

An Empirical Comparison of LLM-based Hardware Design and High-level Synthesis

Anonymous Author(s)

ABSTRACT

Field-Programmable Gate Arrays (FPGAs) are increasingly used for accelerating diverse applications due to their reconfigurability and ability to implement custom hardware architectures. However, programming FPGAs remains challenging, traditionally relying on low-level Hardware Description Languages (HDLs) like Verilog, which are intricate and time-consuming. High-Level Synthesis (HLS) tools, such as Vitis HLS, have emerged to address these issues by allowing hardware functionality description in high-level languages like C/C++, but they come with their own limitations, including less efficient hardware implementations, delay overhead caused by conservative scheduling strategies, and unpredictable solutions due to semantic differences between software and hardware.

This paper explores the potential of Large Language Models (LLMs) in FPGA design, particularly for generating complex Verilog kernels. We present a novel approach that guides LLMs to generate synthesizable and efficient Verilog code for complex FPGA kernels. Our method addresses key challenges in LLM-based hardware design. Through a case study on the PolyBench suite, we demonstrate that our LLM-guided approach can generate HDL implementations that surpass HLS tools in performance and resource utilization. The experimental results show that our approach reduces latency by an average of 28.88% compared to HLS, with a maximum reduction of 66.94% in pipelined designs and an average reduction of 8.41%, peaking at 55.76% in sequential stages. Furthermore, it decreases LUT usage by an average of 25.18% and flip-flop usage by 57.23% compared to HLS.

1 INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) have emerged as powerful platforms for accelerating a wide range of applications across domains such as artificial intelligence, high-performance computing, and data processing. [7, 28] Their reconfigurability and ability to implement custom hardware architectures make them ideal for adopting cutting-edge microarchitectural technologies to meet the demands of emerging applications. However, despite their effectiveness, programming FPGAs remains a significant challenge, hindering their wider adoption and utilization.

Traditionally, FPGA design has relied on Hardware Description Languages (HDLs) such as Verilog and VHDL. HDLs are praised for their ability to provide circuit representations that closely mimic inherent hardware structures, offering several advantages. They generally produce high-quality results in terms of performance and resource usage, allow fine-grained control over hardware implementation details, and give designers a deep understanding of the underlying hardware architecture. However, the low-level nature of HDL programming requires designers to manage intricate details of hardware implementation, leading to lower productivity due to longer development times and a more complex, time-consuming

verification process. It also demands in-depth hardware design expertise, making it challenging to explore different architectural options quickly. Additionally, maintaining and modifying HDL designs can be more difficult compared to higher-level approaches.

In response to these challenges, High-Level Synthesis (HLS) tools [3, 8, 22, 27], such as Vitis HLS [25], have emerged as an alternative approach, attempting to raise the abstraction level of hardware design to the software domain. HLS allows designers to use high-level programming languages like C or C++ to describe hardware functionality, which is then automatically translated into Register-Transfer Level (RTL) representations. It significantly increases productivity, and reduce the learning curve for FPGA programming compared to HDLs programming. Besides, it enables faster design space exploration through tool directives and constraints, and simplifies the verification process by often allowing the use of software-level verification tools. It's more accessible to software engineers, lowering the barrier for hardware design, and facilitates faster prototyping and shorter time-to-market [11, 12]. HLS also makes it easier to target new platforms or adapt to changing requirements, and reduces the likelihood of introducing bugs due to its higher abstraction level.

However, HLS is not without its drawbacks [14]. The quality of HLS-generated designs often lags behind hand-crafted RTL in terms of performance and resource usage, and designers have less direct control over hardware implementation details, which can be obfuscated. HLS is highly tool-dependent, with results varying significantly between different tools, and it may struggle with certain types of algorithms, especially those with dynamic behaviors. Additionally, HLS requires C/C++ code to be written in a specific, synthesizable style, which can lack hardware awareness and lead to challenges in exploiting FPGA parallelism, resulting in less efficient hardware implementations compared to hand-coded RTL. Furthermore, the semantic differences between software and hardware domains can sometimes lead to unpredictable or suboptimal solutions.

Given these challenges in both HDL and HLS approaches, there is a growing interest in exploring alternative methodologies for FPGA design [4, 21, 26]. One such promising avenue is the application of Large Language Models (LLMs) to hardware description languages [5, 16–18]. LLMs, which have demonstrated remarkable capabilities in natural language processing and code generation tasks, present an intriguing possibility for bridging the gap between high-level design intent and low-level hardware implementation. The potential of LLMs in FPGA design lies in their ability to understand and generate code based on natural language descriptions or high-level specifications. This approach could potentially combine the high-level abstraction benefits of HLS with the fine-grained control and efficiency of HDLs. By leveraging the contextual understanding and pattern recognition capabilities of LLMs, designers could potentially express their design intent in natural language

117 or high-level pseudocode, which the LLM could then translate into
 118 efficient and user-friendly HDL code.

119 However, applying LLMs to hardware design, particularly for
 120 complex FPGA kernels, presents its own set of challenges and opportu-
 121 nities that warrant thorough investigation. The primary obstacle
 122 is the scarcity of high-quality HDL datasets, which severely con-
 123 strains the capability of LLMs to generate accurate and efficient
 124 code. This limitation stems from the specialized and often propri-
 125 etary nature of hardware design, as well as the relatively smaller
 126 community compared to software development. As a result, there
 127 is a limited pool of high-quality, diverse Verilog code available for
 128 training LLMs. Previous studies have focused on collecting and syn-
 129 thesizing more diverse Verilog code to improve the performance of
 130 these models [5, 17, 18]. However, they primarily concentrated on
 131 simpler modules, leaving a significant gap in addressing complex
 132 hardware designs. Consequently, LLMs trained on available data
 133 often struggle with the intricacies and specific requirements of
 134 HDLs, particularly when it comes to clock-cycle timing constraints,
 135 resource utilization, and microarchitecture optimizations.

136 Despite these challenges, there is significant potential to over-
 137 come these limitations and leverage LLMs effectively for FPGA
 138 programming. In this paper, we conduct an empirical comparison
 139 of LLM-based hardware design and traditional HLS approaches.
 140 We develop and evaluate specialized prompting methodologies to
 141 guide LLMs in generating more accurate and contextually appro-
 142 priate HDL code. Our approach demonstrates how carefully crafted
 143 prompts can significantly improve the quality and relevance of LLM-
 144 generated Verilog code. Additionally, we propose an integration of
 145 domain-specific knowledge into the LLM pipeline, incorporating
 146 FPGA architecture details, timing constraints, and resource utili-
 147 zation guidelines. This integration enhances the LLMs' ability to
 148 generate optimized HDL code, bridging the gap between high-level
 149 abstractions and efficient hardware implementations. Our work
 150 introduces and assesses post-processing techniques designed to
 151 refine and optimize LLM-generated code, addressing FPGA-specific
 152 requirements and improving overall code quality. Through a thor-
 153ough study of the strengths and limitations of LLMs across a diverse
 154 range of FPGA design tasks, we provide insights into the potential
 155 of LLMs to revolutionize FPGA programming.

156 Our contributions are summarized as follows:

- 158 • We conduct a comprehensive analysis of the fundamental
 159 limitations LLMs face when generating complex hardware
 160 HDLs for FPGA kernels.
- 161 • We propose a novel, iterative methodology that leverages
 162 LLMs for HDL generation, incorporating step-by-step code
 163 generation, few-shot examples, and targeted human inter-
 164 vention to address the identified limitations and produce
 165 synthesizable, efficient and user-friendly hardware designs.
- 166 • We demonstrate that our LLM-guided approach can gener-
 167 ate HDL implementations that surpass HLS tools in terms of
 168 latency and resource utilization for complex FPGA kernels.

170 The experimental results show that our approach reduces latency by
 171 an average of 28.88% compared to HLS, with a maximum reduction
 172 of 66.94% in pipelined designs and an average reduction of 8.41%,
 173 peaking at 55.76% in sequential stages. Furthermore, it decreases

175 LUT usage by an average of 25.18% and flip-flop usage by 57.23%
 176 compared to HLS.

177 2 BACKGROUND

178 In this section, we discuss programming frameworks for FPGA,
 179 including hardware description languages (HDLs) and High-Level
 180 Synthesis (HLS). Additionally, We introduce the emerging trend
 181 of using Large Language Models (LLMs) for FPGA programming,
 182 highlighting their potential to further simplify and accelerate the
 183 hardware design process.

184 **Table 1** provides a comprehensive comparison of LLM-based
 185 Design, HDLs and HLS across seven key aspects of FPGA program-
 186 ming, illustrating the trade-offs between traditional, emerging, and
 187 intermediate approaches to hardware design.

188 2.1 FPGA Programming Frameworks

189 Field-Programmable Gate Array (FPGA) programming has under-
 190 gone significant transformations to address the increasing complex-
 191 ity of hardware designs and the need for improved productivity.
 192 This evolution spans multiple generations of programming frame-
 193 works, each offering different levels of abstraction and control over
 194 hardware implementation.

195 Traditional HDLs such as Verilog and VHDL primarily operate
 196 at the Register-Transfer Level (RTL). At this level, they describe
 197 digital circuits by specifying the flow of data and control signals
 198 between registers and through combinational logic elements. While
 199 HDLs offer fine-grained control over hardware implementation,
 200 they require designers to manage intricate low-level details. This
 201 approach, although effective, often results in poor productivity and
 202 increased development time, especially for complex designs.

203 To address the limitations of HDLs, HLS tools [3, 8, 22, 27]
 204 like Vitis HLS [25] and Intel HLS Compiler [10] have been de-
 205 veloped. These tools employ a subset of software languages, such
 206 as C/C++ and SystemC, to describe hardware functionality at a
 207 higher level of abstraction. HLS tools automatically convert these
 208 high-level specifications into HDL representations such as VHDL
 209 or Verilog [6, 9, 13, 27]. This automation simplifies the design of
 210 complex hardware systems, reducing the time and effort required
 211 for low-level coding and allowing designers to optimize and ma-
 212 nipulate designs more efficiently [11, 12]. HLS also facilitates rapid
 213 design space exploration through the use of synthesis directives,
 214 constraints and DSE frameworks [20], enabling designers to quickly
 215 evaluate different implementation trade-offs.

216 However, there are notable drawbacks to HLS [14]. The hardware
 217 implementations generated by HLS often lack efficiency compared
 218 to hand-coded RTL, primarily due to the higher level of abstraction.
 219 This abstraction can lead to several key limitations. HLS-generated
 220 designs typically require more FPGA resources than their RTL
 221 counterparts due to the inherent overhead associated with RTL
 222 generation in HLS. Consequently, these designs may not efficiently
 223 utilize the hardware's capabilities. Performance constraints also
 224 present challenges for HLS designs. They frequently exhibit higher
 225 latency than optimized RTL implementations due to the complexi-
 226 ties involved in translating high-level code into hardware. Although
 227 HLS can sometimes achieve comparable performance metrics, it
 228 generally struggles with latency issues. Additionally, achieving

Aspect	LLM-based Design	HDL	HLS
Abstraction Level	Highest; uses natural language prompts for design description	Low-level; describes hardware at Register Transfer Level (RTL)	High-level; uses C/C++-like languages for algorithmic description
Learning Curve	Lower; focuses on effective prompt engineering and iterative refinement	Steep; requires in-depth understanding of hardware architecture and digital logic	Moderate; combines high-level programming with hardware design concepts
Code Quality and Complexity	Variable; may require significant refinement for complex designs	High; allows precise control but can be verbose for complex systems	High; benefits from years of compiler optimizations for efficient hardware generation
Design Process	Flexible and iterative; relies on natural language interaction and prompt refinement	Structured and detailed; requires explicit specification of hardware behavior	Algorithmic; focuses on functionality with automated hardware inference
Optimization Capabilities	Limited; requires explicit guidance and may not capture all hardware-specific optimizations	Extensive; allows fine-grained control over optimizations but requires manual implementation	Strong; incorporates built-in optimizations for various hardware architectures
Productivity	Potentially high for simple to moderate designs; faster initial prototyping	Lower for complex designs; time-consuming to write and verify detailed RTL code	High for complex algorithmic designs; accelerates development of computation-intensive systems
Scalability	Suitable for small to medium designs; may face challenges with very large or intricate systems	Highly scalable but requires significant manual effort for large designs	Scalable and efficient for large, algorithm-centric designs; may have limitations for fine-grained control

Table 1: Comparison between LLM-based hardware design, HDL, and HLS

precise control over timing and resource allocation can be more challenging with HLS compared to direct HDL programming [24].

2.2 LLMs for FPGA Programming

Large Language Models (LLMs) have emerged as powerful tools for code generation across various programming languages. Trained on vast amounts of code and natural language data, these models can learn the statistical patterns and relationships within the training data, enabling them to generate code that adheres to the syntax and style of the target programming language.

The application of LLMs to HLS spans various stages of the design process, from high-level specification to hardware verification [4, 21, 26]. LLMs have demonstrated significant potential in facilitating and optimizing the high-level synthesis (HLS) process for FPGAs by automating the conversion of high-level programming languages into hardware-compatible code. By leveraging their code generation capabilities, LLMs can effectively assist in transforming standard C/C++ code into HLS-compatible formats. This transformation process involves refactoring constructs that are not directly supported by HLS tools, such as dynamic memory allocation and recursion, into forms suitable for hardware implementation. Moreover, LLMs can be integrated into pipelines to automate the entire FPGA HLS workflow, from specification generation to optimization and deployment. These models can generate HLS-compatible code from natural language descriptions or standard C/C++ code, allowing software engineers to engage in hardware design with reduced expertise requirements. In the realm of hardware verification, LLMs show promise in automating the generation of test cases and identifying potential design flaws. This application could lead to more thorough and efficient verification processes, crucial for

ensuring the reliability of complex hardware designs. Additionally, LLMs are being explored as assistants in design space exploration, suggesting optimizations and exploring design alternatives based on specified constraints and objectives.

Apart from integrating LLMs into HLS workflows, researchers in the field of hardware design have shown a growing interest in leveraging LLMs for generating Hardware Description Languages (HDLs) directly. The quality of the generated code depends heavily on the quality and quantity of the training data. However, the hardware design community is significantly smaller and more closed-source than the mainstream programming languages community, resulting in a limited pool of high-quality and diverse Verilog code for training LLMs. Furthermore, LLMs trained on large amounts of software code tend to be biased towards producing software-like Verilog code. This bias can lead to inefficiencies in FPGA programming, as LLMs may not fully grasp the inherent parallelism of HDLs, which is crucial for efficient hardware design. To address these challenges, several studies have attempted to synthesize high-quality Verilog datasets using LLMs [5, 16–18]. While these efforts have yielded improvements in generating smaller modules, they still struggle with complex kernels. The generated code often fails to meet the stringent requirements of sophisticated hardware designs.

Recognizing these limitations, some researchers [4, 15, 21] have emphasized the necessity of expert guidance for complex kernels. However, the lack of an efficient approach has led to scenarios where designers spend a significant number of prompts and considerable effort guiding LLMs to generate correct code, which is highly inefficient and time-consuming. Our research addresses this gap by exploring more effective methods for utilizing LLMs in FPGA programming. We propose that while LLMs may not yet be capable of

fully generating correct code without expert guidance, employing the right methods can drastically reduce the number of prompts required. Moreover, our approach produce hardware code with resource utilization and latency performance compared to traditional methods. By focusing on optimizing the interaction between human experts and LLMs, we seek to leverage the strengths of both to create a more efficient and effective FPGA programming process. This approach not only addresses the current limitations of LLMs in hardware design but also paves the way for future advancements in the field, potentially revolutionizing the way complex FPGA designs are developed and optimized.

3 METHODOLOGY

In this section, we present a detailed introduction to our experimental methodology. In [subsection 3.1](#), we discuss the limitations of applying LLMs to complex HDLs in FPGA kernels. [subsection 3.2](#) details the step-by-step process of our generation methodology. Finally, in [subsection 3.3](#), we provide an in-depth case study of a kernel from PolyBench to illustrate our approach in a practical context.

3.1 Limitations of LLM

The application of LLMs to generate complex HDL FPGA kernels faces significant challenges, primarily stemming from the models' training on software-centric datasets. Even the most advanced models, such as Claude3.5-Sonnet [2] and GPT-4o [19], can generate hardware code correctly for most relatively small-scale modules but tend to generate software-like HDLs when faced with complex kernels. These limitations are primarily due to the fundamental differences between software and hardware paradigms. Software paradigms focus on sequential execution and high-level abstractions, whereas hardware paradigms require parallelism, timing control, and resource constraints to ensure synthesizability and efficiency. [Figure 1](#) illustrates two common issues:

Resource Insensitivity. As shown in [Figure 1a](#), LLMs tend to describe hardware using software-like constructs, such as multi-dimensional arrays ($A [0:7][0:7]$), rather than employing hardware-specific port and address specifications. This approach, while intuitive from a software perspective, fails to align with hardware programming guidelines necessary for synthesizing efficient IP cores. Efficient FPGA designs require careful consideration of resource allocation, timing constraints, and parallel processing capabilities—aspects that software-trained LLMs struggle to incorporate inherently.

Syntax Errors. The examples in [Figure 1b](#) and [Figure 1c](#) illustrate the propensity of LLMs to misapply software-specific syntax in hardware descriptions, leading to syntax errors in HDL code generation. [Figure 1b](#) demonstrates a common issue in LLM-generated testbenches for hardware like the atax kernel. Here, the model incorrectly employs the software increment operator $i++$, which is invalid in Verilog. The correct hardware syntax requires explicit assignment, such as $i = i + 1$. This misalignment between software and hardware syntax conventions often results in non-synthesizable code. Furthermore, LLMs trained on software paradigms struggle to grasp hardware-specific concepts such as loop unrolling or the implementation of loops as state machines, which are crucial for

```

499 module atax_kernel(
500   input wire clk,
501   input wire reset,
502   input wire start,
503   input wire [31:0] A [0:7][0:7],
504   input wire [31:0] x [0:7],
505   output reg [31:0] y [0:7],
506   output reg done
507 );
508
509   for (int i = 0; i < 8; i++) begin
510     for (int j = 0; j < 8; j++) begin
511       A[i][j] = i * 8 + j + 1;
512       x[i] = i + 1;
513     end
514   end
515
516   (a) IO Port Issue
517
518   (b) Loop Counter Issue
519
520 module incorrect_wire_assignment(
521   input wire clk,
522   input wire reset,
523   input wire [31:0] a,
524   input wire [31:0] b,
525   output wire [31:0] result
526 );
527
528   always @ (posedge clk or posedge reset) begin
529     if (reset) begin
530       result = 32'b0;
531     end else begin
532       result = a + b;
533     end
534   end
535
536   endmodule
537
538   (c) Incorrect wire assignment in always block
539
540

```

Figure 1: Limitations of LLM-generated Verilog code influenced by software-like patterns

efficient FPGA designs. A more fundamental misunderstanding of hardware concepts is evident in [Figure 1c](#). This example reveals LLMs' difficulty in distinguishing between wire and register assignments in sequential logic—a cornerstone principle of HDL design. The generated code erroneously attempts to assign values to wires within sequential blocks, a practice that violates basic HDL principles and leads to non-functional designs. This type of error represents one of the most prevalent issues in LLM-generated HDL code, underscoring the significant gap between software-oriented training data and the specific requirements of hardware description languages.

Insufficient Awareness of Clock Cycles and Register Properties. Another critical limitation of LLMs in generating HDL code is their inadequate understanding of clock cycle behavior and register properties inherent to hardware design. This deficiency often leads to functionally incorrect or inefficient implementations, particularly in complex pipelined designs. [Figure 2](#) illustrates part of an atax kernel implementation in SystemVerilog, highlighting two significant issues that stem from this limitation:

Misunderstanding of Pipeline Cycle Relationships. In the example, the LLM-generated code incorrectly sets the condition for completing a pipeline stage as $j == N - 1$. This demonstrates a fundamental misunderstanding of how pipeline stages relate to clock cycles in hardware. The correct condition should be $j == N$, accounting for the two-cycle latency of the single-cycle multiplier and register-based address storage in the matrix-vector product calculation. This error reveals that the LLM is treating the hardware implementation

```

465 COMPUTE_TMP: begin
466     if (j == 0) begin
467         acc <= 32'b0;
468         we_tmp <= 0;
469     end else begin
470         acc <= acc + mult_p;
471     end
472
473     if (j == N - 1) begin
474         j <= 0;
475         we_tmp <= 1;
476         di_tmp <= acc + mult_p;
477         addr_tmp_in <= i;
478         addr_x_out <= 0;
479
480         if (i == N - 1) begin
481             i <= 0;
482             addr_a_out <= 0;
483             state <= COMPUTE_Y;
484         end else begin
485             i <= i + 1;
486             addr_a_out <= i * N;
487         end
488     end else begin
489         j <= j + 1;
490         addr_a_out <= addr_a_out + 1;
491         addr_x_out <= addr_x_out + 1;
492     end
493 end
494
495 end

```

Figure 2: Example of Insufficient Awareness of Clock Cycles and Register Properties

as if it were software, assuming N operations can be completed in N iterations. In reality, this pipeline requires N executions with an interval of 1 and a latency of 2 for each execution. The LLM's software-centric approach fails to account for the temporal aspects of hardware execution, potentially leading to incorrect results or timing issues in the synthesized design.

Overlooking Register Update Timing. The second issue pertains to the property of registers and the timing of their value updates. In the code, the counter i is a register using non-blocking assignment (\leq). However, the LLM fails to account for the fact that the value of i will not be updated until the next clock cycle. This is evident in the line $addr_a_out \leq i * N$, where the LLM assumes an immediate update of i . In hardware design, it's crucial to consider that register values change only on clock edges, and these changes are not immediately reflected in subsequent operations within the same clock cycle. This oversight can lead to off-by-one errors or more complex timing issues in the implemented design.

These limitations highlight a significant disconnect between the software-centric training of LLMs and the specialized requirements of hardware description languages. The gap becomes increasingly problematic as the complexity of hardware designs grows, making it crucial to address these challenges for the successful application of LLMs in FPGA programming.

Despite these challenges, there exist viable strategies to mitigate each of these limitations:

Resource Insensitivity. To address the LLMs' tendency to generate resource-inefficient HDL code, we can implement a guided approach that incorporates hardware-aware constraints and optimization goals into the prompts. By providing specific guidelines and clear definitions of the expected IP characteristics, we can steer LLMs towards producing more resource-efficient designs. This method leverages the LLMs' ability to adapt to detailed instructions

while compensating for their lack of inherent hardware resource understanding.

Syntax Errors. The prevalence of software-specific syntax in LLM-generated HDL can be mitigated through an iterative feedback process. By feeding compiler error messages back to the LLM and requests corrections, many syntax errors can be resolved without extensive human intervention. This approach not only corrects immediate errors but also helps the LLM learn and adapt to HDL-specific syntax over time, potentially improving its performance in subsequent code generation tasks.

Insufficient Awareness of Clock Cycles and Register Properties. This limitation, being the most complex, often requires a combination of strategies. We propose a two-pronged approach: a) Develop specialized prompts that explicitly highlight the temporal nature of hardware execution, emphasizing concepts such as clock synchronization and parallel processing. b) Integrate human expertise into the process to identify cycle-related errors and provide clear, cycle-accurate explanations to the LLM. This approach ensures that subtle timing issues are addressed and helps guide the LLM towards a more accurate understanding of hardware execution principles.

In the following sections, we will detail our methodology for implementing these solutions. We will provide concrete examples of how each strategy is applied in practice, showcasing the potential of LLM-assisted hardware design when properly guided and constrained.

3.2 Generation Flow

Our generation methodology, as illustrated in Figure 3, addresses the limitations of LLMs in hardware design through a structured, iterative approach that minimizes the need for human intervention while maximizing the LLM's ability to generate accurate and efficient hardware designs.

The process begins with a C++ kernel to be generated, which the LLM analyzes to produce a step-by-step description of the code's behavior. This description is crucial, as it breaks down complex kernels into manageable stages, allowing the LLM to focus on smaller, more tractable parts of the design. This approach directly addresses the LLM's tendency to generate software-like solutions when faced with complex hardware tasks.

For each stage of the kernel, the LLM is tasked with generating both RTL code and a corresponding testbench. The RTL code is required to include comments that explicitly address clock cycle behavior, forcing the LLM to consider timing and state in a hardware-specific context. This step-by-step generation process, combined with cycle-aware commenting, significantly improves the LLM's awareness of clock cycles and register properties - a key limitation we identified earlier. The testbench for each stage serves a dual purpose: it verifies the correctness of the generated RTL and provides a mechanism for the LLM to self-correct. If a stage fails verification, the LLM is prompted to print intermediate variables and debug its own code. This self-correction mechanism is a critical feature of our methodology, as it minimizes the need for human intervention and allows the LLM to learn from its mistakes. Human guidance is reserved for situations where the LLM fails to self-correct or when hardware-specific optimizations, such

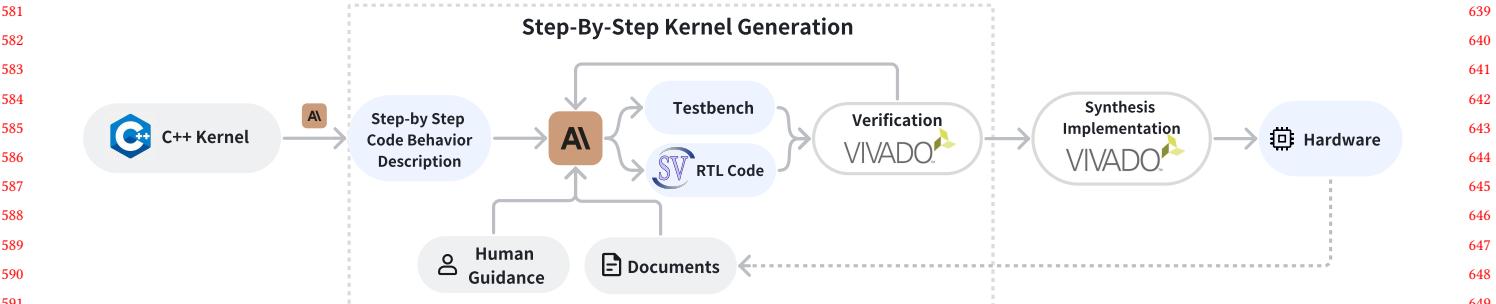


Figure 3: Overview of Generation Methodology

as pipelining, are required. This targeted use of human expertise ensures that the process remains largely automated while still benefiting from specialized knowledge when necessary.

Resource constraints and cycle behavior specifications for the desired IP are provided in the documentation available to the LLM. This information helps address the resource insensitivity issue we previously identified, guiding the LLM towards more efficient hardware implementations from the outset.

The optimization process is also iterative. The LLM first generates a base sequential implementation, which serves as a foundation for more optimized versions. These optimizations are developed through a combination of LLM-generated improvements, guided by the sequential baseline, relevant documentation, and targeted human input. Once the full RTL code passes verification, it undergoes synthesis and implementation in Vivado, generating a resource utilization report.

It is to be noticed that a key strength of this methodology is its ability to leverage previously generated designs. As more kernels are successfully implemented, they serve as few-shot examples for the LLM. For new designs, the most similar previously generated kernel is used as a reference, often allowing the LLM to generate complete kernels and testbenches in just one or two prompts, without the need for step-by-step separation. For entirely new or complex kernels without close analogs, the step-by-step approach is still employed. In these cases, a related generated hardware design is still provided as a reference, helping the LLM understand the general structure and expectations of hardware implementations.

This comprehensive approach not only addresses the limitations of LLMs in hardware design but also creates a scalable, self-improving system. By focusing on manageable steps, incorporating self-correction mechanisms, and building a library of reference designs, our methodology enables LLMs to generate increasingly complex and efficient FPGA kernels with minimal human intervention.

3.3 Case Study

In this section, we present a detailed case study that illustrates the practical application of our LLM-based hardware design methodology. This case study focuses on the generation and optimization of the gemver kernel, demonstrating each step of our process.

We begin by demonstrating how to generate a testbench for the gemver kernel. Figure 4 displays the prompt used to instruct

the LLM in this task. This prompt provides the LLM with the key operations of the gemver kernel, extracted from the Polybench benchmark suite. Importantly, it also directs the LLM to refer to a previously generated gemm testbench as a reference, which provides the LLM with guidance on the structure, variable declarations, and verification techniques typical in hardware design testbenches. By leveraging this prior knowledge, we enable the LLM to generate a more accurate and syntactically correct testbench for the new kernel.

```

Here are the key operations of the gemver testbench in
Polybench.
#pragma scop
for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
        A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];
for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
        x[i] = x[i] + beta * A[j][i] * y[j];
for (i = 0; i < _PB_N; i++)
    x[i] = x[i] + z[i];
for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
        w[i] = w[i] + alpha * A[i][j] * x[j];
#pragma endscop
Please write a testbench for it, you can refer to the gemm
testbench as follows...

```

Figure 4: Prompt for generate testbench

After generating the testbench, we proceed to the critical step of creating the gemver kernel. This process exemplifies how our methodology addresses the limitations of LLMs in hardware design, particularly in terms of resource sensitivity and cycle-accurate behavior. Figure 5 shows the prompt used to guide the LLM in this task. This prompt is carefully crafted to address the specific challenges of hardware design and to leverage the LLM's capabilities effectively. It explicitly specifies the use of particular multiplier and RAM components. This directive ensures that the LLM considers actual hardware resources in its design, rather than defaulting to software-like, resource-insensitive implementations. To align the LLM's output with realistic hardware constraints and ensure fair comparisons with HLS tools in delay, we specify a single-cycle

```

697 Please write the gemver kernel further. Here are some
698 constraints.
699 1. You need to use the specified multiplier and RAM to perform
700 multiplication operations and read or write matrices or
701 vectors.
702 [documents]...
703 2. Please note that completing one Multiply-Accumulate (MAC)
704 operation requires three cycles.
705 In the first cycle, assign addresses for reading or writing
706 values; these will not be valid until the second cycle.
707 During the second cycle, wait for the output from the
708 single-cycle multiplier.
709 Finally, in the third cycle, assign the accumulated value. For
710 further reference, you can consult the 2mm kernel.
711 3. Please write the code for update A first, which is  $A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j]$ ...
712
713 You can refer to the gemm kernel as follows ...

```

Figure 5: Prompt for generate sequential design

```

714 COMPUTE_X: begin
715     //  $x[i] = x[i] + beta * A[j][i] * y[j] + z[i]$ ;
716     case (stage)
717         0: begin // First stage: Assign addresses
718             addr_a_out <= j * N + i;
719             addr_y_out <= j;
720             addr_x_out <= i;
721             addr_z_out <= i;
722             stage <= 1;
723             valid_reg <= 0;
724             we_x <= 0;
725         end
726         1: begin // Second stage: Wait for multiplication results
727             stage <= 2;
728         end
729         2: begin // Third stage: Accumulate
730             acc_x <= (j == 0) ? mult1_p : acc_x + mult1_p;
731
732             if (j == N - 1) begin
733                 stage <= 3;
734             end else begin
735                 j <= j + 1;
736                 stage <= 0;
737             end
738         end
739         3: begin // Fourth stage: Start beta multiplication
740             stage <= 4;
741         end
742         4: begin // Fifth stage: Add z[i] and write result
743             di_x <= dout_x + mult2_p + dout_z;
744             we_x <= 1;
745             addr_x_in <= i;
746             data_out_reg <= dout_x + mult2_p + dout_z;
747             valid_reg <= 1;
748             if (i == N - 1) begin
749                 state <= COMPUTE_W;
750                 i <= 0;
751                 j <= 0;
752             end else begin
753                 i <= i + 1;
754                 j <= 0;
755             end
756             stage <= 0;
757         end
758     endcase
759 end

```

Figure 6: Code for the COMPUTE_X state in the gemver kernel (sequential implementation)

multiplier. Otherwise, it generates a combinational multiplier by default. The prompt also provides detailed instructions on the timing of Multiply-Accumulate (MAC) operations, breaking them down into a three-cycle process. This level of specificity compels the LLM

to explicitly consider clock cycles and register properties, directly addressing a major limitation we identified earlier in LLM-based hardware design. Our step-by-step methodology is evident in the prompt's instruction to focus first on the matrix update operation. This approach helps manage complexity by allowing the LLM to concentrate on generating correct, efficient code for each part of the kernel separately. Additionally, the prompt references a previously generated gemm kernel as an example, similar to the approach used in testbench generation.

Code of COMPUTE_X state...
 Can you implement a pipelined approach for the process between
 stage 0 and 2? Currently, completing one Multiply-Accumulate
 (MAC) operation requires three cycles. In the first cycle,
 addresses are assigned for reading or writing values, but
 these will not be valid until the second cycle. The second
 cycle waits for the output from a single-cycle multiplier. In
 the third cycle, the accumulated value is assigned. As a
 result, it takes $3N$ cycles to perform this operation.

I propose a pipelined method as follows:
 - Cycle 0: Assign addresses for accumulation 0.
 - Cycle 1: Assign address for accumulation 1 and wait for
 multiplication of accumulation 0 (do nothing).
 - Cycle 2: Assign address for accumulation 2 and wait for
 multiplication of accumulation 1 (do nothing), then assign
 accumulated value from cycle 0 to $x0$.

This pattern continues:
 - For each subsequent cycle up to $N - 1$:

- Assign address of accumulation $N - 1$,
- Wait on multiplication of previous accumulations while
 assigning earlier accumulated values accordingly.

In summary:
 - Cycle $N - 1$: Wait on multiplication of accumulation $N - 2$ (do
 nothing) and assign accumulated value from $N - 3$ to $x(N - 3)$.
 - Cycle N : Wait on multiplication of accumulation $N - 1$ (do
 nothing) and assign accumulated value from $N - 2$ to $x(N - 2)$.
 - Cycle $N + 1$: Assign accumulated value from $N - 1$ to $x(N - 1)$.

This pipelining reduces latency from $3N$ cycles down to just $N + 2$ cycles. Please revise your code accordingly.

Figure 7: Prompt for implementing the sequential kernel in pipeline

Figure 6 shows the Verilog code generated by the LLM for the COMPUTE_X stage of the gemver kernel in its sequential implementation. It adheres to the specified hardware constraints and uses a 5-stage process to compute the required operation, carefully managing timing for memory access and multiplication. The design includes single-cycle multipliers mult1 and mult2, and it implements the three-cycle MAC operation as requested. By assigning addresses, waiting for multiplier outputs, accumulating results, and handling beta multiplication separately, the code showcases efficient loop control and state transitions. This implementation reflects the LLM's ability to generate hardware-aware designs when provided with appropriate guidance and constraints.

The most challenging aspect was leveraging the LLM for pipeline optimization. **Figure 7** presents the prompt designed to guide the LLM in implementing a pipelined version of the COMPUTE_X state in the gemver kernel. Notably, this prompt is itself generated by the LLM, demonstrating how an abstract pipeline representation can yield detailed guidelines with minimal human input.

```

813 COMPUTE_X: begin
814     // x[i] = x[i] + beta * A[j][i] * y[j] + z[i];
815     case (stage)
816         0: begin // First stage: Multiply and Accumulate
817             valid_reg <= 0;
818             we_x <= 0;
819             addr_x_out <= i;
820             addr_z_out <= i;
821
822             if (pipeline_stage < N) begin
823                 // Assign addresses for next accumulation
824                 addr_a_out <= j * N + i;
825                 addr_y_out <= j;
826                 j <= j + 1;
827             end
828
829             if (pipeline_stage >= 2) begin
830                 acc_x <= (pipeline_stage == 2) ? mult1_p :
831                                     acc_x + mult1_p;
832             end
833
834             if (pipeline_stage == N + 1) begin
835                 pipeline_stage <= 0;
836                 stage <= 1;
837             end else begin
838                 pipeline_stage <= pipeline_stage + 1;
839             end
840         end
841         1: begin // Second stage: Start beta multiplication
842             stage <= 2;
843         end
844         2: begin // Third stage: Add z[i] and write result
845             di_x <= dout_x + mult2_p + dout_z;
846             we_x <= 1;
847             addr_x_in <= i;
848             data_out_reg <= dout_x + mult2_p + dout_z;
849             valid_reg <= 1;
850             if (i == N - 1) begin
851                 state <= COMPUTE_W;
852                 i <= 0;
853                 j <= 0;
854             end else begin
855                 i <= i + 1;
856                 j <= 0;
857             end
858             stage <= 0;
859         end
860     endcase
861 end

```

Figure 8: Code for the COMPUTE_X state in the gemver kernel (pipeline implementation)

The prompt begins by referencing the previous sequential implementation, providing context and highlighting the inefficiency of the current three-cycle Multiply-Accumulate (MAC) operation, which results in a 3N cycle latency. At its core, the prompt offers a detailed, cycle-by-cycle breakdown of the proposed pipelined method, illustrating how overlapping multiple accumulations can significantly reduce overall latency. This step-by-step explanation is crucial in guiding the LLM to grasp the concept of hardware pipelining. By providing a concrete, cycle-specific example of the pipeline's operation, the prompt offers the LLM a clear template to follow.

The LLM-generated pipelined implementation of the COMPUTE_X state in the gemver kernel is illustrated in Figure 8. It focuses on pipelining the inner accumulation loop, previously encompassed in stages 0 to 2, while maintaining the subsequent beta multiplication and result writing stages. The design introduces a pipeline_stage variable to manage concurrent processing, enabling efficient overlap

of multiple accumulations. In the pipeline stage, the code simultaneously handles address assignments for the next accumulation, multiplication waiting, and accumulation of previous results. This pipelined structure efficiently utilizes hardware resources, reducing overall latency from 3N to approximately N+2 cycles.

Despite the LLM's ability to generate complex hardware designs, the process is not without challenges. The LLM occasionally struggles to understand hardware-specific timing nuances as mentioned earlier. For instance, it may overlook the inherent one-cycle delay in registers and multipliers, incorrectly assuming immediate output validity after input assignment. These timing oversights are particularly evident in the pipeline implementation, where the LLM may struggle with the intricate timing requirements of start-up and shutdown phases, or neglect to reset critical signals. In such cases, human intervention becomes necessary to refine the LLM-generated implementation and ensure its correctness and efficiency.

4 EXPERIMENTAL RESULTS

In this section, we present the experimental setup and results for our comparison of hardware designs generated using LLMs, traditional HLS tools, and a specialized tool, Cement. We aim to compare the quality of hardware generated by LLMs with the efficiency of traditional and state-of-the-art tools in terms of performance and resource usage. Below, we elaborate on the experiment settings, evaluation criteria, and present a detailed discussion of the findings.

4.1 Experimental Setup

We conducted our experiments using eleven benchmarks selected from the PolyBench benchmark suite. These kernels were chosen for their diverse computational behaviors and control structures, including various types of branches and loops. This diversity allows us to comprehensively assess the performance and resource efficiency of the generated circuits across different approaches. Our experiment aims to compare the quality of hardware designs generated using three distinct approaches:

- (1) An LLM-based approach utilizing Claude 3.5 Sonnet [1], which employs our methodology to generate System Verilog.
- (2) The Cement framework [24], which allows users to determine the occurrence cycle time of hardware operations and generates control logic with lower overhead than that of HLS, and achieves best results in latency in PolyBench compared to existing HLS or DSL tools.
- (3) A commercial HLS tool Vitis HLS [25].

For our experiments with the PolyBench benchmark suite, we compare cycle count and resource utilization across three approaches. For the LLM-based approach, we collect cycle counts by simulating the produced SystemVerilog code with Icarus Verilog [23]. Cement cycle counts are estimated using Verilator. Resource utilization for both the LLM-based approach and Cement is estimated by running synthesis with Vivado 2022.2, targeting the Virtex UltraScale+ XCVU9P FPGA. For Vitis HLS 2022.2, we collect metrics including cycle count and resource utilization from the co-simulation and implementation reports. To ensure fair comparison, we set a consistent target clock period of 7ns across all approaches, including Vitis

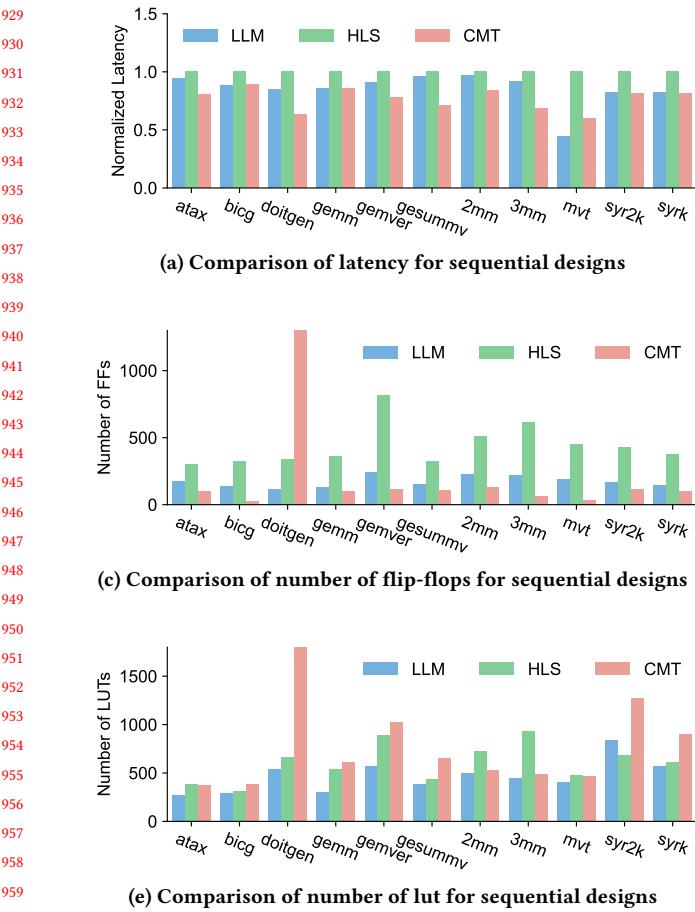


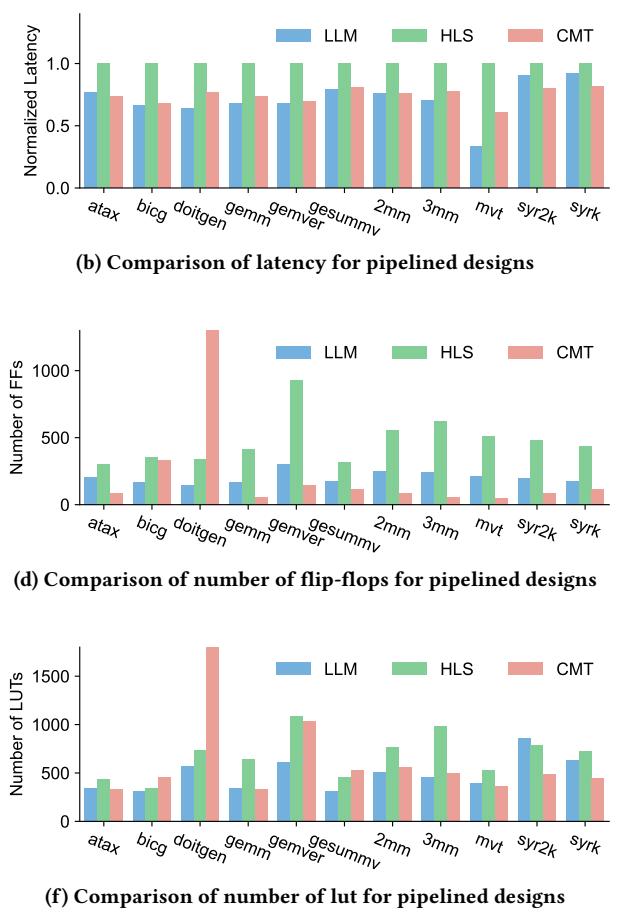
Figure 9: Comparison of hardware resources and latency for LLM-based approaches, Cement, and HLS in both sequential and pipelined designs

HLS designs and the synthesis of LLM-generated SystemVerilog and Cement implementations.

4.2 Experiments on PolyBench

The experimental results on the PolyBench benchmark suite is illustrated in Figure 9. Figure 9(a) demonstrates that the LLM-based approach consistently achieves lower cycle counts for all kernels compared to HLS in sequential designs. This improvement can be attributed to the reduction of loop overhead inherent in RTL generation of HLS. Our method yields an average latency improvement of 8.41%, with a remarkable maximum improvement of 55.76% in a certain kernel. While slightly inferior to Cement in some cases, this can be explained by our deliberate choice to avoid aggressive optimizations and the inclusion of assisted registers. These design decisions, while introducing some unnecessary latency, are made to facilitate better understanding by LLMs and to provide more comprehensible few-shot examples. For instance, our use of registers to store read addresses, rather than employing wires and combinational logic, introduces one cycle latency overhead in each iteration of sequential designs.

The advantages of our approach become even more pronounced in pipelined implementations, as shown in Figure 9b. Here, we



observe an average latency reduction of 28.88% compared to HLS, with a maximum improvement of 66.94%. Notably, our method also outperforms Cement in pipelined designs, achieving a 5.23% average improvement and a 45.21% maximum improvement. This enhanced performance in pipelined versions can be attributed to the fact that the redundant latency primarily affects the pipeline initiation, rather than each iteration as in sequential designs.

Other figures provide a comprehensive comparison of hardware resource utilization. Our LLM-based approach demonstrates significant reductions in resource requirements compared to HLS, enhancing overall utilization efficiency. Specifically, we observe an average reduction of 25.18% in LUT usage (with a maximum reduction of 53.31%) and an average reduction of 57.23% in flip-flop usage (with a maximum reduction of 70.45%). When compared to Cement, our approach shows advantages in LUT utilization but is less efficient in flip-flop usage. This discrepancy can be attributed to Cement's state compression techniques, which result in more complex state transition logic with fewer flip-flops but more LUTs. It's worth noting that the hardware resource utilization of cement for the doitgen benchmark appears anomalous.

The effectiveness of our LLM-based approach is evident in the significant reduction of prompts required to generate for various

benchmarks from the PolyBench suite. Table 2 presents a comparative analysis of the number of prompts used in our method versus the previous work[15]. Our method demonstrates a marked improvement in efficiency across all benchmarks. For sequential designs, we achieved successful generation with as few as 1 to 5 prompts, a substantial reduction compared to the 11 to 50 prompts required in previous work. This efficiency is particularly notable in complex benchmarks such as syrk, where our approach required only 4 prompt compared to 50 in the previous study. The pipelined versions of these designs, which are derived from sequential ones, are inherently more complex but still maintain a low prompt count of 4 to 8. It's worth noting that the atax benchmark required more attempts in our study, as it was the first benchmark processed and lacked reference implementations. This highlights the learning curve in our approach and the potential for further efficiency gains as the system accumulates more examples.

Table 2: Comparison of the Number of Prompts for LLM-Based Approaches

Benchmark	Sequential	Pipelined	Previous work [15]
atax	5	8	11
bicg	3	7	16
doitgen	2	7	-
gemm	2	5	22
gemver	2	4	-
gesummv	2	4	23
2mm	1	5	29
3mm	2	4	21
mvt	1	5	36
syr2k	4	5	20
syrk	4	8	50

In conclusion, our LLM-based approach demonstrates a significant advancement in hardware design methodology. While HLS has long been praised for its high-level abstraction, LLM based hardware design enable an even higher-level description while simultaneously achieving superior performance. The consistent improvements in both latency and resource utilization across a diverse set of benchmarks underscore the potential of LLM-based approaches in revolutionizing hardware design processes. Despite the capabilities of LLM, the complexity of hardware design, particularly in areas such as pipelining and cycle-accurate implementations, often requires expert guidance and refinement. Our method, while showing promise in reducing the number of prompts needed compared to previous work, still relies on human expertise to navigate the intricacies of hardware-specific concepts and to validate and optimize the generated designs.

5 CONCLUSION

This paper demonstrates the potential of LLMs for generating FPGA hardware designs, providing significant improvements in design efficiency and performance compared to traditional HLS tools through an empirical comparison of LLM-based hardware Design and HLS. Despite the inherent challenges posed by the software-centric training of these models, our structured and iterative methodology, which integrates guided prompts, step-by-step

breakdowns, and human interventions, has enabled the effective generation of complex FPGA kernels. The experimental results show that our LLM-based approach significantly reduces latency and resource utilization compared to HLS-generated designs, especially in pipelined implementations. It achieves an average latency reduction of 28.88% relative to HLS, with a maximum reduction of 66.94% in pipelined designs and an average reduction of 8.41%, peaking at 55.76% in sequential stages. Additionally, it achieves a 25.18% average reduction in LUT usage and a 57.23% decrease in flip-flop usage compared to HLS. Furthermore, our approach requires fewer prompts for kernel generation, demonstrating improved efficiency and scalability over previous LLM-based methods. In conclusion, our study contributes to bridging the gap between high-level algorithmic descriptions and efficient FPGA implementations, we hope it will serve as a useful foundation for further advancements.

REFERENCES

- [1] Anthropic. 2024. Claude 3.5 Sonnet. <https://www.anthropic.com/news/clause-3-5-sonnet>
- [2] Anthropic. 2024. Introducing the next generation of Claude. <https://www.anthropic.com/news/clause-3-family>
- [3] Andrew Canis, Jongsook Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. 33–36.
- [4] Luca Collini, Siddharth Garg, and Ramesh Karri. 2024. C2HLS: Can LLMs Bridge the Software-to-Hardware Design Gap? *arXiv preprint arXiv:2406.09233* (2024).
- [5] Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Demin Song, Dahu Lin, Xingcheng Zhang, et al. 2024. OriGen: Enhancing RTL Code Generation with Code-to-Code Augmentation and Self-Reflection. *arXiv preprint arXiv:2407.16237* (2024).
- [6] Steve Dai and Zhiru Zhang. 2019. Improving scalability of exact modulo scheduling with specialized conflict-driven learning. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.
- [7] Hagai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. {NICA}: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 345–362.
- [8] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fioriti, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Bambu: an open-source research framework for the high-level synthesis of complex applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1327–1330.
- [9] Hsuan Hsiao and Jason Anderson. 2019. Thread weaving: Static resource scheduling for multithreaded high-level synthesis. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.
- [10] Intel. [n. d.]. Intel High Level Synthesis Compiler. <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html>
- [11] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. 2021. Tensorlib: A spatial accelerator generation framework for tensor algebra. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 865–870.
- [12] Liancheng Jia, Yuyue Wang, Jingwen Leng, and Yun Liang. 2022. EMS: efficient memory subsystem synthesis for spatial accelerators. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 67–72.
- [13] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 127–136.
- [14] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. 2018. Are we there yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2018), 898–911.
- [15] Yuchao Liao, Tosiron Adegbija, and Roman Lysenky. 2024. Are LLMs Any Good for High-Level Synthesis? *arXiv preprint arXiv:2408.10428* (2024).
- [16] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–8.
- [17] Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. 2024. CraftRTL: High-quality Synthetic Data Generation for Verilog Code Models with Correct-by-Construction Non-Textual Representations and Targeted Code Repair. *arXiv*

1161 *preprint arXiv:2409.12993* (2024). 1219

1162 [18] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 1220

1163 2023. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open- 1221

1164 source dataset and lightweight solution. *arXiv preprint arXiv:2312.08617* (2023). 1222

1165 [19] OpenAI. 2024. chatgpt-4o. <https://openai.com/index/hello-gpt-4o/> 1223

1166 [20] Benjamin Carrion Schafer and Zi Wang. 2019. High-level synthesis design space 1224

1167 exploration: Past, present, and future. *IEEE Transactions on Computer-Aided 1225*

1168 Design of Integrated Circuits and Systems 39, 10 (2019), 2628–2639. 1226

1169 [21] Sneha Swaroopa, Rijoy Mukherjee, Anushka Debnath, and Rajat Subhra 1227

1170 Chakraborty. 2024. Evaluating Large Language Models for Automatic Register 1228

1171 Transfer Logic Generation via High-Level Synthesis. *arXiv preprint 1229*

1172 arXiv:2408.02793

1173 [22] Shuo Wang, Yun Liang, and Wei Zhang. 2017. FlexCL: An analytical performance 1230

1174 model for OpenCL workloads on flexible FPGAs. In *Proceedings of the 54th Annual 1231*

1175 Design Automation Conference 2017

1176 [23] Stephen Williams and Michael Baxter. 2002. Icarus verilog: open-source verilog 1232

1177 more than a year later. *Linux Journal* 2002, 99 (2002), 3. 1233

1178 [24] Youwei Xiao, Zizhang Luo, Kexing Zhou, and Yun Liang. 2024. Cement: Stream- 1234

1179 lining FPGA Hardware Design with Cycle-Deterministic eHDL and Synthesis. 1235

1180 In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field 1236*

1181 Programmable Gate Arrays

1182 [25] AMD Xilinx. 2023. Vitis High-Level Synthesis User Guide (UG1399). [https://docs.xilinx.com/r/en-US/ug1399-vitis-hls](https:// 1237</p>
<p>1183 //docs.xilinx.com/r/en-US/ug1399-vitis-hls) 1238

1184 [26] Chenwei Xiong, Cheng Liu, Huawei Li, and Xiaowei Li. 2024. HLSPilot: LLM- 1239

1185 based High-Level Synthesis. *arXiv preprint arXiv:2408.06810* (2024). 1240

1186 [27] Ruifan Xu, Youwei Xiao, Jin Luo, and Yun Liang. 2022. HECTOR: A multi- 1241

1187 level intermediate representation for hardware synthesis methodologies. In 1242

1188 *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided 1243*

1189 Design

1190 [28] Xinyi Zhang, Yawen Wu, Peipei Zhou, Xulong Tang, and Jingtong Hu. 2021. 1244

1191 Algorithm-hardware co-design of attention mechanism on FPGA devices. *ACM 1245*

1192 *Transactions on Embedded Computing Systems (TECS)* 20, 5s (2021), 1–24. 1246

1193

1194

1195

1196

1197

1198

1199

1200

1201

1202

1203

1204

1205

1206

1207

1208

1209

1210

1211

1212

1213

1214

1215

1216

1217

1218