

# ChatHLS: Towards Systematic Design Automation and Optimization for High-Level Synthesis

Runkai Li<sup>†1,2</sup>, Jia Xiong<sup>†1,2</sup>, Xiuyuan He<sup>2</sup>, Jieru Zhao<sup>3</sup>, Jiaqi Lv<sup>1</sup>,  
Haowen Fang<sup>2</sup>, Lei Qi<sup>1</sup>, Xi Wang<sup>\*1,2</sup>

<sup>1</sup>Southeast University, Nanjing, China

<sup>2</sup>National Center of Technology Innovation for EDA, Nanjing, China

<sup>3</sup>Shanghai Jiao Tong University, Shanghai, China

## Abstract

High-Level Synthesis (HLS) improves IC development productivity by enabling hardware design from C-like languages. However, strict coding constraints and design-specific optimizations limit its widespread adoption. While recent efforts employ large language models (LLMs) to assist HLS design, they often struggle with synthesizability rules and directive semantics. To this end, we introduce ChatHLS, a multi-agent HLS design framework that leverages specialized LLMs for automated debugging and directive tuning. ChatHLS incorporates an adaptive error case expansion mechanism, combined with a reasoning-to-instruction analysis method to accurately diagnose HLS errors. To optimize hardware performance, it enables QoR-aware reasoning to learn the impact of HLS directives on the quality of results (QoR). Experimental results demonstrate that ChatHLS outperforms Gemini-3-pro with a 32.6% relative improvement in debugging, while achieving significant speedups across various HLS kernels and neural network accelerators. These results underscore the potential of ChatHLS for agile hardware development.

## 1 Introduction

High-Level Synthesis (HLS) accelerates hardware design by abstracting hardware description languages (HDLs) to C/C++ (Cong et al., 2022). By allowing designers to focus on algorithmic logic rather than cycle-by-cycle circuit behavior, HLS enables rapid development with reduced coding complexity and shorter simulation cycles (Chen et al., 2024). This design agility enables rapid iteration in demanding applications such as deep learning and high-frequency trading (Chi et al., 2022). Although HLS improves hardware development efficiency, its practical adoption is severely stymied by a prohibitive design space exploration (DSE) challenge.

<sup>†</sup> Equal Contribution.

<sup>\*</sup> Corresponding Author: xi.wang@seu.edu.cn

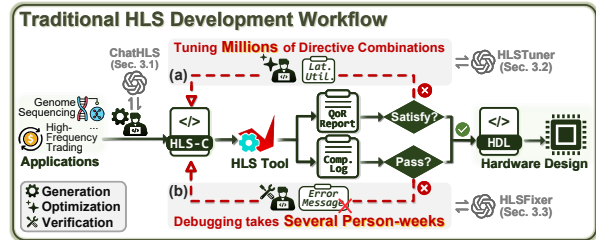


Figure 1: Traditional HLS generation, optimization, and verification workflow. Bottlenecks are: (a) balancing performance and resource utilization requires time-consuming directive tuning and (b) fixing simulation and synthesis errors relies on HLS domain expertise.

Achieving optimal performance requires navigating an exponentially large search space of directive combinations (e.g., loop unrolling, pipelining, and tiling factors). These directives often exhibit complex interdependencies, where a single parameter change can lead to drastic fluctuations in resource utilization and throughput. Crucially, the iterative directive tuning is hindered by high synthesis latency, as a single trial often consumes minutes to hours. This creates a bottleneck of exhaustive trial-and-error, making it nearly impossible for manual designers to reach the optimal HLS design within a reasonable development cycle (Figure 1).

The success of large language models (LLMs) in mainstream programming languages has inspired LLM-aided design (LAD) in IC design and verification (Wang et al., 2024; Xing et al., 2026; Fu et al., 2026; Wan et al., 2026). This trend provides a novel solution to address the challenges in HLS development. Recent efforts have introduced LLMs to automate the refactoring of the C/C++ code into HLS-C and the insertion of directives for hardware optimization (Xiong et al., 2024; Collini et al., 2025; Xu et al., 2024a). While these approaches explore the feasibility, the scarcity of HLS-specific datasets limits the correctness of generated HLS-C and the effectiveness of performance optimization (Wang et al., 2025a). Specifically, we identify three critical bottlenecks:

**Challenge 1: HLS Data Scarcity.** Constructing high-quality HLS datasets is extremely labor-intensive, relying on manual construction by domain experts (Xu et al., 2025). More critically, existing datasets rarely expose synthesizability constraints, the rationale behind directive selections, and their correlation with quality of results (QoR) (e.g., latency and resource usage). This scarcity hinders LLMs from learning hardware constraints and the intricate semantics of HLS directives. Consequently, the generated code frequently suffers from compatibility errors, as evidenced by pass rates generally **below 60%** shown in Figure 2.

**Challenge 2: Inefficient Performance Optimization.** HLS optimization suffers from a combinatorial explosion of directive choices, whose QoR effects are highly non-linear and design-dependent. Consequently, directive tuning becomes inherently time-consuming. Existing LLMs lack the architectural intuition required for design-specific tuning, struggling to determine the optimal combination, configuration, and insertion of directives, resulting in suboptimal hardware performance.

**Challenge 3: Limited HLS-C debugging capabilities.** Since HLS prohibits non-synthesizable constructs (e.g., dynamic arrays), general-purpose LLMs, which are pretrained on large-scale standard C/C++ datasets, struggle to identify and correct these HLS compatibility errors. Meanwhile, invalid optimizations or synthesis errors caused by incorrect directive syntax, placement, or functional conflicts further limit their debugging ability.

To address the challenges in LLM-driven HLS design, we propose **ChatHLS**, a multi-agent framework for agile HLS-C generation and optimization. ChatHLS incorporates a verification dataset construction method based on dual-agent collaboration, directing LLMs toward more comprehensive HLS error correction. By analyzing verification feedback from HLS tools, we extract expert-like debugging reasoning patterns to augment LLMs for reasonable error diagnosis. To achieve effective HLS design optimization, ChatHLS learns to perform *QoR-aware reasoning*. This enables automated directive tuning, striking a balance between performance gains and resource consumption. Our contributions are summarized as follows:

- We propose **HLSTuner**, a *QoR-aware directive optimization framework* that explicitly models the directive  $\rightarrow$  synthesized hardware  $\rightarrow$  resulting QoR and performs constraint-aware search to navigate performance-cost trade-offs.

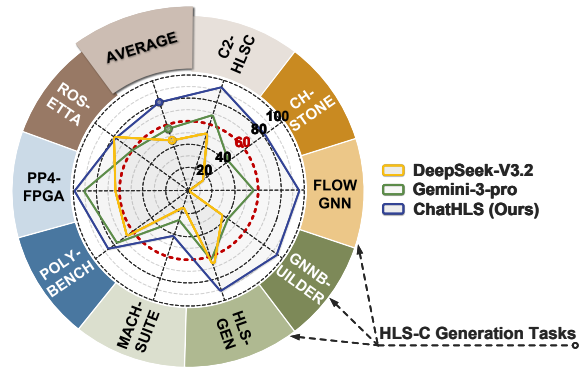


Figure 2: Average simulation pass rates of existing LLMs generating HLS-C from natural language algorithm descriptions (Abi-Karam and Hao, 2025), calculated across 108 tasks and averaged over 20 repetitions (see Appendix D.3 for detailed experimental settings).

- We introduce **HLSFixer**, a *hierarchical feedback-augmented debugging framework* that grounds LLM reasoning in tool feedback. HLSFixer formulates debugging as error diagnosis and correction instruction, using a reasoning-to-instruction procedure to correct HLS-specific errors.
- We propose **Verification-Oriented Data Augmentation (VODA)**, a *self-evolving error case expansion mechanism* that automates the capture of error cases detected in HLS design to strengthen the debugging capabilities of LLMs.
- Experimental results show that ChatHLS outperforms Gemini-3-pro, improving the HLS-C generation success rate by 41.8% and error analysis accuracy by 32.6%, while achieving a **3.3 $\times$**  performance gain over the RAG-based method.

## 2 Related Work

### 2.1 Traditional Alignment to HLS Design

Unlike C/C++ programming, HLS developers must refactor C code to align with HLS paradigms. HeteroRefactor automates the refactoring of C to HLS-C through dynamic invariant analysis (Lau et al., 2020). HeteroGen advances this approach with fuzzing tests for automated test input generation and exception handling (Zhang et al., 2022). However, both solutions require predefined templates and manual oversight to ensure the synthesizability. Domain-specific languages (DSLs) further abstract algorithmic representations and hardware optimization in HLS (Chen et al., 2024; Nigam et al., 2020; Lai et al., 2019; Ye et al., 2022). While DSLs mitigate certain coding pitfalls, they introduce additional learning curves and exhibit limited expressivity, restricting applicability to nuanced use cases.

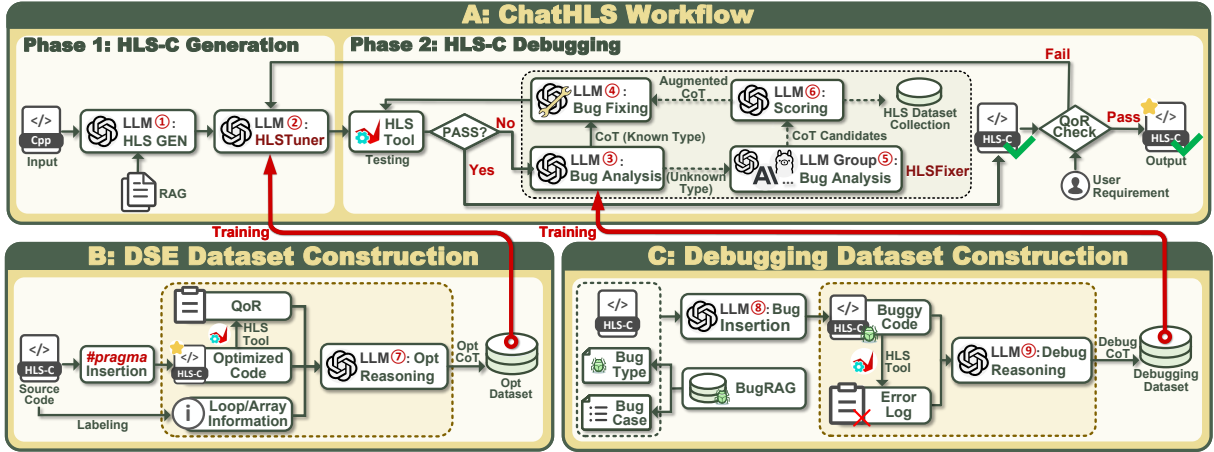


Figure 3: ChatHLS workflow and dataset construction.

To achieve satisfactory synthesized hardware performance, HLS designers also need to consider the quality of results (QoR) and strategically apply hardware-specific directives. However, the combinatorial explosion of directives constitutes an overwhelming design space (Schafer and Wang, 2020). Traditional methods rely on heuristics (Sohrabzadeh et al., 2022) or prediction models (Kuang et al., 2023; Li et al., 2025b) to identify optimal combinations and configurations of directives. Nevertheless, heuristic-based approaches require numerous iterations to converge, while learning-based approaches have limited generalization beyond the training distribution.

## 2.2 LLM-Aided HLS Design

LLM-aided design (LAD) has garnered significant attention in low-level HDL code generation and verification (Yao et al., 2025a; Wang et al., 2025b), which has also catalyzed research interest in HLS. Previous work has incorporated retrieval-augmented generation (RAG) to provide HLS domain knowledge, with the aim of debugging and design optimization (Wang et al., 2025a; Xiong et al., 2024; Xu et al., 2024a; Li et al., 2025a). However, RAG struggles to provide accurate search results, which may impair the reasoning of LLMs due to partially matched contexts (Qiu et al., 2025; Mavromatis and Karypis, 2025). Therefore, some efforts have improved the accuracy of generating HLS-C from natural language by fine-tuning LLMs. While these approaches support syntax and function error correction, they lack a comprehensive analysis of HLS compatibility issues.

Some studies treated LLMs as directive comparators within a Bayesian optimization framework (Yao et al., 2025b), or adopted graph-level supervi-

sion to fine-tune LLMs (Prakriya et al., 2025) for HLS design optimization. While these methods demonstrate improvements in both optimization quality and efficiency, they fail to establish the interrelationship among HLS designs, directives, and corresponding QoR, limiting their effectiveness in design-specific optimization.

## 3 Design & Philosophy

### 3.1 ChatHLS Architecture & Workflow

To address the aforementioned challenges, we propose the ChatHLS workflow to optimize HLS designs while incorporating robust code error correction capability. As illustrated in Figure 3.A, the architecture comprises two primary phases: *HLS-C generation* and *HLS-C debugging*.

In the **HLS-C generation** phase, LLM ① leverages retrieved HLS-related context to transform input C algorithms or natural language descriptions. A fine-tuned LLM ② then selects effective HLS directive combination strategies and inserts directives within the specific structure (e.g., loops and arrays) of the generated HLS-C. However, the inherent hallucinations of LLMs and their misalignment with HLS specifications may introduce errors during optimization, such as invalid pointer usage and type confusion during directive insertion.

The **HLS-C debugging** phase is designed to ensure the correctness of the generated HLS-C within ChatHLS workflow. Initially, the generated code is tested by the HLS tool. Upon detection of errors during C simulation and synthesis, we parse the compilation report and pair it with the erroneous code for a fine-tuned LLM ③ specifically tailored for error diagnosis. This model formulates explicit modification instructions with detailed analysis,

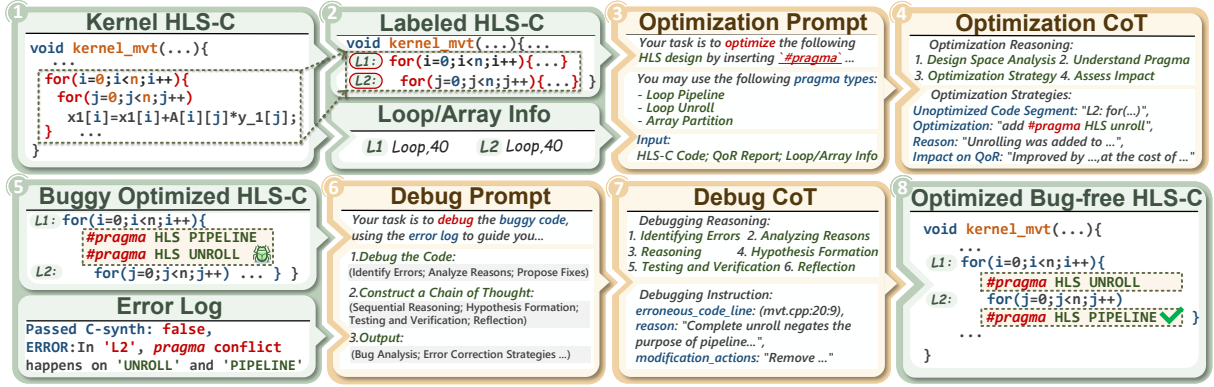


Figure 4: An example of HLS-C optimization and error diagnosis in ChatHLS workflow.

which are then passed to LLM ④. Operating under strict instruction adherence, this agent adopts the instructions to implement debugging.

For errors beyond the training distribution, we forward the parsed error message to LLM Group ⑤ for multifaceted evaluation. Subsequently, LLM ⑥ evaluates the proposed solutions and selects the most appropriate one to repair the code. Furthermore, the errors encountered at this stage are collected via a self-evolving framework. This allows ChatHLS to tackle a broader spectrum of complex HLS errors beyond generic C++ debugging.

### 3.2 HLSTuner

As a core component of the HLS-C generation, we propose HLSTuner to automate HLS directive selection, combination, configuration and insertion. To navigate the expansive design space of HLS-C, HLSTuner enables *QoR-aware reasoning* to align optimization goals with hardware constraints.

**HLSTuner Architecture.** The input pair to HLSTuner includes source HLS-C, design metadata (e.g., array dimensions and loop trip counts), and initial QoR, including latency (cycles) and resource utilization percentages (DSP, LUT, FF), as shown in Figure 4. HLSTuner processes these inputs to select appropriate directives that match specific HLS design structures. It then coarsely estimates their impact on resource consumption, such as: *"apply PIPELINE for deeply nested loops or inner loops with high trip counts significantly increases DSP and LUT resource consumption."* Following in-context guidance, HLSTuner formulates a detailed plan that specifies: (1) the combination of HLS directives with their types and factors, (2) the target code segments for optimization, and (3) the insertion actions. Finally, an insertion agent executes this plan for HLS-C optimization.

When the initial attempt fails to meet the desired performance, HLSTuner activates an iterative refinement incorporating current directives and the resulting QoR. HLSTuner analyzes QoR to scale loop parallelism up or down. For example, if hardware utilization exceeds the budget, HLSTuner halves parallelism and modifies related directives to support memory accesses. Conversely, if significant idle DSP or LUT resources exist, it prioritizes increasing the parallelism of deeply nested loops.

**Training Strategy.** The parallelism between loop execution and memory access primarily determines the HLS design performance. Therefore, our training aims to enable LLMs to understand the semantics of HLS directives such as PIPELINE, UNROLL, and ARRAY\_PARTITION to optimize these structures, including their usage, effects, and interplay. To enable the LLM to capture the causal relationship between directive changes and QoR variations, we move beyond simply mapping the source code to the optimized code by instilling a *QoR-aware reasoning* capability. This reasoning explicitly provides a step-by-step rationales linking directive modifications to synthesized hardware architecture and performance gains. We use optimization chain-of-thought (CoT) generated by a teacher model to supervise this training.

As shown in Figure 3.B, we employ NSGA-II to generate diverse optimized HLS designs and collect the corresponding QoR reports (Ferikoglou et al., 2024). These samples are structured for LLM ⑦ to generate the optimization CoT. This CoT analyzes the QoR variations (changes in latency and resource usage). It identifies data dependencies to justify PIPELINE choices or balance memory bandwidth with ARRAY\_PARTITION and evaluate hardware parallel processing architectures. Finally, we construct a dataset that pairs source HLS-

| Category                  | Error Type                        | Error Message   | Debugging Instruction  |
|---------------------------|-----------------------------------|---|--|
| HLS-C Incompatible Errors | Dynamic Array Allocation (DAA)    | <b>Error:</b> In function A: Undefined function <code>malloc</code> | <b>Cause:</b> Dynamic memory allocation is not synthesizable. $\Rightarrow$ <b>Diagnosis:</b> Replace dynamic allocation <code>malloc()</code> with fixed-size static array <code>A[]</code> .                           |
|                           | Loop Index Out of Bounds (OOB)    | <b>Error:</b> C TB testing failed, stop generating test vectors     | <b>Cause:</b> Out-of-bounds access creates faulty hardware, failing HLS co-simulation. $\Rightarrow$ <b>Diagnosis:</b> Analyze array access patterns and correct loop boundary <code>&lt;=</code> to <code>&lt;</code> . |
|                           | Pointer Access Error (PTR)        | <b>Error:</b> @E Simulation failed: SIGSEGV                         | <b>Cause:</b> Unconstrained pointers are not synthesizable. $\Rightarrow$ <b>Diagnosis:</b> Replace unsafe pointer <code>*p</code> with explicit static array <code>p[]</code> to produce determined hardware.           |
| HLS Directive Errors      | Dataflow-Pipeline Conflict (DPC)  | <b>Error:</b> PIPELINE and DATAFLOW are incompatible                | <b>Cause:</b> Apply conflict directives at same scope. $\Rightarrow$ <b>Diagnosis:</b> Resolve producer-consumer dependency by removing DATAFLOW from logically interdependent loop.                                     |
|                           | Multi-Layer Pipeline (MLP)        | <b>Error:</b> Forced nested loop full UNROLL cause synth time-out   | <b>Cause:</b> PIPELINE on deep nested or large footprint loops cause resource explosion. $\Rightarrow$ <b>Diagnosis:</b> Analyze loop structure and restrict PIPELINE to critical inner loops.                           |
|                           | Array Partition Invalid Dim (AID) | <b>Warning:</b> PARTITION failed: size mismatch or dim too deep     | <b>Cause:</b> PARTITION exceeds declared dimensions. $\Rightarrow$ <b>Diagnosis:</b> Correct dim parameter to match array declaration and intended memory access pattern.  |

Table 1: Examples of *BugRAG* entries and representative HLS-specific error types.

C, inserted directives, and the CoT. This equips LLM ② with *QoR-aware reasoning*, enabling rapid identification of high-performance optimization strategies within specific resource constraints.

### 3.3 HLSFixer

At the core of the HLS-C debugging phase, we develop HLSFixer, a hierarchical code repair framework designed to address syntax incompatibility and directive misuse errors during HLS-C generation and optimization. Within this framework, an analysis agent adopts a *reasoning-to-instruction* method, analyzing the HLS tool feedback to formulate error modification actions. For errors beyond the scope of the single analysis model, HLSFixer supports an LLM-as-a-judge system that augments the instruction to improve debugging accuracy.

**HLSFixer Architecture.** As depicted in Figure 4, debugging begins with HLS simulation and synthesis log parsing, which uses keyword matching to extract error messages. The analysis LLM ③ then examines the error causes and provides debugging instructions, as illustrated in Table 1. After applying targeted modifications to the HLS-C based on this analysis, HLSFixer retests the corrected HLS design against the *golden results* to ensure semantic equivalence with the original design intent.

When LLM ③ fails to correct errors, we implement an LLM-as-a-judge system to refine the debugging instructions. We provide the modified code and error messages to LLM Group ⑤, which generates diverse debugging instructions. These candidates are then compiled and evaluated by the scoring agent ⑥ (functioning as the judge). Based on clarity, logical soundness, alignment with error messages, and the scope of code modification, this agent selects the optimal suggestion to improve the quality of debugging instruction feedback.

**Training Strategy.** To train the analysis agent

(LLM ③) to perform reasoning and generate accurate debug instructions, we decouple debugging into error identification, diagnosis and repair, as shown in Figure 3.C Specifically, we construct test-bench with verified *golden results* to extract key error messages. These messages enable the LLM to pinpoint error locations and propose targeted modifications. Subsequently, we prompt LLM ⑨ to review the correct HLS-C code, pairing buggy code segments with corresponding error messages to construct debugging CoT. The generated CoT analyzes within `<reasoning>` how the specific error types lead to the error message, and reflects on the code after assuming the fix to ensure the modification is reasonable. The `<instruction>` pinpoints the exact error line, explains the cause, and details the precise repair action. Finally, we construct a dataset comprising buggy code, associated error messages, and generated CoT.

We further employ direct preference optimization (DPO) (Rafailov et al., 2024), which learns implicit reward functions from a preference dataset composed of CoT and non-CoT pairs. For rejected non-CoT, we remove error messages when LLM generates the debug CoT. This may confuse the relationship between HLS-specific errors and error messages. This method enables the fine-tuned LLM ③ to rely more closely on parsed error messages during debugging, thereby aligning with expert error analysis preferences.

### 3.4 Verification-Oriented Data Augmentation

Given that LLMs struggle with strict hardware constraints of synthesizable HLS-C and HLS dataset scarcity, we propose the *Verification-Oriented Data Augmentation (VODA)* paradigm. VODA leverages HLS tool feedback to enable the LLM to learn from failures and self-evolve to address more complex errors in the debugging phase. The philosophy of

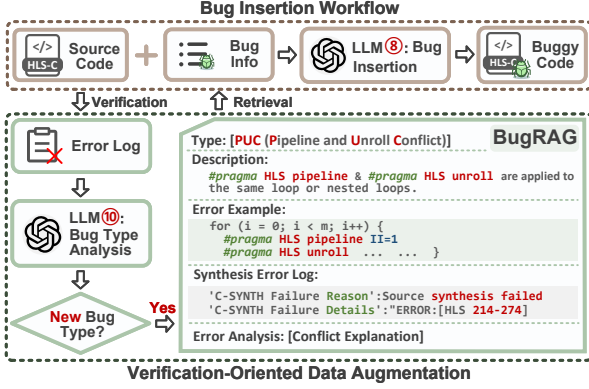


Figure 5: Verification dataset construction workflow.

VODA is to construct and progressively expand a repository of buggy code, featuring various syntax and logic errors. Each entry in the repository is annotated with its error message and detailed root-cause analysis.

**BugRAG.** We design *BugRAG* that dynamically collects and expands the range of HLS-specific error types. Its construction is based on a comprehensive analysis of AMD forum inquiries, prior research (Zhang et al., 2022; Xu et al., 2024b; Wan et al., 2024), and VODA-expanded error cases. We categorize errors encountered during HLS-C generation and optimization phases, as representative cases shown in Table 1. We define *HLS-C Incompatible Errors* as syntax (e.g., dynamic arrays and out-of-bounds array access) that, while valid in standard C, result in incorrect hardware behavior after synthesis. We classify *HLS Directive Errors* as incorrect placement, combination, or configuration of HLS directives, leading to invalid optimizations or synthesis failures. These cases are structured into modular error slices within *BugRAG*, incorporating mnemonic identifiers to improve retrieval accuracy (Wan et al., 2024). We provide a list of collected error types in Appendix B.

**VODA Workflow.** Based on *BugRAG*, VODA operates in two stages. The first stage is the continuous expansion of error cases to populate the error repository, as illustrated in Figure 5. When an HLS design fails verification, an inspection agent (LLM ⑩) examines the erroneous code and error messages parsed from the HLS tool test results. It then generates an error slice containing descriptions, examples, and analysis of a specific HLS error type and queries *BugRAG* to check for existing entries. If unmatched, the inspection agent identifies a new error type and integrates the slice into the error repository with a new mnemonic identifier.

In the second stage, we generate a verification

dataset through a controlled bug injection process. This process is facilitated by an insertion agent (LLM ⑧), which generates buggy code by integrating retrieved error slices from *BugRAG* as context. The agent assesses the contextual applicability of potential bugs, reducing the probability that the LLM forcibly generates trivial results.

## 4 Evaluations

### 4.1 Dataset Construction & LLM Training

For HLSFixer, we constructed 10,878 buggy code covering 33 error types to train LLM ③ through SFT and 3,716 preference pairs for DPO. These buggy samples were injected from 35 base designs from *Kernel* (Pouchet and Yuki, 2016) and *Vitis* (Xilinx Inc., 2024) sources. For HLSTuner, we collected 4,804 samples from 20 Rosetta kernels (Ferikoglou et al., 2024) to fine-tune LLM ②. Appendix D.1 details the train-test similarity analysis. All data were generated by DeepSeek-V3.2.

The training was conducted on  $8 \times$  NVIDIA H800-80G GPUs, based on Qwen-2.5-Coder-14B-Instruct. We used full parameter tuning, AdamW optimizer, 3 epochs with  $1e-5$  learning rate for SFT, and 2 epochs with  $5e-6$  learning rate for DPO. Training details are provided in Appendix D.2.

### 4.2 Benchmarks & Metrics

We evaluated ChatHLS on 108 natural language to HLS-C tasks from (Abi-Karam and Hao, 2025) and our own (e.g., FSMs, counters, rotate operators), each with a validation testbench. For HLSFixer, we constructed 591 test cases from *Kernel*, *Vitis*, and *Manual* (15 manual designs).

For each test case  $c_i$ , we adopt  $pass@k$  to quantify the HLS-C generation and debugging performance. It estimates the probability that at least one correct solution  $c_i^*$  is found among  $k$  generated samples. We set  $n$  trials per  $c_i$  ( $n \geq k$ ). A trial is correct if it passes HLS toolchain verification, including C-Simulation (CSIM), Synthesis (CSYN), and C/RTL Co-simulation (COSIM). Here, CSIM runs the C/C++ testbench against the HLS-C code to validate functional correctness, CSYN compiles HLS-C into RTL and reports timing/resource QoR, and COSIM executes the same testbench against the RTL to verify functional equivalence between the high-level design and generated hardware.

$$pass@k := \mathbb{E}_i \left[ 1 - \frac{\binom{n-c_i^*}{k}}{\binom{n}{k}} \right] \quad (1)$$

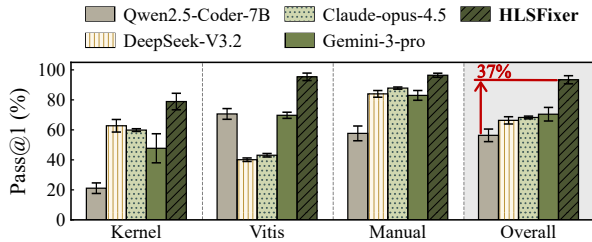


Figure 6: Comparison of debugging capability between HLSFixer and general-purpose LLMs.

| Model                | CSIM         |              | CSYN         |              | COSIM        |              |
|----------------------|--------------|--------------|--------------|--------------|--------------|--------------|
|                      | Pass@1       | Pass@5       | Pass@1       | Pass@5       | Pass@1       | Pass@5       |
| DeepSeek-V3.2        | 47.0%        | 56.8%        | 43.2%        | 55.1%        | 31.5%        | 45.2%        |
| Gemini-3-pro         | 57.9%        | 68.7%        | 56.5%        | 68.2%        | 48.1%        | 60.0%        |
| ChatHLS (w/RAG)      | 59.0%        | 77.7%        | 57.3%        | 75.4%        | 52.3%        | 70.3%        |
| ChatHLS (w/HLSFixer) | <b>82.1%</b> | <b>90.1%</b> | <b>81.2%</b> | <b>90.0%</b> | <b>77.2%</b> | <b>87.6%</b> |

Table 2: Comparison of the generation capability.

We generate  $n = 5$  candidates for each debugging case using  $pass@1$ . For HLS-C generation, we set  $n = 20$  to evaluate the pass rate in stages.

We evaluated HLSTuner on various HLS kernels, including linear algebra, cryptographic algorithms, and neural network accelerators (Abi-Karam et al., 2024). Synthesis was performed with Vitis HLS 2022.1 targeting the Xilinx ZCU106 MPSoC at 100 MHz. The QoR includes execution latency (cycles), utilization of digital signal processors (DSP), flip-flops (FF) and look-up tables (LUT). To quantify HLSTuner performance, we utilized Vitis HLS auto pipeline optimization as a baseline and measured *Speedup*, as defined in Appendix D.4.

In practice, each kernel requires two full verification passes (CSIM/CSYN/COSIM for generation and debugging) and up to five CSYN iterations for optimization. The runtime remains within practical budgets:  $< 3$  minutes for generation,  $< 10$  minutes for debugging, and  $< 30$  minutes for optimization.

### 4.3 HLSFixer Capability Analysis

#### 4.3.1 Comparison with General LLM.

We compare HLSFixer with general-purpose LLMs for error analysis. Figure 6 shows that across comprehensive test cases, HLSFixer achieves a 93.4% pass@1, outperforming Claude-opus-4.5 by 36.8% and Gemini-3-pro by 32.6%. This gap highlights the specialized debugging reasoning of HLSFixer, which enables more accurate identification, analysis, and correction of HLS-specific errors. We provide a comparison of models with different parameter scales in the Appendix F.3.

With HLSFixer, ChatHLS shows a remarkable

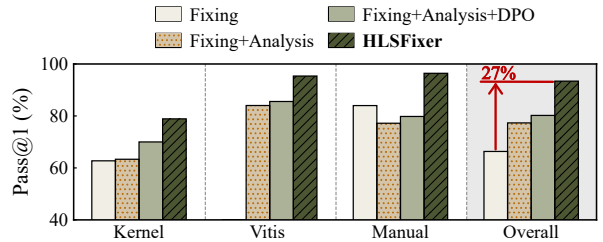


Figure 7: Ablation study of HLSFixer design.

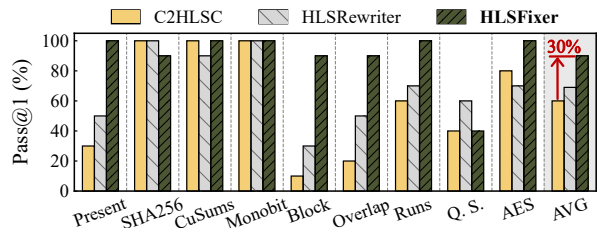


Figure 8: Comparison of HLS-C generation pass rate.

improvement in pass rates for HLS-C generation tasks, as shown in Table 2. By incorporating RAG (using DeepSeek-V3.2 as LLM ①) to provide basic HLS programming specification, combined with a multi-agent debugging approach, ChatHLS ensures the generation robustness, outperforming single Gemini-3-pro by 41.8%. As shown in Figure 8, we compare ChatHLS with C2HLS (Collini et al., 2025) and HLSRewriter (Xu et al., 2025). These results highlight the efficacy of our framework in navigating the complexities of HLS constraints. Appendix E highlights the debugging advantages of HLSFixer in internalizing HLS-specific reasoning beyond retrieval alone.

#### 4.3.2 Ablation Study.

We compared HLSFixer with (1) fixing only with DeepSeek-V3.2, (2) fixing combined with analysis agent using the fine-tuned LLM ③, and (3) fixing and analysis agent augmented with DPO, as shown in Figure 7. After applying fine-tuned analysis agent using *reasoning-to-instruction*, the repair pass rate increased by 16.6% compared to using a single fixing agent. Augmenting LLM ③ with DPO further increased the overall pass rate by 3.7%. For errors unresolved in a single attempt, we performed multifaceted evaluations, which led to an additional 16.5% improvement. Three debugging instructions from different LLMs (GPT-5, Claude-opus-4.5, and Qwen3-8B in this experiment) were evaluated by a long-context LLM (Gemini-3-Pro). These results validate the effectiveness of the fine-tuned analysis model and multi-LLM verification system for HLS-specific error debugging.

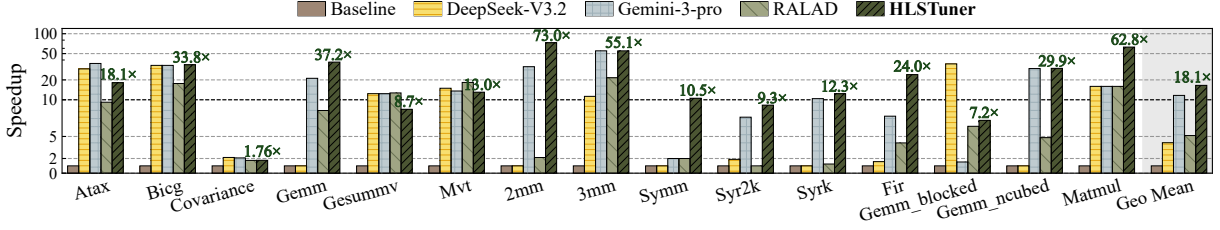


Figure 9: Comparison of optimization capability between Vitis HLS auto optimization (Baseline), general-purpose LLMs, retrieval-augmented method (RALAD) and HLSTuner.

| Accelerators | Metric  | HLSTuner       | RALAD          | Baseline      |
|--------------|---------|----------------|----------------|---------------|
| MobileNet    | Latency | <b>5.88M</b>   | 13.33M         | 13.14M        |
|              | DSP     | 573 (33.2%)    | 729 (42.2%)    | 69 (4.0%)     |
|              | FF      | 36.6K (8.0%)   | 108.7K (23.6%) | 4.5K (1.0%)   |
|              | LUT     | 69.5K (30.1%)  | 111.1K (48.2%) | 12.8K (5.6%)  |
|              | Speedup | <b>2.233x</b>  | 0.986x         | 1.000x        |
| Transformer  | Latency | <b>68.51K</b>  | 88.34K         | 83.30K        |
|              | DSP     | 1215 (70.3%)   | 274 (15.9%)    | 222 (12.8%)   |
|              | FF      | 207.1K (44.9%) | 108.3K (23.5%) | 95.1K (20.6%) |
|              | LUT     | 165.0K (71.6%) | 133.3K (57.8%) | 84.0K (36.4%) |
|              | Speedup | <b>1.216x</b>  | 0.943x         | 1.000x        |

Table 3: Comparison of performance optimization results in hardware accelerators. HLSTuner achieves significant speedup over baseline and RAG-based method.

| Post-Implementation Metric | Min   | Mean  | Max   |
|----------------------------|-------|-------|-------|
| Critical path (ns)         | 2.768 | 5.859 | 8.328 |
| Total on-chip power (W)    | 0.598 | 0.705 | 0.862 |

Table 4: Implementation validity of the best HLSTuner designs across 16 workloads. All designs satisfy the target frequency (critical path < 10 ns) and remain within the acceptable on-chip power budget (< 0.9 W).

## 4.4 HLSTuner Capability Analysis

### 4.4.1 Comparison with General LLMs.

Figure 9 compares HLSTuner with Gemini-3-pro, DeepSeek-V3.2, and RALAD (reproduced using GPT-5.1) (Xu et al., 2024a) in achieving the optimal optimization of HLS kernel performance within 15 trials. HLSTuner achieved a geometric mean speedup of 18.1 $\times$  compared to the baseline, 4.0 $\times$  over DeepSeek-V3.2, 1.5 $\times$  over Gemini-3-pro and 3.3 $\times$  over RALAD, while maintaining resource utilization *under 80%* on target hardware. Current general-purpose LLMs often produce excessive resource utilization or even synthesis failures. In contrast, HLSTuner, driven by a fine-tuned agent with specialized hardware optimization knowledge, achieves optimal speedups by intelligently tuning directive combinations and configurations. Furthermore, HLSTuner deepens its understanding of directive semantics, minimizing errors that may arise during directive insertion. We provide a detailed analysis in Appendix G.

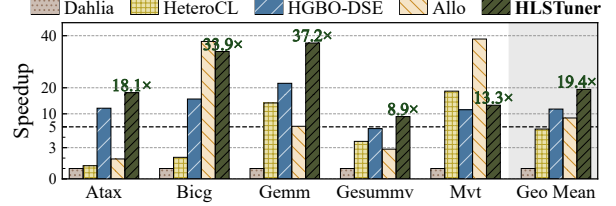


Figure 10: Latency speedup of DSL-based (Dahlia, HeteroCL, Allo), learning-based (HGBO-DSE) methods.

To assess HLSTuner on more complex designs, we applied it to MobileNet and Transformer. Table 3 shows that HLSTuner obtains notable speedups over Vitis HLS baseline. Compared to RALAD, it delivers a 2.3 $\times$  speedup for MobileNet, and a 1.3 $\times$  speedup for Transformer. This highlights the benefit of *QoR-aware reasoning*, which enables LLMs to control optimization trajectories by iteratively sensing the impact of directives on QoR, thereby improving performance while meeting acceptable hardware resource constraints. Furthermore, we observed that even with GPT-5.1, RALAD struggles to comprehend the usage rules and interactions between directives. Out of 15 trials, most resulted in inefficient optimization due to directive misuse or mismatched loop unrolling and array partitioning.

ChatHLS does not invoke algorithm-level C/C++ refactoring during optimization. In HLS, unconstrained structural rewriting may introduce functional deviations that are difficult to attribute from synthesis logs alone. By restricting HLSTuner to directive-level transformations, ChatHLS keeps the kernel semantics unchanged while still covering the standard HLS design space exploration problem of pragma selection, insertion, and factor tuning. Table 4 reports post-implementation results after Place-and-Route. All optimized designs satisfy the target frequency, showing that the reported latency reduction in cycles is physically realizable rather than an artifact of infeasible unrolling. Moreover, ChatHLS generates *dedicated circuits*. This explains the workload-dependent resource variation

in Table 3: MobileNet and Transformer exhibit different arithmetic intensity and memory-access structures, and HLSTuner specializes pragma decisions to each design while keeping resource utilization within the target budget.

#### 4.4.2 Comparison with DSL-based and learning-based methods.

Figure 10 compares HLSTuner against DSL-based (Nigam et al., 2020; Lai et al., 2019; Chen et al., 2024) and learning-based (Kuang et al., 2023) methods on the same HLS kernels. HLSTuner achieved a geometric mean speedup of  $19.4\times$  over Dahlia,  $4.0\times$  over HeteroCL,  $2.3\times$  over Allo and  $1.6\times$  over HGBO-DSE. Among them, DSL-based methods require users to invest considerable time in mastering the usage of primitives for optimization. Meanwhile, heuristic or learning-based DSE methods necessitate numerous iterations (100 searches for HGBO-DSE) to converge on an optimal solution. In contrast, HLSTuner bridges the gap in directive semantics comprehension between existing LLMs and HLS design optimization. This allows it to avoid evaluating ineffective directive combinations and configurations and rapidly identify promising optimization strategies. Moreover, this approach optimizes performance by exploring *directive* parallelism without altering *code* functional semantics, thus avoiding the correctness concerns associated with LLM-based code rewriting.

## 5 Conclusion

This paper proposes ChatHLS, which features a reasoning-augmented error analysis model and an optimization model proficient in HLS directives to automate HLS-C generation and optimization. We strengthen the robustness of the system by continually expanding error cases through VODA. Experimental results indicate that ChatHLS boosts HLS-C debugging pass rate by **32.6%** relative to Gemini-3-pro. It also achieves a **3.3** $\times$  geo mean speedup against RAG-based methods across various HLS designs. These improvements pave the way for a more efficient and reliable hardware design.

## Acknowledgement

This work is supported by the National Natural Science Foundation of China (Grant No.92464301), the National Key Research and Development Program (Grant No.2024YFB4405600), and the Key Research and Development Program of Jiangsu Province (Grant No.BG2024010).

## Limitation

1) As loop optimization constitutes the core performance bottleneck in scientific computing and neural network acceleration, ChatHLS focuses on tuning directives for loop execution parallelism. It lacks support for complex DATAFLOW control directives with producer-consumer logic or for inserting AXI interface directives to implement the HLS-C design as a valid IP for FPGA deployment.

2) While ChatHLS adjusts its optimization strategies during iteration under specific resource utilization constraints, its effectiveness has so far been validated on the Xilinx ZCU106 MPSoC. Broader case studies on FPGAs with different resource constraints would benefit its portability.

## Ethical Considerations

The LLMs fine-tuned and utilized are intended exclusively for the scientific purpose of HLS-C generation, debugging and optimization. We emphasize that LLM-generated HLS-C should undergo standard simulation and verification before deployment to ensure hardware safety. We confirm that both the fine-tuning and test datasets do not contain offensive or proprietary content.

## References

- Stefan Abi-Karam and Cong Hao. 2025. HLS-Eval: A Benchmark and Framework for Evaluating LLMs on High-Level Synthesis Design Tasks. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pages 219–226.
- Stefan Abi-Karam, Rishov Sarkar, Allison Seigler, Sean Lowe, Zhigang Wei, Hanqiu Chen, Nanditha Rao, Lizy John, Aman Arora, and Cong Hao. 2024. HLS-Factory: A Framework Empowering High-Level Synthesis Datasets for Machine Learning and Beyond. In *2024 ACM/IEEE 6th Symposium on Machine Learning for CAD (MLCAD)*, pages 1–9.
- Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proceedings of the ACM on Programming Languages*, 8.
- Yuze Chi, Weikang Qiao, Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. 2022. Democratizing Domain-Specific Computing. *Communications of the ACM*, 66(1):74–85.
- Luca Collini, Siddharth Garg, and Ramesh Karri. 2025. C2HLS-C: Leveraging Large Language Models to Bridge the Software-to-Hardware Design Gap. *ACM Transactions on Design Automation of Electronic Systems*, 30(6).

- Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems*, 15(4).
- Aggelos Ferikoglou, Andreas Kakolyris, Dimosthenis Masouros, Dimitrios Soudris, and Sotirios Xydis. 2024. CollectiveHLS: A Collaborative Approach to High-Level Synthesis Design Optimization. *ACM Transactions on Reconfigurable Technology and Systems*, 18(1).
- Lik Tung Fu, Jie Zhou, Shaokai Ren, Mengli Zhang, Jia Xiong, Hugo Jiang, Nan Guan, Xi Wang, and Jun Yang. 2026. ChatSVA: Bridging SVA Generation for Hardware Verification via Task-Specific LLMs. *arXiv preprint arXiv:2604.02811*.
- Huizhen Kuang, Xianfeng Cao, Jingyuan Li, and Lingli Wang. 2023. Hgbo-dse: Hierarchical gnn and bayesian optimization based hls design space exploration. In *2023 International Conference on Field Programmable Technology (ICFPT)*, pages 106–114.
- Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 242–251.
- Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. 2020. HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 493–505.
- Runkai Li, Jia Xiong, and Xi Wang. 2025a. iDSE: Navigating Design Space Exploration in High-Level Synthesis Using LLMs. *arXiv preprint arXiv:2505.22086*.
- Weikai Li, Ding Wang, Zijian Ding, Atefeh Sohrabizadeh, Zongyue Qin, Jason Cong, and Yizhou Sun. 2025b. Hierarchical mixture of experts: Generalizable learning for high-level synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 18476–18484.
- Costas Mavromatis and George Karypis. 2025. GNN-RAG: Graph neural retrieval for efficient large language model reasoning on knowledge graphs. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 16682–16699. Association for Computational Linguistics.
- Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhi-jing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 393–407.
- Louis-Noel Pouchet and Tomofumi Yuki. 2016. *Poly-Bench/C 4.2*.
- Neha Prakriya, Zijian Ding, Yizhou Sun, and Jason Cong. 2025. LIFT: LLM-Based Pragma Insertion for HLS via GNN Supervised Fine-Tuning. *arXiv preprint arXiv:2504.21187*.
- Yifu Qiu, Varun R. Embar, Yizhe Zhang, Navdeep Jaitly, Shay B Cohen, and Benjamin Han. 2025. Eliciting in-context retrieval and reasoning for long-context large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 3176–3192. Association for Computational Linguistics.
- Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct preference optimization: your language model is secretly a reward model. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*.
- Benjamin Carrion Schafer and Zi Wang. 2020. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639.
- Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Transactions on Design Automation of Electronic Systems*, 27(4).
- Gwok-Waa Wan, SamZaak Wong, Shengchu Su, Chenxu Niu, Ning Wang, Xinlai Wan, Qixiang Chen, Mengnv Xing, Jingyi Zhang, Jianmin Ye, Yubo Wang, Rongchang Song, Tao Ni, Qiang Xu, Nan Guan, Zhe Jiang, Xi Wang, Yong Chen, and Jun Yang. 2026. FIXME: Towards End-to-End Benchmarking of LLM-Aided Design Verification. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(2):1087–1095.
- Lily Jiabin Wan, Yingbing Huang, Yuhong Li, Hanchen Ye, Jinghua Wang, Xiaofan Zhang, and Deming Chen. 2024. Software/hardware co-design for LLM and its application for design verification. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 435–441.
- Jing Wang, Shang Liu, Yao Lu, and Zhiyao Xie. 2025a. HLSDebugger: Identification and Correction of Logic Bugs in HLS Code with LLM Solutions. *2025 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*.
- Ning Wang, Bingkun Yao, Jie Zhou, Yuchen Hu, Xi Wang, Zhe Jiang, and Nan Guan. 2025b. Large language model for verilog generation with code-structure-guided reinforcement learning. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pages 164–170.
- Xi Wang, Gwok-Waa Wan, Sam-Zaak Wong, Layton Zhang, Tianyang Liu, Qi Tian, and Jianmin Ye. 2024.

ChatCPU: An Agile CPU Design and Verification Platform with LLM. In *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC '24*.

Xilinx Inc. 2024. [Vitis-HLS-Introductory-Examples](#).

Changwen Xing, SamZaak Wong, Xinlai Wan, Yanfeng Lu, Mengli Zhang, Zebin Ma, Lei Qi, Zhengxiong Li, Nan Guan, Zhe Jiang, Xi Wang, and Jun Yang. 2026. ChipMind: Retrieval-Augmented Reasoning for Long-Context Circuit Design Specifications. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(2):1337–1345.

Chenwei Xiong, Cheng Liu, Huawei Li, and Xiaowei Li. 2024. HlSPilot: LLM-Based High-Level Synthesis. In *2024 ACM/IEEE International Conference On Computer Aided Design (ICCAD)*, pages 1–9.

Haocheng Xu, Haotian Hu, and Sitao Huang. 2024a. Optimizing High-Level Synthesis Designs with Retrieval-Augmented Large Language Models. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pages 1–5.

Kangwei Xu, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, Ulf Schlichtmann, and Bing Li. 2024b. 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*, pages 1–9.

Kangwei Xu, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, Ulf Schlichtmann, and Bing Li. 2025. HLSRewriter: Efficient Refactoring and Optimization of C/C++ Code with LLMs for High-Level Synthesis. *ACM Transactions on Design Automation of Electronic Systems*.

Bingkun Yao, Ning Wang, Jie Zhou, Xi Wang, Hong Gao, Zhe Jiang, and Nan Guan. 2025a. Location is Key: Leveraging LLM for Functional Bug Localization in Verilog Design. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pages 1–7.

Xufeng Yao, Wenqian Zhao, Qi Sun, Cheng Zhuo, and Bei Yu. 2025b. High-level synthesis directives design optimization via large language model. *ACM Transactions on Design Automation of Electronic Systems*, 30(5).

Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 741–755.

Qian Zhang, Jiyuan Wang, Guoqing Harry Xu, and Miryung Kim. 2022. HeteroGen: transpiling C to heterogeneous HLS code with automated test generation and program repair. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22*, page 1017–1029.

## Appendix

### A Implementation Details of ChatHLS

This section details how ChatHLS retrieves HLS-related context during the generation phase, parses error messages during the debugging phase, and training methods during the optimization phase.

#### A.1 The Generation Stage

The end-to-end HLS design process begins with transforming natural language or algorithms into HLS-C. To equip the LLM with essential domain knowledge, we employ a Retrieval-Augmented Generation (RAG) technique. A specialized knowledge base is constructed from official Vitis HLS documentation (*Vitis High-Level Synthesis User Guide (UG1399)*), which is segmented into chunks with a length of 1000 and an overlap of 200 between adjacent chunks to ensure contextual integrity during retrieval. When the LLM processes a critical code segment during the transformation task, it queries this embedded knowledge base. The retrieved text, detailing specific definitions, constraints, or usage patterns, is then appended to the prompt as context. This process significantly improves the success rate of the initial conversion by grounding the generation in HLS specification.

#### A.2 The Verification and Debugging Stage

Both the initial generation and subsequent optimization phases may introduce HLS-specific errors. These may stem from a designer’s unfamiliarity with HLS constraints or from LLM-driven processes. To this end, we propose a debugging framework that relies on verification using vendor HLS tools. Current HLS tools provide detailed error messages when code issues arise, which can be leveraged to strengthen the verification process.

Initially, we collect golden results of various HLS designs from the open-source community. These results originate from the outputs generated by HLS-C running with fixed inputs that yield correct outcomes. By applying the same inputs to the generated HLS-C and comparing its outputs against the golden results, we can verify the functional correctness of HLS-C. Additionally, beyond providing standard C compilation error messages, the HLS tool generates error feedback for HLS incompatibility errors and directive usage errors during the C-Simulation and Synthesis stages.

The debugging process identifies and addresses issues in HLS-C by first detecting errors from the

parsed error message. However, the raw compilation and verification reports generated by HLS tools are extremely verbose for debugging, often spanning thousands of lines of allocation, scheduling, and binding information. To address this, we implement a parsing mechanism that distills the essential diagnostic information. By employing regular expression matching and keyword detection, our parser identifies sentinel phrases that mark the beginning and end of each stage, as well as critical lines containing `ERROR` and `WARNING` indicators. The result is a concise, structured error message that encapsulates a clear pass/fail status for each stage, a high-level classification of the failure reason (e.g., "Synthesis time-out," "Undeclared identifier," or "Inconsistent simulation result"), and a list of the specific lines detailing the issue.

### A.3 The Optimization Stage

To inform HLSTuner, we guide the model to perform a qualitative analysis rather than a precise quantitative prediction. By focusing on the trend of QoR changes, the LLM learns generalizable design rules. For example, consider a case where the `UNROLL` directive is inserted into a loop. The LLM compares the QoR before and after optimization. It observes that the latency decreases significantly, while the resource utilization (specifically DSP and LUTs) increases. Based on this contrast, the model generates a reasoning step explicitly linking the directive to its hardware impact: *"Applying UNROLL increases parallelism, which reduces total execution time but requires more logic resources to instantiate parallel hardware units."* This approach ensures that HLSTuner learns the causal logic of optimization rather than overfitting to the specific numerical statistics of the training set. Consequently, the model can effectively generalize to unseen code structures by applying these learned hardware design principles to balance trade-offs.

### A.4 Reliability of CoT Generation

To address potential biases in training data, we do not prompt the teacher model (DeepSeek-V3.2) to generate solutions from scratch. Instead, it is limited to explaining existing, verified results. This supervision ensures that CoT generation is based on objective data rather than internal prior knowledge.

**Optimization CoT Generation.** For HLSTuner, we provide the teacher model with source code, initial QoR, optimized code, and optimized QoR. The LLM explains *why* the provided solution leads

to the performance improvements. This ensures that the reasoning is based on real performance metrics rather than LLM assumptions. Our training objective focuses on reasoning about the impact of directive combinations on QoR variations, rather than fitting absolute resource utilization and latency metrics. Hardware platform differences primarily affect the resource budget. ChatHLS avoids overfitting to a single hardware platform by learning QoR-aware reasoning and understanding the causal link between directives and performance. Our approach is constrained by utilization budgets and adapts directives based on QoR feedback.

**Debugging CoT Construction.** Similarly, for HLSFixer, we use verified *golden code* (source code) as a reference. We provide the teacher model with both the buggy code and the correct version. The model compares them to generate the diagnosis and repair steps. This prevents the model from suggesting incorrect fixes.

## B Examples of BugRAG entries

*Verification-Oriented Data Augmentation (VODA)* methodology continuously collects and categorizes common error types encountered during HLS-C generation and optimization. BugRAG contains 33 modular error slices and uses dense retrieval (all-MiniLM-L6-v2 in a Chroma vector store) with  $\text{top-}k = 2$ , injecting only the matched slices instead of the full database. As detailed in Table 5, we have organized these errors into five principal categories that span the typical HLS design workflow: (1) *HLS-C Incompatible Errors*, which involve C++ constructs not synthesizable by HLS tools, such as dynamic memory or unsupported data types. (2) *HLS-C Simulation Errors*, which arise during runtime and include issues like infinite loops or out-of-bounds array access. (3) *HLS-C Compilation Errors*, covering standard C++ syntax and scope issues. (4) *HLS-C Functional Errors*, where the code compiles and runs but produces incorrect results due to logical flaws. (5) *HLS Directive Errors*, related to the incorrect application or syntax of HLS pragmas. From a training perspective, curating this error repository provides structured examples that pair a specific error message and code context with a root cause analysis and a validated solution. This enables the model to learn the direct mapping between tool-specific feedback (e.g., a synthesis warning or a simulation failure) and the underlying coding mistake.

| Category                        | Error Type                         | Error Message   | Debugging Instruction  |
|---------------------------------|------------------------------------|---|--|
| HLS-C<br>Incompatible<br>Errors | Undefined Methods (UDM)            | <b>Error:</b> `X` was not declared in this scope  | <b>Cause:</b> Reference to an undeclared identifier. ⇒ <b>Diagnosis:</b> Ensure its declaration (e.g., via <code>#include</code> or forward declaration) precedes its use.   |
|                                 | Unsupported Data Types (UDT)       | <b>Error:</b> `X` has incomplete type and cannot be defined                                   | <b>Cause:</b> Utilizes non-synthesizable constructs that cannot be mapped to static hardware resources. ⇒ <b>Diagnosis:</b> Replace dynamic constructs with static arrays.   |
|                                 | Type Mismatch Bug (TMB)            | <b>Error:</b> Ambiguous overload for `X` (operand types are ...)                              | <b>Cause:</b> Mixed-type arithmetic between HLS arbitrary-precision and native C/C++ floating-point types. ⇒ <b>Diagnosis:</b> Cast native floating-point types to the HLS type.   |
|                                 | Illegal Keywords (IGK)             | <b>Error:</b> `X` does not name a type or was not declared in scope                           | <b>Cause:</b> The use of non-C/C++ keywords (e.g., <code>def</code> , <code>self</code> ). ⇒ Replace the invalid keywords and syntax with their standard C/C++ counterparts.   |
| HLS-C<br>Simulation<br>Errors   | Faulty Indexing (FIN)              | <b>Error:</b> @E Simulation failed: nonzero return value                                      | <b>Cause:</b> Array access with an index outside its declared bounds. ⇒ <b>Diagnosis:</b> Correct loop boundaries and ensure data-dependent indices are within valid range.  |
|                                 | Top Function Not Found (TFF)       | <b>Error:</b> Undefined reference to ... or a multiple definition of ...                      | <b>Cause:</b> Top function name in settings mismatches the source code. ⇒ <b>Diagnosis:</b> Correct the function name to match.  |
|                                 | Infinite Loop (INF)                | <b>Error:</b> @E Simulation failed: SIGSEGV   | <b>Cause:</b> Symptomatic of a loop with a missing or logically flawed termination condition. ⇒ <b>Diagnosis:</b> Review the loop's exit condition to ensure it is reachable.  |
|                                 | Initialization Missing (INIT)      | <b>Error:</b> @E Simulation failed: nonzero return value                                      | <b>Cause:</b> Referencing a variable that is either undeclared or uninitialized. ⇒ <b>Diagnosis:</b> Ensure every variable is declared within the correct scope before its first use.  |
| HLS-C<br>Compilation<br>Errors  | Misconfigured Loop Unit (MLU)      | <b>Error:</b> @E Simulation failed: nonzero return value                                      | <b>Cause:</b> A logic error within the kernel code, such as incorrect loop termination conditions. ⇒ <b>Diagnosis:</b> Review the algorithm, particularly loop control logic.  |
|                                 | Illegal Comment (ICT)              | <b>Error:</b> Expected ... before `/*` token  | <b>Cause:</b> Malformed comment syntax (e.g., unclosed <code>/*</code> , incorrect <code>/**</code> ). ⇒ <b>Diagnosis:</b> Verify comment syntax and ensure no critical code has been disabled by comments.  |
|                                 | Missing Colons (MCS)               | <b>Error:</b> Expected `;` or `:` before `}` token  | <b>Cause:</b> Missing a semicolon ( <code>;</code> ) for statement termination or a colon ( <code>:</code> ), etc. ⇒ <b>Diagnosis:</b> Examine the line number reported and insert the required symbol.  |
|                                 | Unclosed String (UCS)              | <b>Error:</b> Missing terminating ` ` character or ` ` character                              | <b>Cause:</b> An unclosed string or character literal. ⇒ <b>Diagnosis:</b> Locate the line reported and inspect the string and character literals.   |
|                                 | Undefined Objects (UDO)            | <b>Error:</b> `X` was not declared in this scope  | <b>Cause:</b> A variable is referenced before its declaration, or its declaration is outside the accessible scope. ⇒ <b>Diagnosis:</b> Verify the declaration.   |
|                                 | Use of Undeclared Identifier (UII) | <b>Error:</b> `X` was not declared in this scope  | <b>Cause:</b> A function is referenced before its declaration. ⇒ <b>Diagnosis:</b> Verify the function spelling and ensure its declaration precedes any use.   |
|                                 | Unclosed Parentheses (UCP)         | <b>Error:</b> Expected `)` or `}` before ... token  | <b>Cause:</b> Missing parentheses ( <code>()</code> ) or braces ( <code>{}</code> ). ⇒ <b>Diagnosis:</b> Ensuring code blocks (e.g., for loops, conditionals) are correctly enclosed in braces.  |
|                                 | Illegal Separation (ILS)           | <b>Error:</b> Expected `,` or `;` before ... token  | <b>Cause:</b> Incorrect or missing syntactic elements that disrupt code structure, such as commas. ⇒ <b>Diagnosis:</b> Scrutinize the code for C++ syntax compliance.  |
|                                 | Head File Missing (HFM)            | <b>Error:</b> `X` was not declared in this scope or unknown type `X`                          | <b>Cause:</b> Missing <code>#include</code> directive for the header file that defines the type. ⇒ <b>Diagnosis:</b> Ensure all required headers (esp. for custom/library types) are included.   |
|                                 | HLS-C<br>Functional<br>Errors      | Misused Equal (MUE)   | <b>Error:</b> lvalue required as left operand of assignment  |
| Condition Error (CDE)           |                                    | <b>Error:</b> @E Simulation failed: nonzero return value.                                     | <b>Cause:</b> The logical checks or conditions are incorrect or misaligned with the intended logic. ⇒ <b>Diagnosis:</b> Trace intermediate values to find where the logic fails.   |
| Misaligned Zero Output (ZERO)   |                                    | <b>Error:</b> @E Simulation failed: nonzero return value.                                     | <b>Cause:</b> A variable was accidentally initialized to zero. ⇒ <b>Diagnosis:</b> Focusing on variable initializations and calculations to find the mistake.  |
| Operation Error (OPE)           |                                    | <b>Error:</b> @E Simulation failed: nonzero return value.                                     | <b>Cause:</b> Mistakes in the use of operators, or misuse of functions. ⇒ <b>Diagnosis:</b> Trace intermediate variables based on golden results and identify the logical inconsistency.   |
| Bit Shift Error (SHT)           |                                    | <b>Error:</b> @E Simulation failed: nonzero return value.                                     | <b>Cause:</b> A bit shift amount is too large for the data type, causing data loss or errors. ⇒ <b>Diagnosis:</b> Check the variables involved in the bit shift to find the mistake.   |
| HLS<br>Directive<br>Errors      | Misplaced Loop Label (MLL)         | <b>Error:</b> Stray `#` in program; `pragma` was not declared ...                             | <b>Cause:</b> A C label or pragmas are incorrectly positioned relative to loop structures. ⇒ <b>Diagnosis:</b> Position the pragma on the line directly preceding the loop.  |
|                                 | Pipeline-Unroll Conflict (PUC)     | <b>Error:</b> Pragma conflict happens on UNROLL and PIPELINE                                  | <b>Cause:</b> Applying both <code>PIPELINE</code> and <code>UNROLL</code> to the same loop or nested loops. ⇒ <b>Diagnosis:</b> Inspect the loop structure to find and resolve the conflicting pragmas.  |
|                                 | Array Partition Type (APT)         | <b>Warning:</b> Unable to schedule `load` operation on array `X` due to limited memory ports  | <b>Cause:</b> Using the <code>PARTITION</code> with an inappropriate partition type ( <code>complete</code> , <code>block</code> , or <code>cyclic</code> ). ⇒ <b>Diagnosis:</b> Analyze the memory access patterns to select an appropriate partitioning type that provides sufficient concurrent access ports. |
|                                 | Factor Not Divisible (FND)         | <b>Warning:</b> Unable to schedule `store` operation on array `X` due to limited memory ports | <b>Cause:</b> The specified partition <code>factor</code> is incongruent with the array dimensions or the loop memory access patterns. ⇒ <b>Diagnosis:</b> Verify that the partition <code>factor</code> is a divisor of the corresponding array dimension.  |
|                                 | Unknown Pragma Ignored (UNP)       | <b>Warning:</b> Unknown HLS pragma ignored  | <b>Cause:</b> A pragma contains a syntax error, such as a misspelled directive. ⇒ <b>Diagnosis:</b> Select the correct syntax compatible with vendor HLS tools.  |

Table 5: Examples of *BugRAG* Entries

You are given a piece of HLS code that contains bugs, the compiler error log. Your task is to debug the **buggy code** by analyzing the differences between it and the original code, using the error log to guide you. Generate the **COT debugging strategy** through the buggy code and error log

// CoT guidance

- **Instructions:**
  - **Debug the Code:**
    - **Identify Errors:** List the erroneous code line in the buggy code.
    - **Analyze Reasons:** Explain why each bug occurred, referencing the error log and differences from the original code.
    - **Propose Fixes:** Specify the exact changes needed to fix each bug, including code modifications.
  - **Construct a Chain of Thought (CoT):**
    - **Sequential Reasoning:** Address one error at a time, starting with syntax errors from the compiler.
    - **Hypothesis Formation:** Hypothesize why logical errors occur based on the error log and code comparison.
    - **Testing and Verification:** Validate hypotheses through analysis.
    - **Reflection:** Reflect on the findings after each step and plan the next steps.
- **Output:**
  - **Chain of Thought (CoT):** Write a detailed explanation of your debugging process.
  - **Bug Analysis in JSON Format:** Provide a structured summary as per the format below.

// Input/Output format

- **Inputs:** Buggy Code: {buggy code};  
Vitis HLS Compiler Error Log: {error log}.
- **Expected Output Format:**
  - **Chain of Thought (CoT):** [ Your detailed reasoning here ]
  - **Bug Analysis:**

```

"Erroneous code line": "[code line]",
"Reason": "[Explanation of why the bug occurred]",
"Modification actions":
  "Action": "[Specific modification to be made]",
  "Reason": "[Reason for the modification]"
// Add more entries if necessary
// Add more error analyses if necessary

```

Figure 11: Prompt for debugging analysis agent.

## C Prompt Design

### C.1 Prompt for HLS-C Debugging Analysis

To obtain a precise and structured debugging response from LLM, we designed a comprehensive prompt that meticulously guides its reasoning, as shown in Figure 11. This *reasoning-to-instruction* method enables the analysis LLM to leverage error messages to examine the error code thoroughly. The generated instructions are sequentially structured to mirror expert debugging workflows. First, identify specific error lines by cross-referencing the code with the parsed error message. Second, formulate and validate hypotheses about the root cause of each bug, considering both syntax errors and logical inconsistencies. Finally, propose concrete code modifications.

### C.2 Prompt for HLS Design Optimization

As illustrated in Figure 12, this prompt comprises three steps: (1) define the optimization scope by specifying three target directives, (2) analyze contextual inputs including original code, performance metrics, and loop/array information to identify op-

Your task is to optimize the following HLS operator (kernel) by inserting **#pragma** directives to enhance loops and arrays for lower latency and relatively low resource utilization. You may use the following pragma types: **Loop Pipeline**; **Loop Unroll**; **Array Partition**.

// Input format

- **Inputs:** Provided code: {hls code};  
Current latency and hardware resource utilization: {hls code ppa};  
Loop and array information: {loop array info}.

Please **suggest pragma modifications**, specifying where to insert them and the reasoning behind each.

// Output format

- **Expected Output Format:**
  - **Detailed Reasoning:** [ Your step-by-step analysis here ]
  - **Optimization Strategies:**

```

"Code Segment Before Optimization (Loop/Array)": "[Code line before optimization]",
"Code Segment After Optimization (Loop/Array)": "[Code line after inserting the pragma]",
"Reason": "[Explanation of why this pragma was inserted]",
"Result Analysis":
  "Impact of Inserting This Pragma on PPA": "[Description of the impact on Performance, Power, and Area]",
  "Reason": "[Explanation of why this impact occurred based on the performance analysis]"
// Add more entries if necessary

```

Figure 12: Prompt for HLS design optimization agent.

timization bottlenecks, and (3) generate structured optimization strategies with explicit before-after code comparisons and performance impact assessments. This structured process closely mirrors expert-like HLS optimization, facilitating directive tuning that achieves an optimal balance between latency reduction and resource utilization constraints.

## D Experiment Setting

### D.1 Similarity Analysis

To quantitatively evaluate the dataset similarity for our training and testing settings, HLSFixer and HLSTuner, we employed the Rouge-L metric. This metric yields a score within the  $[0, 1]$  range, where smaller values correspond to lower similarity. This analysis contrasted the internal homogeneity of each SFT training set (serving as a baseline) with the similarity between individual test tasks and their corresponding training sets. The metric quantifies similarity by computing the Longest Common Subsequence (LCS) between the aggregated SFT training set ( $S_{agg}$ ) and an individual test task ( $T_i$ ), with the F-measure ( $\beta = 1$ ) formulated as follows:

$$Rouge - L = \frac{2 \cdot LCS(S_{agg}, T_i)}{length(T_i) + length(S_{agg})} \quad (2)$$

The results validate the efficacy of our data partitioning strategy, as depicted in Figure 13. All *Rouge - L* scores were found to be substantially below 0.15, indicating a significant dissimilarity between the training and test sets. This distinction is crucial for mitigating the risk of data leakage

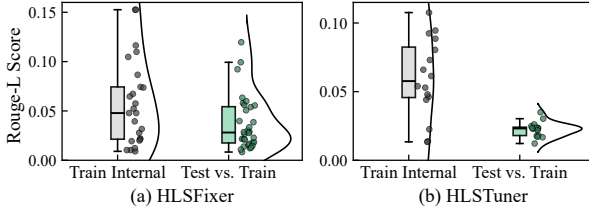


Figure 13: Comparison of Rouge-L similarity.

and ensuring a rigorous evaluation of the model’s generalization. Specifically, the training set was markedly lower than its internal similarity baseline. This trend was even more pronounced in HLSTuner stage, where the mean similarity experienced a sharp decrease. This discrepancy indicates that our training set encompasses a diverse range of HLS-specific errors and provides extensive coverage of HLS-C structures, which enables our model to perform well on unseen data.

## D.2 Detailed Training Settings

The experiments were conducted on a server equipped with  $2\times$  Intel Xeon Platinum 8480+ CPUs and  $8\times$  NVIDIA H800-80G GPUs. The system ran Ubuntu 22.04.2 LTS with CUDA 12.2. The training, based on the Qwen-2.5-Coder-14B-Instruct model, consisted of two main stages: Supervised Fine-Tuning (SFT) followed by Direct Preference Optimization (DPO). We utilized the AdamW optimizer with bfloat16 mixed-precision.

For the SFT stage, we employed full-parameter fine-tuning, leveraging the DeepSpeed ZeRO stage 3 strategy to efficiently manage resources. The model was trained on datasets of 10,878 samples for debugging and 4,804 samples for optimization, with training durations of 72 minutes and 50 minutes, respectively. We trained for 3 epochs with a learning rate of  $1e-5$  under a cosine schedule with 0.1 warmup ratio, using a per-device batch size of 1 and 2 gradient accumulation steps.

Subsequently, for the DPO stage, we initialized from the SFT model and transitioned to use Low-Rank Adaptation (LoRA) with a rank of 8 applied to all target modules. The model was trained for 2 epochs on 3,716 preference dataset took 30 min. We used a sigmoid loss function with  $\beta = 0.1$ . The learning rate was set to  $5e-6$ , again with a cosine scheduler and 0.1 warmup ratio. For this phase, the gradient accumulation steps were increased to 8.

## D.3 HLSFixer Setting

**HLS-C Generation Task.** We used 108 design tasks, covering 85 designs from HLS-Eval (Abi-

| Kernel        | Atax  | Bicg  | Gemm  | Gesummv | Mvt   |
|---------------|-------|-------|-------|---------|-------|
| Lat. (Cycles) | 1702  | 1658  | 15661 | 470     | 1629  |
| DSP (Util.)   | 14.6% | 13.2% | 10.5% | 10.8%   | 13.9% |
| FF (Util.)    | 1.6%  | 1.6%  | 0.4%  | 0.8%    | 1.7%  |
| LUT (Util.)   | 3.6%  | 3.3%  | 2.1%  | 2.4%    | 3.8%  |
| Loop & Array  | 4 / 4 | 3 / 5 | 4 / 3 | 2 / 5   | 4 / 5 |
| # Directives  | 20    | 21    | 17    | 19      | 23    |

Table 6: QoR metrics and design structure of representative baseline computation kernels in linear algebra.

Karam and Hao, 2025) and 23 additional custom designs. This benchmark includes scientific computing (PolyBench), embedded systems (MachSuite), and cryptography (CHStone) to ensure broad representativeness. In Table 2, we compare DeepSeek-V3.2 and Gemini-3-pro. Both models are accessed via API with the temperature set to 0.7. In Figure 8, baseline results for HLSRewriter (Xu et al., 2025) and C2HLSC (Collini et al., 2025) are derived from their reported transformation pass rates (defined as pass@1 over 10 attempts).

**HLS-C Debugging Task.** We constructed 591 debugging test cases derived from 32 correct HLS designs. We used automated error injection techniques to introduce the 34 errors compiled in Table 5. We parsed Xilinx Vitis HLS compilation reports and only retained the cases where the tool explicitly reported the type of injected error. This ensures that the specific buggy samples are unique and unseen during the model pretraining or fine-tuning. In the debugging experiments, the temperature for the analysis model is set to 0.7. For the code modification model, we use DeepSeek-V3.2 and set its temperature to 0.1 to ensure strict instruction adherence. To evaluate the debugging capability of HLSFixer, we keep the code modification model fixed and change the analysis model for comparison. Each benchmark is equipped with a golden testbench to verify the functional correctness. A debugging attempt is considered successful if the corrected code passes CSIM, CSYN, and COSIM.

## D.4 HLSTuner Baseline Settings and Metrics

**Vitis HLS Auto-Optimization Baseline.** The HLS tool applied default pipeline optimizations to the loops in the given design, which causes the baseline test results to generally be better than those without any optimization. The synthesis results of the HLS kernels selected in the optimization task are shown in Table 6, including loop/array information, QoR (latency and resource utilization).

**Optimization Objective.** Given the strict performance and throughput requirements of compute-

| Structure | Optimization Directive | Configuration                            |
|-----------|------------------------|--|
| Loop      | PIPELINE               | "off", "on"                              |
|           | UNROLL                 | integer                                  |
| Array     | ARRAY_PARTITION        | "complete", "block", "cyclic"<br>integer |

Table 7: HLS directive type and configuration.

intensive operations, our evaluation specifically focuses on minimizing latency while maintaining hardware-constrained resource efficiency to validate the effectiveness. By imposing fixed hardware constraints on resource utilization, we explored optimization schemes within a reasonable design space to ensure effective design. To validate the efficiency of HLSTuner, we evaluated the speedup ratios achieved through fewer than five optimization iterations, demonstrating its ability to generate effective solutions under limited search budgets.

**Performance Metrics for HLSTuner.** In the context of HLS design evaluation, the latency is derived from synthesis timing analysis rather than post-implementation routing delays. This metric captures circuit performance determined by logic-level optimization decisions, eliminating variations introduced by layout and wiring during implementation. For an HLS design  $\lambda(\theta)$ ,  $\theta$  represents the inserted HLS directives. Under vendor HLS tool  $\mathcal{H}$ , we analyze the QoR of the explored design using latency  $Lat(\mathcal{H}, \lambda(\theta))$  and resource utilization  $Util(\mathcal{H}, \lambda(\theta))$ . To quantify the impact of our optimization strategies, we measure the *Speedup*:

$$Speedup = \frac{Lat(\mathcal{H}, \lambda(\theta_{baseline}))}{Lat(\mathcal{H}, \lambda(\theta_{optimized}))} \quad (3)$$

$Util_r(\lambda) \leq 80\%, \forall r \in \{DSP, FF, LUT\}$

This metric reflects the performance improvement attributable to the chosen optimization directives  $\theta_{optimized}$  for a given design  $\lambda$ .

**Experimental Setup.** We set the synthesis time limit to 1 hour. Excessively high parallelism configurations cause the scheduling and binding phases of vendor HLS tool to spend more time adjusting the Initiation Interval (II). Exceeding this time limit will likely result in resource over-utilization. We consider this to be a failed optimization attempt. If all 15 attempts fail, or if the optimized latency falls below the baseline, or if resource utilization exceeds limits, we set its penalty speedup in Figure 9. For example, DeepSeek-V3.2 yields no valid optimization across all 15 attempts in *gemm*, *2mm*, *symm*, *syrrk* and *gemm\_ncubed*, we set its speedup to 1 to penalize its geo mean result.

| Model                | CSIM         | CSYN         | COSIM        |
|----------------------|--------------|--------------|--------------|
| DeepSeek-V3.2        | 47.0%        | 43.2%        | 31.5%        |
| DeepSeek-V3.2 + RAG  | 42.6%        | 35.8%        | 32.1%        |
| Gemini-3-pro         | 57.9%        | 56.5%        | 48.1%        |
| Gemini-3-pro + RAG   | 57.7%        | 57.4%        | 50.9%        |
| ChatHLS (w/HLSFixer) | <b>82.1%</b> | <b>81.2%</b> | <b>77.2%</b> |

Table 8: Generation pass@1 under matched RAG baselines. RAG uses retrieved context from the official Vitis HLS documentation.

## D.5 HLS Directives Supported by HLSTuner.

The optimization of loop and array parallelism constitutes a critical bottleneck in HLS design, as these structural elements predominantly determine the performance characteristics of the synthesized hardware implementation. Specifically, for loops, HLSTuner supports the pragma PIPELINE and UNROLL. The PIPELINE pragma allows overlapping execution of loop iterations to improve throughput, while the UNROLL pragma replicates loop bodies to exploit parallelism. For arrays, HLSTuner supports ARRAY\_PARTITION pragma, which divides arrays into smaller memories to enable parallel access and reduce memory bottlenecks. Table 7 summarizes these optimization directive configurations.

## E RAG Baselines

To address the concern that the comparisons against general-purpose LLMs may be unfair without retrieval augmentation, we conduct a controlled study on *matched RAG baselines* for both HLS-C generation and debugging. In all experiments, we use identical task descriptions, identical prompts, and the same retrieved context for all models whenever RAG is enabled. This setting is to isolate the contribution of retrieval from that of our specialized data curation, model adaptation, and hierarchical feedback-driven reasoning.

### E.1 RAG Baselines for HLS-C Generation

For HLS-C generation, we augment general LLMs with retrieved context from the official Vitis HLS documentation. This setting is intended to test whether direct access to domain documentation alone is sufficient to close the performance gap to ChatHLS. Table 8 shows that adding documentation retrieval alone yields only marginal gains, and in some cases even degrades early-stage correctness. For DeepSeek-V3.2, RAG decreases CSIM from 47.0% to 42.6% and CSYN from 43.2% to 35.8%, while only slightly improving COSIM from

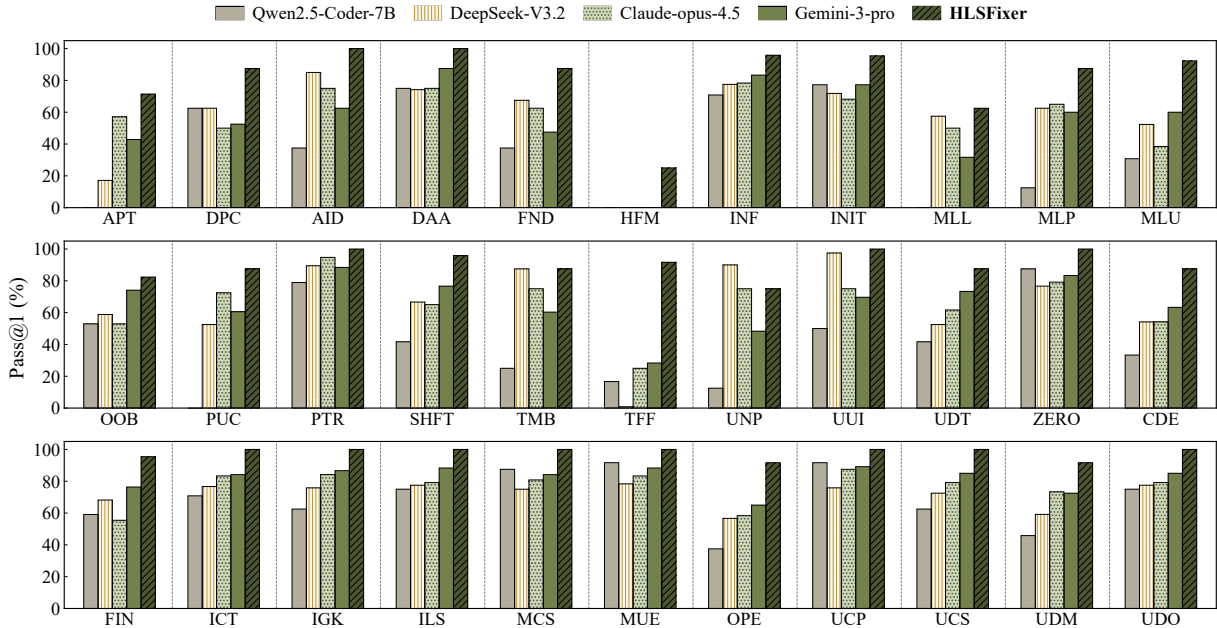


Figure 14: Comparison of code repair pass rates on different HLS-specific errors.

| Model                 | Kernel       | Vitis        | Manual       | Overall      |
|-----------------------|--------------|--------------|--------------|--------------|
| DeepSeek-V3.2         | 62.7%        | 40.1%        | 84.0%        | 66.3%        |
| DeepSeek-V3.2 + RAG-1 | 67.8%        | 74.2%        | 85.4%        | 79.2%        |
| DeepSeek-V3.2 + RAG-2 | 68.9%        | 74.7%        | 84.1%        | 78.9%        |
| Gemini-3-pro          | 47.7%        | 69.7%        | 83.0%        | 70.4%        |
| Gemini-3-pro + RAG-1  | 55.6%        | 80.4%        | 93.9%        | 84.5%        |
| Gemini-3-pro + RAG-2  | 60.0%        | 81.4%        | 91.6%        | 83.6%        |
| HLSFixer              | <b>78.9%</b> | <b>95.4%</b> | <b>96.4%</b> | <b>93.4%</b> |

Table 9: Debugging *pass@1* under RAG baselines. RAG-1 uses Vitis HLS documentation and BugRAG, RAG-2 uses the full training set as the retrieval corpus.

31.5% to 32.1%. For Gemini-3-pro, the effect is also limited. In contrast, ChatHLS substantially outperforms the strongest RAG baseline. These results suggest that access to domain documents is helpful but insufficient. In HLS-C generation, the main bottleneck is not simply missing reference material, but the lack of HLS-specific reasoning patterns for synthesizability, directive semantics, and feedback-aware correction.

## E.2 RAG Baselines for HLS Debugging

We further evaluate matched RAG baselines for debugging under two retrieval settings, as shown in Table 9. RAG-1 represents a practical retrieval setup with compact domain knowledge and structured error slices, whereas RAG-2 tests whether scaling the retrieval corpus with the full training set can replace model adaptation.

Unlike generation, debugging benefits more substantially from retrieval augmentation. DeepSeek-V3.2 improves from 66.3% to 79.2% overall under RAG-1, while Gemini-3-pro improves from 70.4%

to 84.5%. This indicates that explicit retrieval is indeed useful for grounding error analysis, especially when the model can directly map retrieved examples to error logs and local code regions. However, HLSFixer still achieves the best overall pass rate at 93.4%, outperforming the strongest RAG baseline. This result shows that retrieval helps general-purpose LLMs, but it does not replace the need for specialized training. Our fine-tuned analysis model internalizes the hierarchical logic of error diagnosis, which leads to more stable performance across heterogeneous error sources.

Another observation is that more retrieved data does not necessarily yield better results. For both DeepSeek-V3.2 and Gemini-3-pro, RAG-2 performs slightly worse than RAG-1 in overall pass rate. This suggests that simply retrieving from the full training set can introduce weakly matched or noisy contexts, which may dilute the relevance of the evidence and interfere with the model reasoning. Therefore, the main advantage of ChatHLS is not retrieval volume, but the combination of structured error knowledge, targeted analysis supervision, and feedback-grounded reasoning.

## F HLSFixer Supplementary Results

### F.1 HLSFixer Error Diagnosis Reasoning

HLSFixer demonstrates systematic error diagnosis and correction through its hierarchical analysis workflow, exemplified by two representative HLS design errors: dynamic memory allocation violations and dataflow pragma conflicts.

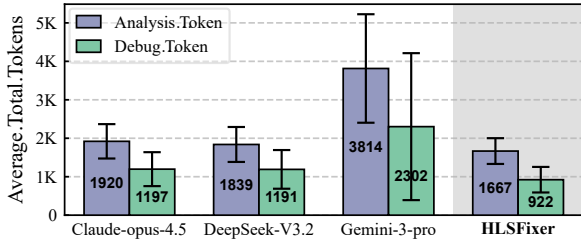


Figure 15: Token consumption for debugging agents.

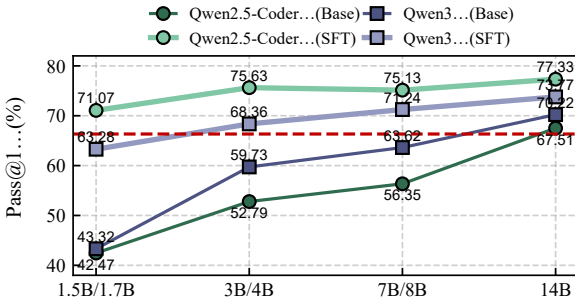


Figure 16: Performance scaling of fine-tuned LLMs on HLS-C debugging tasks. (The red dashed line represents the Deepseek-V3.2 Pass@1 of 66.3%.)

**Dynamic Array Allocation.** A critical HLS design transformation challenge arises from dynamic memory allocation patterns. When synthesizing code containing C++ dynamic allocation, HLSFixer identifies unrecognized memory operators breaking synthesis flow, and unbound memory references contradicting hardware-resource pre-allocation principles. HLSFixer replaces dynamic allocation with static declaration to ensure deterministic memory footprint while preserving the original algorithmic intent.

**Dataflow Pragma Conflict.** Additionally, HLSFixer demonstrates its capability to validate and correct optimization directives by resolving non-canonical dataflow region conflicts. An incorrectly applied `DATAFLOW` to logically interdependent loops resulted in scheduling failures due to unmanaged producer-consumer dependencies. By recognizing the semantic contradiction between the dataflow pragma requirements for independent processes and the sequential dependencies of actual loop structures, HLSFixer resolves the conflicts through targeted pragma removal or pipeline stage reconstruction, ensuring valid task-level parallelism aligned with HLS scheduling semantics.

## F.2 Error Correction Results

Figure 14 compares the pass rates of HLSFixer against baseline models across 33 error types. The results demonstrate that HLSFixer significantly out-

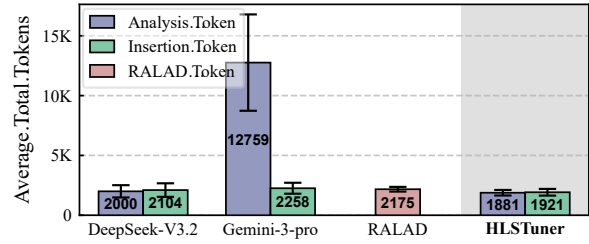


Figure 17: Token consumption for optimization agents.

performs general LLMs, particularly in addressing HLS-specific programming constraints. For compatibility errors, HLSFixer successfully resolves Dynamic Array Allocation (DAA) and Pointer Access Error (PTR). This indicates that HLSFixer learned to avoid standard C/C++ constructs and acquired hardware-oriented coding capabilities during training. Furthermore, HLSFixer exhibits superior performance in resolving HLS directive errors, effectively fixing complex optimization issues like APT and PUC.

In Figure 15, we report the token consumption of the error analysis and code modification models used in the debug task. Gemini-3-pro consumes more tokens because it involves a reasoning process. In most cases, HLSFixer can resolve errors that arise during HLS generation and optimization tasks in a single attempt. Since scenarios requiring multifaceted evaluation to resolve complex problems are infrequent, the overall token cost in the end-to-end workflow is consequently lower.

## F.3 Impact of Model Scale

To evaluate the impact of model scale and domain-specific fine-tuning on the HLS-C debugging task, we benchmarked the debugging performance of the code-centric Qwen2.5-Coder against the general-purpose Qwen3 series. To ensure a fair comparison, we employed identical training data and settings during SFT for all base models.

As illustrated in Figure 16, for both Qwen2.5-Coder and Qwen3, the debugging performance scales consistently with LLM parameters. Fine-tuning with the expertise required for HLS coding constraints has comprehensively enhanced debugging capabilities across all models. Notably, this process enables the base model to approach or surpass the performance of DeepSeek-V3.2. Furthermore, we observe that Qwen2.5-Coder achieves greater performance gains through SFT compared to Qwen3. We attribute this advantage to its extensive pre-training on massive code corpora. This specialized foundation grounds the fine-tuning, en-

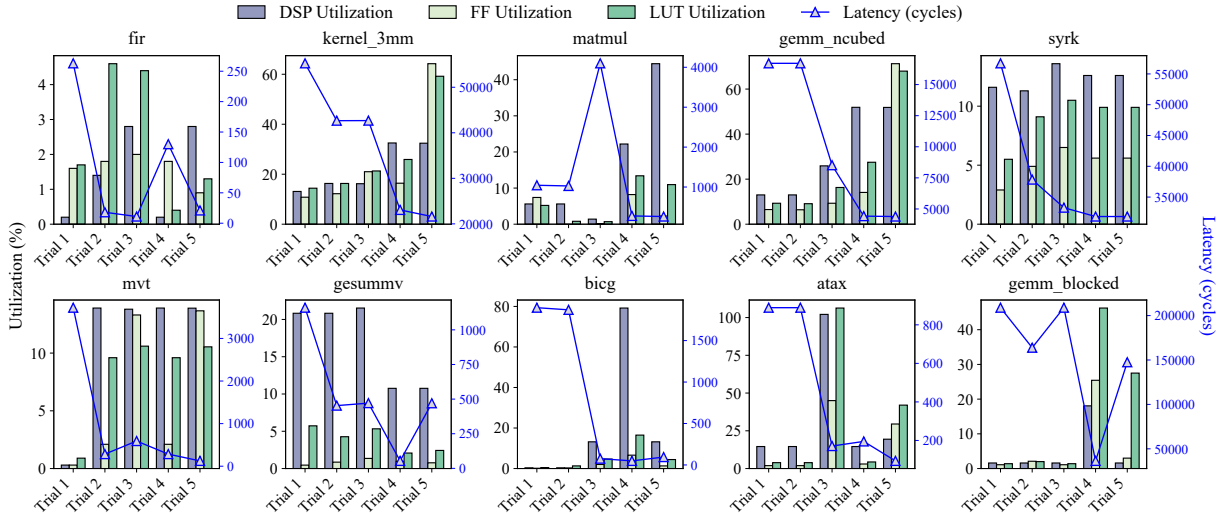


Figure 18: Optimization trajectories HLSTuner within five optimization iterations. By analyzing QoR and dynamically tuning directives, LLM progressively maximizes performance while adhering to resource constraints.

abling the LLM to absorb the subtle nuances specific to HLS debugging tasks more efficiently.

## G HLSTuner Supplementary Results

### G.1 Analysis of Optimization Trajectories

Our analysis reveals a significant performance disparity in DeepSeek-V3.2, which performs well on PolyBench but fails to generalize to other benchmarks. This issue is particularly evident with the compute-intensive kernels *kernel\_2mm* and *kernel\_symm*. For these kernels, DeepSeek-V3.2 consistently applied overly aggressive parallelization strategy, resulting in synthesis failures or excessive resource consumption across all 15 optimization attempts. This suggests that it has trivially memorized specific optimal configurations, likely due to data contamination, rather than genuinely learning a holistic tuning policy.

In contrast, Gemini-3-pro and HLSTuner dynamically adjust directive tuning strategies based on the QoR feedback from each iteration. As illustrated by the optimization trajectories in Figure 18, HLSTuner progressively increases parallelism for performance gains or reduces it when resource utilization exceeds the specified limit. This dynamic navigation of the design space enables them to achieve sustained performance improvements while adhering to resource constraints.

In the experiment, we set DeepSeek-V3.2 as the insertion agent. Following the generated optimization strategy, including detailed directive combinations, configurations, and placements, this agent inserts HLS instructions into the source code. The

| Kernel              |     | Atax  | Bicg  | Gemm  | Gesummv | Mvt   |
|---------------------|-----|-------|-------|-------|---------|-------|
| Dahlia              | DSP | 14.6% | 13.5% | 10.5% | 10.8%   | 13.9% |
|                     | FF  | 1.8%  | 1.8%  | 0.4%  | 0.9%    | 2.1%  |
|                     | LUT | 3.0%  | 2.9%  | 1.7%  | 1.9%    | 5.1%  |
| HeteroCL            | DSP | 1.4%  | 20.1% | 4.5%  | 10.6%   | 13.9% |
|                     | FF  | 0.1%  | 1.3%  | 0.4%  | 0.4%    | 0.8%  |
|                     | LUT | 0.5%  | 10.6% | 1.9%  | 2.1%    | 2.6%  |
| Allo                | DSP | 1.4%  | 13.9% | 4.5%  | 7.2%    | 13.9% |
|                     | FF  | 0.1%  | 1.1%  | 1.4%  | 3.4%    | 0.1%  |
|                     | LUT | 0.7%  | 2.8%  | 3.1%  | 5.1%    | 1.9%  |
| HLSTuner (Our work) | DSP | 19.4% | 79.2% | 49.9% | 10.8%   | 13.9% |
|                     | FF  | 29.5% | 6.6%  | 1.2%  | 1.0%    | 13.7% |
|                     | LUT | 42.0% | 16.5% | 6.9%  | 2.1%    | 10.5% |

Table 10: Hardware cost of optimized HLS design.

average token consumption is shown in Figure 17, where Gemini-3-pro consumed a large number of tokens during reasoning due to the complexity of the optimization task. For RALAD, we combined the retrieved guidance to insert directives into the source code to reproduce the original RAG settings. In contrast, our optimization strategy focused the LLM on analyzing the QoR and specific HLS directives, achieving superior optimization results.

### G.2 Analysis of DSL-based and DSE Methods

To ensure a fair comparison with DSL methods (Dahlia, HeteroCL, Allo) and HGBO-DSE, we selected PolyBench kernels with identical design scales. All evaluations were conducted using consistent HLS tool versions, synthesis time limits, and target FPGA platforms. Table 10 summarizes the resource utilization. For *Mvt* and *Gesummv*, our method achieves DSP usage parity with Dahlia. For *Gemm* and *Bicg*, HLSTuner leverages more resources, demonstrating its ability to strategically implement aggressive parallelization strategies that fully utilize the target hardware budget.

Dahlia focuses primarily on memory banking and loop unrolling factor alignment, but lacks loop pipelining capabilities and has limited optimization options for compute-intensive applications. Allo and HeteroCL both require manual specification of unroll factors for each compute node and rely on expert knowledge to navigate the design space effectively. This reliance on hardware expertise creates significant usability barriers to achieving optimal performance. In contrast, HLSTuner eliminates the need for source code modification, achieving performance goals through automated optimization directive tuning. Although this approach does not achieve the same high efficiency in target hardware resource utilization compared to methods like Allo, it reduces developer effort by allowing brief descriptive sentences to prompt optimization.

The HGBO-DSE method based on Bayesian optimization still struggles to determine and tune critical HLS directives even after 100 iterations. This indicates that without human guidance, such methods tend to converge rapidly to suboptimal solutions in complex HLS designs. In contrast, HLSTuner identifies superior solutions within a short iteration period, as it inherently learns from expert optimization experience. This simplifies the time-consuming design space exploration process, allowing developers to focus on adjusting algorithmic structures to adapt to rapidly evolving application requirements.

## **H Information of Assets**

We strictly adhere to the licenses (MIT, Apache 2.0) and terms of use for all existing artifacts utilized in this work. Our use of these open-source assets is consistent with their intended purposes. For the artifacts we created, including fine-tuned LLMs, we specify that they are intended exclusively for scientific research. This usage is compatible with the original access conditions of the source data.

The Repo. of code is available at: <https://github.com/GEAR-SEU/ChatHLS-ACL-26>