
Article:
Yang, K. orcid.org/0000-0002-2844-3810, Liu, H., Zhao, Y. et al. (1 more author) (2023) A new design approach of hardware implementation through natural language entry. IET Collaborative Intelligent Manufacturing, 5 (4). e12087. ISSN 2516-8398

https://doi.org/10.1049/cim2.12087

IET Collaborative Intelligent Manufacturing

Special issue Call for Papers

Be Seen. Be Cited.
Submit your work to a new IET special issue

Connect with researchers and experts in your field and share knowledge.

Be part of the latest research trends, faster.

Read more

The Institution of Engineering and Technology

Received: 6 July 2023 | Revised: 28 September 2023 | Accepted: 17 October 2023

IET Collaborative Intelligent Manufacturing

DOI: 10.1049/cim2.12087

The Institution of Engineering and Technology WILEY

ORIGINAL RESEARCH

A new design approach of hardware implementation through natural language entry

Kaiyuan Yang | **Haotian Liu** | **Yuqin Zhao** | **Tiantai Deng**

Department of Electronics and Electrical Engineering, The University of Sheffield, Mappin Street, Sheffield, UK

Correspondence

Yuqin Zhao.
Email: yzhao229@sheffield.ac.uk

Funding information

Royal Society, Grant/Award Number: IEC\NSFC \223008

Abstract

OpenAI's ChatGPT (GPT-4) ushers in a superior mode of computer interaction through natural language dialogues. Notably, it generates not only engaging dialogues but also codes aligned to queries and requirements. The potential of ChatGPT in hardware implementation via natural language is implemented and a strategy for "asking the right questions" is outlined. The versatility of ChatGPT is demonstrated through three mainstream hardware designs – systolic array, ResNet and MobileNet accelerators – comparing these with hand-coded designs. The evaluation metrics include design quality, design efforts, and limitations of code generated by ChatGPT/GPT-4/Cursor against prevalent High-Level Synthesis or hand-coded HDL designs. Consequently, a novel design workflow is proposed and the constraints of using GPT, particularly in AI accelerators, are highlighted.

KEYWORDS

computer integrated manufacturing, human computer interaction, human-robot interaction, intelligent manufacturing systems

1 | INTRODUCTION

Modern algorithms or applications usually require a huge amount of computing power to maintain real-time performance [1]. With this requirement, modern hardware tries to keep up with the demanding computing power from the algorithm and many academics and industries have developed new hardware architectures, for example, the NVidia A100/H100 GPU, FPGA-based accelerator and Google TPUs [2, 3]. These hardwares are usually designed using Hardware Description Language (HDL), which requires designers to have rich hardware knowledge, and only well-experienced hardware engineers can produce high-quality hardware designs. However, the modern hardware design is tightly combined with the requirements from the algorithms and application side, which means the design philosophy has shifted from hardware-oriented to applications-oriented. Hence, it is necessary to include algorithm/application developers as the designers of the hardware.

There are quite a lot of efforts that have been made to lower the bar of doing hardware design by generating HDL from High-Level Languages (HLL), which is the High-Level Synthesis (HLS) [4]. The HLS tools can take HLLs as input, for example, Vivado HLS (C/C++), MyHDL (Python), LabVIEW FPGA (LabVIEW) or HDL Coder (MatLab) [4–7]. These HLS tools significantly reduce the design effort by hiding some of the hardware complexity from the users and allowing users to use the HLL syntax for implementing the design. With HLS tools involved, the design level is moved from the RTL level to the HLL level.

If we look at the hardware design from the design-level perspective, we can clearly see that it has been moved from the switch level to the HLL level over the last several decades. The design level moves along with the available resources, fabrication of the hardware, and the design complexity. Let us take FPGA as an example. FPGA stands for Field Programmable Gate Array, which is a type of programmable logic that allows users to build hardware on it with limited resources. In the first stage (the age of invention) of FPGA (1984–1991) [8], there were very few logic resources, thus, the automatic synthesis, and place & route functions were not necessary on the design tool chain as users usually tried to reduce the resource

consumption by manually drawing schematic on the tool [9]. At this stage, some designs require multiple FPGAs, such as [10]. Thus, an automated tool for multi-FPGA partitioning was an important part of the tool. In the second stage (1992–1999), there were more available resources because of Moore's Law, optimal resource utilisation was less necessary, and helpful features for the design process became essential. It became possible to trade-off areas for performance and ease of use. The resulting devices were less silicon-efficient. In terms of design automation, FPGAs were becoming too large for manual design. In 1992, the flagship Xilinx XC4010 had 10,000 gates. By 1999, the number had risen to 1,000,000 in the Virtex XCV1000. Until then, automatic place and route were preferred, but not entirely trusted [8]. By the end of the 1990s, automated synthesis [11] and place and route [12] were required steps in the design flow. The low-level hardware design became more automatic than manual, and vendors tried to find abstractions for designing hardware on FPGAs since the design effort was gradually becoming more difficult. As FPGAs became more popular, Electronic Design Automation (EDA) companies became more interested in providing tools for them. In the third stage (2000–2007), FPGAs were common components of digital systems. The available resources and the complexity of designs both increased significantly [8]. At this stage, FPGAs often had more available resources than the application needed. Venders produced libraries of intellectual property cores (IP cores) for important functions. A valuable IP core was the soft processor (Xilinx MicroBlaze and Altera Nios) [13, 14]. The characteristics of designs changed in the 2000s. Large FPGAs enabled large designs that were complete subsystems. A significant change in this stage was the design level moving up to the RTL and behaviour level and away from the gate level. FPGA users were no longer working simply on implementing logic. Usually, FPGA designs primarily focused on communication standards for signals and protocols either to interface with an external system or to communicate among internal blocks. From 2008 to the present, FPGAs are no longer thought of as arrays of gates, but as collections of larger-scale functional blocks, integrated using programmable logic. They are still programmable but are not restricted to Program Logic (PL) and sometimes come equipped with an on-chip ARM processor (e.g., Xilinx Zynq series and Intel SoC series). When implementing an image processing system on FPGAs, the design effort and risks are emerging as critical project requirements. Very large image processing systems are difficult to design efficiently and require very detailed hardware knowledge to achieve high efficiency. To address this challenge, vendors have released their High-Level Synthesis (HLS) tools to shorten the design time. HLS tools have enabled the syntax of designs in this stage to move up from VHDL/Verilog to the C/C++ level. Applications are becoming more complex, given the available resources.

It is easy to observe from history that the design level has been continually pushed up from the gate level to the RTL level, then the behaviour level to the HLL level. Thus, given the current development of ChatGPT/GPT-4, it is now possible to push the design level from the HLL to natural languages through generative AI [15–17]. In this paper, we focus on analysing the possibility of using the natural language as the design entry for hardware design through generative AIs like ChatGPT/GPT-4.

The main contributions of this work are as follows:

(1) We propose a new way of doing hardware design using generative AI and choose ChatGPT/GPT-4 as an example to show the feasibility of this design entry.
(2) We analyse and evaluate the design quality of the hardware by generative AI by comparing them with existing hardware designs in terms of area consumption, performance and design efforts. Then, we come up with a strategy for asking the "right questions" to GPT-4 to generate a valid design.
(3) We provide 3 case studies that include a Systolic array and two AI accelerators to demonstrate the functionality of our strategy.
(4) We point out the challenges and future directions in using generative AI for designing hardware.

The rest of the paper is organised as follows: In Section 2, we introduce how the generative AI (GPT-4) can help us with the designing of the hardware and the detailed reasons why we need to push the design level urgently. In Section 3, we demonstrate our strategy of "asking the right question" to GPT-4 and discuss the design limitations and challenges of using GPT-4 to design and optimise hardware. In Section 4, we present the case studies: the Systolic Array and two AI accelerators for ResNet and MobileNet, comparing GPT-4 implementation with existing results using HLS or hand-coded HDL. Finally, we draw our conclusions in Section 5.

2 | AVAILABLE GPTS AND THEIR CAPACITY

2.1 | Available GPTs – ChatGPT, GPT-4 and others

ChatGPT [15] is an advanced language model developed by OpenAI. GPT stands for "Generative Pre-trained Transformer," which describes the structure and nature of the model. ChatGPT is designed to understand and generate human-like text based on the input it receives, making it capable of engaging in conversation, answering questions, and providing information on a wide range of topics.

The underlying technology of ChatGPT is based on deep learning, specifically using a type of neural network architecture called transformers. These networks are adept at handling sequential data and understanding the context of words and phrases within a sentence. ChatGPT is pre-trained on a large dataset of text from various sources, allowing it to have a wide knowledge base up until its cut-off date.

ChatGPT (GPT-3.5) is not specifically designed for programming tasks. However, it can provide guidance, suggestions, and explanations on various programming languages,

concepts, and best practices. According to itself, while it may not be able to write complex code or debug sophisticated programs, it can still assist users with general programming inquiries and help beginners understand key concepts.

GPT-4 [16], the fourth instalment in OpenAI's "GPT-n" series of generative pre-trained transformer models, is a highly advanced, multimodal language model. It was launched on March 14, 2023, and is available to the public in a restricted manner through ChatGPT Plus, a premium edition of ChatGPT.

Built on the transformer architecture, GPT-4 is pre-trained to predict subsequent tokens using a combination of publicly available data and data acquired through licensing agreements with third-party providers. The model undergoes fine-tuning through reinforcement learning, incorporating both human and AI feedback to ensure alignment with human values and adherence to policy guidelines.

There are also some other approaches to generate HDL coding using LLM, for example, Cursor [17], CoT [18], ChipGPT [19], and RTLLM [20]. All these approaches are based on ChatGPT and have some modifications to the original ChatGPT LLM.

Reference [17] is an AI-powered integrated development environment (IDE) that incorporates ChatGPT-style natural language processing capabilities with traditional programming tools. It aims to create a more interactive and intuitive coding experience by leveraging AI-assisted features. Cursor uses GPT-3.5 or GPT-4 to help developers with various programming tasks, making the development process more efficient and accessible. Some of its key features include:

(1) Coding Assistance: The cursor generates 10–150 lines of code using an AI that is more advanced than Copilot, providing intelligent code suggestions and completion.
(2) Check Diffs: Developers can ask the AI to edit a block of code and review the proposed changes, allowing them to assess and accept or reject modifications more efficiently.
(3) One-click Import: Cursor is a fork of VSCode and bringing over your extensions, settings, and keybindings is a breeze [17].

CoT [18], an LLM HDL automatic generation especially for Communication System applications on the FPGA, is applying ChatGPT to generate a simple FFT function. However, although this method is approachable, it still requires deep knowledge of hardware to solve the timing error during the design. Meanwhile, more and more AI architectures are being applied in the communication system to enhance its performance; therefore, it is necessary to have AI architectures, such as RES-18, mobile net, and Systolic structures, which are done in this work as example applications of a communication System. Only the FFT function is not sufficient.

ChipGPT [19] is an approach established on the ChatGPT, which could provide designers better performance and performance-power efficiency compared with other approaches like pure GPT or HLS, etc. However, it widens the gap between the non-hardware engineers and hardware realm as it requires users to give detailed hardware specifications and descriptions like I/O pair and interface information. It runs in opposite directions of ChatGPT. Moreover, ChatGPT is a continuously developing LLM, which will provide users with HDL coding with less and less hardware knowledge required. Therefore, ChipGPT could only be regarded as a transitional HDL code generation approach.

RTLLM [20] has similar characteristics as ChipGPT. Unfortunately, RTLLM is even worse than ChipGPT when it is used by non-hardware engineers. It requires users to input a designed RTL file (.v), a testbench file (.v), and a function description file. This approach is more suitable for hardware design optimisation rather than new hardware code generation. When non-hardware engineers chose this approach to generate HDL coding, they did not have an example of HDL coding. Furthermore, from the comparison table provided in the paper, ChatGPT-family-generated coding accounts for 16 better aspects, whereas RTLLM-generated coding only counts 9. This has proved that RTLLM is less powerful and has a wider gap between hardware and non-hardware engineers compared with pure ChatGPT. Also, same as ChipGPT, this RTLLM will only be a transitional HDL code generation approach as ChatGPT develops further.

In a word to sum up, all these approaches will be gradually disused as the ChatGPT develops. At this stage, these approaches are usable, but they require users to have adequate hardware knowledge still. It is the opposite direction compared with ChatGPT, which they are based on. Moreover, not all of them could provide a much better design than the pure-ChatGPT. Therefore, it is more suitable to use pure ChatGPT as an HDL code generation tool in the aspect of performance and future prospect.

2.2 | Design capacity of GPTs

We evaluate the design capacity of GPTs by the following factors: design speed, supporting languages, testing, continuous development and processing records.

(1) Speed: GPTs operate at a speed that is significantly faster than human researchers, particularly those who are just starting out.
(2) Language Diversity: GPTs support a variety of hardware description languages, including Verilog, VHDL, and High-Level Synthesis (HLS).
(3) Testbench Generation: GPTs are capable of generating a testbench directly based on the module provided by the user. This enables quick and easy testing of the design.
(4) Continuous Development: GPTs are continually developing and learning from user input as well as gathering information from the Internet. This leads to a constantly improving performance and knowledge base.
(5) Processing Records: Users can readily access records of previous development processes. This can be valuable for reference, further development, and learning.

2.3 | What we need versus what GPTs provide

The FPGA design involves designs for different parts as modern FPGAs are usually associated with an ARM processor. The initial point of FPGA design is usually the Hardware Description Language (HDL), such as VHDL, Verilog, and System Verilog. Then, according to the processor attached to the FPGA programmable logic, there might be a need to use C/C++ to implement the software part. Functional simulations are then performed to verify the design corrections. Further, the design is synthesised into a gate-level representation and optimised for performance and resource utilisation. Also, the design is placed and routed to the FPGA device, following which a static timing analysis is performed to ensure that timing constraints are satisfied.

GPTs can help us generate hardware description languages with defined design requirements and specifications [21]. They can also help resolve timing conflicts and optimise performance as well as resource utilisation. However, the hardware description languages they generate contain occasional inaccuracies. For instance, there may be instances of inappropriate use of the for-loop construct in the code. Furthermore, GPTs have limited efficacy in optimising performance and resource utilisation. They can only execute all the given functions if the instructions provided are sufficiently clear and detailed or through many patient and detailed instructions. Based on the design requirements for modern FPGA-based systems, GPT cannot generate the software part as it relies on the massive information that comes from the hardware, including the address, driver, specialised header files (e.g. xparameters.h) and non-standard functions (e.g. xil_printf instead of printf) [22].

2.4 | Design-level evolution – natural language entry

FPGA hardware design has undergone a remarkable evolution, beginning with manual design, transitioning to HDL and HLS, and now venturing into the realm of natural language interfaces [23, 24].

In the early stages, before the HDL, developers usually use manual design and manual place and route for FPGA designs, because the resources were limited and precious [23]. In addition, the available resources are usually several hundreds of logic blocks and it is possible to manually implement FPGA-based systems.

The introduction of hardware description languages (HDLs) like VHDL or Verilog was a step forward in making FPGA design more manageable and somewhat more abstract [24]. Even though it supports the behaviour-level design, users still have to think in terms of the hardware.

Over time, high-level synthesis (HLS) tools have the revolutionised FPGA design. These tools allow for hardware designs to be described in high-level languages, such as C or C++, making the process more accessible and significantly reducing the length of code [25]. By abstracting from the hardware's intricacies, designers can work faster, manage complexity better, and operate in an environment closer to HLL, enhancing both productivity and approachability in the field.

The most recent progression in the FPGA hardware design involves natural language entry interfaces, enabled by advanced AI models like ChatGPT. These systems translate natural language commands into the functional hardware design code, eliminating the need for specific coding knowledge and detailed hardware knowledge [26]. This innovative approach democratises hardware design further, making it accessible to a broader user base [26]. Users can now "converse" with the system, detailing what they want the design to accomplish, and the AI model generates the corresponding code. This not only simplifies the design process but also opens up the field of FPGA hardware design to individuals who may have creative ideas but lack the necessary technical expertise.

3 | GENERATE VALID CODE USING GPTS

3.1 | The strategy of "asking the right questions"

Through the extensive interaction and systematic experimentation with GPT, our team has distilled a strategic methodology that elucidates how to frame precise enquiries to GPT, enabling it to formulate comprehensive hardware designs effectively.

(1) Elucidating the Desired HDL Code Type:

Prior to initiating an interaction, it is imperative to meticulously specify the type of HDL code that one desires to generate. A clear delineation of whether the required code is in Verilog, VHDL, or another HDLs is crucial to aligning the subsequent output with the intended design objectives.

(2) Unveiling the Structure of the Envisioned Module:

A concise yet comprehensive introduction of the envisaged module's structure is paramount. The delineation of the structural elements, including the interface definitions, internal states, and variable declarations, provides a foundational framework upon which GPT can construct the detailed code segments.

(3) Detailing Distinct Module Functions:

It is equally vital to methodically introduce the functions of the module, addressing each one individually. For each function, one should furnish sufficient detail, elucidating the role, the expected input and output, and any intricate logic or computations involved. By decomposing the module into its constituent functions and detailing their intricacies, users can guide GPT in crafting nuanced and accurate functional blocks.

(4) Adhering to Specific Formatting Conventions:

In circumstances where the generated code must conform to formatting conventions or styles, users are advised to incorporate adequate explanatory notes and illustrations. Presenting contextual examples, accompanied by succinct explanations, facilitates a clearer understanding for GPT, allowing it to generate codes that are congruent with the specified formatting and stylistic norms.

By adhering to this nuanced strategy, users can significantly enhance the precision and relevance of the generated hardware design, thereby achieving a more harmonious synergy between human intent and machine-generated output. This refined approach ensures that GPT understands the requisites and nuances of the task at hand, culminating in outputs that are coherent, logically sound, and well-aligned with the user's design objectives.

3.2 | Design limits of GPT

The first limitation of using GPT to generate a hardware design is the limited domain-specific knowledge. For some algorithms, GPT will generate incorrect codes, which need more detailed descriptions and orders. Since GPT is a generative AI tool, it could only pop out based on training data. Therefore, in some specific fields where the information is limited in its training process, GPT sometimes could generate codes with syntax errors and users have to point it out to prevent GPT from continuously generating code with syntax errors.

The second limitation is that the GPT does not understand the coding style of hardware design and the philosophy of hardware design. For example, for the For Loop, GPT normally could not generate the correct For Loop until the user gives an order. It requires really detailed instructions like "use always and counter to replace For Loop. As mentioned before, GPT's outputs are more based on its training data". Verilog coding is totally different from the software because it focuses on how the hardware is built and organised. Therefore, it is not available to generate correct Verilog coding by purely mimicking software codings. For example, GPT prefers to generate software-style For Loop in Verilog coding even if it is given the order to stop using it. For Loop sometimes could lead to iterative hardware and large resource consumption in hardware designs. Moreover, GPT has no idea on how to proceed with parallel computing in Verilog. Researchers at least can read Verilog code to revise codes manually or give instructions in detail.

The third limitation of GPT is that it may sometimes generate code with syntax errors. This issue sometimes aligns with the software-style coding, for example, the declaration of variables inside a look or an always block or using some functions in software only. GPT prefers to declare reg or wire only before the line where they need to be used rather than do it at the beginning. This could lead to a syntax error of "declaration inside should be out of unnamed block". Moreover, GPT sometimes uses the System Verilog code format for Verilog array definition. GPT-generated array-containing codes always lead to syntax errors or define wrongly into a memory type. Although System Verilog and Verilog have been merged since 2018, some FPGA design tools still do not support this and this will lead to a syntax error.

The fourth limitation of GPT is the synthesis errors it may bring to the code. For example, it may generate code over the limitation of 1000000 input or output or an empty design. There are some limitations in Verilog such as input/output number limitation required by Xilinx Vivado. For a large module like the convolution layer, the number of inputs or outputs always exceed the limits of 1000000, which leads to a synthesis error. Furthermore, GPT sometimes will define reg or wire randomly, which could lead to synthesis errors.

The fifth limitation of GPT is that the GPT does not understand logic and hardware. When GPT receives instructions specifically based on hardware, sometimes it could not fully understand it and generates codes differently from the expectations. There are still knowledge gaps between the instructions and GPT; therefore, some keywords and patterns need to be summarised.

The sixth limitation of GPS is the confusion of the code formats. GPT-generated code sometimes ignores the code formats of Verilog and SystemVerilog. This causes the generated Verilog code to be more formatted closer to the system Verilog format. However, the code generated by GPT has randomness. Thus, this statement is not absolute.

The last limitation of GPT is forgetfulness. GPT cannot remember all the details in a conversion task. If users do not emphasise the previous content, GPT might forget some important details, such as ports size and module's name, in the code generated later. Therefore, when you need GPT to modify the code, it is recommended to ask questions in the form of the code with the modified content.

3.3 | Challenges of using GPT as the design entry

The first challenge of using GPT to generate hardware design is large resource consumption. Compared with the hand-coded hardware design, the first GPT-generated design always consumes more resources and power consumption. Therefore, GPT-generated codes need domain-specific knowledge to optimise either by giving GPT instructions or manually.

The second challenge of GPT is the hardware knowledge gap. Due to the limitation of training data, there are still some gaps between GPT and hardware design. These gaps could lead to synthesis errors, syntax errors and limitations for optimisations.

The third challenge of GPT is misunderstanding caused by the knowledge gap. As GPT lacks hardware domain-specific knowledge, sometimes it will generate Verilog codes, which do not meet the given requirements. In other words, it will misunderstand the instructions and use software philosophy to

generate hardware codes. Therefore, researchers need to check the functionality by Testbench (input-output relationship equation) or by their knowledge and experience.

4 | CASE STUDY

To evaluate the design capacity of GPT, we choose 3 case studies, which are the current mainstream and benchmarking designs, including the systolic array, accelerator for ResNet and accelerator for MobileNet. The systolic array is the architecture used in many AI chips, for example, DaVinci from Huawei, TPU from Google and NVidia A100 architecture. ResNet and MobileNet are the networks that are used as the benchmark to evaluate the capacity of the hardware accelerator.

4.1 | Systolic array

The convolution occupied the main operation in the Convolution Neural Networks (CNNs) [27]. Researchers have been exploring the possible potential of systolic arrays by pipelined parallel computing for hardware acceleration of convolutions. In this work, we attempt to employ GPT-4 to replicate the fundamental systolic array architecture unit processed by Cao et al. [28]. This attempt can achieve a similar function to the original design.

Figure 1 illustrates the internal structure of the processing element (PE) in the systolic array. It has a multiplier, an adder and three registers to store the weight, feature map and partial sum (PSUM). The PE consists of four mode constants:

1. INIT_MODE: achieve data initialisation
2. WEIGHT_TRANSFER_MODE: weight transmittance
3. CALC_MODE: multiplication and accumulation operations
4. WAIT_MODE: waiting for operations

Figure 2 presents the 8 rows and 4 columns of systolic arrays, consistent with the verification design by Cao et al. [28]. The feature would transfer along the row lines, and the weight and PSUM would transfer along the column lines. The buffer

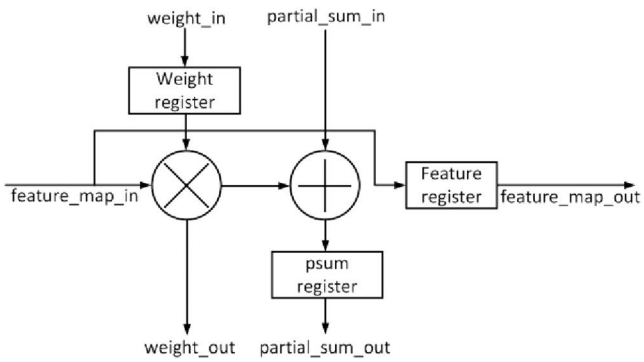

FIGURE 1 The internal structure of the processing element.

module operates three FIFO storages to reproduce the basic function of dual FIFO storage structure as shown in Figure 3. The FeatureMapFIFO and WeightFIFO control the feature and weight input of PEs. The PartialSumFIFO stores the calculation result of PEs.

We evaluated our design on the Xilinx Zedboard. Table 1 presents the hardware utilisation of our systolic array, which uses 32 and 128 DSP operations. As the GPT cannot automatically optimise the internal PE structure, achieving the same DSP utilisation as Cao's design. We used matrix multiplication as verification. The data requirements were significantly lower than those of LeNet-5 as Cao's design. Thus, the BRAM, LUT and FF are less applicable in this work.

The model was built based on 47 valid conversations with GPT and partial manual modifications. Among them, the PE module, Systolic Array module, FIFO-related module, and TOP module occupied 5, 5, 19, and 18 conversations separately. In this process, interesting phenomena were observed, generating preliminary insights.

(1) GPT can be used to generate simple and regular basic building blocks with sufficient details and reasonable descriptions. For instance, it can provide highly accurate

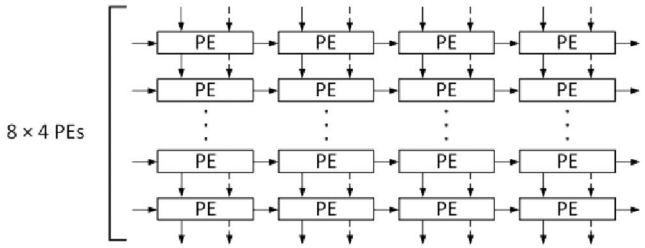

FIGURE 2 The structure of the systolic array.

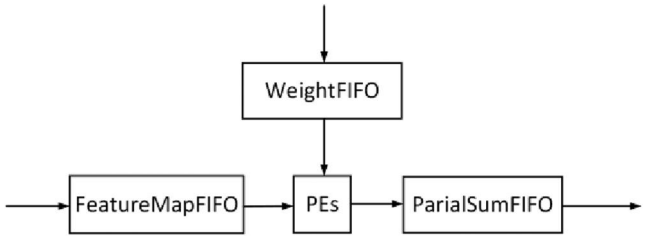

FIGURE 3 The connection between the systolic array and FIFO storage.

TABLE 1 FPGA resource utilisation.

	[28]	Ours (int_16)	Ours (int_32)
FPGA	Zedboard	Zedboard	Zedboard
DSP	64	32	128
BRAM	69	6	12
LUT	28861	2774	6710
FF	41828	3336	6408

codes when given clear function and data flow rules with PE and systolic array architectures.
(2) GPT can implement simple modifications of code. For further optimisation, the optimal approach involves providing an example of how to optimise the code.
(3) GPT might forget details of conversations that happened long ago. For example, it forget the port size of the SystolicArray8 × 4 module, which had previously been generated.
(4) GPT can solve the error statement from Quartus and Vivado. However, GPT might be trapped in it and forget the code's original purpose, when modifying the same code many times. The optimal solution involves finding the errors in the code manually and fixing them by ourselves.
(5) GPT generates a random code style for Verilog. In this work, the code style is biased towards System Verilog. However, in another attempt, it can generate in the standard Verilog format code.

4.2 | Accelerator for ResNet

ResNet is one of the most common-used Deep Learning architectures nowadays [29, 30]. The structure of a ResNet with 18 layers is shown in Figure 4. In Figure 4, the solid line indicates the directly used identified shortcut calculation as shown in Equation (1), whereas the dotted line indicates a non-matched dimension shortcut. In a non-matched dimension condition, we could choose either the extra 0s mapping method or Equation (2) to proceed with the shortcut step [29].

$$Y = \mathcal{F}(x, \{W_i\}) + x \tag{1}$$

$$Y = \mathcal{F}(x, \{W_i\}) + W_s x \tag{2}$$

In Equations (1) and (2), $\mathcal{F}(x, \{W_i\})$ indicates residual mapping to be learned, while W_s indicates a linear projection.

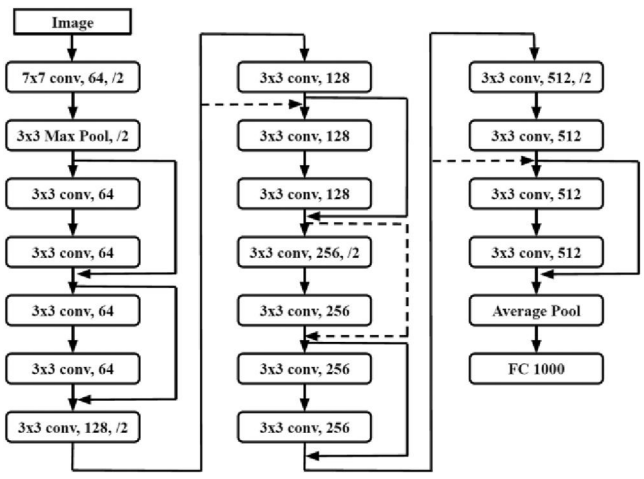

FIGURE 4 Structure of a ResNet 18 architecture.

Table 2 indicates the summary of layers used in ResNet 18. A ResNet 18 architecture includes one 7 × 7 64-channel (stride 2) convolution layer, one 3 × 3 max pool (stride 2) layer, one average pool with 1000-d fc layer, four 3 × 3 64-channel convolution layers (stride 1) and four 3 × 3 (3 for stride 1 and 1 for stride 2) with the channel numbers of 128, 256, and 512, respectively.

Through the Verilog codes generation process, GPT has imposed the following problems:

(1) GPT prefers to use For Loop, which is not commonly used in Verilog and could lead to unnecessary resource consumption under some circumstances.
(2) GPT-generated array definition will always disobey the synthesis rules in Vivado.
(3) There is a limit to the input and output numbers in Vivado synthesis, which GPT does not know. Therefore, for large designs like the convolution layer, it will always exceed the limit.
(4) GPT sometimes declares a register or wire in an unnamed block, which could also lead to a synthesis error.

Codes generated by GPT-4 are synthesised by Vivado 2019.2. The resource usage is shown in Table 3. Since Vivado 2019.2 has a limitation of 1,000,000 on the number of inputs and output, for most convolution layers, only single-channel design is synthesisable. In Table 3, all the resource consumption refers to single-channel layers except Average Pool (input/output number is still under the limitation).

Table 3 shows that the GPT-4-generated codes consume a large quantity of LUT, especially for the convolution layer. For a single channel 3 × 3 convolution layer, the LUT usage is even more than the whole ResNet design [30]. Also, for other resources like FF usage in the FC layer, the consumption has already reached the limitation of the implementation FPGA board. If we calculate the resource usage among the whole design, in which all the layers are multi-channel, it will consume multiple time resources more than Zedboard could provide. Adding up the resources of the individual models, it is clear that their resource consumption will far exceed that of the same ResNet 18 design in ref. [30].

Compared with the ResNet 18 design in refs. [30, 31], we could see that both refs. [30, 31] consume a lot of BRAM in the design, whereas in the ChatGPT design, there is no BRAM usage. Moreover, in the aspect of DSP usage, this work consumes much less than in ref. [31]. On the contrary, our design utilises a lot of LUT compared with refs. [30, 31] as a cost.

4.3 | Accelerator for MobileNet

MobileNet is a family of convolutional neural networks (CNNs) developed by Google researchers for efficient execution on mobile and other low-power devices [32]. The architecture of MobileNet is based on depthwise separable convolutions, a form of factorised convolutions, which significantly reduces the computational cost compared to

TABLE 2 Layer summary of a ResNet 18 architecture.

Layer	Number of layers	Output size
7 × 7, 64, stride 2	1	112 × 112
3 × 3 max pool, stride 2	1	(After Conv stride 1 layer) 56 × 56
Conv, 3 × 3, 64, stride 1	4	
Conv, 3 × 3, 128, stride 2	1	(After Conv stride 1 layer) 28 × 28
Conv, 3 × 3, 128, stride 1	3	
Conv, 3 × 3, 256, stride 2	1	(After Conv stride 1 layer) 14 × 14
Conv, 3 × 3, 256, stride 1	3	
Conv, 3 × 3, 512, stride 2	1	(After Conv stride 1 layer) 7 × 7
Conv, 3 × 3, 512, stride 1	3	
Average pool, 1000-d fc, softmax	1	1 × 1
FLOPS	1.8×10^9	

TABLE 3 Resource usage of ResNet 18.

	Input size	LUT	FF	DSP	BRAM	BUFG
FPGA board	NA	53200	106400	220	280	32
Max pool 3 × 3/2	112 × 112	213265	25220	6	NA	1
	8 bits					
Conv 3 × 3	3072	1005265	23328	NA	NA	1
	8 bits					
Average pool 512 channel	7 × 7	107773	8464	NA	NA	1
	16 bits					
FC layer	512	139992	106400	NA	NA	1
	16 bits					
ResNet 18 in [30]	224 × 224	596081	1175373	NA	30854	NA
ResNet 18 in [31] (HLS)	95% of 4-bit weights and 5% of 8-bit weights	180100	NA	2092	440.5	NA

standard convolutions while maintaining similar performance, so they can provide good performance while keeping the computational resources required as low as possible [32].

FPGA hardware accelerators also can greatly enhance the performance of MobileNet by providing a platform for highly parallel computation and by reducing the data transfer latency that is often a bottleneck in GPU-based acceleration. This allows the low-power, high-performance characteristics of MobileNet to be fully leveraged [32].

Figure 5 depicts the structural layout of the MobileNet-V2 CNN. In this structure, the layers are classified into two types of convolutional blocks (Conv_block), each characterised by distinct stride values.

There are many research studies on MobileNet-based FPGA hardware acceleration designs, from which we chose "Generating Efficient FPGA-based CNN Accelerators from High-Level Descriptions" by Ali et al. [33], and reproduced it via GPT-4.

This article is validated using a new sample evaluation CNN to produce an optimised accelerator. The evaluation CNN, described in Figure 6, comprises six layers, which are: 3 × 3 and 5 × 5 2D convolution, 3 × 3 and 5 × 5 pooling and 2 fully connected layers [33]. For this architecture, we generated each module separately using GPT-4 and tested its hardware resource consumption.

During code generation using GPT-4, the same problem occurs as in the ResNet 18 case. Codes generated by GPT-4 are synthesised by Vivado 2019.2 with Zedboard Zync 7000 configuration. The resource usage of conv1_5 × 5 and pool1_3 × 3 modules is shown in Table 4.

With Table 4 for those two modules, GPT-4 produces code that is close to the results in [33]. This shows that GPT-4 has an advantage in generating modules with small numbers of code lines. Combined with the problems mentioned above when generating codes, the use of GPT for hardware project development also requires manual adjustments and modifications.

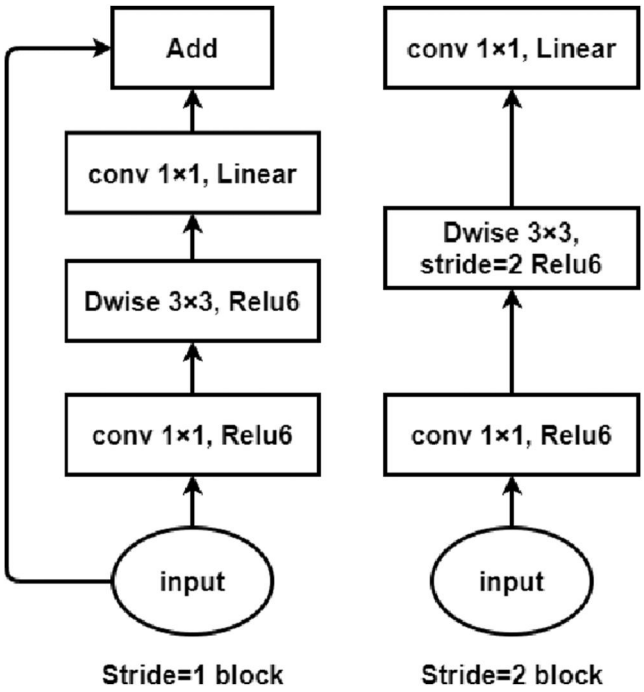

FIGURE 5 Structural layout of MobileNet-V2 [32].

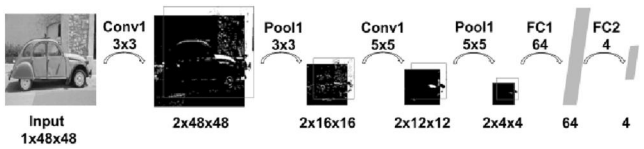

FIGURE 6 Details of the layers of the evaluation CNN [33].

TABLE 4 Resource usage of MobileNet.

Utilisation	LUT	FF	DSP	BRAM
conv1_5 × 5 GPT-4	0.3665	0.0182	0	0
conv1_5 × 5 [33]	0.191	0.109	0.25	0
pool1_3 × 3 GPT-4	0.0085	0.0021	0	0
pool1_3 × 3 [33]	0.104	0.057	0	0

5 | CONCLUSION

In this study, we explored the feasibility and challenges of using the large language model, GPT-4, in generating hardware description language codes for the development of FPGA accelerators for deep learning networks. Through a detailed analysis of its application in three case studies, we unveiled its potential and limitations.

The results highlighted the capacity of GPT-4 to produce valid and functional Verilog code blocks that could be synthesised using Xilinx's Vivado. It was found to be efficient in generating simple, regular basic building blocks with sufficient detail and reasonable descriptions when given clear functions and data flow rules. This capability, however, was accompanied by several challenges that necessitated manual intervention.

Among the problems encountered were the excessive usage of For Loops in Verilog code, non-compliance with synthesis rules in Vivado, exceeding the limit of input or output number in Vivado synthesis, and erroneous placement of registers or wires. Furthermore, GPT-4 showed a tendency to forget previous conversation details, a crucial aspect in the generation of code blocks.

When dealing with large architectures, such as ResNet-18 and MobileNet, it was noted that GPT-4 generated codes led to significant resource usage on the FPGA, often exceeding the available resources. This limitation posed a significant challenge in the practical application of GPT-4 for complex hardware designs. However, GPT performs better when writing modules with fewer lines of code.

In conclusion, despite the identified limitations, the results of this study suggest that GPT-4 holds promise as a tool for hardware development. It can effectively generate simple, individual hardware description language codes, but struggles with complex, large-scale architectures, which may require substantial manual adjustments and optimisation. Further research is needed to improve the model's capability to understand and manage hardware resources and remember conversation details. This could potentially lead to the more efficient use of GPT-4 in hardware accelerator development, thereby enabling an automated and streamlined hardware design process for deep learning architectures.

AUTHOR CONTRIBUTIONS

Kaiyuan Yang: Data curation; Formal analysis; Project administration; Validation; Writing – original draft; Writing – review and editing. **Haotian Liu**: Data curation; Formal analysis; Software; Validation; Writing – original draft; Writing – review and editing. **Yuqin Zhao**: Data curation; Formal analysis; Resources; Software; Validation; Writing – original draft; Writing – review and editing. **Tiantai Deng**: Data curation; Funding acquisition; Resources; Supervision; Validation; Writing – review and editing.

CONFLICT OF INTEREST STATEMENT

The authors declare no conflicts of interest.

DATA AVAILABILITY STATEMENT

All the source-code and chat history to the ChatGPT could be made available to the public once the paper is published.

ORCID

Kaiyuan Yang https://orcid.org/0000-0002-2844-3810
Tiantai Deng https://orcid.org/0000-0003-4507-5746

REFERENCES

1. Synopsys – what Is an AI Accelerator. https://www.synopsys.com/ai/what-is-an-ai-accelerator.html. [accessed 08/03/2023]
2. NVidia – Cloud & Data Center NVDLA. https://www.nvidia.com/en-us/data-center/a100/. accessed 28 03 2023
3. Venkataramani, S., et al.: RaPiD: AI accelerator for ultra-low precision training and inference. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp. 153–166. IEEE (2021)

4. Winterstein, F., et al.: High-level synthesis of dynamic data structures: a case study using Vivado HLS. In: 2013 International Conference on Field-Programmable Technology (FPT), pp. 362–365. IEEE (2013)
5. Decaluwe, J.: MyHDL: a python-based hardware description language. Linux J. 127, 5 (2004)
6. Kehtarnavaz, N., Mahotra, S.: Digital Signal Processing Laboratory: LabVIEW-Based FPGA Implementation. Universal-Publishers (2010)
7. Kintali, K., Gu, Y.: Model-based Design with Simulink, HDL Coder, and Xilinx System Generator for DSP, pp. 1–15. MathWorks, Inc (2012)
8. Trimberger, S.M.S.: Three ages of FPGAs: a retrospective on the first thirty years of FPGA technology. IEEE Solid-State Circ. Mag. 10(2), 16–29 (2018). https://doi.org/10.1109/mssc.2018.2822862
9. Von Herzen, B.: Signal processing at 250 MHz using high-performance FPGA's, Proc. ACM fifth international symposium on Field-programmable gate arrays. ACM, 62 (1997)
10. Vuillemin, J.E., et al.: Programmable active memories: reconfigurable systems come of age. IEEE Trans. Very Large Scale Integr. Syst. 4(1), 56–69 (1996). https://doi.org/10.1109/92.486081
11. Cong, J., Ding, Y.: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs, the Best of ICCAD, pp. 235–248. Springer, Boston (2003)
12. McMurchie, L., Ebeling, C.: PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs, Reconfigurable Computing, pp. 365–381. Morgan Kaufmann (2008)
13. Altera: 5 Reasons to Switch from SOPC Builder to Qsys. Altera (2012)
14. Xilinx: Mircroblaze Processor Reference Guide. Xilinx (2014)
15. ChatGPT. https://openai.com/blog/chatgpt. accessed 28 03 2023
16. GPT-4. https://openai.com/research/gpt-4. accessed: 28 03 2023
17. Cursor. https://www.cursor.so/. accessed: 28 03 2023
18. Du, Y., et al.: The Power of Large Language Models for Wireless Communication System Development: A Case Study on FPGA Platforms (2023). arXiv preprint arXiv:2307.07319
19. Chang, K., et al.: ChipGPT: How Far Are We from Natural Language Hardware Design (2023). arXiv preprint arXiv:2305.14019
20. Lu, Y., et al.: RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model (2023). arXiv preprint arXiv:2308.05345
21. Smith, A.: Exploring the use of AI in FPGA-based systems design: a study on hardware Description Language. IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 42(7), 1400–1410 (2023)
22. Doe, J.: Limitations and potential of generative pretrained transformers for hardware Description Language generation. IEEE Trans. Very Large Scale Integr. Syst. 31(6), 1005–1015 (2023)
23. Doe, J.: The evolution of FPGA design: from manual to HDL. IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 41(5), 600–610 (2023)
24. Smith, A.: HDL and HLS: stepping stones in FPGA design. IEEE Trans. Very Large Scale Integr. Syst. 30(9), 900–910 (2023)
25. Johnson, K.: The rise and impact of high-level synthesis in FPGA design. IEEE Design & Test 40(2), 120–130 (2023)
26. White, L.: AI and Natural Language processing: a new era for FPGA design. IEEE Trans. Comput. 72(3), 400–410 (2023)
27. Devaraddi, V.S., Rao, N.: An FPGA based tiled systolic array generator to accelerate CNNs. In: Proceedings of the 2022 25th Euromicro Conference on Digital System Design (DSD), pp. 316–323 (2022). https://doi.org/10.1109/DSD57027.2022.00050
28. Cao, Y., et al.: FPGA-based accelerator for convolution operations. In: Proceedings of the 2019 IEEE International Conference on Signal, Information and Data Processing (ICSIDP), pp. 1–5 (2019). https://doi.org/10.1109/ICSIDP47821.2019.9172934
29. He, K., et al.: Deep Residual Learning for Image Recognition (2015). arXiv preprint arXiv:1512.03385
30. Baskin, C., et al.: Streaming architecture for large-scale quantized neural networks on an FPGA-based dataflow platform. In: Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (2018). https://doi.org/10.1109/IPDPSW.2018.00032
31. Sun, M., et al.: FILM-QNN: efficient FPGA acceleration of deep neural networks with intra-layer, mixed-precision quantization. In: Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 134–145. Association for Computing Machinery (2022)
32. Howard, A.G., et al.: MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications (2017). arXiv preprint arXiv:1704.04861
33. Ali, N., et al.: Generating efficient FPGA-based CNN accelerators from high-level descriptions. J Sign Process Syst 94(10), 945–960 (2022). https://doi.org/10.1007/s11265-022-01797-w

How to cite this article: Yang, K., et al.: A new design approach of hardware implementation through natural language entry. IET Collab. Intell. Manuf. e12087 (2023). https://doi.org/10.1049/cim2.12087