Automated Accelerator Optimization Aided by Graph Neural Networks

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

Abstract

Using High-Level Synthesis (HLS), the hardware designers must describe only a high-level behavioral flow of the design. However, it still can take weeks to develop a high-performance architecture mainly because there are many design choices at a higher level to explore. Besides, it takes several minutes to hours to evaluate the design with the HLS tool. To solve this problem, we model the HLS tool with a graph neural network that is trained to be used for a wide range of applications. The experimental results demonstrate that our model can estimate the quality of design in milliseconds with high accuracy, resulting in up to 79× speedup (with an average of 48×) for optimizing the design compared to the previous state-of-the-art work relying on the HLS tool.

ACM Reference Format:
Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2022. Automated Accelerator Optimization Aided by Graph Neural Networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530409

1 Introduction

High-Level Synthesis (HLS) was introduced to simplify the FPGA programming by raising the abstraction level in design and soon was embraced by both academia and industry [4, 16]. This is because the HLS tools let the designers optimize their microarchitecture quickly by inserting a few synthesis directives in the form of pragmas. This feature can potentially help shorten the design development cycle. However, not every HLS design has a good quality of results [17]. Thus, one often has to explore many design choices for each new application since the solution space grows exponentially by the number of candidate pragmas. This can negatively impact the design turn-around times.

To speed up the design optimization, a new line of research has been created with the focus on automating the design space exploration (DSE) for optimizing the microarchitecture. As summarized in [14], the previous studies either use the HLS tool directly [17, 24], or develop a model to mimic the HLS tool [11, 26] for evaluating a design point. Relying on the HLS tool to evaluate a solution can increase the DSE time significantly as each design candidate would have a long evaluation time (minutes to hours) that forces us to explore a reduced set of the solution space. While utilizing a model can potentially speed up the process, a simple analytical model cannot capture the different heuristics used by the tool [14]. Adopting a learning algorithm can help with increasing the accuracy. However, the related works build a separate learning model per application and the results from one application are not transferred to another one. A nice effort was made in Kwon et al. [7] for transfer learning using a Multi-Layer Perceptron (MLP) network. Nonetheless, they only use the pragma configurations as the input to the model, which can result in considerable loss since the program semantics are missing (see Section 5.2).

A few of the very recent works have proposed to use Graph Neural Network (GNN) for predicting the design's quality [18, 21]. Ustun et al. [18] proposes a GNN-based model to learn the operation mapping to FPGA's resources for delay prediction in HLS. IronMan [21] uses GNN to predict the performance of the program under different resource allocations (DSP or LUT) to the computation nodes. Although their studies clearly demonstrate the value and power of using GNNs, none of these works include the pragmas in their input representation so their models cannot be used for finding the best design configuration.

In this paper, we aim to automate the design optimization using GNN with the support for *model generalization* by developing a framework called GNN-DSE. We first build a model to evaluate a design quickly, in milliseconds, without the invocation of the HLS tool. Since the HLS tools employ many heuristics to optimize a design and the design parameters affect each other, we let a deep learning model learn their impact. Furthermore, as the current HLS tools optimize the design based on specific code patterns, it is important to identify the different code patterns and learn their effect to be able to transfer the knowledge we gained from one application to another. As such, we represent the program as a graph which includes the program information in the form of control, data, call, and pragma flows and exploit a GNN to extract the required features of the graph for predicting the objectives. We propose several techniques for improving the accuracy of the model including Jumping Knowledge Network (JKN) [23], node attention [9], and multi-head objective prediction. To demonstrate the effectiveness of our model, we build a DSE on top of it to find the Pareto-optimal design points. We show that not only can GNN-DSE find the Pareto-optimal design points for the kernels that were included in its training set, it can also generalize to the kernels outside of its database and detect their Pareto-optimal design points. This paper is the first work to employ a graph representation that captures both the program semantics and the pragmas, and to build a *single* predictive model for several applications with transferring learning capability. In this paper, we target Xilinx FPGAs as an example but our approach is tool-independent and extendable to Intel FPGAs as well.

In summary, this paper makes the following contributions:

- We propose a graph-based program representation for optimizing FPGA designs which includes both the program context and the pragma flow.
- We develop a learning model based on Graph Neural Network (GNN) as a surrogate of the HLS tool for assessing a design point's quality in milliseconds and propose several techniques for improving its accuracy.
- We build an automated framework, GNN-DSE[1], to gather a database of FPGA designs, train a learning model for predicting the design's objectives, and run a design space exploration based on the model to close-in on a high-performance design point.
- The experimental results demonstrate that not only can GNN-DSE find the Pareto-optimal design points for the kernels in its database, but can also optimize the *unseen* kernels by generalizing the knowledge it learned from its training set.

[1]The codes are open-sourced at https://github.com/UCLA-VAST/GNN-DSE

2 Background

2.1 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 intermediate representation (IR) in LLVM [8]. 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.

2.2 Graph Neural Networks

A Graph Neural Network (GNN) [22] extracts the graph information by learning the features (embeddings) of each node in the graph via a series of layers, in which it aggregates (AGG) the neighboring nodes' ($\mathcal{N}(i)$) information and applies a *transformation function* (TF) on the aggregated result. *Graph Convolutional Network* (GCN) [6] is a popular form of a GNN which adopts a simple AGG function that performs a weighted summation of the embeddings of $\mathcal{N}(i)$ using the degree (d_i) of the nodes:

$$\boldsymbol{h}_i' = \sigma(\boldsymbol{W} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{d_j d_i}} \boldsymbol{h}_j) \tag{1}$$

where $\boldsymbol{h}_i \in \mathbb{R}^F$ ($\boldsymbol{h}_i' \in \mathbb{R}^{F'}$) denotes the input (output) embeddings of node i which is a vector of F (F') features. $\boldsymbol{W}$ is a trainable weight matrix for the TF step to act as a filter, and σ is an activation function to introduce non-linearity to the model. Here, as the AGG step employs a fixed set of weights (determined by the degree of the nodes), the model has no way of prioritizing any of the neighbors to learn better embeddings. *Graph Attention Networks* (GAT) [20] were introduced to learn the *importance* (*attention*) of the node's neighbors so that they can contribute to updating its embeddings accordingly. The computation of a GAT layer can be seen as:

$$\boldsymbol{h}_i' = \sigma(\mathbf{W} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{i,j} \boldsymbol{h}_j) \tag{2}$$

$\alpha_{i,j}$s are the attention coefficients computed by multi-head dot-product attention. The computation for each head is as follows:

$$\alpha_{i,j} = \frac{\exp\left(\text{LeakyReLU}\left(\boldsymbol{a}^\top [\boldsymbol{W}\boldsymbol{h}_i \,\|\, \boldsymbol{W}\boldsymbol{h}_j]\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\text{LeakyReLU}\left(\boldsymbol{a}^\top [\boldsymbol{W}\boldsymbol{h}_i \,\|\, \boldsymbol{W}\boldsymbol{h}_k]\right)\right)} \tag{3}$$

where $\|$ denotes the concatenation operation and $\boldsymbol{a}$ is a learnable vector controlling the attention that node i receives from node j. Note that compared to a GCN only the AGG step is changed.

2.3 The Merlin Compiler

The Merlin Compiler [1, 2], recently open-sourced by Xilinx, was developed to make FPGA programming easier by introducing a reduced set of high-level pragmas for optimizing the design. Based on these pragmas, it performs source-to-source code transformation and automatically generates the respective HLS code along with the required HLS pragmas. It also automatically employs code transformations to implement memory coalescing, apply memory burst, and cache the required data based on the architectural optimizations. While it only uses three pragmas (`pipeline`, `parallel`, and `tile`), it generates several HLS pragmas based on them, including `pipeline`, `unroll`, `array_partition`, `inline`, `dependence`, `loop_flatten`. We build our tool on top of the Merlin Compiler to not only greatly reduce the solution space size, but also apply its automated code transformations to achieve a better design. Nonetheless, the GNN model must work harder to learn when (and where) the Merlin Compiler applies its automated optimizations.

3 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 (θ). 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 $Util(\mathbf{H}, \mathcal{P}(\theta))$:

$$\mathbf{Q}_\mathbf{H}(\mathcal{P}(\theta)) = \left(Cycle(\mathbf{H}, \mathcal{P}(\theta)), Util(\mathbf{H}, \mathcal{P}(\theta))\right) \tag{4}$$

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

$$\min_{\mathbf{F}} \left(\underset{\theta}{average} \left(Loss(\mathbf{Q}_\mathbf{F}(\mathcal{P}(\theta)), \mathbf{Q}_\mathbf{H}(\mathcal{P}(\theta)))\right)\right) \tag{5}$$

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. Formally, our objective is:

$$\min_{\theta} Cycle(\mathbf{F}, \mathcal{P}(\theta)) \tag{6}$$

$$\text{subject to} \quad \theta \in \mathbb{R}_\mathcal{P}, \quad \forall u \in Util(\mathbf{F}, \mathcal{P}(\theta)), u < T_u \tag{7}$$

where u is the utilization of one type of the FPGA on-chip resources and T_u is a user-defined threshold for that type on the FPGA.

4 Our Proposed Methodology

Fig. 1(a) depicts a high-level overview of GNN-DSE which operates in three modes: training, inference, and DSE. We first collect a database from various applications (Section 4.1) and represent each design in the database as a graph (Section 4.2). Then, we train a *predictive model* for estimating the design's objectives (Section 4.3). Finally, the *predictive model* can be used as a surrogate to the HLS tool to run the *inference* and *DSE* stages (Section 4.4).

4.1 Database Generation

We adapt a related prior work, AutoDSE [17], which reported superior results over previous studies (e.g. [24]), to generate the initial database for each of the applications. Fig. 2 demonstrates our approach for it. Each `for` loop can take up to three pragmas: `pipeline`, `parallel`, and `tile` (Section 2.3). We also exploit AutoDSE's rules for pruning a design configuration (e.g., when fine-grained pipelining is applied on a loop, the inner loops would not take any pragma). Since the model needs to see a variety of design points from "bad" to "good" to learn to distinguish them, GNN-DSE extends AutoDSE to exploit three types of explorers for building the training set:

- The existing explorer of AutoDSE, bottleneck-based optimizer, which can find high-quality designs.
- A hybrid explorer combining the bottleneck-based optimizer with a local search, which evaluates up to P neighbors of the best design point after $X\%$ improvement in its quality. Thus, the model can see the effect of modifying only one of the pragmas.
- A random explorer which may consider those configurations skipped 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 4.2). GNN-DSE gradually collects results from *different* applications in a shared space to be used for training the model.

4.2 Program Representation

As mentioned in Section 2.1, CDFG is a popular choice for representing FPGA kernels. On the downside, the CDFGs ignore the precision of the operands and their values, which are crucial in determining design's objectives. Recently, a more convenient program representation is proposed, ProGraML [3], which extends the CDFG by explicitly assigning separate nodes to operands to retrieve the missing information. It also keeps the function hierarchies by including

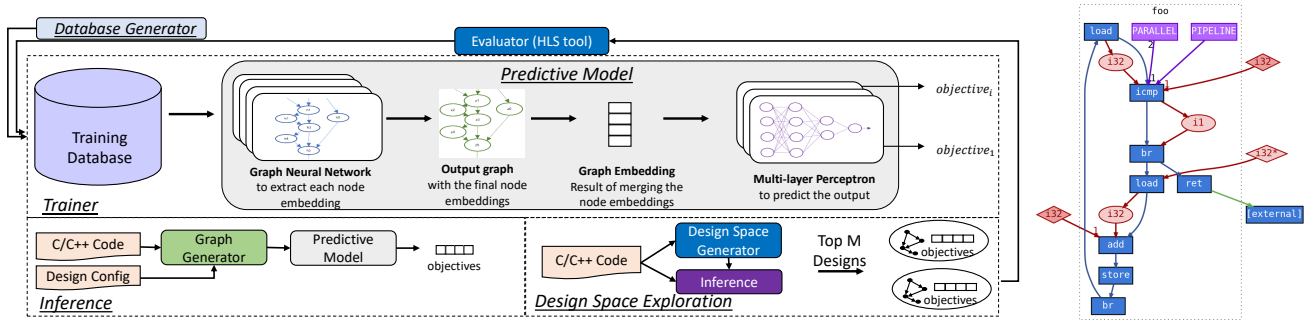

Figure 1: (a) High-level overview of the GNN-DSE framework; (b) The graph representation of Code 1.

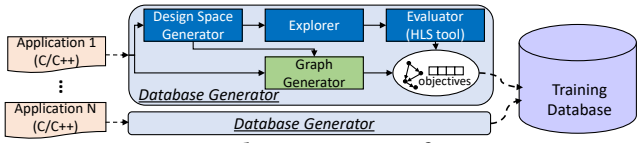

Figure 2: Database generator of GNN-DSE.

the design's call flow. As such, we adapt ProGraML and extend it by including the *pragma* flow to represent a program. Each of the candidate pragmas are defined in either of the following forms:

```
#pragma ACCEL pipeline auto{pragma_name}
#pragma ACCEL parallel factor=auto{pragma_name}
#pragma ACCEL tile factor=auto{pragma_name}
```

the `pragma_name` is a placeholder for its option, which can be `(off|cg|fg)` for `pipeline` and a numerical value for the other two. cg (fg) refers to coarse-grained (fine-grained) pipelining [17].

Code 1: Code snippet of an input toy example to GNN-DSE.

```
1 void foo(int input[N]) {
2 #pragma ACCEL pipeline auto{_PIPE_L1}
3 #pragma ACCEL parallel factor=auto{_PARA_L1}
4   for (int i = 0; i < N; i++) { input[i] += 1; } }
```

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`. Code 1 shows a toy example having a simple `for` loop with two candidate pragmas. Fig. 1(b) depicts its graph representation. We only show the relevant nodes here for illustration purposes. As Fig. 1(b) demonstrates, there are three types of nodes in each graph. The first kind (in blue) is for the LLVM instructions that together demonstrate the control flow of the program. The second kind (in red) exhibits the constant values and variables that capture the data flow of the program. The pragma nodes (third kind) are presented as purple boxes connecting to the respective `icmp` node. The edges also have different kinds which show the different flows of the graph: control (blue), data (red), call (green), and pragma (purple). When there are two or more edges of the same type connected to a node, they are numbered to further distinguish them (see the edges connecting from pragma nodes to the `icmp` node).

To distinguish the different candidate pragmas of a nested loop, one must know the loop level for each pragma. As a rule of thumb, the HLS tools perform better when the pragmas are applied to inner loops since they can implement fine-grained optimizations easier [17]. Besides capturing the control flow, we explicitly encode this information in each node via the LLVM block ID of the `for` loop. More specifically, each node / edge has the following attributes:

```
Node = {'block': LLVM block ID, 'key_text': Node key task, 'function': Function
       ID, 'type': Node type}
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
flow	0: control	1: data	2: call	3: pragma
position	0: tile	1: pipeline	2: parallel	-

The `key_text` attribute shows a keyword corresponding to that node. For example, `PIPELINE`, `load`, `i32*` for each of the pragma, control, and data node types. For each design point, the `auto` variables in the pragma placeholders are replaced with their respective values. Hence, among the graphs for different design configurations of the same application, only the attributes of their pragma nodes are different. Fig. 3 demonstrates the graph generation process.

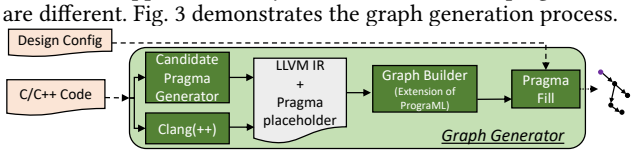

Figure 3: Graph generator of GNN-DSE.

4.3 Predictive Model

Fig. 4 depicts our model architecture for predicting the design's objectives. It takes the graph representation of the program as the input and creates the initial node/edge embeddings by concatenating the one-hot encoding of their attributes (Section 4.2) and the pragma options. This encoding helps the model assign a higher weight to the attributes that contribute more to the final prediction. For this matter, the model exploits a *GNN encoder* (Section 4.3.1) to update the embeddings. The GNN encoder, then, passes the graph embeddings to a set of MLPs to estimate the outputs (Section 4.3.2).

4.3.1 GNN Encoder: It assigns $\boldsymbol{h}_{\mathcal{G}} \in \mathbb{R}^D$ to a graph $\mathcal{G}$ via three stages: (1) stacked TransformerConv layers to produce node embeddings, (2) a Jumping Knowledge Network for combining the output of different layers to make the final node embeddings with dynamic ranges of neighborhoods, and (3) an attention mechanism to merge the node-level embeddings into a graph-level embedding.

TransformerConv: We reviewed GCN [6] and GAT [20] in Section 2.2. One drawback of these layers is that they both overlook the edge embeddings. TransformerConv [15], inspired by the Transformer model [19], is a state-of-the-art GNN architecture, which builds attention coefficients ($\alpha_{i,j}$) for aggregating the neighbors in a different manner than GAT:

$$\alpha_{i,j} = \text{softmax}\left(\frac{(\mathbf{W}_1\boldsymbol{h}_i)^\top(\mathbf{W}_2\boldsymbol{h}_j + \mathbf{W}_3\boldsymbol{e}_{ij})}{\sqrt{D}}\right) \tag{8}$$

where $\boldsymbol{W}_1$, $\boldsymbol{W}_2$, and $\boldsymbol{W}_3$ are learnable weight matrices, and $\boldsymbol{e}_{ij}$ denotes the embedding of the edge between nodes i and j. Including edge attributes is a desirable feature for our task since the edges in our graph representation contain useful information (Section 4.2).

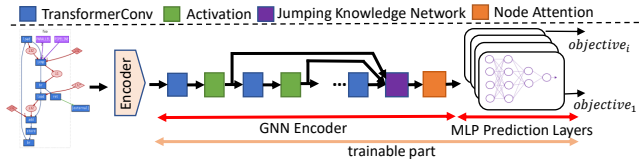

Figure 4: The architecture of GNN-DSE's predictive model.

In addition, TransformerConv makes use of gated residual connections when updating the node embeddings that can prevent the model from over-smoothing. Consequently, we adopt TransformerConv as the basic building block of our model.

Jumping Knowledge Network (JKN): Each layer of a GNN gathers the embeddings of the first-order neighbors. By adding each layer, the nodes will receive the embeddings from one hop further since their first-order neighbors are now updated with theirs. The different nodes in the graph may need information from different ranges of neighborhoods. For example, in the graph of Fig. 1(b), the `load` and `add` nodes are affected by the pragma nodes after 3 and 4 layers, respectively. To fully leverage the embeddings generated by different layers of the GNN model, we exploit JKN [23] which as Fig. 4 illustrates, takes in the output of all the layers to flexibly pick different ranges of neighborhood for each node:

$$\boldsymbol{h}_i = \max\left(\boldsymbol{h}_i^{(1)}, \ldots, \boldsymbol{h}_i^{(T)}\right) \tag{9}$$

where $\boldsymbol{h}_i^{(k)}$ denotes the embedding of node i after the k-th layer.

Node attention-based graph-level embedding generation: To generate one vector representation for the entire graph, one can simply add all the node embeddings. However, given the fact that our graph representation contains both the pragma nodes and the program context nodes, it is preferable to introduce attention [9] to learn which node is more important for the prediction tasks:

$$\boldsymbol{h}_{\mathcal{G}} = \sum_{i=1:N} \text{softmax}\left(\text{MLP}_1(\mathbf{h}_i)\right) \cdot \text{MLP}_2(\mathbf{h}_i) \tag{10}$$

where MLP_1 maps the node embedding from $\mathbb{R}^D$ to $\mathbb{R}$ followed by a global softmax to obtain one attention score per node. The attention scores are then applied to the transformed node embeddings, $\text{MLP}_2(\mathbf{h}_i)$, to obtain the final graph-level embedding. Fig. 5 depicts the graph for a design of the stencil kernel in MachSuite benchmark [13]. Each node's circle size is proportional to the attention that its embedding receives in building the graph-level embedding. As we expected, the pragma nodes are among the most *important* nodes. Yet, the model could learn that not all the pragma nodes are equally important. As the figure suggests, the loop trip count (`icmp` node and `i32` node connecting to it) and other contextual information of the loop determine their importance.

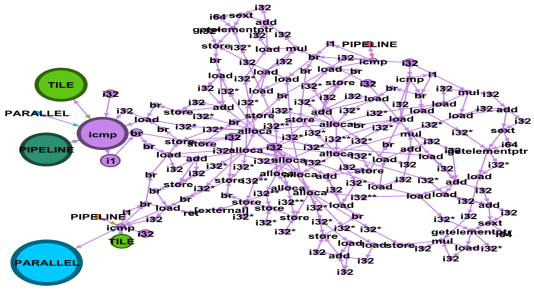

Figure 5: Node attention scores of a design of the stencil kernel. The larger the circle, the higher its attention is.

As the embeddings are high-dimensional vectors (124/64-D vectors for initial/final embeddings), we utilize t-SNE [10] to visualize them. t-SNE is a powerful technique that can model high-dimensional data by 2-D points in a way that nearby (distant) points

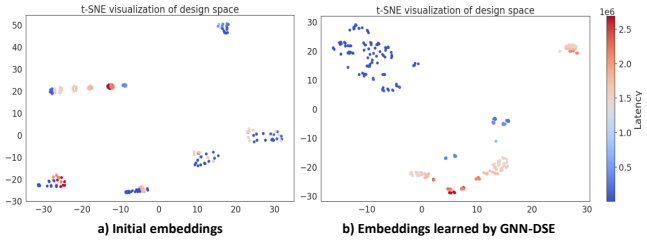

Figure 6: t-SNE [10] visualization of the design configurations of stencil. Each point represents a design with colors indicating its latency value.

model similar (dissimilar) data. Fig. 6(a) depicts the t-SNE plot for the stencil kernel based on its initial embeddings. The node embeddings are added together to create one graph-level embedding. Each point represents a design configuration which is color-coded by its latency (cycle counts). Fig. 6(b) demonstrates the t-SNE plot if we use the graph-level embeddings generated by our GNN encoder instead. While the initial features show high similarity between design points with huge differences in their latencies, the GNN encoder could successfully assign embeddings to the graphs in a way that only the designs with similar latency be clustered together.

4.3.2 MLP Prediction Layers: After encoding the graph as a vector, further transformation is needed to perform the final prediction. We have the following learning tasks for assessing a design point:

- **Classification task for determining whether a design configuration is valid.** The source of invalidity can come from many factors such as: 1) it may be hard for the HLS tool to optimize some of the pragma combinations. If synthesis of a design does not finish in 4 hours, we mark it as invalid; 2) the HLS tool may refuse to synthesize designs that have high parallelization factors; 3) a combination of the pragmas may be infeasible, etc.
- **Regression task to estimate the design's objectives.** Once we identify that a design is valid, we estimate the quality of design by predicting its cycle count and resource utilization.

For each of these tasks, we exploit an MLP to do the prediction based on the graph-level embeddings. Note that our regression task is seeking to predict multiple objectives. Sharing the GNN encoder as the backbone and applying multi-task prediction with different MLPs (as seen in Fig. 4) is desirable here. This is because, as the objectives are correlated with each other, they can help each other in creating a better graph-level embedding.

4.4 Design Space Exploration

Once we have an accurate model, we can use it for finding the Pareto-optimal designs. Since our models can finish in milliseconds, we can explore a large number of design points very quickly. Nevertheless, for enormous solution spaces, we still may not be able to search through the whole space in a timely fashion. Therefore, we set a time limit for running the DSE and employ a heuristic to prioritize searching through the most-promising candidates first. 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 more than one pragma at a loop-level, we prioritize `parallel` over `pipeline` over `tile`. If the picked pragma (A) is dependent on another pragma (B) from the same loop level or one loop level further, we move pragma B up in the ordered list. A pragma has a dependency when we want to prune the invalid design combinations. For example, there is always a dependency between the `parallel` pragma of one loop level with the `pipeline` pragma of its upper loop level. This is because `fg` pipelining completely

unrolls the sub-loops so we no longer need the `parallel` pragma. After evaluating this pragma, we do the same process for the next loop section and continue until all the pragmas are visited.

Since getting the true value of design's objectives are time-consuming, building the dataset is the main bottleneck of our approach. After building an initial database (Section 4.1), we exploit our DSE to augment the database. Note that the DSE wants to run the model on many of the unseen data points so we must have good representatives of all of the design choices in our database. On the other hand, if our DSE mistakenly believes that an unseen design point is good, it means that the model does not have a sufficient set of data to generalize for the whole space. Since these data points are the ones that made the model mispredict the results, they are more likely to build a better dataset in the next round.

5 Evaluation

5.1 Experimental Setup

We choose our target kernels from the commonly-used MachSuite benchmark [13], and the Polyhedral benchmark suite (Polybench) [25]. The initial database is generated as explained in Section 4.1 with the Xilinx Virtex Ultrascale+ VCU1525 as the target FPGA. It consists of kernels with different computation intensities including matrix and vector operations, stencil operation, encryption, and a dynamic programming application (`nw`). Our model predicts the `latency` in the form of *cycle counts*, and the resource utilization for `DSP`, `BRAM`, `LUT`, and `FF`. Our framework is deployed and trained using PyTorch [12]. We use 80% (20%) of the dataset for training (testing), 3-fold cross-validation during training with Adam optimizer [5] and a learning rate of 0.001. The reported models' performance are based on the test set. The initial embeddings have 124 features. We build completely separate models for classification and regression having 6 GNN layers with 64 features followed by 4 MLP layers for each. Table 1 summarizes the number of pragmas, the total solution space size, the total number of configurations in our database, and the number of valid configurations among them for each kernel, in addition to the number of designs after augmenting the database as explained in Section 4.4. In our database, the `latency` is in the range of 660 to 12,531,777 cycles. `DSP`/ `BRAM` / `LUT` / `FF` counts are in the range of 0 / 0 / 913 / 0 to 28,672 / 7,464 / 2,639,487 / 3,831,357, showing a wide range for all the objectives.

5.2 Model Evaluation

5.2.1 Pre-processing the Data: We pre-process our data to limit their ranges so that they can contribute to the loss equally. For this matter, we normalize the resource utilizations by dividing them by the available number of resources on the FPGA and apply the following formula for `latency`:

$$T_{latency} = \log_2 \frac{NormalizationFactor}{latency} \tag{11}$$

therefore, the model spends more time on reducing the loss for large values of $T_{latency}$ which corresponds to low latency values, i.e., the high-performance designs. The $\log_2$ factor is used to make the data distribution more even, as because of the intrinsic features of this problem, the number of high-performance values are limited and the data is originally biased towards low-performance ones. After this normalization, the lower range for all the objectives is 0.0 and the upper range is 12.7414 / 4.1900 / 1.7200 / 2.2300 / 1.6600 for `latency`/ `DSP`/ `BRAM` / `LUT` / `FF` respectively. Our database shows that the `BRAM` utilization has a weak correlation with the rest of the objectives. Consequently, we train two models for regression, one is responsible solely to predict the `BRAM` utilization while the other one predicts the rest of the objectives.

5.2.2 Comparative Studies: We first test the performance of two models which only use an MLP network with no considerations for the graph structure. The first one (M1) follows the same approach as in [7][2] and just uses the pragma settings as the input. The second model (M2) takes all the nodes of the graph with their initial embeddings as the input but does not exploit the GNN techniques for updating the embeddings and rather only uses an MLP. As the results suggest, including the program context in the input is crucial for improving the accuracy of the model since it wants to predict the objectives across applications with different semantics.

Additionally, we assess the effect of our optimizations to the model. We first tested the model's performance when it uses either of the GCN, GAT, or TransformerConv as the GNN layer with normal summation to create the graph-level embeddings (M3 to M5). Then, we added the JKN (M6) and replace the normal summation with our node attention layer (M7). As Table 2 shows, the fact that these models include the different flows (control, data, call, and pragma) of the program using a graph structure can decrease their loss. The results further demonstrate the effectiveness of our optimizations as explained in Section 4.3.1.

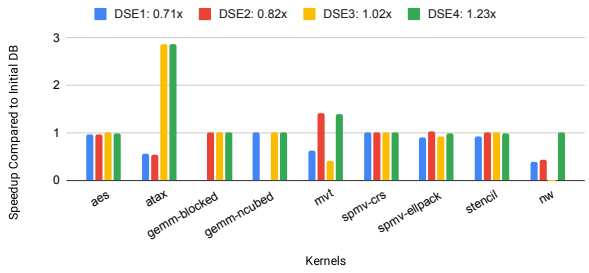

Figure 7: GNN-DSE's speedup compared to the the best design in the initial database. After each round of DSE, the top designs are added to the database to refine the predictions.

5.3 Results of Design Space Exploration

Using our models, we are able to run 22 inferences per second. As a result, we can exhaustively search through all the design choices for our target kernels, except for `mvt`, in a few minutes. We adopt the heuristic proposed in Section 4.4 to search through `mvt` for one hour. We run the DSE on all the kernels and evaluate their top 10 designs using the HLS tool. Depending on how it performs, we add a various number of design points with their true objectives to the database as explained in Section 4.4. Fig. 7 depicts the speedup each kernel achieved compared to the best design in the initial database for different rounds of DSE. As the figure shows, after 3 rounds of database expansion, the DSE can find a design configuration with better or equal performance. The chart's legend summarizes the average speedup of all the kernels after each round.

5.4 Results on Unseen Kernels

To test whether our tool is extensible to unseen kernels, we have chosen four new kernels from Polybench which were not included in our database: `bicg`, `doitgen`, `gesummv`, and `2mm`. `bicg` is doing two matrix-vector multiplications, `doitgen` multiplies a 3-D tensor with a matrix, `gesummv` has two matrix-vector multiplications and a weighted vector addition, and `2mm` consists of two matrix multiplications. Note that four of the kernels in our database are working with matrix-vector operations, although, in general, they have a different problem size and coding structure. Table 3 summarizes the number of pragmas and the design configurations for each of these new kernels. Like in Section 5.3, we set a time limit of one hour for

[2] As the codes are not available, we re-implemented their model as closely as possible.

Table 1: Design space and the database of the kernels used for training our model.

Kernel name	aes	atax	gemm-blocked	gemm-ncubed	mvt	spmv-crs	spmv-ellpack	stencil	nw	Total
# pragmas	3	5	9	7	8	3	3	7	6	-
# Designs configs	45	3,354	2,314	7,792	3,059,001	114	114	7,591	15,288	3,095,613
Initial database (# Total / # Valid)	15 / 15	605 / 101	616 / 149	432 / 149	571 / 180	98 / 35	114 / 60	1,066 / 281	911 / 66	4,428 / 1,036
Final database (# Total / # Valid)	44 / 44	636 / 129	667 / 183	476 / 193	621 / 224	114 / 51	114 / 60	1,098 / 291	982 / 103	4,752 / 1,278

Table 2: Model evaluation on the test set of our database. RMSE loss is used as the evaluation metric for the regression task. For the classification task, the accuracy and F1-score are reported.

Model	Method	Latency	DSP	LUT	FF	BRAM	All	Accuracy	F1 - score
M1	MLP-pragma (as in [7])	3.2756	0.5857	0.3115	0.2483	0.3356	4.7567	0.52	0.42
M2	MLP-pragma-program context	2.9444	0.4650	0.2401	0.1349	0.1597	3.9442	0.78	0.40
M3	GNN-DSE- GCN	1.6825	0.4265	0.1642	0.1277	0.1593	2.5602	0.79	0.51
M4	GNN-DSE- GAT	1.1819	0.2557	0.1266	0.1009	0.1178	1.7829	0.85	0.68
M5	GNN-DSE- TransformerConv	1.1323	0.2540	0.1245	0.0938	0.1231	1.7277	0.85	0.76
M6	GNN-DSE- TransformerConv + JKN	1.0846	0.2521	0.1112	0.0933	0.0912	1.6324	0.92	0.86
M7	GNN-DSE (TransformerConv + JKN + node att.)	**0.5359**	**0.1253**	**0.0762**	**0.0632**	**0.0515**	**0.8521**	**0.93**	**0.87**

2mm, which has more than $492M$ design choices. For the rest of the kernels, we exhaustively search through all their configurations which takes less than 2 minutes. For all of them, we then pass the top 10 designs to the Merlin Compiler and run them in parallel to evaluate them. The 4th column of Table 3 lists the overall runtime of this process for each kernel.

To measure the quality of the top designs generated here, we ran the original explorer of AutoDSE for up to 21 hours (its search for doitgen finished after 3 hours). During this time, it explored up to 163 design configurations for each of the cases achieving a maximum speedup of 350× compared to the design with no optimizations. GNN-DSE could achieve about the same performance (from −2% and +5% difference with a mean of +1%) but in much less time. Table 3 summarizes the speedup in running the DSE and synthesizing the design with HLS that GNN-DSE achieved for each kernel compared to AutoDSE. The results show that GNN-DSE can accelerate this process by up to 79× with an average of 48×.

Table 3: GNN-DSE's performance on unseen kernels. The speedup numbers are with respect to a prior state-of-the-art work, AutoDSE [17], after running it for up to 21 hours.

Kernel	#pragma	#Design configs	DSE + HLS runtime (m)	#Explored	Runtime speedup
bicg	5	3,536	18	3,536	69×
doitgen	6	179	16	179	11×
gesummv	4	1,581	16	1,581	79×
2mm	14	492,787,501	74	78,676	17×

6 Conclusion

In this work, we developed a push-button framework, GNN-DSE, to build a learning model for predicting the design's objectives in milliseconds. We proposed a graph-based program representation which includes both the program semantics and the candidate pragmas and implemented a GNN-based model to help us extract the required information for estimating our targets. We exploited our model to optimize the target applications by searching through their different design configurations. The experimental results show that GNN-DSE can build a single model with high accuracy to be used among different domains. They also demonstrate that GNN-DSE is able to not only find the Pareto-optimal designs quickly for the applications in its database, but also extend the knowledge it gained from them to optimize new applications from its existing domains. In the future, we will expand our tool to cover more domains.

Acknowledgments

This work is supported by the CAPA award jointly funded by NSF (CCF-1723773) and Intel (36888881), the RTML award funded by NSF (CCF-1937599), the NSF III-1705169 award, Okawa Foundation grant, Amazon research awards, CISCO research grant, Picsart gifts, Snapchat gifts, and CDSC industrial partners (https://cdsc.ucla.edu/partners/). We would also like to thank Marci Baun for editing the paper.

References

[1] J. Cong et al. 2016. Source-to-source optimization for HLS. In *FPGAs for Software Programmers*. 137–163.
[2] J. Cong et al. 2016. Software infrastructure for enabling FPGA-based accelerations in data centers. In *ISLPED*. 154–155.
[3] C. Cummins et al. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. (2021).
[4] J. Duarte et al. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation* 13, 07 (2018), P07027.
[5] D. P. Kingma et al. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[6] T. N. Kipf et al. 2017. Semi-supervised classification with graph convolutional networks. *ICLR* (2017).
[7] J. Kwon et al. 2020. Transfer Learning for Design-Space Exploration with High-Level Synthesis. In *2020 ACM/IEEE MLCAD*.
[8] C. Lattner et al. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on CGO*.
[9] Y. Li et al. 2016. Gated graph sequence neural networks. *ICLR* (2016).
[10] L. v. d. Maaten et al. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, Nov (2008), 2579–2605.
[11] A. Mahapatra et al. 2014. Machine-learning based simulated annealer method for high level synthesis design space exploration. In *ESLsyn*.
[12] A. Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*.
[13] B. Reagen et al. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *IISWC*.
[14] B. C. Schafer et al. 2019. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE TCAD* (2019).
[15] Y. Shi et al. 2021. Masked label prediction: Unified message passing model for semi-supervised classification. *IJCAI* (2021).
[16] A. Sohrabizadeh et al. 2020. End-to-End Optimization of Deep Learning Applications. In *FPGA*. 133–139.
[17] A. Sohrabizadeh et al. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 4 (2022), 1–27.
[18] E. Ustun et al. 2020. Accurate operation delay prediction for FPGA HLS using graph neural networks. In *ICCAD*.
[19] A. Vaswani et al. 2017. Attention is all you need. In *NeurIPS*. 5998–6008.
[20] P. Veličković et al. 2018. Graph attention networks. *ICLR* (2018).
[21] N. Wu et al. 2021. IronMan: GNN-assisted Design Space Exploration in High-Level Synthesis via Reinforcement Learning. *arXiv:2102.08138* (2021).
[22] Z. Wu et al. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.
[23] K. Xu et al. 2018. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*. PMLR, 5453–5462.
[24] C. H. Yu et al. 2018. S2FA: an accelerator automation framework for heterogeneous computing in datacenters. In *DAC*. 1–6.
[25] T. Yuki et al. PolyBench/C. ([n. d.]). https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
[26] J. Zhao et al. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *ICCAD*. 430–437.