

000 HLSTRANS: DATASET FOR C-TO-HLS HARDWARE 001 CODE SYNTHESIS 002 003 004

005 **Anonymous authors**

006 Paper under double-blind review

007 008 ABSTRACT 009

010 High-Level Synthesis (HLS) enables hardware design from C/C++ kernels but
011 requires extensive transformations, such as restructuring code, inserting prag-
012 mas, adapting data types, and repairing non-synthesizable constructs, to achieve
013 efficient FPGA implementations. While large language models (LLMs) show
014 promise in automating these transformations, progress has been limited by the
015 absence of large-scale, well-structured datasets. Existing HLS datasets focus pri-
016 marily on resource estimation, lack paired C and HLS examples with testbenches,
017 and cover only a narrow set of optimizations. We introduce HLStrans, the first
018 benchmark-scale dataset for LLM-driven C-to-HLS synthesis. HLStrans con-
019 tains over 124K paired C and HLS programs for real-world applications, with
020 full testbenches and synthesis-based annotations of latency and resource usage.
021 The dataset systematically captures five categories of transformations and is en-
022 riched by an automated augmentation pipeline combining LLMs, Monte Carlo
023 Tree Search (MCTS), and Design Space Exploration (DSE). We benchmark state-
024 of-the-art LLMs on HLStrans, demonstrating that retrieval and fine-tuning signif-
025 icantly improve success rates and performance.

026 027 1 INTRODUCTION

028 Specialized computing systems, particularly FPGAs, are increasingly deployed to accelerate
029 compute-intensive workloads in domains such as machine learning, signal processing, and data
030 analytics. High-Level Synthesis (HLS) has emerged as a key methodology for bridging software
031 and hardware, allowing engineers to describe functionality in C/C++ and automatically generate
032 hardware-ready RTL. However, generating high-performance HLS code is far from a direct trans-
033 lation: it requires structural code refactoring, insertion of optimization pragmas, adaptation of data
034 types, replacement of functions with hardware-friendly intrinsics, and strict compliance with HLS
035 coding styles. Therefore, we define the **C-to-HLS transformation** task as follows: given a sequen-
036 tial C/C++ kernel, generate a synthesizable HLS implementation that achieves efficient hardware
037 acceleration on an FPGA platform. This task exemplifies the challenges at the intersection of AI
038 and EDA, demanding not only correctness but also hardware-aware optimization. [The impact of
039 this task is described in Appendix A.5.](#)

040 Recent work has demonstrated the potential of large language models (LLMs) for HLS code gen-
041 eration. Early studies explored direct translation from C++ to synthesizable HLS code, while oth-
042 ers focused on automating pragma insertion, repairing unsynthesizable constructs, or leveraging
043 retrieval-augmented and chain-of-thought prompting to improve optimization quality (Collini et al.,
044 2024; Xiong et al., 2024; Bhattacharyya et al., 2024; Xu et al., 2024; Prakriya et al., 2025). While
045 promising, these approaches are constrained by the lack of comprehensive benchmarks: existing
046 evaluations are conducted on small, fragmented collections of kernels, making it difficult to repro-
047 duce results or compare methods fairly. Without a unified, large-scale dataset, it remains challenging
048 to systematically assess or advance LLMs on the C-to-HLS task.

049 Although several datasets for HLS exist, such as HLSsyn (Bai et al., 2023), HLSdataset (Wei et al.,
050 2023), MLSBench (Goswami et al., 2022), and HLSfactory (Abi-Karam et al., 2024), they fall
051 short for this purpose. Most are designed for resource estimation rather than code transformation,
052 are limited in scale (typically a few hundred to a few thousand kernels), and rarely include paired
053 examples of original C code, optimized HLS code, and testbenches. Moreover, they capture only

054 a narrow slice of transformation diversity, focusing mainly on pragma insertion and overlooking
 055 critical steps such as code restructuring, data type adaptation, and repair of unsupported C constructs.
 056 As a result, current datasets cannot serve as a benchmark foundation for training or evaluating LLMs
 057 on realistic C-to-HLS synthesis.

058 To address these gaps, we present HLStrans¹, the first benchmark-scale dataset explicitly designed
 059 for LLM-driven C-to-HLS transformation. HLStrans contains over 124,200 C and HLS pairs drawn
 060 from diverse real-world applications, covering domains such as linear algebra, machine learning,
 061 DSP, image processing, and cryptography. Each entry includes a triple: the original C kernel, an
 062 optimized HLS implementation, and a validation testbench, with annotations of latency and re-
 063 source metrics obtained via synthesis. The dataset systematically captures five categories of trans-
 064 formations: code restructuring, pragma insertion, data type adaptation, function replacement, and
 065 HLS-compliant repair, ensuring broad coverage of hardware-oriented optimizations. To further en-
 066 rich this corpus, we introduce an automated augmentation framework that combines LLMs, Monte
 067 Carlo Tree Search (MCTS), and Design Space Exploration (DSE) to generate diverse, synthesizable
 068 variants guided by synthesis feedback.

069 In summary, our contributions are threefold:

- 070 • We release HLStrans, the first large-scale dataset for C-to-HLS transformation, enabling LLM
 071 training and fair benchmarking;
- 072 • We propose a novel augmentation pipeline that produces diverse, high-quality HLS implemen-
 073 tations;
- 074 • We provide extensive evaluations of open-source and closed-source LLMs, showing that retrieval
 075 and fine-tuning on HLStrans significantly boost synthesis success rates and performance. By
 076 positioning HLStrans as both a resource and a benchmark, we aim to catalyze progress in LLM-
 077 powered hardware design and accelerate the integration of AI into future EDA workflows.

079 2 BACKGROUND AND RELATED WORKS

081 **LLM aided C to HLS.** There is an increasing body of literature on applying LLMs to generate
 082 HLS design from original C code. Collini et al. (2024) evaluates the basic task of translating naive
 083 C++ into synthesizable HLS C++. Bhattacharyya et al. (2024) demonstrates that LLMs can au-
 084 tomate HLS pragmas and optimizations to produce synthesizable, high-performance RTL from C
 085 on image-processing benchmarks. Xu et al. (2024) presents an LLM-driven HLS program-repair
 086 framework that transforms C/C++ into synthesizable HLS-C. Xiong et al. (2024) extends this ap-
 087 proach with retrieval-augmented generation and chain-of-thought prompting to deliver optimized
 088 HLS implementations across nine applications. However, to date, no work has evaluated LLM’s
 089 capabilities transforming C code to HLS codes on a large-scale dataset.

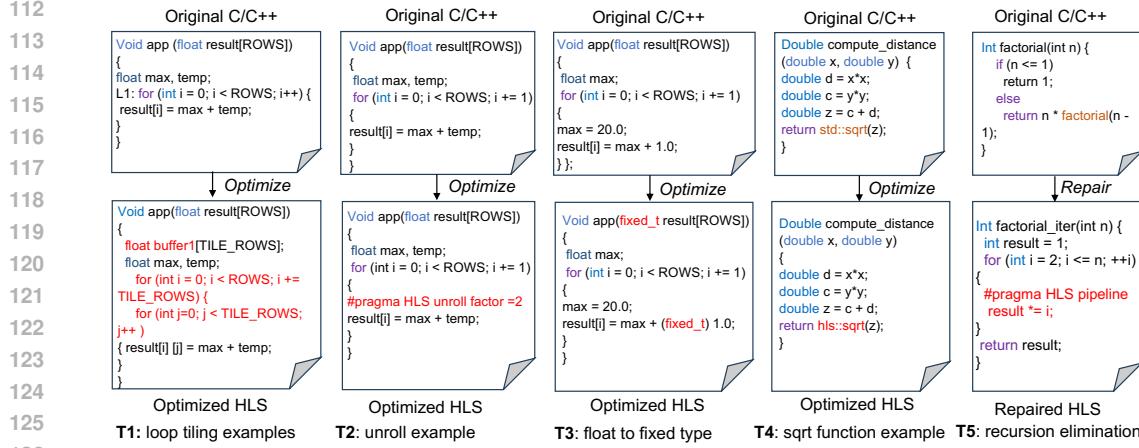
090 **HLS code dataset.** HLSsyn (Bai et al., 2023) focuses on incorporating a diverse set of optimization
 091 pragmas but contains only 42 kernels for training and evaluating design-quality prediction models.
 092 HLSDataset (Wei et al., 2023), which aggregates 34 data sources into roughly 18K samples, targets
 093 power, resource, and timing estimation. MLSBench (Goswami et al., 2022) is an open-source corpus
 094 produced with the Xilinx Vivado HLS flow; it covers 17 C/C++ and 13 SystemC benchmarks, but
 095 provides only HLS log files and reports. DB4HLS (Ferretti et al., 2021) introduced a database
 096 of more than 100,000 HLS design points generated from MachSuite via exhaustive design-space
 097 exploration. Likewise, Dai et al. (2018) released about 1,300 designs created from benchmarks.
 098 Despite these valuable resources, they suffer from three key limitations when used to evaluate LLMs’
 099 ability to translate C code into HLS:

100 First, prior HLS datasets have primarily targeted **quality-of-results (QoR) estimation rather than**
 101 **C-to-HLS code generation**, and the underlying program sources are limited. Though varying tool
 102 configurations can yield many synthesized samples, the scarcity of distinct source programs pre-
 103 vents an LLM from learning diverse program structures needed for C-to-HLS tasks. Moreover, the
 104 selected programs are typically short, making them inadequate for fully assessing LLMs’ capability.

105 Second, Existing datasets **inadequately capture comprehensive C-to-HLS transformations**, fo-
 106 cusing largely on pragma insertion. Generating high-performance HLS code from standard C/C++

1^{https://anonymous.4open.science/r/HLStrans-B578/}

108 for FPGAs requires a series of systematic transformations to expose parallelism, optimize data
 109 movement, and conform to HLS-friendly coding styles. While the detailed transformations are
 110 in Appendix A.1, these transformations fall into five broad categories, shown in Figure 1.
 111



127 Figure 1: C/C++ to HLS code transformation examples. T1: Apply loop tiling and local buffering
 128 to improve data locality. T2: Unroll inner loops to increase parallelism and throughput. T3: Convert
 129 floating-point to fixed-point types to reduce resource use and latency. T4: Replace standard math
 130 calls with HLS intrinsics (e.g. `hls::sqrt`) for synthesizable implementations. T5: Eliminate recursion
 131 by refactoring to iterative code so the design can be synthesized.

132 *T1: Code Restructuring.* Refactor algorithms to expose pipelining and dataflow, apply loop tiling,
 133 memory coalescing, ping-pong buffering, and reorganize control logic for parallel or streaming execu-
 134 tion. *T2: Directive (Pragma) Insertion.* Place HLS pragmas to guide the tool scheduler, such
 135 as data flow, pipeline, loop partition, and interface specifications, to fine-tune performance and
 136 resource usage. *T3: Data-Type Adaptation.* Replace generic C types with HLS-specific arbitrary-
 137 precision types: convert floating point to fixed point (`ap_fixed`) for resource optimization, standard
 138 integers to bit-accurate (`ap_int/ap_uint`), and customize bit widths to match application precision re-
 139 quirements. *T4: Transformation of Functions.* Transform standard C functions into HLS-optimized
 140 kernels or intrinsics (such as converting the `std::sqrt` function to the `hls::sqrt` function) to better
 141 leverage FPGA fabric and specialized accelerators. *T5: HLS-Compliant Coding Style.* Eliminate
 142 unsupported C constructs such as dynamic memory allocation (`malloc/free`), recursion, and certain
 143 pointer arithmetic patterns; restructure code to use static arrays, simple loops, and explicit hand-
 144 shaking for communication.

145 Third, they are not organized as **paired C-and-HLS examples and omit the corresponding test-**
 146 **benches** needed for LLM-based HLS code optimization, which are not ready for LLM to verify its
 147 output.

148 Compared with previous works, Table 1 concludes that our dataset has more kinds of sources and
 149 supports more transformations, making it ready for LLM code generation.

3 HLSTRANS DATASETS CONSTRUCTION

153 Open-source HLS datasets are scarce and poorly structured, which limits their usefulness for training
 154 LLMs. We propose an automated pipeline to generate high-quality HLS datasets from existing
 155 resources. The pipeline has three stages: (1) collect high-quality human optimized open-source
 156 HLS examples; (2) perform targeted data augmentation on human optimized kernels to produce
 157 many viable candidates; (3) select the efficient HLS implementations from those candidates. Figure
 158 2 shows our dataset construction process.

3.1 DATASET COLLECTION

159 Firstly, we harvest code from GitHub, selecting repositories with at least ten stars. However, man-
 160 ually optimized codebases often exhibit inconsistent formatting and sparse documentation, which

162 Table 1: Comparison of HLS datasets. *QoR*: quality of result prediction. *Transformation*: C to HLS
 163 transformations mentioned in Figure 1. ✓: included. ✗: not included
 164

Attributes	Dai	MLSBench	DB4HLS	HLSdataset	HLSSyn	HLStrans
Samples	1,300	6,000	124,106	18,876	42,000	124,200
Programs	65	30	19	34	42	309
Purpose	QoR	QoR	QoR	QoR	QoR	Code generation
Transformations	T2	T2	T2	T1,T2	T2	T1, T2, T3, T4, T5
Testbench	No	No	No	No	No	Yes
Programs						
CHStone(Hara et al., 2008)	✓	✓	✗	✓	✗	✓
Polybench(Pouchet & Yuki, 2012)	✗	✗	✗	✓	✓	✓
Rodinia(Che et al., 2009)	✗	✗	✗	✗	✗	✓
Machsuite(Reagen et al., 2014)	✓	✓	✓	✓	✓	✓
Rosetta(Zhou et al., 2018)	✗	✗	✗	✓	✗	✓
C2HLS(Collini et al., 2024)	✗	✗	✗	✗	✗	✓
PP4FPGA(Kastner et al., 2018)	✗	✗	✗	✗	✗	✓
Forgebench(Wanna et al., 2025)	✗	✗	✗	✗	✗	✓
HLSfactory(Abi-Karam et al., 2024)	✗	✗	✗	✗	✗	✓
Others (GitHub)	✗	✗	✗	✗	✗	✓

183
 184
 185 hinders LLM-driven code generation. Public kernels also frequently depend on unexpanded macros
 186 and bundle extraneous utility functions that obscure the core algorithm. To make C to HLS tasks
 187 readily consumable by LLMs, we package each design with the following files:

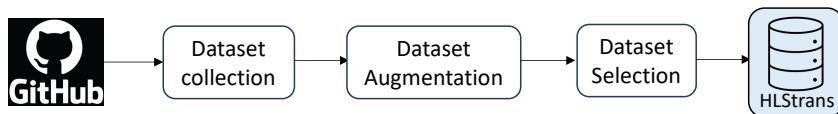
- 188 • Single original file x that is a slow original C/C++ codes.
- 189 • Single optimized HLS file y that implements the kernel, including a top function and, if neces-
 190 sary, any sub-functions and specialized data types. The file must be synthesizable and not exceed
 191 the resources of the platform.
- 192 • Self-contained C++ testbench tb includes all test cases and validation logic necessary to verify
 193 the kernel’s outputs against expected results. We manually write all the testbenches and adjust
 194 the optimized HLS code to ensure it passes all tests. [The coverage of testbenches are described
 195 in Appendix A.7.](#)

196 Therefore, we construct triples (x, y, tb) . If the original file x is synthesisable, the execution cycles
 197 from synthesis reports of y must be less than x . If the original file x is not synthesisable, y should
 198 be synthesisable.

200 3.2 DATASET AUGMENTATION

201 Relying solely on collected repositories is insufficient because high-quality hardware codes are far
 202 scarcer than general software. To generate richer, more useful examples, we designed an automated
 203 dataset-augmentation framework that synthesizes additional C to HLS variants.

204 We formulate the C to HLS translation as a combinatorial search problem: selecting appropriate
 205 combinations of code transformations to meet performance and resource targets. Our approach
 206 proceeds in two stages. First, an LLM agent guided by Monte Carlo Tree Search (MCTS) proposes
 207 and explores semantic-preserving code transformations that expose parallelism and produce HLS-
 208 friendly structure. Second, for each candidate design we apply automated design-space-exploration



211
 212 Figure 2: HLStrans Dataset Construction Process.
 213
 214
 215

(DSE) tools to tune pragmas and low-level implementation choices. Both stages are evaluated in the loop using EDA feedback (performance, resource utilization, and compilation outcomes), enabling MCTS and DSE to efficiently navigate the large, combinatorial action space (see Figure 3).

First, MCTS performs structured exploration by balancing the exploitation of high-reward actions with the exploration of uncertain or less-visited regions of the search space. The optimization policy is generated by the retrieval augmentation generation (RAG) module. The search is guided to choose the suitable policy by both the verification pipeline and a reward model. The reward model incorporates detailed feedback from the HLS toolchain, including synthesis success or failure, compile warnings, and performance metrics such as resource usage, latency, and throughput. This heuristic-driven strategy enables the agent to iteratively refine transformation sequences and produce more high-quality, synthesizable HLS designs. Second, HLS directive design space exploration using genetic algorithms (Ferikoglou et al., 2023) is adopted. It inserts pipeline, unroll, and partition pragmas to produce more effective data samples. Through iterative refinement, the framework converges toward optimized and synthesizable HLS code.

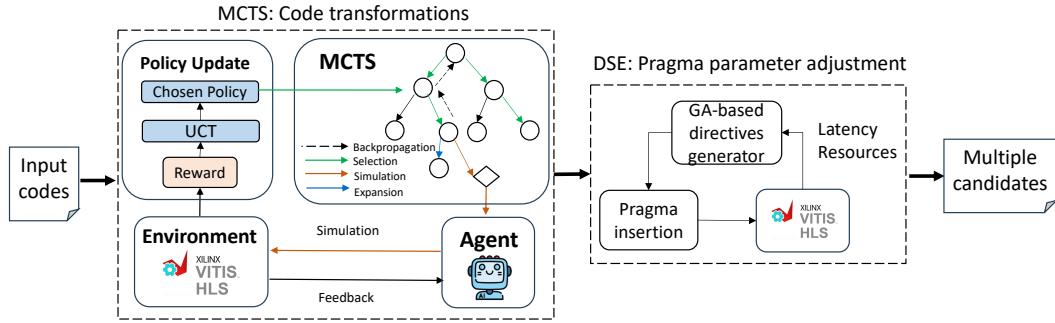


Figure 3: HLStrans Dataset Augmentation Framework.

3.2.1 MONTE CARLO TREE SEARCH (MCTS)

We formulate HLS optimization as an MCTS problem. The *environment* is the Vitis HLS toolchain, which provides synthesis, resource, and performance feedback. The *agent* is an LLM that applies code transformations. *Actions* include (i) RAG-based retrieval of known optimization policies and (ii) ReAct-based reasoning over compiler warnings. The *state* is the current HLS code, and the *reward* follows rule-based shaping: -2 for verification failure, -1 for synthesis/resource failure, 0 if worse, 1 if improved, and 2 if improved with timing met. In our cases, the MCTS agent begins at the initial state S_0 (the root node), which is the naive HLS code. From a state S_t , the agent applies a optimization policy π , i.e., an action $a_t \in \mathcal{A}$, transitioning to the subsequent state S_{t+1} . This new state optimizes the existing code sequence by applying the new optimizations. Upon reaching a terminal state S_T , the agent receives a deferred reward $R(S_T)$. $N(S_t)$, the total number of times S_t has been visited.

Selection: We employ the upper confidence bounds for trees (UCT) algorithm (Gelly & Wang, 2006) to choose nodes. The UCT formula includes the average reward for the current state, which encourages the path that can bring high reward, while U term measures the associated uncertainty, which encourages the exploration of new paths. This approach effectively balances the trade-off between exploration and exploitation.

Expansion, simulation and backpropagation: Expansion is to explore the unchosen action. We leverage LLM to determine the next action from the unexplored. The decision process is driven by program analysis in conjunction with the history of adopted optimizations, enabling LLM to accurately assess and select the most promising action. After the analysis of LLM for state s_t at time steps t , the next action a_{t+1} will decided by $a_{t+1} = llm(s_t)$. Simulation employs the agent to apply transformations and evaluates them via HLS synthesis measuring estimated latency and resource usage to compute the reward $R(s_t, a_t)$; During backpropagation, these rewards are propagated up the search tree to update node values, refining the agent's estimates and guiding future action selection. Once we no longer observe significant improvements, the search process is halted, and the best-performing rewritten design is selected. The detailed description of MCTS framework is described in Appendix A.2.

270 3.2.2 DESIGN SPACE EXPLORATION
271

272 The tool implements an automated HLS design-space explorer that uses a genetic-algorithm op-
273 timizer to discover effective directive combinations, specifically loop pipelining, loop unrolling,
274 and array partitioning, that maximize performance and resource utilization. To traverse the solution
275 space, we utilize the NSGAII algorithm (Deb et al., 2002) implemented in PyMOO library (Blank &
276 Deb, 2020), known for its ability to bypass local optimal and quickly converge to efficient solutions.
277 The detail DSE implementation is introduced in Appendix A.2.2.

278 3.3 DATASET SELECTIONS
279

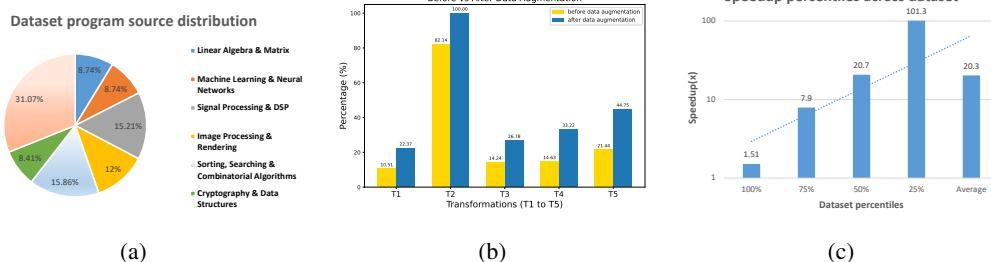
280 After generating multiple dataset candidates, we select the efficient samples. If the input codes can
281 not be synthesized, we choose the candidates which can be synthesized. If the input codes can be
282 synthesized, we choose the candidates whose latency is less than input codes. To give the model
283 clearer guidance, we borrow ideas from Shypula et al. (2023) and attach a “performance tag” and
284 “resource tag” to each solution during training. Each tag reflects how close that program comes to
285 the best attainable performance with resources minimized, using a scale from 0 to 10, respectively.
286

287 3.4 HLSTRANS STATISTICS
288

289 Overall, we leverage DeepSeek-R1 to generate high-quality synthetic code examples, the AMD
290 Vitis HLS EDA tool, and DSE tools to validate, annotate, and collect performance/resource metrics
291 within our framework, yielding an effective HLS code dataset. Our dataset has the following merits:

292 **Diverse Application Coverage.** Table 1 shows that HLStrans provides the largest number of HLS
293 kernels and the longest average lines of code, incorporating commonly used HLS benchmarks as
294 well as real-world examples. Our curated corpus spans diverse application domains and covers all
295 six transformation categories listed in Appendix A.1. Figure 4a visualizes the program distribution
296 across these five tasks, and the source kernels themselves fall into seven distinct application cate-
297 gories. This rich, well-balanced dataset offers broad coverage of real-world HLS patterns required
298 to train and evaluate LLMs’ hardware-synthesis capabilities.

299 **Diverse types of transformations.** To evaluate the LLM’s ability to assess different C/C++ to HLS
300 transformations, every transformation shown in Figure 1 must be supported. Each dataset sample
301 may correspond to one or more types of transformations. Figure 4b illustrates how the number of
302 samples for each transformation increases after data augmentation.



313 Figure 4: (a) Program source distribution. (b) Percentage of different transformations. (c) Speedup
314 percentiles across dataset.
315

316 **High quality of dataset samples.** To evaluate the quality of the dataset, we measured the execution-
317 cycle ratio between the original and target codes using reports from Vitis HLS. The speedup is ratio
318 between the latency of the original design and the generated design, as reported by the synthesis
319 tool. We then computed the percentile distribution of these speedup values across all pairs. As
320 shown in the Figure 4c, 100% of the pairs, the target code is $\geq 1.5\times$ faster, and for 25% of the
321 pairs, it achieves a speedup of $\geq 50.3\times$. Different samples are annotated with performance and
322 resource usage tags, allowing the LLM to understand the detailed effects of C-to-HLS transforma-
323 tions. This enables the LLM to generate code that achieves higher performance while consuming
fewer resources. The detailed information on dataset generation is in Appendix A.2. The datasets

324 are released under the MG0-2.0 Non-Commercial (NC) license (Duan et al., 2024).² These licenses
 325 permit both academic and commercial reuse provided that attribution is given. The dataset release
 326 also includes provenance metadata and third-party license notices.
 327

328 4 EXPERIMENTAL EVALUATION OF LLMs ON OUR DATASET

330 To evaluate LLM performance on our dataset and assess the dataset’s impact on model capability,
 331 we explore different prompting strategies and fine-tune smaller models using supervised fine-tuning
 332 (SFT) (Ouyang et al., 2022).
 333

334 4.1 PROMPTING METHODS

336 **Zero-shot Prompting:** We craft concise, HLS-specific prompts that instruct the model to perform
 337 code optimization or transformation from its pretrained knowledge, without any additional fine-
 338 tuning or example demonstrations (Liu et al., 2021) (Wei et al., 2021). **Chain-of-Thought Prompt-
 339 ing:** Building on the chain-of-thought approach of Wei et al. (2022), our prompts first guide the
 340 model through a transformation reasoning phase before asking it to emit the refined code. **Retrieval-
 341 Based Prompting:** Recent studies (Shrivastava et al., 2023) (Shypula et al., 2023) have shown that
 342 retrieval-based techniques can substantially boost code generation quality in large language models.
 343 In our approach, we first encode each program using CodeBERT (Zhou et al., 2023) to produce rich,
 344 semantically informed embeddings. We then index these vectors with FAISS (Johnson et al., 2019)
 345 (Facebook AI Similarity Search) and perform a K-nearest-neighbors lookup to retrieve the top K
 346 most similar code snippets from our training corpus. Finally, we supply these retrieved examples
 347 alongside the original code as additional context to the LLM, guiding it to produce more accurate
 348 and effective edits. In our experiments, we set K to 1. The detailed prompt information is in Ap-
 349 pendix A.3. The following experiments includes results of Vitis_HLS tools. [Results of other HLS
 350 tools are shown in Appendix A.6.](#)

351 4.2 EXPERIMENT SETTING

353 To evaluate LLMs on our dataset, we have the following evaluation setting. **Task setup:** Given a
 354 C/C++ kernel, the model must generate an optimized HLS implementation. Success requires not
 355 only functional correctness but also synthesizability under FPGA toolchains. **Models:** We bench-
 356 mark both closed-source (GPT-5 (Wang et al., 2025), DeepSeek-R1 (Chua & Evans, 2025), Grok 4
 357 (xAI, 2025), Gemini 2.5 Pro (Comanici et al., 2025)) and open-source (Qwen 2.5 Coder (Hui et al.,
 358 2024)) models, under different prompting strategies (zero-shot, chain-of-thought, retrieval) and fine-
 359 tuning. **Dataset split:** Following standard machine-learning protocol, we reserve 270 applications
 360 for training and validation and hold out 39 applications for evaluation. The held-out set includes
 361 both unsynthesizable designs that require repair and synthesizable designs that require optimization.
 362 Crucially, these 39 held-out applications were excluded from the LLM-based data-augmentation
 363 pipeline to prevent any risk of data leakage into the evaluation. **Infrastructure:** All synthesis is
 364 conducted with the Xilinx Vitis HLS toolchain targeting a datacenter FPGA (Alveo U55C). Training
 365 was conducted using 2 NVIDIA H100 GPUs, each with 80 GB of memory. The computing
 366 environment was configured with CUDA 12.2 and cuDNN 9.1 to ensure optimal deep learning
 367 performance. **Metrics:** Unlike conventional code generation benchmarks that stop at functional
 368 correctness, the C-to-HLS task requires models to satisfy both software and hardware constraints.
 369 We therefore report four complementary metrics:

- 370 • Functional Accuracy: The share of test programs that preserve the original functionality test-
 371 bench.
- 372 • Synthesis Accuracy: Percentage of programs that compile successfully into FPGA-ready hard-
 373 ware.
- 374 • Speedup (Latency reduction): Ratio between the latency of the original design and the generated
 375 design, as reported by the synthesis tool.
- 376 • Optimization Rate (%OPT): Fraction of generated programs that both pass correctness checks
 377 and achieve speedup $> 1\times$.

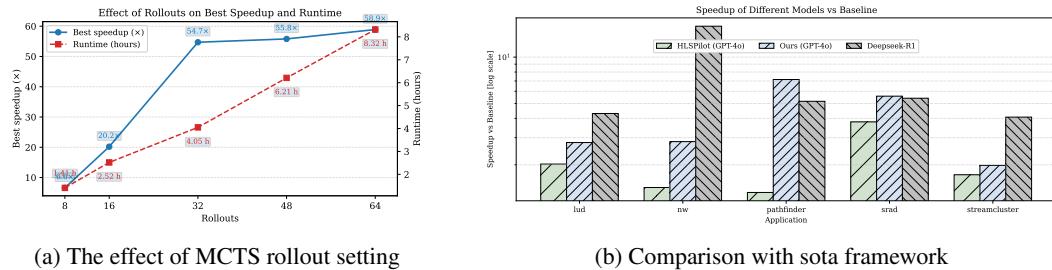
²<https://www.modelgo.li/>.

378 Previous works (Li et al., 2022) show that generating multiple program candidates per input and
 379 selecting the optimal one improves code synthesis performance. We generate k program variants
 380 for each input, then select the fastest one that successfully passes all test cases; we refer to this
 381 sampling-and-selection strategy as *Best@k*.
 382

383 4.3 EXPERIMENT RESULTS

385 4.3.1 EVALUATION OF DATA AUGMENTATION FRAMEWORK.

386 The MCTS component of our framework can produce variable iteration lengths. To quantify this
 387 behavior, we evaluated both runtime and achieved speedup on the PolyBench suite (Pouchet & Yuki,
 388 2012) while sweeping the number of rollouts. Figure 5a reports these results and indicates that 32
 389 rollouts is the “sweet spot”; consequently, we set the rollout count to 32 for subsequent experiments.
 390 We also compared our framework against state-of-the-art approaches on the Rodinia benchmarks
 391 by measuring the runtime of the optimized programs on real FPGA cards. Figure 5b shows the
 392 runtime comparison for five Rodinia benchmarks (Che et al., 2009). With the same base model,
 393 our framework attains more than 5 \times average speedups than Xiong et al. (2024). We additionally
 394 observed that DeepSeek-R1 produces even better results; therefore, we selected DeepSeek-R1 as the
 395 generator for new data used in later experiments. These findings motivated both our choice of rollout
 396 parameter and our selection of the data-generation model. The detailed experiments comparison
 397 results are in Appendix A.2. The detailed experiments analysis are in Appendix A.9



400 Figure 5: Evaluation of our dataset augmentation framework (a) MCTS rollout setting. (b) Rollout
 401 comparison.

402 4.3.2 RESULTS OF FINE-TUNING MODELS

403 Both the Qwen2.5 Coder 3B and 7B fine-tuned models show consistent gains in optimization quality,
 404 latency reduction, and synthesis success rate in Table 2. They generate HLS code that not only exe-
 405 cutes faster but also synthesizes more reliably, even though the function-correct rate has slightly
 406 dropped. These results demonstrate that training on our curated dataset significantly boosts an
 407 LLM’s ability to produce correct, high-performance HLS implementations directly from C sources.

408 Table 2: Fine-Tuning results comparison. *Transformation*: the T1–T5 transformation in Figure 1
 409 applied to examples that are functionally and synthesis correct.

410 Method	411 Model	412 Speedup				413 Transformation					414 Functional 415 Accuracy	416 Synthesis 417 Accuracy
		417 Opt	418 Min	419 Avg	420 Max	421 T1	422 T2	423 T3	424 T4	425 T5		
422 Pretrain	Qwen coder 7B	2.6%	0.27 \times	1.03 \times	3.6 \times	0	5.1%	0	0	2.6%	12.8%	10.3%
	Qwen coder 3B	0%	0.38 \times	0.97 \times	1 \times	0	2.6%	0	0	2.6%	7.7%	10.3%
423 SFT	Qwen coder 7B	15.4%	0.6\times	4.2\times	21.8\times	5.1%	20.5%	5.1%	5.1%	2.6%	20.5%	28.2%
	Qwen coder 3B	10.3%	0.4 \times	3.7 \times	17.2 \times	5.1%	17.9%	2.6%	5.1%	2.6%	17.9%	20.5%

426 Efficient HLS kernels require a mix of C to HLS transformations. We measure how our dataset
 427 improves LLM C to HLS optimization: Table 2 demonstrates that fine-tuning on our corpus raises
 428 success rates across transformation types.

429 4.3.3 RESULTS OF PRETRAINED MODELS

430 Table 3 reports the *Best@1* and *Best@5* accuracies for different prompts and models. To evaluate the
 431 utility of our dataset, we constructed retrieval databases from the HLSdataset (Wei et al., 2023) and

432 from our training data, and applied retrieval-based prompting using these two databases to measure
 433 its effect. Overall, incorporating our dataset into retrieval improved pretrained models performance
 434 compared with other prompt methods.

436 Table 3: Best@1 and Best@5 results for various methods and models.
 437

438	439	440	Method	Model	Best@1					Best@5						
					Speedup				Speedup				Functional			
					Opt	Min	Avg	Max	Opt	Min	Avg	Max	Accuracy	Synthesis Accuracy		
441	442	443	Zero-shot	Deepseek-R1	20.5%	0.17 \times	1.82 \times	16.03 \times	43.6%	38.5%	23.1%	0.19 \times	1.97 \times	16.15 \times	46.2%	51.3%
				GPT-5	20.5%	0.04 \times	14.32 \times	506.07 \times	48.7%	48.7%	0.34 \times	14.35 \times	506.07 \times	53.8%	61.5%	
				Grok-4	20.5%	0.48 \times	2.35 \times	46.51 \times	43.6%	43.6%	33.3%	0.50 \times	2.46 \times	46.84 \times	56.4%	53.8%
				Gemini-2.5-pro	25.6%	0.98 \times	2.74 \times	35.57 \times	41.0%	41.0%	30.8%	1.21 \times	2.89 \times	36.01 \times	46.2%	51.3%
				Qwen coder 32B	10.3%	0.28 \times	1.10 \times	3.71 \times	56.4%	53.8%	17.9%	0.43 \times	1.22 \times	4.09 \times	59.0%	56.4%
444	445	446	COT	Deepseek-R1	25.6%	0.21 \times	2.1 \times	19.01 \times	48.7%	46.2%	28.2%	0.34 \times	2.18 \times	19.07 \times	51.3%	53.8%
				GPT-5	25.6%	0.19 \times	17.1 \times	425.07 \times	53.8%	53.8%	38.5%	0.41 \times	17.12 \times	437.07 \times	56.4%	66.7%
				Grok-4	20.5%	0.53 \times	2.56 \times	49.77 \times	48.7%	51.3%	33.3%	0.82 \times	2.92 \times	50.12 \times	61.5%	53.8%
				Gemini-2.5-pro	30.8%	0.98 \times	2.98 \times	37.57 \times	46.2%	46.2%	41.0%	1.28 \times	3.39 \times	37.58 \times	51.3%	56.4%
				Qwen coder 32B	15.4%	0.37 \times	1.910 \times	5.87 \times	61.5%	59.0%	20.5%	0.52 \times	2.03 \times	6.25 \times	71.8%	66.7%
447	448	449	Retrieval Prompt (HLSdataset)	Deepseek-R1	20.5%	0.47 \times	30.10 \times	953.30\times	33.3%	28.2%	23.1%	0.60\times	30.28 \times	954.41 \times	35.9%	35.9%
				GPT-5	18.0%	0.01 \times	1.96 \times	31.60 \times	33.3%	28.2%	30.8%	0.23 \times	2.04 \times	33.86 \times	35.9%	41.0%
				Grok-4	12.8%	0.07 \times	1.59 \times	19.19 \times	33.3%	25.6%	25.6%	0.36 \times	2.32 \times	26.23 \times	46.2%	28.2%
				Gemini-2.5-pro	18.0%	0.02 \times	5.57 \times	137.19 \times	33.3%	28.2%	28.2%	0.32 \times	6.39 \times	137.35 \times	38.5%	38.5%
				Qwen coder 32B	10.3%	0.33 \times	2.67 \times	65.31 \times	35.9%	30.8%	15.4%	0.48 \times	2.92 \times	72.97 \times	46.2%	38.5%
450	451	452	Retrieval Prompt (HLStrans)	Deepseek-R1	25.6%	0.21 \times	2.10 \times	19.01 \times	48.7%	46.2%	33.3%	0.41 \times	2.19 \times	20.40 \times	53.8%	56.4%
				GPT-5	25.6%	0.19 \times	37.10\times	425.07 \times	53.8%	53.8%	46.2%	0.46 \times	37.10\times	962.12\times	66.7%	71.8%
				Grok-4	20.5%	0.53 \times	2.56 \times	49.77 \times	64.1%	51.3%	30.8%	0.57 \times	2.84 \times	51.85 \times	51.3%	56.4%
				Gemini-2.5-pro	33.3%	0.98\times	2.98 \times	37.57 \times	46.2%	46.2%	33.3%	1.26 \times	3.35 \times	38.62 \times	51.3%	59.0%
				Qwen coder 32B	15.4%	0.37 \times	1.910 \times	5.87 \times	61.5%	59.0%	20.5%	0.63 \times	2.12 \times	6.32 \times	64.1%	64.1%

453 4.4 RESULTS ANALYSIS

454 **Observation 1. Retrieval-augmented generation and finetuning on HLStrans can improve**
 455 **model’s performance on C to HLStrans task.** This demonstrates that our dataset by providing a rich
 456 cache of validated pragmas and code transformation examples serves as an indispensable “best prac-
 457 tices” repository, steering LLMs toward hardware-friendly idioms and dramatically reducing syn-
 458 thesis failures.

459 **Observation 2. Sampling diversity substantially boosts results.** Allowing up to five candidate
 460 generations per input (Best@5) improves both synthesis success rate and achieved acceleration. For
 461 example, GPT-5 (retrieval prompt with HLStrans) raises synthesis accuracy from 53.8% to 71.8%
 462 under Best@5, underscoring the benefit of n-best generation.

463 **Observation 3. LLM optimization may harm HLS code performance.** We observe that some
 464 LLM-optimized kernels actually degrade performance (speedup < 1 \times). This can happen for two
 465 reasons: First, the restructuring performed by the LLM can introduce new loop dependencies, in-
 466 creasing latency; Second, the pragmas inserted by LLMs may be less effective than the default
 467 optimizations inferred by the HLS compiler. Therefore, it is necessary to set up dataset to guide
 468 LLM’s proper optimizations.

469 **Observation 4. Trade-off between pass rate and optimizations.** Applying retrieved, optimized
 470 code examples increases performance but reduces both functional and synthesis accuracy. For ex-
 471 ample, Deepseek-R1 (retrieval prompt with HLSdataset) increase speedup but decrease functional
 472 accuracy from 43.6% to 33.3% compared with zero-shot prompt. This highlights a trade-off between
 473 aggressive optimization and correctness when the LLM’s capability is unchanged.

474 **Observation 5. LLMs perform differently across transformations.** As shown in Figure 1, pre-
 475 trained models more easily apply T2 and T5. Fine-tuning on HLStrans improves the success rates
 476 of all the transformations, as reported in Table 2.

477 5 CONCLUSION

478 We introduce a novel dataset that transforms C or C++ kernels into richly annotated HLS imple-
 479 mentations, empowering LLMs to learn hardware-aware optimizations such as loop pipelining,
 480 unrolling, and memory buffering. Our experiments demonstrate that retrieval and fine-tuning on this
 481 dataset significantly boosts both latency reduction and synthesis success rates, proving its effective-
 482 ness in accelerating and automating electronic design flows. By releasing the dataset and training
 483 scripts, we aim to catalyze further exploration at the intersection of LLMs and hardware design.

486 ETHICS STATEMENT
487488 We release a dataset that converts C/C++ kernels into richly annotated HLS implementations, to-
489 gether with training scripts, to accelerate LLM-driven hardware optimizations. While retrieval and
490 fine-tuning improve latency and synthesis success, automated optimizations can produce incorrect or
491 biased transformations; therefore the dataset and models are for research-only use and not intended
492 for safety-critical deployment. Users should apply human review, evaluate functional correctness
493 and synthesis safety alongside performance gains, and publish datasheets/model cards to promote
494 transparency. Continued work on verification, robustness, and responsible reporting of failure cases
495 is strongly encouraged.496
497 REPRODUCIBILITY STATEMENT
498500 We are committed to ensuring the reproducibility of our findings. All datasets, code, and
501 experimental scripts are publicly available at [https://anonymous.4open.science/r/](https://anonymous.4open.science/r/HLStrans-B578/)
502 HLStrans-B578/.503 LLM USAGE DECLARATION
504505 We used Gemini 2.5 Pro³ to polish grammar and phrasing during the writing process. No part of the
506 analysis, experimental design, or results was generated by a large language model.508 REFERENCES
509510 Stefan Abi-Karam, Rishov Sarkar, Allison Seigler, Sean Lowe, Zhigang Wei, Hanqiu Chen, Nan-
511 ditha Rao, Lizy John, Aman Arora, and Cong Hao. HLSfactory: A framework empowering high-
512 level synthesis datasets for machine learning and beyond. In *Proceedings of the 2024 ACM/IEEE*
513 *International Symposium on Machine Learning for CAD*, pp. 1–9, 2024.514 Yunsheng Bai, Atefeh Sohrabizadeh, Zongyue Qin, Ziniu Hu, Yizhou Sun, and Jason Cong. Towards
515 a comprehensive benchmark for high-level synthesis targeted to FPGAs. *Advances in Neural*
516 *Information Processing Systems*, 36:45288–45299, 2023.518 Sutirtha Bhattacharyya, BG Sutharshan, and Chandan Karfa. LLM vs HLS for RTL code genera-
519 tion: Friend or foe? In *2024 IEEE 33rd Asian Test Symposium (ATS)*, pp. 1–6. IEEE, 2024.520 Julian Blank and Kalyanmoy Deb. Pymoo: Multi-objective optimization in python. *IEEE Access*,
521 8:89497–89509, 2020.523 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee,
524 and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009*
525 *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009. doi:
526 10.1109/IISWC.2009.5306797.527 James Chua and Owain Evans. Are deepseek r1 and other reasoning models more faithful? *arXiv*,
528 abs/2501.08156, 2025.529 Luca Collini, Siddharth Garg, and Ramesh Karri. C2HLSc: Can LLMs bridge the software-to-
530 hardware design gap? In *2024 IEEE LLM Aided Design Workshop (LAD)*, pp. 1–12. IEEE, 2024.532 Gheorghe Comanici, Eric Bieber, Mike Schaeckermann, Ice Pasupat, Noveen Sachdeva, Inderjit
533 Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the
534 frontier with advanced reasoning, multimodality, long context, and next generation agentic capa-
535 bilities. *arXiv preprint arXiv:2507.06261*, 2025.536 Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru
537 Zhang. FPGA HLS today: successes, challenges, and opportunities. *ACM Transactions on Re-
538 configurable Technology and Systems (TRETS)*, 15(4):1–42, 2022.539
3<https://deepmind.google/models/gemini/pro/>

540 Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline FY Young, and Zhiru Zhang. Fast
 541 and accurate estimation of quality of results in high-level synthesis with machine learning. In
 542 *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing
 543 Machines (FCCM)*, pp. 129–132. IEEE, 2018.

544 Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multi-
 545 objective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, 6(2):
 546 182–197, 2002.

547 Moming Duan, Qinbin Li, and Bingsheng He. ModelGo: A practical tool for machine learning
 548 license analysis. In *Proceedings of the ACM Web Conference 2024*, pp. 1158–1169, 2024. doi:
 549 10.1145/3589334.3645520.

550 Aggelos Ferikoglou, Andreas Kakolyris, Vasilis Kypriotis, Dimosthenis Masouros, Dimitrios
 551 Soudris, and Sotirios Xydis. CollectiveHLS: Ultrafast knowledge-based HLS design optimiza-
 552 tion. *IEEE Embedded Systems Letters*, 16(2):235–238, 2023.

553 Lorenzo Ferretti, Jihye Kwon, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca Carloni, and Laura
 554 Pozzi. Db4hls: a database of high-level synthesis design space explorations. *IEEE Embedded
 555 Systems Letters*, 13(4):194–197, 2021.

556 Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS:
 557 Neural Information Processing Systems Conference On-line trading of Exploration and Exploita-
 558 tion Workshop*, 2006.

559 Pingakshya Goswami, Masoud Shahshahani, and Dinesh Bhatia. Mlsbench: A benchmark set for
 560 machine learning based FPGA HLS design flows. In *2022 IEEE 13th Latin America Symposium
 561 on Circuits and System (LASCAS)*, pp. 1–4. IEEE, 2022.

562 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu,
 563 Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in
 564 LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

565 Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. Chstone: A
 566 benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International
 567 Symposium on Circuits and Systems (ISCAS)*, pp. 1192–1195. IEEE, 2008.

568 Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang,
 569 Bowen Yu, Keming Lu, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng,
 570 Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou,
 571 and Junyang Lin. Qwen2.5-Coder technical report. *arXiv*, abs/2409.12186, 2024.

572 Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE
 573 Transactions on Big Data*, 7(3):535–547, 2019.

574 Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. Parallel programming for FPGAs. *arXiv
 575 preprint arXiv:1805.03648*, 2018.

576 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom
 577 Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation
 578 with alphacode. *Science*, 378(6624):1092–1097, 2022.

579 Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What
 580 makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.

581 Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong
 582 Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kel-
 583 ton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike,
 584 and Ryan Lowe. Training language models to follow instructions with human feedback. *arXiv
 585 preprint arXiv:2203.02155*, 2022. URL <https://arxiv.org/abs/2203.02155>.

586 Louis-Noël Pouchet and Tomofumi Yuki. The polyhedral benchmark suite. *On-line: http://www.
 587 cse.ohiostate.edu/pouchet/software/polybench*, 2012.

594 Neha Prakriya, Zijian Ding, Yizhou Sun, and Jason Cong. LIFT: LLM-based pragma insertion for
 595 HLS via GNN supervised fine-tuning. *arXiv preprint arXiv:2504.21187*, 2025.
 596

597 Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite:
 598 Benchmarks for accelerator design and customized architectures. In *2014 IEEE International
 599 Symposium on Workload Characterization (IISWC)*, pp. 110–119. IEEE, 2014.
 600

601 Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for
 602 large language models of code. In *International Conference on Machine Learning*, pp. 31693–
 603 31715. PMLR, 2023.
 604

605 Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Gra-
 606 ham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning
 607 performance-improving code edits. *arXiv preprint arXiv:2302.07867*, 2023.
 608

609 S. Wang et al. Capabilities of GPT-5 on multimodal medical reasoning. *arXiv*, abs/2508.08224,
 610 2025.
 611

612 Andy Wanna, Hanqiu Chen, and Cong Hao. Forgebench: A machine learning benchmark suite
 613 and auto-generation framework for next-generation HLS tools. *arXiv preprint arXiv:2504.15185*,
 614 2025.
 615

616 Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du,
 617 Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint
 618 arXiv:2109.01652*, 2021.
 619

620 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny
 621 Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in
 622 neural information processing systems*, 35:24824–24837, 2022.
 623

624 Zhigang Wei, Aman Arora, Ruihao Li, and Lizy John. HLSdataset: Open-source dataset for ML-
 625 assisted FPGA design using high level synthesis. In *2023 IEEE 34th International Conference on
 626 Application-specific Systems, Architectures and Processors (ASAP)*, pp. 197–204. IEEE, 2023.
 627

628 xAI. Grok 4 model card. <https://data.x.ai/2025-08-20-grok-4-model-card.pdf>, 2025. Accessed: YYYY-MM-DD.
 629

630 Chenwei Xiong, Cheng Liu, Huawei Li, and Xiaowei Li. Hlspilot: Llm-based high-level synthesis.
 631 In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp.
 632 1–9, 2024.
 633

634 Kangwei Xu, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, Ulf Schlichtmann, and Bing Li. Auto-
 635 mated c/c++ program repair for high-level synthesis via large language models. In *Proceedings
 636 of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, pp. 1–9, 2024.
 637

638 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao.
 639 React: Synergizing reasoning and acting in language models. In *International Conference on
 640 Learning Representations (ICLR)*, 2023.
 641

642 Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code
 643 generation with pretrained models of code. *arXiv preprint arXiv:2302.05527*, 2023.
 644

645 Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Feath-
 646 erston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, et al. Rosetta: A realistic high-
 647 level synthesis benchmark suite for software programmable FPGAs. In *Proceedings of the 2018
 648 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 269–278, 2018.

648 **A APPENDIX**
649650 **A.1 HLS CODE TRANSFORMATIONS**
651652 **A.1.1 HLS CODE OPTIMIZATION: CODE RESTRUCTURING.**
653654 In our datasets, we apply a suite of code-reconstruction techniques designed to optimize memory ac-
655 cess patterns, alleviate computational bottlenecks, and resolve loop dependencies. By restructuring
656 data flow and exploiting hardware parallelism, these methods boost throughput and shorten overall
657 execution time. Table 4 list the main Code Restructuring adopted in our dataset.
658659 **Table 4: HLS Code Reconstruction Methods**

Optimization	Explanation	Performance Benefit
Memory coalescing	Merge multiple memory accesses into one memory transaction.	Reduces memory access latency and improves bandwidth utilization.
Local tiling	Divide loops into tiles to improve cache reuse and spatial locality.	Enhances data locality and on-chip buffer efficiency.
Ping pong buffer	Alternate between two buffers for simultaneous load and compute.	Hides memory latency by overlapping computation with memory access.
Dataflow	Separate tasks into pipeline stages for concurrent execution.	Allows function-level parallelism, boosting throughput.
Control flow optimization	Replace if-else with ternary or simplified logic conditions.	Reduces combinational path length, improving timing and synthesis.

660 **A.1.2 HLS CODE OPTIMIZATION: HLS DIRECTIVE (PRAGMA) INSERTION.**
661662 Our dataset features an extensive catalog of HLS pragmas ranging from memory-access directives
663 (array partitioning, streaming) through loop-level transformations (unrolling, merging, tiling) to
664 fine-grained pipeline controls (initiation interval tuning, dataflow regions). By systematically ap-
665 plying and combining these pragmas, these directives empower automated HLS flows to tailor
666 synthesized hardware for domain-specific latency, throughput, and area requirements making our
667 dataset a valuable reference for exploring pragma-driven performance tuning. Table 5 introduces
668 these applied pragma optimizations.
669670 **Table 5: HLS Directive (Pragma) Insertion Methods**
671

Optimization	Explanation	Pragma Example
Array partition	Split a large array into multiple smaller memories	<code>#pragma HLS ARRAY_PARTITION variable=arr complete</code>
Memory type	Specify the on-chip storage type (BRAM/URAM/SMALL_RAM)	<code>#pragma HLS RESOURCE variable=buf core=RAM_2P</code>
Loop unroll	Replicate loop body to create parallel compute units	<code>#pragma HLS UNROLL factor=4</code>
Loop merge	Merge consecutive loops to reduce control overhead	<code>#pragma HLS LOOP_MERGE</code>
Function inline	Inline functions to eliminate call overhead	<code>#pragma HLS INLINE</code>
Pipeline	Pipeline loops or functions to lower initiation interval	<code>#pragma HLS PIPELINE II=1</code>
Dataflow	Enable task-level parallelism between functions	<code>#pragma HLS DATAFLOW</code>
Dependence	Declare data dependencies to allow safe loop optimizations	<code>#pragma HLS DEPENDENCE variable=arr inter false</code>
Stream	Use streaming interfaces to transfer data via FIFOs	<code>#pragma HLS STREAM variable=fifo depth=8</code>

694 **A.1.3 HLS CODE OPTIMIZATION: DATA-TYPE ADAPTATION.**
695696 Our dataset also incorporates a comprehensive suite of data-type adaptations optimized for FPGA
697 synthesis. We translate generic C types (e.g., `int`, `float`, `struct`) into precise HLS constructs
698 such as `ap_uint<W>`, `ap_fixed<TOTAL, INT>`, and `hls::stream<T>` to fully exploit on-
699 chip LUT/FF and DSP resources. These mappings enable fine-grained control over bit-width, data
700 alignment, and streaming interfaces, ensuring maximal throughput, minimal logic utilization, and
701 lower power consumption in FPGA deployments. Table 6 lists the specific datatype conversions
702 applied in our framework.

702
703
704 Table 6: Adaptation of C Data Types to HLS Data Types
705
706
707
708
709
710
711

Original C Type	HLS Type	Purpose
int/short/char	ap_uint<W>/ ap_int<W>	Precisely control integer bit-width to save LUT/FF resources
float/double	ap_fixed<TOTAL, INT>/ ap_ufixed<TOTAL, INT>	Replace floating-point with fixed-point to reduce DSP usage and power
struct/union	struct { ap_uint<...> field; } with bitfields	Precisely specify field bit-widths and alignment, eliminate padding
pointer/array	hls::stream<T>	Map to hardware FIFO streams for streaming transmission

712
713
714 A.1.4 HLS CODE OPTIMIZATION: TRANSFORMATION OF FUNCTIONS
715716 By transforming standard C/C++ functions into their corresponding HLS intrinsics, developers can
717 leverage highly optimized FPGA kernels. This approach dramatically boosts execution performance
718 by exploiting dedicated hardware units for math operations and data manipulation. At the same
719 time, it conserves FPGA resources, reducing logic utilization and power consumption compared to
720 generic software approximations. Table 7 lists the transformations of standard math functions to
721 HLS intrinsics.
722723
724 Table 7: Transformation of Standard Functions to HLS Intrinsics
725

Standard C/C++ Function	HLS Intrinsic	Purpose
std::sqrt (x)	hls::sqrt (x)	Generates a pipelined square root unit instead of slow software approximation
std::exp (x)	hls::exp (x)	Synthesizes an exponential function hardware block (LUT-based)
std::log (x)	hls::log (x)	Provides a hardware-friendly implementation of natural logarithm
std::sin (x)	hls::sin (x)	Efficient sine computation using CORDIC or LUTs
std::cos (x)	hls::cos (x)	Efficient cosine computation using CORDIC or LUTs
a / b	hls::div(a, b)	Replaces division with a synthesizable divider core
a % b	hls::mod(a, b)	Synthesizes modulo operation in hardware

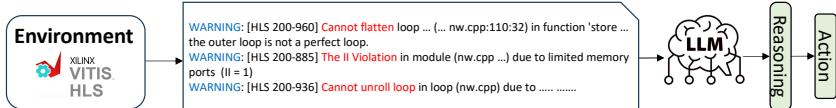
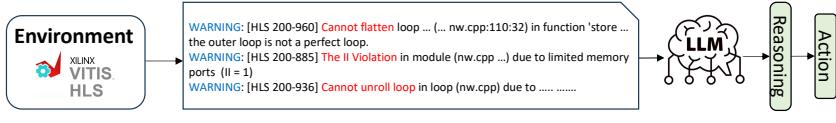
736
737
738 A.1.5 HLS CODE REPAIR: HLS-COMPLIANT CODING STYLE.
739740 High-level synthesis (HLS) cannot synthesize all idiomatic C constructs directly. To enable hard-
741 ware generation, we must refactor unsupported patterns like dynamic memory allocation, recursion,
742 and pointer arithmetic into HLS-compliant coding styles that the tool can analyze and map to on-chip
743 resources. Table 8 lists these common transformations.
744745
746 Table 8: Transformation of Unsupported C Constructs for HLS Compatibility
747

Unsupported C Construct	Recommended HLS-Compatible Transformation	Purpose
Dynamic memory allocations	Use static arrays with fixed size at compile time	HLS tools require compile-time memory size to synthesize physical resources (BRAM/LUTRAM)
Recursion	Convert to iterative form using for/while loops	Recursion creates a dynamic call stack, which is not synthesizable
Pointer arithmetic beyond array indexing	Use bounded array indexing	Allows compiler to infer memory access patterns and pipeline-optimize
Function pointers or callbacks	Inline or manually instantiate function variants	HLS requires all control flow to be static and analyzable at compile time
Variable-length arrays	Replace with fixed-size arrays defined by constants or macros	HLS cannot synthesize dynamically sized buffers

756 A.2 DATASET AUGMENTATION
757758 A.2.1 MCTS FRAMEWORK
759760 MCTS enables an agent to learn to navigate the vast space of possible code transformations while
761 balancing multiple optimization objectives. The agent’s decisions are guided by comprehensive
762 feedback from the HLS toolchain, including synthesis success, resource utilization, and performance
763 metrics. Our MCTS has the following elements:764 **Environment E:** The environment is composed of the HLS toolchain, specifically Xilinx Vitis HLS,
765 which compiles the code and provides critical feedback such as performance estimates.766 **Agent G:** We propose to use an LLM as the agent that leverages its pretrained knowledge of hardware
767 design and in-context learning abilities.768 **Action A:** At each time step t , the agent selects an action a_t , which corresponds to a prompt or trans-
769 formation applied to the current HLS code. We define two complementary action types: RAG-based
770 actions retrieve optimization policies directly from our pre-built table and accompanying code ex-
771 amples shown in Figure 6, leveraging retrieval-augmented generation to surface proven trans-
772 formations rapidly and reliably. Reasoning-based actions with ReAct prompt (Yao et al., 2023), in
773 contrast, analyze compiler warnings such as pipeline-interval breaches or loop-unroll violations and
774 apply targeted code reforms by interpreting warning semantics within the current code context.

775

Strategies: Loop Tiling
Introduction: partitions large loops into smaller tiles to enhance data locality and cache reuse.
Examples:
Baseline: `for (int i=0; i<N; i+=1) { for (int j=0; j<N; j+=1) { C[i][j] = A[i][j] + B[i][j]; } }`
Optimized: `for (int jj = 0; jj < N; jj += TILE_SIZE) { int localC[TILE_SIZE][TILE_SIZE]; for (int kk = 0; kk < N; kk += TILE_SIZE) { int localA[TILE_SIZE][TILE_SIZE]; int localB[TILE_SIZE][TILE_SIZE]; } }`

776 (a) Loop tiling Code examples by RAG
777778 (b) LLM reasoning about environment warning and tool hint
779780 Figure 6: Actions design of MCTS
781782 **State S:** The state S_t at time step t is defined as the current version of the HLS code after applying
783 the previous actions.784 **Reward R:** Rule-based reward shaping has proven effective in guiding agent behavior in previous
785 work Guo et al. (2025). In our framework, the reward function $R(s_t, a_t)$ is computed by applying
786 rule-based scoring to verification results and feedback provided by the HLS tool. A penalty of
787 -2 is applied when the verification fails, and -1 if synthesis fails or the design exceeds resource
788 constraints. A neutral reward of 0 is given when the transformed design performs worse than the
789 original, while a reward of 1 is granted when it performs better. If the design not only surpasses the
790 original but also meets timing constraints, a higher reward of 2 is assigned.791 In our cases, the MCTS begins at the initial state S_0 (the root node), which is the naive HLS code.
792 From a state S_t , the agent applies an optimization policy π , i.e., an action $a_t \in \mathcal{A}$, transitioning to
793 the subsequent state S_{t+1} . MCTS consists of four key phases: selection, expansion, simulation, and
794 backpropagation. $N(S_t)$, the total number of times S_t has been visited.795 **Selection:** We employ the upper confidence bounds for trees (UCT) (Gelly & Wang, 2006) algorithm
796 to choose nodes.

807
$$\pi(s_t) = \arg \max_{a_t \in \mathcal{A}} \left(\underbrace{R(s_t, a_t)}_{\text{reward}} + \beta \times \underbrace{\frac{\sqrt{1 + N(s_t)}}{1 + N(s_t, a_t)}}_{U \text{ Term}} \right).$$

808
809

810 **Expansion:** From the current state s_t , generate one or more child nodes to explore untried actions.
 811
 812 **Simulation:** Perform a rollout from the chosen child node by applying a_{t+1} , running HLS synthesis
 813 to estimate latency and resource utilization, and computing the reward $R(s_t, a_{t+1})$.
 814 **Backpropagation:** Propagate the obtained reward back up the visited path, updating each node's
 815 statistics (e.g., visit count and value estimate) to improve future selection.
 816
 817 **Retrieval-Augmented Generation:** To broaden the range of HLS code-transformation techniques
 818 that our LLM can learn, we built an automated framework (see Appendix A.2) that programmatically
 819 generates optimized variants via Monte Carlo Tree Search. Central to this system is a Retrieval-
 820 Augmented Generation (RAG) table of optimization strategies including code-reconstruction pat-
 821 terns, directive (pragma) insertions, data-type adaptations, and function-level transformations each
 822 entry pairing a concise description with a few-shot example illustrating the baseline code and its
 823 optimized counterpart listed in A.1. During search, these RAG-driven actions guide the MCTS
 824 policy to apply specific transformations, yielding a diverse corpus of HLS kernels ready for LLM
 825 fine-tuning and evaluation. One kind of Retrieval-Augmented strategies is shown in Figure 7 and
 826 prompt template is shown in Figure 8.

827
 828 **Strategies: Loop Tiling**
 829 **Type:** Need to refactor the code
 830 **Introduction:** partitions large loops
 831 into smaller tiles to enhance
 832 data locality and cache reuse.
 833 **Examples:**
 834 **Baseline:** `for (int i=0; i<N; i+=1) {`
`for (int i=0; i<N; i+=1) { C[i][j] = A[i][j] + B[i][j]; }`
 835 **Optimized:** `for (int jj = 0; jj < N; jj += TILE_SIZE)`
`{`
 836 `int localC[TILE_SIZE][TILE_SIZE];`
 837 `for (int kk = 0; kk < N; kk += TILE_SIZE) {`
 838 `int localA[TILE_SIZE][TILE_SIZE];`
 839 `int localB[TILE_SIZE][TILE_SIZE];`
 840

Figure 7: Example of Retrieval-Augmented strategies in MCTS framework

841
 842 You are a FPGA engineer, You should obey Xilinx HLS code guidelines. The name of top_function is
 843 {function_name}, it can not be changed
 844 The code should have a header(h) file named {top_function}.h and a cpp file named
 845 {top_function}.cpp The defination of variables, constants and functions are only in header file.
 846 In cpp file, you should firstly give sub functions of code, the codes of top function should
 847 be at the end of cpp file.
 848 Your aim is to make sure the function of code is right and the pipeline interval from Xilinx
 849 HLS log to be one to achieve better performance.
 850 You should optimize the following HLS code using these strategies:
`\n\n" +`
`"\n".join(strategies)`

Figure 8: Prompt Template for the optimization with MCTS framework

854
 855 **Framework Evaluation:** We evaluated our dataset-augmentation framework on the widely adopted
 856 Rodinia benchmark suite (Che et al., 2009), using a Xilinx Alveo U55C FPGA board running at a
 857 300 MHz kernel clock. Our goal was to measure how effectively our MCTS-based sampler could
 858 guide Deepseek-R1 and GPT-4o toward highly optimized HLS kernels.
 859

860 Figure 9 shows the average success rate in the benchmarks. As the figure shows, their success rate is
 861 Qwen32B > Deepseek-R1 > GPT-4o > Qwen7B while Deepseek-R1 can achieve highest average
 862 speedup. The results show performance-increase will degrade the ability of LLM to produce the
 863 correct HLS code.

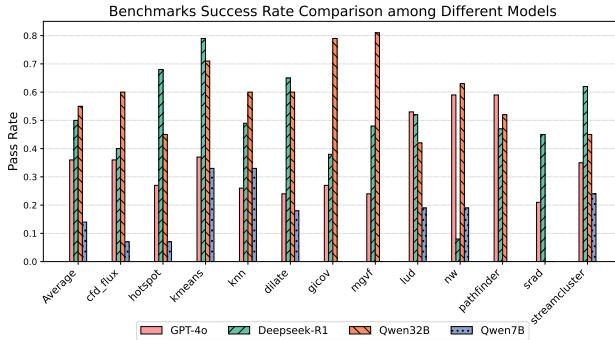


Figure 9: Success rate on different benchmarks with different models.

Table 9 summarizes the Best@1 kernel runtimes (in milliseconds) across twelve diverse applications, comparing four configurations: Baseline (The unmodified, compiler-generated HLS implementation), HLSPilot (Xiong et al., 2024) (A recent LLM based optimization framework), GPT-4o with our framework and Deepseek-R1 with our framework. Our results reveal several key findings:

- Consistent Improvement over previous work. In every benchmark, both of our enhanced pipelines outperform HLSPilot, demonstrating that the combination of large-model code generators with MCTS exploration yields more hardware-efficient HLS designs.
- Deepseek-R1 with our framework achieves up to average 28 \times reduction in real execution time compared to the baseline. GPT-4o with our framework attains up to average 20 \times reduction in real execution time.
- Robust Gains Across Diverse Kernels. From compute-bound codes such as kmeans, mgvf, and streamcluster, to memory-sensitive workloads like hotspot and nw, our framework consistently identifies and applies scheduling, pipelining, and data-partitioning transformations that exploit the parallelism and memory hierarchy of the Alveo U55C.

Table 9: Runtime (ms) of different benchmarks across models.

Application	Baseline	HLSPilot Xiong et al. (2024)		Ours	
		GPT-4o	Deepseek-R1	GPT-4o	Deepseek-R1
cfd.flux	13	6.71		4.57	1.61
hotspot	1879.1	712.7		300.5	22.3
kmeans	2243.2	65.9		17.9	15.7
knn	17.0	2.8		0.83	0.82
dilate	48.8	16		0.75	1.64
gcov	107.0	93		82.3	30.7
mgvf	8047.5	3212		1231	446
lud	226.4	112		81.2	52.6
nw	206.4	145		73	13
pathfinder	7.8	5.9		1.09	1.51
srad	35.7	9.4		6.4	6.6
streamcluster	16173	9388		8162.3	3966

A.2.2 DESIGN SPACE EXPLORATION.

The tool is an automated HLS design-space explorer that employs a genetic-algorithm optimizer to discover effective directive combinations—specifically loop pipelining, loop unrolling, and array partitioning—that maximize performance and resource efficiency. We traverse the search space with the NSGA-II algorithm (Deb et al., 2002) as implemented in the PyMOO library (Blank & Deb, 2020), chosen for its ability to escape local optima and rapidly converge to high-quality solutions. NSGA-II is executed for 24 generations with a population size of 40. Each generation performs three steps: (1) generate or initialize the population, (2) apply each candidate configuration to the source code using compiler B2 and synthesize with Xilinx Vitis, and (3) return the synthesis metrics

918 to NSGA-II. Configurations that exceed device resources or demand prohibitive HLS runtimes (e.g.,
 919 $\gtrsim 1$ hour) are deemed infeasible and discarded. Genetic operators are configured as follows: random
 920 sampling/selection (mutation sampling probability = 0.1), simulated binary crossover (probability
 921 = 1.0, $\eta = 15$), and polynomial mutation ($\eta = 20$); all other operator parameters use PyMOO
 922 defaults.

923
 924
 925
 926
 927
 928
 929 **A.2.3 DATASET EXAMPLES**
 930
 931

932 This section presents three real HLS code transformation pair examples: performance optimization
 933 (Code restructuring and Directive insertion), synthesizability correction (Code repair), and
 934 adaptation from C-style to HLS-style code (Data-type adaptation and transformation of func-
 935 tions). The dataset is hosted at [https://huggingface.co/datasets/qingyun777yes/](https://huggingface.co/datasets/qingyun777yes/HLStrans)
 936 HLStrans.

937
 938
 939
 940
 941
 942 **1. Performance Optimization** Figures 10a and 10b show a simple K-Nearest Neighbors (KNN)
 943 implementation before and after HLS optimization. The optimized version achieves better perfor-
 944 mance due to improved pipelining, parallelism, and memory optimization.

945
 946
 947
 948
 949
 950
 951 **2. Synthesizability Transformation** Figures 11a and 11b illustrate the transformation from a
 952 non-synthesizable function into a valid HLS-compatible version.

953
 954
 955
 956
 957
 958
 959 **3. C-style to HLS-style Conversion** Figures 12a and 12b demonstrate how C-style data types and
 960 functions can be adapted into HLS-friendly forms.

961
 962
 963
 964
 965
 966
 967 **A.3 PROMPT DETAILS**
 968
 969
 970

971 We explore three types of prompts used for HLS code transformation: zero-shot, chain-of-thought,
 and retrieval-augmented.

```

972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025

```

```

extern "C"{
void workload(
    float inputQuery[NUM_FEATURE],
    float searchSpace[NUM_PT_IN_SEARCHSPACE*NUM_FEATURE],
    float distance[NUM_PT_IN_SEARCHSPACE]
){
    #pragma HLS INTERFACE m_axi port=inputQuery offset=slave bundle=gmem
    #pragma HLS INTERFACE s_axilite port=inputQuery bundle=control
    #pragma HLS INTERFACE m_axi port=searchSpace offset=slave
    bundle=gmem
    #pragma HLS INTERFACE s_axilite port=searchSpace bundle=control
    #pragma HLS INTERFACE m_axi port=distance offset=slave bundle=gmem
    #pragma HLS INTERFACE s_axilite port=distance bundle=control
    #pragma HLS INTERFACE s_axilite port=return bundle=control

    float sum;
    float feature_delta;
L1: for(int i = 0; i < NUM_PT_IN_SEARCHSPACE; ++i){
    sum = 0.0;
L2:   for(int j = 0; j < NUM_FEATURE; ++j){
        feature_delta = searchSpace[i*NUM_FEATURE+j] - inputQuery[j];
        sum += feature_delta*feature_delta;
    }
    distance[i] = sum;
}
return;
}
}

```

(a) Unoptimized KNN implementation

```

...
void workload(
    float inputQuery[NUM_FEATURE],
    INTERFACE_WIDTH searchSpace[NUM_PT_IN_SEARCHSPACE*NUM_FEATURE/WIDTH_FACTOR],
    INTERFACE_WIDTH distance[NUM_PT_IN_SEARCHSPACE/WIDTH_FACTOR]
){
    #pragma HLS INTERFACE m_axi port=inputQuery offset=slave bundle=gmem
    #pragma HLS INTERFACE s_axilite port=inputQuery bundle=control
    #pragma HLS INTERFACE m_axi port=searchSpace offset=slave bundle=gmem
    #pragma HLS INTERFACE s_axilite port=searchSpace bundle=control
    #pragma HLS INTERFACE m_axi port=distance offset=slave bundle=gmem
    #pragma HLS INTERFACE s_axilite port=distance bundle=control
    #pragma HLS INTERFACE s_axilite port=return bundle=control

L7: float local_inputQuery[NUM_FEATURE];
L8: INTERFACE_WIDTH local_searchSpace_0[NUM_PT_IN_BUFFER*NUM_FEATURE/WIDTH_FACTOR];
L9: INTERFACE_WIDTH local_searchSpace_1[NUM_PT_IN_BUFFER*NUM_FEATURE/WIDTH_FACTOR];
L10: INTERFACE_WIDTH local_distance_0[NUM_PT_IN_BUFFER/WIDTH_FACTOR];
L11: INTERFACE_WIDTH local_distance_1[NUM_PT_IN_BUFFER/WIDTH_FACTOR];
L12: LOAD_INPUTQUERY: for (int i(0); i<NUM_FEATURE; ++i){
    #pragma HLS UNROLL
    local_inputQuery[i] = inputQuery[i];
}
L13: TILED_PE: for (int tile_idx(0); tile_idx<NUM_TILES+2; ++tile_idx){
    #pragma HLS pipeline
    int load_flag = tile_idx >= 0 && tile_idx < NUM_TILES;
    int compute_flag = tile_idx >= 1 && tile_idx < NUM_TILES + 1;
    int store_flag = tile_idx >= 2 && tile_idx < NUM_TILES + 2;
    if (tile_idx % 2 == 0) {
        load(load_flag, tile_idx, searchSpace, local_searchSpace_0);
        compute(compute_flag, local_inputQuery, local_searchSpace_1, local_distance_1);
        store(store_flag, tile_idx-2, local_distance_0, distance);
    }
    else {
        load(load_flag, tile_idx, searchSpace, local_searchSpace_1);
        compute(compute_flag, local_inputQuery, local_searchSpace_0, local_distance_0);
        store(store_flag, tile_idx-2, local_distance_1, distance);
    }
}
return;
}
}

```

(b) Optimized KNN implementation

Figure 10: Comparison of KNN implementations: (a) Unoptimized and (b) Optimized for high performance.

```

1026
1027
1028 #include <stdlib.h>
1029 #include <stdio.h>
1030 #include <string.h>
1031 long long compute_sum(int *din, int N, int width) {
1032     long long *out_accum = malloc(sizeof(long long));
1033     int *array_local = malloc(64 * sizeof(int));
1034     for (int i = 0; i < N; i++) {
1035         if (i < width) array_local[i] = din[i];
1036         else      array_local[i] = din[i] >> 2;
1037     }
1038     *out_accum = 0;
1039     for (int j = 0; j < N; j++) {
1040         *out_accum += array_local[j];
1041     }
1042     long long result = *out_accum;
1043     free(out_accum);
1044     free(array_local);
1045     return result;
1046 }

```

(a) Original non-synthesizable code

```

1047
1048 #include <stdlib.h>
1049
1050 long long compute_sum(int *din, int N, int width) {
1051 #ifdef NO_SYNTH
1052     long long *out_accum = malloc(sizeof(long long));
1053     int *array_local = malloc(64 * sizeof(int));
1054 #else
1055     long long _out_accum;
1056     int _array_local[64];
1057     long long *out_accum = &_out_accum;
1058     int *array_local = _array_local;
1059 #endif
1060
1061 #pragma HLS ARRAY_PARTITION variable=_array_local complete
1062     LOOP_SHIFT: for (int i = 0; i < N; i++) {
1063 #pragma HLS PIPELINE II=1
1064         array_local[i] = (i < width) ? din[i] : (din[i] >> 2);
1065     }
1066     *out_accum = 0;
1067     LOOP_ACCUM: for (int j = 0; j < N; j++) {
1068 #pragma HLS PIPELINE II=1
1069         *out_accum += array_local[j];
1070     }
1071     return *out_accum;
1072 }

```

(b) Modified synthesizable code

Figure 11: Transformation from non-synthesizable code to synthesizable code: (a) Original and (b) Modified version.

```

1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133

```

```

#include <math.h>
#include <ap_fixed.h>

#define C 64
#define H 28
#define W 28

void tanh(float input[C][H][W], float output[C][H][W])
{
    for (int c = 0; c < C; ++c) {
        for (int h = 0; h < H; ++h) {
            for (int w = 0; w < W; ++w) {
                output[c][h][w] = std::tanhf(input[c][h][w]);
            }
        }
    }
}

```

(a) Original C-style code using standard data types

```

1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133

```

```

...
typedef ap_fixed<16, 5> data_t;
void store_feature_map(data_t output_buffer[C][H][W], data_t
output_dram[C][H][W])
{
    #pragma HLS inline off
    for (int c = 0; c < C; c++)
    {
        for (int h = 0; h < H; h++)
        {
            for (int w = 0; w < W; w++)
            {
                output_dram[c][h][w] = output_buffer[c][h][w];
            }
        }
    }
}
void compute_exp(data_t input[C][H][W], data_t output[C][H][W])
{
    #pragma HLS inline off
    for (int i = 0; i < C; i++)
    {
        for (int j = 0; j < H; j++)
        {
            for (int k = 0; k < W; k++)
            {
                output[i][j][k] = hls::exp(input[i][j][k]);
            }
        }
    }
}
...

```

(b) Transformed HLS-style code using synthesizable types

Figure 12: Transformation from traditional C-style to HLS-style coding: (a) Original code and (b) Synthesizable HLS code.

1134
1135 If the program can not be synthesized, please turn it into
1136 synthesizable codes. If it is a slow high level synthesis FPGA
1137 program, optimize their performance with minimal resource.
1138
1139 ### program : {src_code}
1140
1141 Must Only return the code use the format. \n
1142 Example response format:
1143
1144
1145 ``cpp \n
1146 // implementation content here
1147

Figure 13: Zero-shot prompt used for HLS code transformation.

1153 Let's think step by step to optimize the HLS code.
1154 Example 2:
1155 Q: This is a slow HLS FPGA program. Please optimize it with array partitioning and loop unrolling to improve
1156 parallelism.
1157 ````cpp`
1158 `void vector_add(const int A[32], const int B[32], int C[32]) { for (int i = 0; i < 32; i++) { C[i] = A[i] + B[i]; } }`
1159 `````
1160 1. Identify memory contention: single-port arrays limit one access per cycle.
1161 2. Partition arrays: use `#pragma HLS ARRAY_PARTITION variable=A/B/C cyclic factor=4` to split each into 4 banks
1162 for parallel access.
1163 3. Unroll the loop: add `#pragma HLS UNROLL factor=4` so 4 additions execute in one cycle, matching the 4-way
1164 partition.
1165 4. Keep pipelining: you may optionally add `#pragma HLS PIPELINE II=1` for consistency.
1166 ````cpp`
1167 `void vector_add(const int A[32], const int B[32], int C[32]) {`
1168 `#pragma HLS ARRAY_PARTITION variable=A cyclic factor=4`
1169 `#pragma HLS ARRAY_PARTITION variable=B cyclic`
1170 `factor=4`
1171 `#pragma HLS ARRAY_PARTITION variable=C cyclic factor=4`
1172 `for (int i = 0; i < 32; i++) { #pragma HLS UNROLL`
1173 `factor=4`
1174 `C[i] = A[i] + B[i]; } }`
1175 `````
1176 Now apply the same step-by-step reasoning to the following slow HLS code and provide the fully annotated,
1177 optimized version:
1178 If the program can not be synthesized, please turn it into synthesizable codes. If it is a slow high level synthesis
1179 FPGA program, optimize their performance with minimal resource.
1180 `### program : {src_code}`
1181 Must Only return the code use the format. `\n`
1182 Example response format:
1183 ````cpp\n`
1184 `// implementation content here`
1185 `````

Figure 14: Chain-of-thought prompt for step-by-step transformation.

A.4 DETAILED EXPERIMENT RESULTS

1187 Unlike traditional software code, which need only pass functional correctness tests, HLS-generated kernels must also successfully synthesize and implement via the Vitis HLS toolchain to be deploy-

```

1188
1189 Let's think step by step to optimize the HLS code.
1190 Q: This is a slow HLS FPGA program, which is similar to the current unoptimized codes.
1191 Retrieval codes {Retrieval codes }
1192
1193 Now apply the same step-by-step reasoning to the following slow HLS code and provide the fully annotated,
1194 optimized version:""
1195 If the program can not be synthesized, please turn it into synthesizable codes. If it is a slow high level synthesis
1196 FPGA program, optimize their performance with minimal resource.
1197 ### program : {src_code}
1198 Must Only return the code use the format. \n
1199 Example response format:
1200   ````cpp \n
1201   // implementation content here
1202

```

Figure 15: Retrieval-augmented prompt for enhanced transformation.

able on FPGA hardware. Below, we briefly describe how we leverage synthesis results for design evaluation. Also, we introduce the fine tuning results during training.

A.4.1 HLS SYNTHESIS RESULTS EXAMPLE

While HLS synthesis cannot yield perfectly accurate timing or resource-utilization numbers, it provides essential estimates for comparing design variants. Figures 16a and 16b show the synthesis reports for the unoptimized and optimized KNN kernels, respectively, targeting a Xilinx Alveo U55C accelerator at a 300 MHz kernel clock.

The optimized design trades increased resource usage more DSP slices, flip-flops (FFs), and lookup tables (LUTs) for a dramatic fourfold reduction in latency cycles (from 2,097,324 cycles down to 508,479 cycles). In a real FPGA deployment, this corresponds to an end-to-end runtime of approximately 3.2 ms versus roughly 17 ms for the unoptimized kernel, while still remaining within resource budgets. We report *acceleration* based on the estimated latency cycles from the synthesis reports. However, HLS synthesis itself can be time-consuming, particularly for large or highly optimized designs.

```

1221
1222 + Performance & Resource Estimates:
1223 PS: '+' for module; 'o' for loop; '*' for dataflow
1224 +-----+-----+-----+-----+-----+-----+-----+-----+-----+
1225 | Modules | Issue | Latency | Latency | | BRAM | DSP | FF | LUT | URAM |
1226 | & Loops | Type | Slack | (cycles) | (ns) | | | | |
1227 +-----+-----+-----+-----+-----+-----+-----+-----+-----+
1228 |+ workload | Timing| -0.00 | 2097324 | 6.990e+06 | -| 7 (-0%) | 27133 (1%) | 10274 (-0%) | -|
1229 |+ workload_Pipeline_VITIS_LOOP_19_1 | Timing| -0.00 | 2097253 | 6.990e+06 | -| 7 (-0%) | 22069 (-0%) | 1406 (-0%) | -|
1230 | o VITIS_LOOP_19_1 | II | 2.43 | 2097251 | 6.990e+06 | -| - | - | - | -|
1231 +-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

(a) Unoptimized KNN HLS synthesis results

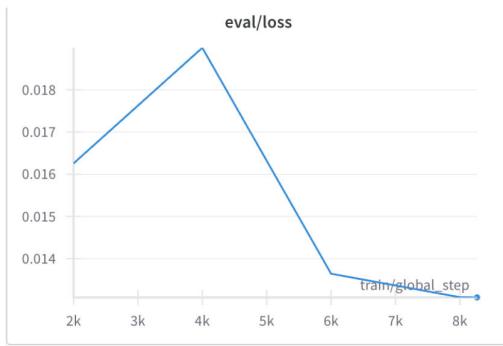
```

1232 + Performance & Resource Estimates:
1233 PS: '+' for module; 'o' for loop; '*' for dataflow
1234 +-----+-----+-----+-----+-----+-----+-----+-----+-----+
1235 | Modules | Issue | Latency | Latency | | BRAM | DSP | FF | LUT | URAM |
1236 | & Loops | Type | Slack | (cycles) | (ns) | | | | |
1237 +-----+-----+-----+-----+-----+-----+-----+-----+-----+
1238 |+ workload | Timing| -0.00 | 508479 | 1.695e+06 | -| 224 (2%) | 102380 (3%) | 28272 (2%) | -|
1239 |+ workload_Pipeline_LOAD_INPUTQUERY | - | 1.61 | 4 | 13,332 | - | - | 133 (-0%) | 119 (-0%) | -|
1240 | o LOAD_INPUTQUERY | - | 2.43 | 2 | 6,666 | - | - | - | - | -|
1241 +-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

(b) Optimized KNN HLS synthesis results

Figure 16: (a) Unoptimized and (b) optimized KNN implementations after HLS synthesis.

1242 A.4.2 FINETUNE RESULTS
1243

(a) Validation results for Qwen2.5-Coder-3B-Instruct during fine-tuning.



(b) Validation results for Qwen2.5-Coder-7B-Instruct during fine-tuning.

1258 Figure 17: Validation performance of Qwen2.5-Coder models during fine-tuning on the held-out
1259 dataset.

1261 With our real-world corpus, we reserve two C programs for the repair task and another 39 programs
1262 for the optimization task. The remaining 270 programs are split into training and validation sets.
1263 Figure 17a presents the validation loss curve for Qwen2.5-Coder-3B-Instruct, while Figure 17b
1264 shows the corresponding curve for Qwen2.5-Coder-7B-Instruct during fine-tuning. In both cases, the
1265 steadily decreasing loss demonstrates that fine-tuning effectively adapts the models to our dataset.
1266

1267 A.5 IMPACT OF C TO HLS TASK
1268

1269 While HLS is syntactically close to C, we believe the task has the following meaning.

1270 **Impact for reducing performance gap.** HLS is designed to accelerate hardware design, but there
1271 remains a substantial gap between plain C code and high-performance HLS code. In our experiments,
1272 LLM-generated samples can achieve up to hundreds of speedup over the original code.

1273 **Impact for reducing coding budget.** To obtain high-quality HLS code, developers must perform
1274 non-trivial semantic transformations—such as loop tiling, bitwidth narrowing, converting buffer-
1275 based designs to streaming, or repairing code to satisfy synthesis constraints. These transformations
1276 are time-consuming and require HLS expertise. A fine-tuned LLM can automate or assist with many
1277 of these steps, significantly reducing development effort and turnaround time.

1278 **Impact for agile hardware design with HLS.** Agile hardware design that starts from HLS
1279 enables software engineers to develop hardware accelerators more easily. However, understanding
1280 the hardware-specific transformations required for optimization is non-trivial. Our dataset and fine-
1281 tuned LLM help software engineers better design hardware accelerators.

1282 **Real Case study: C to HLS task.** We use a genomics application as a real-world case study for the
1283 C-to-HLS conversion task Cong et al. (2022) in Table 10. High-performance HLS implementations
1284 include several components Cong et al. (2022). Converting C to HLS consumes 41% of the engi-
1285 neering effort, covering compiler directives, double buffering, and related transformations, whereas
1286 the function-level C code accounts for 59%. These conversion steps can require days to finish Cong
1287 et al. (2022), indicating that C-to-HLS conversion is a challenging problem that merits deeper study.

1291 A.6 TRANSFERRING ON DIFFERENT PLATFORMS
1292

1293 To clarify our claim: constructing high-performance HLS implementations from C typically requires
1294 the five transformations illustrated in Figure 1. These transformations are common across modern
1295 HLS toolchains such as Vitis HLS, SmartHLS, and Bambu HLS. Table 11 lists some examples of
1296 five transformations for different HLS tools including SmartHLS, Vitis HLS and Bambu HLS.

1296

Table 10: Breakdown of effort for a real-world C-to-HLS conversion task.

1297

1298

1299

1300

1301

1302

1303

1304

1305

1306

1307

1308

1309

1310

1311

1312

1313

1314

1315

1316

1317

1318

1319

1320

1321

1322

1323

1324

1325

1326

1327

1328

Category	Sub-category	LOC	Percentage
Functionality code		308	59%
Optimizations code		216	41%
Optimizations code	Compiler directives	48	22%
	Double buffering	46	21%
	Frequency optimization	38	18%
	PE duplication	32	15%
	Others	52	24%

Table 11: Common HLS transformations and examples in different toolchains

Transformation	Why needed (in HLS)	Vitis HLS example	SmartHLS example	Bambu HLS example
T1: Code Restructuring	Expose data locality and so on	loop tiling, dataflow	loop tiling, dataflow	loop tiling, dataflow
T2: Directive (Pragma) Insertion	Increase parallelism and so on	#pragma HLS UNROLL	#pragma HLS loop unroll	#pragma HLS unroll
T3: Data-Type Adaptation	Adapt to platform	ap_int<64>	ap_int<64>	ap_int<64>
T4: Transformation of Functions	Hardware implementations for expensive math or others	sqrt	sqrt	sqrt
T5: HLS-Compliant Coding Style	Recursion or dynamic memory allocation not synthesizable	recursion	recursion	recursion

The augmentation techniques and the benchmarking methodology operate at the level of HLS transformations and therefore generalize across modern HLS toolchains. To substantiate this claim, we evaluate the generality of our approach on two additional HLS toolchains: Bambu and SmartHLS (LegUp).

We apply our augmentation pipeline to transform C programs into high-performance HLS designs by performing the five targeted transformations described in Figure 1. For each benchmark/toolchain we report two metrics: *Speedup*, the relative performance improvement of the optimized design over the baseline; and *Pass rate*, the fraction of generated designs that both pass the functional tests and successfully synthesize. These results in Table 12 and 13 demonstrate that our augmentation techniques produce measurable performance gains across multiple, independently developed HLS toolchains, supporting the claim that the pipeline and evaluation methodology generalize beyond a single vendor.

1337

1338

1339

1340

1341

1342

1343

1344

1345

1346

1347

1348

1349

Table 12: Augmentation pipeline evaluation for Bambu HLS

Metric	cfd_flux	dilate	gicov	hotspot	kmeans	knn	nw	pathfinder	srad	streamcluster
Speedup	2.31	24.5	1.06	4.10	41.7	9.52	2.72	3.36	2.91	1.20
Pass rate	0.29	0.19	0.21	0.21	0.30	0.20	0.47	0.46	0.16	0.27

Table 13: Augmentation pipeline evaluation for SmartHLS

Metric	cfd_flux	dilate	gicov	hotspot	kmeans	knn	nw	pathfinder	srad	streamcluster
Speedup	2.8	30.1	1.3	5.2	50.1	11.6	3.4	4.2	3.7	1.5
Pass rate	0.36	0.24	0.27	0.27	0.37	0.26	0.59	0.59	0.21	0.35

1350
 1351 We evaluate our benchmarks on two additional HLS toolchains, Bambu HLS and SmartHLS
 1352 (LegUp), using multiple LLMs. Table 14 and Table 15 report the zero-shot best@1 prompting
 1353 results and error breakdown for Bambu HLS; Table 16 and Table 17 provide the corresponding
 1354 results for SmartHLS. Metrics are defined in Section 4.2 of the manuscript. “Speed/Opt” denotes
 1355 the fraction of cases with any improvement (reported as percentage), “Min/Avg/Max” are relative
 1356 speedups, “Functional Accuracy” is the fraction of outputs passing functional tests, and “Synthesis
 1357 Accuracy” is the fraction that both pass functional tests and successfully synthesize.

1358 Table 14: Benchmark results of Bambu HLS
 1359

Model	Opt (%)	Min	Avg	Max	Functional Accuracy	Synthesis Accuracy
Deepseek-R1	12.8%	0.10 \times	1.10 \times	10.2 \times	30.8%	28.2%
GPT-5	15.4%	0.03 \times	8.20 \times	310.5 \times	33.3%	33.3%
Grok-4	12.8%	0.30 \times	1.50 \times	30.3 \times	28.2%	28.2%
Gemini-2.5-pro	17.9%	0.60 \times	1.90 \times	21.2 \times	25.6%	25.6%
Qwen coder 32B	10.3%	0.20 \times	0.70 \times	2.5 \times	38.5%	35.9%

1366
 1367 Table 15: Error analysis of Bambu HLS.
 1368

Model	Compiler Errors (%)	Output Errors (%)	Runtime Exceptions (%)	Resource Errors (%)	Directive Errors (%)
32B	40	8	15	15	22
Deepseek-R1	41	9	17	13	20
Gemini25	52	8	8	15	17
GPT-5	34	11	21	11	23

1378 Table 16: Benchmark results of SmartHLS (LegUp).
 1379

Model	Opt (%)	Min	Avg	Max	Functional Accuracy	Synthesis Accuracy
Deepseek-R1	15.4%	0.08 \times	0.90 \times	9.0 \times	25.6%	23.1%
GPT-5	12.8%	0.02 \times	7.50 \times	200.0 \times	35.9%	35.9%
Grok-4	10.3%	0.25 \times	1.20 \times	25.0 \times	23.1%	25.6%
Gemini-2.5-pro	15.4%	0.70 \times	2.10 \times	25.0 \times	28.2%	25.6%
Qwen coder 32B	12.8%	0.25 \times	0.80 \times	3.0 \times	33.3%	38.5%

1388 Table 17: Error analysis of SmartHLS.
 1389

Model	Compiler Errors (%)	Output Errors (%)	Runtime Exceptions (%)	Resource Errors (%)	Directive Errors (%)
32B	38	9	16	14	23
Deepseek-R1	39	10	16	12	23
Gemini25	43	7	9	14	27
GPT-5	33	12	20	12	23

1398 A.7 TESTBENCH GENERATIONS

1400
 1401 We report the coverage results, lines, branches, tokens, and calls collected from `gcov` for our
 1402 dataset, as shown in Table 18. While full (100%) coverage is not attainable, the table demonstrates
 1403 that our testbench nevertheless yields robust, high-quality coverage for evaluation.

1404 Table 18: Coverage results collected from `gcov` for our dataset.
1405

1406 Range	1407 Lines (%)	1408 Branches (%)	1409 Tokens (%)	1410 Calls (%)
1408 100%	1409 94.82	1410 94.82	1411 79.29	1412 92.88
1408 [75%,100%)	1409 4.85	1410 4.85	1411 10.03	1412 0.65
1408 [50%,75%)	1409 0.00	1410 0.32	1411 9.71	1412 6.47
1408 [25%,50%)	1409 0.32	1410 0.00	1411 0.97	1412 0.00
1408 < 25%	1409 0.00	1410 0.00	1411 0.00	1412 0.00

1413
1414
1415
1416 A.8 CODE STRUCTURE ANALYSIS
14171418 For the code structure analysis we computed per-sample statistics including lines of code (LoC),
1419 number of functions, number of loops, and cyclomatic complexity in Table 19.1420 From these tables, we conclude that the dataset covers a wide variety of code styles and complexity
1421 levels, and is therefore appropriate for evaluating LLM performance on HLS-related tasks.
14221423 Table 19: Dataset distributions for code-structure metrics. Each cell shows the bin range (top) and
1424 the percentage of samples falling in that bin (bottom).
1425

1426 Metric	1427 Bin1 (Range)	1428 Bin2 (Range)	1429 Bin3 (Range)	1430 Bin4 (Range)	1431 Bin5 (Range)
1428 Lines of Code (LoC)	1429 [3.00, 44.40] 39.51%	1430 [44.40, 85.80] 32.33%	1431 [85.80, 127.20] 10.35%	1432 [127.20, 168.60] 10.22%	1433 [168.60, 210.00] 7.60%
1432 Function number	1433 [0.00, 2.00] 57.74%	1434 [2.00, 4.00] 27.84%	1435 [4.00, 6.00] 8.99%	1436 [6.00, 8.00] 3.07%	1437 [8.00, 10.00] 2.36%
1436 Loop number	1437 [0.00, 7.80] 41.88%	1438 [7.80, 15.60] 27.34%	1439 [15.60, 23.40] 15.79%	1440 [23.40, 31.20] 10.75%	1441 [31.20, 39.00] 4.23%
1441 Cyclomatic complexity	1442 [1.00, 14.00] 48.95%	1443 [14.00, 27.00] 27.58%	1444 [27.00, 40.00] 11.60%	1445 [40.00, 53.00] 6.23%	1446 [53.00, 66.00] 4.63%

1447 A.9 EXPERIMENTAL ANALYSIS
14481449 A.9.1 DETAILED ERROR ANALYSIS
14501451 We perform a fine-grained analysis of the failures produced by LLM-generated HLS designs and
1452 identify five dominant error categories: *Compiler Errors*, *Directive Errors*, *Runtime Exceptions*,
1453 *Resource Errors*, and *Output Errors*. Across both Bambu HLS and SmartHLS (LegUp), directive-
1454 related errors are particularly prevalent: models commonly emit Vitis-style pragmas even when the
1455 target tool requires a different pragma syntax. We attribute this behavior to the relative abundance
1456 and higher quality of Vitis HLS examples in training data.1457 **Compiler Errors.** These errors reflect syntactic or structural problems that prevent the HLS front-
1458 end from accepting the program (e.g., malformed C, undefined identifiers, or unsupported language
1459 constructs). Because such errors occur before downstream HLS passes, they represent a primary
1460 bottleneck in the overall workflow and indicate fragile tool compatibility.1461 **Directive Errors.** This category captures incorrect or unsupported pragma usage (e.g., wrong
1462 pragma names, invalid parameters, incorrect placement, or mixing pragmas intended for differ-
1463 ent tools). Directive errors show that models lack fine-grained tool-awareness: even small syntax

1458 differences between toolchains (Vitis vs. Bambu vs. LegUp/SmartHLS) cause a large fraction of
 1459 failures.

1460 **Runtime Exceptions.** A nontrivial fraction of generated programs compile but fail during simulation
 1461 (exceptions, timeouts, memory faults, or sandbox interruptions). These failures indicate
 1462 difficulties in producing correct hardware control-path logic and robust testable code, beyond purely
 1463 numerical computation.

1464 **Resource Errors.** Resource-related failures occur when aggressive transformations (e.g., excessive
 1465 unrolling or partitioning) push designs beyond the target device’s resource budgets. Although less
 1466 frequent than compiler or directive errors, resource errors are critical for practical deployability and
 1467 show that models tend to over-parallelize without awareness of device constraints.

1468 **Output Errors.** Semantic mismatches (wrong algorithmic behavior, off-by-one/boundary mistakes,
 1469 or incorrect output format) are the least common error type. This suggests that, once a design
 1470 compiles and simulates, LLMs generally preserve core algorithmic behavior reasonably well — i.e.,
 1471 functional correctness is easier to achieve than tool-specific syntactic and compilation constraints.

1473 **A.9.2 SPEEDUP ANALYSIS**

1474 We analyze how model-generated transformations affect performance, focusing on the two stages
 1475 with the largest impact: **T2** (pragma/directive insertion) and **T1** (code restructuring). Below we
 1476 report the empirical distribution of optimization actions extracted from generated programs and
 1477 summarize observed performance patterns.

1478 **Breakdown of T2 (pragma / directive insertion):** Table 20–22 summarize the relative proportion
 1479 of common T2 actions observed for each toolchain. Note that proportions reflect the fraction of
 1480 generated designs that include a given action; a single design may include multiple actions, so row
 1481 sums can exceed 100%.

1482 Table 20: T2 (Vitis_HLS) distribution of pragma/directive actions (proportions).

Action	Pragmas	Array-part	MemType	Unroll	Merge	Inline	Pipeline	Dataflow	Dep/Stream
Proportion	43.6%	12.8%	53.8%	5.1%	28.2%	82.1%	10.3%	10.3%	10.3%

1483 Table 21: T2 (Bambu HLS) distribution of pragma/directive actions (proportions).

Action	Pragmas	Unroll	Inline	Dataflow / Cache
Proportion	30.8%	69.2%	10.3%	87.2%

1484 Table 22: T2 (SmartHLS) distribution of pragma/directive (proportions).

Action	Pragmas	Unroll	Inline	Dataflow	Pipeline / Partition
Proportion	53.8%	17.9%	10.3%	84.6%	43.6%

1504 **Breakdown of T1 (code restructuring):** Table 23 reports the observed distribution of common
 1505 T1 restructuring patterns. These transformations are closely related to memory-bound performance
 1506 improvements.

1507 Table 23: T1 code restructuring distribution.

Action	Memory coalescing	Local tiling	Ping-pong buffer	Dataflow	Control-flow opt.
Percentage	0.0%	23.1%	7.7%	2.6%	28.2%

1512 A.9.3 OBSERVATIONS
1513

1514 • **Optimization can degrade performance.** Some LLM-generated transformations yield $<1\times$
1515 speedup. Two common causes are (i) restructuring that introduces additional loop dependencies
1516 (increasing latency), and (ii) pragmas that are less effective than the tool’s default optimizations.
1517 This observation underscores the need for dataset and reward signals that encourage *correct*
1518 (tool-aware) optimizations rather than aggressive but counterproductive rewriting.

1519 • **T1 correlates with memory-bound gains.** For memory-intensive kernels, speedups are pri-
1520 marily driven by T1 transformations that improve memory behavior: memory coalescing (better
1521 burst efficiency), local tiling (reduced off-chip bandwidth), and ping-pong buffering (overlap of
1522 compute and memory).

1523 • **T2 impact depends on application class.** Pipeline and Dataflow pragmas are most beneficial for
1524 streaming and stencil kernels where concurrency is the bottleneck. Unroll and Partition pragmas
1525 are critical for compute-bound kernels (e.g., KNN, GEMM). Inline and loop-merge transforma-
1526 tions matter more in control-heavy applications by reducing scheduling overhead and enabling
1527 deeper pipelining.

1528 • **Tool-specific defaults shape effectiveness.** The observed T2 distribution differs across
1529 toolchains because each HLS tool applies different default transformations and heuristics; con-
1530 sequently, identical pragma insertions can produce different outcomes across tools. This further
1531 motivates our claim that benchmarking at the transformation level (rather than at a single tool’s
1532 syntax) yields more robust conclusions.

1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565