

DAPO: Design Structure-Aware Pass Ordering for HLS via Contrastive and Reinforcement Learning

Jinming Ge[†], Linfeng Du[†], Likith Anaparty[‡], Shangkun Li[†], Tingyuan Liang[†], Afzal Ahmad[†],

Vivek Chaturvedi[‡], Sharad Sinha[§], Zhiyao Xie[†], Jiang Xu[¶], Wei Zhang^{†,||}

[†]The Hong Kong University of Science and Technology [‡]Indian Institute of Technology (Palakkad)

[§]Indian Institute of Technology (Goa) [¶]The Hong Kong University of Science and Technology (Guangzhou)

^{||}Guangzhou HKUST Fok Ying Tung Research Institute

jgeab@connect.ust.hk, linfeng.du@connect.ust.hk, wei.zhang@ust.hk

Abstract—High-Level Synthesis (HLS) tools are widely adopted in FPGA-based domain-specific accelerator design. However, existing tools rely on fixed optimization strategies inherited from software compilations, limiting their effectiveness. Tailoring optimization strategies to specific designs requires deep semantic understanding, accurate hardware metric estimation, and advanced search algorithms—capabilities that current approaches lack.

We propose DAPO, a design structure-aware pass ordering framework that extracts program semantics from control and data flow graphs, employs contrastive learning to generate rich embeddings, and leverages an analytical model for accurate hardware metric estimation. These components jointly guide a reinforcement learning agent to discover design-specific optimization strategies. Evaluations on standard HLS benchmarks demonstrate that our end-to-end flow delivers $1.67\times$ speedup on pragma-free designs and a $2.36\times$ speedup on designs with pragmas over Vitis HLS with comparable resource usage.

Index Terms—High-level Synthesis, Pass Ordering, Reinforcement Learning, Graph Contrastive Learning

I. INTRODUCTION

The computing landscape is rapidly shifting toward specialized hardware acceleration, with Field-Programmable Gate Arrays (FPGAs) emerging as essential platforms for domain-specific computing. High-level synthesis (HLS) stands at the forefront of this evolution, enabling software designers to deploy specialized circuits without mastering hardware description languages. This democratization of hardware design [1] has expanded FPGA adoption across diverse domains, from machine learning to scientific computing.

Despite recent advances in HLS frameworks from both academia [2], [3] and industry [4], these tools remain hampered by fixed optimization strategies inherited from software compilation [5]. Most research addresses HLS optimization peripherally through pragma configuration [6], [7], which provides high-level guidance but ultimately relies on the compiler’s ability to interpret and apply these directives efficiently.

However, the HLS compilation flow in which pragma-assigned transformations and general transformations (passes) are integrated remains largely unexplored for achieving design-specific optimization. Consequently, even designs with well-tuned pragma configurations cannot yield optimal results when processed by rigid optimization sequences that do not adapt to the diverse needs of various applications. This compiler-level

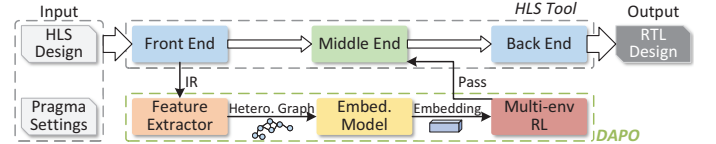


Fig. 1. The DAPO framework utilizes compilation IR from the HLS front end and generates domain-specific pass sequences for the HLS middle end.

limitation persists regardless of pragma configurations—since pragma customization and compiler transformations operate at different abstraction levels, leaving the compiler’s predetermined optimization orders invariant across designs.

Customizing optimization sequences for specific designs—commonly known as the **pass ordering problem**—faces three key problems in the HLS domain: (1) the design-awareness gap between program representation and pass selection, (2) the combinatorial complexity of searching vast non-commutative transformation spaces, and (3) the evaluation difficulty caused by lengthy commercial HLS compilations and their black-box nature. These challenges have prevented existing approaches—whether heuristic-based [8] or learning-based [9]—from effectively balancing optimization efficiency with adaptation to diverse designs.

To overcome these challenges, we present the design structure-aware pass ordering (DAPO) framework¹, a comprehensive solution that integrates graph contrastive learning with reinforcement learning (RL). As illustrated in Fig. 1, DAPO implements a three-stage approach: (1) DAPO first extracts heterogeneous graphs from compilation intermediate representations (IRs), which enables a thorough understanding of the intricate and nuanced characteristics in HLS designs, surpassing traditional homogeneous representations [6], [10] that fail to distinguish program semantics and structural dependencies. (2) DAPO then leverages a novel contrastive learning technique to create rich program embeddings. This self-supervised approach is inherently more amenable to pass ordering tasks than conventional supervised methods by learning structural similarity patterns between programs without requiring pre-labeled optimal pass sequences, which are notoriously difficult and expensive to obtain across diverse HLS designs [11]. (3) Finally, by leveraging these expressive embeddings within an RL model, DAPO achieves superior inference capabilities, significantly reducing search time compared to heuristic al-

J. Ge and L. Du contributed equally to this work. Corresponding authors: Linfeng Du (linfeng.du@connect.ust.hk) and Wei Zhang (wei.zhang@ust.hk).

¹DAPO is open-sourced at <https://github.com/gjskywalker/DAPO>

gorithms while achieving better generalization than existing learning-based methods [9]. With the framework in place, we highlight our main contributions as follows:

- We present the first framework that leverages advanced program structure learning to achieve generalizable pass ordering across diverse FPGA HLS design domains, fundamentally reshaping how compiler optimizations are applied to hardware synthesis.
- We design a novel graph-based program representation that captures the intricate relationship between program structure and optimal pass selection, and enhance it with specialized contrastive learning techniques that enable effective knowledge transfer across diverse designs, demonstrating superior effectiveness over existing academic approaches.
- We conduct extensive experiments on classic HLS benchmarks spanning multiple application domains, with and without pragma specifications, and show improvements of $1.67\times$ over Vitis HLS on designs without pragmas and $2.36\times$ on designs with pragmas.

II. BACKGROUND AND PRELIMINARIES

A. Pass Ordering Challenges in HLS



Fig. 2. HLS Middle End Optimization Pipeline.

HLS systems employ a sophisticated middle end transformation phase that orchestrates two distinct categories of optimization passes: (1) general-purpose passes inherited from software compilation (e.g., *dead code elimination*, *constant propagation*), and (2) pragma-specific passes tailored for FPGA implementation (e.g., *loop pipelining*, *array partitioning*, *loop unroll*). The general optimization passes are typically interleaved with pragma optimizations as shown in Fig. 2, which depicts the optimization phase of the Vitis HLS 2023.2 middle end (a.g.1 $\rightarrow$ a.o.1). It creates a complex optimization landscape with two critical interdependencies: (1) pragma-specific passes may introduce redundant computations requiring cleanup by general passes, while their effectiveness also hinges on prior general optimizations being correctly applied; (2) the numerous general-purpose passes form intricate dependency chains where one pass may enable, amplify, or diminish subsequent transformations. Consequently, given the topological order of pragma-specific passes, DAPO strategically focus on optimizing the order of general-purpose passes.

Another key challenge in the HLS domain is the abstraction gap—the disconnect between the IR-level transformations and their hardware implementation effects. Optimization passes operate on abstract code structures, yet their effectiveness must be measured through concrete hardware metrics like resource usage and performance. Current pass ordering approaches fail to bridge this gap between transformation space and evaluation metrics, often resulting in suboptimal hardware designs. To address this challenge, we build upon and enhance an analytical model [12] to accurately evaluate hardware metrics at the IR level, while avoiding the prolonged HLS synthesis.

B. Foundations of HLS Program Representation Learning

A program can be formally represented as a graph structure $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{A}_u, \mathcal{A}_{uv})$, where $\mathcal{V}$ is the node set, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the edge set. $\mathcal{A}_u$ represents node attributes encoding semantic information, and $\mathcal{A}_{uv}$ represents edge attributes encoding data flow and control flow information. For HLS applications, this homogeneous representation proves insufficient as it fails to capture program hierarchy and diverse relationship types. Instead, we employ a heterogeneous graph that incorporates typed nodes and edges to emphasize different semantic relationships within program structures.

Meanwhile, unlike traditional supervised learning approaches for program representation suffering from scarce labeled data, contrastive learning offers a promising solution for program representation by analyzing structural similarities between programs. This approach maximizes the agreement between similar structures while minimizing the agreement between dissimilar ones through a contrastive loss function.

Central to effective contrastive learning is defining similarity between program graphs. We employ Graph Edit Distance (GED) as our primary metric, which quantifies the minimum cost of transforming one graph into another through a series of edit operations:

$$GED(G_1, G_2) = \min_{(e_1, \dots, e_n) \in \mathcal{P}(G_1, G_2)} \sum_{i=1}^n c(e_i) \quad (1)$$

where $\mathcal{P}(G_1, G_2)$ denotes all possible edit paths and $c(e_i)$ represents the cost of each edit operation.

By combining GNNs with contrastive learning principles, we establish a self-supervised framework for learning program representations that explicitly capture structural characteristics relevant to pass ordering. This approach enables our system to identify pattern similarities across diverse program domains without requiring human annotation, facilitating transfer learning and cross-domain generalization in optimization.

III. MOTIVATING CASE STUDIES

To demonstrate the necessity for design structure-aware pass ordering in HLS optimization and explain the relation between general-purpose passes and pragma directives, we analyze two representative cases that illustrate how program structure fundamentally influences optimization effectiveness. Although most middle end pass effects are orthogonal to the front end design, these two examples we present here are specifically crafted to demonstrate not only the IR structure before and after pass optimization, but also the equivalent HLS design corresponding to the optimized IR², helping users better understand the effects of pass ordering. These examples from different application domains reveal optimization opportunities that conventional fixed-sequence approaches consistently miss.

A. Data Flow Transformation

Our first case examines a classic FPGA design scenario: two-array multiplication with *loop unroll* pragma, as shown in Fig. 3 (top row). However, the inner loop bound (1482) is not divisible

²This step does not exist in the actual DAPO flow.

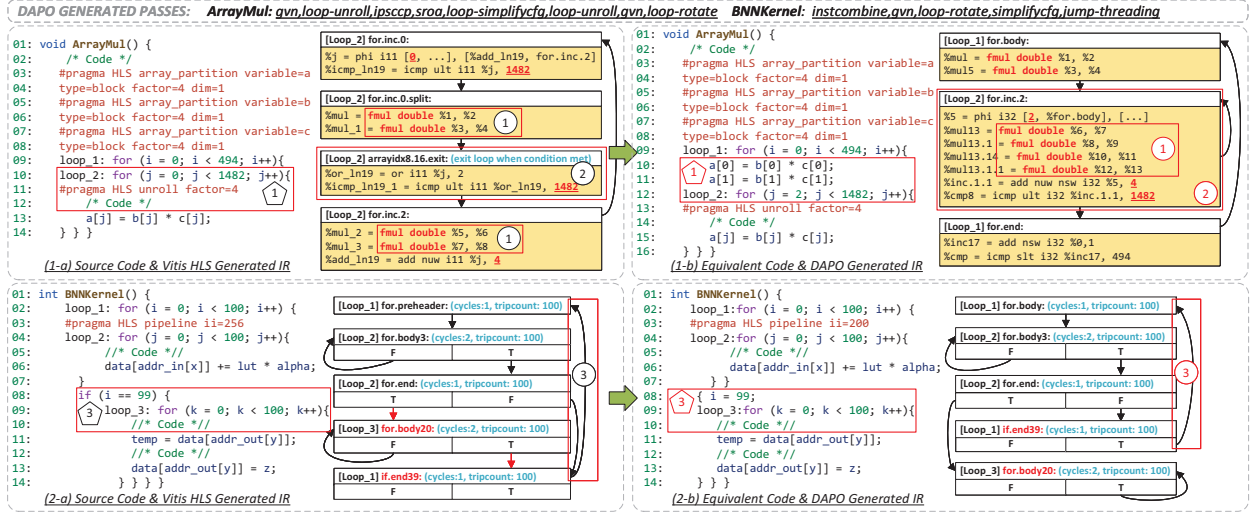


Fig. 3. Top: data flow transformation enables enhanced loop unrolling. Bottom: control flow restructuring improves pipeline efficiency. Optimized elements are highlighted with red rectangles and numbered with pentagonal labels in C++ code and circular labels in IR code.

by the assigned unroll factor (4) (①). In the conventional HLS compilation flow, this mismatch generates an additional block for loop termination condition checking (②) and fragments the inner loop body (①), introducing extra latency.

In contrast, DAPO identifies a critical transformation opportunity: partially hoisting the multiplication operation from the inner loop to the outer loop. This transformation fundamentally restructures the data flow, reducing the inner loop trip count to 1480 (①), which is divisible by 4 and supports the desired parallelism (①).

Specifically, DAPO employs two *loop-unroll* passes: the first creates the imperfect structure with the remainder iterations, while the subsequent *ipscpp* (interprocedural sparse conditional propagation) and *loop-simplifycfg* passes peel off the first 2 iterations, enabling the second *loop-unroll* to achieve the perfect 4-way unrolling on the remaining iterations (②).

B. Control Flow Restructuring

Our second case studies a binary neural network from HLS-Benchmarks [13]. As illustrated in Fig. 3 (bottom row), Loop 3 (③) executes conditionally—only when the outer loop counter reaches 99. The fixed optimization sequence employed by conventional HLS fails to recognize this pattern and incorrectly accounts for Loop 3’s latency in every iteration (③) when calculating Loop 1’s possible initiation interval (II).

This structural misinterpretation significantly impacts performance. Given that Loops 2 and 3 have equivalent latency profiles, this oversight substantially affects Loop 1’s latency estimation. Furthermore, read-after-write and write-after-write dependencies between Loops 2 and 3 introduce pipeline hazards that prevent achieving optimal initiation intervals.

DAPO leverages a coordinated pass sequence for systematic optimization. Initially, *instcombine* and *gvn* (global value numbering) [14] perform pattern analysis to recognize redundant execution patterns, enabling *loop-rotate* to expose the structural independence between the if-condition and Loop 3. Subsequently, *simplifycfg* and *jump-threading* eliminate the conditional structure from Loop 1 (③), achieving an optimal II of 200 and improving performance by 22%.

C. The Need for Heterogeneous Graphs

These two case studies not only highlight the importance of customized pass ordering given the significant differences between their optimal pass sequences, but also reveal a critical limitation in conventional program representations: homogeneous graphs that combine data and control flow features into a unified structure inadequately capture the distinct optimization strategies required for different program components. The optimization approaches effective for data flow structures often differ significantly from those appropriate for control flow structures, necessitating separate modeling and analysis.

This observation motivates our design of a heterogeneous graph representation with dedicated subgraphs for different logical relations. By recognizing the distinct optimization patterns applicable to data versus control flow structures, our approach empowers more precise targeting of transformations that maximize pragma effectiveness, delivering performance improvements beyond what conventional fixed-sequence approaches can achieve.

IV. THE DAPO FRAMEWORK

A. System Architecture

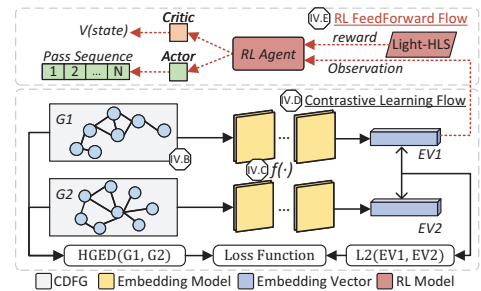


Fig. 4. The DAPO Framework. Graph Contrastive Learning (bottom) extracts design embeddings utilized by the Reinforcement Learning agent (top).

DAPO employs a two-phase training strategy that separates representation learning from policy optimization, as illustrated in Fig. 4. In the first phase (lower portion), a specialized embedding model comprising three Relational Graph Convolutional Network (RGCN) layers [15] followed by a Multi-Layer

Perceptron (MLP) undergoes contrastive pre-training to learn discriminative program embeddings. In the second phase (upper portion), this pre-trained model serves as a feature extractor within a reinforcement learning framework, transforming program graphs into vector representations that enable the RL agent to recognize structural patterns and generalize effective pass ordering strategies across diverse designs.

This decoupled architecture addresses a fundamental challenge in compiler optimization: learning transferable knowledge about program structures that influence pass effectiveness. By separating these concerns, DAPO achieves better cross-domain generalization and more efficient exploration of the pass ordering space.

B. Heterogeneous Graph Representation

A key innovation in DAPO is our specialized heterogeneous graph representation designed specifically for pass ordering. As shown in Fig. 5, our representation explicitly separates three critical aspects of program structure: control flow (captured through connections between basic blocks, with loop blocks highlighted in yellow), data flow (represented by connections between instruction nodes), and hierarchical relationships (modeled via affiliation edges connecting instructions to blocks and blocks to functions).

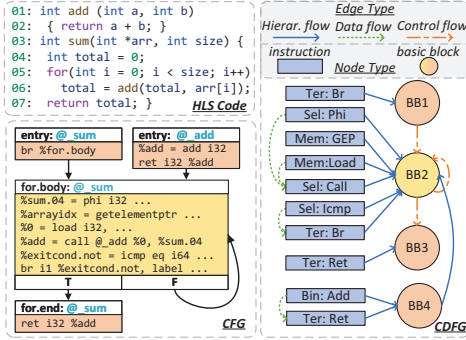


Fig. 5. Heterogeneous graph construction from HLS source and IR.

This representation offers significant advantages over conventional homogeneous approaches [10], [16]. By operating at the IR level where compiler passes directly operate, we accurately capture the elements that passes transform. We assign node attributes following LLVM's instruction classification standard, aligning with specific transformation pass targets. By explicitly modeling the IR hierarchy, we enable recognition of complex optimization opportunities spanning multiple program constructs. This specialized representation forms the foundation for identifying structural patterns that influence pass effectiveness across diverse program domains.

C. Structure-Aware Embedding

Our embedding model is specifically designed to process the diverse relationship types present in program graphs. The RGCN model explicitly handles different relationships through dedicated parameter sets, with message passing defined as:

$$\mathbf{h}_u^{(k)} = \delta \left(\sum_{r \in R} \sum_{v \in \mathcal{N}_r(u)} \frac{1}{c_{v,r}} \mathbf{W}_r^{(k-1)} \mathbf{h}_v^{(k-1)} + \mathbf{W}_0^{(k-1)} \mathbf{h}_u^{(k-1)} \right) \quad (2)$$

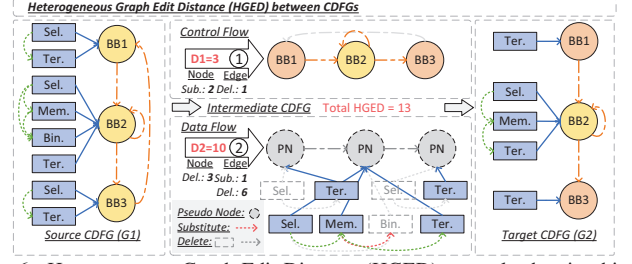


Fig. 6. Heterogeneous Graph Edit Distance (HGED) example showing hierarchical similarity computation for source graph (G1) and target graph (G2).

where $\mathbf{W}_r^{(k-1)}$ represents relationship-specific weights, $\mathbf{W}_0^{(k-1)}$ is the self-connection weights, $c_{v,r}$ is a normalization constant, and $\mathbf{h}_u^{(k-1)}$ represents node features from the previous layer.

This relational learning process first decouples the heterogeneous graph into three connection-specific substructures (data flow, control flow, and hierarchy). Dedicated neural modules then extract features from each substructure's topological patterns before merging them into unified program embeddings through attention-based composition. This captures the complex interplay between different program relationships that influence pass effectiveness.

D. Contrastive Learning for Knowledge Transfer

To enable effective knowledge transfer across program domains, we employ contrastive learning with a novel similarity metric designed for heterogeneous program graphs. Using a dataset of 9,000 distinct IRs from 90 different HLS designs (with an average of 100 randomly generated pass sequences per design), we train our embedding model to recognize structural similarities relevant to pass ordering.

The core innovation is our Heterogeneous Graph Edit Distance (HGED) algorithm, which extends traditional edit distance calculations to account for program hierarchy. HGED operates in two sequential steps: firstly, analyzing the control structure to identify complex constructs like loops and function calls, then evaluating data flow and instruction-block affiliations to analyze computational patterns and dependencies. As shown in Fig. 6, source graph (G1) and target graph (G2) represent two HLS designs: 2D array-constant multiplication and 1D array initialization, respectively. Initially, HGED transforms the nested loop into a single-level loop at the control flow level (①). Following this, HGED analyzes their data flow differences and removes constant multiplication operations (*Bin.* node) and redundant conditional statements (*Sel.* node) (②).

By aligning the embedding space with these structural similarities, we enable effective knowledge transfer across different program domains without requiring explicit human annotation or extensive labeled examples.

E. Reinforcement Learning for Pass Sequence Optimization

Our framework employs multi-environment reinforcement learning to discover effective pass sequences. Program embeddings from the pre-trained model form the observation space, while the action space consists of 45 carefully selected LLVM transformation passes organized into six functional categories (Table I). The multi-RL environment comprises 90 classic HLS designs with varying pragma settings. Performance

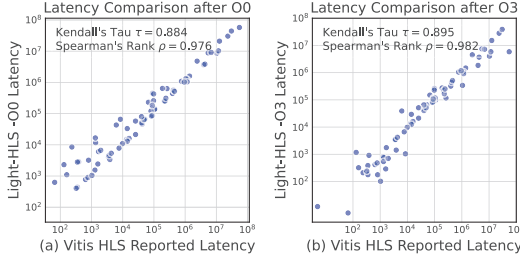


Fig. 7. Latency estimation correlation between Light-HLS and Vitis HLS for programs at different optimization levels.

improvement between consecutive passes adopted serves as the reward function, normalized by the current best performance to stabilize training across environments. Users can also select different HLS QoRs as rewards based on their optimization objectives, requiring minimal modifications to the RL settings.

For policy optimization, we employ Proximal Policy Optimization (PPO) [17] with an actor-critic architecture, where the actor suggests passes based on the current policy, and the critic evaluates suggestions using performance feedback.

TABLE I
LLVM TRANSFORM PASSES IN DAPO'S ACTION SPACE

Category	Passes
Control Flow	simplifycfg, jump-threading, chr, speculative-execution, scp
Instruction	early-cse, vector-combine, bdce, adce, reassociate, instsimplify, aggressive-instcombine, instcombine
Variable	sroa, gvn, float2int, globalopt, typepromotion, argpromotion
Loop	loop-simplifycfg, loop-simplify, lcssa, loop-rotate, loop-sink, loop-idiom, indvars, loop-deletion, licm
Function/Call	coro-early, callsite-splitting, coro-split, wholeprogramdevirt
Memory Access	dse, mem2reg, mldst-motion, loop-load-elim, memcpypopt

* 45 passes are considered in this work, and 37 are listed here.

We enhance Light-HLS [12] to enable the fast and accurate evaluation of hardware implementation metrics from the IR level, and also develop an extendable instruction library to support new instruction types introduced by different passes.

V. EXPERIMENTAL EVALUATION

This section evaluates DAPO's effectiveness across diverse HLS applications. We implemented our framework using PyTorch Geometric [18] and RLlib [19], with experiments conducted on a system equipped with an Intel i9-14900 CPU and NVIDIA A100 GPU. We used Vitis HLS 2023.2 middle end as our primary baseline to isolate the impact of pass ordering optimization. All HLS designs target the AMD Alveo U280 board with a default clock frequency of 100MHz.

A. Experimental Setup

Our evaluation employed 100 representative HLS designs from five established benchmark suites: PolyBench [20], Mach-Suite [21], Rosetta [22], CHStone [23], and HLS-benchmarks [13]. This diverse collection spans multiple application domains and includes designs both with and without pragma annotations.

To ensure reliable performance estimation for our reinforcement learning approach, we validated Light-HLS's latency predictions against hardware co-simulation results from Vitis HLS. As shown in Fig. 7, the strong Kendall's tau and Spearman's rank correlation coefficients confirm that our estimator effectively captures relative performance differences between designs, making it suitable for guiding optimization decisions.

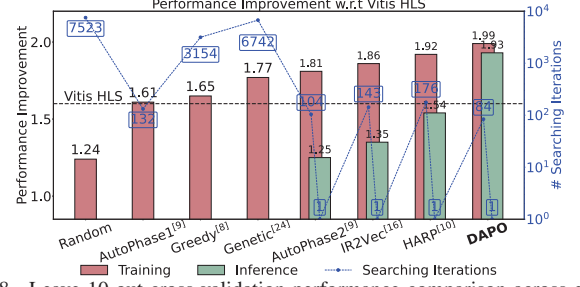


Fig. 8. Leave-10-out cross-validation performance comparison across embedding approaches and optimization methods.

B. Generalization and Real-World Performance

To evaluate generalization capabilities, we implemented leave-10-out cross-validation across our benchmark suite. Given the intensive searching iterations involved in these experiments, we employed Light-HLS to provide performance results for efficient evaluation. Fig. 8 compares five embedding approaches within our RL framework against three classical optimization heuristics (greedy [8], genetic [24], and random search) and the Vitis HLS middle end. For fair comparison with the Vitis HLS middle end, we extracted the final optimized IR files from its middle end optimization phase and evaluated their performance using Light-HLS.

Among the methods compared, AutoPhase1 [9] uses action history as its observation space; however, since action history depends on the training process, this approach naturally lacks inference capability, while AutoPhase2 relies on numeric program features. HARP [10] and IR2Vec [16] represent SOTA embedding methods for HLS and general C/C++ programs.

The results reveal several key insights: (1) DAPO achieves superior performance against both training and search-based exploration methods. (2) This strong inference capability demonstrates DAPO's excellent generalization ability, confirming the effectiveness of our heterogeneous representation and contrastive learning approach. (3) Outperforming general-purpose embedding methods (IR2Vec & HARP) underscores the necessity of task-specific embeddings for pass ordering.

TABLE II
QOR COMPARISON ON CLASSIC HLS DESIGNS

Benchmark	Without Pragma			With Pragma		
	Vitis HLS/DAPO			Vitis HLS/DAPO		
	Cycles (K)	LUT (K)	DSP	Cycles (K)	LUT (K)	DSP
substring	262/114	0.57/0.63	0/0	240/59	1.17/0.72	0/0
bnnkernel	30.3/30.5	0.21/0.21	3/3	26.4/20.3	8.51/3.45	3/3
getTanh	185/137	1.06/1.25	12/12	184/136	1.9/2.4	12/12
atax	24.5/8.2	0.96/0.94	11/11	22.9/6.2	1.67/1.05	19/11
crs	6589/6581	0.85/1.16	11/19	3661/3291	0.95/1.30	11/19
vecnormtrans	28.6/12.3	1.37/1.24	8/5	19.7/3.3	2.49/2.36	10/7
Norm. Geom. Mean	1×/1.67×	1×/1.08×	1×/1.02×	1×/2.36×	1×/0.80×	1×/0.93×

We further integrated DAPO into Vitis HLS and evaluated its inference ability on six HLS designs where Vitis HLS fails to achieve optimal performance, demonstrating DAPO's substantial effectiveness. Each design is assigned with appropriate pragma configurations, including *loop unroll*, *loop pipeline*, *array partition*, and *function inline*. The benchmarks cover a broad range of application domains, from bioinformatics to computer vision. Table II presents these results, revealing two key findings: (1) DAPO achieves substantially higher speedups for pragma-augmented designs (2.36×) compared to pragma-

free implementations ($1.67\times$). It is worth noting that for *crs* and *bnnkernel*, while DAPO's optimizations alone yield limited performance gains, they lay the foundation for pragma directives to achieve optimal performance. These results confirm that optimized pass sequences effectively complement rather than compete with pragma directives. (2) DAPO simultaneously improves performance and reduces resource requirements compared to Vitis HLS, achieving an average $2.36\times$ latency improvement while using fewer LUTs ($0.8\times$) and DSPs ($0.93\times$) on designs with pragma settings.

This dual benefit contrasts with pragma-based design space exploration, which typically involves performance-resource tradeoffs, and stems from the middle end transformations' ability to simplify program structure while improving execution efficiency. These results establish DAPO as a valuable complement to existing HLS optimization techniques.

C. Ablation Study

Besides the embedding methods compared above, we evaluated two additional graph neural architectures during the representation learning phase: node-attribute focused models (GCN [25]) and edge-aware models (PNA [26]) to further illustrate the impact of different embedding models and demonstrate the advantage of our heterogeneous graph. We also employed an RL model, with the observation space set to an all-zero vector as a baseline. As shown in Fig. 9(a), two phenomena can be observed: (1) A properly designed observation space can significantly boost the performance of the RL model with all training results surpassing the baseline model. (2) Only RGCN maintains comparable performance during inference, thus confirming the advantage of heterogeneous graphs compared to homogeneous graphs.

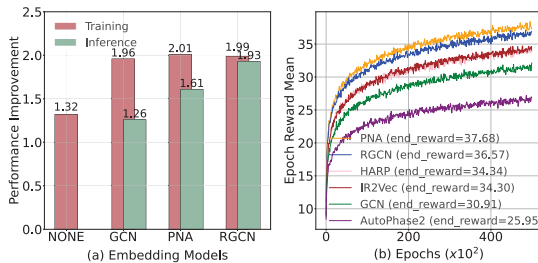


Fig. 9. Performance comparison of different model architectures: (a) performance improvement; (b) reinforcement learning reward.

Meanwhile, Fig. 9(b) reveals a striking inverse correlation between contrastive learning loss and reinforcement learning reward: architectures with lower contrastive learning loss consistently produced higher rewards, validating our approach for generating optimization-relevant embeddings. RGCN and PNA-based approaches achieved the highest final rewards, substantially outperforming existing methods. Notably, graph-based representations (HARP) consistently outperformed sequence-based approaches (IR2Vec), reinforcing the importance of structural modeling for compiler optimization.

VI. RELATED WORKS

A. Pass Ordering Methodologies

Pass ordering optimization has evolved through several research paradigms. Early heuristic-driven approaches [8], [27]

established foundational techniques but lacked generalization capabilities and required exhaustive exploration for each program. Machine learning approaches subsequently emerged, leveraging supervised learning to predict effective pass sequences [11]. However, these methods require extensive high-quality labeled data and struggle to generalize beyond training patterns. Reinforcement learning has emerged as a promising paradigm due to its ability to explore complex decision spaces without explicit supervision. On the contrary, current RL approaches for pass ordering face significant limitations in their observation space construction. Manual feature extraction methods [9] often fail to capture complex structural relationships, while generic program representations like IR2Vec [16] employed in recent work [28] lack specificity to the pass ordering task and HLS domain characteristics.

This analysis reveals a critical research gap: existing pass ordering methodologies lack program representation techniques specifically designed to capture structural patterns most relevant to optimizing HLS compiler passes.

B. Program Representation Learning

Program representation has evolved substantially in recent years, with increasing emphasis on graph-based approaches that capture structural relationships within code. Advanced techniques like IR2Vec [16] and ProGraML [29] have achieved significant progress in improving representation quality, but they primarily model programs as homogeneous graphs lacking specialization for pass ordering and hardware-specific characteristics. The HLS community has recently developed specialized representations for QoR prediction, including IronMan [6], HARP [10], and HGNN4HLS [30]. These methods, nevertheless, target QoR prediction rather than pass ordering and also employ homogeneous graphs that inadequately capture diverse relationship types crucial for understanding pass interactions.

Our DAPO framework addresses these limitations through a heterogeneous graph representation specifically designed for pass ordering, incorporating node attributes, node types, and edge types to capture complex relationships affecting pass effectiveness. Combined with our novel heterogeneous graph edit distance algorithm and relational graph neural networks, this approach enables more effective modeling of program structures for optimization.

VII. CONCLUSION

We present DAPO, the first design structure-aware framework for compiler pass ordering in high-level synthesis. DAPO integrates heterogeneous graph representation with contrastive learning to steer reinforcement learning towards discovering optimized pass sequences. Experimental evaluation on classic HLS designs demonstrates that DAPO consistently outperforms existing approaches, achieving $2.36\times$ speedup over Vitis HLS with comparable resource usage.

VIII. ACKNOWLEDGEMENTS

This work is partially supported by the National Key R&D Program of China (2023YFB4405100, 2023YFB4405103) and the RGC General Research Fund (16214123). We would like to thank all the reviewers for their valuable comments.

REFERENCES

- [1] Y. Chi, W. Qiao, A. Sohrabizadeh, J. Wang, and J. Cong, "Democratizing domain-specific computing," *Communications of the ACM*, vol. 66, no. 1, pp. 74–85, 2022.
- [2] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 127–136.
- [3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 33–36.
- [4] *Vitis High-Level Synthesis User Guide*, Xilinx, Inc., San Jose, CA, USA, 2020, accessed: Dec. 22, 2025. [Online]. Available: <https://docs.xilinx.com/r/2020.2-English/ug1399-vitis-hls/Optimization-Directives>
- [5] J. Cheng, S. Coward, L. Chelini, R. Barbalho, and T. Drane, "Seer: Super-optimization explorer for high-level synthesis using e-graph rewriting," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 1029–1044.
- [6] N. Wu, Y. Xie, and C. Hao, "Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, 2021, pp. 39–44.
- [7] L. Ferretti, A. Cini, G. Zacharopoulos, C. Alippi, and L. Pozzi, "Graph neural networks for high-level synthesis design space exploration," *ACM Transactions on Design Automation of Electronic Systems*, vol. 28, no. 2, pp. 1–20, 2022.
- [8] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson, "The effect of compiler optimizations on high-level synthesis for fpgas," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2013, pp. 89–96.
- [9] L. Haj-Ali, Q. J. Huang, J. Xiang, W. Moses, K. Asanovic, J. Wawrzyniec, and I. Stoica, "Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 70–81, 2020.
- [10] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, "Robust gnn-based representation learning for hls," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.
- [11] Y. Liang, K. Stone, A. Shamel, C. Cummins, M. Elhoushi, J. Guo, B. Steiner, X. Yang, P. Xie, H. J. Leather *et al.*, "Learning compiler pass orders using coreset and normalized value prediction," in *International Conference on Machine Learning*. PMLR, 2023, pp. 20746–20762.
- [12] T. Liang, J. Zhao, L. Feng, S. Sinha, and W. Zhang, "Hi-clockflow: Multi-clock dataflow automation and throughput optimization in high-level synthesis," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–6.
- [13] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "Dass: Combining dynamic & static scheduling in high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 628–641, 2021.
- [14] S. Muchnick, *Advanced compiler design implementation*. Morgan kaufmann, 1997.
- [15] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *The semantic web: 15th international conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, proceedings 15*. Springer, 2018, pp. 593–607.
- [16] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. Srikant, "Ir2vec: Llvm ir based scalable program embeddings," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–27, 2020.
- [17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.
- [18] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- [19] Z. Wu, E. Liang, M. Luo, S. Mika, J. E. Gonzalez, and I. Stoica, "RLlib flow: Distributed reinforcement learning is a dataflow problem," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2021. [Online]. Available: <https://proceedings.neurips.cc/paper/2021/file/2bce32ed409f5ebcee2a7b417ad9beed-Paper.pdf>
- [20] L.-N. Pouchet *et al.*, "Polybench: The polyhedral benchmark suite," *URL: http://www.cs.ucla.edu/pouchet/software/polybench*, vol. 437, pp. 1–1, 2012.
- [21] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proceedings of the IEEE International Symposium on Workload Characterization*, Raleigh, North Carolina, October 2014.
- [22] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 269–278.
- [23] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2008, pp. 1192–1195.
- [24] F.-A. Fortin, F.-M. De Rainville, M.-A. G. Gardner, M. Parizeau, and C. Gagné, "Deap: Evolutionary algorithms made easy," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 2171–2175, 2012.
- [25] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.
- [26] G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Veličković, "Principal neighbourhood aggregation for graph nets," *Advances in Neural Information Processing Systems*, vol. 33, pp. 13260–13271, 2020.
- [27] M. Tartara and S. Crespi Reghizzi, "Continuous learning of compiler heuristics," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–25, 2013.
- [28] S. Jain, Y. Andaluri, S. VenkataKeerthy, and R. Upadrasta, "Poset-rl: Phase ordering for optimizing size and execution time using reinforcement learning," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2022, pp. 121–131.
- [29] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. O'Boyle, and H. Leather, "Programl: A graph-based program representation for data flow analysis and compiler optimizations," in *International Conference on Machine Learning*. PMLR, 2021, pp. 2244–2253.
- [30] M. Gao, J. Zhao, Z. Lin, and M. Guo, "Hierarchical source-to-post-route qor prediction in high-level synthesis with gnns," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.