# Harnessing GNNs for Robust Representation Learning in High-Level Synthesis

Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, *Member, IEEE*, and Jason Cong, *Fellow, IEEE*

***Abstract*—The efficient and timely optimization of microarchitecture for a target application is hindered by the long evaluation runtime of a design candidate, creating a serious burden. To tackle this problem, researchers have started using learning algorithms such as graph neural networks (GNNs) to accelerate the process by developing a surrogate of the target tool. However, challenges arise when developing such models for HLS tools due to the program's long dependency range and deeply coupled input program and transformations (i.e., pragmas). To address them, in this paper, we present HARP (*H*ierarchical *A*ugmentation for *R*epresentation with *P*ragma optimization) with a novel hierarchical graph representation of the HLS design by introducing auxiliary nodes to include high-level hierarchical information about the design. Additionally, HARP decouples the representation of the program and its transformations and includes a neural pragma transformer (NPT) approach to facilitate a more systematic treatment of this process. Our proposed graph representation and model architecture of HARP not only enhance the performance of the model and design space exploration based on it but also improve the model's transfer learning capability, enabling easier adaptation to new environments.**

***Index Terms*—Design automation, graph neural network, high-level synthesis, representation learning.**

## I. Introduction

In RECENT decades, the emergence of domain-specific accelerators (DSAs) has provided a viable solution to the end of Dennard's scaling [11]. Consequently, the field-programmable gate array (FPGA) has become an appealing option for reconfigurable, energy-efficient high-performance computing (e.g., [15], [41]). Despite their potential advantages, FPGAs are not yet widely adopted to create DSAs in either academia or industry, partially due to their poor programmability. High-level synthesis (HLS) [9] has succeeded in reducing this burden, but exploiting HLS remains challenging for non-experts. This is because, even with HLS, a microarchitecture must be designed and described in code, which limits its accessibility to hardware designers.

As a result, a new research direction aims to enhance FPGA programmability by automating the optimization process of microarchitecture design [37], [39], [42], [45]. In HLS C/C++, the main instruments used to define the microarchitecture are compiler directives in the form of pragmas. An essential research question is how to incorporate the right combination of pragmas into the code to enhance the quality of results (QoR). This includes determining the type of required pragmas, where to apply them, and their options, such as the unroll factor or pipelining type. The complexity of this problem arises from the exponential growth in the number of candidate pragmas, the long synthesis time for each design, and the fact that the pragmas do not have a monotonic effect on performance and/or area, which makes it challenging to predict their impact. While the optimal choice of pragmas can yield significant performance improvements in the resulting microarchitecture, such as the 9000× speedup reported in [6], identifying the optimal combination of pragmas remains a challenging task [6], [16], [42].

To address this problem, several previous works, as summarized in [37], have treated the HLS tool as a black box and focused on developing efficient heuristics to explore the solution space more intelligently. Notably, AutoDSE [42] is a state-of-the-art approach that employs a bottleneck optimizer mimicking the optimization strategies of an expert designer. However, these works suffer from long runtimes as they rely on running the tool directly for evaluating the design configurations, with each run taking minutes to hours. This choice stems from the difficulty of capturing the tool's behavior with an analytical model [37], [42]. Recent research has demonstrated that leveraging learning algorithms can mitigate this problem. Graph neural networks (GNNs) [47] have been found to be highly effective in the electronic design automation (EDA) domain [13], [18], [22], [36], [39], [43]. These works represent the input program or circuit as a graph and utilize GNNs to summarize the graph properties and produce a vector for graph/node embeddings. The model then employs a post-processing stage that converts these embeddings to the final objectives that it wants to predict. In addition to GNNs, recent advancements in large language models (LLMs) like AlphaCode [25], ChatGPT [33], and GPT-4 [1] make them potential candidates for addressing the HLS optimization problem. However, all of these take huge computing power to train and none of them has targeted FPGA

Received 16 May 2024; revised 11 August 2024; accepted 7 October 2024. Date of publication 24 October 2024; date of current version 26 November 2024. This work was supported in part by the NSF 2211557, NSF 1937599, NSF 2119643, NSF 2303037, NASA, SRC JUMP 2.0 Center, Okawa Foundation, Amazon Research, Cisco, Picsart, Snapchat, and CDSC industrial partners (https://cdsc.ucla.edu/partners/). The review of this article was arranged by Associate Editor H. Amrouch. *(Corresponding author: Atefeh Sohrabizadeh.)*

The authors are with the Computer Science Department, University of California–Los Angeles, Los Angeles, CA 90095 USA (e-mail: atefehsz@cs.ucla.eduucla.edu; yba@cs.ucla.edu; yzsun@cs.ucla.edu; cong@cs.ucla.edu).

Digital Object Identifier 10.1109/TCASAI.2024.3485522

accelerator designs with performance optimization in mind. Therefore, for now, GNNs are a more practical solution for the problem at hand and we consider utilizing LLMs at a later time.

Although GNN-based models have shown promising performance in the EDA domain, there are still some challenges that need to be addressed to make them more effective. One of the main challenges is how to represent the HLS design (C/C++ program with architectural pragmas) in a way that captures all relevant details and makes it informative for the learning model. Additionally, as the design objectives are influenced by both program context and pragmas (i.e., transformations), it can be beneficial to develop a model that can learn the effect of each component separately. In response to these challenges, we propose and implement HARP. To address the first challenge, it includes a novel hierarchical representation of HLS designs. This representation incorporates program semantics and pragmas, while also introducing auxiliary nodes that provide high-level hierarchical information about the design. This graph representation provides a coarsened view of the design, which can assist with coping with the long-range dependencies within the program. In fact, it helps to reduce the average shortest path of our benchmark by a factor of 5. This permits the GNN model to pass the nodes' messages more easily throughout the whole graph. To tackle the second challenge, HARP intends to enhance modeling the pragma optimizations. Hence, we propose two approaches for decoupling the program representation from its transformations. The first approach separates the vector representation of the program and pragmas generated by the GNN and employs an autoencoder loop to ensure the pragma vector representation can reconstruct its initial features. The second approach introduces a neural pragma transformer (NPT), which models pragmas as learnable functions applied to the program representation. This architectural design aligns more naturally with the transformative nature of pragmas. We compare and evaluate these two approaches in our experiments.

The next challenge emerges when deploying the model in a new environment, where two types of shifts can occur that can lead to different data distributions compared to the training set. First, the *domain shift* arises when the model encounters a kernel that was not seen by the model during the training process. Second, the *task shift* appears when there is a need to predict a new objective that was not included in the model's training. A significant source of task shift occurs when the HLS tool is updated, as changes in tool heuristics can affect design objectives. Fig. 1 shows variations in latency and BRAM usage (skipping the rest of the resources due to space limitations) for 1145 designs during the transition from SDx 2018.3 to Vitis 2020.2. The vertical (horizontal) axis represents results from Vitis 2020.2 (SDx 2018.3). The outcomes are compared against the diagonal line $y = x$ for clarity. Given the cost of regenerating the database and retraining the model, it is preferable to transfer the model using a smaller dataset. Our experimental results show that HARP improves the performance of both the original and transferred models. Even with large datasets, the pretrained model of HARP yields better results after transfer learning. This strong transfer learning capability is due to our novel graph representation and model architecture.[1].

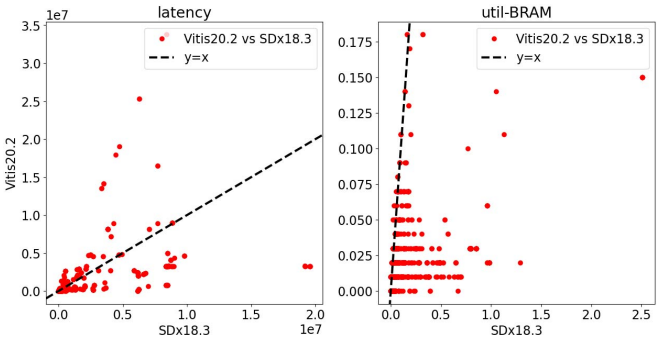



Fig. 1. The design objectives resulted from AMD/Xilinx Vitis 2020.2 over SDx 2018.3. The points are compared against the $y = x$ line.

In summary, in this paper, we make the following contributions:

- We propose a novel hierarchical graph representation to combine both a high-level view (combination of C/C++ level and LLVM IR level) and a low-level view (LLVM IR level) of the HLS designs, which can help to reduce the long range of dependencies.
- We propose how to decouple the representation of programs and their pragmas, allowing the model to learn the individual impact of each component more effectively.
- We evaluate the effectiveness of our proposed hierarchical graph representation and model architectures for transfer learning by showcasing their capacity to enhance the adaptability of the resulting model to changes in the objectives of HLS designs.
- The experimental results demonstrate that our approach can decrease the prediction loss compared to a state-of-the-art (SOTA) GNN-based work by 12-34%.
- In design space exploration (DSE), HARP achieves an average performance improvement of $2.54\times$ over the SOTA model-free DSE within a $25\times$ reduced time limit, and outperforms the SOTA model-based approach by $1.31\times$ on average after transfer learning with limited data.
- Even with large datasets, HARP shows strong transfer learning capability, outperforming the SOTA model-based approach by $1.26\times$ on average.

## II. Background

In this section, we first review a common way to present a program as a graph and provide an overview of GNNs. We then review the pragmas that define the solution space we need to explore.

### A. Programs as Graphs

A popular way of representing a program as a graph is to extract its *control and data flow graph* (CDFG) from its

[1]Preliminary versions of this work were presented in [39] and [40]. A more systematic approach to the transfer learning problem is provided in [12].

intermediate representation (IR) in LLVM [23]. Thus, instead of focusing on the grammar of the code, the semantics of the program flow is captured. In a CDFG, the nodes represent the LLVM instructions that are connected to each other based on the control flow of the program. For the data flow of the program, a second type of edge is added between the nodes based on the operands of the instructions. Note that a CDFG includes many low-level operations (e.g., memory management) which makes it desirable for FPGA kernels.

### B. Graph Neural Networks

A GNN [47] extracts information from a given graph by learning the features, known as embeddings, for its nodes. This is achieved through a sequence of layers, performing aggregation (AGG) of the information from neighboring nodes ($\mathcal{N}(i)$), and applying a transformation function (TF) to the aggregated result. The computation of a single layer in a typical GNN can be represented as follows:

$$\vec{h_i'} = \sigma(\mathrm{TF}(\mathrm{AGG}(\{\vec{h_j} | j \in \mathcal{N}(i)\}))) \tag{1}$$

where $h_i \in \mathbb{R}^F$ ($h_i' \in \mathbb{R}^{F'}$) represent the input (output) embeddings of node $i$, with $F$ and $F'$ denoting the number of features, and $\sigma$ is an activation function to introduce non-linearity to the model.

### C. HLS Design Space and Pragmas

HARP is developed on top of the open-source AMD/Xilinx Merlin Compiler [8], which offers the advantage of reducing optimization pragmas and applying source-level code transformations to enable various architectural optimizations such as memory burst, memory coalescing, and coarse-grained optimizations [6], [42]. The solution space using the Merlin Compiler includes three types of pragmas, as listed in Table I: `pipeline`, `parallel`, and `tile`. However, these pragmas correspond to several HLS pragmas, including `pipeline`, `unroll`, `array_partition`, `inline`, `dependence`, and `loop_flatten`. This is because the Merlin Compiler only needs a high-level description of the design with its pragmas in order to transform the input code and generate an HLS C/C++ code with the required HLS pragmas to implement the described design. The `pipeline` pragma can be configured to implement either fine-grained (fg) or coarse-grained (cg) pipelining. By utilizing the `pipeline` pragma with the *cg* option, the Merlin Compiler eliminates the need for manual code rewriting to implement double buffering, since it automatically transforms the code accordingly. The `parallel` and `tile` pragmas allow us to adjust the duplication factor of the processing elements (in the case of *cg* parallelization) or the arithmetic operations (in the case of *fg* parallelization) as well as the amount of cached data, respectively. As a result, the Merlin Compiler provides a much more compact design space and is used in this study. Our approach, however, can be generalized and applied to other HLS tools directly, such as Vitis HLS [4] or Intel HLS with proper training.

TABLE I
MERLIN PRAGMAS WITH ARCHITECTURE STRUCTURES

| Keyword | Available Options | Architecture Structure |
|---|---|---|
| pipeline | mode=cg/fg/off | CG or FG or no pipelining |
| parallel | factor=<int> | CG & FG parallelism |
| tile | factor=<int> | Loop Tiling |

CG: Coarse-grained; FG: Fine-grained.

## III. PROBLEM FORMULATION

In this work, we aim to speed up the DSE problem for HLS. For this matter, we propose solutions for the following problems:

***Problem 1: Build the Prediction Model.*** Let $\mathcal{P}$ be a C program as the FPGA accelerator kernel with design configurations ($\theta$). Let $\mathbf{H}$ be a vendor HLS tool that outputs the true execution cycle $Cycle(\mathbf{H}, \mathcal{P}(\theta))$ and the true resource utilization $Util(\mathbf{H}, \mathcal{P}(\theta))$:

$$\mathbf{Q_H}(\mathcal{P}(\theta)) = \big(Cycle(\mathbf{H}, \mathcal{P}(\theta)), Util(\mathbf{H}, \mathcal{P}(\theta))\big) \tag{2}$$

Find a prediction function ($\mathbf{F}$) that approximates the results of $\mathbf{H}$ for any given program $\mathcal{P}$ with any design configurations ($\theta$):

$$\min_{\mathbf{F}} \ \big(\underset{\theta}{average} \ \big(Loss(\mathbf{Q_F}(\mathcal{P}(\theta)), \mathbf{Q_H}(\mathcal{P}(\theta)))\big)\big) \tag{3}$$

***Problem 2: Identify the Optimal Configuration.*** For the program $\mathcal{P}$ defined above, find a configuration $\theta \in \mathbb{R}_{\mathcal{P}}$ in a given search time limit so that the generated design $\mathcal{P}(\theta)$ can fit in the FPGA and the execution cycle is minimized.

## IV. RELATED WORK

**Machine Learning for EDA.** Since most problems in EDA are classified as NP-complete, machine learning algorithms are gaining popularity in this domain due to their ability to efficiently solve them and produce high-quality solutions [16]. Additionally, these algorithms can aid in reducing manual effort and introducing greater automation into the design process. Machine learning (ML) and deep learning (DL) models have demonstrated remarkable success in various phases of the EDA flow, such as high-level synthesis [5], [27], [39], [43], [45], logic synthesis [32], [50], placement and routing in physical design [2], [21], [22], [28], [31], [48], etc. Huang *et al.* [16] identify four primary tasks in this field: (1) decision-making in conventional approaches, where an ML model substitutes for brute-force search or empirical configuration selection; (2) performance prediction, in which a model is employed to rapidly estimate QoR; (3) black-box optimization, where a surrogate model is constructed to explore the solution space more efficiently for optimal design; and (4) automated design, where both the predictor and policy are learned and continually adjusted online to significantly reduce human effort in complex design tasks. This work aims to enhance performance prediction to facilitate HLS black-box design optimization.

**GNN for EDA.** When a larger dataset is available, DL algorithms have demonstrated significant performance improvements in EDA. GNNs are one of the most widely used algorithms for this purpose, as graphs provide an intuitive way

to model programs, Boolean functions, netlists, and layouts commonly used in many EDA problems [18], [29], [36]. This is also true for the HLS problem, where analytical models cannot achieve acceptable accuracy [37], [42], but learning algorithms have demonstrated superior performance. However, applying learning algorithms to the HLS problem, which constitutes an early stage of design optimization, can pose considerable challenges due to the extensive and intricate optimization procedures that a design must undergo before reaching its final microarchitecture.

**ML and GNN for HLS.** Although traditional ML algorithms such as random forest, decision tree, and linear regression have been employed to model HLS tools [37], recent studies have shown that GNNs can significantly improve accuracy [5], [39], [43], [45], [46]. Moreover, using GNNs can help unify the model for several applications, as opposed to developing a separate model for each application. For instance, GNN-DSE [39] proposes a graph representation to capture both the program semantics and the pragma flow and develops a GNN-based model to build a surrogate of the HLS tool that can predict the latency and resource utilization for BRAM, DSP, FF, and LUT. Bai *et al.* [5] extend GNN-DSE by presenting a meta-learning-based framework to adapt to domain changes. Ustun *et al.* [43] represent the HLS design (without pragmas) as a data flow graph (DFG) and build a GNN-based model to predict the mapping of arithmetic operations to the DSPs and LUTs, which can improve the accuracy of delay prediction. Similarly, IronMan [45] converts the program (without optimization pragmas) to DFG and predicts the critical path under different resource allocations (DSP or LUT) to the computation nodes using graph convolutional networks (GCNs) [20]. Wu *et al.* [46] also work with HLS designs without pragmas and construct a hierarchical GNN that first performs node-level classification to predict the resource type (DSP, LUT, or FF) for implementing the node and then uses this information to estimate the critical path as the graph-level prediction.

**HLS Design Space Exploration (DSE).** Learning algorithms have been also utilized for expediting the HLS DSE process to discover the Pareto-optimal points [44], [45]. Unlike the prior works that use general-purpose heuristics [37] or dedicated heuristics [42] to explore the solution space, this research approach employs a data-driven method for the search. For instance, IronMan trains a reinforcement learning agent that identifies the optimal resource allocation between DSP and LUT under user-specified constraints, such as minimizing resource consumption or optimizing the critical path.

Although optimization pragmas are the primary source for improving the resulting microarchitecture [6], only a few studies have developed a comprehensive learning model for HLS that utilizes optimization pragmas and can be applied to explore the solution space for numerous applications [5], [39]. In this work, we aim to pinpoint the challenges associated with developing such a model and propose solutions to address them.

## V. Methodology Overview

Our objective is to enhance the efficiency of exploring the HLS design space by developing a model capable of predicting the behavior of the HLS tool. To this end, we first collect a database from various applications as explained in Section V-A. We then explain how we can employ the trained *predictive model* as a surrogate to the HLS tool to run the *inference* and *DSE* stages in Section V-B. In this context, we develop a novel hierarchical graph representation, introduced in Section VI, which facilitates the propagation of graph information throughout the graph. Furthermore, HARP utilizes an advanced model architecture to improve the prediction accuracy. Unlike traditional machine learning models, which may incorrectly carry correlations between program P and transformations (i.e., pragmas) T into predictions, HARP learns the impact of each component separately, as detailed in Section VII. In Section V-C, we explain that this novel graph representation and model architecture can also be advantageous when moving to new programs or tasks that cause shifts in the data distribution, as the model can adapt more easily to the shift.

### A. Database Generation

Followed by our GNN-DSE [39] work, we utilize AutoDSE [42] to generate the initial database for our target applications. As mentioned in Section II-C, each `for` loop can take up to three pragmas: `pipeline`, `parallel`, and `tile` and the solution space is defined as in AutoDSE. To enable the model to learn to differentiate between design points ranging from "bad" to "good", we enhance AutoDSE to utilize three types of explorers:

- The existing explorer of AutoDSE, the bottleneck-based optimizer, which can find high-quality designs.
- A hybrid explorer integrates bottleneck-based optimization with exhaustive search. It assesses up to $P$ neighbors of the optimal design point following a $X\%$ enhancement in quality. In this context, a neighbor is defined as a point where only one pragma option differs from its origin point. This exploration approach enables the model to observe the impact of altering only one pragma within each local neighborhood.
- A random explorer that may explore configurations overlooked by the previous two explorers.

Once the explorer picks a design point, it is passed to the Merlin Compiler for evaluation. The result will be committed to a common database along with the program's graph representation (Section VI). We gradually collect results from *different* applications in a shared space to be used for training the model. Obtaining the true values of a design's objectives is time-consuming. This makes gathering the dataset for training the model to be the primary bottleneck in our approach. After building an initial database, we leverage the top points generated by our DSE (Section V-B) to augment the database. It is important to note that the DSE aims to evaluate the model on numerous unseen data points, necessitating a comprehensive representation of all design choices in our database. On the other hand, if our DSE mistakenly believes that an unseen design point has a high QoR, it indicates that the model lacks enough data to generalize across the entire solution space. These particular data points, which led to mispredictions of

Pareto-optimal design points, are more likely to contribute to a more accurate representation of the data distribution in subsequent rounds.

### B. Design Space Exploration

The predictive model can be used for finding the Pareto-optimal design points. Given that not all pragma configurations may yield valid hardware designs, we first need to have a classification model to determine the validity of each design point. If it is valid, we proceed to utilize a regression model to estimate the objectives of the design. If, for any of the resources, the utilization is higher than 0.8 ($80\%$), we reject that design due to over-utilization. This threshold is empirically set, as exceeding it leads to frequency degradation and mapping difficulties. Subsequently, from the remaining designs, we select the top 10 designs with the lowest latency numbers for evaluation using the HLS tool. As a result, our model enables us to reduce the number of evaluations with the HLS tool to only 10, rather than invoking it for every design point.

Since our models can complete inference for each design point within milliseconds, we can efficiently examine numerous design points. However, when dealing with enormous solution spaces, an exhaustive search within a reasonable timeframe might not be feasible. Therefore, as we proposed in GNN-DSE, we set a time limit for running the DSE and employ a heuristic to prioritize the exploration of the most promising candidates. As the HLS tools can implement the fine-grained optimizations better, we adapt a BFS-like traversal of the pragmas starting with the inner-most loops to create an ordered list of them. As a result, the pragmas of the inner-most loop levels are evaluated sooner.

If there are multiple pragmas at a loop level, we prioritize `parallel` over `pipeline` over `tile`. If the picked pragma ($P_p$) depends on another pragma ($P_d$) from the same loop level or one loop level further, we move pragma $P_d$ up in the ordered list. Pragma dependencies come from the rules AutoDSE sets in defining the solution space. For instance, there is always a dependency between the `parallel` pragma of one loop level and the `pipeline` pragma of its upper loop level because `fg` pipelining fully unrolls the sub-loops, making the `parallel` pragma unnecessary. Since there is always a dependency between the `parallel` pragma of one loop level with the `pipeline` pragma of its upper level, for the second-inner-most loop level upwards, this ordering results in evaluating the pipeline pragma before any other optimizations (even before pipelining or tiling of its inner loop). This prioritization is desirable because successful pipelining can lead to either double buffering or full unrolling of the inner loops, both of which are typically preferred over other optimizations on the inner loop. After evaluating this pragma, the same process is repeated for the next loop section until all pragmas are visited.

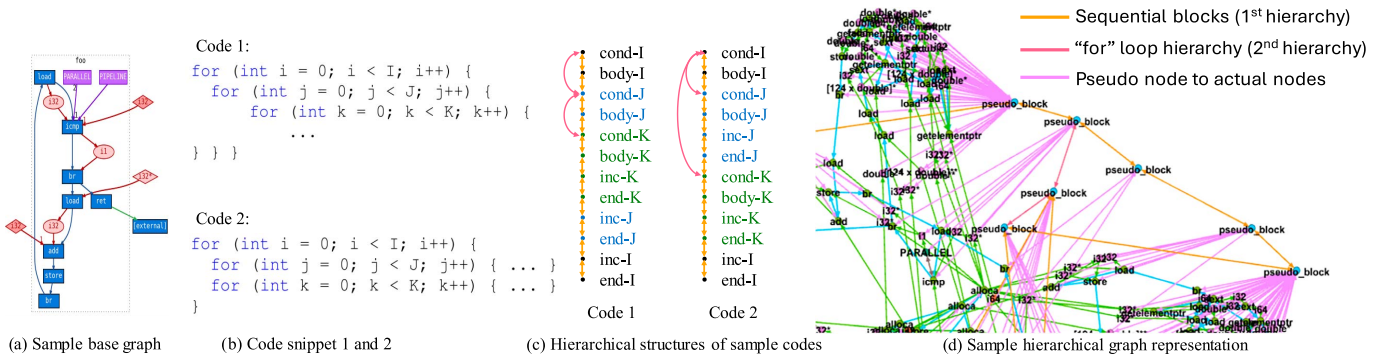
### C. Transfer Learning

When faced with new programs or tasks, the data distribution may shift from the training data distribution, making the prediction model unreliable. In Section I, we discussed one form of task shift that occurs when the HLS tool, used for synthesizing and implementing the design, changes. In such cases, collecting all the labels again, including the latency and resource usage, and retraining the entire pipeline can be time-consuming. To address this issue, we aim to adapt to the new environment using less labeled data by leveraging transfer learning. We might also want to introduce new applications to the model, which further requires this capability. Specifically, we use the model trained on the previous version of the tool and fine-tune it to adapt the predictions to the newly acquired labels with the new version of the tool.

Transfer learning [52] can be viewed as a form of task adaptation, where knowledge learned from a source task is transferred to a target task with limited labeled data. In our case, the source task refers to the previous version of the HLS tool, where a large amount of labeled data is available, and the target task refers to the new version of the tool, where limited labeled data is available. Additionally, as we transition to the new tool version, we introduce new kernels to contain domain shifts as well. We speculate that one important requirement for the success of transfer learning in this context is that the model must have a clear understanding of the components that impact optimization results, namely the program semantics and the impact of transformations. By distinguishing between these two components, the model can better update its predictions when the data distribution shifts.

Notably, we observe a high correlation not only among some objectives within the same version but also among the same objectives across different versions. When the pretrained model's target and the fine-tuned model's target are highly correlated, it suggests that the knowledge encoded in the pretrained model is relevant to the fine-tuning task. This can accelerate the learning process for new tasks. This correlation suggests that the pretrained model has already acquired representations or features beneficial for predicting the fine-tuned target. During fine-tuning, the weights of the pretrained model are adjusted to better align with the new target task. A strong initialization from the pretrained model can facilitate quicker convergence during fine-tuning and potentially yield improved performance. Moreover, the risk of overfitting during fine-tuning is mitigated, as the pretrained model has learned generalizable features or patterns relevant to both tasks, aiding in regularization and preventing the fine-tuned model from excessively fitting the training data. Furthermore, fine-tuning may demand less labeled data to achieve satisfactory performance compared to training the model from scratch, as the pretrained model has already acquired useful representations from a potentially larger dataset.

Our experimental results indeed demonstrate that our graph representation and model architecture are effective in improving the model's performance after transfer learning. Specifically, our approach achieves significant performance gains in terms of both the model accuracy and the DSE results when fine-tuned on the limited labeled data (in this case, about half the size of the previous dataset) from the new version of the tool. We also demonstrate that even in scenarios with ample data availability, transfer learning through fine-tuning remains highly effective,




(a) Sample base graph (b) Code snippet 1 and 2 (c) Hierarchical structures of sample codes (d) Sample hierarchical graph representation

Fig. 2. (a) The base (original) graph representation from GNN-DSE [39]; (b) Two sample code snippets; (c) The hierarchical structures of the two sample code snippets, showing only the pseudo nodes and the connection between them; (d) A sample hierarchical graph focusing on demonstrating the pseudo nodes and their connections.

further enhancing our performance metrics. Importantly, our HARP design exhibits significantly greater capability in transfer learning compared to other models, underscoring the importance of the optimizations we implemented in the graph representation and model architecture. These optimizations enable better learning of each problem component, ultimately leading to a robust representation.

## VI. Graph Representation

As mentioned in Section II-A, CDFG is a popular choice for representing FPGA kernels. However, CDFGs overlook operand precision and values, which are crucial in determining the QoR. ProGraMLL [10] is a proposed alternative that extends CDFG by assigning nodes to operands explicitly and including design call flow to preserve function hierarchies. As such, followed by our previous work GNN-DSE [39], we adapt ProGraML and extend it by including the *pragma* flow to represent a program. Candidate pragmas are defined in the following form

```
#pragma ACCEL [PragmaType] [factor=]auto{name}
```

`name` is a placeholder for the option of the pragma as defined in Table I. The base graph comes from our previous work, GNN-DSE. Specifically, we assign a node for storing the placeholder pragma for each candidate pragma. Since the pragmas are applied to the loops, we connect this node to one of the instruction nodes corresponding to the loop: `icmp`.

### A. Hierarchical Graph Representation

A common issue in GNNs is that their performance tends to degrade as the number of layers increases, leading to a phenomenon known as over-smoothing. This occurs when repeated graph convolutional layers create too similar node embeddings, thus losing important information about the graph structure. Consequently, GNNs typically have shallow networks, which focus on learning local neighborhoods, leading to limited receptive fields and difficulties in capturing a global view of the graph [13], [24]. This poses a significant challenge in effectively learning programs that are typically characterized by extensive dependency chains, wherein the performance of a given program element depends on the operation of another element located far away in the code.

We aim to tackle this challenge by developing a hierarchical graph representation that integrates both high-level and low-level perspectives of the program, specifically, the HLS design. By introducing nodes in the graph that can establish relationships at various levels, we can coarsen the graph representation to mitigate the impact of the long range of dependencies. To this end, our method incorporates a high-level view that combines the C/C++ code level and LLVM IR [23] level and a low-level view that relies solely on the LLVM IR level. We construct the graph starting from LLVM IR and subsequently augment it to include two further abstraction tiers within the program.

To build the second level of representation in the graph, we insert auxiliary nodes (*pseudo nodes*), where each pseudo node corresponds to a distinct LLVM IR block. A block in LLVM IR is a sequence of instructions that end with a terminator instruction, such as a branch, return, or switch. Each basic block in LLVM IR has a single entry point and a single exit point. We define a new node called `pseudo_block` for each block. Fig. 2(b) and 2(c) illustrate two toy examples for showcasing these nodes and the hierarchical structures between them. In LLVM IR, each `for` loop is typically translated into 4 blocks. These blocks consist of the loop condition block, the loop body block, the block for updating the loop iterator, and the final block with a branch instruction to transition to the subsequent block after the loop's completion. Fig. 2(c) portrays the pseudo nodes assigned to each of these blocks, along with their order and connectivity. The pseudo nodes are linked to one another based on their sequential order. Additionally, the pseudo nodes representing the initial blocks of the `for` loops establish connections based on their order in the C/C++ code. As demonstrated, each `for` loop is linked to its parent `for` loop (if any) and its first-level children (if any).

Fig. 2(d) shows a partial view (due to the space limit) of a graph for a real case. Each `pseudo_block` node has three types of edges. First, it links to all instruction and data nodes within that block. Second, it connects to other pseudo-nodes in sequential order, thereby creating the first level of hierarchy. Third, it establishes connections based on the hierarchy level of the `for` loops in the C/C++ code, linking their first blocks

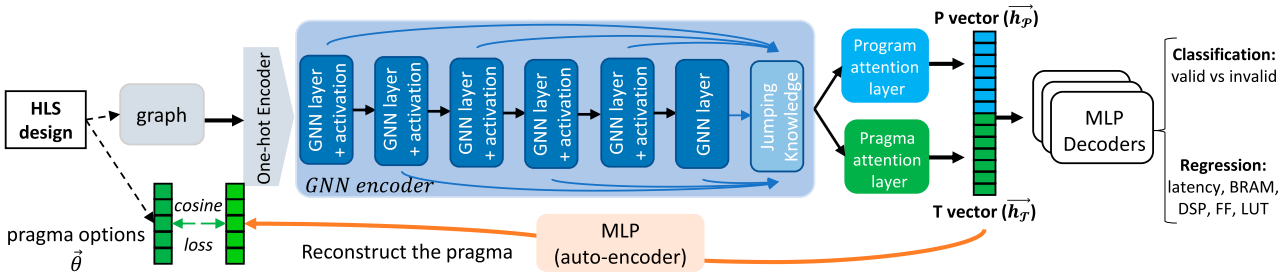



Fig. 3. Separating the vector representation of the program P and its transformation T. Distinct vectors are generated for each one. A further reconstruction loss with an autoencoder is used to enhance the influence of pragmas on the T vector.

according to their hierarchy in the code. This creates the second level of hierarchy in the graph representation. By adopting a hierarchical graph representation that combines high-level and low-level views, our approach can provide a more comprehensive understanding of the design and reduce the complexity of modeling long-range dependencies. This is achieved by decreasing the shortest path between the nodes via the pseudo nodes and their connections, which helps the GNN model to pass messages throughout the graph. For the kernels in our benchmark (comprising 41 unique kernels), the average shortest path between every two nodes in the graph is reduced from 25.3 (24.5) for the original graph to 5.1 (5) for the hierarchy graph, on average (the geometric mean).

To incorporate additional C-level and/or LLVM-level information in the graph, we define various attributes that can assist the model in gaining a deeper understanding of the role of each node within the graph. More specifically, each node/edge has the following attributes:

```
Node = {'function': Function ID, 'block': LLVM block
     ID, 'type': Node type, 'key_text': Node task}
Edge = (Src node ID, Dst node ID, {'flow': Flow type,
     'position': Position ID})
```

the `type`, `flow`, and `position` attributes encode this information (for non-pragma edges, `position` denotes their ordering):

| | | | |
|---|---|---|---|
| **type** | 0: instruction | 1: variable | 2: constant value |
| | 3: pragma | 4: pseudo node | |
| **flow** | 0: control | 1: data | 2: call |
| | 3: pragma | 4: pseudo node | 5: 1st hierarchy |
| | 6: 2nd hierarchy | | |
| **position** | 0: tile | 1: pipeline | 2: parallel |

The `key_text` attribute represents the primary functionality of the node. For example, it may be `pseudo_block`, `PIPELINE`, `load`, `i32*` for each of the pseudo node, pragma, control, and data node types. The pragma nodes have an additional attribute to encode the pragma option. This means that the only differences among the graphs for different design points of the same application are the attributes of their pragma nodes.

## VII. Model Architecture

At a high-level overview, the model begins by taking the graph representation of the program as input. It then proceeds to generate initial node and edge embeddings by concatenating the one-hot encoding of their attributes and important numeric values, including pragma options and loop trip counts. This encoding strategy helps to prioritize attributes that contribute more significantly to the final prediction. Following the initialization of embeddings, the model employs a GNN encoder to update these embeddings. The base GNN encoder is adopted from GNN-DSE. It has stacked TransformerConv [38] layers to produce node embeddings. Subsequently, a Jumping Knowledge Network (JKN) [49] mechanism is utilized to selectively choose varying neighborhood ranges for each node, recognizing that different nodes may require distinct neighborhood ranges to build meaningful embeddings. Finally, to generate a single vector representation for the entire graph, it leverages an attention mechanism [26] to enable the model to learn the importance levels of different nodes within the graph. Formally, the computation here can be modeled as:

$$\vec{h_{\mathcal{G}}} = \sum_{i \in N_{\mathcal{G}}} \text{softmax}\left(\text{MLP}(\vec{h_i})\right) \cdot \vec{h_i} \tag{4}$$

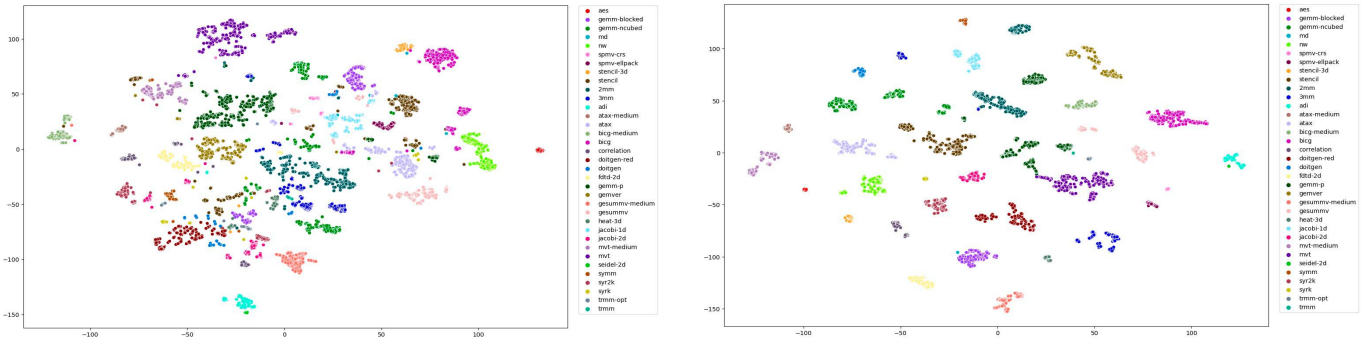
where MLP maps the node embedding from $\mathbb{R}^D$ to $\mathbb{R}$.

After encoding the graph as a vector, additional transformation steps are required to make the final prediction. We define two learning tasks to assess a design point. Firstly, a classification task determines the validity of a design configuration. Invalidity can arise from various factors, including challenges faced by the HLS tool in implementing certain pragma combinations. Designs failing to complete synthesis within a certain timeframe are flagged as invalid. Moreover, some pragma combinations may inherently be infeasible. Although we identified some invalid cases in our previous work, AutoDSE, not all the cases were covered so we let the model learn them.

Once a design is deemed valid, another model estimates its quality by predicting cycle count and resource utilization, forming our regression task. For both tasks, we utilize MLPs to make predictions based on graph-level embeddings. Since our regression task predicts correlated objectives, we share the GNN encoder backbone and apply multi-task prediction with separate MLPs (see Fig. 3). As a result, they can help each other in creating a better graph-level embedding.

### A. Decoupling Program and Transformation

The input to HLS tools is composed of two primary components that significantly influence the final microarchitecture.



(a) The coupled graph-level embedding.

(b) Our proposed P vector ($\vec{h_{\mathcal{P}}}$) representation.

Fig. 4. t-SNE visualization of the generated embeddings that are color-coded by the kernel name. Our P vector embedding is compared to when we learn a coupled embedding of the program and its transformations.

The first component is a high-level program description, denoted as P, expressed in C/C++, which defines the semantics and functionality of the DSA to be designed. The second component is a set of pragmas that include parallelizing, pipelining, and tiling directives, which are applied as transformations (T). As explained in Section II-C, these pragmas modify the microarchitecture which in turn affects the performance, power, and/or area of the DSA. The resulting HLS design is a function of both P and T. This work focuses on minimizing the latency L(P, T) of the design, given the available resource constraints of the FPGA on which the design will be implemented. The resource constraints are determined by the utilization of BRAM, DSP, flip-flops (FF), and lookup-tables (LUT), which are denoted as BRAM(P, T), DSP(P, T), FF(P, T), and LUT(P, T), respectively, and must be within certain preset thresholds. Thus, the GNN task is to learn the impact of T on P. Although GNN-DSE learns a coupled representation vector containing both P and T, we propose to separate the modeling of each component as it allows for a more natural understanding of their individual impacts. In sections VII-A1 and VII-A2, we present two optimizations for effectively implementing such a modeling strategy.

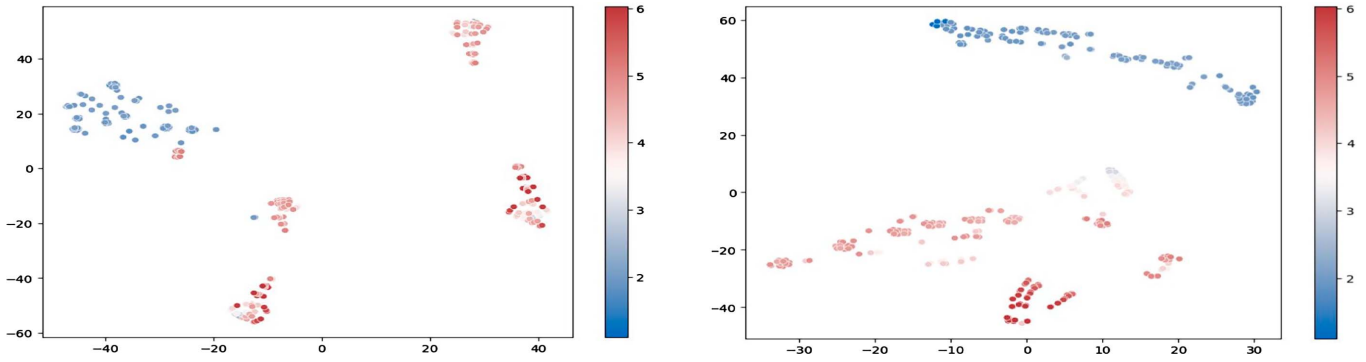
*1) Separating Vector Representation of Program and Transformation:* Fig. 3 depicts the model architecture for separating the vector representation of P and T. Once the GNN encoder is finished, the nodes have seen the program and the pragma structure, and their embeddings are produced based on that. We employ two attention layers to build the final P and T vectors. The attention layer is responsible for learning an attention (importance) score for each node and applying a weighted addition accordingly on their embeddings as expressed in Eq. 4. The *program attention layer* merges the nodes corresponding to the program context ($N_{\mathcal{P}}$) while the *pragma attention layer* pools only the pragma nodes ($N_{\mathcal{T}}$). In addition to separating the learning of the program and its transformation, this architecture helps to amplify the effect of the pragmas in predicting the final objectives.

To make the T vector ($\vec{h_{\mathcal{T}}}$) more meaningful, we utilize an autoencoder [14] structure. Autoencoders are designed to reconstruct part of the input data given its context. We use them to make sure $\vec{h_{\mathcal{T}}}$, which summarizes the pragmas, can reconstruct the input pragmas stored as a vector $\vec{\theta}$. This can help us increase the effect of a change in the input pragma options in the final vector representation. The autoencoder architecture consists of an MLP encoder and an MLP decoder, which take as input $\vec{h_{\mathcal{T}}}$ and aim to produce ($\vec{\theta}$). Despite the varying number of pragmas in different programs (HLS designs), we employ a fixed-sized vector for $\vec{\theta}$ to enable training a shared MLP decoder for all programs. In cases where programs have fewer pragmas, the remaining elements of $\vec{\theta}$ are filled with zeros. The total loss of the model would be calculated as:

$$l_T = l_{CE}\big(\mathbf{AE}(\vec{h_{\mathcal{T}}}), \vec{\theta}\big) + \sum_{o \in obj} l_{MP}\big(\mathbf{F}_o(P,T), H_o(P,T)\big) \tag{5}$$

where $l_{CE}$ and $l_{MP}$ denote the cosine error and the loss in the model's prediction, respectively. For the regression task, we measure $l_{MP}$ by calculating the mean squared error, and for the classification task, we use the cross-entropy loss. $\mathbf{AE}(\vec{h_{\mathcal{T}}})$ is the generated vector from the autoencoder. $\mathbf{F}_o(.)$ and $H_o(.)$ show the predicted value and the groud-truth value (HLS results) for objective $o$, respectively.

The t-SNE [30] visualizations of the embeddings generated by a coupled vector representation and our proposed P vector ($\vec{h_{\mathcal{P}}}$) are compared in Fig. 4. t-SNE is a method that is capable of representing data with high dimensionality through 2-D points, where data points that are close together in the 2-D space are indicative of similar data, and those far apart indicate dissimilar data. Each point in the figure represents a different design point from a different kernel and is color-coded based on its kernel name. The embeddings generated from GNN-DSE are interleaved when labeled by kernel name, whereas our proposed model successfully clusters the embeddings based on the kernel they belong to. To quantitatively assess the improvement in clustering, we compute the Euclidean distance between every pair of embeddings for a given kernel and measure the maximum and average distance among them. The average (across kernels) of the average and maximum distance using $\vec{h_{\mathcal{P}}}$ decreases by $3.7\times$ and $2.5\times$ respectively, compared to the embeddings generated by GNN-DSE. These findings highlight the effectiveness of $\vec{h_{\mathcal{P}}}$ in understanding the program scope and its semantics.



(a) The input pragma options ($\vec{\theta}$). (b) Our proposed T vector ($\vec{h_{\mathcal{T}}}$) representation.

Fig. 5. t-SNE visualization of the T vector compared to input pragma options that are color-coded by the performance value (log of speedup). Warmer colors indicate higher performance (lower latency).

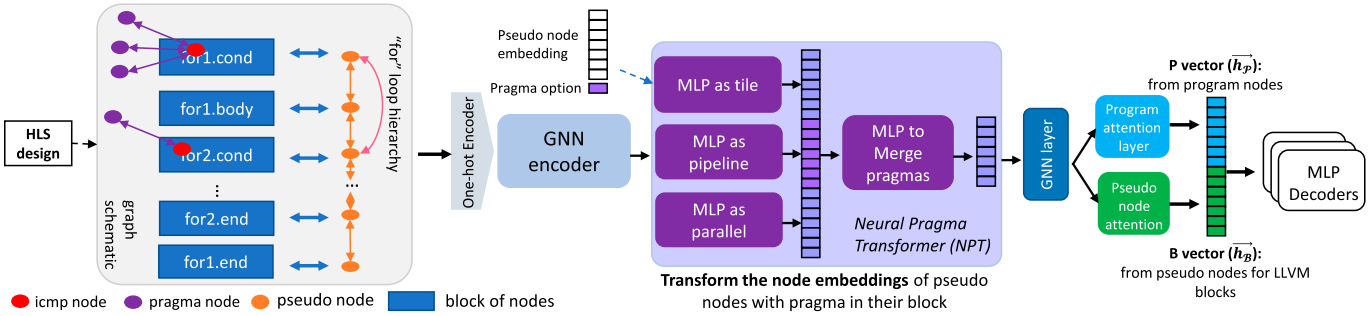



Fig. 6. Modeling pragmas as function transformations using NPT: each pragma type is modeled as a learnable MLP which takes in the embeddings of the pseudo node of the pragma block along with the pragma option. A second level of MLP is used to merge the results.

Furthermore, Fig. 5 shows the t-SNE visualization of $\vec{h_{\mathcal{T}}}$ for a random kernel, `gemm-blocked` from the MachSuite benchmark [35], which is color-coded based on the `perf` value. The `perf` value represents the log speedup of the design points to a reference latency value. In order to better illustrate the effectiveness of $\vec{h_{\mathcal{T}}}$, we compare it with visualization using pragma options $\vec{\theta}$. As the figure shows, there are some points that are similar to each other when they are compared with their pragma options $\vec{\theta}$ but have large differences in their `perf` value. This is expected as a small change in the pragma options (for example, changing the pipelining from coarse-grained to fine-grained) can have a significant effect on the resulting microarchitecture and the final performance. However, $\vec{h_{\mathcal{T}}}$ can effectively capture the impact of transformations, leading to improved clustering of design points. This helps us further in distinguishing the design points within the same program. Therefore, our proposed P and T vector representations together provide a better understanding of the program scope and the transformations that are applied to it. Experimental results (Section VIII-B) reveal that this model can decrease the loss by 10-23%.

*2) Modeling Pragmas as Function Transformation via Neural Pragma Transformer (NPT):* The primary goal of this study is to predict the objectives of an HLS design after applying a specific transformation T to its behavioral description in program P. These transformations are applied in the form of pragmas that alter the microarchitecture of the target application (Section II-C). For example, the parallel pragma duplicates statements within a loop and creates parallel units to process them simultaneously. Therefore, it is appropriate to model the pragmas (T) as functional transformations that are applied to the program P, which is represented as a graph. Our model for achieving this goal is illustrated in Fig. 6.

The model in Section VII-A1 can work with both the original graph and the hierarchy graph. However, this model needs to be applied to the hierarchical graph. Since the actual graphs are too crowded to visualize ($\sim 400$ nodes on average), a schematic of the hierarchical graph is presented in Fig. 6. The blue boxes represent the LLVM blocks, and only one representative node, namely, the `icmp` node, is depicted inside each box, which is connected to the `pragma` node. Each box has a corresponding pseudo node, and these pseudo nodes are connected with the hierarchical structure of the program as described in Section VI-A.

A GNN encoder with the same architecture as the one shown in Fig. 3 encodes the graph. This encoder is intended to focus on the program's structure along with the domain of its pragmas. Therefore, all pragma nodes have the same attribute as their default option (1 for `PARALLEL` and `TILE` pragmas. 'off' for `PIPELINE` pragma). As a result, unlike in Section VII-A1, the input one-hot encoder to this GNN encoder does not encode the

TABLE II
STATISTICS OF OUR THREE DATABASES WHICH CONSIST OF 41 UNIQUE KERNELS, WITH 21 OF THEM BEING SHARED ACROSS ALL VERSIONS

| Version | # kernels | #points (All/Valid) | Original range [min – max] | | | | | Normalized range [min – max] | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | latency | BRAM | DSP | LUT | FF | perf | BRAM | DSP | LUT | FF |
| **SDAccel 2018.3 (v18)** | 35 | 23,524/ 8,481 | [660 – 94.1M] | [0 – 12,950] | [0 – 57,531] | [0 – 7.7M] | [0 – 7.6M] | [-1.62 – 6.94] | [0 – 2.99] | [0 – 8.41] | [0 – 6.54] | [0 – 3.19] |
| **Vitis 2020.2 (v20)** | 27 | 12,168/ 4,569 | [992 – 1.5B] | [0 – 3,182] | [0 – 45,056] | [0 – 6.6M] | [0 – 4.4M] | [-3.59 – 6.65] | [0 – 0.73] | [0 – 6.58] | [0 – 5.59] | [0 – 1.86] |
| **Vitis 2021.1 (v21)** | 40 | 45,371/ 10,886 | [1,243 – 162M] | [0 – 13,750] | [0 – 89,728] | [0 – 13.2M] | [0 – 41.7M] | [-2.01 – 6.49] | [0 – 3.18] | [0 – 13.11] | [0 – 11.23] | [0 – 17.61] |

pragma options. After the GNN encoder has finished, the nodes have gained insight into the program's semantics in addition to the domain of the pragmas. We then utilize the learnable NPT module to apply pragmas as function transformations. NPT takes the embedding of the pseudo nodes that contain a pragma node in their block as the input and transforms it based on the type of the pragmas and their actual options. Each pragma type is modeled using a learnable MLP that accepts the node embedding and the pragma option as input and transforms the node embedding. If a pragma type is not present in the block, the default option is employed. The results of the MLP transformation for each pragma type are concatenated, and another MLP is used to learn their interactions and transform the concatenated result to the final node embedding of the corresponding pseudo node. After this stage, the pseudo nodes have acquired knowledge of the program semantics, the pragma domains, and their options. A further GNN layer is utilized to propagate the new information (pragma options) to the rest of the program nodes via message passing.

Once the final node embeddings have been generated, they are pooled to create the graph-level embedding. Consistent with the approach used in Section VII-A1, two vectors are generated with the attention mechanism in Eq. 4 to represent the program P and transformation T separately. Note that in this architecture, the transformations T are applied to the pseudo nodes. P vector ($\vec{h_{\mathcal{P}}}$) is generated by pooling the program nodes and B vector ($\vec{h_{\mathcal{B}}}$) is the result of pooling the pseudo nodes, which are the primary sources containing the pragma information. As before, $\vec{h_{\mathcal{P}}}$ and $\vec{h_{\mathcal{B}}}$ are concatenated, and the result is passed through MLP decoders to predict the final objectives.

## VIII. EXPERIMENTAL RESULTS

### A. Experimental Setup

Our database includes kernels of intermediate complexity that can be used as building blocks of larger applications. Specifically, we selected 41 kernels from the widely used MachSuite benchmark [35] and the Polyhedral benchmark (PolyBench) [51]. They include kernels with different computation intensities including linear algebra operations on matrices and vectors (e.g., BLAS kernels), data mining (`correlation` and `covariance`), stencil operations, encryption (`aes`), and a dynamic programming application (`nw`). For synthesis, we employ three AMD/Xilinx HLS tools, SDAccel 2018.3 (v18) [3], Vitis 2020.2 (v20), and Vitis 2021.1 (v21) [4], targeting the Xilinx Alveo U200 FPGA with a frequency of 250MHz. The v20 and v21 datasets are solely collected using AutoDSE after running each of its explorers for a day. We allocated 8 CPU cores and 25 GB of memory from the AMD EPYC 7V13 processor to run the explorers for each kernel. In addition, the design points in v18 include those obtained through our active learning approach (Section V-B).

For each design point, we collect the `latency` in terms of cycle counts and resource utilization for `DSP`, `BRAM`, `LUT`, and `FF`. We normalize the resource usage with the available resources on the board and the latency with $Norm_{factor} * log_2(\frac{Latency_{ref}}{latency})$ which we call `perf` with $Norm_{factor}$ being set to $0.5$ and $Latency_{ref}$ to $1e7$. Table II presents our database statistics. As mentioned in Section VII not all combinations of pragmas yield valid design points. The three versions of the database consist of a total of 41 unique kernels, with 21 kernels existing in all versions.

Our framework is implemented and trained using PyTorch [34]. The dataset is split into 70:15:15% for training, validation, and testing. We employ the Adam optimizer [19] with a maximum learning rate of 1e-3, which is linearly increased from zero over the first 2000 updates and then annealed to zero using a cosine schedule. Separate models are trained for classification and regression tasks. The classification/regression model is trained for 200/1500 epochs (taking less than 10h with 1 NVIDIA Tesla V100 GPU) for the first version of the database (v18) and 200 epochs for transfer learning to the v20 database. We pick the model with the lowest validation loss and report its performance on the test set. The initial embeddings have 154 features. We utilize 6 TRANSFORMERCONV [38]) with a feature dimension of 64 for the GNN encoder. The final objective prediction is performed using 4 MLP layers (one MLP network for each objective). The GNN and MLP layers are followed by ELU activation [7]. To mitigate overfitting, we apply dropout with a probability of 0.1 to the neurons in the GNN layers. The NPT module utilizes two layers for each of the MLPs. The autoencoder is an MLP with 4 layers that gradually reduces the feature size from 64 to 8 and then increases it to 21 which is the dimension of the vector containing the pragma options.

### B. Model Accuracy

We conducted a series of experiments to evaluate the effectiveness of various components of our approach. We first investigated our previous work GNN-DSE [39] that employs

TABLE III
TOTAL ROOT MEAN SQUARED ERROR (RMSE), MEAN ABSOLUTE ERROR (MAE), AND `perf` RANKING (TAU) OF THE MODELS. FOR RMSE AND MAE, THE LOWER THE BETTER. FOR TAU, THE HIGHER THE BETTER. THE PERCENTAGE OF DIFFERENCE IS MEASURED WITH RESPECT TO GNN-DSE [39]

| Graph | Model | Name | v18 database | | | v20 database | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Train from scratch | | | Train from scratch | | | Fine-tuned from v18 | | |
| | | | RMSE | MAE | perf tau | RMSE | MAE | perf tau | RMSE | MAE | perf tau |
| Original | Coupled P&T | GNN-DSE | 1.104 | 0.357 | 0.90 | 1.253 | 0.770 | 0.78 | 0.955 | 0.479 | 0.85 |
| | Separate P&T | M2 | 0.991 (-10%) | 0.307 (-14%) | 0.92 | 1.330 (+6%) | 0.790 (+3%) | 0.76 | 0.796 (-17%) | 0.368 (-23%) | 0.89 |
| Hierarchy | Separate P&T | M3 | **0.975** (-12%) | **0.257** (-28%) | **0.93** | 1.443 (+15%) | 0.948 (+23%) | 0.70 | 0.872 (-9%) | 0.348 (-27%) | 0.89 |
| | Sequential pragma as NPT | M4 | 1.083 (-2%) | 0.339 (-5%) | 0.91 | 1.502 (+20%) | 0.938 (+22%) | 0.73 | 0.876 (-8%) | 0.449 (-6%) | 0.86 |
| | Parallel & merge as NPT | M5 | 0.989 (-10%) | 0.277 (-23%) | 0.92 | 1.073 (-14%) | 0.636 (-17%) | 0.81 | 0.739 (-23%) | **0.309** (-35%) | 0.89 |
| | Parallel & merge as NPT + post GNN layer | HARP | **0.974** (-12%) | 0.295 (-18%) | **0.93** | **1.015** (-19%) | **0.601** (-22%) | **0.82** | **0.679** (-29%) | 0.317 (-34%) | **0.90** |

* NPT: neural pragma transformer

the original graph to represent the design and obtained a coupled vector representation for both the program (P) and its transformations (T). Then, we developed M2 by replacing the model architecture with our proposed approach described in Section VII-A1. This involved generating separate vector representations for P and T. Additionally, we constructed M3 by replacing the graph representation with our hierarchical graph. Furthermore, we implemented HARP based on the approach outlined in Section VII-A2. Note that it also exploits the idea of separating vector representations discussed in Section VII-A1. We also examined two variations of this model: M5, where the last GNN layer after the NPT module was excluded, and M4, which additionally applied the pragmas sequentially instead of using the existing parallel and merge structure of the NPT module. For each model, we evaluated its performance under three different scenarios. The first two scenarios involved training the model on datasets v18 and v20, respectively. In the third scenario, we employed the model pre-trained on dataset v18 and fine-tuned it on dataset v20. Our empirical results demonstrated that freezing the parameters of the first GNN layer, which helps reduce the number of parameters requiring updates, resulted in the best performance after fine-tuning.

Table III summarizes the performance of each model, using three metrics to assess their effectiveness. The first metric uses root mean squared error (RMSE) for each objective and calculates the total loss by summing the losses of all objectives. The second one utilizes mean absolute error (MAE) instead. For both metrics, we also provide the percentage difference compared to the results obtained from GNN-DSE. Since our primary objective is to conduct DSE for design optimization, the ranking of the `perf` values holds significant importance. Therefore, we employ Kendall's tau [17], a correlation coefficient that measures the similarity between two variables' rankings. A value of 1 indicates a perfect positive association. Hence, for RMSE and MAE, lower values indicate better performance, while for tau, higher values indicate superior performance.

The analysis of the results reveals several key observations. Firstly, when we employ separate learning of representations for program P and transformation T (M2), we observe a decrease in both losses and an improvement in the tau ranking of `perf`. However, an exception occurs when the model is trained from scratch on the v20 database. In this case, the increased number of parameters in the new model makes it harder to converge in a limited training budget (dataset and training time). Nonetheless, when utilizing the pre-trained model from the v18 database, the performance is able to catch up and even surpass GNN-DSE. A similar trend is observed when incorporating the hierarchical graph (M3), which further improves the results. Additionally, our findings highlight that the optimal architecture for the NPT involves modeling the pragmas as parallel learnable MLPs, with another MLP responsible for managing their interaction and merging their results. Finally, the most effective model for all scenarios (HARP) utilizes the hierarchy graph and consists of NPT employing the parallel and merge structure, followed by an additional GNN layer to propagate the pragma options throughout the program. It is important to note that this architecture, as depicted in Fig. 6, also generates separate embeddings for program P and pseudo nodes B, which contain the pragma (transformation) information here.

Moreover, the results in Table III align with our expectations, indicating a correlation between the objectives obtained from the two different versions of the HLS tool. Importantly, we observe that the pre-trained model from one version can effectively enhance the performance on the other version. This eliminates the need to regenerate the whole training set with each new version of the tool, streamlining the adaptation process. In addition, the results demonstrate that HARP exhibits the best graph representation and model architecture for effectively adapting to task shifts. This validates our hypothesis that by decoupling the learning and representation of the program and its transformations (i.e., pragmas), the model not only acquires

a deeper understanding of each component but also enhances its adaptability to new environments.

### C. DSE Results

To verify the effectiveness of our model, we use it to identify the Pareto-optimal design points by performing a DSE of the design parameters. We adopt the exploration technique mentioned in Section V-B in searching through the solution space. We employ a classification model to assist in pruning invalid design points. Given the ample points available for training the classification model and the relatively simpler task involved, we opt to train a model from scratch for each version using its respective dataset. These models exhibit high accuracy, with rates of 95% for the v18 database and 93% for the v20 database. Given their already high accuracy, we do not employ additional transfer learning methods for them. We set a time limit of 1h/kernel on our exploration and can explore approximately 100,000 points during this time. Once the exploration is finished, we synthesize the top 10 points using the HLS tool to get their true labels for comparison. We also run DSE utilizing the GNN-DSE approach, trained on our datasets in the same fashion. For the baseline comparison, we employ AutoDSE, which directly runs the HLS tool to evaluate design points. Due to the nature of this approach, AutoDSE requires a more extended runtime. Thus, we set a time limit of 25h/kernel for its DSE. During this period, AutoDSE typically explores an average of 250 valid points. The design space for our target kernels consists of 260,762 design points on the geometric mean. Therefore, obtaining an oracle design is impractical due to the complexity and time required for direct HLS tool execution. AutoDSE was evaluated against 45 applications, including those manually optimized by Xilinx, and outperformed previous DSE methods while matching the performance of manual optimizations. Thus, AutoDSE is a strong benchmark for assessing DSE efficiency.

Table IV summarizes the DSE results obtained using versions v18 and v20 of the HLS tool. The DSE is conducted on a total of 35 kernels for SDx 2018.3 (v18) and 27 kernels for Vitis 2020.2 (v20). It is important to note that among the 22 kernels shared between the v18 and v20 databases, the average latency of optimal design in v18 is $5.54\times$ ($1.36\times$ on the geometric mean) higher than that in v20, suggesting improvements in the heuristics of the HLS tool over time. Due to space limitations, we only report the arithmetic (avg) and geometric mean (geo mean) of the speedup of the optimal design found by each DSE with respect to the best design discovered by AutoDSE. As the model-based DSEs get to explore a much larger space, they can find better points compared to a model-free DSE. Notably, for `3mm` kernel from PolyBench with a solution space of over 17 trillion points, both HARP and GNN-DSE demonstrate speedups of $70\times$. The results reveal that HARP outperforms both AutoDSE and GNN-DSE. Specifically, HARP showcases its competence in adapting to new versions of the HLS tool (v20 kernels), surpassing the performance of GNN-DSE by an average (geometric mean) speedup of $1.31\times$ ($1.33\times$). It is important to highlight that when transitioning to the v20 dataset, in addition to the task shift, we also have included 5 new kernels, resulting in a domain shift as well. The results validate our hypothesis that the hierarchical graph structure in addition to the decoupling of program and transformation learning contributes to better adaptation capabilities in the face of shifts from the original training data distribution.

TABLE IV
THE PERFORMANCE OF THE BEST DESIGN FOUND BY EACH DSE WITH RESPECT TO THE BEST ONE FOUND BY AUTODSE [42] IN 25h

| Approach | Time Limit | v18 kernels (#:35) | | v20 kernels (#:27) | |
|---|---|---|---|---|---|
| | | avg | geo mean | avg | geo mean |
| AutoDSE | 25h/kernel | $1\times$ | $1\times$ | $1\times$ | $1\times$ |
| GNN-DSE | 1h/kernel | $3.51\times$ | $0.99\times$ | $0.88\times$ | $0.79\times$ |
| **HARP** | 1h/kernel | $3.61\times$ | $1.23\times$ | $1.15\times$ | $1.05\times$ |

### D. Ablation Study: Transfer Learning With Abundant Data

In Section VIII-B, we demonstrated that higher prediction accuracy can be achieved through transfer learning when limited data is available. This in turn leads to improved DSE results. To investigate the effectiveness of transfer learning with a large dataset, we developed a larger dataset with Vitis 2021.1 (v21) as detailed in Table II. Similar to previous findings, training a classification model on this dataset yielded 99% accuracy, eliminating the need for further transfer learning. However, for the regression model, we pursued two different approaches. Initially, we fine-tuned the v20 model using the v21 dataset for 400 epochs. Then, considering the large number of data available for this version, we trained another model from scratch on the dataset for 1500 epochs until it nearly matched the accuracy of the fine-tuned model on the test set.

We use the same experimental setting as in Section VIII-C. Similar to the improvement we saw in designs' performances when we transitioned from v18 to v20, we see an average latency reduction of $2.69\times$ ($1.61\times$ on the geometric mean) in the optimal design points for the 26 common kernels when we transition from the v20 to the v21 HLS tool. This further shows the improvements in the HLS tools with each new version.

Table V presents a summary of the arithmetic and geometric mean speedup achieved by each approach compared to the best results obtained by AutoDSE. Notably, model-based approaches demonstrate strong performance when trained from scratch due to their extensive training data. This, coupled with their ability to explore a larger solution space, enables them to discover better design points. However, fine-tuning a pretrained model from a previous version of the HLS tool yielded no advantages for GNN-DSE. It is noteworthy to mention that alongside the task shift, we also encounter a domain shift, with 14 new kernels introduced compared to v20 and 6 new kernels compared to v18 (the pretrained model for v20). In contrast, HARP exhibits more robust results and notable improvements post-transfer learning. This feature is highly desirable as it allows us to leverage previously gathered data, a process that incurred significant costs. Rather than discarding efforts and expenses associated with collecting datasets from prior HLS tool versions and kernels, we can utilize them to achieve enhanced

TABLE V
THE TRANSFER LEARNING RESULTS WHEN LARGER DATASET IS PRESENT. THE PERFORMANCE COMPARISON SHOWS THE BEST DESIGN FOUND BY EACH DSE WITH RESPECT TO THE BEST ONE FOUND BY AUTODSE [42] IN 25h

| Approach | Time Limit h/kernel | v21 kernels (#:40) | | | |
|---|---|---|---|---|---|
| | | Trained from scratch | | Fine-tuned from v20 | |
| | | avg | geo mean | avg | geo mean |
| AutoDSE | 25 | 1× | 1× | 1× | 1× |
| GNN-DSE | 1 | 1.17× | 1.04× | 1.16× | 0.98× |
| **HARP** | 1 | 1.19× | 0.82× | 1.46× | 1.15× |

TABLE VI
IMPACT OF THE CLASSIFICATION MODEL. THE BASELINE IS AUTODSE [42] AFTER RUNNING FOR 25h

| Dataset | Without Classification | | With Classification | |
|---|---|---|---|---|
| | avg | geo mean* | avg | geo mean* |
| v18 #:35 | 3.14× | 0.89× | 3.61× | 1.37× |
| v20 #:27 | 0.96× | 0.85× | 1.15× | 1.05× |
| v21 #:40 | 1.34× | 0.92× | 1.46× | 1.09× |

optimization outcomes. Note that in the current setup, the model trained from scratch underwent 3.75× more epochs to achieve the same test set accuracy as the fine-tuned model. Nevertheless, the fine-tuned model demonstrates superior generalization across the entire solution space, which includes many unseen points.

### E. Ablation Study: Impact of the Classification Model

In Section VII, we discussed that numerous pragma combinations result in an invalid design. We defined an invalid design as one that either cannot be synthesized, exceeds a given synthesis time limit, or at least one of the pragmas could not be applied. Table II shows that in our database, only 30% of the points created a valid design point. Therefore, it is crucial for our optimizer to identify and discard design points that cannot produce a valid microarchitecture. Our classification model is responsible for this task. Table VI compares the speedup performance we achieve compared to AutoDSE when our classification model is absent. Here, we focus solely on the top 10 designs generated by the model. Acknowledging that they may all be invalid designs, we calculate a modified version of the geometric mean (denoted as *geo mean**), where we add one to all speedup values and subtract one from the resulting geometric mean. The results show that the classification model can help to improve the DSE results since it can prune the invalid points.

## IX. CONCLUSION Future Work

In this work, we discussed three key challenges in developing a GNN-based model for HLS and developed HARP for addressing them. Firstly, we tackle the long-range dependency issue in HLS kernels by proposing a hierarchical graph structure, reducing the average shortest path in our benchmark kernels by 5×. Secondly, recognizing that the final objectives are influenced by two main components, program structure and its transformations in the form of pragmas, we decouple their representation to enhance the model's performance. This improved graph representation and model architecture enable better adaptation to the inevitable task shifts. Although our focus in this paper is on FPGAs, our design decisions are not dependent on them. We believe that our approach can be applied to other platforms and HLS tools as well. Switching the HLS tool (e.g., from Vitis HLS to Catapult HLS) not only changes the target hardware significantly but also the programming style and pragmas. We need to assess if transfer learning can adapt to these changes or if separate domain-specific models would be more effective. Moving forward, we aim to investigate the minimum number of points required for effective adaptation to these shifts and explore appropriate sampling techniques. Additionally, we plan to extend our data-driven approach to DSE exploration using reinforcement learning methods. We envision further advancements by developing hierarchical GNNs operating at the subgraph level to enhance compositional objective prediction. Finally, we will explore the integration of GNNs and LLMs, leveraging multiple design modalities—graph representation and source code—to address this problem more effectively.

## ACKNOWLEDGMENT

All materials available at https://github.com/UCLA-VAST/HARP

## REFERENCES

[1] J. Achiam et al., "GPT-4 technical report," 2023, *arXiv:2303.08774*.
[2] M. B. Alawieh, W. Li, Y. Lin, L. Singhal, M. A. Iyer, and D. Z. Pan, "High-definition routing congestion prediction for large-scale FPGAs," in *Proc. 25th Asia South Pacific Des. Automat. Conf. (ASP-DAC)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 26–31.
[3] "AMD/Xilinx SDAccel - Vivado HLS," Accessed: 2024. [Online]. Available: https://docs.xilinx.com/v/u/2018.3-English/ug902-vivado-high-level-synthesis
[4] "AMD/Xilinx Vitis HLS," Accessed: 2024. [Online]. Available: https://docs.xilinx.com/v/u/2020.2-English/ug1416-vitis-documentation
[5] Y. Bai, A. Sohrabizadeh, Y. Sun, and J. Cong, "Improving GNN-based accelerator design automation with meta learning," in *Proc. 59th ACM/IEEE Des. Automat. Conf. (DAC)*, 2022, pp. 1347–1350.
[6] Y. Chi, W. Qiao, A. Sohrabizadeh, J. Wang, and J. Cong, "Democratizing domain-specific computing," *Commun. ACM*, vol. 66, no. 1, pp. 74–85, 2022.
[7] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," 2015, *arXiv:1511.07289*.
[8] J. Cong, M. Huang, P. Pan, Y. Wang, and P. Zhang, "Source-to-source optimization for HLS," in *FPGAs Softw. Programmers*, 2016, pp. 137–163.
[9] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS today: Successes, challenges, and opportunities," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, pp. 1–42, 2022.
[10] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O'Boyle, and H. Leather, "ProGraML: A graph-based program representation for data flow analysis and compiler optimizations," in *Proc. Int. Conf. Mach. Learn. (PMLR)*, 2021, pp. 2244–2253.
[11] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.
[12] Z. Ding, A. Sohrabizadeh, W. Li, Z. Qin, Y. Sun, and J. Cong, "Efficient task transfer for HLS DSE," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD)*, Piscataway, NJ, USA: IEEE Press, 2024, pp. 1–9.

[13] Z. Guo, M. Liu, J. Gu, S. Zhang, D. Z. Pan, and Y. Lin, "A timing engine inspired graph neural network model for pre-routing slack prediction," in *Proc. 59th ACM/IEEE Des. Automat. Conf. (DAC)*, 2022, pp. 1207–1212.

[14] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, Jul. 2006.

[15] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 1–9.

[16] G. Huang et al., "Machine learning for electronic design automation: A survey," *ACM Trans. Des. Automat. Electron. Syst. (TODAES)*, vol. 26, no. 5, pp. 1–46, 2021.

[17] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[18] B. Khailany, "Accelerating chip design with machine learning," in *Proc. ACM/IEEE Workshop Mach. Learn. CAD*, 2020, p. 33.

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

[20] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.

[21] R. Kirby, S. Godil, R. Roy, and B. Catanzaro, "CongestionNet: Routing congestion prediction using deep graph neural networks," in *Proc. IFIP/IEEE 27th Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 217–222.

[22] M. Kou, J. Zeng, B. Han, F. Xu, J. Gu, and H. Yao, "GEML: GNN-based efficient mapping method for large loop applications on CGRA," in *Proc. 59th ACM/IEEE Des. Automat. Conf. (DAC)*, 2022, pp. 337–342.

[23] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, IEEE, 2004, pp. 75–86.

[24] Q. Li, Z. Han, and X.-M. Wu, "Deeper insights into graph convolutional networks for semi-supervised learning," in *Proc. AAAI Conf. Artif. Intell.*, vol. 32, no. 1, 2018.

[25] Y. Li et al., "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[26] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2015, *arXiv:1511.05493*.

[27] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proc. 50th Annu. Des. Automat. Conf. (DAC)*, 2013, pp. 1–7.

[28] Y.-C. Lu, S. Pentapati, and S. K. Lim, "VLSI placement optimization using graph neural networks," in *Proc. 34th Adv. Neural Inf. Process. Syst. (NeurIPS) Workshop Mach. Learn. Syst., Virtual*, 2020, pp. 6–12.

[29] Y. Ma, Z. He, W. Li, L. Zhang, and B. Yu, "Understanding graphs in EDA: From shallow to deep learning," in *Proc. Int. Symp. Phys. Des. (ISPD)*, 2020, pp. 119–126.

[30] L. v. d. Maaten and G. Hinton, "Visualizing data using t-SNE," *J. Mach. Learn. Res.*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[31] A. Mirhoseini et al., "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.

[32] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E. Gaillardon, "LSOracle: A logic synthesis framework driven by artificial intelligence," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD),* Piscataway, NJ, USA: IEEE Press, 2019, pp. 1–6.

[33] "OpenAI: Introducing ChatGPT," OpenAI. 2022. Accessed: 2022. [Online]. Available: https://openai.com/blog/chatgpt

[34] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," *Adv. Neural Inf. Process. Syst.*, vol. 32, 2019.

[35] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, IEEE, 2014, pp. 110–119.

[36] H. Ren, S. Nath, Y. Zhang, H. Chen, and M. Liu, "Why are graph neural networks effective for EDA problems?" in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD)*, 2022, pp. 1–8.

[37] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2628–2639, Oct. 2020.

[38] Y. Shi, Z. Huang, W. Wang, H. Zhong, S. Feng, and Y. Sun, "Masked label prediction: Unified message passing model for semi-supervised classification," 2020, *arXiv:2009.03509*.

[39] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, "Automated accelerator optimization aided by graph neural networks," in *Proc. 59th ACM/IEEE Des. Automat. Conf. (DAC)*, 2022, pp. 55–60.

[40] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, "Robust GNN-based representation learning for HLS," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 1–9.

[41] A. Sohrabizadeh, J. Wang, and J. Cong, "End-to-end optimization of deep learning applications," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2020, pp. 133–139.

[42] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "AutoDSE: Enabling software programmers to design efficient FPGA accelerators," *ACM Trans. Des. Automat. Electron. Syst. (TODAES)*, vol. 27, no. 4, pp. 1–27, 2022.

[43] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang, "Accurate operation delay prediction for FPGA HLS using graph neural networks," in *Proc. 39th Int. Conf. Comput. Aided Des. (ICCAD)*, 2020, pp. 1–9.

[44] Z. Wang and B. C. Schafer, "Machine leaming to set meta-heuristic specific parameters for high-level synthesis design space exploration," in *Proc. 57th ACM/IEEE Des. Automat. Conf. (DAC)*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 1–6.

[45] N. Wu, Y. Xie, and C. Hao, "IronMan-pro: Multi-objective design space exploration in HLS via reinforcement learning and graph neural network based modeling," in *Proc. IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 42, no. 3, pp. 900–913, Mar. 2023.

[46] N. Wu, H. Yang, Y. Xie, P. Li, and C. Hao, "High-level synthesis performance prediction using GNNs: Benchmarking, modeling, and advancing," in *Proc. 59th ACM/IEEE Des. Automat. Conf. (DAC)*, 2022, pp. 49–54.

[47] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, 2020.

[48] Z. Xie et al., "RouteNet: Routability prediction for mixed-size designs using convolutional neural network," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 1–8.

[49] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," in *Proc. Int. Conf. Mach. Learn. (PMLR),* 2018, pp. 5453–5462.

[50] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Proc. 55th Annu. Des. Automat. Conf. (DAC)*, 2018, pp. 1–6.

[51] T. Yuki and L.-N. Pouchet, "PolyBench/C," Accessed: 2024. [Online]. Available: https://web.cse.ohio-state.edu/pouchet.2/software/polybench/

[52] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *IEEE*, vol. 109, no. 1, pp. 43–76, 2020.