# ChipGPT: Enabling Template-Free Hardware Design Space Exploration via LLM-Driven Chip Discovery Search

Anonymous Authors

***Abstract*—Recently, using design space exploration to design a better chip draw much attention in EDA community. However, design space exploration requires a profound knowledge to construct a template, which limits the design space. Inspired by DeepMind's Funsearch published in Nature, we leverage the professional knowledge feature of LLM to search for a new hardware design, which does not need a template. Our method is a LLM integrated search method, which leverages existing RTL designs to generate a new RTL design or extract a rule to optimize hardware designs. This process reinforces the knowledge of LLM via the feedback from EDA tool. It supports loop level, RTL level and gate-level module generation, which greatly enlarges the design space by enabling a template-free design space exploration. We evaluate this method using a wide range of LLM. The results show that this method can search for better results. It reveals that our infinite knowledge reinforcement learning can generate previously undefined hardware, which enables template-free design space exploration.**

***Index Terms*—Agile hardware development, Natural language programming, Program synthesis**

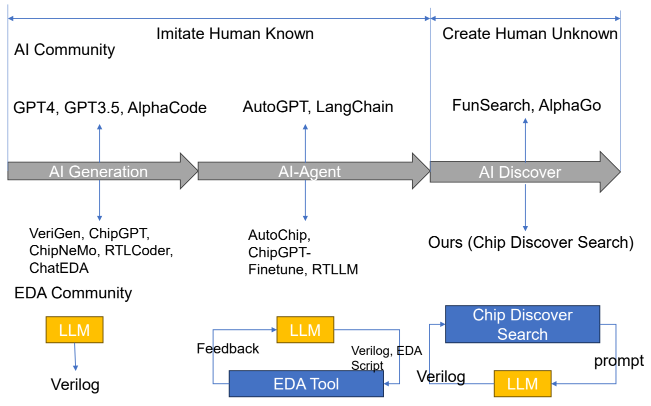

Fig. 1. Overview of Chip Discover Search.

## I. Introduction

Recently, using large language model to generate verilog obtain much attention from EDA community[1–6]. The large language model accepts natural language and outputs verilog module. This paradigm can facilitate fast hardware design via an LLM editor support. Therefore, researchers propose some methods to enhance generated Verilog quality. For example, finetuning the general large language model(*e.g.* GPT, Llama) can make them more adaptable to hardware design[1, 7, 8].

However, when considering design space exploration for chip design, a significant challenge lies in imitating human scientists, especially chip design scientists, to discover new hardware architectures. This task cannot be accomplished solely by Large Language Models (LLMs) because they lack a search function like human; they serve as generation tools. Regardless of the specific LLM used, the generated Verilog or Electronic Design Automation (EDA) scripts must undergo review and approval by human programmers. This underscores the necessity for the final program to adhere to human programming conventions, prompting the question: Why can't LLMs independently generate hardware without human? Is the ultimate goal of hardware Large Language Models restricted to solely generating Verilog or EDA scripts from natural language as auxiliary tools for human programmers? This paper reveals that LLMs can not only complete existing Verilog programs but also discover previously unknown hardware designs.

A key observation for human unknown hardware design consists of two meanings. The first meaning is the programmer's individual unknown. The second meaning is human unknown, even when the world has no solution. Regardless of interpretation, current LLMs have exploitation ability but lack exploration ability to create the optimized program; they do not learn. Inspired by DeepMind's recently published Nature article 'Funsearch' [9], which uses LLM+Search to generate current human unknown programs and discover new solutions to traditional questions, we find two ways to empower LLMs for exploration. First, by imitating humans, LLMs can leverage the advantages of their two known programs and generate an optimized one. Second, inspired by compiler rewrite rules, they have the meta-learning ability to compare two programs and give the rewrite rule, which can then be applied to other programs to generate new optimized programs. Based on these observations, we propose Chip Discovery Search, which iteratively uses these two methods with an optimal selection stage to create human unknown but better hardware designs using Verilog.

- We reveal that Verilog generation Large Language Models (LLMs) can not only generate previously existing hardware design solutions but also generate previously human unknown but better solutions.
- We propose an LLM + Chip Discovery Search paradigm to enable LLMs to analyze current existing solutions, observe common patterns, and optimize a better one.
- We conduct case studies on this novel method, which yields better solutions for the 4-bit adder module and 8-bit adder module.

## II. Background and Motivation

### A. AGI roadmap using Large Language Model

Using Large Language Models (LLMs) to discover new natural phenomena or special structures is a promising field. As depicted in Fig. 1, the first era in the Artificial General Intelligence (AGI) roadmap is to imitate human current knowledge, which comprises the AI generation stage and the AI agent stage. For instance, models like GPT-4 and GPT-3.5 are trained to answer people's questions, whereas ChipGPT utilizes them to generate Verilog code. In the AI agent era, efforts are made to employ LLMs' self-planning methods to decompose the entire task into several subtasks for tool manipulation. This stage enables LLMs to generate EDA scripts, facilitating easy manipulation of EDA tools. For instance, LLMs can decompose tasks, plan how to accomplish them, and receive feedback from EDA tools. In the AI discovery stage, pretrained models can be integrated with

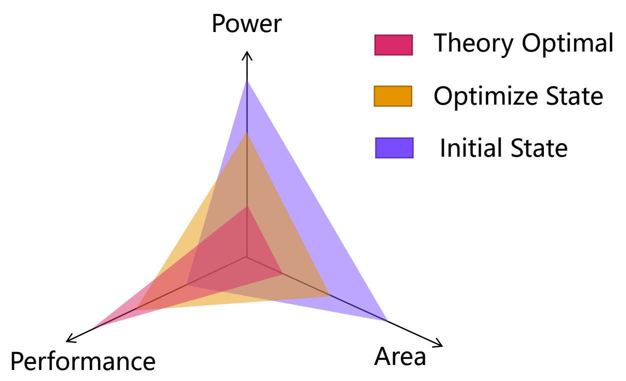

Fig. 2. ChipGPT design space exploration

a search process to enhance their exploration skills. For example, DeepMind attempts to extract new program solutions to old questions using PaLM (e.g., Funsearch).

The AlphaGeometry[10], AlphaStar[11], and Funsearch are representatives using reinforcement learning and prompt engineering. These methods are essential in the discovery process for feedback from external tools. In the reinforcement learning method, RL utilizes observations from SAT solvers and adds new premises that may lead to a more correct program. RL algorithms often require an imitation learning stage to learn from predefined offline expert data. This is because without this offline training stage, RL models cannot converge to a stable state. For the Funsearch method, this stage requires LLM to have composition ability to extract useful information from two programs and then generate a new program from them. This method maintains a program pool to select a better program from this pool. However, chip design follows different rules compared with the general program generation field, as it has PPA metrics rather than only latency metrics. This key observation leads us to propose chip discovery search.

## B. Hardware Large Language Model Search

There is a sequence of hardware generation LLM development stages in the EDA community. First, designing a benchmark to evaluate the weaknesses of general LLMs such as GPT-4 [4, 12]. Second, fine-tuning open-source general LLMs for the Verilog generation field [7, 8, 13]. Third, generalizing the LLMs to other hardware design problems such as code fixing and quantum circuit design [14–16]. Fourth, using LLMs as agents to further enhance their abilities with self-planning [4]. The search-based method has the potential to enhance LLMs' exploration abilities. In ChipGPT [3, 17], the search uses a large language model to generate a pool of Verilog programs. Then, the search process selects the best program with the given target from the pool. However, these approaches all aim to enhance LLM abilities in one aspect of the chip design field. They cannot discover new hardware architectures.

## C. Motivation

Emerging LLM-driven EDA workflows focus on EDA front-ends with higher-level representations. Large language models (LLMs) enable natural language hardware descriptions, covering expressiveness. However, previous LLM-generated hardware researches [6, 18] lack of figuring the following challenges:

*a) Challenge: LLMs are unaware of power, area and performance (PPA), unable to generate ideal programs (red, Fig. 2). :* As shown in Fig. 2, the ideal program PPA results are in the red area, but LLM-generated programs remain in the initial state (PPA-agnostic) due to their training. LLMs like InstructGPT and ChatGPT are trained using general reinforcement learning with a generic reward function to improve performance. This results in programs that are intuitively good but not optimized for PPA. Therefore, we propose an chip discovery search to tackle this challenge and promote the PPA to the yellow area in Fig. 2.

## III. AUTOMATIC CHIP GENERATION FRAMEWORK

ChipGPT serves as an EDA frontend framework, which aims to assist humans in compiling chip specifications to logic design. It takes chip specification as input and generates target hardware module description (*i.e.* Verilog program).

$$HDL = ChipGPT(specification)$$

The chip specification defines the target of HDL implementation. Thus HDL should be controlled by chip specification. Due to the GPT-produced program not always being accurate as well as challenging to analyze and modify, ChipGPT does not attempt to amend the generated raw program directly. Rather, it gives GPT several times of trials to generate different code versions and refine them to obtain the final output. Specification Split and Prompt Manager implicitly control the output. Output Manager and Chip Discovery Search explicitly control the output.

At each iteration, ChipGPT follows these steps.

$$split(spec) = \{eg, iface, func, compose\}$$

$$rawcode = GPT(PM(iface, func, compose, fd))$$

$$code = OM(rawcode, EM)$$

$$codelist = codelist \cup code$$

$$optcode = Search(codelist, requirements)$$

First, to translate the specifications into a formal and unambiguous form, we propose a specification split method $split()$.

Second, to make these irregular split partitions become an available GPT input sequence, we propose a specification serialization method in $PM$.

Third, to improve the quality of the $code$(raw program) generated by the GPT model, we propose an output manager $OM$.

Finally, the code generated in the third stage typically does not meet performance, power and area (PPA) requirements. To obtain an optimized final output program for PPA, we propose a search method compatible with the previous stages.

## IV. THE PROPOSED SEARCH METHOD

Search functionality extends beyond the exploration of known design spaces (DSE) and can be employed to uncover and construct previously unknown knowledge.

### A. Overview

Chip discovery search tries to maximize likelihood to obtain a user satisfied verilog program. It obtains from Equ. 1, where $HDL$ denotes verilog program and $S$ denotes chip specification. Likelyhood function $Score$ is a user custimized function to evaluate the design from area, performance or other perspective.

$$V_{max} = \arg\max Score(HDL; S) \quad (1)$$

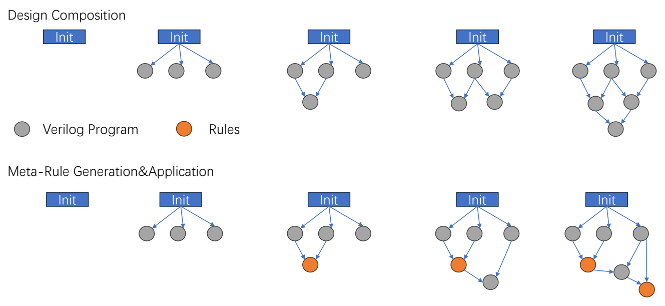

Fig. 3. A case of chip discovery search.

*1) Specification:* The specification consists of two parts: the testbench and the description. The testbench is utilized by the score evaluator to assess the generated Verilog, while the description serves as the initial input for the LLM.

*2) Score Evaluation:* Similar to RLHF, score evaluation comprises both human evaluation and testbench evaluation. If the human evaluation does not pass, indicating that the generated code lacks readability, the score is set to 0. However, if the human evaluation is passed, the final score is determined by the testbench evaluation using EDA tools, representing the design's PPA. Users select a target metric (e.g., latency, area, or power), and the score indicates the improvement over the original human solution.

*3) HDL Database:* As depicted in Lines 7-10 of Algorithm 1, the HDL pool $db$ represents a set of Verilog programs, denoted as $p_0, p_1, \ldots, p_n$. Initializing an HDL database requires the first prompt to describe the module's function. Various solutions rely on the LLM's regenerate function, which utilizes a different sentence instead of the current ones. Subsequently, the solutions depend on the program completion function of each module to finalize some partially generated programs. The newly generated program is then added to $db$ in Line 9.

### *B. Chip Discovery Search Algorithm*

This section introduces the fundamental aspect of Chip Discovery Search, outlining the methodology for uncovering previously unknown hardware designs.

*1) Implicit Discovery by Design composition:* For human programmers, they compare different design solutions with different disadvantages and then propose a new one with all of the advantages of these solutions. We believe AI has the same functions like this, therefore we propose design composition to expand search space.

To elucidate the process of generating new programs, as depicted in Fig. 3, Chip Discovery Search models an HDL search tree, where each node represents a HDL program. In this framework, each new program is synthesized through LLM-composition of two other programs. This LLM-composition process resembles the methodology employed by Funsearch in a recent publication in Nature. Specifically, it involves composing two program candidates into one prompt and leveraging the LLM to amalgamate the strengths of these two programs to generate a new one. For instance, as illustrated in Equation 2, HDL programs $p_i$ and $p_j$ are combined into a single prompt, after which the LLM synthesizes a new program $p_n$ by integrating the advantages of these two programs. The input numbers are limited to two because most LLMs struggle to effectively capture information from more than two programs simultaneously.

$$\{p_i, p_j\} \longrightarrow_{LLM} p_n \tag{2}$$

*2) Explicit Discovery by Meta-rule Generation & Application:* For human programmers, optimizing a program with a compiler typically involves first identifying a better program and then comparing it with the original program, as illustrated in the second row of Fig. 3. During this process, a rewrite rule $P \longrightarrow A$ can be derived, where every Verilog program matching the pattern $P$ can be transformed using action $A$ to achieve a superior program. Drawing inspiration from this methodology, we propose a meta-learning approach. Here, the LLM analyzes the differences between the optimal Verilog and the original Verilog and summarizes a rule in context. Subsequently, it applies this rule to other Verilog programs to obtain improved solutions.

Equ. 3, 4 are the formulation of meta-rule generation. Equ. 3 shows a rule can be obtain from comparing two programs. Equ. 4 show that an optimized program can be inference by applying a rule to a program.

$$\{p, p_{opt}\} \longrightarrow_{LLM} r_n \tag{3}$$

$$\{r_i, p_i\} \longrightarrow_{LLM} p_j \tag{4}$$

Design composition and meta-rule generation can be unified into one algorithm. There are three steps iterately executing to generate rules and augments programs as shown in Line 12-42 in Alg. 1. First, this algorithm generates new program using design composition stage. Second, the algorithm generates meta-rules and applies meta-rules to obtain a new program.

*3) Case Study:* To elucidate the composition rule, we conducted a case study on a 4-bit adder. In this example, we initialized two 4-bit adders in the logic level, as depicted in Fig. 4. The improved version reorganizes the AND and OR gates into two sets, enhancing the structure and readability of the code. The composed version retains the efficiency of the initial solution while presenting the logic in a more modular and readable format.

We utilized Yosys for logic synthesis of the RTL code to AIG. Base Version 0 comprised 49 logic gates, while Base Version 1 and the composed version each had 32 logic gates. This result demonstrates that the composition rule can refine the code while maintaining optimization.

### *C. Select Previous Human Unknown Hardware*

After the LLM generates solutions, Chip Discovery Search employs an evaluator to map the Verilog solution to a score. This score serves as an indicator to guide the search for a better solution. For example, in the proposed algorithm, Line 16 invokes the $evaluator()$ function to obtain the score from $p$. The evaluation function should reflect the errors and potential problems in the provided program. This can be defined as follows:

- Syntax error: The error number from EDA tools(yosys checker), which can not pass verilog syntax checker.
- Semantic error: The error number from EDA Tools(verilator), which can not pass testbench.
- Syntax& Semantic right but PPA worse: Can pass testbench but worse than human solution. The indicator number is the PPA target.
- Syntax& Semantic right but PPA better: Can pass testbench and better than human solution. The indicator number is the PPA target.

As per Section IV-A2, both syntax errors and semantic errors result in a score of zero. The evaluator selects the best score available.

Base Version 0

```
module FourBitAdder (
  input [3:0] A
  input [3:0] B,
input Cin,
  output [3:0] Sum,
  output Cout
);

  …

  // XOR gates for sum bits
  assign Xor_out = A ^ B;

  // AND gates for carry
generation
  assign And_out = A & B;

  // XOR gates for carry out
  assign Xor_Cout = Xor_out
^ Cin;

  // Sum bits
  assign Sum = Xor_out ^ Cin;

  // Carry out
  assign Cout = (And_out |
(Xor_out & Cin)) | Xor_Cout;

endmodule
```

Base Version 1

```
module FourBitAdder (
  input [3:0] A,
  input [3:0] B,
  input Cin,
  output [3:0] Sum,
  output Cout
);

  …

  // XOR gates for each bit
  assign XorOut = A ^ B;

  // AND gates for each bit
  assign AndOut1 = A & B;
  assign AndOut2 = A & XorOut;
  assign AndOut3 = B & XorOut;

  // OR gates for each bit
  assign OrOut1 = AndOut1 | AndOut2;
  assign OrOut2 = OrOut1 | AndOut3;

  // Output sum for each bit
  assign Sum = XorOut;

  // Output carry-out
  assign Cout = OrOut2 | (Cin & OrOut1);

endmodule
```

Compose Version

```
module BetterFourBitAdder (
  input [3:0] A,      // 4-bit input A
  input [3:0] B,      // 4-bit input B
  input Cin,          // Carry input
  output [3:0] Sum,    // 4-bit sum output
  output Cout          // Carry out
);

  …

  // XOR gates for sum bits
  assign XorOut = A ^ B;

  // First set of AND gates for carry generation
  assign AndOut1 = A & B;

  // Second set of AND gates for carry generation
  assign AndOut2 = A & XorOut;

  // First set of OR gates for intermediate carry
  assign OrOut1 = AndOut1 | AndOut2;

  // Second set of OR gates for final carry-out
  assign OrOut2 = OrOut1 | (B & XorOut);

  // Output sum bits
  assign Sum = XorOut;

  // Output carry-out
  assign Cout = OrOut2 | (Cin & OrOut1);

endmodule
```

Fig. 4. Case Study of composition rule on LLM generated 4bit adder. Base Version 0 & 1 are the initialized version while compose version on the right is the derivative version using composition rule.

## V. Algorithm Theory of Chip Discovery Search

*a) Soundness:* In Chip Discovery Search, soundness refers to the assurance that all Verilog generated by the search process is correct. Since LLM-generated Verilog programs may contain errors due to the probabilistic nature of LLMs, the evaluator addresses this issue by verifying the correctness of the generated programs and assigning a final score accordingly. If a program is found to be incorrect, its score is set to zero, indicating that it is not included in the final output. This mechanism is represented as a Horn logic in Equ. 5. As a result, all incorrect programs are excluded from the output set.

$$Score(p) = 0 \implies sol \neq p \tag{5}$$

*b) Completeness:* Completeness in the context of search space exploration typically refers to the ability to explore all possible solutions within the defined space. However, LLMs may struggle to explore programs they are not familiar with. To mimic the human discovery process, we introduce the concept of "evolutionary completeness" to describe the completeness of a discovery search. In evolutionary completeness, the method should be able to find a program with a clear evolutionary trace, utilizing the original set $P$ and any human known rules (implicit or explicit) applied to this set. Evolutionary completeness is weaker than completeness in search, as it focuses on tracing the evolutionary path of discovered solutions rather than exhaustively exploring all possible solutions. Equ. 6 provides a formalized representation of evolutionary completeness, where $i \in 0, \ldots, |P| - 1$.

$$\frac{P[i] \hookrightarrow_{rule_1} program, program \hookrightarrow_{rule_2} V}{P \vdash V} \tag{6}$$

To prove evolutionary completeness, we employ the method of reductio ad absurdum. We assume that a human unknown Verilog program $V$ (which can be constructed using human known rules) cannot be represented in the HDL database. Next, we utilize an LLM capable of applying human known rules. We iteratively leverage this implicit information to apply on previously known programs. Through this process, we are able to generate the previously unknown program $V$. If $V$ cannot be constructed using human known rules, it contradicts our initial assumption. Therefore, we can conclude that Chip Discovery Search achieves evolutionary completeness.

## VI. Evaluation

### A. Experimental Setup

*1) Experimental Environment:* We use the ChatGPT model from OpenAI's website. We use Design Compiler and 65nm technology to evaluate the power and area of output designs. The performance cycle numbers of the designs produced through different approaches are obtained by simulation with common handwritten testbeds. For line of code measurement, the cloc tool is used.

*2) Programbility measurement:* Traditional programmability evaluation in EDA uses lines of code (LOC) as the measurement metric[19]. This LOC subtraction method cannot directly measure natural language code generation for two reasons: 1) The LOC of natural language is undefined. 2) Natural language-based zero-code programming frameworks use neural networks, which cannot ensure output programs strictly follow specifications. To address these challenges, we define LOC subtraction as Equ. 7 to evaluate GPT code generation under the prompt manager:

$$quality = raw + correct - prompt \tag{7}$$

**Algorithm 1** Chip Discovery Search

```
Input: $spec$, $llm$, $score_{human}$, $sol_{human}$ and $evaluator$;
Output: HDL file $.v$;
 1: // Note: every llm calling has prompt
 2: $db = \{sol_{human}\}$; // Verilog database
 3: $score = score_{human}$;
 4: $sol = sol_{human}$;
 5: $rules = \{\}$; // rule list
 6: // initialize HDL database
 7: for all $i = 1, 2, \cdots, InitNum$ do
 8:     $p_{tmp} = llm(spec + \theta)$;
 9:     $db = db + p_{tmp}$;
10: end for
11: // Iteratively generate programs and rules
12: for all $i = 1, 2, \cdots, IterNum$ do
13:     // verilog design composition
14:     $p^0_{tmp}, p^1_{tmp} = RandomSelect(db)$;
15:     $p = llm(p^0_{tmp}, p^1_{tmp})$;
16:     $score_{new} = evaluator(p)$; // verilog design selection
17:     if $score_{new} > score$ then
18:         $score = score_{new}$;
19:         $sol = p$;
20:     else if $p \notin db$ then
21:         $db = db + p$;
22:     end if
23:     // meta-rule generation
24:     for all $i = 0, 1$ do
25:         $r = llm(p^i_{tmp}, sol)$;
26:         if $r \notin rules$ then
27:             $rules = rules + r$;
28:         end if
29:     end for
30:     // meta-rule application
31:     for all $i = 0, 1$ do
32:         $r = RandomSelect(rules)$;
33:         $p^i_r = apply(r, p^i_{tmp})$;
34:         $score_{new} = evaluator(p^i_r)$;
35:         if $score_{new} > score$ then
36:             $score = score_{new}$;
37:             $sol = p^i_r$;
38:         else if $p^i_r \notin db$ then
39:             $db = db + p^i_r$;
40:         end if
41:     end for
42: end for
43: return $sol$.
```

Where $quality$ represents code generation algorithm efficiency, $raw$ is the number of lines in the generated program, $correct$ is the number of lines modified by humans to correct the raw program, and $prompt$ is the number of prompt queries.

*3) Benchmarks:* The benchmarks include several typical hardware structures and algorithm accelerator implementations as shown in Table I. 1) Composition (CM): The most complex architecture, requiring module composition. 2) Complex single module (CSM): More complex individual modules. 3) Simple single module (SSM): The simplest modules, with identical program lists and PPA.

*4) Baseline:* To validate our workflow works with ChatGPT, we compare it to the baseline ChatGPT model. The naive code ChatGPT generates uses only the module description as the prompt. To analyze improvement over traditional agile workflows, we compare with Chisel and high-level synthesis (Xilinx Vivado HLS).

TABLE I
WORKLOAD DETAIL

| Workload | Type | Brief Illustration |
|---|---|---|
| matmul | CSM | Multiplying two matrices 4x4 with 16 bits |
| mux | SSM | A 4x1 Multiplexer |
| 3-8decoder | SSM | Select one of the eight lines in a 3-to-8 decoder |
| button | CSM | Counts the number of button presses |
| vecmat | CSM | Vector matrix multiply with 4-bits element |
| addmulti tree | CM | Add multiply tree with 8-bit operand |
| accumulator | CSM | Sum an array of 8-bits elements |
| simple CPU | CM | A simple CPU implementation |

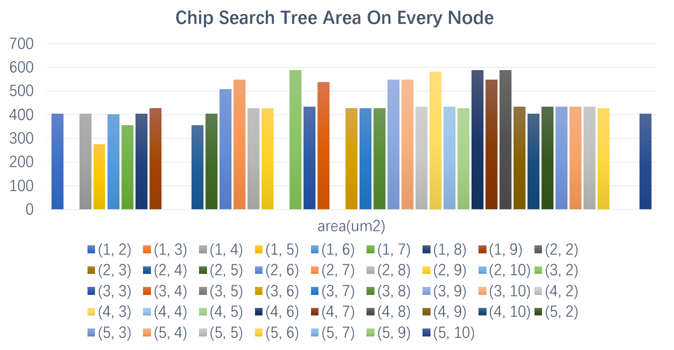

Fig. 5. The Area of Chip Search Tree Nodes on Matrix Multiplication application. $(a, b)$ represents the $a$th layer and $b$th node.

## *B. Experimental Results*

**Finding 1:** Natural language methods could achieve higher programmability than conventional agile hardware design techniques.

Evidence: Fig. 6 compares the number of code lines generated for the same designs by our language model ChipGPT, high-level synthesis (HLS) tools, and the Chisel hardware design framework. On average, ChipGPT decreased the code volume by 9.25 times compared to HLS and 5.32 times compared to Chisel. This substantial reduction clearly demonstrates the enhanced programmability enabled by natural language techniques.

**Finding 2:** Our four-stage framework optimizes PPA metrics while also improving code quality versus the original ChatGPT model.

Evidence: Fig. 7 illustrates the improved program generation efficiency enabled by our framework under the programmability metrics defined in Sec. VI-A2. Maximum quality improvement corresponds to a 2.01 times reduction in the number of incorrect code lines required, demonstrating a more compact, higher-quality final design.

**Finding 3:** Our framework improves program quality across workloads while also primarily optimizing PPA for complicated designs.

Evidence: In the chip search tree, we generate 5 layers, with each layer having a width of 10. Each node in this search tree represents an instance. Some invisible nodes in this figure indicate that the corresponding program instance is not correct. As shown in Fig.

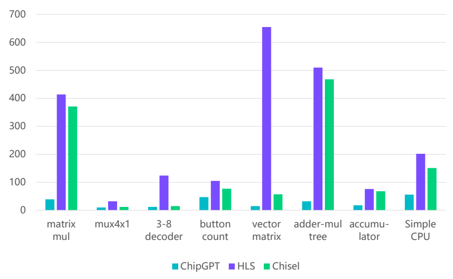

Fig. 6. Generated code line number

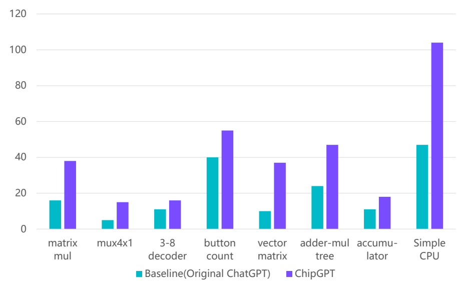

Fig. 7. Code quality

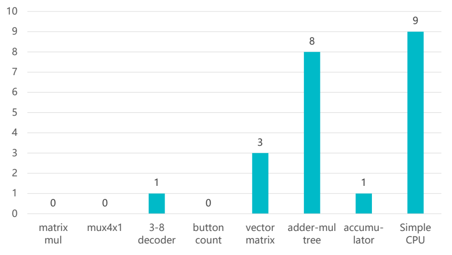

Fig. 8. Human correction effort measured in line number of code

5, the area ranges from 354.09 $\mu m^2$ to 586.81 $\mu m^2$, and it varies across different search layers. The results demonstrate that our search can evolve the program pool. Additionally, the discovery search can identify different scores, ultimately converging to a stable output.

**Finding 4:** Our composition, interface, and post-addition principles improve raw program coherence with natural language specifications, enabling high code quality at a large scale.

Evidence: Fig. 7 show that a 2.01× average increase in code quality is brought by the three principles in the prompt manager to our framework. They can enhance single-module generation quality and consistency for complex, large-scale designs.

**Finding 5:** Our framework requires minimal human feedback for module generation and single large-scale modules due to the effect of prompt management principles.

Evidence: Fig. 8 shows less than 10 lines of code needing correction for all workloads. Simple modules like the 4x1 multiplexer required no feedback, demonstrating autonomous generation. For complex modules and integrated accelerators up to 100 lines of code, only minor corrections were needed. For some cases, fewer than 10 lines of code need to be corrected.

## VII. Conclusion

This paper explores natural language hardware design and proposes ChipGPT, a search-based framework that uses natural language specifications for automatic chip logic design. It integrates language models into EDA tools without retraining. The proposed chip discovery search leverages design composition and rules to apply on hardware generation models. This could enable LLM explore the human unknown hardware design. By harnessing language, ChipGPT significantly accelerates chip development. ChipGPT is an interface for GPT to address natural language hardware design and PPA optimization, which has an area reduction compared with the original ChatGPT in area target optimization mode.

## References

[1] T. E. Mingjie Liu§, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2307.09288*, 2023.

[2] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-chat: Challenges and opportunities in conversational hardware design," *arXiv preprint arXiv:2305.13243*, 2023.

[3] K. Chang, Y. Wang, H. Ren, M. Wang, S. Liang, Y. Han, H. Li, and X. Li, "Chipgpt: How far are we from natural language hardware design," *arXiv preprint arXiv:2305.14019*, 2023.

[4] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," in *Asia and South Pacific Design Automation Conference(ASP-DAC)*, 2023.

[5] Y. Fu, Y. Zhang, Z. Yu, S. Li, Z. Ye, C. Li, C. Wan, and Y. Lin, "Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models," in *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023.

[6] 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(MLCAD)*, pp. 27–32, 2020.

[7] 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.

[8] 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," *arXiv preprint arXiv:2308.00708*, 2023.

[9] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, *et al.*, "Mathematical discoveries from program search with large language models," *Nature*, pp. 1–3, 2023.

[10] T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong, "Solving olympiad geometry without human demonstrations," *Nature*, vol. 625, no. 7995, pp. 476–482, 2024.

[11] T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong, "Solving olympiad geometry without human demonstrations," *Nature*, vol. 625, no. 7995, pp. 476–482, 2024.

[12] 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)*, 2023.

[13] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog rtl code generation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2023.

[14] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language models," *arXiv preprint arXiv:2311.16543*, 2023.

[15] Z. Liang, J. Cheng, R. Yang, H. Ren, Z. Song, D. Wu, X. Qian, T. Li, and Y. Shi, "Unleashing the potential of llms for quantum computing: A study in quantum architecture design," *arXiv preprint arXiv:2307.08191*, 2023.

[16] Z. He, H. Wu, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, “Chateda: A large language model powered autonomous agent for eda,” *arXiv preprint arXiv:2308.10204*, 2023.

[17] K. Chang, H. Ren, M. Wang, S. Liang, Y. Han, H. Li, X. Li, and Y. Wang, “Improving large language model hardware generating quality through post-llm search,”

[18] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, “Benchmarking large language models for automated verilog rtl code generation,” in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2023.

[19] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “Chisel: constructing hardware in a scala embedded language,” *Design Automation Conference*, pp. 1216–1225, 2012.