Bench4HLS: End-to-End Evaluation of LLMs in High-Level Synthesis Code Generation

M Zafir Sadik Khan, Kimia Azar, Hadi Kamali
Department of Electrical and Computer Engineering (ECE), University of Central Florida, Orlando, FL 32816, USA
{mzafirsadik.khan, azar, kamali}@ucf.edu

***Abstract*—In last two years, large language models (LLMs) have shown strong capabilities in code generation, including hardware design at register-transfer level (RTL). While their use in high-level synthesis (HLS) remains comparatively less mature, the ratio of HLS- to RTL-focused studies has shifted from 1:10 to 2:10 in the past six months, indicating growing interest in leveraging LLMs for high-level design entry while relying on downstream synthesis for optimization. This growing trend highlights the need for a comprehensive benchmarking and evaluation framework dedicated to LLM-based HLS. To address this, We present Bench4HLS for evaluating LLM-generated HLS designs. Bench4HLS comprises 170 manually drafted and validated case studies, spanning small kernels to complex accelerators, curated from widely used public repositories. The framework supports fully automated assessment of compilation success, functional correctness via simulation, and synthesis feasibility/optimization. Crucially, Bench4HLS integrates a pluggable API for power, performance, and area (PPA) analysis across various HLS toolchains and architectures, demonstrated here with Xilinx Vitis HLS and validated on Catapult HLS. By providing a structured, extensible, and plug-and-play testbed, Bench4HLS establishes a foundational methodology for benchmarking LLMs in HLS workflows[1].**

***Index Terms*—Large Language Model (LLM), High Level Synthesis (HLS), Design Space Exploration (DSE), Power, Performance, Area (PPA) Efficiency.**

I. Introduction

High-level synthesis (HLS) emerged as a solution to the limitations of conventional register-transfer level (RTL) design by enabling automated hardware generation from high-level languages (HLLs) [1], [2]. Its goal is to boost productivity and ease application-specific design space exploration (DSE) without the need for RTL coding knowledge [3]. Over the past decade, HLS has matured significantly, supporting rapid design cycles and enabling advanced optimizations in hardware [4]–[7]. Nonetheless, achieving optimal power, performance, and area (PPA) remains a challenging, often requiring deep expertise and extensive manual intervention [6], [8].

Now, more than a decade after the emergence of HLS, the rapid rise of large language models (LLMs) and their widespread use in programming tasks have opened new opportunities for hardware design. Recent studies have begun exploring how LLMs can assist in HLS workflows (or even replace portions of them through direct code generation) [9]–[17]. These studies largely address isolated parts of the design pipeline, e.g., improving the synthesizability through `pragma` insertion and modification or code restructuring to be aligned with synthesis tools [10], [14], transforming (repairing or refactoring) general-purpose C/C++ code into HLS-compatible kernels through [12], [15], and even bypassing traditional HLS flows entirely by directly translating HLLs into RTL via reasoning-based approaches [11].

Despite the growing advancements in utilizing LLMs for HLS, systematic evaluation of such solutions remains limited. To show the gap, on the other side, RTL-level evaluation is already supported by established efforts such as VerilogEval [18], RTLLM [19], and more recently CVDP benchmark [20]. While some argue that direct LLM-to-RTL generation may reduce the role of HLS, this overlooks the fact that ***HLS tools are deeply optimized for FPGA-based acceleration and provide essential abstraction layers for DSE*** [21]. Thus, LLM-assisted HLS remains a critical research direction, and having a fully automated, systematic evaluation benchmark is a must. A recent benchmark, HLS-Eval [22], has taken first steps in this space, but its scope is restricted: While it emphasizes syntactic and synthesizability checks, it overlooks PPA analysis and relies only on small-scale kernels. Hence, there remains a gap in robust frameworks that holistically evaluate LLM-generated HLS code across the full synthesis pipeline, including quality-of-results (QoR) considerations essential for practical deployment [23].

To address this gap, we propose Bench4HLS, a comprehensive evaluation framework designed to benchmark LLMs in HLS workflows. While Table I reflects the main advantages of Bench4HLS vs. the prior art, our contributions are as follows:
(i) Diverse Benchmark Suite: Bench4HLS includes 170 curated and verified HLS design tasks spanning small kernels to accelerator-scale applications, collected from widely used open-source repositories and standardized with golden testbenches for reliable reproducibility.
(ii) Pluggable PPA evaluation: Bench4HLS integrates a modular API for PPA analysis across multiple HLS toolchains, shown (as the main testcase) here with Xilinx Vitis HLS and successfully tested on Catapult HLS.
(iii) Design Space Exploration (DSE): Bench4HLS explores synthesis-level (tool-in-the-loop) exploration within its evaluation loop, exposing Pareto frontiers under fixed design budgets.
(iv) Rigorous correctness validation: Bench4HLS enforces rigorous functional verification at multiple stages, pre-synthesis C simulation, HLS co-simulation, and post-implementation netlist validation, ensuring that only timing-clean, correctness-preserving designs are scored.

[1]The code and resources related to this work are publicly available at: https://github.com/zfsadik/Bench4HLS

978-3-9826741-1-7/DATE26/© 2026 EDAA

By combining dataset breadth, multi-objective evaluation, and automated optimization, Bench4HLS provides the first holistic and practically meaningful benchmark for assessing LLM-driven and automated HLS design methodologies[2].

II. Background and Related Work

A. LLMs for Direct RTL Code Generation

RTL is the most widely studied hardware target for LLM-based code generation [24], from fine-tuning [25], [26] to agentic [27], [28] and decoding [29] approaches. In terms of benchmarking and evaluation, several benchmarks established de-facto baselines. VerilogEval evaluates LLMs on HDLBits-style problems and, more recently, specification-to-RTL tasks with improved harnesses; it remains a common point of comparison [30]. RTLLM extends beyond syntax/functional checks to include a "design quality" goal and provides an open repository of tasks and testbenches [19].

B. LLMs for HLS Code Generation and Synthesis

Unlike RTL, there is a relatively limited prior studies focusing on the use of LLM for HLS. HLS reliance on implicit compiler optimizations, hierarchical transformations, and synthesis constraints, challenges LLMs that must generate syntactically correct code while reasoning about structural, timing, and resource constraints [31]. These "LLMs for HLS" efforts can be grouped into four key categories:

(1) Synthesizability and PPA Optimization. Initial studies show that LLMs can generate functionally correct HLS code, but without hardware-aware optimizations, performance and area lag behind expert designs [11]. Feedback-driven approaches such as HLSPilot improve results by incorporating profiling and synthesis constraints during generation [10].

(2) Code Refactoring and Repair. LLMs have been applied to transform general-purpose C/C++ into HLS-ready kernels, addressing unsupported constructs and improving synthesizability. While effective for basic transformations, challenges remain for complex optimizations such as hierarchical architectures and memory dataflow [12], [15].

(3) Structured Reasoning and Agents. Frameworks like SynthAI [13], LIFT [31] and agentic optimization methods extend LLMs with reasoning, graph analysis, and tool interaction. These systems automate pragma insertion, modularize tasks, and iteratively refine implementations, improving synthesis success rate and design quality [33].

(4) Dataset and Model Design. Specialized datasets and model architectures have been introduced to enhance HLS support. For example, SAGE-HLS leverages AST-guided training on synthetic HLS-C corpora, achieving high synthesis accuracy but still limited functional correctness [34].

[2]Benchmarking models and scripts will be released upon paper acceptance.

TABLE I: Comparison of LLM-based HLS Evaluation Frameworks.

Dimension	Gai *et al* [32]	HLS-Eval [22]	Bench4HLS (prop.)
# of Benchmarks	52	94	170
Benchmarks Size (LoC*/Benchmark)	Small to Medium-sized (56)	Small to Medium-sized (43)	Small to Large-sized (88)
Simulation	! Pre-HLS	✔ Pre- and Post-HLS	✔ Pre-, Post-HLS and Post-P&R
Tool (Synthesis)	! Partial Support	! Fixed Tool on Vitis	✔ Pluggable (Vitis/Catapult)
PPA Analysis	✘ No	✘ No	✔ Yes
DSE	✘ No	✘ No	✔ YAML-based
Code Generation	✘ No	✔ Yes	! Not Optimized+
Open-Source	✘ No	✔ Yes	✔ Yes
Parallel Test	✘ No	✔ Yes	✔ Yes

*: LoC/Benchmark: Average # of Code Lines in each Benchmark.
+: Can generate but not optimized as the main aim of Bench4HLS.

C. Benchmarking of LLM Generated HLS Codes

Benchmarking for LLM-based HLS evaluation is still at an early stage, with only a few targeted efforts: (1) Gai et al. [32] introduce one of the first benchmarks coupling natural-language prompts with HLS designs. Their framework focuses on syntactic and semantic correctness checks, focusing on reliable automated HLS code generation from text specifications; (2) HLS-Eval [22] provides an extensible Python-based infrastructure that integrates with commercial HLS tools. It offers a curated suite of designs with testbenches and natural-language prompts, enabling user-configurable evaluation of LLM-generated HLS code; (3) HLSTester [35] augments evaluation by detecting functional mismatches between C/C++ source and synthesized hardware. By combining LLM reasoning with adaptive fuzzing and guided input generation, it improves test coverage and error detection efficiency.

Existing efforts remain narrow in scope, where they primarily assess syntactic or functional correctness, while overlooking hardware-critical metrics such as PPA, as well as its DSE. This gap motivates the development of Bench4HLS, which extends beyond correctness to incorporate QoR metrics and multi-tool evaluation, offering a more rigorous and practically meaningful benchmark for LLM-based HLS. A detailed comparison with prior frameworks is shown in Table I.

III. Proposed Framework: Bench4HLS

To systematically evaluate LLM-based HLS code generation, Bench4HLS consists of four primary stages: (i) dataset construction, (ii) code generation via LLMs, (iii) functional verification and synthesis using Vitis HLS, and (iv) performance characterization and comparative analysis which is demonstrated in Figure 1. Throughout these stages, we form an end-to-end pipeline capable of assessing not only syntactic and semantic correctness, but also hardware-level metrics that are essential for real-world deployment.

A. Dataset: Instructions, Designs, and Testbenches

The first stage involves constructing a curated dataset composed of 170 designs at various scales, each represented as a triplet: a natural language instruction, a synthesizable HLS (C/C++) implementation, and a corresponding HLL (C/C++) testbench. The triplets were constructed as follows:

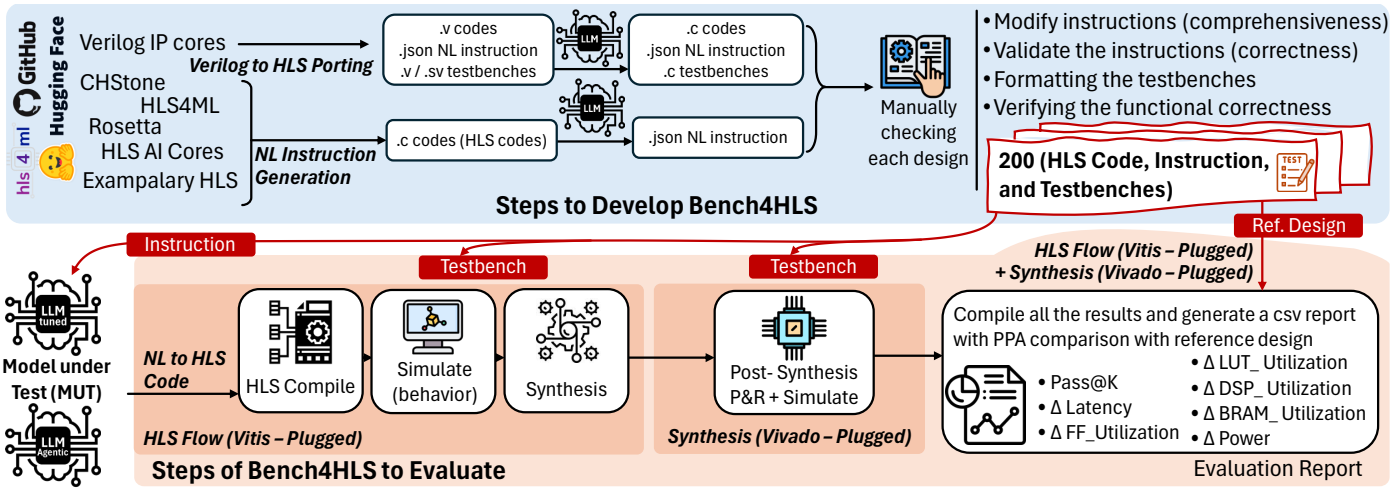


Fig. 1: Overview of Bench4HLS: The whole workflow of the evaluation method

(i) Reference Design Collection: We curated a comprehensive, multiscale dataset of heterogeneous test cases. It spans from small-scale Verilog datasets (e.g., VerilogEval [18]), Vitis-HLS-Introductory-Examples [36], and HLS codes from textbook references [37], [38], to mid- and large-scale accelerators and arithmetic circuits from established repositories such as CHStone [39], HLS4ML [40], and Rosetta [41]. This diversity yields an average of 88 lines of code (LoC) per test case (see Table I), establishing the largest and most representative dataset of its kind to date.

(ii) Formatting and Instruction Preparation: For the selected Verilog test cases, we ported each design into HLS-C++ using the GPT-5 model, followed by manual verification to ensure correctness (see Prompt 1 for an example). In total, 170 HLS codes were produced, from which GPT-5 subsequently generated natural language instructions for each design. To enable comprehensive evaluation, corresponding testbenches were also constructed and manually validated, ensuring the reliability and completeness of the dataset.

Prompt 1: Verilog to HLS (C/C++) Porting.

```
“
You are an expert in HLS-C++ and Verilog code generation. I will give you a prompt for a hardware design written for verilog, the reference code in verilog and it’s testbench.

Your task is to generate the reference design and the testbench for HLS-C++. Your generated testbench must reflect the same logic of the reference code.

All your generated codes will be run in HLS tools. While writing the testbenches consider all possible scenario. Your testbench has to be self checking. Write the codes without the ’RefModule’.

The testbench will detect errors of design file but it must not show compilation and simulation error. It should also always return 0. Use all necessary and applicable pragmas for better optimization in your design code.

The three items for verilog are below:

{Prompt}
{Reference Code}
{Testbench}
”
```

(iii) Manual Dataset Verification: Each instruction was validated across multiple LLM prompting (self-refinement [42]) to confirm its robustness and faithfulness to the design intent. The testbenches were constructed to mirror functional coverage and contain embedded logic derived from the corresponding design. The correctness of each triplet was verified through manual inspection and simulation. While manual, all 170 entries of our curated dataset is tailored for both functional and PPA evaluation of instruction-to-HLS generation tasks.

B. LLM-Based HLS Code Generation

Although efficient code generation is not the primary objective of Bench4HLS, the framework is designed with a flexible, pluggable API that enables the integration of LLMs for HLS code generation across its 200-task dataset. At this stage, the supported natural language instructions serve as task prompts, which are passed through the API to the integrated LLMs for producing the corresponding HLS code implementations. Since no open-source LLMs are currently dedicated to HLS, we employ widely used open- and closed-source coding models, e.g., GPT-5, QwenCoder, and Llama, as case studies.

Using this pluggable model demonstrates how Bench4HLS can be applied in realistic scenarios, where a model-under-test (MUT) is systematically evaluated for functionality, synthesizability, as well as PPA efficiency through the framework’s evaluation pipeline.

(i) Plugging Model (API): For testing phase, we integrated and evaluated several models, including proprietary and open-source models such as GPT-5, QwenCoder, and Llama. The Python API (vLLM and simonw/llm Python libraries) is fully standardized for instruction-based inputs and code-oriented responses, abstracting away vendor-specific differences (especially w.r.t. HLS tools differences in syntax).

(ii) Prompting and Parsing: Each model was queried using the instruction generated by the prompt described in Prompt 2, which directed the model to produce high-level, system-design explanations of the HLS C++ module. Outputs were post-processed to remove commentary, headers, or test scaffolding, ensuring that the retained content focused solely on the module’s functionality, dataflow, pipelining, memory behavior, and

control logic. Custom scripts further standardized formatting and terminology for consistency across results.

Prompt 2: Instruction (Spec.) Generation for HLS Test Cases.

"

You are an expert in writing and explaining HLS-C++ hardware deisgn. Describe the functionality of the HLS design from a high-level, system-design perspective.

Focus on what the module computes, how data flows through it, and how control logic operates across different stages of execution. Explain its functional behavior considering pipelining, memory access, and synthesis constraints.

Avoid unnecessary low-level details like variable names or line-by-line mappings. Instead, communicate the purpose, the algorithmic steps, the I/O behavior, and any state or timing dependencies using precise, practical language that would guide someone implementing or verifying the design.

Include timing models, pipeline stages, and basic handshaking logic if relevant, but keep the explanation focused and hardware-aware.

{Reference HLS-C++ Code}

"

(ii) Dataset Organization: The extracted designs were stored using a systematic naming scheme, enabling seamless batch processing in downstream compilation and evaluation steps. This stage produces a structured set of LLM-generated HLS codes that serve as the main subject for subsequent functional and performance evaluation.

C. Automated Verification and Synthesis via Vitis HLS

The Bench4HLS framework provides a fully automated flow for evaluating high-level synthesis. By design, the framework is pluggable, allowing different HLS and implementation toolchains to be seamlessly integrated. In the current instantiation, the front end is connected to Vitis HLS for compilation, simulation, and synthesis, while the back end is linked to Vivado for post-RTL placement and routing, enabling detailed PPA analysis. This integration delivers a holistic view of design quality, combining cycle-accurate semantic validation with synthesis-proven correctness. Importantly, the framework is architected for batch-mode evaluation, supporting parallel execution across multiple designs. This capability is particularly critical as HLS flows can be computationally demanding for complex benchmarks. The evaluation proceeds through three principal stages: *(1) Compilation:* Designs is compiled to detect syntactical errors. Failing designs are recorded as compilation failures. *(2) Pre-Synthesis Simulation:* compiled designs are simulated using the corresponding testbenches. This verifies the functional behavior of the generated code against expected outputs. *(3) Synthesis and Post-Synthesis Simulation:* Designs that pass simulation are synthesized to obtain RTL-level implementations. This is followed by RTL synthesis for resource utilization metrics (e.g., LUTs, FFs, BRAMs), and then post-synthesis simulation for cycle-accurate functional verification.

All steps are executed through an automated script pipeline comprising batch files and TCL scripts. This ensures reproducibility and scalability of the benchmarking process across hundreds of designs and models. This stage acts as a filter that identifies which LLM-based generated codes are not only syntactically valid, but also functionally correct and suitable for hardware implementation at acceptable PPA.

Algorithm 1 Automated Evaluation w/ DSE in Bench4HLS.

Require: HLS design set $\mathcal{D}$, testbench set $\mathcal{T}$, Pass@K, exploration policy π, part FPGA.
Ensure: Structured report $\mathcal{R}$ with per-solution status and PPA metrics.

Initialize $\mathcal{R} \leftarrow \emptyset$
Partition $\mathcal{D}$ into K batches: $\{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$ ▷ Top-K / Pass@K batching
for $r \leftarrow 1$ **to** K **do**
  for $d_i \in \mathcal{D}_r$ **do**
    Retrieve testbench $t_i \in \mathcal{T}$ and YAML DSE spec Y_i
    Expand Y_i into solution set $\Omega_i \leftarrow \textsc{Expand}(Y_i, \pi)$
    Initialize per-design results $\mathcal{R}_i \leftarrow \emptyset$
    for each solution $s \in \Omega_i$ **do**
      Open Plugged HLS project with d_i and apply directives from s
      CSIM_DESIGN $\rightarrow$ compilation status $\sigma^{\text{com}}_{i,s}$
      Append $(d_i, s, \sigma^{\text{com}}_{i,s})$ to $\mathcal{R}_i$
      if $\sigma^{\text{com}}_{i,s} = \texttt{PASS}$ **then**
        Simulate d_i with t_i in Plugged HLS $\rightarrow$ status $\sigma^{\text{sim}}_{i,s}$
        Append $(d_i, s, \sigma^{\text{sim}}_{i,s})$ to $\mathcal{R}_i$
        if $\sigma^{\text{sim}}_{i,s} = \texttt{PASS}$ **then**
          CSYNTH_DESIGN $\rightarrow$ status $\sigma^{\text{syn}}_{i,s}$, latency $L_{i,s}$, area vector $A_{i,s}$
          Append $(d_i, s, \sigma^{\text{syn}}_{i,s}, L_{i,s}, A_{i,s})$ to $\mathcal{R}_i$
          Export RTL $\rightarrow$ Synth_Tool; set strategy from s
          Parse FPGA timing/power $\rightarrow$ $\text{WNS}_{i,s}$, $F_{\max\ i,s}$, $P_{i,s}$
          Append $(d_i, s, \text{WNS}_{i,s}, F_{\max\ i,s}, P_{i,s})$ to $\mathcal{R}_i$
    Append $(d_i, \mathcal{R}_i, \Pi_i)$ to $\mathcal{R}$
Compute Pass@K across stages using $\{\sigma^{\text{com}}_{i,s}, \sigma^{\text{sim}}_{i,s}, \sigma^{\text{syn}}_{i,s}\}$ aggregated per design
Append Pass@K summary rows and per-design bests to $\mathcal{R}$
return $\mathcal{R}$

D. DSE-based Characterization and Comparative Analysis

Bench4HLS evaluates LLM-generated HLS designs not only in terms of functional correctness but also through detailed PPA metrics[3]. Primarily, the framework integrates DSE to assess how plugged compiler directives and implementation strategies affect these outcomes. Some of the main metrics evaluated in Bench4HLS are (i) latency (in nanoseconds), (ii) hardware resource utilization (LUTs, FFs, DSPs, BRAMs), (iii) power analysis of synthesized RTL. Additionally, Bench4HLS computes Pass@K metrics across compilation, simulation, and synthesis stages.

Additionally, each design can be associated with a declarative YAML specification that enumerates the design space to be further explored. Bench4HLS sweeps through this space by instantiating multiple solutions (serial or in parallel), applying directives in plugged HLS (e.g., Vitis HLS), collecting synthesis reports, and comparing the reports for DSE. The overview of this automated process is outlined in Algorithm 1 Various directives and pragmas will be included w.r.t. the specification of design, e.g., pipelining, tunable intervals, dataflow, unrolling, etc. As shown in Listing 1, a detailed YAML specification has been used in Bench4HLS for this process:

```yaml
clock_period_ns: [3.3, 5.0]
enable_pipeline: [true, false]
pipeline_ii: [1, 2]
enable_dataflow: [true, false]
unroll_factor: [1, 2, 4, 8]
array_partition_factor: [1, 2, 4]
allocation_limit_add: [0, 1, 2] # 0 = disabled
dsp_full_reg: [true, false]
vivado_strategy: [Default, Performance_Explore,
    Area_Explore]
```

Listing 1: Example YAML configuration for DSE in Bench4HLS

[3]All PPA-related results are compared with its corresponding reference implementation (provided by Bench4HLS) using normalized PPA metrics.

TABLE II: Pass@K evaluation of Compilation, Simulation and Synthesis Status for selected Large Language Models

Evaluated Model	Compilation Status			Simulation Status			Synthesis Status		
	Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10
Qwen 2.5 Coder 14B	47.06%	58.82%	65.88%	18.82%	25.88%	30.59%	17.65%	24.71%	28.82%
Qwen 2.5 Coder 32B	82.35%	90.59%	92.35%	28.24%	36.06%	39.41%	24.12%	34.71%	38.24%
Llama 3.3 70B	77.65%	95.88%	97.65%	45.29%	58.24%	64.12%	44.12%	57.65%	63.53%
GPT-5	85.29%	97.65%	97.65%	52.35%	65.88%	72.35%	50.00%	65.88%	71.76%

By exposing the DSE space in a structured, machine-readable YAML format, Bench4HLS enables reproducible and extensible DSE. This design makes Bench4HLS not only a benchmarking platform, but also a practical framework for systematic DSE studies of LLM-generated HLS designs.

Prompt 3: Instruction used for Prompting the Model for Test.

```
"
Your role is to act as an expert in HLS C++ code development. You must thoroughly explore each question through a systematic and deliberate thinking process—engaging in cycles of analysis, summarization, exploration, reassessment, reflection, backtracing, and iteration to develop well-considered solutions.

Based on the provided instructions, you are expected to generate precise, optimized, and accurate synthesizable HLS C++ code that meets the following requirements.

1. Output only synthesizable C++ code (no extra text or explanation).
2. Include necessary headers (e.g. ap_int.h).
3. The top-level function signature must be exactly as given in the instruction.
4. Insert suitable HLS pragmas.
5. Self-contained: do not rely on external libraries or files beyond standard headers.

### Instruction:{}
### Input:{}
### Response:{}
"
```

IV. Experimental Setup

To demonstrate the comprehensiveness of Bench4HLS, we evaluated the HLS code generation of four LLMs: Qwen2.5-Coder (14B and 32B), LlaMA 3.3 70B, and GPT-5. The Qwen and LlaMA models were deployed locally with 4-bit quantization for efficient inference, while GPT-5 was accessed via OpenAI's API. For each of the 170 benchmark tasks, we used a structured prompt mentioned in Prompt 3 that explicitly instructed the model to produce synthesizable, pragma-annotated HLS C++ code (combined with YAML for DSE). While pluggable, for this paper and at post-generation and synthesis stage, all designs were evaluated using Vitis HLS 2024.1 for compilation, simulation, and synthesis, and Vivado 2024.1 for power estimation. All designs targeted the Xilinx Artix-7 xc7a200tffv1156-1 FPGA to ensure consistent comparison. The entire workflow of Bench4HLS is automated using custom pluggable scripts with TCL drivers, enabling scalable evaluation across multiple models and Pass@K configurations.

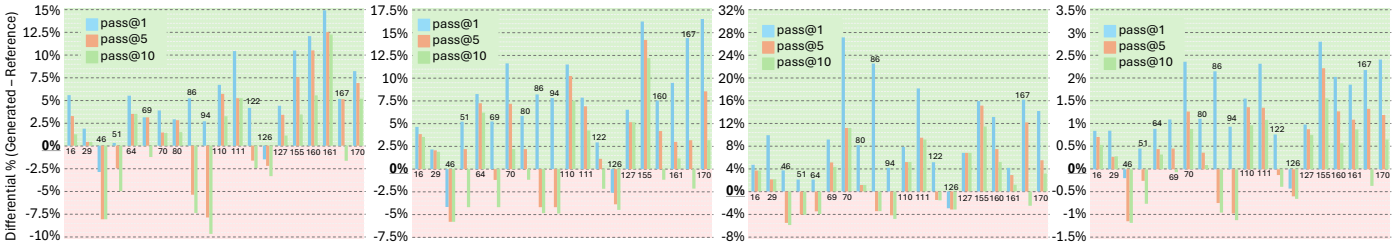
V. Experimental Results

We evaluated Bench4HLS using four state-of-the-art LLMs: **Qwen2.5-Coder 14B**, **Qwen2.5-Coder 32B**, **LlaMA 3.3 70B**, and **GPT-5**. The goal is to assess the ability of Bench4HLS for evaluating the capacity of given models for translating natural language descriptions into functionally correct and synthesizable HLS designs. Bench4HLS provides a fully automated pipeline for design generation, simulation, synthesis, and PPA analysis, enabling reproducible and comprehensive benchmarking. The evaluation flow is organized as a four-stage pipeline: (i) **Syntactic validity**: compilation success via CSIM_DESIGN; (ii) **Semantic correctness**: functional validation using testbench-driven simulation; (iii) **Hardware feasibility**: synthesizability through RTL generation in Vivado; and (iv) **DSE**: automated exploration of HLS directives and Vivado strategies using YAML specifications (see Listing 1) for optimum PPA. For each stage, Bench4HLS computes Pass@K metrics ($K \in \{1, 5, 10\}$), measuring the likelihood that at least one of the top-K generated candidates passes. Table II summarizes the results across compilation, simulation, and synthesis stages. Bench4HLS allows us to integrate any new MUT to be evaluated based on this four-stage pipeline.

Bench4HLS provides model-wise comparison as shown in Table II. For this examplary case, **GPT-5** achieves the strongest performance, with 97.65% compilation success at Pass@10, and 72.35% and 71.76% success in simulation and synthesis respectively. **LlaMA 3.3 70B** follows closely, while the **Qwen2.5** series shows the impact of scaling: moving from 14B to 32B yields significant improvements in syntactic and semantic fidelity. These results highlight the sensitivity of HLS-oriented code generation to model size and capacity.

To further reflect the capability of Bench4HLS, automated YAML-based HLS DSE has been evaluated here. As shown in Figure 3, for four given model, Bench4HLS can simulate and synthesize benchmarks with configurable parameter (as exemplified in Listing 1). Figure 3 reflects the improved results while DSE is applied (in this case, when at least one of PPA metrics are improved less/more than 20%). As shown, for GPT-5, DSE improves at least one PPA metric by $> 20\%$ in $\sim 40\%$ of benchmarks, compared to only $\sim 10\%$ for Qwen2.5-14B. This demonstrates the importance of directive-level search for realizing the full potential of LLM-generated designs. This automated DSE allows us to exhaustively test the MUT effectiveness across various configurations.

Additionally, beyond HLS synthesis, Bench4HLS integrates post-RTL implementation in given (pluggable) synthesis tool (here is Vivado) using a configurable target FPGA (here is *Xilinx Artix-7 xc7a200tffv1156-1*). Four primary metrics are extracted relative to reference designs: latency (% of improvement vs. reference), FF utilization (% of improvement vs. reference), LUT utilization (% of improvement vs. reference),



(a) LUT Utilization (b) FF Utilization (c) Post-Synthesis Power Consumption (d) Latency

Fig. 2: PPA Analysis of LLM-generated (GPT) HLS Codes by Bench4HLS. Bench4HLS reports percentage differences between the generated designs by MUT and the reference designs provided by Bench4HLS across four metrics (LUT utilization, FF utilization, post-synthesis power, and latency – for pass@1, pass@5, and pass@10 generations), with values representing (generated – reference)%. Positive means higher resource usage or latency vs. the reference.

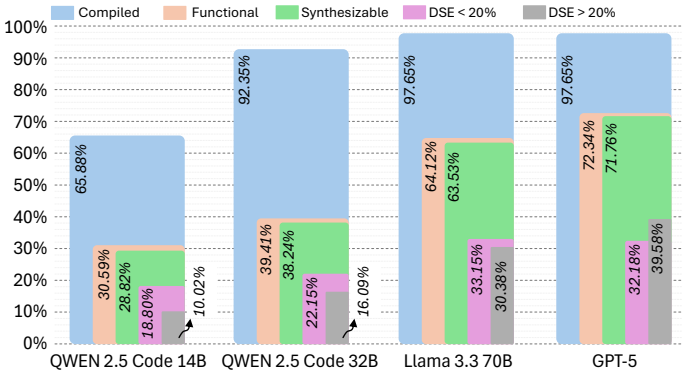
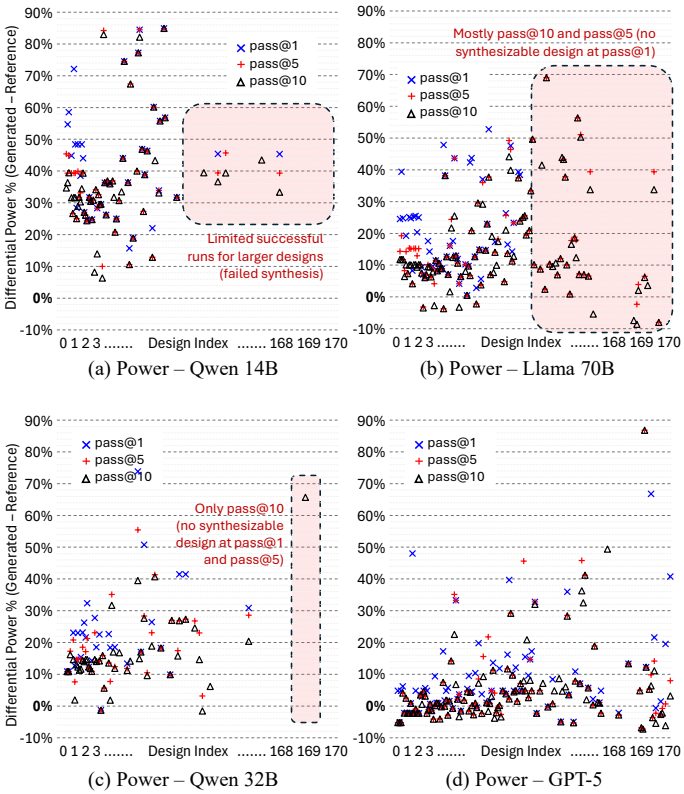


Fig. 3: HLS Code Generation Evaluated by Bench4HLS for Pass@10 (DSE refers to improvement of at least one of PPA metrics less/more than 20%).

and estimated power consumption (% of improvement vs. reference). Figure 2 demonstrates these metrics for the designs generated by GPT-5. A key strength of Bench4HLS lies in its curated set of reference designs, which serve as a high-quality baseline for comparison. As shown in Figure 2, the reference implementations consistently outperform even the strongest model (GPT-5) in terms of PPA efficiency. For smaller benchmarks (problems #1–#100), LLM-generated solutions achieve metrics close to the reference baseline, but for larger and more complex benchmarks (#150+), most models struggle to produce synthesizable designs. In selected designs, the resulting PPAs are significantly worse than the reference, underscoring the efficiency of the curated designs. This trend is even more highlighted for smaller models such as Qwen2.5-14B/32B and LlaMA 3.3 70B, which fail more frequently and diverge further from the baseline. These results highlight that the reference designs in Bench4HLS are not merely functional baselines, but **well-optimized, efficiency-oriented implementations** that establish a meaningful upper bound against which automated approaches can be evaluated.

Figure 4 reveals this scalability issue better, where power distributions across all benchmarks have been illustrated. As shown, smaller models (e.g., Qwen2.5-14B) fail to produce synthesizable designs for many larger problems (sparse results beyond design #120), while GPT-5 produces more valid RTL for a broader range of cases. The breadth of successful synthesis reflects Bench4HLS's automated pipeline for consistent PPA extraction and synthesis validation. These experiments demonstrate that Bench4HLS provides an effective and extensible framework for evaluating LLM-generated HLS designs. By combining Pass@K correctness evaluation with YAML-driven DSE and post-RTL PPA analysis, the framework enables holistic comparison across models.



(a) Power – Qwen 14B (b) Power – Llama 70B

(c) Power – Qwen 32B (d) Power – GPT-5

Fig. 4: Power Consumption Benchmarking in MUTs Tested by Bench4HLS.

VI. Conclusion

This paper proposes Bench4HLS, a comprehensive benchmarking and evaluation framework dedicated to assessing LLMs for HLS code generation. By combining a curated set of 170 optimized reference designs with an automated four-stage evaluation flow using pluggable industry-standard tools for compilation, simulation, synthesis, and YAML-based DSE, Bench4HLS enables systematic variation of HLS directives and backend strategies for rigorous measurement of correctness, feasibility, and PPA efficiency. To assess the capabilities of Bench4HLS, four advanced LLMs have been evaluated, i.e., GPT-5, Llama 3.3 70B, and Qwen 2.5 14B/32B. By offering an extensible, plug-and-play testbed, Bench4HLS establishes a foundation for the systematic study of LLMs in HLS design.

REFERENCES

[1] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[2] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2018.

[3] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[4] S. Liu, F. C. Lau, and B. C. Schafer, "Accelerating fpga prototyping through predictive model-based hls design space exploration," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[5] J. J. Zhang *et al.*, "Towards automatic and agile ai/ml accelerator design with end-to-end synthesis," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2021, pp. 218–225.

[6] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "Fpga hls today: successes, challenges, and opportunities," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 4, pp. 1–42, 2022.

[7] A. Cortes, I. Velez, and A. Irizar, "High level synthesis using vivado hls for zynq soc: Image processing case studies," in *2016 Conference on design of circuits and integrated systems (DCIS)*. IEEE, 2016, pp. 1–6.

[8] S. Shi, N. Pundir, H. M. Kamali, M. Tehranipoor, and F. Farahmandi, "Sechls: Enabling security awareness in high-level synthesis," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, 2023, pp. 585–590.

[9] S. Swaroopa, R. Mukherjee, A. Debnath, and R. S. Chakraborty, "Evaluating large language models for automatic register transfer logic generation via high-level synthesis," *arXiv preprint arXiv:2408.02793*, 2024.

[10] C. Xiong, C. Liu, H. Li, and X. Li, "Hlspilot: Llm-based high-level synthesis," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[11] Y. Liao, T. Adegbija, and R. Lysecky, "Are llms any good for high-level synthesis?" in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–8.

[12] Kangwei Xu, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, Ulf Schlichtmann and Bing Li, "Automated c/c++ program repair for high-level synthesis via large language models," in *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, 2024, pp. 1–9.

[13] S. A. Sheikholeslam and A. Ivanov, "Synthai: A multi agent generative ai framework for automated modular hls design generation," *arXiv preprint arXiv:2405.16072*, 2024.

[14] H. Xu, H. Hu, and S. Huang, "Optimizing high-level synthesis designs with retrieval-augmented large language models," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–5.

[15] L. Collini, S. Garg, and R. Karri, "C2hlsc: Can llms bridge the software-to-hardware design gap?" in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–12.

[16] A. E. Oztas and M. Jelodari, "Agentic-hls: An agentic reasoning based high-level synthesis system using large language models (ai for eda workshop 2024)," *arXiv preprint arXiv:2412.01604*, 2024.

[17] N. Mashnoor, M. Akyash, H. Kamali, and K. Azar, "Timelyhls: Llm-based timing-aware and architecture-specific fpga hls optimization," in *2025 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, 2025, pp. 1–6.

[18] M. Liu *et al.*, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[19] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 722–727.

[20] N. Pinckney, C. Deng, C.-T. Ho, Y.-D. Tsai, M. Liu, W. Zhou, B. Khailany, and H. Ren, "Comprehensive verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on rtl design and verification," *arXiv preprint arXiv:2506.14074*, 2025.

[21] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2628–2639, 2019.

[22] S. Abi-Karam and C. Hao, "Hls-eval: A benchmark and framework for evaluating llms on high-level synthesis design tasks," *arXiv preprint arXiv:2504.12268*, 2025.

[23] J. Cong *et al.*, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[24] J. Pan, G. Zhou, C.-C. Chang, I. Jacobson, J. Hu, and Y. Chen, "A survey of research in large language models for electronic design automation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 3, pp. 1–21, 2025.

[25] S. Liu, W. Fang, Y. Lu, J. Wang, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[26] M. Akyash, K. Azar, and H. Kamali, "Rtl++: Graph-enhanced llm for rtl code generation," in *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. IEEE, 2025, pp. 44–20.

[27] Y. Zhao, H. Zhang, H. Huang, Z. Yu, and J. Zhao, "Mage: A multi-agent engine for automated rtl code generation," in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2025, pp. 1–7.

[28] H. Ping *et al.*, "Hdlcore: A training-free framework for mitigating hallucinations in llm-generated hdl," *arXiv preprint arXiv:2503.16528*, 2025.

[29] M. Akyash, K. Azar, and H. Kamali, "Decortl: A run-time decoding framework for rtl code generation with llms," *arXiv preprint arXiv:2507.02226*, 2025.

[30] N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany, "Revisiting verilogeval: A year of improvements in large-language models for hardware code generation," *ACM Trans. Des. Autom. Electron. Syst.*, Feb. 2025. [Online]. Available: https://doi.org/10.1145/3718088

[31] N. Prakriya, Z. Ding, Y. Sun, and J. Cong, "Lift: Llm-based pragma insertion for hls via gnn supervised fine-tuning," *arXiv preprint arXiv:2504.21187*, 2025.

[32] J. Gai *et al.*, "Exploring code language models for automated hls-based hardware generation: Benchmark, infrastructure and analysis," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, 2025, pp. 988–994.

[33] L. Collini, A. Hennessee, R. Karri, and S. Garg, "Can reasoning models reason about hardware? an agentic hls perspective," *arXiv preprint arXiv:2503.12721*, 2025.

[34] M. Khan, N. Mashnoor, M. Akyash, K. Azar, and H. Kamali, "Sage-hls: Syntax-aware ast-guided llm for high-level synthesis code generation," *arXiv preprint arXiv:2508.03558*, 2025.

[35] K. Xu, B. Li, G. L. Zhang, and U. Schlichtmann, "Hlstester: Efficient testing of behavioral discrepancies with llms for high-level synthesis," *arXiv preprint arXiv:2504.14641*, 2025.

[36] Xilinx, "Vitis hls introductory examples," https://github.com/Xilinx/Vitis-HLS-Introductory-Examples, 2024, accessed: May 18, 2025.

[37] M. Fingeroff, *High-level synthesis: blue book*. Xlibris Corporation, 2010.

[38] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel programming for fpgas," *arXiv preprint arXiv:1805.03648*, 2018.

[39] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[40] F. Fahim *et al.*, "hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," *arXiv preprint arXiv:2103.05579*, 2021.

[41] Y. Zhou *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 269–278.

[42] A. Madaan *et al.*, "Self-refine: Iterative refinement with self-feedback," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 534–46 594, 2023.