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 propose an automated HLS optimization tool that uses multi-graph neural network to estimate the quality of the design in milliseconds with high accuracy, resulting in a speedup of about 79× compared to the previous state-of-the-art tools. We develop a learning model based on Graph Neural Networks (GNN) for predicting the design’s quality [18, 21].

ACM Reference Format:

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. While utilizing a model to explore a reduced set of the solution space. While utilizing a model 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, 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.

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’ \( N(i) \) 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 \( N(i) \) using the degree \( d_i \) of the nodes:

\[
\mathbf{h}_i^t = \sigma(\mathbf{W} \sum_{j \in N(i) \cup \{i\}} \frac{1}{\sqrt{d_jd_i}} \mathbf{h}_j^t)
\]

where \( \mathbf{h}_i \in \mathbb{R}^F \) denotes the input (output) embeddings of node \( i \) which is a vector of \( F \) features. \( \mathbf{W} \) is a trainable weight matrix for the TF step to act as a filter, and \( \sigma \) 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:

\[
\mathbf{h}_i^t = \sigma(\mathbf{W} \sum_{j \in N(i) \cup \{i\}} \alpha_{ij} \mathbf{h}_j^t)
\]

\( \alpha_{ij} \)s are the attention coefficients computed by multi-head dot-product attention. The computation for each head is as follows:

\[
\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T \mathbf{W}_h \mathbf{h}_i \parallel \mathbf{W}_h \mathbf{h}_j))}{\sum_{k \in N(i) \cup \{i\}} \exp(\text{LeakyReLU}(\mathbf{a}^T \mathbf{W}_h \mathbf{h}_i \parallel \mathbf{W}_h \mathbf{k}))}
\]

where \( \parallel \) denotes the concatenation operation and \( \mathbf{a} \) 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 \( P \) be a C program as the FPGA accelerator kernel with design configurations (\( \theta \)). Let \( H \) be a vendor HLS tool that outputs the true execution cycle \( \text{Cycle}(H, P(\theta)) \) and the true resource utilization \( \text{Util}(H, P(\theta)) \):

\[
\begin{align*}
\text{Q}_H(P(\theta)) &= \text{Cycle}(H, P(\theta)), \\
\text{Util}(H, P(\theta)) &= \text{Util}(H, P(\theta))
\end{align*}
\]

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

\[
\min \left( \text{average} \left( \text{Loss}(Q_H(P(\theta)), Q_F(P(\theta))) \right) \right)
\]

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

\[
\min \left( \text{Cycle}(F, P(\theta)) \right)
\]

subject to \( \theta \in \mathbb{R}_F, \quad \forall u \in \text{Util}(F, P(\theta)), u < T_u \)

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...
Evaluator (HLS tool)
2: parallel
1: pipeline
Graph Generator
1: data
0: instruction
(b)
0: control
1: variable
C/C++ Code
Inference
Trainer
for (int i = 0; i < N; i++) { input[i] += 1; }
2
void foo(int input[N]) {
1
Database Generator
(Database Generator)
off|cg|fg
We assign a node for storing the placeholder pragma for each candidate pragma. Each of the candidate pragmas is 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]

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

```c
void foo(int input[N]) {
    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. Fig. 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. 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 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 is defined in either of the following forms:

The `key_text` attribute shows a keyword corresponding to that node. For example, PIPELINE, load, 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.

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 $h_G \in \mathbb{R}^D$ to a graph $G$ 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 $(a_{ij})$ for aggregating the neighbors in a different manner than GAT:

$$a_{ij} = \text{softmax} \left( \frac{(W_1 h_i) (W_2 h_j + W_3 e_{ij})}{\sqrt{D}} \right)$$

where $W_1$, $W_2$, and $W_3$ are learnable weight matrices, and $e_{ij}$ 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).
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 10ad 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:

$$h_i^{(k)} = \text{softmax} (MLP_1(h_i)) \cdot MLP_2(h_i)$$  \hspace{1cm} (9)

where $h_i^{(k)}$ 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 compute the attention for each node and then sum up 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:

$$h_G^{(i)} = \sum_{i=1}^{h} \text{softmax} (MLP_1(h_i)) \cdot MLP_2(h_i)$$ \hspace{1cm} (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, MLP_2$(h_i)$, 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 132 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.](image)

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
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 Mach-Suite 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 CNN 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 range 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:

\[ T_{\text{latency}} = \log_2 \frac{\text{NormalizationFactor}_{\text{latency}}}{\text{latency}} \]

therefore, the model spends more time on reducing the loss for large values of \( T_{\text{latency}} \) 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 \( T_{\text{latency}} / \text{DSP} / \text{BRAM} / \text{LUT} / \text{FF} \) 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 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] 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 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: bipc, doitgen, gesummv, and 2mm. bipc 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.

<table>
<thead>
<tr>
<th>Kernel name</th>
<th>aes</th>
<th>laxx</th>
<th>gemm-blocked</th>
<th>gemm-uncached</th>
<th>mvn</th>
<th>spinv-crs</th>
<th>spinv-cgpack</th>
<th>stencil</th>
<th>nw</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td># pragmas</td>
<td>3</td>
<td>5</td>
<td>23</td>
<td>17</td>
<td>10</td>
<td>1</td>
<td>27</td>
<td>5</td>
<td>5</td>
<td>47</td>
</tr>
<tr>
<td>Initial database</td>
<td>(8 Total / # Valid)</td>
<td>15 / 15</td>
<td>605 / 101</td>
<td>616 / 149</td>
<td>432 / 149</td>
<td>571 / 180</td>
<td>98 / 35</td>
<td>114 / 60</td>
<td>1066 / 281</td>
<td>911 / 66</td>
</tr>
<tr>
<td>Final database</td>
<td>(8 Total / # Valid)</td>
<td>44 / 44</td>
<td>636 / 129</td>
<td>667 / 183</td>
<td>476 / 193</td>
<td>622 / 224</td>
<td>114 / 51</td>
<td>114 / 60</td>
<td>1898 / 291</td>
<td>982 / 103</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Model</th>
<th>Method</th>
<th>Latency</th>
<th>DSP</th>
<th>LUT</th>
<th>FF</th>
<th>BRAM</th>
<th>All</th>
<th>Accuracy</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0</td>
<td>MLP-pragmas (as in [7])</td>
<td>2.2176</td>
<td>0.5807</td>
<td>0.3115</td>
<td>0.2483</td>
<td>0.1596</td>
<td>1.7567</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td>M2</td>
<td>MLP-pragmas-program context</td>
<td>2.9444</td>
<td>0.4650</td>
<td>0.2401</td>
<td>0.1349</td>
<td>0.1597</td>
<td>3.9424</td>
<td>0.78</td>
<td>0.40</td>
</tr>
<tr>
<td>M3</td>
<td>GNN-DSE-GCN</td>
<td>1.6825</td>
<td>0.4265</td>
<td>0.1642</td>
<td>0.1277</td>
<td>0.1593</td>
<td>2.5602</td>
<td>0.79</td>
<td>0.51</td>
</tr>
<tr>
<td>M4</td>
<td>GNN-DSE-GAT</td>
<td>1.1819</td>
<td>0.2557</td>
<td>0.1266</td>
<td>0.1089</td>
<td>0.1178</td>
<td>1.7829</td>
<td>0.85</td>
<td>0.68</td>
</tr>
<tr>
<td>M5</td>
<td>GNN-DSE-TransformerConv</td>
<td>1.1323</td>
<td>0.2540</td>
<td>0.1245</td>
<td>0.0938</td>
<td>0.1231</td>
<td>1.7277</td>
<td>0.85</td>
<td>0.76</td>
</tr>
<tr>
<td>M6</td>
<td>GNN-DSE-TransformerConv + JKN</td>
<td>1.0846</td>
<td>0.2521</td>
<td>0.1112</td>
<td>0.0993</td>
<td>0.0912</td>
<td>1.6324</td>
<td>0.92</td>
<td>0.86</td>
</tr>
<tr>
<td>M7</td>
<td>GNN-DSE-TransformersConv + JKN + node att.</td>
<td>0.8339</td>
<td>0.1253</td>
<td>0.0762</td>
<td>0.0632</td>
<td>0.0615</td>
<td>0.8521</td>
<td>0.93</td>
<td>0.87</td>
</tr>
</tbody>
</table>

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, Piscart gifts, Snapchat gifts, and CDSC industrial partners (https://cdsc.ucla.edu/u/partners/). We would also like to thank Marci Baun for editing the paper.

References