HLSRewriter: Efficient Refactoring and Optimization of C/C++ Code with LLMs for High-Level Synthesis

KANGWEI XU, Chair of Electronic Design Automation, Technical University of Munich, Munich, Germany
GRACE LI ZHANG, Hardware for Artificial Intelligence Group, Technical University of Darmstadt, Darmstadt, Germany
XUNZHAO YIN, College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China
CHENG ZHUO, College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China
ULF SCHLICHTMANN, Chair of Electronic Design Automation, Technical University of Munich, Munich, Germany
BING LI, Research Group of Digital Integrated Systems, University of Siegen, Siegen, Germany

In High-Level Synthesis (HLS), refactoring a standard C/C++ code into its HLS-compatible version (HLS-C) still requires significant human effort. While various program scripts have been introduced to automate this process, the resulting code still contains many HLS-incompatible issues that need to be manually refactored and optimized by developers. Since Large Language Models (LLMs) have the ability to automate code generation, they can also be used for automated code refactoring and optimization in HLS. However, due to the limited training of LLMs, considering hardware and software simultaneously, hallucinations may occur when using LLMs for HLS, leading to synthesis failures. To address these challenges, we introduce **HLSRewriter**, an LLM-aided code refactoring and optimization framework that takes regular C/C++ code as input and automatically generates its corresponding optimized HLS-C code for hardware synthesis with minimal human intervention. To mitigate LLM hallucinations, a step-wise reasoning process is employed to analyze and detect HLS-incompatible errors. Afterwards, a repair library containing reference templates is efficiently created by scanning the HLS tool manual, followed by cooperation with a Retrieval-Augmented Generation (RAG) paradigm to guide the LLMs toward correct refactoring. In addition, a pipeline-aware decomposition strategy is introduced to progressively break down complex loop structures into smaller tasks with a balanced trade-off between latency and area, thereby enabling efficient pipelining and parallel execution. To further improve hardware efficiency, a bit width adjuster module is incorporated into this framework to optimize the precision of floating-point variables. Moreover, LLM-aided HLS optimization strategies are introduced to add/tune hardware directives in HLS-C code, thereby enhancing the performance of the final synthesized hardware. Experimental results demonstrate that the proposed LLM-aided framework can achieve higher refactoring pass rates and superior hardware performance in 24 real-world tasks compared with traditional approaches and the direct application of LLMs for code refactoring and optimization. The codes are open-sourced at this link: https://github.com/code-source1/catapult.

Authors' Contact Information: Kangwei Xu, Chair of Electronic Design Automation, Technical University of Munich, Munich, Germany; e-mail: kangwei.xu@tum.de; Grace Li Zhang, Hardware for Artificial Intelligence Group, Technical University of Darmstadt, Darmstadt, Germany; e-mail: grace.zhang@tu-darmstadt.de; Xunzhao Yin, College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: xzyin1@zju.edu.cn; Cheng Zhuo, College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: czhuo@zju.edu.cn; Ulf Schlichtmann, Chair of Electronic Design Automation, Technical University of Munich, Munich, Germany; e-mail: ulf.schlichtmann@tum.de; Bing Li, Research Group of Digital Integrated Systems, University of Siegen, Siegen, Germany; e-mail: bing.li@uni-siegen.de.

ACM 1557-7309/2025/7-ART
https://doi.org/10.1145/3749986

CCS Concepts: • **Hardware → Electronic design automation**; **High-level and register-transfer level synthesis**.

Additional Key Words and Phrases: Large language models, electronic design automation, high-level synthesis

1 Introduction

High-Level Synthesis (HLS) tools have achieved significant progress in automatically generating Hardware Description Languages (HDL) such as Verilog from general-purpose programming languages such as C/C++. This advancement allows software engineers to contribute to hardware design more actively, thereby lowering the expertise barrier in hardware design [1–7]. However, HLS tools only support a subset of C/C++ code, necessitating significant manual refactoring to transform regular C/C++ code into its HLS-compatible C/C++ version (HLS-C) [8–11]. In other words, some given C/C++ code should be *refactored* before it can be processed by HLS tools to generate the corresponding circuits. For example, the use of dynamic pointers, recursion, and dynamic memory should be manually refactored, as such code is not compatible with the existing HLS tools. In addition, refactoring from sequential execution of C/C++ on CPU to parallel execution of HLS-C on hardware accelerators remains a significant challenge, particularly for complex nested loop structures. In CPU-based execution, nested loops often run sequentially, resulting in higher latency. In contrast, hardware accelerators rely on efficient pipelining and parallel computing to fully exploit their customizable architectures. Moreover, CPU instructions typically use standard data types such as int and float, while in hardware accelerators, the bit width is customized to improve hardware efficiency [15]. An unnecessarily large bit width could lead to increased area overhead, high power consumption, and large clock periods in hardware implementation [16].

Several methods have been proposed for *code refactoring* in HLS, taking C/C++ code as input. [8] statically analyzes the pointers in C/C++ code to reduce global connections. However, the challenge remains when the application accesses memory dynamically with various complex pointers. [10] provides Domain Specific Languages (DSLs) on top of C++ code to support recursion in HLS, but control statements need to be added manually. [11] supports dynamic memory by providing data structure templates. However, only a limited set of data structures is supported, and significant refactoring is still required. [12] presents an HLS backend for generating a customized accelerator using C++ template-based, parameterized types. However, this approach requires the user to manually specify the bit width. [13] reduces bit width through both code profiling and bit analysis. When inputs exceed the typical range, a fallback function is triggered, which is an error-prone process.

Large Language Models (LLMs) have exhibited great potential in automating both software and hardware design, supporting engineers throughout the design flow from initial concepts, algorithms, and architectures to debugging, verification, and optimization [19–33]. Fan et al. [20] build HLS benchmarks with natural language prompts to evaluate LLM performance in hardware design, demonstrating that feedback loops significantly improve the functional correctness of generated HLS code. HLS-Eval [21] provides a comprehensive benchmark suite and evaluation framework tailored for LLM-driven HLS design tasks, establishing clear criteria and thorough infrastructure to advance future research in automated hardware design. Recent studies have also demonstrated that LLMs can be used to correct syntax and logic errors in Verilog and C/C++ code [34–37]. [23] proposes an iterative and conversational-based approach to generate Verilog code using LLMs, and [34] harnesses the capabilities of LLMs as autonomous agents, incorporating human knowledge for reasoning and action planning to automate syntax error fixing for Verilog code. For C/C++ code, [38] introduces iterative approaches that repeatedly query LLMs based on feedback from previous fix attempts to automate code refactoring. [39] proposes an LLM-aided code refactoring approach, where the buggy code is removed, and the LLM directly predicts the correct code given the prefix and suffix context.

To address errors in HLS, several LLM-aided C/C++ code refactoring frameworks for HLS have been proposed [40, 41, 43], leveraging LLM-based iterative refactoring to eliminate HLS-incompatible errors. However, these approaches still face several unresolved challenges and limitations: *1)* Error analysis lacks comprehensive reasoning, leading to understanding the ambiguity of LLMs; *2)* Manual intervention is required to create a repair library; *3)* Refactoring strategies for HLS-incompatible recursion have not been systematically evaluated, resulting in suboptimal designs; *4)* Refactoring techniques tailored for efficient pipelining are often overlooked. Moreover, when targeting ASIC implementation through HLS, it is essential to balance latency and area during the refactoring process. *5)* Precision tuning and bit width adjustments for floating-point computations are not considered.

In this paper, we propose an efficient LLM-aided C/C++ code refactoring and optimization framework for HLS, which is a more advanced framework that integrates step-wise reasoning, efficient library creation, pipeline-aware decomposition, and a bit width adjuster module. The key contributions of this paper are summarized as follows.

- We propose an efficient LLM-aided C/C++ code refactoring and optimization framework for HLS with minimal human effort. The proposed workflow covers runtime profiling and C/C++ code refactoring for hardware generation and performance optimization.
- A step-wise reasoning process is introduced to systematically analyze and detect HLS-incompatible errors. This approach finds potential errors, identifies their locations, and provides explanations step by step, thereby providing a reliable basis for subsequent refactoring.
- To mitigate LLM hallucinations, a repair library containing reference templates is efficiently created by scanning the HLS tool manual, which is then integrated with a Retrieval-Augmented Generation (RAG) paradigm to guide the LLM toward correct refactoring. This approach improves the refactoring pass rate by 30.84% compared to the direct use of the LLM for refactoring.
- To enable efficient pipelining of the refactored code, a pipeline-aware decomposition strategy is introduced to progressively break down complex loop structures into smaller tasks with a balanced trade-off between latency and area, ensuring that the generated HLS-C code achieves higher parallel execution efficiency, thereby reducing hardware latency by an average of 36.58%.
- To achieve customized bit width, a bit width adjuster module is integrated into the framework to analyze and fine-tune the bit width of floating-point variables. This module automatically determines optimal bit width configurations, achieving an average area reduction of 17.78% and a power consumption reduction of 7.52% while maintaining precision.
- To achieve better power, performance, and area (PPA) design of the synthesized circuit, successfully refactored HLS-C code is collected into a potential list for subsequent optimization. Critical code segments with large area, high power, and high latency are identified by HLS tools. LLMs are used to further optimize these code segments to achieve a more efficient PPA design.

The rest of this paper is organized as follows. Section 2 provides the background and motivation of this work. Section 3 explains the details of the proposed method. The experimental results are provided in Section 4. Section 5 concludes the paper.

2 Background and Motivation

Regular C/C++ code are usually designed to be executed by CPUs; HLS tools often cannot compile them correctly because only a subset of the C/C++ code, i.e., HLS-compatible C/C++, can be mapped to hardware design directly.

Traditionally, making regular C/C++ code synthesizable requires developers to possess interdisciplinary expert knowledge in both hardware and software, often involving substantial manual rewriting. It is crucial to eliminate or at least reduce such manual effort to allow designers to focus on logic and performance optimization. Table I summarizes several types of HLS incompatibility types with their corresponding error symptoms and refactoring edits [45]. Key incompatibility types are summarized as follows:

○ *Pointer:* Pointers are strictly forbidden in HLS tools, except for statically analyzable ones. Thus, developers need to manually convert pointer access to array access.

Table 1. Examples of HLS-incompatible types with their corresponding refactorings.

Incompatibility Type	Typical Symptom (# Error / Warning:)	Corresponding Refactoring
Dynamic Pointer	Unsupported feature 'pointers-to-pointers' (CIN-243)	Convert to array access
Dynamic Array	Unsupported feature 'variable length array' (CIN-15)	Specify the static array size
Recursion	Unsupported feature 'recursion' (CIN-15)	Replace with a loop iteration
Operator Constraint	Incrementing a bool value is deprecated (CRD-708)	Convert to integer type
Incomplete Statement	Expected a statement (CRD-127)	Add a complete statement
Unsupported Struct	Unsupported structs create deadlocks. (HIER-10)	Add explicit descriptions
Exception-Handling	Out-of-bounds read/write access (CRD-175)	Add a read/write bound check

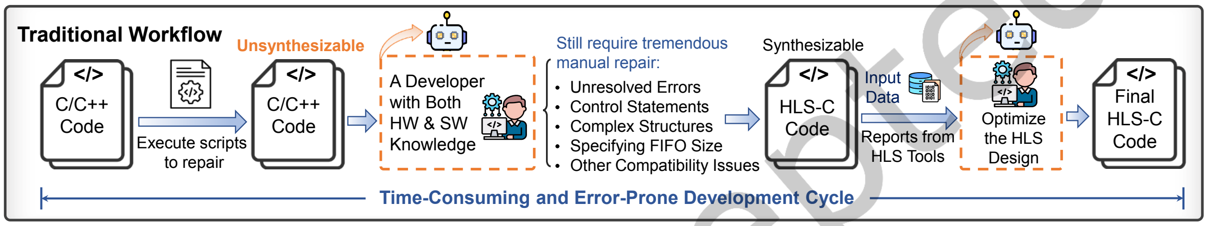


Fig. 1. Traditional workflow for refactoring and optimizing HLS-incompatible C/C++ code.

○ *Dynamic array:* HLS tools do not support dynamic arrays because hardware designs cannot manage data structures with unbounded size. Hence, dynamic array functions such as *malloc()* must be replaced with pre-allocated arrays.

○ *Recursion:* This design style requires dynamic storage in a stack of execution states and is not supported by HLS tools. Manual transformation into loops is required.

○ *Operator Constraint:* HLS tools do not support assignment operators (+=, −=) or increment operators (++, −−) for Boolean operations. An integer operation for these operators needs to be used to replace the Boolean operation.

○ *Incomplete Statement:* HLS tools generate a violation when the case statement does not completely cover the range of values used in conditional expressions. Considering the uncovered values is needed to complete the statement.

○ *Unsupported Structures:* HLS tools do not support virtual functions in C++, necessitating the use of strategy or template methods to achieve polymorphic behavior in hardware designs.

○ *Exception-handling:* Violations are reported when there are out-of-bound array reads/writes and illegal shifts. Additionally, whereas CPU code uses stack popping to handle exceptions, on ASIC, exception-handling modules need to be built to deliver the termination message to other modules.

Fig. 1 illustrates the traditional workflow, where developers rely on scripts to refactor the HLS-incompatible code. However, these scripts can only address basic errors, leaving complex or unforeseen issues unresolved. Therefore, manual refactoring by a developer with deep hardware expertise becomes essential. After the incompatibility errors are fixed, the developers then optimize the HLS-C code and add/tune hardware directives to further improve the design.

Several techniques have been proposed to address the incompatible issues of regular C/C++ code in HLS tools [10–12]. However, these techniques can only partially resolve the aforementioned errors in regular C/C++ code. Developers still need to continually apply various scripts implementing these individual techniques to enhance the hardware synthesizability of C/C++ code, thus leading to error-prone and time-consuming development cycles.

To automate code refactoring, recent studies have reported promising results of applying LLMs [40–43] to refactor and optimize C/C++ code for HLS. C2HLSC [40, 41] uses an LLM to transform C code into HLS-compatible code through iterative prompting and tool feedback. It hierarchically decomposes complex designs, generates unit tests, applies compilation and synthesis feedback loops for refactorings. However, C2HLSC relies on manually crafted prompts and possible human-in-the-loop checks, which limits its generalizability across diverse HLS-incompatible issues. HLSPilot [42] leverages the LLM to automate HLS for hardware acceleration on hybrid CPU-FPGA platforms. However, it is not designed for ASIC-oriented optimizations. It prioritizes maximum throughput, which often results in excessive area and power consumption, overlooking the trade-off between latency and area. Auto-Repair [43] presents an LLM-driven repair framework for HLS that uses a Retrieval-Augmented Generation (RAG) strategy, incorporating an external correction template library, extracted from official HLS tool manuals, to guide the LLM towards correct repair. However, the refactoring process with LLMs for HLS still faces several main challenges: ① The analysis of potential errors in C/C++ code lacks a comprehensive reasoning process, leading to incomplete error analysis, which in turn impacts subsequent refactoring. ② Existing framework relies on a manually created library to provide reference templates for refactoring. ③ The impact of various refactoring strategies, particularly for handling recursion, has not been systematically explored, which in some cases leads to suboptimal designs. ④ More importantly, refactoring the C/C++ code for latency-area balanced pipelining is not considered, which is essential for optimizing the synthesized hardware. ⑤ Optimization of the fractional part of floating-point variables, which is crucial for hardware efficiency, is also overlooked.

Contrary to the previous work, the proposed framework is designed to comprehend refactoring guidelines and combine the relevant design tools with LLMs to refactor C/C++ code for HLS. It takes regular C/C++ code as input and automatically generates corresponding optimized HLS-C designs.

3 LLM-Aided C/C++ code refactoring and optimization for High-Level Synthesis

3.1 Overview of the Proposed Framework

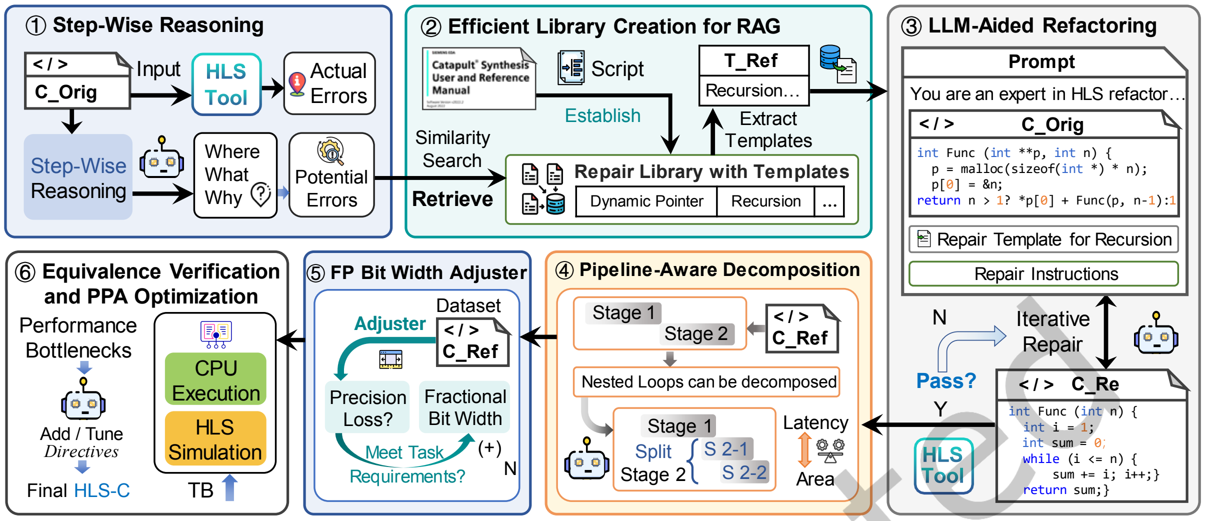
As shown in Fig. 2, the LLM-aided C/C++ code refactoring and optimization framework for HLS consists of six stages:

1) Step-Wise Reasoning: An original C/C++ code C_Orig is compiled using the HLS tool, and actual errors are reported. Since the compiler may not be able to detect all errors in a single compilation, the code, along with common HLS incompatible errors, is subsequently provided to an LLM for analysis and detection of other potential errors step by step.

2) Efficient Library Creation for Retrieval-Augmented Generation (RAG): A repair library containing correction templates tailored for HLS is efficiently created by a script that scans the HLS tool manual. This script employs keyword matching to extract and formulate correction templates for various HLS-incompatible types. The RAG technique is then used to retrieve similar reference templates T_Ref for the subsequent LLM-Aided refactoring.

3) LLM-Aided Iterative Refactoring: The extracted reference templates T_Ref are used to enhance the quality of LLM's prompts while generating a more accurate code C_Re. If the compilation of C_Re fails, the error message will be fed back to the LLM for iterative refactoring. Otherwise, the successfully compiled code is saved as C_Ref.

4) Pipeline-Aware Decomposition: Leveraging the natural language understanding and in-context learning capabilities of the LLM, complex loop structures are decomposed into multiple smaller subtasks by refactoring the C_Ref. This process employs a progressive, balanced decomposition that adapts to latency requirements: each benchmark starts with coarse-grained decomposition to reduce latency at low cost, and finer-grained strategies are only introduced when necessary, achieving a more balanced trade-off between latency and area in ASIC design.



```
int Func (int **p, int n) {
  p = malloc(sizeof(int *) * n);
  p[0] = &n;
return n > 1? *p[0] + Func(p, n-1):1
```

```
int Func (int n) {
  int i = 1;
  int sum = 0;
  while (i <= n) {
     sum += i; i++;}
  return sum;}
```

Fig. 2. The proposed LLM-Aided efficient C/C++ code refactoring and optimization framework for HLS.

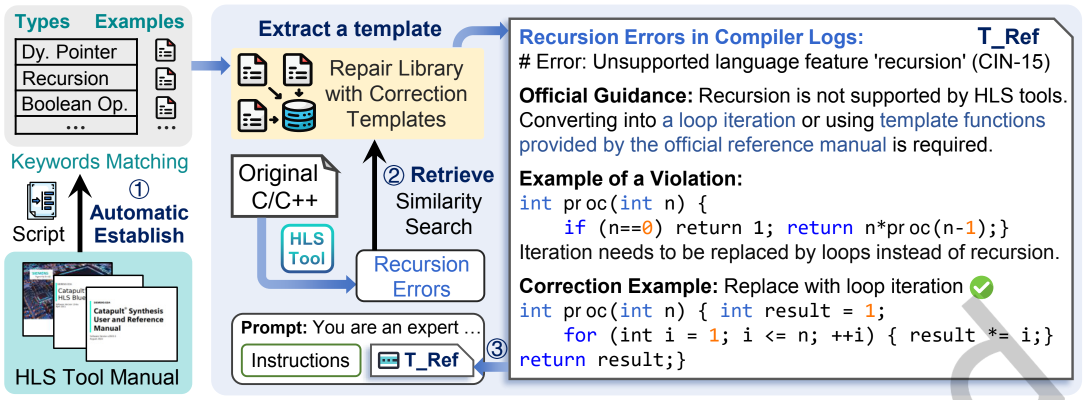
5) Floating-Point Bit Width Adjuster: To optimize both integer and floating-point variables in hardware designs, the existing dataset associated with the C/C++ code is used to determine their optimized bit width. The integer portion is analyzed by extracting the minimum and maximum values to compute the necessary bit width. The fractional portion is then initialized with a minimal number of bits, which is iteratively adjusted until the desired precision is achieved.

6) Equivalence Verification and PPA Optimization: After synthesizing the HLS-C code into a corresponding RTL code, the C-RTL co-simulation is performed for equivalence verification. Based on the original C++ test benchmarks, the HLS tool automatically compares the RTL simulation results with the *C_Orig* results to verify the correctness of the synthesized RTL design. Then, the successfully refactored HLS-C code with large area, high power, and high latency are identified by the HLS tool. The LLM is used to optimize these key segments by adding/tuning pragmas with the guidance of the HLS official manual. This framework integrates efficient interaction between LLM and HLS tools, thereby greatly improving the efficiency of the refactoring and optimization process.

Before executing the six-stage LLM-aided workflow, the framework performs a series of offline preparation steps to ensure consistent interaction between the LLM and the HLS tool, the offline processes of the proposed framework include: ① The creation of an HLS repair library consisting of correction templates; ② Design of general LLM prompts and code examples for both refactoring and optimization tasks; ③ The collection of pragma-based optimization strategies from the official HLS manual. An example of refactoring a regular C++-based breadth-first search (BFS) algorithm is provided in Appendix I to illustrate the proposed framework. Each of the above stages is explained in detail in the following subsections.

3.2 Step-Wise Reasoning

Inspired by the Chain of Thought (CoT) technique, LLM-aided step-wise reasoning framework is introduced to perform progressive analysis on HLS-incompatible C/C++ code, denoted as C_Orig. In practice, the HLS tool first compiles *C_Orig*, and reports a set of errors that actually exist in the code. Since the compiler may not be able to detect all errors in a single compilation, an LLM-aided step-wise reasoning is employed to iteratively analyze the code. As illustrated in Fig. 2, this reasoning process is structured into three key stages: (1) What: The LLM enumerates all potential HLS-incompatible issues by analyzing the *C_Orig* in conjunction with a knowledge



Recursion Errors in Compiler Logs: **T_Ref**

# Error: Unsupported language feature 'recursion' (CIN-15)

Official Guidance: Recursion is not supported by HLS tools. Converting into a loop iteration or using template functions provided by the official reference manual is required.

Example of a Violation:

```
int proc(int n) {
    if (n==0) return 1; return n*proc(n-1);}
```

Iteration needs to be replaced by loops instead of recursion.

Correction Example: Replace with loop iteration ✓

```
int proc(int n) { int result = 1;
    for (int i = 1; i <= n; ++i) { result *= i;}
return result;}
```

Fig. 3. Example of the LLM using RAG to refactor recursion errors.

base of common incompatible types, ensuring that compiler-missed issues are identified. (2) Where: The LLM identifies the precise location of each issue (e.g., function name, line number) by generating structured outputs, such as "dynamic array access at line X of the for-loop." (3) Why: For each identified issue, the model provides a technical rationale (e.g., "dynamic allocation cannot be statically unrolled in hardware"), enabling more targeted and accurate subsequent refactoring.

These three steps form a progressive reasoning chain that systematically analyzes what errors exist, where they occur, and why they arise, ensuring explainable analysis of HLS-incompatible issues, effectively preventing the LLM from skipping reasoning steps or generating unsupported assumptions, thereby improving the correctness of subsequent refactoring. An example of analyzing HLS-incompatible errors is demonstrated in Fig. 14 in Appendix II.

3.3 Efficient Library Creation for LLM-Aided Refactoring

As depicted in Fig. 3, a repair library is efficiently created by scanning the HLS tool manual, alleviating the need for manual effort in its creation. A keyword-based script is employed to scan the manual, match relevant terms, and extract the corresponding text segments, which are then stored in separate files. Developers then review and refine these templates to ensure correctness in formatting and content. Through this process, illustrative examples for each type of HLS incompatibility are systematically gathered, ensuring that an up-to-date repair library is maintained with minimal human effort. In our library, 74 independent T_ref files are extracted from the HLS tool manual, covering seven common categories of HLS-incompatible types (e.g., dynamic pointer, recursion, unsupported structure, etc.). Each T_ref file contains approximately 200 tokens and includes explanations of incompatibility issues and correction examples.

To enhance the refactoring capability of the LLM, we incorporate a Retrieval-Augmented Generation (RAG) mechanism, which introduces external, domain-specific knowledge into the reasoning process via a retriever. By leveraging RAG, the LLM is empowered to generate functionally correct and tool-compliant code refactorings, thereby significantly increasing the refactoring pass rate of the corrected designs. The RAG-enhanced refactoring process for handling HLS-incompatible issues involves the following key stages:

First, an external repair library containing correction templates is created. Each template extracted from the HLS manual consists of four components: (i) the error message reported in the compiler log, extracted from real compiler outputs, (ii) a violation example, (iii) the corresponding explanation and guidance text extracted from the official HLS manual, and (iv) a corrected version of the code. This library serves as a domain-specific knowledge base tailored to the HLS-incompatible issues.

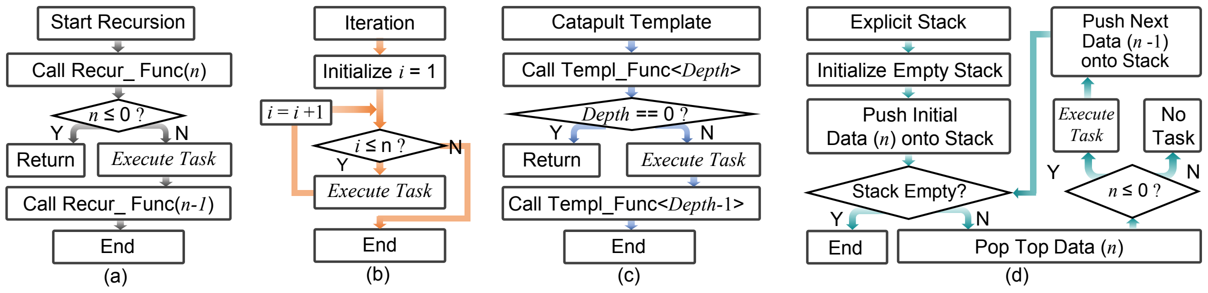


Fig. 4. Flowchart of different schemes for refactoring recursion. (a) Original recursion; (b) Use loop iteration to replace recursion; (c) Use HLS template for compile-time unrolling; (d) Use explicit stacks to simulate recursion.

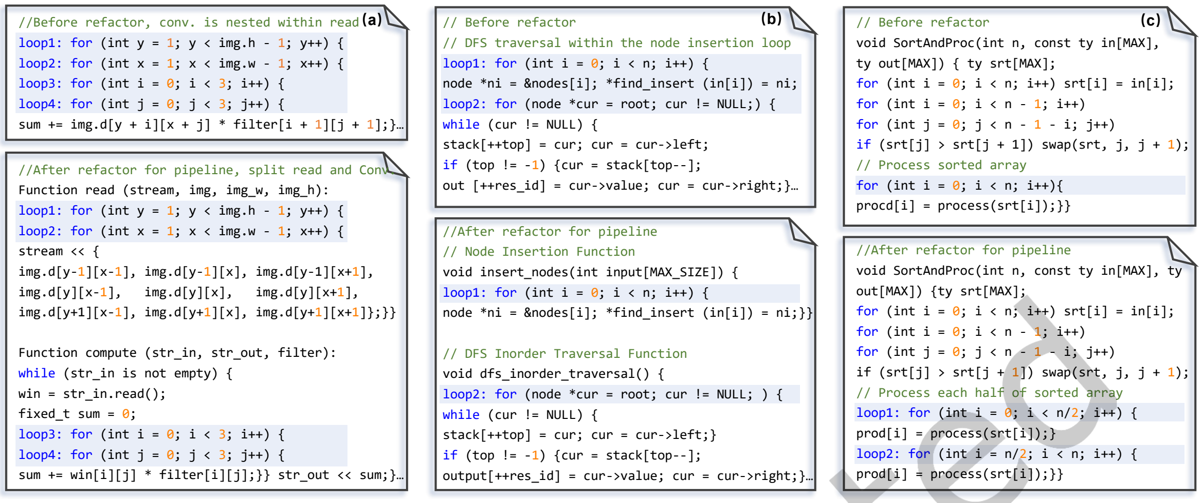
Second, when a compilation error is detected in the original C/C++ code, the framework parses the compiler log and employs a similarity search, such as a Sentence Transformer [46], to retrieve the most relevant correction templates from the repair library.

Third, the retrieved correction template is then integrated into the prompt provided to the LLM. This augmented prompt contains retrieved domain-specific repair examples, enabling the LLM to produce high-quality refactorings that are more likely to pass HLS compilation and simulation.

Since injecting all correction templates at once would result in an excessively long context, we adopt a batching strategy to gradually inject templates, limiting each batch to a maximum of three T_ref files. This approach mitigates issues caused by long-context inputs such as ambiguous reasoning or overlooked guidance.

Fig. 3 illustrates the workflow of employing RAG for refactoring the HLS-incompatible '*recursion*' error. Initially, when the compiler detects a '*recursion*' error, it generates an error log. This log is then used to retrieve the most appropriate correction template in the HLS repair library by the sentence transformer [46]. Once the correction template with the highest similarity is retrieved, it is used as part of the prompts for the LLM. This correction template (T_Ref in Fig. 3) includes HLS guidance on how to refactor recursion, examples of a violation, and corresponding refactored versions. By integrating this external guidance into the input prompts of the LLM, the output responses are improved significantly.

Generally, three primary strategies are often employed to resolve compiler "recursion" errors for HLS: loop iteration-based recursion replacement, HLS template-based unrolling, and explicit stack simulation. Each approach refactors the recursion differently, thereby affecting the performance of the synthesized hardware. In Fig. 4(a), the original recursion function is depicted, where a function repeatedly calls itself until a base case is reached. This recursion paradigm is often incompatible with HLS tools, as dynamic function calls cannot be directly synthesized. In Fig. 4 (b), recursion is replaced by loop iteration. Each iteration performs the same computation as a single recursion call. This strategy is generally straightforward to implement and is well supported by HLS tools, but performance may degrade when handling deep recursion due to increased loop latency. Fig. 4 (c) illustrates the HLS template-based unrolling, where each recursion is unrolled at compile time. However, if the recursion depth is large, hardware resources could become a limiting factor. Fig. 4 (d) shows the explicit stack simulation approach. Rather than relying on recursion calls, a static stack data structure is used to push parameters and pop results, thereby simulating the behavior of recursion. However, this technique introduces additional memory overhead and increases control complexity. Three primary strategies are tried to find the optimal hardware design. Experimental results in Section 4 demonstrate that loop iteration-based recursion replacement achieves comparable performance to template-based unrolling, while explicit stack simulation exhibits lower efficiency due to the additional overhead of stack management. Another example of RAG-based code refactoring for HLS via the LLM, including the prompts and responses, is demonstrated in Fig. 15 of Appendix II.



```
//Before refactor, conv. is nested within read
loop1: for (int y = 1; y < img.h - 1; y++) {
loop2: for (int x = 1; x < img.w - 1; x++) {
loop3: for (int i = 0; i < 3; i++) {
loop4: for (int j = 0; j < 3; j++) {
sum += img.d[y + i][x + j] * filter[i + 1][j + 1];}…
```

```
//After refactor for pipeline, split read and Conv.
Function read (stream, img, img_w, img_h):
loop1: for (int y = 1; y < img.h - 1; y++) {
loop2: for (int x = 1; x < img.w - 1; x++) {
stream << {
img.d[y-1][x-1], img.d[y-1][x], img.d[y-1][x+1],
img.d[y][x-1],   img.d[y][x],   img.d[y][x+1],
img.d[y+1][x-1], img.d[y+1][x], img.d[y+1][x+1]};}}

Function compute (str_in, str_out, filter):
while (str_in is not empty) {
win = str_in.read();
fixed_t sum = 0;
loop3: for (int i = 0; i < 3; i++) {
loop4: for (int j = 0; j < 3; j++) {
sum += win[i][j] * filter[i][j];}} str_out << sum;}…
```

```
// Before refactor
// DFS traversal within the node insertion loop
loop1: for (int i = 0; i < n; i++) {
node *ni = &nodes[i]; *find_insert (in[i]) = ni;
loop2: for (node *cur = root; cur != NULL;) {
while (cur != NULL) {
stack[++top] = cur; cur = cur->left;
if (top != -1) {cur = stack[top--];
out [++res_id] = cur->value; cur = cur->right;}…
```

```
//After refactor for pipeline
// Node Insertion Function
void insert_nodes(int input[MAX_SIZE]) {
loop1: for (int i = 0; i < n; i++) {
node *ni = &nodes[i]; *find_insert (in[i]) = ni;}}

// DFS Inorder Traversal Function
void dfs_inorder_traversal() {
loop2: for (node *cur = root; cur != NULL; ) {
while (cur != NULL) {
stack[++top] = cur; cur = cur->left;}
if (top != -1) {cur = stack[top--];
output[++res_id] = cur->value; cur = cur->right;}…
```

```
// Before refactor
void SortAndProc(int n, const ty in[MAX],
ty out[MAX]) { ty srt[MAX];
for (int i = 0; i < n; i++) srt[i] = in[i];
for (int i = 0; i < n - 1; i++)
for (int j = 0; j < n - 1 - i; j++)
if (srt[j] > srt[j + 1]) swap(srt, j, j + 1);
// Process sorted array
for (int i = 0; i < n; i++){
procd[i] = process(srt[i]);}}
```

```
//After refactor for pipeline
void SortAndProc(int n, const ty in[MAX], ty
out[MAX]) {ty srt[MAX];
for (int i = 0; i < n; i++) srt[i] = in[i];
for (int i = 0; i < n - 1; i++)
for (int j = 0; j < n - 1 - i; j++)
if (srt[j] > srt[j + 1]) swap(srt, j, j + 1);
// Process each half of sorted array
loop1: for (int i = 0; i < n/2; i++) {
prod[i] = process(srt[i]);}
loop2: for (int i = n/2; i < n; i++) {
prod[i] = process(srt[i]);}}
```

Fig. 5. Example of code before and after pipeline-aware decomposition.

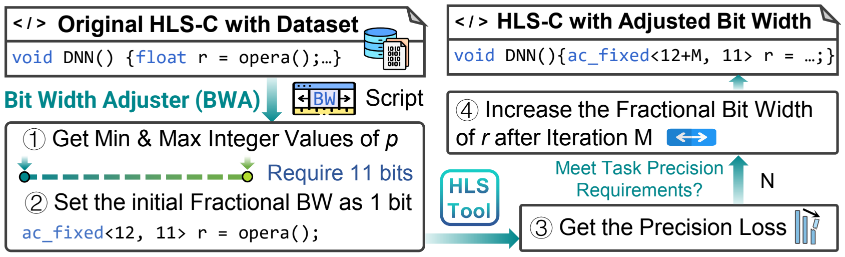
3.4 Pipeline-Aware Decomposition

To realize efficient pipelining and parallel execution, the complex loop structures are decomposed into multiple smaller subtasks. This process is particularly important in ASIC designs implemented via the HLS Tool, where the loop structure significantly influences the latency and area of the synthesized hardware [42, 47, 48]. By leveraging LLMs with in-context learning capabilities, task decomposition is guided by natural language rules and representative examples, ensuring that the refactored code is optimized for parallel execution with minimal area overhead.

A major challenge in loop-based task decomposition lies in selecting an appropriate strategy. If loops are divided in an unbalanced manner, inefficient pipeline stages may be introduced, leading to increased latency and area. To address this issue, a progressive balanced decomposition method is employed, where coarser-grained decomposition is first applied to achieve performance improvement with minimal overhead, and finer-grained decomposition is introduced only when the expected low latency is not achieved. This ensures that latency is effectively reduced while minimizing area overhead. Once decomposition is confirmed, the code is refactored accordingly. The correctness of the refactored code is also verified by the HLS tool to ensure consistency. The testing begins with the current version of the refactored code. If an error is detected, the refactoring process is repeated based on the error prompt. If the error still exists, it rolls back to the previous code version and constrains the LLM to avoid the same error in the subsequent new iteration. If multiple attempts fail to resolve the error, the previous version is retained as the final refactored code. Three strategies are shown in Fig. 5, and detailed examples illustrating these strategies are provided below:

Strategy 1: Multiple Nested Loops as a Subtask: In the edge detection task, four layers of nested loops are present. The first two loops (loop1 and loop2) handle pixel reading, while the remaining two (loop3 and loop4) perform convolution operations. Before refactoring, convolution is performed within the nested loops responsible for reading image pixels. After refactoring, the convolution operations are moved to a separate compute function to enable efficient pipelining. Strategy 1 performs a preliminary coarse-grained decomposition to reduce latency at a small area cost. Strategies 2 and 3 can achieve even lower latency, they typically introduce a large number of additional registers, increasing the area overhead of the target ASIC.

Strategy 2: Single Loop Layer as a Subtask: In the depth-first search (DFS) algorithm, traversal and node insertion operations are typically performed within nested loops. However, executing both tasks within a single loop



</> Original HLS-C with Dataset
void DNN() {float r = opera();…}
Bit Width Adjuster (BWA)
BW
Script
① Get Min & Max Integer Values of p
Require 11 bits
② Set the initial Fractional BW as 1 bit
ac_fixed<12, 11> r = opera();
HLS Tool
③ Get the Precision Loss
Meet Task Precision Requirements?
N
④ Increase the Fractional Bit Width of r after Iteration M
</> HLS-C with Adjusted Bit Width
void DNN(){ac_fixed<12+M, 11> r = …;}

Fig. 6. Workflow of the bit width adjuster (BWA) for determining integer and fractional bit width of floating-point variables.

structure could limit opportunities for pipelining. To mitigate this issue, the loop is refactored so that node insertion and DFS traversal are treated as independent subtasks, each executed as a separate pipeline stage. Compared to Strategy 1, Strategy 2 enables finer-grained pipelining and low latency at the cost of increased area. Strategy 2 is only applied when Strategy 1 cannot achieve sufficient latency reduction.

Strategy 3: Each half of the loop as a Subtask: Prior to refactoring, the sorted-data processing occurs serially in a single loop that traverses the entire array. Rather than relying on direct loop unrolling, which typically requires the total iteration count n to be an exact multiple of the unroll factor and can mishandle or ignore the “tail” iterations. In addition, to avoid dependency violation, one should carefully determine the configuration of the loop unrolling (or split). Our approach explicitly splits the loop into two carefully configured segments, each with its own iteration range, and inserts a tiny residual loop to deal with the tail iteration. In this scheme, the first segment handles indices $[0, \lfloor n/2 \rfloor - 1]$ and the second handles $[\lfloor n/2 \rfloor, n-1]$, guaranteeing that data dependencies within each half are preserved and that no element is dropped. Strategy 3 provides the finest decomposition and is used only when the first two strategies fail to realize the desired low latency.

The pipeline-aware decomposition employs a progressive, balanced method that adapts to latency requirements: each benchmark starts with coarse-grained decomposition (e.g., Strategy 1) to reduce latency at low cost, and finer-grained strategies (e.g., Strategy 2 and Strategy 3) are only introduced when necessary. By introducing only the necessary granularity, our method guides the LLM to enhance pipelining efficiency while minimizing area overhead, achieving a more balanced trade-off between latency and area in ASIC design.

By automating the loop decomposition process with LLMs, the refactored code achieves efficient pipelining, significantly reducing the latency of the generated hardware design while minimizing area overhead. Another example of pipeline-aware decomposition via the LLM, including the prompts and responses, is presented in Fig. 16 in Appendix II.

3.5 Floating-Point Bit Width Adjuster

In traditional HLS development, floating-point numbers are often declared as basic C/C++ data types, which may exceed the bit width actually required and thus lead to low hardware efficiency. Traditional constraint-based approaches often require designers to manually configure parameters in a GUI or develop custom scripts for steps such as data collection, value range estimation, fixed-point bit width computation, and type substitution. These methods frequently involve manual debugging and iteration, which are labor-intensive.

To address this issue, we introduce an LLM-aided floating-point bit width optimization module, as illustrated in Fig. 6, which analyzes variable-associated datasets to optimize bit widths for both integer and fractional portions. This module leverages the generative capabilities of LLMs to generate a bit width optimization script, assisting the developer through the following steps: (a) collect simulation traces and extract extrema of variables; (b) compute optimal integer and fractional bit width based on precision requirements; and (c) systematically replace float types with ac_fixed<M,N> types throughout the code.

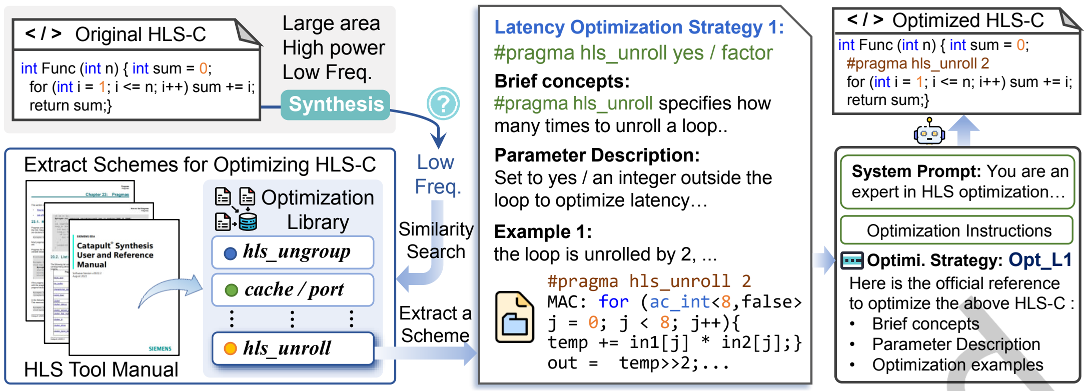

Original HLS-C
int Func (int n) { int sum = 0;
for (int i = 1; i <= n; i++) sum += i;
return sum;}
Large area
High power
Low Freq.
Synthesis
Low Freq.
Extract Schemes for Optimizing HLS-C
Optimization Library
hls_ungroup
cache / port
hls_unroll
HLS Tool Manual
Similarity Search
Extract a Scheme
Latency Optimization Strategy 1:
#pragma hls_unroll yes / factor
Brief concepts:
#pragma hls_unroll specifies how many times to unroll a loop..
Parameter Description:
Set to yes / an integer outside the loop to optimize latency…
Example 1:
the loop is unrolled by 2, ...
#pragma hls_unroll 2
MAC: for (ac_int<8,false> j = 0; j < 8; j++){
temp += in1[j] * in2[j];}
out = temp>>2;...
Optimized HLS-C
int Func (int n) { int sum = 0;
#pragma hls_unroll 2
for (int i = 1; i <= n; i++) sum += i;
return sum;}
System Prompt: You are an expert in HLS optimization…
Optimization Instructions
Optimi. Strategy: Opt_L1
Here is the official reference to optimize the above HLS-C :
Brief concepts
Parameter Description
Optimization examples

Fig. 7. LLM-aided HLS optimization scheme.

Specially, for the integer portion of the floating-point variable, the Bit Width Adjuster (BWA) determines the integer bit width by identifying the maximum absolute value of 'x' in the dataset and then computing:

$$m = \begin{cases} 1, & \text{if } \max|x| < 1, \\ \lceil \log_2(\max|x|) \rceil + 1, & \text{otherwise.} \end{cases} \tag{1}$$

Here, m represents the number of bits required to accommodate both positive and negative values of the integer variable 'x' (i.e., a signed integer range of $[-2^{m-1}, 2^{m-1}-1]$). Once m is determined, the variable 'x' is updated to use `ac_uint` or `ac_int` types, replacing the standard integer data types.

For the fractional portion of the floating-point variable, a minimal fractional bit width (e.g., 1 bit) is initially assigned by using the `ac_fixed` types, after which the bit width adjuster iteratively checks the precision loss and adjusts the fractional bit width as needed. If the desired accuracy is not achieved, the fractional bit width is incremented until precision requirements are satisfied. By eliminating the need for custom script development and manual check iterations, our approach reduces human effort and meets the target precision.

3.6 Equivalence Verification and PPA Optimization

After synthesizing the HLS-C program into RTL, C-RTL co-simulation is performed to verify functional equivalence. Given that Power, Performance, and Area (PPA) are critical metrics in hardware design [44], it is essential to identify performance bottlenecks. To enable efficient HLS optimization, the critical code segments with performance bottlenecks are identified using design analyzers in the HLS tool. Strategies such as adding or modifying pragmas are then applied to these critical code segments to optimize them.

The optimization strategies are identified by an HLS optimization strategy library, which is built by us according to the manual of the HLS tool [45]. As shown in Fig. 7, the optimization strategy information *Opt_L1* consists of a brief concept, parameter descriptions, and optimization examples. The most suitable optimization strategies will be matched via a similarity search, such as a sentence transformer [46], and then integrated into the prompts, utilizing the LLM's in-context learning capabilities to generate an optimized code. For example, loop unrolling (*#pragma hls_unroll*) can be added by the LLM to transform a serial iteration of a loop into a parallel execution, which helps reduce circuit latency. This pragma needs to be set to yes or a value less than or equal to the maximum number of loop iterations. In addition, when memory interfaces become limiting factors, the LLM can optimize memory read and write operations by adding dual-port capabilities or increasing cache mechanisms in the HLS-C code. Another example of the area optimization for the hardware design, including prompts and responses, is demonstrated in Fig. 17 in Appendix II. After optimizing the HLS-C code, circuit optimization strategies such

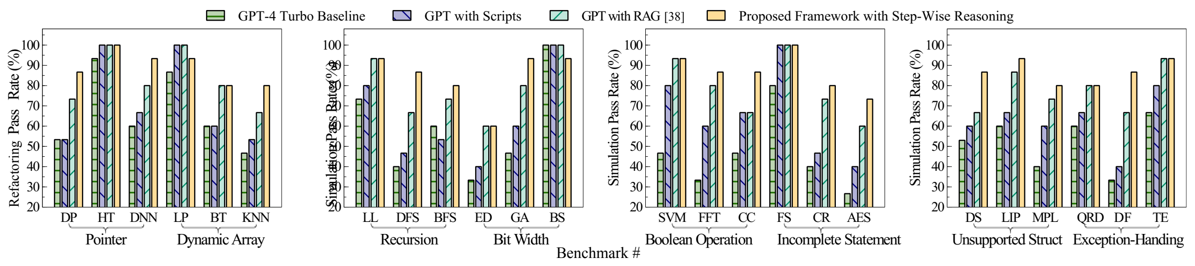


Fig. 8. Comparison of HLS refactoring pass rate of the proposed framework **HLSRewriter** (including step-wise reasoning) with GPT-4 Turbo baseline, GPT with scripts, GPT with RAG on 24 real-world tasks, shown on the x-axes of the figures.

as retiming are further adopted to minimize the clock period and the number of sequential components of the synthesized circuit.

4 Experimental Results

We demonstrate the performance of the proposed LLM-aided framework for HLS across various real-world tasks, evaluating both the refactoring pass rate and the optimized hardware performance. These benchmarks, sourced from related work [10, 11, 13, 14], contain HLS-incompatible errors. Specifically, our evaluation includes the following tasks: Double Pointer (DP), Hash Table (HT), Deep Neural Network (DNN), Linear Programming (LP), Binary Tree (BT), K-Nearest Neighbor (KNN), Linked List (LL), Depth-First Search (DFS), Breadth-First Search (BFS), Edge Detection (ED), Greedy Algorithm (GA), Bubble Sort (BS), Support Vector Machine (SVM), Fourier Transform (FT), Color Correction (CC), Fibonacci Sequence (FS), Cyclic Rotation (CR), AES Encryption (AES), Data Stream (DS), Longest Increasing Path (LIP), Max Points on Line (MPL), QR Decomposition (QRD), Dump Filter (DF), and Turbo Encoder (TE).

To provide a thorough analysis, we conducted a comprehensive evaluation across all 24 benchmarks with respect to the framework's two major design goals: RAG-based repair and PPA (Power, Performance, Area) optimization. This ensures that our reported results reflect the framework's overall effectiveness. For other optimization techniques, we adopted a benchmark-specific evaluation strategy. For example, pipeline-aware decomposition targets complex tasks that can be meaningfully decomposed, often the bottlenecks in achieving low latency, whereas simple tasks do not benefit from such decomposition. Similarly, the floating-point bit-width adjuster is applied only to benchmarks that involve floating-point operations.

During the evaluation, GPT-4 Turbo Model was used as the LLM via OpenAI APIs [49]. All experiments were conducted with the Catapult HLS Tool on an Intel(R) Xeon(R) Silver 4314 2.40 GHz CPU. The logic circuits in implementing HLS-C code were synthesized with Synopsys Design Compiler using the Nangate 45nm open-cell library. To demonstrate the refactoring capability of the proposed framework, each experiment for a particular task was repeated n instances ($n = 15$). For each instance, the LLM was queried five times to refactor the code based on errors reported by the HLS tool. The refactoring pass rate in stage 3 is defined as Pass Rate (%) = m/n, where m is the number of successfully refactored instances and n is the number of all instances. Here, the refactoring is considered successful only if the generated HLS-C code (1) is successfully synthesized by the HLS tool and (2) passes functional simulation. Stages 4-6 target the PPA trade-off, so we evaluated latency, power, and area, which is standard practice for hardware optimization research.

Fig. 8 compares the refactoring pass rate of the proposed HLSRewriter framework with those using GPT-4 Turbo directly, GPT with scripts, and GPT with RAG for code refactoring, respectively. The baseline method using GPT alone achieves a refactoring pass rate of 55.83%. Incorporating refactoring scripts [10–13] improves the pass rate to 65.83%. The RAG-enhanced approach further improves the pass rate to 79.73%, representing a 23.90% improvement over the baseline. In addition, we conducted additional experiments by randomly sampling

10%, 50%, and 80% of the template files (T_ref). The results show that with only 10% of the templates included, the pass rate dropped from 79.73% to 60.54% (a decrease of 19.19%), and performance gradually improved as more templates were incorporated. At the 50% mark, the pass rate recovered to 72.50%. Eventually, when a large portion of templates was included, such as 80%, the pass rate approached 76.38%, which is close to the performance of the full framework. This incremental improvement highlights the value of integrating HLS-specific templates extracted from the HLS manual into the refactoring process. By leveraging the LLM's adaptive learning capability, RAG enables more accurate code refactoring, significantly enhancing the pass rate.

Building upon these methods, our proposed framework with step-wise reasoning achieves the highest refactoring pass rate of 86.67%, outperforming the baseline by 30.84% and delivering an additional 6.94% improvement over the RAG-based method. This improvement is attributed to the step-wise reasoning strategy, which enables the LLM to conduct a more comprehensive error analysis. By guiding the model through a structured reasoning process, it reduces the risk of skipping steps or misjudging errors, thereby establishing a reliable foundation for effective code refactoring.

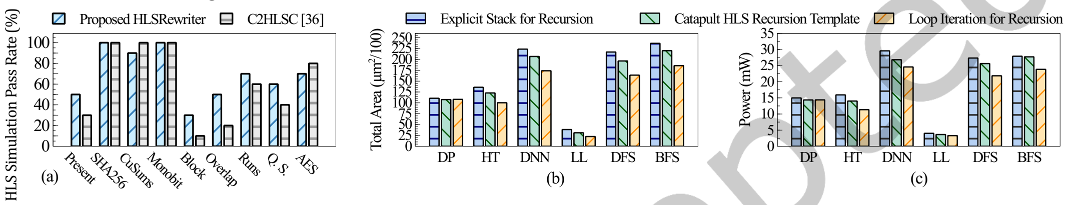


Fig. 9. (a) Comparison with C2HLSC [41] in HLS refactoring pass rate. (b) Area of different schemes for refactoring recursion; (c) Power of different schemes for refactoring recursion

To facilitate a fair and comprehensive comparison with the related work C2HLSC [41], we reused the C2HLSC benchmark suite and replicated its experimental setup, including the same HLS tool and LLM configurations (e.g., temperature, top_p, and number of runs). As shown in Fig. 9 (a), our HLSRewriter achieves higher simulation pass rates in most cases, with an average improvement of 14.81%. These improvements are attributed to its structured step-wise reasoning and the use of correction templates systematically extracted from the official HLS manual.

To evaluate the impact of different refactoring strategies for recursion, three approaches, explicit stack simulation, template-based compile-time unrolling, and loop iteration-based recursion replacement, are compared in terms of area overhead and power consumption in Fig. 9 (b) and (c). Compared with explicit stack simulation, template-based unrolling achieves a 9.3% reduction in area and a 6.84% reduction in power consumption, while loop iteration-based replacement reduces the area by 23.13% and power consumption by 16.92%, further demonstrating its efficiency in HLS-based recursion refactoring.

To illustrate the impact of pipeline-aware decomposition on hardware acceleration, we include ten benchmarks that can be decomposed. We made this choice because pipeline-aware decomposition is targeted at certain complex tasks that can be truly decomposed, while simple tasks do not benefit from such decomposition. As shown in Fig. 10, the baseline latency, area, and power, before applying the decomposition, are normalized to 100%. Our decomposition strategy achieves an average latency reduction of 36.58%, with an average area increase of only 7.38%, demonstrating an effective trade-off between performance and hardware cost. This is because pipeline-aware decomposition follows a progressive and balanced strategy, where coarser-grained decomposition is first applied to achieve performance improvements with minimal area overhead, and finer-grained decomposition is introduced only when necessary. This ensures that latency is effectively reduced while keeping area and power consumption under control. In addition, compared to directly prompting the LLM (e.g., "decompose the task"), our structured strategy reduces decomposition iterations by 2.61× on average, avoiding random refactorings through progressive, example-based guidance.

Moreover, to provide a comprehensive and fair comparison with the related work HLSPilot [42], we evaluated both frameworks on the benchmark under the same experimental settings and LLM configurations. As shown in

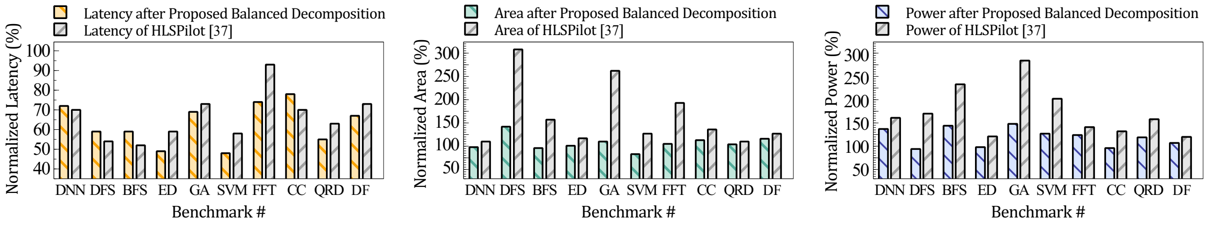


Fig. 10. Comparison with HLSPilot [42] in terms of latency, area, and power.

Fig. 10, under the premise of achieving comparable latency, HLSRewriter achieves significantly better hardware efficiency, with a 53.39% average reduction in area and a 41.46% average reduction in power consumption compared to HLSPilot. These reductions are attributed to its progressive, balanced decomposition and comprehensive optimization strategies tailored for ASIC design.

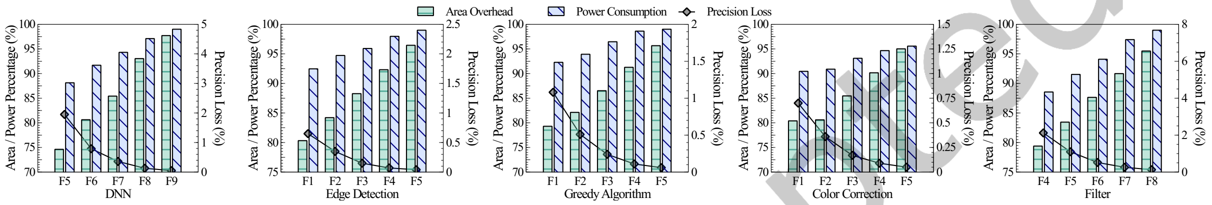


Fig. 11. Ratios of area and power with the corresponding precision loss using the proposed bit width adjuster.

Fig. 11 illustrates the function of the floating-point bit width adjuster module, where the proposed scheme is compared with the default data type in terms of area overhead, power consumption, and precision loss across various floating-point tasks, including floating-point operations. The labels F(n) on the x-axis correspond to configurations using n-bit fractional precision. The results show that this module achieves average reductions of 17.78% in area and 7.52% in power while preserving precision, offering designers a flexible trade-off between hardware efficiency and computational accuracy. When using a lower fractional bit width, the module reduces both area and power consumption at the cost of increased precision loss. As the fractional bit width increases, area and power consumption rise due to the growing complexity of arithmetic units, but this leads to an improvement in computational accuracy.

To demonstrate the effectiveness of the proposed LLM-aided PPA optimization on logic circuits, we compared the area, power, and latency across all benchmarks before and after using such optimization. As shown in Fig. 12, the proposed optimization strategy further achieves an average reduction of 24.99%, 12.69%, and 18.34% in area, power, and latency, respectively, resulting in more efficient designs. These improvements stem from the pragma tuning and circuit optimizations. By applying hardware directives such as loop unrolling, the HLS-generated design can exhibit enhanced parallelism. The retiming technique then redistributes registers to shorten critical paths and minimize area overhead. This co-optimization effectively enhances the performance of the final synthesized circuit.

5 Conclusion

In this paper, we propose an LLM-aided code refactoring and optimization framework for HLS to resolve incompatibility issues in standard C/C++ code while achieving optimized hardware designs. To mitigate LLM hallucinations, step-wise reasoning is employed to analyze HLS-incompatible errors, and a repair library containing reference templates is efficiently created. These templates are then integrated with a Retrieval-Augmented Generation (RAG) paradigm to guide the LLMs towards correct refactoring. Moreover, a pipeline-aware decomposition strategy is introduced to progressively break down complex loop structures into smaller tasks, thereby enabling efficient pipelining and reducing hardware latency. In addition, a bit width adjuster module is incorporated

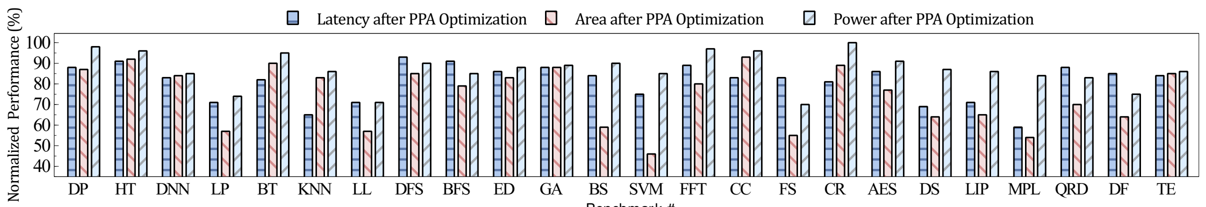


Fig. 12. Ratios of latency, area, and power after applying hardware directive tuning and retiming.

to determine the optimal bit width for both integer and floating-point variables. Experimental results on 24 real-world tasks have demonstrated that the proposed framework significantly improves the refactoring pass rate and hardware performance compared to traditional approaches and the direct use of the LLM.

6 Acknowledgement

This work is funded by the Deutsche Forschungsgemeinschaft (*DFG*, German Research Foundation), under Project-ID 504518248, and by the *NSFC–DFG Sino-German Center for Research Promotion (SGC) Cooperation Project* under Grant No. M-0612. This work is also supported by the TUM International Graduate School of Science and Engineering (*IGSSE*).

Appendix I: An example of code refactoring and optimization framework for HLS

Consider a part of the Breadth-First Search program (in a total of 139 lines in the original code) shown in Fig. 13. This program involves creating a binary tree, initializing the tree structure, adding nodes, and then performing a Breadth-First Search (BFS) to visit and process each node.

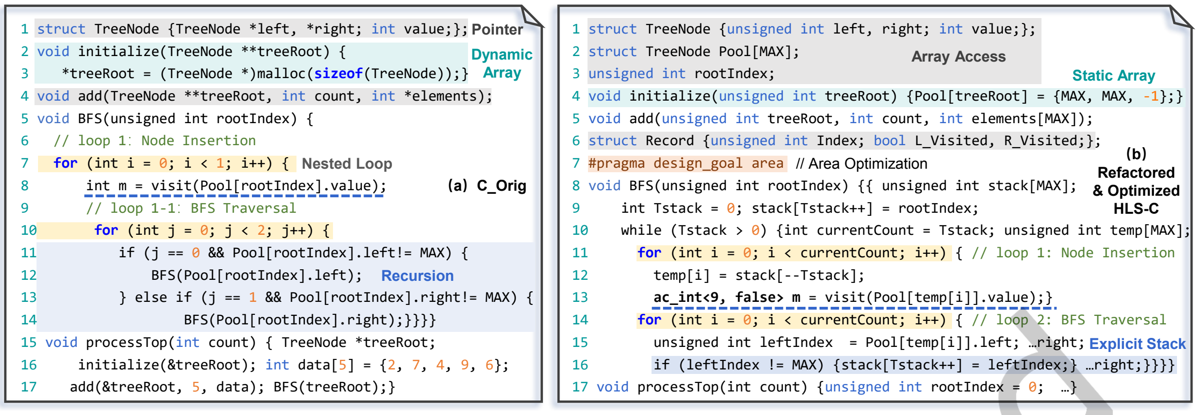
In Section 4, we conducted a comprehensive evaluation of all seven types of errors. Here, to clearly illustrate the refactoring process, we use this example containing three typical incompatible errors, i.e., '*pointer*', '*dynamic array*', and '*recursion*'. The detailed refactoring process is as follows:

1) Step-Wise Reasoning: The original C++ program, *C_Orig*, is compiled using the HLS tool, and the first compilation reports two incompatible errors: '*pointer*' and '*dynamic array*'. Subsequently, both the common HLS-incompatible error types along with the complete program are fed into the LLM to systematically analyze what errors exist, where they occur, and why they arise, during which the LLM identifies an additional incompatible error, '*recursion*'. Traditional scripts are used to refactor the above three errors, but the compilation fails again. The process then moves to the stage of matching error correction templates.

2) and 3) LLM-Aided Refactoring with RAG: In this stage, a repair library is efficiently built according to the HLS tool manual, which is used to match detected errors with their corresponding correction templates. The matched templates, T_Ref, are then used to enhance the quality of prompts provided to the LLM, guiding the LLM to refactor the *C_Orig*, generating a more accurate HLS-C program. As shown in Fig. 13, LLM implements three main refactorings on *C_Orig*:

① Modify Pointer Access to Array Access: In *C_Orig*, tree nodes are managed through pointers directly manipulating memory locations (Line 1 in Fig. 13(a): `TreeNode *left, *right`). The HLS-C version modified by the LLM replaces these pointer-based node accesses with array indices (Line 1-3 in Fig. 13(b)). This modification uses a pre-allocated array (Pool) of TreeNode structures, where each node can be accessed via its index rather than pointers.

② Rewrite Dynamic Array into Static Array: In *C_Orig*, `malloc` is used to create a new TreeNode with dynamic memory allocation (Line 2 and 3 in Fig. 13(a): `malloc(sizeof())`), which is not supported by HLS tools because the synthesized circuit cannot manage data structures with unbounded size. The LLM manages all tree nodes through a predefined array Pool (Line 4 in Fig. 13(b): `Pool[treeRoot] = {MAX, MAX, -1};`), which avoids dynamic allocation.



```
1  struct TreeNode {TreeNode *left, *right; int value;};
2  void initialize(TreeNode **treeRoot) {
3     *treeRoot = (TreeNode *)malloc(sizeof(TreeNode));}
4  void add(TreeNode **treeRoot, int count, int *elements);
5  void BFS(unsigned int rootIndex) {
6    // loop 1: Node Insertion
7    for (int i = 0; i < 1; i++) {
8        int m = visit(Pool[rootIndex].value);
9        // loop 1-1: BFS Traversal
10        for (int j = 0; j < 2; j++) {
11          if (j == 0 && Pool[rootIndex].left!= MAX) {
12              BFS(Pool[rootIndex].left);
13          } else if (j == 1 && Pool[rootIndex].right!= MAX) {
14                  BFS(Pool[rootIndex].right);}}}}
15 void processTop(int count) { TreeNode *treeRoot;
16     initialize(&treeRoot); int data[5] = {2, 7, 4, 9, 6};
17    add(&treeRoot, 5, data); BFS(treeRoot);}
```

```
1  struct TreeNode {unsigned int left, right; int value;};
2  struct TreeNode Pool[MAX];
3  unsigned int rootIndex;
4  void initialize(unsigned int treeRoot) {Pool[treeRoot] = {MAX, MAX, -1};;}
5  void add(unsigned int treeRoot, int count, int elements[MAX]);
6  struct Record {unsigned int Index; bool L_Visited, R_Visited;};
7  #pragma design_goal area  // Area Optimization
8  void BFS(unsigned int rootIndex) {{ unsigned int stack[MAX];
9     int Tstack = 0; stack[Tstack++] = rootIndex;
10    while (Tstack > 0) {int currentCount = Tstack; unsigned int temp[MAX];
11      for (int i = 0; i < currentCount; i++) { // loop 1: Node Insertion
12        temp[i] = stack[--Tstack];
13        ac_int<9, false> m = visit(Pool[temp[i]].value);}
14      for (int i = 0; i < currentCount; i++) { // loop 2: BFS Traversal
15        unsigned int leftIndex  = Pool[temp[i]].left; …right;
16        if (leftIndex != MAX) {stack[Tstack++] = leftIndex;} …right;}}}}
17 void processTop(int count) {unsigned int rootIndex = 0;  …}
```

Fig. 13. Working example (Breadth-First Search) of the proposed LLM-aided refactoring framework. (a) Original C++ program using pointers, dynamic arrays, and recursion; (b) Refactored HLS-C program.

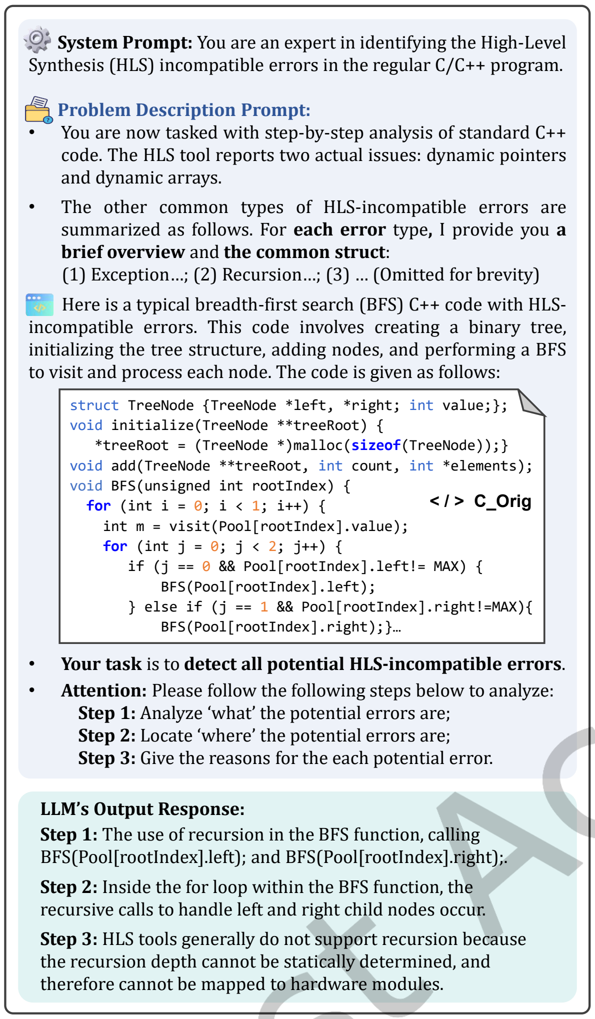
③ Convert Recursion to Iteration: As shown in lines 11-14 of Fig. 13(a), the original function BFS employs recursion to call itself for processing the left and right child nodes of the current node. The LLM replaces the recursion by using an explicit stack data structure, as refactored in lines 16 of Fig. 13(b). This stack simulates the automatic management of the function call stack used in recursion. After these refactorings, the HLS tool compiles this program again. If the compilation still fails, iterative refactoring continues until a successful compilation is achieved.

4) Pipeline-Aware Decomposition: In breadth-first search (BFS) algorithms, node insertion in lines 7 and traversal in line 10 of Fig. 13(a) are often performed in a single loop. However, keeping these two operations within one loop could limit opportunities for pipeline optimizations. By refactoring the BFS, each operation in Fig. 13(b)—node insertion (lines 11) and BFS (lines 14) traversal—is treated as a separate subtask. This separation not only maintains the overall BFS logic but also enables more pipelining.

5) Bit Width Adjuster: To optimize the design of hardware implementations, a bit width adjuster module is integrated into the framework to analyze and fine-tune the bit width of variables. As shown in line 6 of Fig. 13(a) and the corresponding line 15 in Fig. 13(b), if a variable '`m`' is declared as a 32-bit integer and the optimization script finds that its minimum value is 0 and its maximum is 481, then only 9-bit are needed instead of 32-bit. Then, the previously generated script assigns the optimal bit width by replacing `unsigned int m` with '`ac_int<9, false> m`' in *C_Orig*.

6) Equivalence Verification PPA Optimization: After synthesizing the C++ code to obtain the corresponding hardware RTL model, C-RTL co-simulation is then performed to verify the correctness of the synthesized RTL design. Then, a large area is identified by the HLS tool. The proposed LLM-aided optimization strategy is then applied to reduce the area of design by adding the pragma such as '`#pragma design_goal area`' at the top of the function BFS. Retiming is further used to minimize the clock period and the number of sequential components in implementing the synthesized circuit to achieve a more efficient PPA design.

Appendix II: Examples of prompts and output responses of the LLM at different stages



System Prompt: You are an expert in identifying the High-Level Synthesis (HLS) incompatible errors in the regular C/C++ program.

Problem Description Prompt:

- You are now tasked with step-by-step analysis of standard C++ code. The HLS tool reports two actual issues: dynamic pointers and dynamic arrays.
- The other common types of HLS-incompatible errors are summarized as follows. For **each error** type, I provide you **a brief overview** and **the common struct**:
  (1) Exception...; (2) Recursion...; (3) ... (Omitted for brevity)

Here is a typical breadth-first search (BFS) C++ code with HLS-incompatible errors. This code involves creating a binary tree, initializing the tree structure, adding nodes, and performing a BFS to visit and process each node. The code is given as follows:

```
struct TreeNode {TreeNode *left, *right; int value;};
void initialize(TreeNode **treeRoot) {
   *treeRoot = (TreeNode *)malloc(sizeof(TreeNode));}
void add(TreeNode **treeRoot, int count, int *elements);
void BFS(unsigned int rootIndex) {                    </> C_Orig
  for (int i = 0; i < 1; i++) {
    int m = visit(Pool[rootIndex].value);
    for (int j = 0; j < 2; j++) {
       if (j == 0 && Pool[rootIndex].left!= MAX) {
           BFS(Pool[rootIndex].left);
       } else if (j == 1 && Pool[rootIndex].right!=MAX){
           BFS(Pool[rootIndex].right);}...
```

- **Your task** is to **detect all potential HLS-incompatible errors**.
- **Attention:** Please follow the following steps below to analyze:
  Step 1: Analyze 'what' the potential errors are;
  Step 2: Locate 'where' the potential errors are;
  Step 3: Give the reasons for the each potential error.

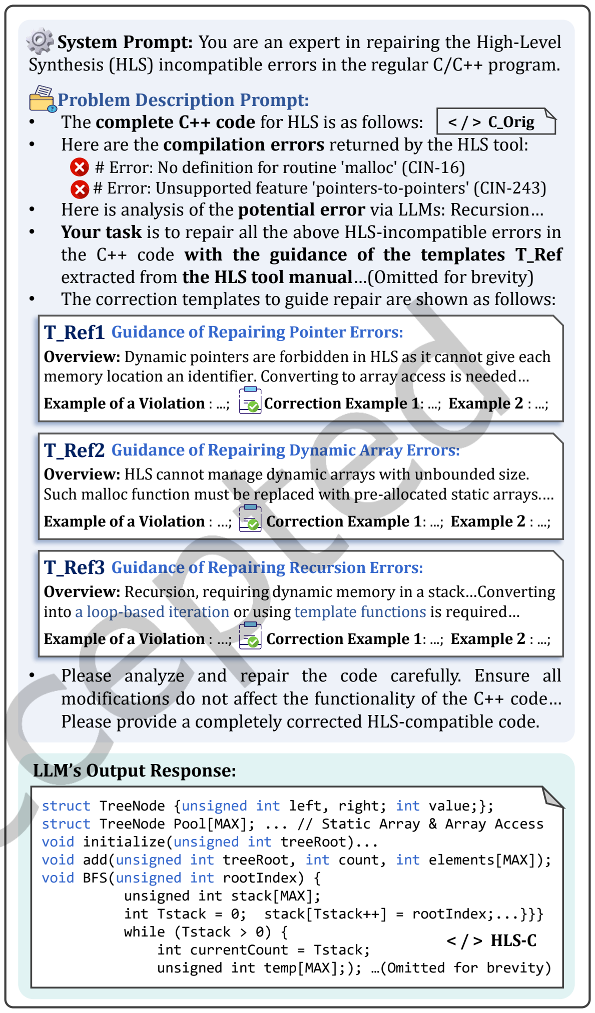
LLM's Output Response:

Step 1: The use of recursion in the BFS function, calling BFS(Pool[rootIndex].left); and BFS(Pool[rootIndex].right);.

Step 2: Inside the for loop within the BFS function, the recursive calls to handle left and right child nodes occur.

Step 3: HLS tools generally do not support recursion because the recursion depth cannot be statically determined, and therefore cannot be mapped to hardware modules.

Fig. 14. Step-Wise Reasoning: Identify potential HLS-incompatible errors in the regular C/C++ code via the LLM.



System Prompt: You are an expert in repairing the High-Level Synthesis (HLS) incompatible errors in the regular C/C++ program.

Problem Description Prompt:

- The **complete C++ code** for HLS is as follows: </> C_Orig
- Here are the **compilation errors** returned by the HLS tool:
  # Error: No definition for routine 'malloc' (CIN-16)
  # Error: Unsupported feature 'pointers-to-pointers' (CIN-243)
- Here is analysis of the **potential error** via LLMs: Recursion...
- **Your task** is to repair all the above HLS-incompatible errors in the C++ code **with the guidance of the templates T_Ref** extracted from **the HLS tool manual**...(Omitted for brevity)
- The correction templates to guide repair are shown as follows:

T_Ref1 Guidance of Repairing Pointer Errors:
Overview: Dynamic pointers are forbidden in HLS as it cannot give each memory location an identifier. Converting to array access is needed...
Example of a Violation : ...; **Correction Example 1**: ...; **Example 2** : ...;

T_Ref2 Guidance of Repairing Dynamic Array Errors:
Overview: HLS cannot manage dynamic arrays with unbounded size. Such malloc function must be replaced with pre-allocated static arrays....
Example of a Violation : ...; **Correction Example 1**: ...; **Example 2** : ...;

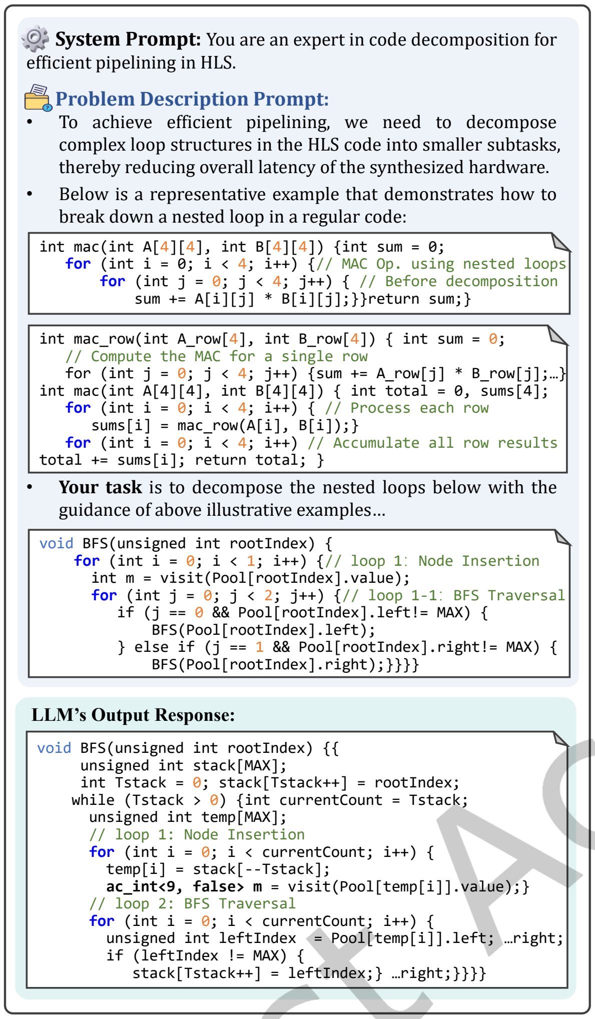
T_Ref3 Guidance of Repairing Recursion Errors:
Overview: Recursion, requiring dynamic memory in a stack...Converting into a loop-based iteration or using template functions is required...
Example of a Violation : ...; **Correction Example 1**: ...; **Example 2** : ...;

- Please analyze and repair the code carefully. Ensure all modifications do not affect the functionality of the C++ code... Please provide a completely corrected HLS-compatible code.

LLM's Output Response:

```
struct TreeNode {unsigned int left, right; int value;};
struct TreeNode Pool[MAX]; ... // Static Array & Array Access
void initialize(unsigned int treeRoot)...
void add(unsigned int treeRoot, int count, int elements[MAX]);
void BFS(unsigned int rootIndex) {
        unsigned int stack[MAX];
        int Tstack = 0;  stack[Tstack++] = rootIndex;...}}
        while (Tstack > 0) {                        </> HLS-C
            int currentCount = Tstack;
            unsigned int temp[MAX];); ...(Omitted for brevity)
```

Fig. 15. Refactoring with RAG: LLM refactorings the HLS-incompatible errors according to external guidance from the HLS tool manual.



System Prompt: You are an expert in code decomposition for efficient pipelining in HLS.

Problem Description Prompt:

- To achieve efficient pipelining, we need to decompose complex loop structures in the HLS code into smaller subtasks, thereby reducing overall latency of the synthesized hardware.
- Below is a representative example that demonstrates how to break down a nested loop in a regular code:

```
int mac(int A[4][4], int B[4][4]) {int sum = 0;
   for (int i = 0; i < 4; i++) {// MAC Op. using nested loops
       for (int j = 0; j < 4; j++) { // Before decomposition
           sum += A[i][j] * B[i][j];}}return sum;}
```

```
int mac_row(int A_row[4], int B_row[4]) { int sum = 0;
   // Compute the MAC for a single row
   for (int j = 0; j < 4; j++) {sum += A_row[j] * B_row[j];…}
int mac(int A[4][4], int B[4][4]) { int total = 0, sums[4];
   for (int i = 0; i < 4; i++) { // Process each row
      sums[i] = mac_row(A[i], B[i]);}
   for (int i = 0; i < 4; i++) // Accumulate all row results
total += sums[i]; return total; }
```

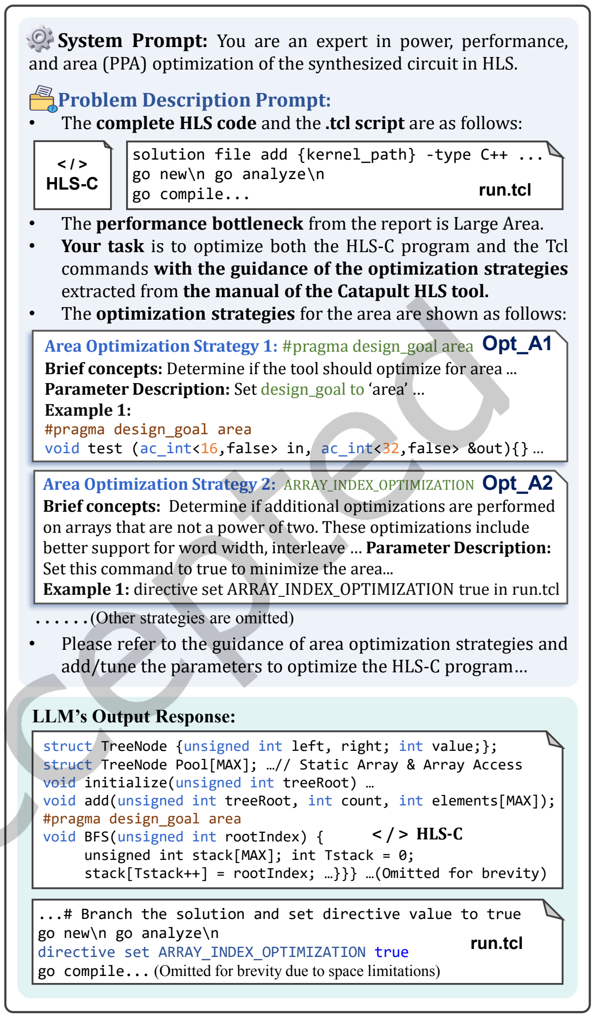
- **Your task** is to decompose the nested loops below with the guidance of above illustrative examples…

```
void BFS(unsigned int rootIndex) {
    for (int i = 0; i < 1; i++) {// loop 1: Node Insertion
      int m = visit(Pool[rootIndex].value);
      for (int j = 0; j < 2; j++) {// loop 1-1: BFS Traversal
         if (j == 0 && Pool[rootIndex].left!= MAX) {
             BFS(Pool[rootIndex].left);
         } else if (j == 1 && Pool[rootIndex].right!= MAX) {
             BFS(Pool[rootIndex].right);}}}}
```

LLM's Output Response:

```
void BFS(unsigned int rootIndex) {{
     unsigned int stack[MAX];
     int Tstack = 0; stack[Tstack++] = rootIndex;
    while (Tstack > 0) {int currentCount = Tstack;
      unsigned int temp[MAX];
      // loop 1: Node Insertion
      for (int i = 0; i < currentCount; i++) {
        temp[i] = stack[--Tstack];
        ac_int<9, false> m = visit(Pool[temp[i]].value);}
      // loop 2: BFS Traversal
      for (int i = 0; i < currentCount; i++) {
        unsigned int leftIndex  = Pool[temp[i]].left; …right;
        if (leftIndex != MAX) {
           stack[Tstack++] = leftIndex;} …right;}}}}
```

Fig. 16. Pipeline-Aware Decomposition via the LLM.



System Prompt: You are an expert in power, performance, and area (PPA) optimization of the synthesized circuit in HLS.

Problem Description Prompt:

- The **complete HLS code** and the **.tcl script** are as follows:

</> HLS-C

```
solution file add {kernel_path} -type C++ ...
go new\n go analyze\n
go compile...
```

run.tcl

- The **performance bottleneck** from the report is Large Area.
- **Your task** is to optimize both the HLS-C program and the Tcl commands **with the guidance of the optimization strategies** extracted from **the manual of the Catapult HLS tool.**
- The **optimization strategies** for the area are shown as follows:

Area Optimization Strategy 1: #pragma design_goal area **Opt_A1**
Brief concepts: Determine if the tool should optimize for area ...
Parameter Description: Set design_goal to 'area' ...
Example 1:

```
#pragma design_goal area
void test (ac_int<16,false> in, ac_int<32,false> &out){} ...
```

Area Optimization Strategy 2: ARRAY_INDEX_OPTIMIZATION **Opt_A2**
Brief concepts: Determine if additional optimizations are performed on arrays that are not a power of two. These optimizations include better support for word width, interleave ... **Parameter Description:** Set this command to true to minimize the area...
Example 1: directive set ARRAY_INDEX_OPTIMIZATION true in run.tcl

......(Other strategies are omitted)

- Please refer to the guidance of area optimization strategies and add/tune the parameters to optimize the HLS-C program…

LLM's Output Response:

```
struct TreeNode {unsigned int left, right; int value;};
struct TreeNode Pool[MAX]; …// Static Array & Array Access
void initialize(unsigned int treeRoot) …
void add(unsigned int treeRoot, int count, int elements[MAX]);
#pragma design_goal area
void BFS(unsigned int rootIndex) {
     unsigned int stack[MAX]; int Tstack = 0;
     stack[Tstack++] = rootIndex; …}}} …(Omitted for brevity)
```

</> HLS-C

```
...# Branch the solution and set directive value to true
go new\n go analyze\n
directive set ARRAY_INDEX_OPTIMIZATION true
go compile... (Omitted for brevity due to space limitations)
```

run.tcl

Fig. 17. PPA Optimization via the LLM.

References

[1] Stefan Abi-Karam, Rishov Sarkar, Allison Seigler, Sean Lowe, Zhigang Wei, Hanqiu Chen, Nanditha Rao, Lizy John, Aman Arora, Cong Hao, “HLSFactory: A Framework Empowering High-Level Synthesis Datasets for Machine Learning and Beyond,” ACM/IEEE International Symposium on Machine Learning for Computer-Aided Design (MLCAD), 2024.

[2] Yunsheng Bai, Atefeh Sohrabizadeh, Zongyue Qin, Ziniu Hu, Yizhou Sun, Jason Cong, “Towards a Comprehensive Benchmark for High-Level Synthesis Targeted to FPGAs,” Advances in Neural Information Processing Systems (NeurIPS), 2023.

[3] Jiho Kim, Cong Hao, “RealProbe: An Automated and Lightweight Performance Profiler for In-FPGA Execution of High-Level Synthesis Designs,” IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2025.

[4] Kangwei Xu, Bing Li, Grace Li Zhang, Ulf Schlichtmann, “HLSTester: Efficient Testing of Behavioral Discrepancies with LLMs for High-Level Synthesis,” ACM/IEEE International Conference on Computer-Aided Design (ICCAD), 2025.

[5] Qi Sun, Tinghuan Chen, Siting Liu, Jianli Chen, Hao Yu, and Bei Yu, “Correlated Multi-objective Multi-fidelity Optimization for HLS Directives Design,” ACM Transactions on Design Automation of Electronic Systems (TODAES), 2022.

[6] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong, “AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators,” ACM Transactions on Design Automation of Electronic Systems (TODAES), 2022.

[7] Andrew Nazareth, Bernardo Perez, Rachel Paul, James Root, Ritarka Samanta, William Vaught, Stefan Abi-Karam, Rishov Sarkar, Cong Hao, “Cask HLS: A Better Development Tool for Vitis HLS,” IEEE International Opportunity Research Scholars Symposium (ORSS), 2023.

[8] Nadesh Ramanathan, George A. Constantinides and John Wickerson, “Precise pointer analysis in high-level synthesis,” ACM/IEEE International Conference on Field-Programmable Logic and Applications (FPL), 2020.

[9] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, Miryung Kim, “Refactoring for Heterogeneous Computing with FPGA," ACM/IEEE International Conference on Software Engineering (ICSE), 2020.

[10] David B. Thomas, “Synthesisable recursion for C++ HLS tools,” IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2016.

[11] Zeping Xue, David B. Thomas, “SynADT: Dynamic data structures in high level synthesis,” IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), 2016.

[12] David B. Thomas, “Templatised Soft Floating-Point for High-Level Synthesis,” IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), 2019.

[13] Hsuan Hsiao, Jason H. Anderson, “Sensei: An area-reduction advisor for FPGA high-level synthesis,” ACM/IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018.

[14] “Leetcode Problem Set,” Accessed: 2023. [Online]. Available: https://leetcode.com/problemset/.

[15] “Siemens EDA Catapult Synthesis User and Reference Manual,” Accessed: 2023. [Online]. Available: https://support.sw.siemens.com/.

[16] Kangwei Xu, Grace Li Zhang, Ulf Schlichtmann, Bing Li, “Logic Design of Neural Networks for High-Throughput and Low-Power Applications,” ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2024.

[17] Kangwei Xu, Yuanqing Cheng, “Fault Testing and Diagnosis Techniques for Carbon Nanotube-Based FPGAs,” ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2022.

[18] Siyuan Lu, Kangwei Xu, Peng Xie, Rui Wang, Yuanqing Cheng, “Testing and Fault Tolerance Techniques for Carbon Nanotube-Based FPGAs,” Elsevier Integration, VLSI Journal, 2025.

[19] Bingyang Liu, Haoyi Zhang, Xiaohan Gao, Xiyuan Tang, Yibo Lin, Runsheng Wang, Ru Huang, "LayoutCopilot: LLM-Empowered Analog Layout Design Towards Enhanced Human-Machine Interaction,” IEEE International Symposium on Circuits and Systems (ISCAS), 2025.

[20] Jiahao Gai, Hao Chen, Zhican Wang, Hongyu Zhou, Wanru Zhao, Nicholas Lane, Hongxiang Fan, “Exploring code language models for automated hls-based hardware generation: Benchmark, infrastructure and analysis,” ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2025.

[21] Stefan Abi-Karam, Cong Hao, “HLS-Eval: A Benchmark and Framework for Evaluating LLMs on High-Level Synthesis Design Tasks,” IEEE International Conference on LLM-Aided Design (ICLAD), 2025.

[22] Ali Rizvi, Nia Simon, Janet Tocho, Asil Yongaci, Stefan Abi-Karam, Cong Hao, “Evaluating Large Language Models for High-Level Synthesis,” IEEE Opportunity Research Scholars Symposium (ORSS), 2024.

[23] Jason Blocklove, Siddharth Garg, Ramesh Karri, Hammond Pearce, “Chip-Chat: Challenges and Opportunities in Conversational Hardware Design,” IEEE Workshop on Machine Learning for CAD (MLCAD), 2023.

[24] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, Ramesh Karri, “AutoChip: Automating HDL Generation Using LLM Feedback,” arXiv preprint: 2311.04887, 2023.

[25] Zhuorui Zhao, Ruidi Qiu, Ing-Chao Lin, Grace Li Zhang, Bing Li, Ulf Schlichtmann, “VRank: Enhancing Verilog Code Generation from Large Language Models via Self-Consistency,” ACM/IEEE International Symposium on Quality Electronic Design (ISQED), 2025.

[26] Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, Bing Li, “CorrectBench: Automatic Testbench Generation with Functional Self-Correction using LLMs for HDL Design,” ACM/IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), 2025.

[27] Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, Bing Li, "AutoBench: Automatic Testbench Generation and Evaluation Using LLMs for HDL Design," ACM/IEEE International Symposium on Machine Learning for Computer-Aided Design (MLCAD), 2024.
[28] Wenhao Sun, Bing Li, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, Ulf Schlichtmann, "Paradigm-Based Automatic HDL Code Generation Using LLMs," ACM/IEEE International Symposium on Quality Electronic Design (ISQED), 2025.
[29] Luca Collini, Andrew Hennessee, Ramesh Karri, Siddharth Garg, "Can reasoning models reason about hardware? an agentic HLS perspective," arXiv preprint: 2503.12721, 2025.
[30] Charlie Campbell, Hao Mark Chen, Wayne Luk, Hongxiang Fan, "Enhancing LLM-based Quantum Code Generation with Multi-Agent Optimization and Quantum Error Correction," ACM/IEEE Design Automation Conference (DAC), 2025.
[31] Chandan Kumar Jha, Muhammad Hassan, Khushboo Qayyum, Sallar Ahmadi-Pour, Kangwei Xu, Ruidi Qiu, Jason Blocklove, Luca Collini, Andre Nakkab, Ulf Schlichtmann, Grace Li Zhang, Ramesh Karri, Bing Li, Siddharth Garg, Rolf Drechsler, "Large Language Models (LLMs) for Verification, Testing, and Design," IEEE European Test Symposium (ETS), 2025.
[32] Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, Zhuoliang Zhao, Yuan Cheng, Yudong Pan, Yiqi Liu, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, Xiaowei Li, "Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework," ACM/IEEE Design Automation Conference (DAC), 2024.
[33] Jingcun Wang, Yu-Guang Chen, Ing-Chao Lin, Bing Li, Grace Li Zhang, "Basis Sharing: Cross-Layer Parameter Sharing for Large Language Model Compression," International Conference on Learning Representations (ICLR), 2025.
[34] Yun-Da Tsai, Mingjie Liu, Haoxing Ren, "Automatically Fixing RTL Syntax Errors with Large Language Model," ACM/IEEE Design Automation Conference (DAC), 2024.
[35] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, Brendan Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models," IEEE Symposium on Security and Privacy, 2023
[36] Harshit Joshi, Jose Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, Gust Verbruggen, "Repair is nearly generation: multilingual program repair with LLMs," Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI), 2023.
[37] Kangwei Xu, Ruidi Qiu, Zhuorui Zhao, Grace Li Zhang, Ulf Schlichtmann, Bing Li, "LLM-Aided Efficient Hardware Design Automation," arXiv preprint: 2410.18582, 2024.
[38] He Ye, Martin Monperrus, "ITER: Iterative Neural Repair for Multi-Location Patches," ACM/IEEE International Conference on Software Engineering (ICSE), 2024.
[39] Chunqiu Steven Xia, Yuxiang Wei, Lingming Zhang, "Automated program repair in the era of large pre-trained language models," ACM/IEEE International Conference on Software Engineering (ICSE), 2023.
[40] Luca Collini, Siddharth Garg, Ramesh Karri, "C2HLSC: Can LLMs Bridge the Software-to-Hardware Design Gap?," IEEE International Workshop on LLM-Aided Design (LAD), 2024.
[41] Luca Collini, Siddharth Garg, Ramesh Karri, "C2HLSC: Leveraging Large Language Models to Bridge the Software-to-Hardware Design Gap," arXiv preprint: 2412.00214, 2024.
[42] Chenwei Xiong, Cheng Liu, Huawei Li, Xiaowei Li, "HLSPilot: LLM-based High-Level Synthesis," ACM/IEEE International Conference on Computer-Aided Design (ICCAD), 2024.
[43] Kangwei Xu, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, Ulf Schlichtmann, Bing Li, "Automated C/C++ Program Repair for High-Level Synthesis via Large Language Models," ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD), 2024.
[44] Tinghuan Chen, Grace Li Zhang, Bei Yu, Bing Li, Ulf Schlichtmann, "Machine Learning in Advanced IC Design: A Methodological Survey," IEEE Design & Test, 2023.
[45] "Siemens EDA Catapult High-Level Synthesis Tools," Accessed: 2023. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/.
[46] Nils Reimers, Iryna Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," ACL Empirical Methods in Natural Language Processing (EMNLP), 2019.
[47] Kalyan Muthukumar and Gautam Doshi, "Software Pipelining of Nested Loops," ACM International Conference on Compiler Construction, 2001.
[48] Junyi Liu, John Wickerson, George A. Constantinides, "Loop Splitting for Efficient Pipelining in High-Level Synthesis," IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2016.
[49] "GPT-4 Turbo through OpenAI API," Accessed: 2024. [Online]. Available: https://platform.openai.com/.

Received 15 March 2025; revised 13 June 2025; accepted 5 July 2025