RTLCoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution

Shang Liu
HKUST
sliudx@connect.ust.hk

Wenji Fang
HKUST (GZ)
wfang838@connect.hkust-gz.edu.cn

Yao Lu
HKUST
yludf@connect.ust.hk

Qijun Zhang
HKUST
qzhangcs@connect.ust.hk

Hongce Zhang
HKUST (GZ), HKUST
hongcezh@ust.hk

Zhiyao Xie*
HKUST
eezhiyao@ust.hk

ABSTRACT

The automatic generation of RTL code (e.g., Verilog) using natural language instructions and large language models (LLMs) has attracted significant research interest recently. However, most existing approaches heavily rely on commercial LLMs such as ChatGPT, while open-source LLMs tailored for this specific design generation task exhibit notably inferior performance. The absence of high-quality open-source solutions restricts the flexibility and data privacy of this emerging technique. In this study, we present a new customized LLM solution with a modest parameter count of only 7B, achieving better performance than GPT-3.5 on two representative benchmarks for RTL code generation. This remarkable balance between accuracy and efficiency is made possible by leveraging our new RTL code dataset and a customized LLM algorithm, both of which will be made fully open-source. Furthermore, we have successfully quantized our LLM to 4-bit with a total size of 4GB, enabling it to function on a single laptop with only slight performance degradation. This efficiency allows the RTL generator to serve as a local assistant for engineers, ensuring all design privacy concerns are addressed.

1 INTRODUCTION

In recent years, large language models (LLMs) such as GPT [18] have demonstrated remarkable performance in natural language processing (NLP). Inspired by this progress, researchers have also started exploring the adoption of LLMs in agile hardware design. Many new LLM-based techniques emerge and attract wide attention in 2023. For example, LLM-based solutions are proposed to generate design flow scripts to control EDA tools [7, 12], design AI accelerator architectures [6, 27], design quantum architectures [11], hardware security assertion generation [9], fix security bugs [1], and even directly generate the target design RTL [3, 4, 12, 13, 15, 16, 24, 25].

Among the above explorations, a promising direction that perhaps attracts the most attention is automatically generating design RTL based on natural language instructions [3, 4, 12, 13, 15, 16, 24,

Works	New Training Dataset	New LLM Model	Outperform GPT-3.5
Prompt Engineering [3, 4, 15, 16, 25]	N/A	N/A	N/A
Thakur et al. [24] from NYU	**Open-Source**	**Open-Source**	No
VerilogEval [13] & ChipNeMo [12] from NVIDIA	Closed-Source	Closed-Source	Comparable
RTLCoder from HKUST	**Open-Source**	**Open-Source**	**Yes**

Table 1: LLM-based works on automatic design RTL (e.g., Verilog) generation based on natural language instructions.

25]. Specifically, given design functionality descriptions in natural language, LLM can directly generate corresponding hardware description language (HDL) code[1] such as Verilog, VHDL, and Chisel from scratch. Compared with well-explored *predictive* machine learning (ML)-based solutions in EDA [19], such *generative* methods benefit the hardware design and optimization process more directly. This LLM-based design generation technique can potentially revolutionize the existing HDL-based VLSI design process, relieving designers from the tedious HDL coding tasks.

Table 1 summarizes existing works in LLM-based design RTL generation. Some works [3, 4, 15, 16, 25] focus on prompt engineering methods based on commercial LLMs like GPT, without proposing new datasets or models for RTL code generation. As we will discuss later, reliance on commercial LLM tools limits in-depth research exploration and incurs serious privacy concerns in industrial IC design scenarios. Thakur et al. [24] generate a large unsupervised training[2] dataset by collecting Verilog-based projects from online resources like GitHub, then fine-tuning its own model. However, this unsupervised dataset is quite unorganized with a mixture of code and text. Evaluations on a third-party benchmark [15] show that the performance of its fine-tuned model is still inferior to commercial tools like GPT-3.5. The VerilogEval [13] from the NVIDIA

---

[1]Most existing works focus on generating design RTL in Verilog code. In this work, we also choose Verilog, while the method should be general and applicable to other HDL types like VHDL. We will use terms *RTL code* and *Verilog code* interchangeably.
[2]Most customized LLM solutions (including RTLCoder) are developed by fine-tuning pre-trained LLMs based on a training dataset about the specific task. In this paper, we use the terms *training* and *fine-tuning* interchangeably.

research team proposes its own labeled training dataset and benchmark, then fine-tunes its own new model. This may be the first non-commercial model that claims comparable performance with GPT-3.5, but according to their authors, neither the training dataset nor fine-tuned LLM model will be released to the public in the near future [13]. Besides these customized RTL-generation solutions, according to our study, all other software code (e.g., Python) generation models like CodeGen2 [17], StarCoder [10], and Mistral [8] are significantly inferior to GPT-3.5 in this RTL generation task.

Compared with solutions based on closed-source commercial LLM tools like GPT, the open-source LLM solution is vitally important from both research and application perspectives: 1) For research purposes, obviously, closed-source commercial tools prevent most in-depth studies and customizations of this emerging technique. 2) For realistic applications, users of commercial LLM tools unavoidably have data privacy concerns, since all instructions have to be uploaded to LLM providers like OpenAI. The privacy concern is especially critical in the VLSI design industry, where information leakage of intellectual property (IP) or key technical innovations can seriously hurt the competitive advantage of users' companies. In comparison, each user's own local LLM developed based on an open-source solution can eliminate all privacy concerns and also ensure a reliable service.

However, as mentioned, high-performance open-source RTL generation models are currently unavailable. According to our study, a major challenge is the unavailability of high-quality circuit design data for training: 1) Organized design data is mostly owned by semiconductor companies, who are almost always unwilling to share design data. 2) Design data directly collected online is messy and unorganized, either leading to inferior model performance or requiring prohibitive human efforts to clean the dataset.

In this work, we finally fill this gap with our new open-source LLM solution named **RTLCoder**[3]. To the best of our knowledge, it is the first non-commercial LLM method that clearly outperforms GPT-3.5 in design RTL code generation. We validate this on two representative benchmarks [13, 15] and observe consistent trends. To build this RTLCoder, we first propose an automated data generation flow and have generated a high-quality labeled dataset with over 10,000 samples for the RTL generation task.

RTLCoder obviously achieves state-of-the-art trade-offs between performance and efficiency. Besides demonstrating unprecedented RTL generation correctness in non-commercial solutions, it only has 7 billion (B) parameters and can be trained with only 4 consumer-level GPU cards. After further quantizing the parameters to 4 bits, the RTLCoder-4bit takes only 4GB of memory and can work on a laptop with limited accuracy loss. As a result, our open-source lightweight RTLCoder solution is accessible to almost every research group. The contributions of RTLCoder can be summarized below:

- Targeting Verilog code generation, we propose an automated flow to generate a large labeled dataset with over 27,000 diverse Verilog design problems and answers. It addresses the serious data availability challenge in IC design-related tasks, and its potential applications are not limited to LLMs. The LLM directly trained on it can already achieve comparable accuracy with GPT-3.5.

- We introduce a new LLM training scheme based on code quality feedback. It further boosts the ultimate model performance to outperform GPT-3.5, being comparable with GPT-4. We further revised the training process from an algorithm perspective to reduce its GPU memory consumption. The training process only requires 4 commercial-level GPU cards.
- We designed RTLCoder to be a lightweight solution with only 7B parameters. After quantizing its parameters into 4 bits, it takes only 4GB of memory, allowing it to serve as a local assistant for engineers without privacy concerns.
- RTLCoder will ultimately be fully open-sourced, including our data generation flow, complete generated dataset, LLM training algorithm, and the fine-tuned model. Considering RTLCoder's lightweight property and low hardware barrier, it allows anyone to easily replicate and further improve based on our existing solution.

2 AUTOMATIC DATESET GENERATION

In this work, we first propose a new automated training dataset generation flow. Based on this flow, we have generated over 27 thousand training samples, with each sample being a pair of design description instruction (i.e., model input) and the corresponding reference RTL code (i.e., expected model output). The instruction can be viewed as the input question for LLMs, describing the desired circuit functionality in natural language. The reference code is the expected answer from LLMs, implementing the circuit functionality in Verilog code. We observe that these generated training samples exhibit high diversity and complexity in the RTL-generation domain, encompassing a diverse spectrum of difficulty levels.

We build this automated generation flow by taking full advantage of the powerful general text generation ability of the commercial tool GPT. Please notice that GPT is only used for dataset generation in this work, and we adopt GPT-3.5 in this data generation task. The automated dataset generation flow is illustrated in Figure 1, which includes three stages: 1) RTL domain keywords preparation, 2) instruction generation, and 3) reference code generation. We designed several general prompt templates to control GPT generating the desired outputs in each stage.

2.1 Stage 1: Keywords Preparation

The first stage of our data generation flow targets preparing RTL domain keywords for subsequent stages. At process ❶ shown in Figure 1, we request GPT to generate keywords related to digital IC design (i.e., commonly used logic components) based on a set of prompts P_{key}. We obtain a keyword pool $\mathcal{L}_{key}$ with hundreds of digital design keywords.

Specifically, in this process ❶, to collect a comprehensive range of RTL design task topics, we utilize a tree-like structure with multiple branches to issue queries to GPT. We first prompt GPT at the root node to provide categories and examples of frequently used block keywords in RTL design as Figure 2 illustrated. The response from GPT has a tree structure that consists of some subfields as Figure 3 shows. With the response, we could use the categories and examples as branches to continue prompting GPT for more design keywords within each topic. For example, we can use scripts to ask GPT about more types of the block "multiplier", it will return
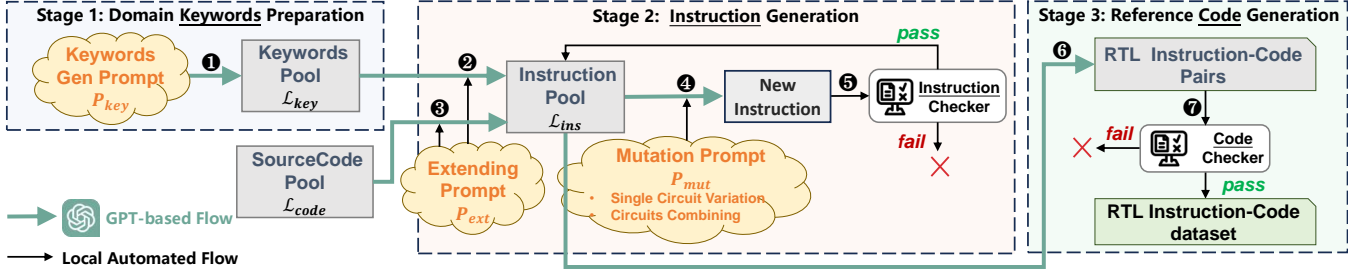
Figure 1: Our proposed automated training dataset generation flow.

Figure 2: An example of Prompt P_{key} in ❶ [4]

Figure 3: A GPT response example to Prompt P_{key} in ❶

more specific design names such as "Booth multiplier, Wallace tree multiplier, etc.". After this process, we obtain hundreds of keywords related to RTL design in the Keywords pool $\mathcal{L}_{key}$.

2.2 Stage 2: Instruction Generation

The second stage targets generating sufficient instructions based on the initial keywords and Verilog source code. At process ❷, we extend existing keywords from $\mathcal{L}_{key}$ to complete design instructions. Specifically, we randomly sample one or two keywords from $\mathcal{L}_{key}$ each time, combined with prompts P_{ext}, and feed them into GPT. The output is a complete RTL design instruction.

In addition to keyword-based instruction generation in process ❷, we also propose to generate instructions based on existing source code collected by us, as shown in process ❸. This is partially inspired by the work of [26]. By providing GPT with either part or a complete Verilog code $\mathcal{L}_{code}$ collected by [24], we can inspire it to create a related Verilog design problem. By adopting this new ❸ together with ❷, we further enhance the diversity of our dataset by utilizing a vast and varied collection of source code.

Process ❷ and ❸ help generate the initial design instruction pool $\mathcal{L}_{ins}$ based on our customized prompt P_{ext}. Two types of prompt P_{ext} are proposed for processing $\mathcal{L}_{key}$ and $\mathcal{L}_{code}$, denoted as P_{ext}^{key} and P_{ext}^{code}, respectively. As shown in Figure 4, our prompt P_{ext}^{key} in process ❷ adopts the few-shot prompting technique, which means we provide an example of the question (i.e., keyword) and answer

Figure 4: An example of Prompt P_{ext}^{key} in ❷

(i.e., instruction) in the input prompt. Figure 5 shows an example of GPT's corresponding response. As for the prompt P_{ext}^{code} used in process ❸, an example of prompt and the response of GPT are provided in Figure 6 and Figure 7. The prompt P_{ext}^{code} asks GPT to convert the given Verilog code snippet to the corresponding description instruction.

After generating the initial instruction pool $\mathcal{L}_{ins}$ with hundreds of initial instructions, we will iteratively use mutation methods to significantly augment the scale and complexity of this pool. At ❹, we use P_{mut} to apply two types of mutation operations on instructions sampled from the design instruction library $\mathcal{L}_{ins}$. The process ❹ would check every new design instruction using a set of rules and only passed valid instructions are added to $\mathcal{L}_{ins}$. We cover more details of this iterative process below.

For the mutation operation in ❹, we propose two types of prompts P_{mut}^s and P_{mut}^c, performing single circuit variation and circuits combination, respectively.

- The P_{mut}^s focuses on changing the single circuit module by requiring different functionalities while keeping the same hardware type, or implementing the same functionality but using different hardware. Figure 8 is an example of P_{mut}^s and Figure 9 is a GPT's response example with given instructions.
- The P_{mut}^c focuses on combining two circuit designs together. They are combined by either adding control logic to select using one of the two functionalities (i.e., parallel combination) or feeding one output into the other's input (i.e., serial combination). Figure 10 gives a sample of prompt P_{mut}^c and Figure 11 is a GPT's response example with given instructions.

---

[4]We use *red* text boundary to denote GPT *input* examples, and *green* text boundary to denote GPT *output* examples in this work. Please notice that some *green* GPT *output* in this data generation flow are instructions, which will be the input of LLMs.

Figure 5: A GPT response example (green) to Prompt P_{ext}^{key} when given topic: Pulse width modulators (PWM) (red) in ❷

Figure 6: An example of Prompt P_{ext}^{code} in ❸

Figure 7: A GPT response example to Prompt P_{ext}^{code} in ❸.

Figure 8: An example of Prompt P_{mut}^{s} in ❹

Figure 9: A GPT response example (green) to Prompt P_{mut}^{s} with Given Instruction (red) in ❹

In addition, we will further request GPT to generate its reasoning steps (i.e., how it analyzes the generation task step-by-step). These reasoning steps further enhance the detailed information of our instruction pool.

For the instruction checking in ❺, we automatically check the correctness and diversity of each new instruction generated in the mutation process, based on a series of rules. The rules include 1) invalid-word checking, and 2) similarity measurement with existing $\mathcal{L}_{ins}$. For example, in invalid-word checking, if words such as "image" and "text" are detected, it indicates that GPT has likely provided a problem that cannot be implemented solely with RTL code. As for similarity rate checking, we calculate the similarity between the new instruction and existing $\mathcal{L}_{ins}$. The new instruction is discarded if the similarity is above a certain threshold since it can't add enough new information to $\mathcal{L}_{ins}$. Such checking of instructions in

You should create a **new task** based on the two given instructions. The output of one module could be the input of the other. Here is an example for you.

[Given instructions]
1. Create a module that implements a **binary counter**...
2. Create a module that implements a **comparator**...

[New instruction]
Create a module that implements a **binary counter** that counts from 0 to 15, ..., Additionally, include a **comparator** that compares the value of the counter to a given value, and outputs a signal indicating whether the counter value is greater than or equal to the given value.

Figure 10: An example of Prompt P^c_{mut} in ❹

[Given Instruction]
1. Please act as a professional Verilog coder. Create a module that implements a **finite state machine (FSM)**, used for **5-bit sequence detection**...
2. You are tasked with designing a **4-bit binary to BCD (Binary Coded Decimal) converter** using Verilog. The binary input should be in two's complement notation, and the output should be in BCD format.

Response:
[New Instruction]
Create a module that combines a **5-bit sequence detection finite state machine (FSM)** and **a 4-bit binary to BCD converter**. The FSM module should detect a specific 5-bit sequence in the input data stream. When the sequence is detected, the FSM should output a 'match' signal. The binary to BCD converter should accept a 4-bit two's complement binary input and convert it to BCD format. The BCD conversion should only be performed when the 'match' signal from the FSM is high.

Figure 11: A GPT response example (green) to Prompt P^c_{mut} with Given Instruction (red) in ❹

❺ is imperfect, but it is fully automated and accurate enough to generate a high-quality ultimate instruction pool $\mathcal{L}_{ins}$, including over 50,000 instructions.

2.3 Stage 3: Reference Code Generation

The third stage targets generating the reference code. In the third stage, as shown in ❻, we feed each instruction from $\mathcal{L}_{ins}$ into GPT, generating the corresponding reference design code as the solution. Specifically, we will ask GPT to provide at least five answers. After that, in ❼, we will evaluate these answers using a code checker. In this work, we adopt an automated syntax checker to verify the correctness of RTL code syntax. Only syntax-correct design code can be kept. If all five answers fail the syntax checking, this instruction will be discarded. Finally, only valid instruction-code pairs are saved as our dataset. Ideally, process ❼ should also check whether the functionality of the generated RTL code is consistent with the instruction, but currently generating testbenches for functionality verification cannot be automated. Similar to the checker in stage 2, this imperfect automated checking can already filter out the most serious mistakes in the dataset.

After going through all three proposed stages, we generate the ultimate training dataset with more than 27,000 data samples. An

interesting observation is that, although we generate our training dataset based on GPT-3.5, RTLCoder turns out to outperform the GPT-3.5 baseline on representative benchmarks [13, 15]. One important reason is that, for each instruction, we have employed a syntax checker to evaluate the reference code generated based on GPT-3.5. Therefore, among all correct and incorrect code from GPT-3.5, we filter out the obviously incorrect ones and retain the largely correct ones for training RTLCoder. This process can be viewed as a refinement of GPT-3.5's Verilog generation capabilities.

3 NEW TRAINING SCHEME INCORPORATING CODE QUALITY FEEDBACK

Besides the new training dataset, we propose a new LLM training scheme that incorporates code quality scoring. It significantly improves the RTLCoder's performance on the RTL generation task. Also, we revised the training process from the algorithm perspective to reduce the GPU memory consumption of this new training method, allowing implementation with limited hardware resources.

3.1 Existing Supervised Training on LLMs

This part will first introduce the existing supervised training method for LLMs. Then we will further discuss its limitations in RTL generation tasks. Suppose we have a training data dateset $\{x_i, y_i\}$ for $i = 1, ..., N$, where x_i represents an design instruction, y_i represents the corresponding correct reference code. Each sample of data will be split into a sequence of tokens by certain rules during the preprocessing process. In this paper, we use $x_i = \{x_i^t\}$ and $y_i = \{y_i^t\}$ for $t = 1, 2, ..., T$ to represent the tokenized sequence.

LLMs generate a sequence by continuously predicting the next token based on the already generated previous ones. For a decoder-only language model, which is the mainstream LLM architecture, the probability of producing the next token depends only on the previous output tokens and the input instruction. We denote the probability of generating the t-th token r_t (r_t can be any single token in the vocabulary) as $P_\pi \left(r_t \mid x_i, y_i^{<t} \right)$ where π represents the model parameters and $y_i^{<t}$ denotes the already generated previous tokens $\{y_i^1, .., y_i^{t-1}\}$. Then the log probability of generating the whole sequence can be written as: $\sum_{t=1}^{T} \log P_\pi \left(y_i^t \mid x_i, y_i^{<t} \right)$.

In the existing training method, Maximum Likelihood Estimation (MLE) is commonly used to find the best parameters π that maximize the log probability. The training flow is shown in Figure 12(a). The loss is usually defined as below:

$$loss_{mle} = -\sum_{t=1}^{T} \log P_\pi \left(y_i^t \mid x_i, y_i^{<t} \right)$$

However, there exists a phenomenon named *exposure bias* [2, 14]. Since the above sequence generation is autoregressive, which means the model always predicts the next token based on its own generated previous ones $r_i^{<t}$ rather than the reference tokens $y_i^{<t}$. Therefore, even though the probability of producing y_i^t is high when given $y_i^{<t}$ in the training, it can still result in a huge deviation from the reference code in the generation process.

We have also observed this phenomenon in our experiments. After the supervised training, the qualities of multiple generated code candidates for the same instruction may diverse greatly in
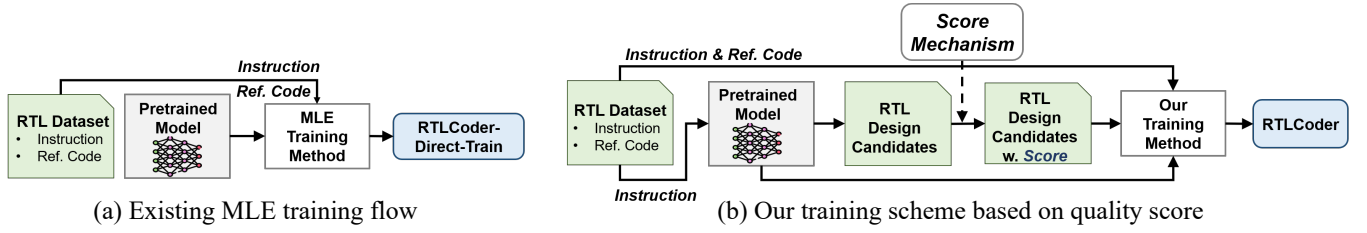
(a) Existing MLE training flow   (b) Our training scheme based on quality score

Figure 12: Comparison between (a) existing MLE-based LLM training flow and (b) our proposed LLM training flow.

the performance aspect. They can include correct code while at the same time including many low-quality answers. Some candidates exhibit serious nonsense duplication[5].

To alleviate the *exposure bias* phenomenon, we suggest that in addition to the reference code y_i, the model's generation should also be considered in the training process. Since the generation may be different from the reference code, it is necessary to introduce a scoring mechanism to judge the quality of generated candidates. We will give our detailed solution in Section 3.2.

3.2 Our Proposed Training Method

Our proposed training scheme is illustrated in Figure 12(b). For each instruction, we will now collect multiple code candidates generated by the initial pre-trained model. Then, we pack these candidates and the original reference code y_i together as $\mathbf{y}_i = \{y_{i,k}\}$, $k = 1, 2, .., K$, where K represents the number of generated code for one instruction. Next, all these candidates will be scored by the scoring mechanism $R(x_i, y_{i,k})$ which could be a syntax checker or unit test for functionality check. We will then obtain a set of score $\mathbf{z}_i = \{z_{i,k}\}$, $k = 1, 2, .., K$, denoting the quality for the code sample $\{y_{i,k}\}$. In the training process, we aim to make the model learn to assign relatively higher generation probabilities to answers with higher scores. In this way, the model not only learns from the reference code, but also from the new information introduced by the quality score feedback.

The conditional log probability (length-normalized) of generating the entire code $y_{i,k}$ is commonly written as:

$$p_{i,k} = \frac{\sum_t \log P_\pi \left(y_{i,k}^t \mid x_i, y_{i,k}^{<t} \right)}{\|y_{i,k}\|}$$

We calculate $p_{i,k}$ for all code candidates $\mathbf{y}_i = \{y_{i,k}\}$, $k = 1, 2, .., K$, then we normalize these $p_{i,k}$ values using a *softmax* function, defining the probability of each code being selected as:

$$s_{i,k} = \frac{e^{p_{i,k}}}{\sum_{\tau=1}^{K} e^{p_{i,\tau}}}$$

This $s_{i,k}$ reflects the model's tendency to output the kth code candidate, with higher probabilities indicating a greater likelihood that the model will generate it.

To encourage the model to assign higher probability scores to high-quality code, we can define a new loss function term as:

---

[5]We notice that this duplication couldn't be simply dealt with by adding repetition penalty to the decoding process like other works in natural text generation. Because some correct RTL design code also contain similarly repetitive expressions.

Algorithm 1 Training scheme using gradients splitting

Input: The single data sample $\{x_i, \mathbf{y}_i, \mathbf{z}_i\}$. Model forward function $s_{i,k} = f_\pi(x_i, y_{i,k}, z_{i,k})$. Loss calculation function $L_\pi(\mathbf{s}_i, \mathbf{z}_i)$. GPU affordable batch size J. Model parameters w.
Output: The derivative of the loss with respect to model parameters: g_i.
1: Group the sample $\{x_i, y_{i,k}\}$ for $k = 1, 2, ..., K$ into Q parts based on batch size J.
2: initialize empty vector list $temp$. Initialize the gradients $g_i = 0$.
3: **for** $q \in Q$ **do**
4:      Calculate $s_{i,k} = f_\pi(x_i, y_{i,k}, z_{i,k})$, for $k \in q$.
5:      Empty the computation graph
6: Calculate $loss = L_\pi(\mathbf{s}_i, \mathbf{z}_i)$ // $\mathbf{s}_i = \{s_{i,k}\}$ for $k = 1, .., K$
7: Backward process: $temp_k = \partial loss / \partial s_{i,k}$, for $k = 1, ..., K$
8: **for** $q \in Q$ **do**
9:      Calculate $s_{i,k} = f_\pi(x_i, y_{i,k}, z_{i,k})$, for $k \in q$
10:      Backward process: $g_i = g_i + \sum_{k \in q} temp_k \partial s_{i,k} / \partial w$
11:      Empty the computation graph
12: Return g_i

$$loss_{compare} = \sum_{z_{i,k} < z_{i,\tau}} max \left(s_{i,k} - s_{i,\tau} + \lambda, 0 \right)$$

where λ is a threshold value.

To provide an intuitive explanation of this loss function term, we provide a simple example. Suppose we have the ith instruction and only two code candidates with initial selection probability $s_{i,1}$ and $s_{i,2}$ with $s_{i,1} + s_{i,2} = 1$ and $s_{i,1} > s_{i,2}$. But the first candidate has a lower quality score, i.e., $z_{i,1} < z_{i,2}$. Then the positive loss would drive model parameters to update until the model assigns a new set of $s_{i,1}^*$ and $s_{i,2}^*$ so that $s_{i,2}^* - s_{i,1}^* \geq \lambda$ is satisfied.

It is worth noting that this loss only depends on the relative scores among multiple code candidates, so it can still be used when answer quality cannot be precisely quantified. Finally, We define the total loss as:

$$loss = loss_{compare} + loss_{mle}$$

3.3 Reduced Memory by Splitting Gradients

Directly calculating our new *loss* function even with 1 batch size would still require forwarding all code candidates in a sample at once to maintain all the activation values. This will lead to the $O(K)$ space complexity and make the GPU memory consumption prohibitively high in many large language model training scenarios.

We propose a gradient-splitting approach for model training based on quality score from an algorithm perspective. It can achieve a $O(1)$ space complexity as illustrated in Algorithm 1. The gradients of *loss* with respect to w can be computed as below:

$$\frac{\partial \text{loss}}{\partial w} = \sum_k \frac{\partial \text{loss}}{\partial s_{i,k}} \frac{\partial s_{i,k}}{\partial w}$$

The property of the chain rule indicates that we can decompose the gradient updates into several parts. Assume J is the maximum allowable batch size for GPU consumption. We divide the K candidates into Q groups based on the batch size J. Firstly, we pass these groups through the forward function separately and collect the obtained $\mathbf{s}_i$ values as lines 1-5 illustrate. In the second step, we calculate the loss function and compute the derivative of the loss with respect to $\mathbf{s}_i$ in lines 6-7, storing the temporary results in vector $temp$. In the third step, we perform the forward operation on the original Q groups again and for each forward operation, the obtained $s_{i,k}$ is multiplied by $temp_k$ in a dot product, followed by a backward pass to accumulate the gradient in lines 9-12.

4 EXPERIMENTAL RESULTS

4.1 Evaluation Benchmark and Metric

To evaluate the performance of Verilog code generation, there are two representative benchmarks VerilogEval [13] and RTLLM [15].

The VerilogEval [13] benchmark consists of two parts, EvalMachine and EvalHuman, each including more than 100 RTL design tasks. We follow the original paper [13] and use pass rate as the metric: passrate $= E_i \left(c_i / n \right)$ where n is the total number of trials for each instruction and c_i is the number of correct code generations for task i. We set $n = 20$ in this experiment. This metric reflects the expected probability of getting the correct answer on one attempt, thus indicating the model's task-solving ability

The updated version of RTLLM V1.1 [15] benchmark contains 29 RTL design tasks at a larger design scale. It has fixed some problems in the original RTLLM V1.0. We mostly follow the testing method in the original paper [15], but further proposes two slightly different metrics for evaluating syntax correctness, using either Synopsys VCS [22] or Design Compiler [21]. They are denoted as Syn-VCS and Syn-DC, respectively. 1) For the Syn-VCS metric, VCS not only requires the design to comply with the Verilog syntax rules, but also requires that the interface of the design correspond to the testbench, so that the circuit can be simulated. 2) For the Syn-DC metric, DC requires the design to be physically synthesizable. The functionality result is obtained by VCS simulation. We calculate the scores of the design syntax part and design functionality part separately. In both parts, it will count as success for each problem as long as *any* of the generated 5 code samples pass the test. The experiment result of the older version RTLLM V1.0 is also provided only for reference. We suggest our readers only use the result of RTLLM V1.1 instead of V1.0 for comparison.

In the generation process, we set $top_p = 0.95$ and $temperature = \{0.2, 0.5, 0.8\}$. For all tested models (i.e., baselines, RTLCoder, and ablation studies), we evaluate all 3 $temperature$ conditions and report the best performance for each model.

4.2 Examine Training Set for Fair Evaluation

To ensure a fair evaluation of our proposed RTLCoder, before training, we explicitly examined the similarity between samples in our proposed training dataset and those test cases in benchmarks [13, 15], then we get rid of our training samples that are similar to test cases during the training process.
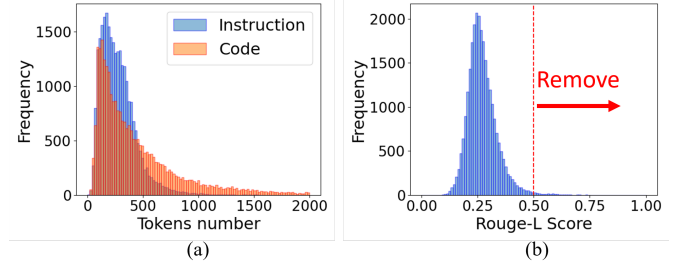
Figure 13: Training dataset analysis. (a) Tokens number distribution of instruction and code part. (b) Similarity measurement between training dataset and two benchmarks based on Rouge-L metric.

To measure the similarity between two text sequences, we employed the Rouge-L metric, which is a widely-used similarity calculation scheme in the LLM domain such as by OpenAI [18]. The Rouge-L score $\in [0, 1]$, with values closer to 1 indicating higher similarity between the two sequences. For each instruction-code concatenated sample in the training dataset, we computed its Rouge-L value with all test cases in the benchmarks. In addition, we also separately analyzed the distribution of token counts for instructions and code in the dataset. The resulting statistic is in Figure 13.

From Figure 13 (a), we can see that a sample that consists of one instruction and one code candidate is generally within 2048 token length. So we can set 2048 as the max length in our finetuning. In Figure 13 (b), we observed that the majority of training samples in the dataset have a low overlap compared with the benchmark, with Rouge-L scores < 0.3. However, there are still a small number of samples with higher similarity. To ensure fair evaluation of the RTLCoder, we get rid of training samples with Rouge-L values > 0.5 during training.

4.3 Model Training

Based on our generated dataset with 27K instruction-code pairs, we choose the latest Mistral-7B-v0.1 [8] as the basic pre-trained model for finetuning. In all experiments, we opted for the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and learning rate $\gamma = 1e-5$, while abstaining from the use of weight decay. Concurrently, we established a context length of 2048 and a global batch size of 256. We trained the model on only 4 consumer-level RTX 4090 GPUs (24GB each), each of which could only afford 2×2048 context length using DeepSpeed stage-2 [20]. Under the hardware constraint, the training is impossible without the proposed gradient-splitting method.

To implement our proposed training scheme, we first generated 3 code candidates for each instruction using the pre-trained model with the Beam search method. Then we use Pyverilog [23] as the syntax checker to score the code candidates. Specifically, we assigned a full score (i.e., 1) for the reference code from the dataset and those candidates who can pass the syntax check. For those who failed syntax checks, we used the Rouge-L metric to assign the code similarity between the candidate and reference code as its score.

In addition, considering GPU memory consumption is a crucial factor that limits the applicability of LLMs, based on quantization

Model Description	Evaluated Model	Num of Params	VerilogEval Benchmark [13]		RTLLM V1.0 [15]		RTLLM V1.1 [15]®	
			Eval-Machine (%)	Eval-Human (%)	Syntax (%)	Func (%)	Syn-VCS (%)	Func (%)
Closed-Source Baseline	GPT-3.5	N/A	46.7	26.7	63.3	33.3	89.7	37.9
	GPT4	N/A	60.0	**43.5**	86.7	**50.0**	**100**	**65.0**
	VerilogEval★ [13]	16B	46.2	28.8	N/A	N/A	N/A	N/A
	ChipNeMo★ [12]	13B	43.4	22.4	N/A	N/A	N/A	N/A
Open-Source Baseline	Codegen2 [17]	16B	4.90	1.28	46.7	5.77	72.4	6.90
	Starcoder [10]	15B	46.8	18.1	30.0	16.7	93.1	27.6
	Thakur et al. [24]	16B	44.0	30.3	40.0	16.7	86.2	24.1
Old-version **RTLCoder** (10K training samples)	**RTLCoder-4bit**	7B * 4bit	49.8	28.8	80.0	30.0	82.8	31.0
	RTLCoder	7B	56.5	31.7	83.0	36.7	86.2	34.5
Ablation Study of **RTLCoder**	Mistral-7B-v0.1 [8]	7B	36.9	4.49	76.7	10.0	72.4	20.7
	Direct-Train	7B	58.9	34.4	86.7	33.3	89.7	41.4
This-version **RTLCoder** (27K training samples)	**RTLCoder-4bit**	7B * 4bit	59.5	33.8	**90.0**	33.3	86.2	41.4
	RTLCoder	7B	**62.5**	36.7	**90.0**	40.0	96.6	48.3

★We cannot evaluate VerilogEval [13] and ChipNeMo [12] on RTLLM Benchmark [15] due to the unavailability of closed-source models. We fully understand and respect the authors' privacy concerns. The accuracy values of VerilogEval [13], ChipNeMo [12], GPT-3.5, and GPT-4 on the VerilogEval Benchmark [13] are directly cited from the original publication [13]. Please also notice that the authors [13] revised their reported GPT-4 accuracy on Dec 10th, 2023, fixing prior measurement errors. We used their latest values.
®In this table, we report Syn-VCS as the syntax metric of RTLLM V1.1. As for the previous RTLLM V1.0, the syntax metric was not precisely defined. We suggest our readers only use the measurement result of RTLLM V1.1 instead of V1.0 for comparison.

Table 2: Performance comparison of RTL code generators on VerilogEval Benchmark [13] and RTLLM Benchmark [15]. RTLCoder is only second to GPT-4, outperforming GPT-3.5 and all others when measured on both [13] and [15] benchmarks.

methodologies [5], we further quantize the parameters of the obtained RTLCoder into 4 bits, generating RTLCoder-4bit, consuming only 4GB memory.

4.4 Experiment Results Overview

Table 2 summarizes the comparison of all relevant solutions, including commercial models GPT3.5/GPT4, models customized for Verilog generation [13, 24], software code generators [8, 10, 17], both old and new versions of our proposed RTLCoder and RTLCoder-4bit, and ablation studies of RTLCoder.

In the VerilogEval benchmark [13], for both EvalHuman and EvalMachine categories, RTLCoder scores 62.5 and 36.7 respectively. It clearly outperforms GPT-3.5 and is only inferior to GPT-4 among all the models in EvalHuman. Specifically, in the Eval-Machine part, RTLCoder even outperforms GPT4 by an absolute value of 2.5%. A similar trend can be observed in the RTLLM benchmark V1.1 [15]. RTLCoder is also second only to GPT-4. In summary, RTLCoder outperforms GPT-3.5 and all non-commercial baseline models in all metrics on both benchmarks. It is surprising that the lightweight RTLCoder with only 7 billion parameters could achieve such impressive accuracy despite its smaller size.

Furthermore, we validate the effectiveness of our proposed dataset and algorithm through an ablation study. The RTLCoder-Direct-Train is directly trained with the existing method mentioned in Figure 12(a). Using our training dataset, it can already significantly outperform the base model Mistral-7B-v0.1 and even GPT-3.5 on the VerilogEval benchmark [13] and RTLLM V1.1 [15]. Then the RTLCoder trained with our proposed training scheme further outperforms RTLCoder-Direct-Train on all benchmarks, indicating that our training method greatly further improves the model performance.

In addition, although the quantized model RTLCoder-4bit shows a slight performance degradation compared to the original model, it

(a) EvalMachine syntax

(b) EvalMachine functionality

(c) EvalHuman syntax

(d) EvalHuman functionality
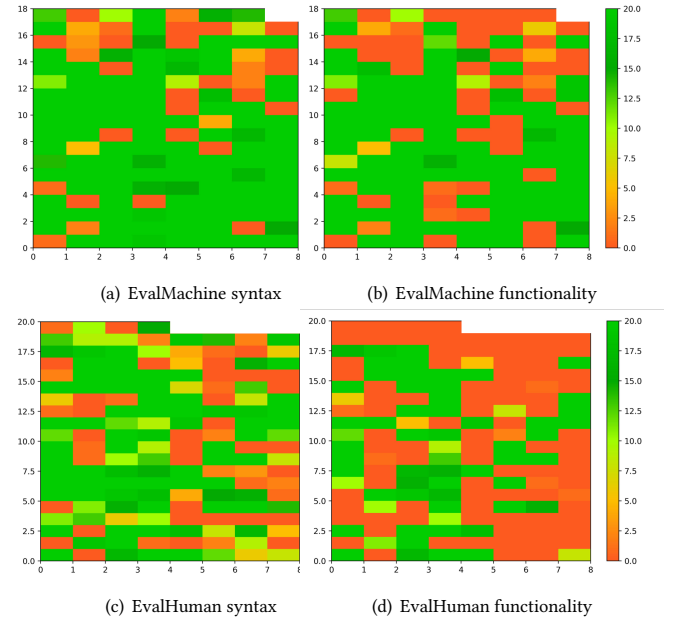
Figure 14: Detailed syntax and functionality results of RTLCoder on VerilogEval Benchmark [13], reporting EvalMachine and EvalHuman separately. Each sub-figure has 8 columns, and thus cell at (i, j) represents the $((j-1) \times 8 + i)^{\text{th}}$ task. The color of each cell indicates the count of correct cases among 20 trials. EvalMachine contains 143 tasks, so the last 1 cell is empty. EvalHuman contains 156 tasks, so the last 4 cells are empty.

is still superior to GPT-3.5 on the VerilogEval benchmark and comparable to it on RTLLM V1.1 with only 4GB size. Such RTLCoder-4bit can work on a simple laptop, allowing it to serve as a local assistant for engineers, addressing privacy concerns.

Table 3: Detailed Syntax and Functionality Evaluation Results using sampling generation method in RTLLM V1.1 [15]

Design	GPT-3.5			GPT-4			Thakur et al. [24]			StarCoder[10]			RTLCoder-4bit			RTLCoder		
	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func
accu	2	2	✔	5	5	✔	4	4	✗	3	4	✗	5	5	✗	4	4	✗
adder_8bit	3	3	✔	4	4	✔	3	3	✔	2	4	✗	5	5	✔	5	5	✔
adder_16bit	1	0	✗	3	3	✔	3	4	✗	2	3	✗	0	0	-	3	3	✗
adder_32b	0	0	-	2	2	✔	1	0	✗	1	3	✗	1	0	✗	1	0	✗
adder_pipe_64b	5	5	✗	5	5	✔	0	0	-	0	0	-	1	1	✗	3	2	✗
multi_booth_8b	5	2	✗	5	5	✗	3	3	✗	4	3	✗	5	5	✔	5	5	✔
multi_16b	5	0	✗	5	5	✔	3	3	✗	3	4	✗	4	2	✗	5	5	✔
multi_pipe_4b	0	0	-	2	2	✗	1	0	✗	3	1	✗	4	1	✗	2	0	✗
multi_pipe_8b	2	0	✗	5	5	✗	3	1	✗	2	3	✗	0	0	-	2	0	✗
div_8bit	3	1	✗	5	1	✗	0	1	-	3	0	✗	3	1	✗	4	1	✗
div_16bit	4	0	✗	5	4	✔	1	2	✗	1	1	✗	0	0	-	0	0	-
JC_counter	5	5	✗	5	5	✗	3	3	✗	4	5	✗	5	5	✔	5	4	✔
right_shifter	4	4	✔	5	5	✔	0	2	-	3	3	✔	5	5	✔	5	5	✔
synchronizer	5	5	✔	4	4	✔	4	4	✔	5	5	✔	4	4	✔	5	5	✔
counter_12	5	5	✔	5	5	✔	2	4	✔	2	4	✔	5	5	✔	5	5	✔
freq_div	5	5	✔	5	5	✔	4	4	✔	4	4	✗	5	5	✔	5	3	✔
signal_gen	5	5	✔	5	5	✔	4	5	✗	4	4	✗	5	5	✗	5	5	✗
serial2parallel	4	4	✗	5	5	✔	4	4	✗	4	4	✗	5	3	✗	5	3	✗
parallel2serial	2	2	✗	5	5	✗	1	2	✗	3	4	✔	3	3	✗	3	2	✔
pulse_detect	4	4	✗	5	3	✗	4	3	✗	3	3	✗	5	5	✗	2	2	✗
edge_detect	5	5	✔	5	5	✔	4	5	✔	3	4	✔	4	2	✔	5	4	✔
FSM	5	4	✗	5	2	✗	4	4	✗	5	5	✗	4	4	✗	5	5	✗
width_8to16	4	3	✔	5	5	✔	4	1	✔	3	4	✗	5	5	✗	5	4	✔
traffic_light	4	0	✗	4	3	✔	5	2	✗	5	3	✗	4	0	✔	4	3	✔
calendar	5	5	✔	5	5	✔	2	1	✗	5	4	✔	1	0	✗	5	5	✔
RAM	4	0	✔	5	2	✔	5	5	✔	2	0	✔	3	0	✔	3	0	✔
asyn_fifo	0	0	-	3	2	✗	0	0	-	0	0	-	0	2	-	1	3	✗
ALU	2	0	-	5	4	-	2	2	✗	1	0	✗	2	1	✗	1	0	✗
PE	5	5	✔	5	5	✔	3	3	✗	3	5	✔	1	1	✔	5	5	✔
Success rate	89.7%	65.5%	11/29	100%	100%	19/29	86.2%	86.2%	7/29	93.1%	82.8%	8/29	86.2%	75.9%	12/29	96.6%	79.3%	14/29

Finally, compared with the old version RTLCoder which is trained on a 10K size dataset, this new version RTLCoder trained on a 27K dataset is clearly superior to it on all metrics. We can see that increasing the size of the training dataset and enhancing its diversity can further improve the model performance.

4.5 Experiment Results in Detail

To further examine the performance in detail, for both benchmarks [13, 15], we report RTLCoder's performance on each individual design case in both syntax and functionality correctness.

We list the test results of RTLCoder and available baseline models on the RTLLM V1.1 benchmark for each design task in Table 3. Given 5 trials of generation, here we counted the number of passed cases in terms of Syn-VCS, Syn-DC, and Functionality. As introduced, for both syntax and functionality, we count one success if any of the 5 trials pass the test. Generally, Syn-VCS is easier to pass than Syn-DC.

We further inspect the wrong answers in Table 3. We observed that the overall code structures of wrong answers from GPT-3.5, GPT-4, and RTLCoder exhibit no obvious mistakes, despite the functionality incorrectness. In comparison, the code generated by other open-source baselines occasionally contains redundant content or deviates considerably from the given description. In terms of syntax, we observed that both GPT and RTLcoder frequently assign 0 directly to two-dimensional arrays, resulting in syntax errors. Regarding functionality, we noticed that for more complex combinational logic circuits such as multi_pipe_4bit and multi_pipe_8bit, and sequential logic circuits like pulse_detect and FSM, some of the logical behaviors described in the instructions are not adequately captured by all LLM solutions, leading to functional errors.

The RTLCoder's results on VerilogEval Benchmark are reported in Figure 14. Each cell in the image represents one design case,

Table 4: Evaluation results using different decoding methods in RTLLM V1.1 Benchmark [15]. The result of the sampling decoding method is adopted and reported in the Table 2.

Model	Sampling decoding [used in experiment]			Beam search decoding [for ablation study]		
	Syn-VCS	Syn-DC	Func	Syn-VCS	Syn-DC	Func
Thakur et al. [24]	86.2	86.2	24.1	69.0	51.7	17.2
StarCoder[10]	93.1	82.8	27.6	58.6	58.6	17.2
RTLCoder-4bit	86.2	75.9	41.4	75.9	65.5	31.0
RTLCoder	96.6	79.3	48.3	75.9	72.4	37.9

with color indicating the number of successful ones among all 20 trails. There are 8 columns in each image. The location of cell (i, j) represents the $((j-1) \times 8 + i)^{\text{th}}$ design case in the provided description file. So we used white cells to fill the cells in the last row (18$^{\text{th}}$ row for EvalMachine and 20$^{\text{th}}$ row in the EvalHuman) that do not correspond to a design task.

During the process of generating text sequences, the model continuously repeats the behavior of predicting the next token. For all models in our experiment, we adopt the sampling method, which randomly selects the next token from the vocabulary dictionary based on the probability distribution. Here we further add an ablation study based on the beam search method. A beam of the top "beam size" sub-sequences with the highest generation probabilities is maintained and updated during the generation process. We conduct experiments using beam search method with a beam size 5 on RTLLM V1.1 for RTLCoder and open source baselines. The results are shown in Table 4. The accuracies of all methods drop after adopting beam search. RTLCoder is still superior to all the open-source baselines with beam search.

5 CONCLUSION

This work proposes a new LLM solution named RTLCoder for RTL code generation, achieving state-of-the-art performance in non-commercial solutions and outperforming GPT-3.5. We contribute a new data generation flow and a complete dataset with over 27 thousand labeled samples, addressing the serious data availability problem in hardware-design-related tasks. Also, we contribute a new training scheme based on design quality scoring. It greatly boosts the model performance. Importantly, RTLCoder will be fully open-sourced. RTLCoder's lightweight property and low hardware barrier allow anyone to easily replicate and further improve based on our existing solution. We expect more brilliant LLM-based solutions in this agile hardware design direction.

REFERENCES

[1] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2023. Fixing Hardware Security Bugs with Large Language Models. *arXiv preprint arXiv:2302.01215* (2023).

[2] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In *NeurIPs*.

[3] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. *arXiv preprint arXiv:2305.13243* (2023).

[4] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. 2023. ChipGPT: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019* (2023).

[5] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate Post-training Compression for Generative Pretrained Transformers. *arXiv preprint arXiv:2210.17323* (2022).

[6] Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, and Yingyan Lin. 2023. GPT4AIGChip: Towards Next-Generation AI Accelerator Design Automation via Large Language Models. *arXiv preprint arXiv:2309.10730* (2023).

[7] Zhuolun He, Haoyuan Wu, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. 2023. ChatEDA: A Large Language Model Powered Autonomous Agent for EDA. In *MLCAD Workshop*.

[8] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[9] Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. 2023. LLM-assisted Generation of Hardware Assertions. *arXiv preprint arXiv:2306.14027* (2023).

[10] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[11] Zhiding Liang, Jinglei Cheng, Rui Yang, Hang Ren, Zhixin Song, Di Wu, Xuehai Qian, Tongyang Li, and Yiyu Shi. 2023. Unleashing the Potential of LLMs for Quantum Computing: A Study in Quantum Architecture Design. *arXiv preprint arXiv:2307.08191* (2023).

[12] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. 2023. ChipNeMo: Domain-Adapted LLMs for Chip Design. *arXiv preprint arXiv:2311.00176* (2023).

[13] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. *arXiv preprint arXiv:2309.07544* (2023).

[14] Yixin Liu, Pengfei Liu, Dragomir Radev, and Graham Neubig. 2022. BRIO: Bringing order to abstractive summarization. *arXiv preprint arXiv:2203.16804* (2022).

[15] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2023. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model. *arXiv preprint arXiv:2308.05345* (2023).

[16] Madhav Nair, Rajat Sadhukhan, et al. 2023. Generating secure hardware using chatgpt resistant to cwes. *Cryptology ePrint Archive* (2023).

[17] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309* (2023).

[18] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[19] Martin Rapp, Hussam Amrouch, Yibo Lin, Bei Yu, David Z Pan, Marilyn Wolf, and Jörg Henkel. 2021. MLCAD: A survey of research in machine learning for CAD keynote paper. *IEEE TCAD* (2021).

[20] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD*.

[21] Synopsys. 2021. Design Compiler® RTL Synthesis. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-nxt.html.

[22] Synopsys. 2021. VCS® functional verification solution. https://www.synopsys.com/verification/simulation/vcs.html.

[23] Shinya Takamaeda-Yamazaki. 2015. Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In *Applied Reconfigurable Computing*.

[24] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2023. Benchmarking Large Language Models for Automated Verilog RTL Code Generation. In *DATE*.

[25] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. 2023. AutoChip: Automating HDL Generation Using LLM Feedback. *arXiv preprint arXiv:2311.04887* (2023).

[26] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120* (2023).

[27] Zheyu Yan, Yifan Qin, Xiaobo Sharon Hu, and Yiyu Shi. 2023. On the Viability of using LLMs for SW/HW Co-Design: An Example in Designing CiM DNN Accelerators. *arXiv preprint arXiv:2306.06923* (2023).