Robust GNN-based Representation Learning for HLS

Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong
Computer Science Department, University of California - Los Angeles, USA
{atefehsz, yba, yzsun, cong}@cs.ucla.edu

2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323853

***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[1].**

I Introduction

In recent decades, the emergence of domain-specific accelerators (DSAs) has provided a viable solution to the end of Dennard's scaling [9]. Consequently, the field-programmable gate array (FPGA) has become an appealing option for reconfigurable, energy-efficient high-performance computing (e.g., [12], [38]). 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) [8] 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 [35], [37], [39], [43]. 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 $9000\times$ speedup reported in [5], identifying the optimal combination of pragmas remains a challenging task [5], [13], [39].

To address this problem, several previous works, as summarized in [35], have treated the HLS tool as a black box and focused on developing efficient heuristics to explore the solution space more intelligently. Notably, AutoDSE [39] 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 [35], [39]. Recent research has demonstrated that leveraging learning algorithms can mitigate this problem. Graph neural networks (GNNs) [45] have been found to be highly effective in the electronic design automation (EDA) domain [10], [16], [20], [34], [37], [40]. 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 [23], ChatGPT [30], and GPT-4 [31] 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 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*

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

appears when there is a need to predict a new objective that was not included in the model's training. Bai *et al.* [4] discuss how we can leverage meta-learning techniques to tackle domain shift. Thus, in this work, we concentrate on addressing the task shift. A significant source of task shift occurs when the HLS tool is updated, and the heuristics used in these tools change, which, in turn, impact the design's objectives. Fig. 1 showcases the variations in latency and BRAM utilization (skipping the rest of the resources due to space limitations) for a total of 1145 designs during the transition from SDx 2018.3 to Vitis 2020.2, the HLS tools from AMD/Xilinx. The vertical axis represents the objectives obtained using Vitis 2020.2, while the horizontal axis corresponds to the results obtained with SDx 2018.3. To provide a clearer comparison, the outcomes are contrasted with the diagonal line $y = x$. Given the non-trivial cost of regenerating the entire database (which requires running the HLS tool) and retraining the model, it is preferable to transfer the model to the new shift using a smaller dataset. The experimental results demonstrate that our proposed graph representation and model architecture can enhance both the original and the transferred model's performance.

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 design two approaches 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%.
- When utilized in design space exploration (DSE), HARP achieves an average performance improvement of 2.54× compared to the SOTA model-free DSE while operating within a significantly reduced time limit of 25×. Moreover, it outperforms the SOTA model-based approach after transfer learning by 1.31× on average.

II Background

In this section, we first provide an overview of GNNs. We then review the pragmas that define the solution space we need to explore. Finally, we present a summary of GNN-DSE [37], a previously published state-of-the-art work, which we use as our baseline.

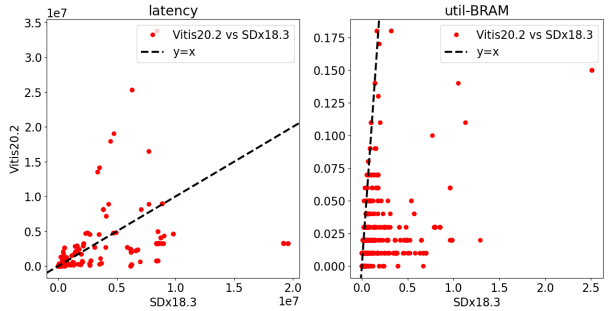


Fig. 1: The design objectives resulted from AMD/Xilinx Vitis 2020.2 over SDx 2018.3. The points are compared against the $y = x$ line.

II-A Graph Neural Networks

A GNN [45] 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:

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

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

II-B HLS Design Space and Pragmas

HARP is developed on top of the open-source AMD/Xilinx Merlin Compiler [7], 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 [5], [39]. The solution space using the Merlin Compiler includes three types of pragmas: `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 [3] or Intel HLS [14] with proper training.

II-C GNN-DSE

The GNN-DSE framework [37] is a state-of-the-art GNN-based approach that predicts the resource utilization and cycle counts of an FPGA design. It starts with a C kernel and generates a design space for optimizing the kernel based on the viable pragma candidates. The C kernel is represented as a graph that captures its control, data, call, and pragma flow. One-hot encoders are used to create the initial node and edge embeddings based on their attributes. The graph is then fed into a GNN encoder which takes in designs from different kernels and learns to assign appropriate node embeddings. The GNN encoder is composed of TransformerConv [36] layers, which are followed by a jumping knowledge network (JKN) [47] that can flexibly pick different ranges of neighborhoods for each node. The node embeddings are merged via a node attention layer, which assigns an importance score to each node and uses them as weights to pool their embeddings. The resulting graph-level embedding is then fed into several multi-layer perceptrons (MLP) networks to predict the final objectives. The open-source implementation of GNN-DSE is our baseline in this paper.

III 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 [13]. 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 [4], [24], [37], [40], [43], logic synthesis [29], [48], placement and routing in physical design [1], [19], [20], [25], [28], [46], and design verification [41]. Huang *et al.* [13] 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 [16], [26], [34]. This is also true for the HLS problem, where analytical models cannot achieve acceptable accuracy [35], [39], 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 [35], recent studies have shown that GNNs can significantly improve accuracy [4], [37], [40], [43], [44]. 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 [37] 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.* [4] extend GNN-DSE by presenting a meta-learning-based framework to adapt to domain changes. Ustun *et al.* [40] 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 [43] 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) [18]. Wu *et al.* [44] 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 [42], [43]. Unlike the prior works that use general-purpose heuristics [35] or dedicated heuristics [39] 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 [5], 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 [4], [37]. In this work, we aim to pinpoint the challenges associated with developing such a model and propose solutions to address them.

IV HARP Methodology

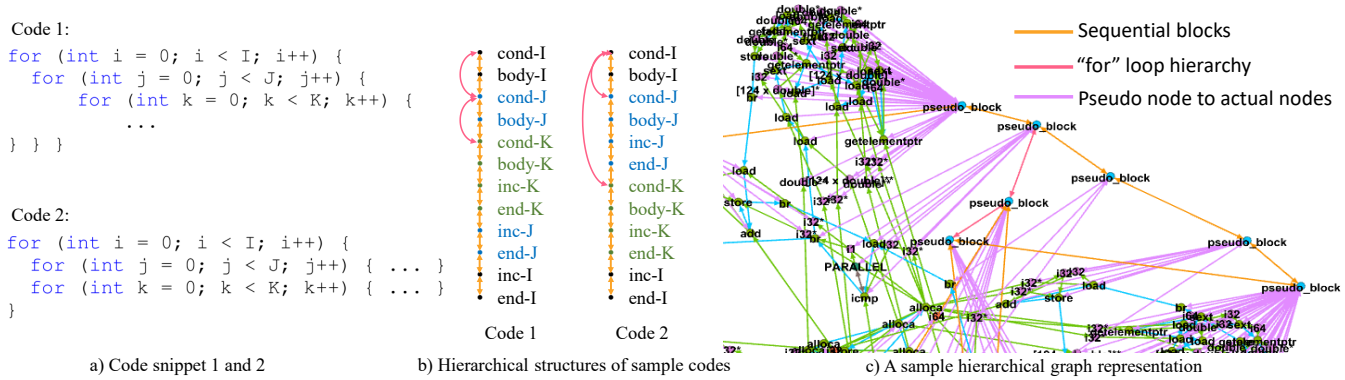
The objective of our study is to enhance the efficiency of exploring the HLS design space by developing a model capable of predicting the behavior of the HLS tool. As mentioned in Section II-B, our solution space consists of three types of pragmas (`PIPELINE`, `PARALLEL`, and `TILE`) which are considered as transformations T applied to the program (i.e., kernel) P. In this context, HARP includes a novel hierarchical graph representation, introduced in Section IV-A, which facilitates the propagation of graph information throughout the graph. Furthermore, HARP utilizes an advanced model architecture to increase the accuracy of the prediction. Applying traditional machine learning models to determine the objectives may erroneously carry the correlation between program P and transformations T in the collected data into the prediction. In contrast, HARP individually learns the impact of each component, as we discuss in Section IV-B. In Section IV-C, we explain that this attribute can also be advantageous when moving to new tasks which cause shifts in the data distribution, as the model can adapt more easily to the shift.

IV-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 [10], [22]. 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 [21] level and a low-level view that relies solely on the LLVM IR level. We leverage the graph representation provided by GNN-DSE to build the graph from LLVM IR and extend it to incorporate two additional abstraction levels of the program.

In GNN-DSE, separate nodes represent the instructions and their operands (data) in LLVM IR, and they are connected according to the control, data, and call flow of the program. The pragmas are modeled as extra nodes that link to the `icmp` instruction of their respective 'for' loop, where the optimization pragmas are applied. 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(a) and (b) 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.



a) Code snippet 1 and 2 b) Hierarchical structures of sample codes c) A sample hierarchical graph representation

Fig. 2: (a) Two sample code snippets; (b) The hierarchical structures of the two sample code snippets, showing only the pseudo nodes and the connection between them; (c) A sample hierarchical graph focusing on demonstrating the pseudo nodes and their connections.

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 (b) 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(c) 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 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 40 unique kernels), the average shortest path between every two nodes in the graph is reduced from 25 (24.2) for the original graph to 5 (4.9) for the hierarchy graph, on average (the geometric mean).

IV-B Decoupling Program and Transformation

The input to HLS tools is composed of two primary components that significantly influence the final microarchitecture. 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-B, 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 [37] 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 IV-B1 and IV-B2, we present two distinct approaches for implementing such a modeling strategy.

IV-B1 Separating Vector Representation of Program and Transformation

Fig. 3 depicts the model architecture for separating the vector representation of P and T. As in GNN-DSE (reviewed in Section II-C), we start with encoding the node and edge attributes using one-hot encoders. The graph is then passed through a series of GNN layers and a JKN which lets the model dynamically adjust the range of neighborhood for each node and has been shown to be effective for improving the performance of this problem [37]. 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. 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. Formally, the computation here can be modeled as:

$$\forall \mathcal{V} \in \mathcal{P}, \mathcal{T} \quad \vec{h_{\mathcal{V}}} = \sum_{i \in N_{\mathcal{V}}} \text{softmax}\left(\text{MLP}(\vec{h_i})\right) \cdot \vec{h_i} \qquad (2)$$

where $\mathcal{V}$ can denote either the program context $\mathcal{P}$ or the transformation context $\mathcal{T}$ and $N_{\mathcal{V}}$ designates the set of nodes that are in the context of $\mathcal{V}$.

To make the T vector ($\vec{h_{\mathcal{T}}}$) more meaningful, we utilize an autoencoder [11] 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:

$$l_T = l_{CE}\left(\mathbf{AE}(\vec{h_{\mathcal{T}}}), \vec{\theta}\right) + \sum_{o \in obj} l_{MSE}\left(\mathbf{F}_o(P,T), H_o(P,T)\right) \qquad (3)$$

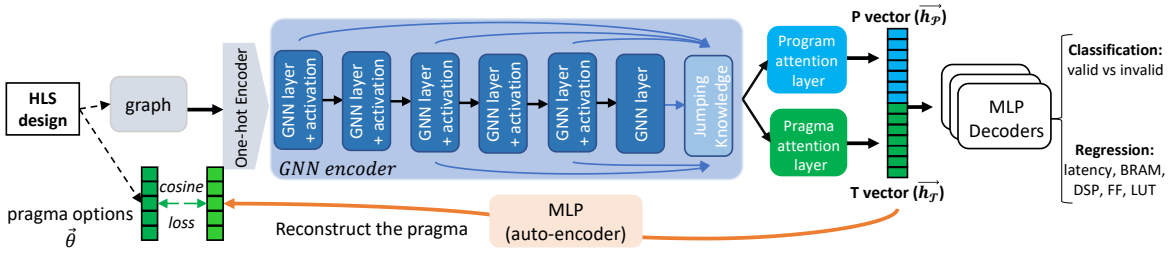
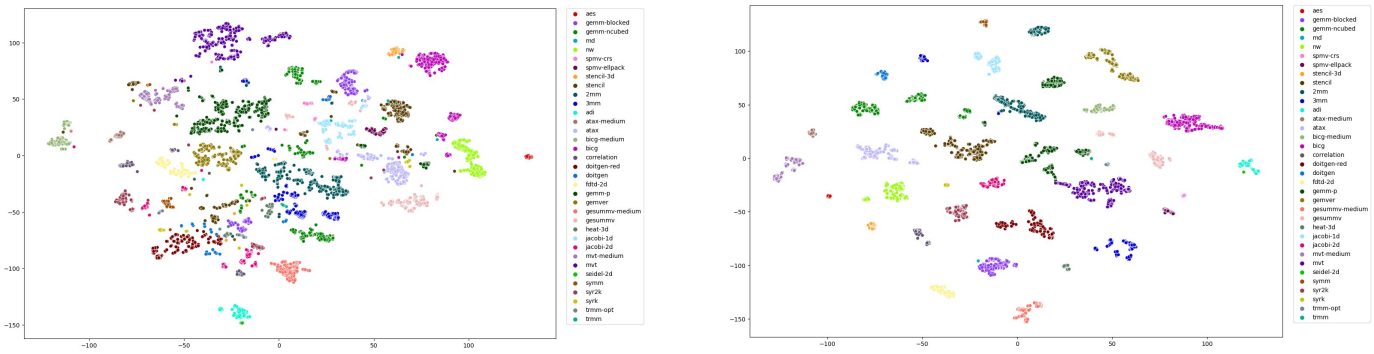


Fig. 3: Separating the vector representation of the program P and its transformation T. Distinct vectors are generated for each one. A further reconstruction loss with an autoencoder is used to enhance the influence of pragmas on the T vector.



(a) The graph-level embedding generated by GNN-DSE [37].

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

Fig. 4: t-SNE visualization of the generated embeddings that are color-coded by the kernel name.

where l_{CE} and l_{MSE} denote the cosine error and mean squared error, respectively. $\mathbf{AE}(\vec{h_T})$ 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 [27] visualizations of the embeddings generated by GNN-DSE and our proposed P vector ($\vec{h_P}$) 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 $\vec{h_P}$ decreases by 3.7× and 2.5× respectively, compared to the embeddings generated by GNN-DSE. These findings highlight the effectiveness of $\vec{h_P}$ in understanding the program scope and its semantics.

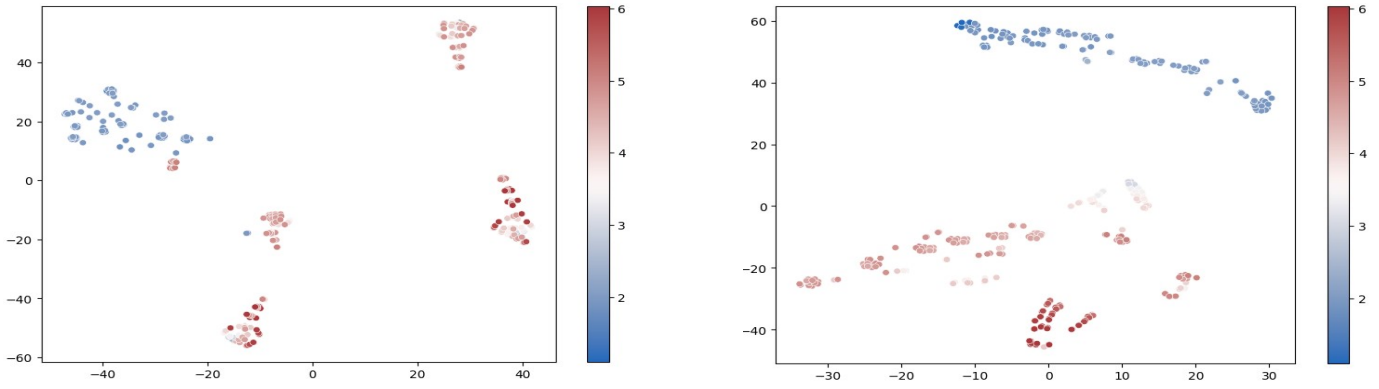
Furthermore, Fig. 5 shows the t-SNE visualization of $\vec{h_T}$ for a random kernel, `gemm-blocked` from the MachSuite benchmark [33], which is color-coded based on the `perf` value. The `perf` value represents the log speedup of the design points to a reference latency value. In order to better illustrate the effectiveness of $\vec{h_T}$, 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, $\vec{h_T}$ 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. To do the final prediction, we concatenate these two vectors and pass them to MLP decoders. Like GNN-DSE, we define two types of tasks, classification to predict whether a pragma candidate creates a valid design or not, and regression to predict the latency and resource utilization. Experimental results (Section V-B) reveal that this model can decrease the loss by 10-23%.

IV-B2 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-B). 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 IV-B1 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 IV-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,



(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).

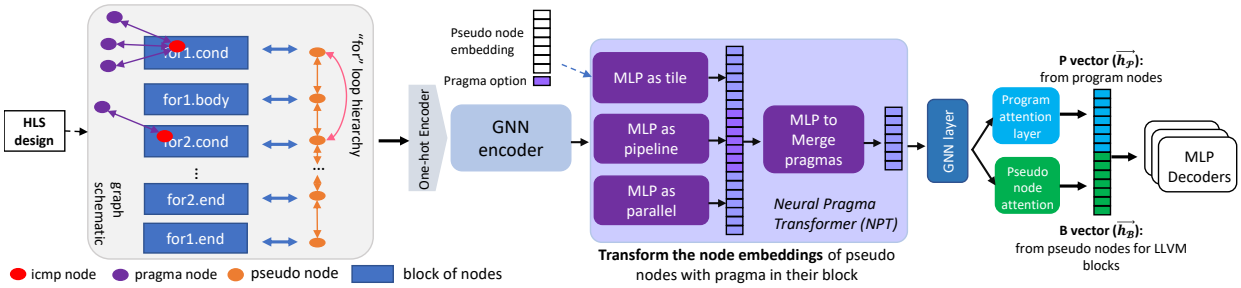


Fig. 6: Modeling pragmas as function transformations using NPT: each pragma type is modeled as a learnable MLP which takes in the embeddings of the pseudo node of the pragma block along with the pragma option. A second level of MLP is used to merge the results.

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 IV-B1, the input one-hot encoder to this GNN encoder does not encode the 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 IV-B1, two vectors are generated with the attention mechanism in Eq. 2 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.

IV-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. Specifically, we use the model trained on the previous version of the tool and fine-tune it to adapt the predictions to the labels of the new version of the tool.

Transfer learning [50] 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. 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. 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.

V Experimental Results

V-A Experimental Setup

Our database includes kernels of intermediate complexity that can be used as building blocks of larger applications. Specifically, we selected 40 kernels from the widely used MachSuite benchmark [33] and the Polyhedral benchmark (PolyBench) [49]. 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`). We use AutoDSE [39] to gather a database of these kernels. For synthesis, we employ two AMD/Xilinx HLS tools, SDAccel 2018.3 [2] and Vitis 2020.2 [3], targeting the Xilinx Alveo U200 FPGA with a frequency of 250MHz. 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 $0.5 * log_2(\frac{1e7}{latency})$ which we call `perf`. Table I presents our database statistics. It is important to note that not all combinations of pragmas yield valid design points. Invalid design points include those with excessively long synthesis times (more than 200 minutes) or cases where either the Merlin Compiler or the HLS tool failed to implement them [37]. The two versions of the database consist of a total of 40 unique kernels, with 22 kernels existing in both versions. When fine-tuning the model for transfer learning, we freeze the first GNN layer and update the rest of the network.

Our framework is implemented and trained using PyTorch [32] on NVIDIA Tesla V100 GPUs. The dataset is split into 70% for training, 15% for validation, and 15% for testing. We employ the Adam optimizer [17] 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 GPU) for the first version of the database and 200 epochs for transfer learning. 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 [36]) 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 [6]. 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.

V-B Model Accuracy

We conducted a series of experiments to evaluate the effectiveness of various components of our approach. Firstly, we retrained the GNN-DSE model using our database as the baseline (M1). Then, we developed M2 by replacing the model architecture of GNN-DSE with our proposed approach described in Section IV-B1. This involved generating separate vector representations for the program (P) and the transformation (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 IV-B2. Note that it also exploits the idea of separating vector representations discussed in Section IV-B1. 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 v1 and v2, respectively. The third scenario involved utilizing the model pre-trained on dataset v1 and fine-tuning it on dataset v2. 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 II 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 [15], 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 v2 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 v1 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 improved 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 II 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.

V-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 same exploration technique as GNN-DSE in searching through the solution space. Specifically, we employ a bottom-up approach that utilizes a Breadth-First Search (BFS) traversal of the pragmas, starting from the innermost loops. This exploration strategy has shown to be very effective for this problem

TABLE I: Statistics of our two databases which consist of 40 unique kernels among which 22 of them are shared in both databases.

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 (v1)	35	23,524/ 8,481	[660 – 94,129,840]	[0 – 12,950]	[0 – 57,531]	[0 – 7,739,313]	[0 – 7,558,355]	[-1.62 – 6.94]	[0 – 2.99]	[0 – 8.41]	[0 – 6.54]	[0 – 3.19]
Vitis 2020.2 (v2)	27	12,168/ 4,569	[992 – 1,453,575,296]	[0 – 3,182]	[0 – 45,056]	[0 – 6,611,687]	[0 – 4,411,806]	[-3.59 – 6.65]	[0 – 0.73]	[0 – 6.58]	[0 – 5.59]	[0 – 1.86]

TABLE II: 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 [37].

Graph	Model	Name	v1 database			v2 database					
			Train from scratch			Train from scratch			Fine-tuned from v1		
			RMSE	MAE	perf tau	RMSE	MAE	perf tau	RMSE	MAE	perf tau
Original	GNN-DSE [37]	M1	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

as it prioritizes the exploration of fine-grained optimizations over coarse-grained ones. HLS tools can usually perform better for such optimizations, making this approach particularly relevant.

We employ HARP for conducting the DSE with a classification model to assist in pruning invalid design points. The classification model achieves an accuracy of 95% on the v1 database and after fine-tuning on the v2 database, gets to an accuracy of 93%. During the DSE, the classification model first determines the validity of the point, and if deemed valid, the regression model assesses its quality. The DSE seeks to optimize the `perf` value (minimize the latency) while ensuring that resource utilizations remain below 80%. 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 200 points. Note that not all of them can finish the synthesis as some of them may be invalid points.

Table III summarizes the DSE results obtained using both versions of the HLS tool. The DSE is conducted on a total of 35 kernels for SDx 2018.3 (v1) and 27 kernels for Vitis 2020.2 (v2). It is important to note that among the 22 kernels shared between the two versions, the average latency of the optimal design in v1 is $5.54\times$ ($1.36\times$ on the geometric mean) higher than that in v2, suggesting improvements in the heuristics of the HLS tool over time. Due to space limitations, we only report the average (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 (v2 kernels), surpassing the performance of GNN-DSE by an average (geometric mean) speedup of $1.31\times$ ($1.33\times$). This validates 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.

Approach	Time Limit	v1 kernels (#:35)		v2 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	$3.61\times$	$1.23\times$	$1.15\times$	$1.05\times$

TABLE III: The performance of the best design found by each DSE with respect to the best one found by AutoDSE [39] in 25h.

VI 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 $5\times$. 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. 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.

Acknowledgement

This work was partially supported by 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/). We would also like to thank Marci Baun for editing the paper.

REFERENCES

[1] 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 *ASP-DAC*. IEEE, 2020, pp. 26–31.

[2] AMD/Xilinx SDAccel - Vivado HLS, "https://docs.xilinx.com/v/u/2018.3-English/ug902-vivado-high-level-synthesis."

[3] AMD/Xilinx Vitis HLS, "https://docs.xilinx.com/v/u/2020.2-English/ug1416-vitis-documentation."

[4] Y. Bai, A. Sohrabizadeh, Y. Sun, and J. Cong, "Improving GNN-based accelerator design automation with meta learning," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1347–1350.

[5] 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.

[6] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.

[7] J. Cong, M. Huang, P. Pan, Y. Wang, and P. Zhang, "Source-to-source optimization for HLS," in *FPGAs for Software Programmers*, 2016, pp. 137–163.

[8] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS Today: Successes, Challenges, and Opportunities," *ACM TRETS*, vol. 15, no. 4, pp. 1–42, 2022.

[9] 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 Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[10] 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 *DAC*, 2022, pp. 1207–1212.

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

[12] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs," in *ICCAD*. IEEE, 2021, pp. 1–9.

[13] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen, J. Wu, Y. Xu, H. Zhang, K. Zhong *et al.*, "Machine learning for electronic design automation: A survey," *ACM TODAES*, vol. 26, no. 5, pp. 1–46, 2021.

[14] Intel High-Level Synthesis Compiler, "https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html."

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

[16] B. Khailany, H. Ren, S. Dai, S. Godil, B. Keller, R. Kirby, A. Klinefelter, R. Venkatesan, Y. Zhang, B. Catanzaro *et al.*, "Accelerating chip design with machine learning," *IEEE Micro*, vol. 40, no. 6, pp. 23–32, 2020.

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

[18] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *ICLR*, 2017.

[19] R. Kirby, S. Godil, R. Roy, and B. Catanzaro, "CongestionNet: Routing congestion prediction using deep graph neural networks," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2019, pp. 217–222.

[20] 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 *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 337–342.

[21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on CGO*, 2004.

[22] Q. Li, Z. Han, and X.-M. Wu, "Deeper insights into graph convolutional networks for semi-supervised learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[23] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[24] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *DAC*, 2013, pp. 1–7.

[25] Y.-C. Lu, S. Pentapati, and S. K. Lim, "VLSI placement optimization using graph neural networks," in *Proceedings of the 34th Advances in Neural Information Processing Systems (NeurIPS) Workshop on ML for Systems, Virtual*, 2020, pp. 6–12.

[26] Y. Ma, Z. He, W. Li, L. Zhang, and B. Yu, "Understanding graphs in EDA: From shallow to deep learning," in *Proceedings of the 2020 International Symposium on Physical Design*, 2020, pp. 119–126.

[27] L. v. d. Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[28] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi *et al.*, "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.

[29] 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 *ICCAD*. IEEE, 2019, pp. 1–6.

[30] OpenAI, "OpenAI: Introducing ChatGPT https://openai.com/blog/chatgpt," 2022.

[31] ——, "GPT-4 Technical Report," 2023.

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, 2019.

[33] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *IISWC*, 2014.

[34] H. Ren, S. Nath, Y. Zhang, H. Chen, and M. Liu, "Why are Graph Neural Networks Effective for EDA Problems?" in *ICCAD*, 2022, pp. 1–8.

[35] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE TCAD*, 2019.

[36] Y. Shi, Z. Huang, W. Wang, H. Zhong, S. Feng, and Y. Sun, "Masked label prediction: Unified message passing model for semi-supervised classification," *IJCAI*, 2021.

[37] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, "Automated Accelerator Optimization Aided by Graph Neural Networks," in *2022 59th ACM/IEEE Design Automation Conference (DAC)*, 2022.

[38] A. Sohrabizadeh, J. Wang, and J. Cong, "End-to-End Optimization of Deep Learning Applications," in *FPGA*, 2020, pp. 133–139.

[39] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 27, no. 4, pp. 1–27, 2022.

[40] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang, "Accurate operation delay prediction for FPGA HLS using graph neural networks," in *ICCAD*, 2020.

[41] F. Wang, H. Zhu, P. Popli, Y. Xiao, P. Bodgan, and S. Nazarian, "Accelerating coverage directed test generation for functional verification: A neural network-based framework," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, 2018, pp. 207–212.

[42] Z. Wang and B. C. Schafer, "Machine leaming to set meta-heuristic specific parameters for high-level synthesis design space exploration," in *DAC*. IEEE, 2020, pp. 1–6.

[43] 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," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[44] N. Wu, H. Yang, Y. Xie, P. Li, and C. Hao, "High-level synthesis performance prediction using gnns: Benchmarking, modeling, and advancing," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 49–54.

[45] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[46] Z. Xie, Y.-H. Huang, G.-Q. Fang, H. Ren, S.-Y. Fang, Y. Chen, and J. Hu, "RouteNet: Routability prediction for mixed-size designs using convolutional neural network," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[47] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," in *ICML*. PMLR, 2018, pp. 5453–5462.

[48] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

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

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