From Pragmas to Partners: A Symbiotic Evolution of Agentic High-Level Synthesis

(Invited Paper)

Niansong Zhang
Cornell University
Ithaca, USA
nz264@cornell.edu

Sunwoo Kim
Cornell University
Ithaca, USA
sk3463@cornell.edu

Andrew Butt
Cornell University
Ithaca, USA
atb78@cornell.edu

Shreesha Srinath
Cerebras Systems
Sunnyvale, USA
ss2783@cornell.edu

Zhiru Zhang
Cornell University
Ithaca, USA
zhiruz@cornell.edu

Abstract

The rise of large language models has renewed interest in AI-driven hardware design and whether high-level synthesis (HLS) still matters in the agentic era. While we expect mature agentic hardware systems to leverage both HLS and RTL, we argue that HLS remains essential. Its faster iteration, portability, and design permutability make it a natural abstraction layer and golden reference for agentic optimization. We identify key limitations of current HLS tools, namely inadequate performance feedback, rigid interfaces, and limited debuggability that agents are uniquely positioned to address. We also propose a a taxonomy for the symbiotic evolution of agentic HLS, clarifying how responsibility shifts from human designers to AI agents as systems advance from copilots to autonomous design partners. Across PolyBench and 16K FlashAttention, our experiments demonstrate more token-efficient HLS search and 2.9× higher throughput than a human-designed Fused Systolic Attention baseline.

CCS Concepts

• **Hardware → High-level and register-transfer level synthesis**; • **Computing methodologies → Intelligent agents**.

Keywords

high-level synthesis, agentic AI, large language models, hardware design automation

ACM Reference Format:
Niansong Zhang, Sunwoo Kim, Andrew Butt, Shreesha Srinath, and Zhiru Zhang. 2026. From Pragmas to Partners: A Symbiotic Evolution of Agentic High-Level Synthesis (Invited Paper). In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '26), November 08–12, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3831252.3845856

1 Introduction

High-level synthesis (HLS) lifts accelerator design from RTL to high-level software specification, but achieving high performance still requires substantial manual, iterative, expertise-driven optimization: restructuring code, rewriting kernels, and adding pragmas guided by toolchain feedback to shape compute, memory, data layout, and communication.

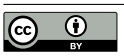
The rise of large language models and agentic AI has sparked renewed interest in automating this process. A natural question arises: does HLS still matter, or will AI simply generate RTL directly? We argue that HLS remains not only relevant but essential in the agentic era. We do not claim that HLS alone or RTL alone solves agentic hardware design. We expect a mixed workflow where agents move between high-level intent, HLS source, and RTL, depending on the task. This paper focuses on the role of HLS in that workflow.

HLS provides an executable high-level specification that serves as a golden reference for fast simulation. Its highly permutable representation lets agents move between design points while preserving functional semantics, whereas equivalent RTL transformations require invasive rewrites. Our experiments quantify these benefits. The experiments also show that RTL agents can exploit testbench gaps by weakening arithmetic semantics, underscoring the need for formal verification, which is faster with HLS specifications than at the RTL level.

RTL remains important when a task requires cycle-accurate control, custom interfaces, or low-level refinement. Our guardrailed 16K FlashAttention workflow shows how agents extend HLS: an agent optimizes the compute kernel through HLS and writes an RTL adaptor to connect its AXI streams to a custom UCIe interface unsupported by HLS. Combining mixed-fidelity feedback, formal equivalence, workload testing, and RTL interface generation, the workflow achieves 2.9× higher throughput than a resource-matched human-designed Fused Systolic Attention baseline.

This work makes the following contributions:

- **HLS as a strong abstraction layer for agentic hardware design.** We show how HLS enables faster iteration, portability, and design permutability beyond pure RTL generation while retaining an executable specification for verification.
- **HLS limitations that agents can address.** We identify gaps in feedback, interfaces, verification, and debugging, then address them in a guardrailed 16K FlashAttention workflow that generates RTL for UCIe integration and outperforms a resource-matched human design by 2.9×.
- **An autonomy-inspired taxonomy for the symbiotic evolution of agentic HLS.** We introduce a taxonomy that clarifies capabilities and shifting human–agent responsibilities.
- **A controlled comparison of HLS and RTL agents.** A controlled PolyBench study quantifies design quality, search cost, and verification behavior, including 5.6× fewer HLS tokens on PolyBench.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ICCAD '26, San Jose, CA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/10.1145/3831252.3845856

2 Related Work

Agentic RTL Design. Early work established natural language interaction for RTL design. Chip Chat used a conversational model to coarchitect an 8 bit processor through tapeout with a hardware engineer [5]. AutoChip closed the loop with compiler and simulation feedback [27]. Later systems made this feedback more structured [36]. RTLFixer uses compiler diagnostics for syntax repair [28], while VerilogCoder adds graph based planning and source level waveform tracing for functional debugging [9]. Spec2RTL Agent processes complex documents through pseudocode, Python, and synthesizable C++ before invoking HLS [35]. This progression shows that effective design agents plan across representations and revise designs using tool feedback. Our experiments quantify how HLS and RTL affect search cost and design quality under the same agent loop.

Agentic HLS Design. HLS gives agents executable source, optimization directives, and structured compiler feedback [8]. HLSPilot combines application profiling, C/C++ transformation, and pragma search [16, 32]. RALAD retrieves optimization examples to guide source edits [33], while C2HLSC and related repair work use tool feedback to produce synthesizable C/C++ [7, 34]. Recent systems extend these capabilities into longer workflows. LLM DSE combines an agent with a learned performance model for parameter search [30], and ChatHLS uses specialized agents for debugging and directive tuning [13]. These systems automate generation, repair, and bounded optimization [1]. Our taxonomy organizes these capabilities by autonomy, while our guardrailed workflow connects optimization to verification, mixed fidelity feedback, workload testing, and system integration.

High Level Accelerator Design Languages. Accelerator languages expose architectural choices without requiring RTL. T2S-Tensor and Spatial separate algorithms from spatial mappings for parallelism, memory, and communication [11, 25]. Dahlia uses time sensitive affine types to make resource use predictable [17], while Filament captures interface timing with timeline types [18]. Allo decouples algorithms from typed customizations and composes them across function boundaries [6, 31]. These languages expose useful agentic design structures: separate compute and schedule, explicit memory and communication, typed timing, and composable interfaces. They motivate tools that recover the same structure from HLS artifacts while preserving source level context.

Verification and Mixed Fidelity Feedback. Formal methods can guard transformations at several boundaries in the HLS flow. Translation validation checks HLS scheduling and other compiler transformations [12], while PEQC proves equivalence between original and optimized HLS programs across substantial loop, storage, and dataflow changes [20]. Performance tools provide signals at different costs. LightningSimV2 combines execution traces and HLS schedules for fast incremental simulation [22], and OmniSim supports advanced dataflow behavior at near C simulation speed [21]. Multi-fidelity HLS models correlations across HLS, logic synthesis, and implementation [26], while HLPerf provides approximate simulation for data dependent dataflow accelerators [37]. Our feedback agent combines such signals, maps them to source regions, and records whether each conclusion is exact, approximate, or conditional.

3 Why HLS Remains Relevant

Despite advances in AI-driven RTL generation, HLS offers fundamental advantages that make it the natural abstraction layer for agentic hardware design.

Rapid Design Space Exploration. Agentic workflows require many iterations to test correctness and reach design goals/constraints. HLS enables rapid design space exploration (DSE) through two mechanisms. First, it provides an executable high-level software specification that serves as a golden reference, allowing for fast software simulation before hardware generation. Second, HLS representations are highly permutable: one design point can be easily transformed into another while preserving functional semantics. In contrast, direct RTL generation lacks a clear golden reference and equivalent RTL transformations require invasive rewrites that are harder to automate and more error-prone. Operating primarily at RTL also raises the cost of iteration. RTL simulation is slower, and code and traces are more verbose, which increases the token cost for agentic loops that depend on repeated inspections and refinements. Higher abstraction is therefore even more important when optimization is driven by many agentic iterations.

Portability. The same high-level software specification can serve as a single source to target different FPGA families, vendors, or accelerator platforms with minimal modification. Beyond platform portability, HLS enables constraint portability: the same source can be synthesized for high throughput, low latency, or low power by adjusting directives and synthesis settings. For agentic HLS systems, learned optimizations and design patterns can be transferred across platforms and constraints, enabling reuse across projects.

Verification and Debugging. HLS improves verification by centering the workflow around a readable and executable specification. The high-level source serves as a golden reference for functional validation and supports fast regression testing before synthesis. This reduces the burden of debugging in RTL while still allowing RTL checks when needed.

4 HLS Limitations That Agents Can Address

Despite rapid DSE, portability, and ease of verification being central to the value proposition of HLS, current tools fall short in all three. We identify key gaps that agents are uniquely positioned to address.

Performance Feedback Limits DSE. Fast and meaningful performance feedback is crucial for effective DSE, yet current HLS tools fall short. Synthesis reports provide latency estimates, resource utilization, and scheduling information, but these are often difficult to interpret and relate back to source code. Worse, many designs exhibit data-dependent control flow, causing tools to report unknown or variable cycle counts that provide little actionable guidance. Agents can address this through mixed-fidelity performance modeling: extracting operation schedule, loop-level information, and control-flow regions from HLS artifacts to construct performance estimates and replayable traces. This enables actionable feedback even for designs where traditional HLS reports provide only question-mark estimates, presenting insights at the same abstraction level where designers make optimization decisions.

Table 1: Autonomy-inspired levels for agentic HLS.

Level	Name	Human Effort	AI Effort	Key Technology
L0	Manual DSE	High	Low	Standard HLS compilation and reports, with manual code and pragma iteration and largely human-driven feedback and debugging.
L1	HLS Copilot	High	Low	Explains HLS reports with source attribution, suggests local code and pragma edits, helps triage failures, and improves RTL readability for follow-up debugging.
L2	Autotuning Agent	Medium	Medium	Runs closed-loop DSE over pre-defined search space using tool feedback, integrates HLS IPs using standard interfaces, generates lightweight replayable traces for performance analysis, and performs basic differential testing.
L3	Guardrailed Agents	Medium	High	Performs multi-step tool use with strong guardrails and leverages verification to provide strong correctness guarantees, including equivalence checking, assertion synthesis, and counterexample explanations. Constructs mixed-fidelity performance models and performs localized failure and bottleneck analysis for DSE.
L4	Domain Architect	Low	High	Handles system-level optimization and integration with customized interfaces, use complete workloads to drive performance and correctness tests.
L5	Silicon Partner	Low	High	End-to-end autonomy across architectures and tools, with continual learning across projects and improvements to the HLS stack itself (compiler passes, scheduling heuristics, codegen, tracing, and verification infrastructure).

Rigid Interface Limits Portability. A major pain point is composing accelerators with the surrounding system. Tools such as Vitis HLS [3] support standard interfaces such as AXI, but offer limited ability to customize or extend these protocols. Integrating HLS-generated IP with systems built using different frameworks, such as the LiteX SoC builder [10], requires manual writing interface adapters and bridge logic. Tools such as Catapult HLS [24] support more flexible interfaces, but defining protocols remains a complex task. This *composability gap* undermines the portability promise of HLS. AI agents can bridge this gap by understanding both the HLS design intent and the target system's interface requirements, then generating the necessary adapter logic and protocol bridges.

Lack of End-to-End Verification and Debugging. HLS verification today is still split between high-level software simulation, RTL simulation, and ad hoc debugging, with limited help connecting failures back to source intent. Agents can use HLS as a golden reference for both humans and RTL, enabling differential testing, equivalence checking, assertion synthesis, and targeted counterexample explanation across abstraction levels. For many workloads, we also want testbenches written at an even higher level, such as PyTorch models, to drive software simulation and RTL co-simulation with realistic inputs and oracle outputs. This integration is not available in current flows, but an agent could mediate it by generating adapters, harnesses, and simulation drivers. Finally, agents can improve the readability of the generated RTL by refactoring the structure, naming signals, and adding documentation, which reduces the cost of debugging when designers must resort to RTL.

5 Levels of Automation

Recent work explores LLM assistance for HLS, including code generation and guidance [14, 32], QoR prediction [19], automated repair [34], retrieval-augmented optimization [33], software-to-hardware bridging [7], and agentic DSE and multi-agent design [23, 30]. These systems span a wide range of autonomy and human involvement. To characterize this spectrum of autonomy and human involvement, we propose the taxonomy in Table 1. The taxonomy is useful not just for labeling systems, but for highlighting what must improve in agentic HLS to enable greater autonomy, especially actionable performance feedback, flexible interface, improved verification, and debuggability.

At **L0 (Manual DSE)**, designers iterate manually from HLS reports. **L1 (HLS Copilot)** assists with report interpretation and localized edit suggestions, but humans still implement and validate. **L2 (Autotuning Agent)** runs closed-loop optimization over a predefined space and automates routine evaluation and integration. **L3 (Guardrailed Agents)** performs multi-step tool use with strong guardrails, with verification and mixed-fidelity feedback for targeted debugging and DSE. **L4 (Domain Architect)** shifts humans to intent definition and review, while agents drive system-level optimization and workload-level validation within a bounded domain. Finally, **L5 (Silicon Partner)** extends autonomy across architectures and tools, including improving the HLS stack itself.

Current systems operate primarily at L1–L2, with research pushing toward L3. Advancing further depends directly on addressing the composability and feedback limitations discussed in §4.

6 Experimental Results

6.1 HLS versus RTL Agents on PolyBench

Experimental setup. We first ask how the abstraction presented to an agent affects design-space exploration. In the taxonomy of Section 5, both flows use L2 (Autotuning Agent) systems for closed-loop performance optimization within a fixed task and evaluation harness. We compare HLS-Agent, which edits C/C++ and HLS directives, with RTL-Agent, which edits the Verilog generated by Vitis HLS from the same unoptimized kernel. Both use Claude Opus 5 through the same Claude Code harness. Each round launches a fresh agent process with read, edit, write, search, and file-discovery tools, but no shell or direct access to the EDA tools. The harness compiles, simulates, and synthesizes the candidate, then includes the measured result and the complete iteration history in the next prompt. HLS-Agent can modify only the kernel source and header, and receives a read-only optimization reference condensed from the Vitis HLS 2023.2 User Guide (UG1399) [3], covering directives, scheduling, dependence restructuring, memory banking, buffering,

Table 2: Performance and search cost after 20 rounds. Speedup is RTL/HLS cycles.

Kernel	Flow	Cycles	Valid rounds	Tokens (M)	Cost (USD)	Total time (h)
atax	RTL	14,544	18	32.9	68.0	6.0
	HLS	14,570 (1.0×)	17	3.3	21.3	3.6
bicg	RTL	14,395	13	19.8	50.6	4.8
	HLS	14,523 (1.0×)	20	3.8	21.7	3.7
cholesky	RTL	2,015	17	30.9	74.8	8.1
	HLS	2,126 (0.9×)	20	4.2	28.6	6.2
covariance	RTL	987	20	16.3	49.0	5.0
	HLS	991 (1.0×)	19	3.5	22.1	3.8
gemm	RTL	9,172	20	20.2	48.7	6.2
	HLS	5,987 (1.5×)	20	3.4	24.5	5.5
gesummv	RTL	8,111	20	15.4	42.7	4.7
	HLS	8,124 (1.0×)	20	2.6	15.2	2.9
jacobi_1d	RTL	944	19	12.3	56.9	6.2
	HLS	235 (4.0×)	20	5.4	31.9	5.0
jacobi_2d	RTL	16,813	19	27.9	55.2	10.7
	HLS	13,963 (1.2×)	20	3.2	19.1	3.7
lu	RTL	1,745	20	20.8	52.2	5.3
	HLS	2,013 (0.9×)	19	4.8	34.2	5.2
nussinov	RTL	15,289	16	21.5	62.0	5.9
	HLS	1,970 (7.8×)	20	5.6	28.9	5.2
seidel_2d	RTL	3,679	20	21.1	55.1	6.1
	HLS	3,356 (1.1×)	19	7.3	50.5	6.5
symm	RTL	149,285	5	45.2	88.3	6.6
	HLS	5,644 (26.5×)	17	6.0	28.2	5.3
syr2k	RTL	7,098	17	27.3	56.6	6.3
	HLS	6,499 (1.1×)	19	3.9	20.5	5.4
trisolv	RTL	7,327	17	30.6	57.7	5.1
	HLS	7,484 (1.0×)	18	8.7	32.1	4.2
two_mm	RTL	7,288	20	46.1	70.1	6.5
	HLS	5,340 (1.4×)	18	3.5	28.4	6.2

and operator binding. RTL-Agent can modify the generated Verilog and receives a separate read-only reference condensed from the Vivado UltraFast Design Methodology Guide (UG949) and Synthesis User Guide (UG901) [2, 4], covering pipelining, timing, resource inference and sharing, control logic, and FSM optimization. Both agents optimize the same 15 PolyBench kernels for 20 rounds, target an AMD Alveo U280 at 300 MHz, and preserve the baseline memory interface. Every candidate is checked against golden outputs before its cycle count is measured by simulation and its resources and timing are measured by Vivado out-of-context synthesis. We retain only timing-feasible points when constructing each Pareto frontier.

Experimental results. Figure 1 plots representative Pareto frontiers. HLS reaches beyond the RTL frontier on GEMM, Jacobi 1D, and SYMM, while the two approaches trace similar tradeoffs on SYR2K. Table 2 reports the lowest-cycle point on each Pareto frontier together with the cost of the complete search. A valid round produces a correct, measurable candidate. Total time includes both agent and evaluation wall time. Under the same iteration budget, HLS finds a faster design on 8 of 15 kernels and achieves a median speedup of 1.1× over RTL. Its search is also substantially cheaper: 69.2 million tokens and $407.3 in API cost, compared with 388.4 million tokens and $888.1 for RTL. HLS produces 286 valid rounds versus RTL's 261.

Reward hacking and verification cost. Passing a fixed testbench did not always preserve the intended floating-point semantics. In GESUMMV, the RTL agent exploited the workload's non-negative values by removing sign, cancellation, NaN, infinity, and denormal handling and by truncating multiplier operands. Although unspecified, this design passed the 10^{-3}-tolerance check and achieved 8,111 cycles with 408 LUTs, 324 FFs, and 2 DSPs. Replacing its arithmetic with the general Xilinx operators used by HLS raises usage to 640 LUTs, 895 FFs, and 10 DSPs, with nearly unchanged latency (8,121 cycles). Similarly, Trisolv uses a reciprocal specialized to finite, normalized, nonzero inputs. It accounts for approximately 694 of the 807-LUT gap and 531 of the 1,079-FF gap against a matched HLS architecture.

This reward hacking is difficult to detect at RTL without comprehensive corner-case tests or formal equivalence. Figure 2 shows the cost gap for an int16 matrix multiplier: verification takes 1.83 ms for HLS versus 9.97 s for RTL at 2×2 (5,450×), and 325 s versus 3 h 27 min at 256×256. The search costs above therefore exclude a potentially dominant verification expense, and the nominal RTL frontier is not a frontier of fully IEEE 754-compliant implementations. HLS does not eliminate specification gaming, but source-level types and operators narrow the semantics an agent can silently weaken.

Discussion. RTL wins seven kernels, but most HLS losses are near parity while its largest wins are substantial. With 5.6× fewer tokens and 2.2× lower API cost, HLS more efficiently converts a fixed round budget into valid architectural progress, although RTL remains competitive on cases such as SYR2K.

SYMM illustrates how HLS permutability benefits agentic exploration. Both agents identified the same opportunity to replace the triangular computation with a symmetric GEMM. The HLS agent expressed the transformation directly in the source program, obtained a functionally correct design in round 1, and refined it to its fastest point by round 8. The RTL agent attempted the same restructuring in rounds 7, 11, 15, and 16, but every candidate failed functional simulation. In generated RTL, operand roles and loop dependences are distributed across memories, pipeline registers, and control states. The failed candidates conflated distinct operand streams while reconstructing the matrix expression. A high level algorithm description keeps these relationships explicit, allowing an agent to make large changes to loop order, buffering, and parallelism while preserving functional intent. This case shows that HLS permutability directly improves the effectiveness of agentic design space exploration.

6.2 Guardrailed Agentic HLS

System design. Moving from fixed-loop autotuning toward L3 autonomy requires agents that can interpret tool feedback, prove correctness, and adapt designs for their system context. Figure 3 shows the target workflow. The system design combines guarded optimization, source-attributed feedback, formal equivalence, PyTorch ingestion, and Universal Chiplet Interconnect Express (UCIe) integration. An executable Python specification and a JSON interface contract remain immutable, while typed tools let the optimization agent edit the HLS kernel, evaluate it, checkpoint accepted candidates, and roll back failed changes. These components turn

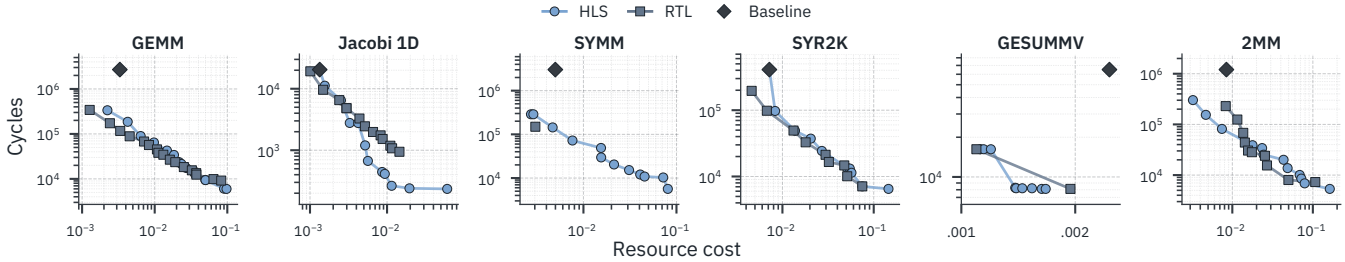


Figure 1: Cycle–resource Pareto frontiers for representative PolyBench kernels.

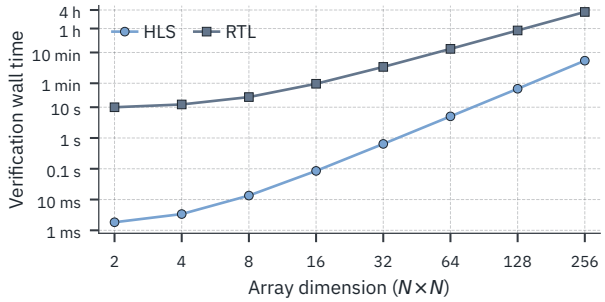


Figure 2: Equivalence-checking time for int16 output-stationary systolic matrix multipliers using PEQC for HLS and single-threaded Cadence Conformal LEC for RTL.

the feedback, verification, and composability gaps identified in Section 4 into explicit agent roles and promotion gates.

Input and output. The system takes an executable Python program that specifies the intended design behavior, with its inputs, outputs, and intermediate values annotated with data types. A separate specification defines the design constraints, such as the target device and available resources, and the optimization objectives, such as latency and throughput. Rather than returning a single implementation, the system produces a Pareto frontier of functionally valid design points that trade off the requested objectives while satisfying the constraints.

Mixed-fidelity performance feedback. The feedback agent combines a fast analytical performance model with accurate simulation and synthesis feedback. The optimization agent can use first-order architecture estimates to explore many candidates without generating or synthesizing each one, then submit promising designs to the tools for validation. The feedback agent maps reported latency, initiation interval, schedule violations, and resource use back to source regions, provides actionable bottleneck diagnoses, and uses these measurements to refine its analytical model. This loop makes inexpensive model-guided iteration and accurate tool-guided correction complementary rather than separate modes. This separation matters at scale: Table 3 estimates RTL simulation to be 819×–3074× slower than HLS C simulation. We calibrate XSim separately for each array shape using a 20,000-cycle active-stream run, measuring 219.5–284.9 simulated cycles per wall second; compilation, elaboration, and simulator startup are excluded. The gap motivates using analytical feedback and HLS C simulation for exploration while reserving RTL simulation for validation.

Formal verification and debugging. For transformations within its supported scope, a verification agent can invoke PEQC [20], an

Table 3: HLS–RTL simulation cost for systolic FlashAttention arrays.

Array	Points	RTL cycles (M)	HLS C sim. (min)	RTL sim. (h, est.)	Slowdown
16 × 16	1	356.8	6.8	347.9	3074×
32 × 16	1	207.4	6.4	254.1	2366×
16 × 32	1	193.8	7.2	236.2	1976×
32 × 32	10	77.1	7.2	97.6	819×

equivalence checker for HLS C/C++ programs, to prove semantic equivalence between the original HLS C/C++ kernel and the agent-transformed kernel across loop restructuring, bufferization, and FIFO-based dataflow changes. With formal verification, a passing proof becomes a promotion gate rather than a post-hoc check. On failure, the agent translates the reported semantic difference into a source-local explanation. When a concrete failing input can be derived, it also adds that input to the immutable regression suite before optimization continues. This guardrail directly addresses the reward hacking observed in the PolyBench study, while exploiting the lower verification cost available at the HLS level.

UCIe integration and workload-driven testing. An integration agent separates the compute contract from its physical protocol and generates a flit-aware die-to-die interface (FDI), the logical UCIe interface. It supports an arbitrary number of AXI-stream inputs and outputs and either handshake or free-running control. Using a raw-flit protocol, it packs stream data and control commands such as start into tagged flits, then specializes an FSM-based RTL wrapper to the kernel's FDI RX/TX channels. The agent generates a flit-level cocotb testbench and runs it with an FDI verifier, which may be agent-generated, pre-existing, or supplied by a commercial UCIe suite. The artifact validates the mainband adapter in RTL simulation and records HLS synthesis against the target; implementation and PHY integration require additional steps. A production flow must also establish the flit protocol and generate a matching adapter for the peer chiplet, or align with an existing peer's protocol.

For workload-driven testing, the agent extracts shapes and data types from the high-level program, materializes representative inputs and oracle outputs, and generates a shared harness for C and RTL simulation. The current experiment uses the executable Python FlashAttention specification; the same path can front a PyTorch model to capture real model inputs and reference outputs. This connects workload-level intent directly to compute and interface verification rather than relying on hand-written synthetic tests.

Experiment result. Figure 4 compares the agent-generated designs with the human-designed Fused Systolic Attention (FSA) baseline [15] using its open-source implementation [29]. FSA augments a parameterized FP16-multiply/FP32-accumulate systolic array so

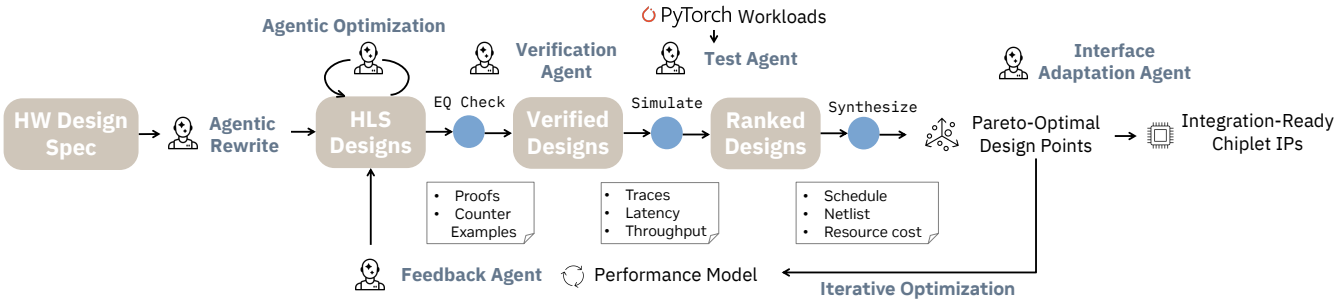


Figure 3: Overview of the guardrailed agentic HLS workflow.

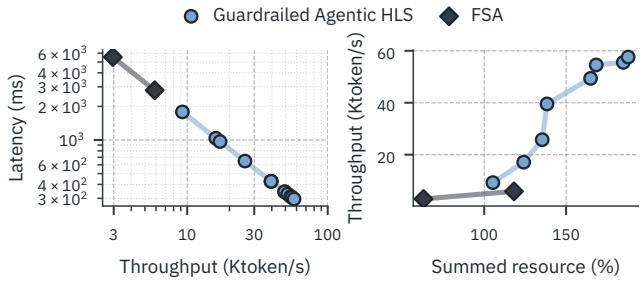


Figure 4: Guardrailed Agentic HLS versus the human-designed Fused Systolic Attention (FSA) baseline on FlashAttention ($N = 16{,}384$, $d = 128$).

that the QK and PV products and the intervening online-softmax operations execute within the same array, avoiding transfers to an external vector unit. Its SystolicAttention schedule overlaps these operations at element granularity while preserving their floating-point order. We evaluate all FSA design points, but only the 128×4 and 128×8 configurations fit within the U280 resource constraints, exposing the architecture's inefficient resource scaling on this device.

We use non-causal FP16 FlashAttention with batch size one, one head, $N = 16{,}384$, and $d = d_v = 128$ on a U280. Agent designs target 200 MHz, whereas FSA can be clocked at only 50 MHz. The left panel plots all 13 agent points, spanning 9.2–57.6 Ktoken/s and 296–1,784 ms; the right retains their resource–throughput Pareto frontier. Resource cost sums LUT, FF, DSP, BRAM, and URAM utilization. At the closest resource-matched points, the agent reaches 17.1 Ktoken/s at 124.1%, versus FSA's 5.9 Ktoken/s at 118.2%, a 2.9× improvement.

The agent explores neighbor-connected arrays with square and rectangular shapes: 16×16, 16×32, 32×16, and 32×32. Each PE represents a query–key pair and accumulates the 128 feature products over time. K/V tiles reside in banked URAM, while online softmax and PV form a separate stage whose parallelism is tuned independently. FSA instead fixes 128 rows to the head dimension and equips every PE with flexible FP16/FP32 arithmetic, exponential, and routing control; increasing query parallelism therefore replicates LUT-heavy PEs, with the 128×8 design consuming 95% of U280 LUTs. The agent designs distribute work across DSPs and URAM, use shorter specialized datapaths, and reach 200 MHz. This 4× clock advantage is the primary source of lower latency and higher throughput, while rectangular tiling and independent PV sizing improve resource efficiency. Across 35 evaluations, the exploration consumes 16 million model tokens and 9.3 hours of wall time. In an RTL protocol evaluation, the generated FDI adapter passes all 15 checker tests and all 10 workload tests, preserving byte-identical transport across seven delivery conditions; PHY integration and conformance signoff are out of scope.

Discussion. This workflow makes the two-way relationship between HLS and agents concrete. HLS exposes loops, types, dataflow, and memory structure so the optimization agent can reshape tiling, array geometry, and banking rather than merely tune pragmas. Conversely, specialized agents address the feedback, interface, and verification limitations identified in Section 4. The feedback agent fuses analytical, simulation, and synthesis signals into source-level guidance. Because HLS cannot express the customized UCIe FDI protocol, the integration agent writes and tests an RTL wrapper that adapts AXI streams to the flit-level interface. The verification agent connects the high-level specification, HLS transformations, workload tests, and RTL interface checks. Independent verification remains essential: an LLM can exploit weak tests by substituting fixed-point internal arithmetic, but C simulation and HLS-level equivalence checking catch this without RTL co-simulation. Thus, HLS supplies a tractable optimization representation, while agents supply feedback interpretation, interface adaptation, and cross-level verification missing from current HLS flows.

7 Conclusion

We argue that HLS will remain a valuable layer in the agentic era, supporting faster iteration, portability, and design exploration. We highlight performance feedback, rigid interfaces, and debuggability as key gaps in today's HLS workflows where agents can help, and propose a taxonomy of automation levels to structure the space. Mixed-fidelity feedback is especially important because it lets agents act on incomplete scheduling evidence while preserving uncertainty and source-level context. We hope that this paper encourages joint progress between the HLS and AI communities.

Acknowledgments

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, NSF Award #2403135, and a research gift from AMD.

References

[1] Stefan Abi-Karam and Cong Hao. 2025. HLS-Eval: A Benchmark and Framework for Evaluating LLMs on High-Level Synthesis Design Tasks. In *2025 IEEE International Conference on LLM-Aided Design*. IEEE, Piscataway, NJ, USA, 219–226. doi:10.1109/ICLAD65226.2025.00021

[2] AMD. 2023. *UltraFast Design Methodology Guide for FPGAs and SoCs*. AMD. https://docs.amd.com/r/2023.1-English/ug949-vivado-design-methodology

[3] AMD. 2023. *Vitis High-Level Synthesis User Guide*. AMD. https://docs.amd.com/r/2023.2-English/ug1399-vitis-hls

[4] AMD. 2023. *Vivado Design Suite User Guide: Synthesis*. AMD. https://docs.amd.com/r/2023.2-English/ug901-vivado-synthesis

[5] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD*. IEEE, Piscataway, NJ, USA, 1–6. doi:10.1109/MLCAD58807.2023.10299874

[6] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 593–620. doi:10.1145/3656401

[7] Luca Collini, Siddharth Garg, and Ramesh Karri. 2025. C2HLSC: Leveraging Large Language Models to Bridge the Software-to-Hardware Design Gap. *ACM Transactions on Design Automation of Electronic Systems* 30, 6, Article 96 (2025), 24 pages. doi:10.1145/3734524

[8] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems* 15, 4, Article 51 (2022), 42 pages. doi:10.1145/3530775

[9] Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2025. VerilogCoder: Autonomous Verilog Coding Agents with Graph-Based Planning and Abstract Syntax Tree (AST)-Based Waveform Tracing Tool. *Proceedings of the AAAI Conference on Artificial Intelligence* 39, 1 (2025), 300–307. doi:10.1609/AAAI.V39I1.32007

[10] Florent Kermarrec, Sébastien Bourdeauducq, Jean-Christophe Le Lann, and Hannah Badier. 2019. LiteX: An Open-Source SoC Builder and Library Based on Migen Python DSL. OSDA 2019: Workshop on Open-Source Design Automation, co-located with DATE 2019. arXiv:2005.02506 [cs.AR] https://arxiv.org/abs/2005.02506

[11] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 296–311. doi:10.1145/3192366.3192379

[12] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. 2008. Validating High-Level Synthesis. In *Computer Aided Verification (Lecture Notes in Computer Science, Vol. 5123)*. Springer, Berlin, Heidelberg, 459–472. doi:10.1007/978-3-540-70545-1_44

[13] Runkai Li, Jia Xiong, Xiuyuan He, Jieru Zhao, Jiaqi Lv, Haowen Fang, Lei Qi, and Xi Wang. 2026. ChatHLS: Towards Systematic Design Automation and Optimization for High-Level Synthesis. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, San Diego, California, United States, 20996–21015. doi:10.18653/v1/2026.acl-long.962

[14] Yuchao Liao, Tosiron Adegbija, and Roman Lysecky. 2024. Are LLMs Any Good for High-Level Synthesis?. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. Association for Computing Machinery, New York, NY, USA, Article 29, 8 pages. doi:10.1145/3676536.3699507

[15] Jiawei Lin, Yuanlong Li, Guokai Chen, and Thomas Bourgeat. 2025. SystolicAttention: Fusing FlashAttention within a Single Systolic Array. arXiv preprint. arXiv:2507.11331 [cs.AR] doi:10.48550/arXiv.2507.11331

[16] Nowfel Mashnoor, Mohammad Akyash, Hadi Kamali, and Kimia Azar. 2025. TimelyHLS: LLM-Based Timing-Aware and Architecture-Specific FPGA HLS Optimization. In *2025 IEEE International Conference on Omni-layer Intelligent Systems*. IEEE, Piscataway, NJ, USA, 1–6. doi:10.1109/COINS65080.2025.11125726

[17] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable Accelerator Design with Time-Sensitive Affine Types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 393–407. doi:10.1145/3385412.3385974

[18] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 343–367. doi:10.1145/3591234

[19] Ali Emre Oztas and Mahdi Jelodari. 2024. Agentic-HLS: An Agentic Reasoning Based High-Level Synthesis System Using Large Language Models. AI for EDA Workshop, co-located with NeurIPS. doi:10.48550/arXiv.2412.01604

[20] Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodríguez, and Zhiru Zhang. 2024. Formal Verification of Source-to-Source Transformations for HLS. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, USA, 97–107. doi:10.1145/3626202.3637563

[21] Rishov Sarkar and Cong Hao. 2025. OmniSim: Simulating Hardware with C Speed and RTL Accuracy for High-Level Synthesis Designs. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, USA, 1334–1346. doi:10.1145/3725843.3756033

[22] Rishov Sarkar, Rachel Paul, and Cong Callie Hao. 2024. LightningSimV2: Faster and Scalable Simulation for High-Level Synthesis via Graph Compilation and Optimization. In *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, Piscataway, NJ, USA, 104–114. doi:10.1109/FCCM60383.2024.00021

[23] Seyed Arash Sheikholeslam and Andre Ivanov. 2024. SynthAI: A Multi-Agent Generative AI Framework for Automated Modular HLS Design Generation. arXiv preprint. doi:10.48550/arXiv.2405.16072

[24] Siemens EDA. n.d.. Catapult High-Level Synthesis. Accessed August 24, 2026. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/

[25] Nitish Kumar Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David H. Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, Geoff Lowney, Adam Herr, Christopher J. Hughes, Timothy G. Mattson, and Pradeep Dubey. 2019. T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, Piscataway, NJ, USA, 181–189. doi:10.1109/FCCM.2019.00033

[26] Qi Sun, Tinghuan Chen, Siting Liu, Jin Miao, Jianli Chen, Hao Yu, and Bei Yu. 2021. Correlated Multi-Objective Multi-Fidelity Optimization for HLS Directives Design. In *2021 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, Piscataway, NJ, USA, 46–51. doi:10.23919/DATE51398.2021.9474241

[27] Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. 2023. AutoChip: Automating HDL Generation Using LLM Feedback. arXiv preprint. arXiv:2311.04887 [cs.PL] doi:10.48550/arXiv.2311.04887

[28] Yunda Tsai, Mingjie Liu, and Haoxing Ren. 2024. RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Model. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. Association for Computing Machinery, New York, NY, USA, Article 53, 6 pages. doi:10.1145/3649329.3657353

[29] VCA, EPFL. 2025. FSA: Fusing FlashAttention within a Single Systolic Array. Open-source artifact. Accessed August 23, 2026. https://github.com/VCA-EPFL/FSA

[30] Hanyu Wang, Xinrui Wu, Zijian Ding, Su Zheng, Chengyue Wang, Neha Prakriya, Tony Nowatzki, Yizhou Sun, and Jason Cong. 2025. LLM-DSE: Searching Accelerator Parameters with LLM Agents. arXiv preprint. doi:10.48550/arXiv.2505.12188

[31] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. 2022. HeteroFlow: An Accelerator Programming Model with Decoupled Data Placement for Software-Defined FPGAs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, USA, 78–88. doi:10.1145/3490422.3502369

[32] Chenwei Xiong, Cheng Liu, Huawei Li, and Xiaowei Li. 2024. HLSPilot: LLM-Based High-Level Synthesis. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. Association for Computing Machinery, New York, NY, USA, Article 226, 9 pages. doi:10.1145/3676536.3676781

[33] Haocheng Xu, Haotian Hu, and Sitao Huang. 2024. Optimizing High-Level Synthesis Designs with Retrieval-Augmented Large Language Models. In *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, Piscataway, NJ, USA, 1–5. doi:10.1109/LAD62341.2024.10691855

[34] Kangwei Xu, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, Ulf Schlichtmann, and Bing Li. 2024. Automated C/C++ Program Repair for High-Level Synthesis via Large Language Models. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. Association for Computing Machinery, New York, NY, USA, Article 15, 9 pages. doi:10.1145/3670474.3685953

[35] Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine Lin, Yong Liu, and Haoxing Ren. 2025. Spec2RTL-Agent: Automated Hardware Code Generation from Complex Specifications Using LLM Agent Systems. In *2025 IEEE International Conference on LLM-Aided Design*. IEEE, Piscataway, NJ, USA, 37–43. doi:10.1109/ICLAD65226.2025.00013

[36] Niansong Zhang, Chenhui Deng, Johannes Maximilian Kuehn, Chia-Tung Ho, Cunxi Yu, Zhiru Zhang, and Haoxing Ren. 2025. ASPEN: LLM-Guided E-Graph Rewriting for RTL Datapath Optimization. In *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD*. IEEE, Piscataway, NJ, USA, 1–9. doi:10.1109/MLCAD65511.2025.11189222

[37] Chenfeng Zhao, Clayton J. Faber, Roger D. Chamberlain, and Xuan Zhang. 2025. HLPerf: Demystifying the Performance of HLS-Based Graph Neural Networks with Dataflow Architectures. *ACM Transactions on Reconfigurable Technology and Systems* 18, 1, Article 2 (2025), 26 pages. doi:10.1145/3655627