIFFERENT TRANSCOMPILATION DIRECTIONS. REFER TO THE ORIGINAL PAPER [55] FOR MORE DETAILS AND COMPLETE RESULTS .

Source-Target	Method	Compilation Accuracy (%)				Computation Accuracy (%)			
		CUDA C	BANG C	HIP	C With VNNI	CUDA C	BANG C	HIP	C With VNNI
CUDA C	GPT-4	-	50.6	97.0	84.5	-	7.7	96.4	30.4
	OpenAI o1	-	51.8	98.2	85.1	-	48.2	98.2	55.4
	QiMeng-Xpiler	-	100.0	100.0	100.0	-	91.7	100.0	95.2
BANG C	GPT-4	69.0	-	66.1	23.8	6.5	-	6.5	13.1
	OpenAI o1	71.4	-	97.0	41.7	10.1	-	7.7	23.2
	QiMeng-Xpiler	100.0	-	100.0	100.0	95.8	-	97.0	95.2
HIP	GPT-4	97.0	35.1	-	85.1	97.0	5.4	-	24.4
	OpenAI o1	98.8	42.3	-	88.7	98.2	9.0	-	30.4
	QiMeng-Xpiler	100.0	100.0	-	100.0	100.0	86.9	-	96.4
C With VNNI	GPT-4	81.5	41.7	74.7	-	14.3	6.0	12.5	-
	OpenAI o1	87.5	55.4	97.0	-	51.2	10.7	96.4	-
	QiMeng-Xpiler	100.0	99.4	100.0	-	98.2	88.7	99.4	-

Currently, AI techniques in compilers are mainly focused on improving the optimization sequences within the high-dimensional space of existing compiler frameworks, also known as the Phase Ordering Problem. Yet they struggle to generate an end-to-end compiler that handles both two fundamental tasks for processors.

To address the long-term goal of creating an end-to-end compiler capable of both translation and optimization tasks, we have explored the automated compiler tool-chain design methods based on the Software Design Agent, investigating two different approaches: 1) automatically generating compiler backend code. Building upon existing architectures such as LLVM, we construct compiler backend datasets ComBack [56] and fine-tune the LLMs to improve and fully exploit LLMs’ comprehension ability for compiler backend code VEGA [57]. As a result, we successfully generated compiler backend code tailored to a specific processor with an accuracy rate exceeding 70%, with explicit confidence scores highlighting critical regions requiring minimal manual refinement. This approach promises to revolutionize conventional backend development workflows. 2) Using LLM as an end-to-end compiler. We discover that the translation task of compilers shares significant similarities with natural language translation, an area where LLMs excel. This suggests that LLMs have the potential to revolutionize compiler construction to act as a real compiler. However, since natural languages are inherently ambiguous while programming languages have precise semantics defined by grammar, directly applying LLMs to translation tasks leads to suboptimal results. For instance, using GPT-4 for translating C language to RISC-V assembly yields an accuracy rate below 50%, with complex functions performing near zero. Therefore, we proposed an end-to-end neural compiler method [58] based on the Software Design Agent. This method combines grammar information from programming languages and compiler domain knowledge to guide the generation of specialized LLMs. On one hand, data augmentation techniques guided by compiler expertise were used to create high-quality datasets to fine-tune LLMs. On the other hand, we leverage the program’s grammar information during the inference stage for LLMs

tailored to the specific translation task. This combination enabled us to achieve over 99% accuracy for C language translation on the ExeBench [59] dataset and successfully compile code from real-world datasets like AnsiBench [60] and CoreMark [61], confirming the feasibility of this approach. Going forward, we will continue to refine how to enhance the performance of the Software Design Agent based on LPCM used directly as end-to-end compilers.

E. Automated Tensor Program Transcompiler

Contemporary LLMs, including prominent examples like GPT and DeepSeek, exhibit deep dependencies on NVIDIA’s CUDA ecosystem. This reliance encompasses both vendor-provided libraries such as cuBLAS [62], cuDNN [63], TensorRT [64], and community-developed kernels such as FlashAttention-v1 [65], FlashAttention-v2 [66], FlashAttention-v3 [67]. Even the domestic open-source LLM DeepSeek [68] has also developed tailored acceleration libraries like FlashMLA [69] and DeepGEMM [70] for NVIDIA GPUs. However, the software ecosystem for domestic AI chips faces significant fragmentation, as different chip manufacturers develop their own independent operator libraries. This makes it challenging to unify the software ecosystems across domestic AI chips, hindering the widespread adoption of these chips.

To address this challenge, we have developed an automated tensor program transcompiler, QiMeng-Xpiler [55], based on the Software Design Agent, enabling “Write Once, Run Anywhere” across different AI chips, including both NVIDIA GPUs and domestic AI Chips. The key is that the program translation process is automatically conducted as a series of neural-symbolic transformation passes based on the function adaptation feedback loop, where LLMs generate high-level program sketches, and the incorrect code details are repaired by small-scale symbolic synthesis. Meanwhile, the optimal transformation passes are identified via hierarchical auto-tuning based on the performance optimization feedback loop. Specifically, we combine inter-pass Monte Carlo Tree Search [27] for optimal transformation sequencing and intra-

TABLE VI

PERFORMANCE COMPARISON OF MATRIX MULTIPLICATION METHODS ACROSS DIFFERENT HARDWARE PLATFORMS C910(GFLOPS), NVIDIA RTX4070(TFLOPS), NVIDIA A100(TFLOPS). SPEEDUP RATIOS FOR QIMENG-GEMM ARE CALCULATED AGAINST OPENBLAS (C910) AND CUBLAS (RTX 4070, A100). THE A100 AND RTX4070 GPU UTILIZES CUDA CORES.

Hardware	Method	Dimension (M = K = N)		
		1024	2048	4096
C910 (RISC-V)	GPT-4o [4]	0.14	0.10	0.09
	Claude 3.5 Sonnet [75]	2.64	1.56	0.74
	OpenBLAS [76]	5.01	5.11	4.85
	QiMeng-GEMM	9.91(1.98x)	10.08(1.97x)	10.23(2.11x)
RTX 4070 (NVIDIA)	GPT-4o	1.77	1.78	1.65
	Claude 3.5 Sonnet	1.71	1.79	1.61
	cuBLAS [62]	10.79	12.77	12.78
	QiMeng-GEMM	11.47(1.06x)	13.31(1.04x)	14.16(1.11x)
A100 (NVIDIA)	GPT-4o	4.19	4.27	4.71
	Claude 3.5 Sonnet	4.64	5.33	5.27
	cuBLAS	16.26	17.20	18.97
	QiMeng-GEMM	12.61(0.77x)	16.17(0.94x)	18.27(0.96x)

pass constraint-based auto-tuning of critical tuning parameters, such as memory tiling configurations. Ultimately, our solution enables an automated tensor program transcompiler across various processors like Nvidia GPUs [71], Cambricon MLU [72], AMD MI accelerators [73], Intel DLBoost [74], and programming models like SIMT, SIMD. In real-world applications such as LLMs, experiments on those 4 diverse processors demonstrate that QiMeng-Xpiller correctly translates different tensor programs at the accuracy of 95% on average, as shown in Table V.

F. Automated High-Performance Library Generation

Leading hardware vendors, such as NVIDIA, ARM, Intel, AMD, and Cambricon, invest heavily in manually optimized libraries to extract peak performance from their processors. These expert-crafted solutions demand intimate knowledge of microarchitecture details, requiring careful parallelization of computations and memory operations, often implemented in vendor-specific languages or assembly code. While delivering exceptional performance, this manual optimization paradigm fundamentally lacks scalability and portability across different hardware architectures.

To address these challenges, in addition to leveraging existing software ecosystems through the aforementioned automated tensor program transcompiler, we pioneer an automated approach called QiMeng-GEMM [77] based on Software Design Agent for generating high-performance libraries with matrix multiplication, i.e. GEMM, as our primary target due to its central role in LLMs [68], [78], deep learning [79], [80], and scientific computing [81]. The proposed QiMeng-GEMM is the first to automatically generate high-performance GEMM code by exploiting LLMs. Specifically, we have abstracted common GEMM optimization methods and hardware architecture features, and created a set of general meta-prompts for LLMs to generate high-performance matrix multiplication operators. These meta-prompts enable LLMs to understand and implement optimization goals by capturing the architectural features of different platforms. We then integrate the performance feedback loop in the Software Design Agent with Tree

TABLE VII

PERFORMANCE COMPARISON OF GEMM AND CONVOLUTION OPERATIONS ACROSS DIFFERENT HARDWARE PLATFORMS, MEASURED IN GFLOPS (K1, A76) AND TFLOPS (A100). THE A100 GPU UTILIZES TENSOR CORES, WITH SPEEDUP RATIOS (IN PARENTHESES) FOR QIMENG-TENSOROP CALCULATED AGAINST OPENBLAS (K1, A76) AND CUBLAS/CUDNN (A100).

Hardware	Method	Matrix Multiplication (M=K=N)		
		1024	2048	4096
K1 (RISC-V)	DeepSeek-V3 [68]	0.33	0.31	0.23
	OpenBLAS [76]	4.19	4.46	4.76
	QiMeng-TensorOp	9.74(2.32x)	10.29(2.31x)	11.74(2.47x)
A76 (ARM)	DeepSeek-V3	0.04	0.04	0.04
	OpenBLAS	31.25	33.48	34.27
	QiMeng-TensorOp	35.70(1.14x)	36.77(1.10x)	37.31(1.09x)
A100 (NVIDIA)	DeepSeek-V3	17.74	17.31	18.76
	cuBLAS [62]	246.10	292.20	298.44
	QiMeng-TensorOp	262.05(1.06x)	290.86(1.00x)	293.44(0.98x)
Hardware	Method	Shape of Feature Map (N, C, H, W) Shape of Filter (K, C, R, S)		
		(64,64,56,56) (64,64,3,3)	(64,128,56,56) (128,128,3,3)	(32,512,14,14) (512,512,3,3)
K1 (RISC-V)	DeepSeek-V3	0.01	0.01	0.01
	OpenBLAS	6.33	6.51	7.31
	QiMeng-TensorOp	6.55(1.03x)	8.08(1.24x)	8.96(1.23x)
A76 (ARM)	DeepSeek-V3	0.06	0.03	0.05
	OpenBLAS	12.97	19.33	27.92
	QiMeng-TensorOp	28.82(2.22x)	30.84(1.60x)	32.98(1.18x)
A100 (NVIDIA)	DeepSeek-V3	14.77	20.51	14.79
	cuDNN [63]	117.96	120.59	136.63
	QiMeng-TensorOp	116.73(0.99x)	121.48(1.01x)	125.71(0.92x)

of Thoughts [82] (ToT) techniques to systematically explore optimization primitive combinations. This allows us to explore all possible optimization sequences generated by the meta-prompts, thus enabling the generation of high-performance matrix multiplication operators that are tailored to different hardware architecture features.

Further extending our LLM-based automation framework, we propose QiMeng-TensorOp [83], the first approach to automatically generate high-performance tensor operators with hardware primitives by leveraging LLMs. We develop structured hardware-intrinsic optimization prompts and a knowledge-guided workflow, enabling LLMs to comprehend platform-specific architectures and optimization strategies. To optimize the generated operators, we design an LLM-guided Monte Carlo Tree Search (MCTS) algorithm, which effectively enhances the efficiency and performance of tuning primitive-level tensor operators on specific hardware.

We further propose QiMeng-Attention, the first hardware-aware automated framework for cross-platform Attention operator generation. We propose an LLM-friendly Thinking Language (LLM-TL) to help LLMs decouple the generation of high-level optimization logic and low-level implementation on GPU, and enhance LLMs' understanding of the attention operator. Along with a 2-stage reasoning workflow, TL-Code generation and translation, the LLMs can automatically generate FlashAttention implementation on diverse GPUs, establishing a self-optimizing paradigm for generating high-performance attention operators in attention-centric algorithms.

We have validated these approaches on diverse platforms such as the Xuantie C910 development board [84], MuseBook (K1) [85], ARM A76 [86], and NVIDIA GPUs (RTX

TABLE VIII
PERFORMANCE (TFLOPS) COMPARISON ACROSS ATTENTION
OPERATORS, NVIDIA GPUS (T4, RTX 8000, A100) UNDER THE
CONFIGURATION OF HEAD DIMENSION 128, SEQUENCE LENGTH 2048,
BATCH SIZE 8, HEAD NUMBER 16, GQA GROUPS 8 AND WITHOUT CAUSAL
MASK. SPEEDUP RATIOS ARE CALCULATED AGAINST THE PYTORCH
IMPLEMENTATION OF DEEPSEEK-V3.

Hardware	Method	Variant of Attention		
		MQA	GQA	MQA
T4 (NVIDIA)	cuDNN [63]	12.95	13.02	13.03
	FlexAttention [91]	14.83	14.95	14.64
	Flash Attention v1 [65]	10.95	10.95	11.01
	DeepSeek-V3 [68]	6.11	3.97	5.99
	QiMeng-Attention	18.59(3.04x)	18.82(4.74x)	18.14(3.03x)
RTX 8000 (NVIDIA)	cuDNN	32.2	32.1	31.2
	FlexAttention	33.2	33.4	33.5
	Flash Attention v1	21.2	21.1	21.3
	DeepSeek-V3	13.4	8.8	13.2
	QiMeng-Attention	44.9(3.35x)	43.3(4.92x)	43.4(3.29x)
A100 (NVIDIA)	cuDNN	190.0	189.6	189.9
	FlexAttention	143.2	143.5	143.5
	Flash Attention v2 [67]	208.2	200.0	200.7
	DeepSeek-V3	52.4	23.1	38.4
	QiMeng-Attention	201.1(3.84x)	186.2(8.06x)	187.6(4.89x)

4070 [87], RTX 8000 [88], T4 [89] and A100 [90]), see Table V-E, Table VII and Table VIII. On the RISC-V platform, the high-performance matrix multiplication operator generated by QiMeng-GEMM and QiMeng-TensorOp achieves up to 211% and 251% of OpenBLAS's performance, respectively. On the NVIDIA platform, they reach up to 115% and 124% of cuBLAS's performance, respectively. Compared to conventional LLM prompt methods, our approach significantly improves the performance of the generated code and boosts development efficiency. To validate the performance of the Qimeng-Attention, we conducted experiments across various NVIDIA hardware architectures. On the NVIDIA T4 platform and NVIDIA RTX8000 platform, the high-performance attention operator generated by Qimeng-Attention consistently achieves superior performance metrics compared to all four implementations.

VI. RELATED WORK

A. Automated Chip Design

Automating the design of processor chips is one of the key challenges in computer science. Early EDA tools, based on predefined rules and Boolean logic, enabled the automation of specific design steps such as logic synthesis, placement, and routing. Later, researchers introduced automated design methodologies based on domain-specific languages (DSLs), HLS, and DSE, etc. With advancements in AI, the automated design of processor chips has evolved into a more intelligent, data-driven phase by leveraging AI technologies. Techniques like random forests, RL, and GNNs are now being applied to enhance the EDA workflow in tasks such as automated performance evaluation, placement, and routing. However, these approaches primarily use AI to optimize the efficiency or performance of existing EDA processes without altering the fundamental processor chip design flow. In recent years, the concept of fully automated processor chip design has become a prominent research area, with approaches utilizing RL, random forests, and LLMs to design processor chips from functional

requirements or design specifications without human effort. Nonetheless, current efforts still face challenges in improving the scale and accuracy of designed processor chips.

1) *EDA-based Automated Chip Design*: The conventional design flow based on EDA tools can be roughly categorized into three stages: logic design, circuit design, and physical design [92]. With AI technology advancements, AI-based methods have been integrated into these three stages. The primary objective of these methods is to enhance specific steps within the conventional flow, thereby improving flow efficiency and design performance, instead of fundamentally altering the conventional flow [92], [93].

The logic design stage aims to generate a hardware description, represented by HDLs such as Verilog and VHDL. This is achieved by either manually programming based on functional requirements or utilizing HLS tools based on hardware functionalities described in high-level programming languages such as C, C++, or SystemC. The former approach simplifies the design flow through hardware abstraction. For instance, Nurvitadhi et al. [94] propose an automated transaction-to-pipeline transcompilation methodology. The ASSIST framework [95] supports RISC architecture design via micro-operation languages but lacks control over pipeline optimization. TL-Verilog [96] partitions combinational logic through temporal abstraction but exhibits deficiencies in data hazard detection. Languages such as BSV [97] and Koika [98] facilitate formal verification but enforce single-cycle rule execution without dynamic scheduling. The latter approach generates hardware descriptions from C/C++. For example, Rokicki et al. [99] generate processor cores from C++ while requiring manual handling of bypass logic. Josipović et al. [100] introduce dynamic scheduling to optimize pipeline performance, while Dahlia [101] leverages affine types to ensure predictability in statically scheduled accelerators. However, these conventional methods rely on formal language template conversion, which incurs high learning costs and constrains design spaces. Thus, recent advancements employ AI algorithms for rapid estimation of quality, performance, and timing to enhance HLS efficiency. For example, Zhao et al. [102] utilize linear regression and Artificial Neural Networks (ANNs) to predict routing congestion in HLS. Makrani et al. [103] propose a neural network (NN)-based approach to predict resource utilization and performance on specific field programmable gate arrays (FPGAs), thereby improving the efficiency of DSE. Ferianc et al. [104] employ Gaussian processes for latency estimation to optimize accelerator configuration selection.

The circuit design stage, also known as logic synthesis, aims to transform hardware descriptions into gate-level circuits, i.e., netlists. During this stage, Boolean expressions and logical structures are optimized based on specified process libraries to achieve minimal logical expressions and netlists. LSOracle [105] employs Deep Neural Networks (DNNs) to intelligently differentiate circuit modules, dynamically selecting the most effective optimizers between And-Inverter Graph (AIG) and Majority-Inverter Graph (MIG) representations. Haaswijk et al. [106] and Zhu et al. [107] reformulate conventional logic optimization as Markov Decision Processes (MDPs), developing deep RL systems that utilize Graph Convolutional Neural

Networks (GCNN) as policy networks. Hosny et al. [108] implement an Advantage Actor-Critic (A2C) RL algorithm to minimize area while adhering to strict timing constraints. Deep-PowerX [109] establishes an accurate error prediction model using DNNs to evaluate approximation circuit errors, enabling significant reductions in dynamic power while maintaining acceptable accuracy thresholds.

The physical design stage aims to generate layouts through placement, clock tree synthesis (CTS), and routing. During the placement stage, AI techniques are primarily employed to produce superior layouts. For example, Google formalizes placement as a sequential decision-making problem that can be addressed using RL, resulting in human-competitive placement within 6 hours [110]. During the CTS stage, AI techniques play a crucial role in optimizing clock tree structures and predicting performance. Lu et al. [111] integrate generative adversarial networks (GANs) with RL to minimize clock skew and total clock tree length through topology prediction and optimization. Nagaria and Deb [112], along with Kwon et al. [113], utilized convolutional neural networks (CNNs) and DNNs, respectively, to predict critical CTS parameters, including gating cell counts, buffer distributions, and wireload characteristics, thereby significantly enhancing the quality of synthesis. During the routing stage, AI techniques contribute to routing prediction and estimation. He and Bao [114] apply RL to train agents for autonomous decision-making regarding spatial search strategies during routing optimization, dynamically selecting optimal neighboring nodes to enhance design quality. Liang et al. [115] and Alawieh et al. [116] model routing congestion prediction as image-to-image translation tasks using CNNs and conditional GANs, respectively, achieving high-precision hotspot predictions to guide routing optimization.

2) *Fully Automated Chip Design*: In recent years, fully automated chip design has emerged as a prominent research focus, particularly in automating front-end chip design directly from functional specifications and design specifications. For instance, PrefixRL [29] applies RL to synthesize circuits of approximately 100 gates; Chen et al. [30] uses random forests to design 200-gate circuits; and Rai et al. [31] employ ensemble learning techniques to automatically design circuits with up to 2500 gates. However, these efforts are limited in scale and fall short on the precision requirements for complex circuits like CPUs. Moreover, the limited validation through small test cases in existing approaches fails to ensure the robustness needed for industrial chip fabrication.

Additionally, academia and industry have begun exploring LLM-based chip logic design, leveraging the natural language comprehension capabilities of LLMs to enable end-to-end chip generation. These approaches generally fall into two categories: foundation models and generation frameworks. Foundation models focus on generating HDL code for module-level designs. Among them, DAVE [117] fine-tunes GPT-2 [118] to produce Verilog code with 94.8% accuracy. VeriGen [119] improves generation quality by training on a hybrid dataset of source code and textbooks. ChipNeMo [40] addresses data scarcity through domain-adaptive pretraining of Llama2 [78] on internal datasets from NVIDIA. RTLCoder [41] creates a high-quality SFT dataset by evolving language prompts,

generating code with GPT-3.5, and manually refining it, ultimately surpassing the original model's performance. The Large Circuit Model (LCM) [120] enables feature extraction to accelerate SAT solving and aid in logic synthesis and equivalence checking. Notably, NVIDIA's proprietary CraftRTL model [43] previously led the field, achieving 53.1% accuracy on the RTLLM benchmark.

Generation frameworks capitalize on LLMs' planning and reflection capabilities to decompose complex hardware design tasks into manageable subtasks, iteratively refining the output based on environmental feedback. For example, ChipGPT [121] implements a four-phase pipeline—prompt generation, initial Verilog generation, correction and optimization, and best-design selection—which has proven effective for CPU design. AutoChip [122] reduces syntax errors by integrating compile-time diagnostics into the generation loop, boosting test pass rates by 24.2%. Chip-Chat [123] uses dialogue-based task decomposition and human feedback to design an 8-bit accumulator microprocessor. RTLFixer [124] introduces retrieval-augmented generation to iteratively repair syntax errors in RTL code. ChatEDA [32] automates the RTL-to-GDSII pipeline through task planning, script generation, and execution via LLMs.

Despite these advances, existing LLM-based techniques remain constrained to small-scale module synthesis, document retrieval, and syntax correction. The generation of complex designs still heavily relies on human involvement, limiting the practical application of current methods in meeting stringent design cost and correctness requirements.

B. Automated Software Design

The field of automated software design encompasses two primary research directions: 1) functional adaptation, which enables functional-correctness automated cross-platform/cross-language software migration, and 2) performance optimization, which improves computational efficiency while maintaining platform/language compatibility.

1) *Automated Software Adaptation*: In the area of foundational software function adaptation, typical applications include automated compiler tool-chain design and automated tensor program transcompiler. Existing research methods can be broadly categorized into three approaches: conventional rule-based automation, SMT-based symbolic synthesis, and data-driven methods.

Conventional rule-based methods rely on experts manually defining transformation rules for ASTs and achieving program translation through pattern matching. Notable works include the FCUDA [20] framework, which automates the translation of CUDA to FPGA by defining rules for data communication, computation optimization, and parallel mapping; AMD's HIPIFY tool [125], which automates the migration of CUDA code from Nvidia GPUs to AMD GPUs; and source-to-source compilers like cxgo [126] and C2Rust [127] that follow similar approaches. However, the architectural differences between platforms and languages make it extremely difficult to manually design efficient translation rules, and this approach struggles to handle the exponentially growing combinatorial space of foundational software adaptation.

SMT-based symbolic synthesis methods generate semantically equivalent target code through domain-specific languages or input-output examples. Key works include the Chlorophyll [128] framework, which defines a domain-specific language for the GreenArrays GA144 [129] architecture, breaking the translation process into subproblems such as partitioning, layout, and code generation, and using symbolic synthesis to produce functionally equivalent code. The FACC [130] framework also uses program synthesis based on I/O examples, generating adaptation layer code to bridge the semantic gap between conventional Fourier transform programs and dedicated hardware accelerators. While these methods rely heavily on SMT solvers for constraint solving, they have two main limitations: First, search-based solvers (like Z3) are difficult to scale to general large-scale programs, and second, manual specification of input constraints adds significant engineering overhead.

Data-driven methods have rapidly advanced in recent years, utilizing vast amounts of data to train neural networks for software function migration and adaptation. Early examples include the neural compiler which uses the Transformer model for end-to-end compilation from C to x86 assembly [131], and Meta AI’s Transcoder [22] framework which uses back-translation learning to achieve cross-language translation between C, Python, and JAVA. There have also been some significant breakthroughs in recent years. CodeXGLUE [132] is a code intelligence benchmarking system, and their CodeGPT [132] model has significantly improved code generation capabilities; BabelTower [133] significantly optimizing the C to CUDA translation via the large-scale datasets and proposed metrics; and QiMeng-GEMM [77] which introduces the first automatically framework for generating high-performance matrix multiplication code without human effort. Although these methods have made remarkable progress, the correctness of the generated code is not yet fully assured, requiring manual verification and correction, which remains a key challenge in the field.

2) *Automated Software Optimization*: In the field of foundational software performance optimization, typical application scenarios include the automatic OS configuration optimization and the automatic high-performance library generation. Existing research methods can be broadly categorized into three approaches: conventional expert manual optimization, online learning-based search methods, and LLM-guided efficient search methods.

Conventional expert manual optimization methods rely on domain experts who use their experience to develop optimization strategies. Notable works include: the default operating system configuration options provided by processor manufacturers to maximize hardware performance; high-performance libraries such as cuBLAS [62], cuDNN [63], and TensorRT [64], manually optimized by thousands of engineers at Nvidia; and the FlashAttention series libraries [65]–[67], which were meticulously optimized by expert community developers. However, with the rapid evolution of algorithm models and the diversification of hardware architectures, this method which heavily depends on manual optimization, is increasingly facing significant engineering cost challenges.

Online learning-based search methods leverage AI algorithms such as machine learning and deep learning to automatically explore the optimization space. In OS configuration optimization, HiPerBOT [134] uses Bayesian optimization to adjust application and platform configuration parameters to improve performance; Wayfinder [135] system applies Bayesian optimization to automatically optimize over 200 network configuration parameters in the operating system kernel. In high-performance library generation and optimization, AutoTVM [136] uses XGBoost [137] to train cost models, minimizing the overhead of hardware performance testing, and automates tuning within a search space defined by expert-specified scheduling templates. Ansor [24] generates additional candidate program templates based on expert rules and uses genetic algorithms [138] to efficiently explore the search space. Although these methods have automated certain aspects, they still rely on manually defined search spaces and face challenges such as the long time required for online tuning, limiting their widespread application.

LLM-guided efficient search methods, with their built-in expert knowledge, robust natural language understanding ability, and code comprehension capabilities, introduce a new paradigm for automated software optimization. The AutoOS [26] framework implements a “observe-prune-prose-act-correct” performance feedback loop, leveraging the prior knowledge of LLMs to extend the OS configuration optimization problem to the full space of about 15,000 Linux kernel configuration options. The researchers from the University of Science and Technology of China introduced the TLM [139] framework, transforming the automatic tuning of high-performance programs into a probability generation problem based on LLMs, facilitating more efficient automatic search than random sampling through enhanced semantic understanding. These methods combine LLMs’ prior knowledge to implement efficient search strategies that resemble those of human experts, and have become a significant development direction for foundational software performance optimization.

VII. CONCLUSION AND FUTURE WORK

The conventional paradigm of processor chip design is confronting three fundamental challenges: physical constraints in fabrication technology, requirements of design resources, and growing diversity of ecosystems. To achieve automated processor chip design based on LLMs and AI technologies, this work proposes QiMeng, an innovative paradigm for fully automated hardware and software design for processors and chips. QiMeng establishes a domain-specific LPCM and further develops Hardware Design Agent and Software Design Agent by leveraging the powerful knowledge representation and inferencing capabilities of LPCM. Then the two agents are applied to various application scenarios in processor hardware/software design. Currently, several components of QiMeng have been applied in multiple applications, providing viable solutions for hardware/software design.

With the deeper convergence of AI technologies and EDA tools, automated processor chip design will evolve toward greater efficiency, generality, and intelligence. In future work,

we will follow the roadmap of QiMeng to accomplish the top-down phase, then proceed to the bottom-up and iteration phases. Simultaneously, we will continue exploring the integration of reinforcement learning, continual learning, and evolutionary algorithms to further enhance the capabilities of QiMeng. For the Hardware Design Agent, we will explore combining the framework of LPCM-based automated HDL generation with verify-repair-driven feedback and performance-driven feedback, establishing the entire Hardware Design Agent which achieves both correctness and performance optimization. Additionally, we will investigate end-to-end design from functional specifications to transistor-level implementation, breaking through conventional Boolean logic and CMOS paradigms, while scaling up the design to achieve industrial-grade automated processor chip design. For the Software Design Agent, the current implementation primarily exploits the textual comprehension from LLMs. Future enhancements will integrate the graph-structured representation. Moreover, we will extend the agent's applicability to autonomous software migration, functional adaptation, and performance optimization for more foundational software. By realizing and continuously improving the capabilities of QiMeng, we aim to address increasingly diverse scenario demands, driving the entire processor chip domain toward intelligence and automation.

REFERENCES

- [1] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.
- [2] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.
- [3] "Qwen3: Think Deeper, Act Faster." [Online]. Available: <https://qwenlm.github.io/blog/qwen3/>
- [4] OpenAI, "Hello GPT-4o," <https://openai.com/index/hello-gpt-4o/>, 2024.
- [5] "Gemini 2.5: Our most intelligent AI model." [Online]. Available: <https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/#gemini-2-5-thinking>
- [6] Q. Zhang, K. Ding, T. Lv, X. Wang, Q. Yin, Y. Zhang, J. Yu, Y. Wang, X. Li, Z. Xiang *et al.*, "Scientific large language models: A survey on biological & chemical domains," *ACM Computing Surveys*, vol. 57, no. 6, pp. 1–38, 2025.
- [7] K. M. Jablonka, Q. Ai, A. Al-Feghali, S. Badhwar, J. D. Bocarsly, A. M. Bran, S. Bringuier, L. C. Brinson, K. Choudhary, D. Circi *et al.*, "14 examples of how llms can transform materials science and chemistry: a reflection on a large language model hackathon," *Digital discovery*, vol. 2, no. 5, pp. 1233–1250, 2023.
- [8] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024.
- [9] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.
- [10] A. Church, "Application of recursive arithmetic to the problem of circuit synthesis," *Journal of Symbolic Logic*, vol. 28, no. 4, 1963.
- [11] J. von Neumann and A. W. Burks, *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966, scanned book online. [Online]. Available: https://archive.org/details/theoryofselfrepro0vonnn_0
- [12] K. Chang, M. Chen, Y. Chen, Z. Chen, D. Fan, J. Gong, N. Guo, Y. Han, Q. Hao, S. Hou, X. Huang, P. Jin, C. Ke, C. Li, G. Li, H. Li, K. Li, N. Li, S. Liang, C. Liu, H. Liu, J. Liu, J. Lv, J. Mu, J. Qin, B. Sun, C. Wang, D. Wang, M. Wang, Y. Wang, C. Wu, P. Wu, T. Wu, X. Xiao, M. Xie, C. Xiong, R. Xu, M. Yan, X. Ye, K. Yu, R. Zhang, S. Zhang, and J. Zhao, "Large processor chip model," *arXiv preprint arXiv:2505.06302*, 2025.
- [13] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.
- [14] L. Kong, J. Cui, H. Sun, Y. Zhuang, B. A. Prakash, and C. Zhang, "Autoregressive diffusion model for graph generation," in *International conference on machine learning*. PMLR, 2023, pp. 17391–17408.
- [15] Z. Hu, Y. Dong, K. Wang, K.-W. Chang, and Y. Sun, "Gpt-gnn: Generative pre-training of graph neural networks," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 2020, pp. 1857–1867.
- [16] S. Cheng, P. Jin, Q. Guo, Z. Du, R. Zhang, X. Hu, Y. Zhao, Y. Hao, X. Guan, H. Han *et al.*, "Automated cpu design by learning from input-output examples," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, 2024, pp. 3843–3853.
- [17] M. Zhou, X. Hu, and W. Xiong, "openeuler: Advancing a hardware and software application ecosystem," *IEEE Software*, vol. 39, no. 2, pp. 101–105, 2022.
- [18] R. Han, J. Lee, J. Sim, and H. Kim, "Cox: Exposing cuda warp-level functions to cpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 4, pp. 1–25, 2022.
- [19] A. Johnson, C. Coti, A. D. Malony, and J. Doerfert, "Martini: The little match and replace tool for automatic application rewriting with code examples," in *European Conference on Parallel Processing*. Springer, 2022, pp. 19–34.
- [20] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "Efficient compilation of cuda kernels for high-performance computing on fpgas," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, pp. 1–26, 2013.
- [21] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.
- [22] B. Roziere, M.-A. Lachaux, L. Chausson, and G. Lample, "Unsupervised translation of programming languages," *Advances in neural information processing systems*, vol. 33, pp. 20601–20611, 2020.
- [23] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcode: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.
- [24] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.
- [25] J. Bi, Q. Guo, X. Li, Y. Zhao, Y. Wen, Y. Guo, E. Zhou, X. Hu, Z. Du, L. Li *et al.*, "Heron: Automatically constrained high-performance library generation for deep learning accelerators," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 314–328.
- [26] H. Chen, Y. Wen, L. Cheng, S. Kuang, Y. Liu, W. Li, L. Li, R. Zhang, X. Song, W. Li *et al.*, "Autoos: make your os more powerful by exploiting large language models," in *Forty-first International Conference on Machine Learning*, 2024.
- [27] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.
- [28] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
- [29] R. Roy, J. Raiman, N. Kant, I. Elkin, R. Kirby, M. Siu, S. Oberman, S. Godil, and B. Catanzaro, "Prefixrl: Optimization of parallel prefix circuits using deep reinforcement learning," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 853–858.
- [30] P.-W. Chen, Y.-C. Huang, C.-L. Lee, and J.-H. R. Jiang, "Circuit learning for logic regression on high dimensional boolean space," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.
- [31] S. Rai, W. L. Neto, Y. Miyasaka, X. Zhang, M. Yu, Q. Yi, M. Fujita, G. B. Manske, M. F. Pontes, L. S. Da Rosa *et al.*, "Logic synthesis meets machine learning: Trading exactness for generalization," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1026–1031.

- [32] H. Wu, Z. He, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "Chateda: A large language model powered autonomous agent for eda," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.
- [33] S. Cheng, R. Zhang, W. He, P. Jin, C. Li, Z. Du, X. Hu, Y. Hao, G. Xu, Y. Wen, L. Li, Q. Guo, and Y. Chen, "Qimeng-cpu-v2: Automated superscalar processor design by learning data dependencies," *arXiv preprint arXiv:2505.03195*, 2025.
- [34] S. Cheng, C. Li, Z. Du, R. Zhang, X. Hu, X. Li, G. Xu, Y. Wen, and Q. Guo, "Revisiting automatic pipelining: Gate-level forwarding and speculation," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.
- [35] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.
- [36] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.
- [37] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming—the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.
- [38] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang *et al.*, "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.
- [39] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2.5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.
- [40] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.
- [41] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," *arXiv preprint arXiv:2312.08617*, 2023.
- [42] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, "Bettver: Controlled verilog generation with discriminative guidance," *arXiv preprint arXiv:2402.03375*, 2024.
- [43] M. Liu, Y.-D. Tsai, W. Zhou, and H. Ren, "Craftrtl: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair," *arXiv preprint arXiv:2409.12993*, 2024.
- [44] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.
- [45] Q. Team, "Qwq-32b: Embracing the power of reinforcement learning," March 2025. [Online]. Available: <https://qwenlm.github.io/blog/qwq-32b/>
- [46] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," *arXiv preprint arXiv:2302.06590*, 2023.
- [47] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.
- [48] Y. Zhao, D. Huang, C. Li, P. Jin, Z. Nan, T. Ma, L. Qi, Y. Pan, Z. Zhang, R. Zhang *et al.*, "Codev: Empowering llms for verilog generation through multi-level summarization," *arXiv preprint arXiv:2407.10424*, 2024.
- [49] 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.
- [50] Q. Yu, Z. Zhang, R. Zhu, Y. Yuan, X. Zuo, Y. Yue, T. Fan, G. Liu, L. Liu, X. Liu *et al.*, "Dapo: An open-source llm reinforcement learning system at scale," *arXiv preprint arXiv:2503.14476*, 2025.
- [51] Y. Zhu, D. Huang, H. Lyu, X. Zhang, C. Li *et al.*, "Codev-r1: Reasoning-enhanced verilog generation," *arXiv preprint arXiv:2505.2418*, 2025.
- [52] J. Oh, N. F. Yildiran, J. Braha, and P. Gazzillo, "Finding broken linux configuration specifications by statically analyzing the kconfig language," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 893–905.
- [53] P. Franz, T. Berger, I. Fayaz, S. Nadi, and E. Groshev, "Configfix: Interactive configuration conflict resolution for the linux kernel," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 91–100.
- [54] Y. Xia, Z. Ding, and W. Shang, "Comsa: A modeling-driven sampling approach for configuration performance testing," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1352–1363.
- [55] S. Dong, Y. Wen, J. Bi, D. Huang, J. Guo, J. Xu, R. Xu, X. Song, Y. Hao, X. Zhou, T. Chen, Q. Guo, and Y. Chen, "Qimeng-xpiler: Transcompiling tensor programs for deep learning systems with a neural-symbolic approach," *arXiv preprint arXiv:2505.02146*, 2025.
- [56] M. Zhong, F. Lyu, L. Wang, H. Geng, L. Qiu, H. Cui, and X. Feng, "Comback: A versatile dataset for enhancing compiler backend development efficiency," in *Thirty-eighth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.
- [57] M. Zhong, F. Lv, L. Wang, L. Qiu, Y. Wang, Y. Liu, H. Cui, X. Feng, and J. Xue, "Vega: Automatically generating compiler backends using a pre-trained transformer model," in *2025 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2025.
- [58] S. Zhang, J. Zhao, C. Xia, Z. Wang, Y. Chen, and H. Cui, "Introducing compiler semantics into large language models as programming language translators: A case study of c to x86 assembly," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024, pp. 996–1011.
- [59] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. d. S. Magalhães, and M. F. O'Boyle, "Exebench: an ml-scale dataset of executable c functions," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 50–59.
- [60] "ANSI C (and mostly compatible) Benchmarks for Unix and Unix-like systems." [Online]. Available: <https://github.com/nfinit/ansibench>
- [61] "CoreMark - CPU Benchmark." [Online]. Available: <https://www.eembc.org/coremark/>
- [62] "Basic Linear Algebra on NVIDIA GPUs." [Online]. Available: <https://developer.nvidia.com/cublas>
- [63] "NVIDIA cuDNN." [Online]. Available: <https://developer.nvidia.com/cudnn>
- [64] "NVIDIA TensorRT." [Online]. Available: <https://developer.nvidia.com/tensorrt>
- [65] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in neural information processing systems*, vol. 35, pp. 16 344–16 359, 2022.
- [66] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," *arXiv preprint arXiv:2307.08691*, 2023.
- [67] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," *Advances in Neural Information Processing Systems*, vol. 37, pp. 68 658–68 685, 2024.
- [68] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.
- [69] S. L. Jia Shi Li, "Flashmla: Efficient mla decoding kernels," 2025. [Online]. Available: <https://github.com/deepseek-ai/FlashMLA>
- [70] C. Zhao, L. Zhao, J. Li, and Z. Xu, "Deepgemm: clean and efficient fp8 gemm kernels with fine-grained scaling," 2025. [Online]. Available: <https://github.com/deepseek-ai/DeepGEMM>
- [71] "NVIDIA Data Center GPUs." [Online]. Available: <https://www.nvidia.com/en-us/data-center/data-center-gpus/>
- [72] "Cambricon MLU." [Online]. Available: <https://www.cambricon.com/>
- [73] "AMD Instinct Accelerators." [Online]. Available: <https://www.amd.com/en/products/accelerators/instinct.html>
- [74] "Intel AI Engines Simplify and Accelerate AI." [Online]. Available: <https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/ai-engines.html>
- [75] Anthropic, "Introducing Claude 3.5 Sonnet," <https://www.anthropic.com/news/claude-3-5-sonnet>, 2024.
- [76] OpenMathLib, "Openblas," <https://github.com/OpenMathLib/OpenBLAS>, 2024.
- [77] Q. Zhou, Y. Wen, R. Chen, K. Gao, W. Xiong, L. Li, Q. Guo, Y. Wu, and Y. Chen, "Qimeng-gemm: Automatically generating high-performance matrix multiplication code by exploiting large language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 21, 2025, pp. 22 982–22 990.
- [78] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

- [79] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.
- [80] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.
- [81] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, and Y. Ding, "Egemtc: accelerating scientific computing on tensor cores with extended precision," in *Proceedings of the 26th ACM SIGPLAN symposium on principles and practice of parallel programming*, 2021, pp. 278–291.
- [82] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *Advances in neural information processing systems*, vol. 36, pp. 11 809–11 822, 2023.
- [83] X. Zhang, S. Peng, Q. Zhou, Y. Wen, Q. Guo, R. Chen, X. Zhu, W. Xiong, H. Chen, C. Ma, K. Gao, C. Zhao, Y. Wu, Y. Chen, and L. Li, "Qimeng-tensorop: Automatically generating high-performance tensor operators with hardware primitives," *arXiv preprint arXiv:2505.06302*, 2025.
- [84] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen *et al.*, "Xuante-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)*. IEEE, 2020, pp. 52–64.
- [85] "SPACEMIT MUSEBook." [Online]. Available: <https://www.spacemit.com/spacemit-muse/>
- [86] "Second-Generation, High-Performance CPU Based on DynamIQ Technology." [Online]. Available: <https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a76>
- [87] NVIDIA, "Nvidia geforce rtx 4070 family graphics cards." [Online]. Available: <https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4070-family>
- [88] —, "Quadro legacy graphics cards, workstations, and laptops." [Online]. Available: <https://www.nvidia.com/en-us/design-visualization/quadro>
- [89] —, "Nvidia t4 gpu tensor core gpu for ai inference." [Online]. Available: <https://www.nvidia.com/en-us/data-center/tesla-t4>
- [90] "NVIDIA A100 Tensor Core GPU," n.d., accessed: 2024-06-14. [Online]. Available: <https://www.nvidia.com/en-us/data-center/a100/>
- [91] J. Dong, B. Feng, D. Guessous, Y. Liang, and H. He, "Flex attention: A programming model for generating optimized attention kernels," *arXiv preprint arXiv:2412.05496*, 2024.
- [92] Y. Chen, Z. Du, Q. Guo, W. Li, and Y. Tan, "From chip design to chip learning," *Bulletin of Chinese Academy of Sciences (Chinese Version)*, vol. 37, no. 1, pp. 15–23, 2022.
- [93] W. He, X. Li, X. Song, Y. Hao, R. Zhang, Z. Du, and Y. Chen, "Chip design with machine learning: A survey from algorithm perspective," *Science China Information Sciences*, vol. 66, no. 11, pp. 1–31, 2023.
- [94] E. Nurvitadhi, J. C. Hoe, T. Kam, S.-L. L. Lu *et al.*, "Automatic pipelining from transactional datapath specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 3, pp. 441–454, 2011.
- [95] G. Liu, J. Primmer, and Z. Zhang, "Rapid generation of high-quality risc-v processors from functional instruction set specifications," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.
- [96] S. F. Hoover, "Timing-abstract circuit design in transaction-level verilog," in *2017 IEEE International Conference on Computer Design (ICCD)*, 2017, pp. 525–532.
- [97] R. S. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Proceedings Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04*, 2004, pp. 69–70.
- [98] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The essence of bluespec: a core language for rule-based hardware design," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 243–257.
- [99] S. Rokicki, D. Pala, J. Paturel, and O. Sentieys, "What you simulate is what you synthesize: Designing a processor core from c++ specifications," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.
- [100] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 127–136.
- [101] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, "Predictable accelerator design with time-sensitive affine types," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 393–407.
- [102] J. Zhao, T. Liang, S. Sinha, and W. Zhang, "Machine learning based routing congestion prediction in FPGA high-level synthesis," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, 2019, pp. 1130–1135.
- [103] H. M. Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. P. Dinakarrao, H. Homayoun, and S. Rafatirad, "Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2019, pp. 397–403.
- [104] M. Ferianc, H. Fan, R. S. Chu, J. Stano, and W. Luk, "Improving performance estimation for FPGA-based accelerators for convolutional neural networks," in *Proceedings of the Applied Reconfigurable Computing*, 2020, pp. 3–13.
- [105] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E. Gaillardon, "LSOracle: A logic synthesis framework driven by artificial intelligence," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2019, pp. 1–6.
- [106] W. Haaswijk, E. Collins, B. Seguin, M. Soeken, F. Kaplan, S. Süstrunk, and G. De Micheli, "Deep learning for logic optimization algorithms," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2018, pp. 1–4.
- [107] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, "Exploring logic optimizations with reinforcement learning and graph convolutional network," in *Proceedings of the ACM/IEEE Workshop on Machine Learning for CAD*, 2020, pp. 145–150.
- [108] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, "DRiLLS: Deep reinforcement learning for logic synthesis," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2020, pp. 581–586.
- [109] G. Pasandi, M. Peterson, M. Herrera, S. Nazarian, and M. Pedram, "Deep-PowerX: A deep learning-based framework for low-power approximate logic synthesis," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 73–78.
- [110] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae *et al.*, "Chip placement with deep reinforcement learning," *arXiv preprint arXiv:2004.10746*, 2020.
- [111] Y.-C. Lu, J. Lee, A. Agnesina, K. Samadi, and S. K. Lim, "GAN-CTS: A generative adversarial framework for clock tree prediction and optimization," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2019, pp. 1–8.
- [112] S. Nagaria and S. Deb, "Designing of an optimization technique for the prediction of CTS outcomes using neural network," in *Proceedings of the IEEE International Symposium on Smart Electronic Systems*. IEEE, 2020, pp. 312–315.
- [113] Y. Kwon, J. Jung, I. Han, and Y. Shin, "Transient clock power estimation of pre-CTS netlist," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2018, pp. 1–4.
- [114] Y. He and F. S. Bao, "Circuit routing using monte carlo tree search and deep neural networks," *arXiv preprint arXiv:2006.13607*, 2020.
- [115] J. H. Liang, H. G. VK, P. Poupard, K. Czarnecki, and V. Ganesh, "An empirical study of branching heuristics through the lens of global learning rate," in *Proceedings of the Theory and Applications of Satisfiability Testing*. Springer, 2017, pp. 119–135.
- [116] M. B. Alawieh, W. Li, Y. Lin, L. Singhal, M. A. Iyer, and D. Z. Pan, "High-definition routing congestion prediction for large-scale FPGAs," in *Proceedings of the Asia and South Pacific Design Automation Conference*. IEEE, 2020, pp. 26–31.
- [117] H. Pearce, B. Tan, and R. Karri, "Dave: Deriving automatically verilog from english," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, 2020, pp. 27–32.
- [118] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [119] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 1–31, 2024.
- [120] L. Chen, Y. Chen, Z. Chu, W. Fang, T.-Y. Ho, R. Huang, Y. Huang, S. Khan, M. Li, X. Li *et al.*, "The dawn of ai-native eda: Op-

- portunities and challenges of large circuit models,” *arXiv preprint arXiv:2403.07257*, 2024.
- [121] K. Chang, Y. Wang, H. Ren, M. Wang, S. Liang, Y. Han, H. Li, and X. Li, “Chipppt: How far are we from natural language hardware design,” 2023.
 - [122] 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.
 - [123] J. Blocklove, S. Garg, R. Karri, and H. Pearce, “Chip-chat: Challenges and opportunities in conversational hardware design,” in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2023, pp. 1–6.
 - [124] Y. Tsai, M. Liu, and H. Ren, “Rtlfixer: Automatically fixing rtl syntax errors with large language model,” in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.
 - [125] “HIPIFY.” [Online]. Available: <https://github.com/ROCm/HIPIFY>
 - [126] “C to Go translator.” [Online]. Available: <https://github.com/gotranspile/cxgo>
 - [127] “C2Rust.” [Online]. Available: <https://github.com/immunant/c2rust>
 - [128] P. M. Phothisilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik, “Chlorophyll: Synthesis-aided compiler for low-power spatial architectures,” *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 396–407, 2014.
 - [129] “Ga144 144-computer chip.” [Online]. Available: <https://www.greenarraychips.com/home/documents/greg/GA144.htm>
 - [130] J. Woodruff, J. Armengol-Estapé, S. Ainsworth, and M. F. O’Boyle, “Bind the gap: Compiling real software to hardware fft accelerators,” in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 687–702.
 - [131] J. Armengol-Estapé and M. F. O’Boyle, “Learning c to x86 translation: An experiment in neural compilation,” *arXiv preprint arXiv:2108.07639*, 2021.
 - [132] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” *arXiv preprint arXiv:2102.04664*, 2021.
 - [133] Y. Wen, Q. Guo, Q. Fu, X. Li, J. Xu, Y. Tang, Y. Zhao, X. Hu, Z. Du, L. Li *et al.*, “Babeltower: Learning to auto-parallelized program translation,” in *International Conference on Machine Learning*. PMLR, 2022, pp. 23 685–23 700.
 - [134] H. Menon, A. Bhatele, and T. Gamblin, “Auto-tuning parameter choices in hpc applications using bayesian optimization,” in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 831–840.
 - [135] A. Jung, H. Lefevre, C. Rotsos, P. Olivier, D. Oñoro-Rubio, F. Huici, and M. Niepert, “Wayfinder: Towards automatically deriving optimal os configurations,” in *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2021, pp. 115–122.
 - [136] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, “Learning to optimize tensor programs,” *Advances in Neural Information Processing Systems*, vol. 31, 2018.
 - [137] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.
 - [138] P. A. Vikhar, “Evolutionary algorithms: A critical review and its future prospects,” in *2016 International conference on global trends in signal processing, information computing and communication (ICGTSPICC)*. IEEE, 2016, pp. 261–265.
 - [139] Y. Zhai, S. Yang, K. Pan, R. Zhang, S. Liu, C. Liu, Z. Ye, J. Ji, J. Zhao, Y. Zhang *et al.*, “Enabling tensor language model to assist in generating high-performance tensor programs for deep learning,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 289–305.