POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Graph Neural Network Model for Fast and Accurate Quality of Result Estimation for High-Level Synthesis

Original
A Graph Neural Network Model for Fast and Accurate Quality of Result Estimation for High-Level Synthesis / Jamal, M.U., Li, Z., Lazarescu, M.T., Lavagno, L.. - In: IEEE ACCESS. - ISSN 2169-3536. - ELETTRONICO. - 11:(2023), pp. 85785-85798. [10.1109/ACCESS.2023.3303840]

Availability:
This version is available at: 11583/2979565 since: 2023-06-26T10:27:45Z

Publisher:
IEEE

Published
DOI:10.1109/ACCESS.2023.3303840

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

17 November 2026

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 12 July 2023, accepted 28 July 2023, date of publication 9 August 2023, date of current version 16 August 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3303840

RESEARCH ARTICLE

A Graph Neural Network Model for Fast and Accurate Quality of Result Estimation for High-Level Synthesis

M. USMAN JAMAL, (Graduate Student Member, IEEE), ZHUOWEI LI, (Graduate Student Member, IEEE), MIHAI T. LAZARESCU, (Senior Member, IEEE), AND LUCIANO LAVAGNO, (Senior Member, IEEE)

Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Turin, Italy

Corresponding author: M. Usman Jamal (muhammad.jamal@polito.it)

This work was supported in part by the Key Digital Technologies Joint Undertaking under the REBECCA Project under Grant 101097224; in part by the European Union, Greece, Germany, Netherlands, Spain, Italy, Sweden, Turkey, Lithuania, and Switzerland; and in part by the Italian ICSC National Research Centre for High-Performance Computing, Big Data and Quantum Computing in the context of the NextGenerationEU Program.

ABSTRACT High-level synthesis (HLS) is a solution for rapid prototyping of application-specific hardware using the C/C++ behavioral programming language. Designers can apply HLS directives to optimize hardware implementations by making trade-offs between cost and performance. However, current HLS tools do not provide reliable quality of results (QoR) estimates, which prevents designers from making these trade-offs efficiently to ensure that the design meets the constraints. Taking advantage of the widespread use of machine learning (ML) to improve the predictability of electronic design automation (EDA) tools, we propose several graph neural network (GNN)-based models that learn and predict the post-implementation QoR from the pre-schedule control data flow graph (CDFG) representation of an HLS design targeting field-programmable gate array (FPGA) implementation, considering also the user HLS optimization directives. Experimental results show that our model can estimate the timing and resource usage of a previously unseen design (i.e, a completely new CDFG) within milliseconds with high accuracy, reducing prediction errors by up to 74 % compared to the estimate generated by the *Vitis HLS* tool itself after going through time-consuming scheduling and binding, and by 29 % and 22 % for resource usage and timing prediction, respectively, compared to the state-of-the-art.

INDEX TERMS Electronic design automation (EDA), graph neural network (GNN), machine learning (ML), high-level synthesis (HLS), quality of results (QoR), field-programmable gate array (FPGA).

I. INTRODUCTION

With the rapid expansion of hardware development and the increase in design complexity, there is a growing need for more efficient and effective design techniques. Traditionally, hardware design has been an expensive and time-consuming process requiring a high level of expertise and specialized knowledge. In recent years, high-level synthesis (HLS) has emerged as an important method for hardware design, allowing high-level programming languages such as C/C++ to be automatically transformed into hardware designs [1]. The introduction of HLS has allowed designers to create hardware using high-level programming languages, allowing both faster design and faster simulation than register-transfer level. By using HLS, designers can benefit from the abstraction and modularity of high-level languages, enabling them to produce more sophisticated, powerful, and portable hardware designs in less time. In addition, designers can use HLS directives

The associate editor coordinating the review of this manuscript and approving it for publication was Larbi Boubchir.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

(also called pragmas) to optimize hardware implementations by considering trade-offs between cost and performance. This flow allows designers to quickly and effectively experiment with different design configurations before working on the final implementation.

HLS was developed to meet the above requirements, but current commercial HLS tools do not provide reliable estimates of the final quality of results (QoR) [2]. As a result, designers are unable to make cost/performance trade-offs and guarantee that the design will meet the requirements because the estimation results in terms of timing and resource usage often significantly differ from the actual QoR after implementation.

Graphs are a widely used model in electronic design automation tools [3]. In addition, electronic design automation tools have recently begun to use machine learning approaches, especially for analysis tasks [4]. Recent research has shown that *graph-based machine learning* techniques can be successfully applied to various phases of the electronic design automation flow, including logic synthesis [5], [6], placement and routing [7], [8], [9], [10], power estimation [11], verification [12], and testing [13].

Graph-based ML approaches have also been used to improve the quality of results estimates in HLS [6], [14], [15], [16]. Although these studies clearly show the advantage of using graph neural networks (GNNs), they either do not include the pragmas in their input representation or still require features extracted from the HLS synthesis report. The former ignores important designer input that affects QoR, while the latter requires the execution of the scheduling, binding, and register-transfer level generation phases, which are more time-consuming than the front-end compilation phase on which our approach relies. In addition, our approach is tool agnostic because we use the low level virtual machine (LLVM) intermediate-representation (IR), which serves as the basis for almost all HLS tools, but the ML models we generate after training are of course both tool and platform specific.

In this work, we aim to predict the post-implementation QoR of an HLS design, starting from both the user C/C++ code and the user-defined optimization directives. We use the output of the *Vitis HLS* open source front-end, without any scheduling or binding information. The design input is represented as LLVM IR from which control flow, data flow, and call flow graphs can be derived using standard compiler techniques. To define our ML problem, we model the program as a single graph that combines the previously mentioned graphs. Furthermore, we use GNNs to learn the underlying heuristics and optimization techniques to predict the desired objectives, namely lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path timing (C.P.). We do not target block RAM, latency and throughput estimation, since the commercial HLS tools predict them relatively accurately.

We propose a GNN-based framework to predict the quality of results of an HLS design based on its HLS IR.

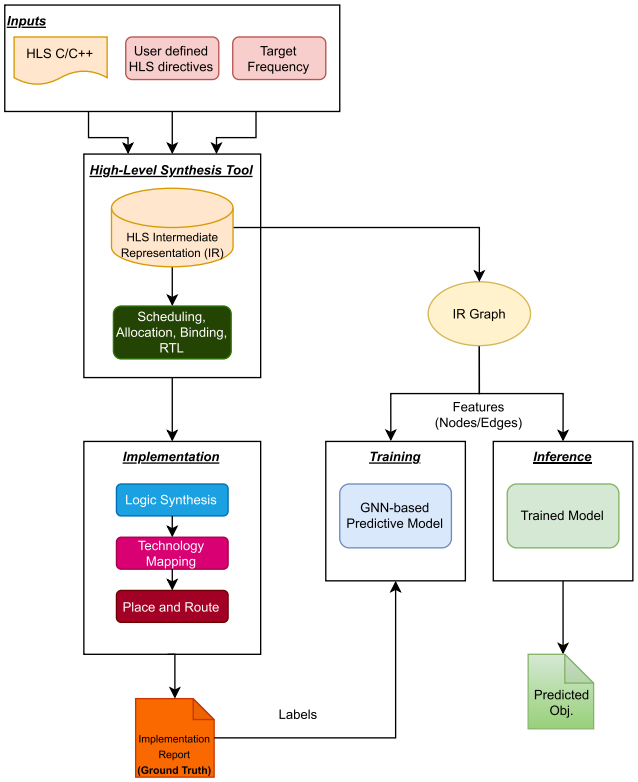


FIGURE 1. Overall framework flow and the relationship between the general HLS-based hardware design workflow (left-hand side) and our proposed framework (right-hand side) for estimating quality of results (QoR) of an HLS-based design.

We formulate the QoR prediction problem as a multi-objective regression task to estimate post-implementation resource usage and timing without invoking the back-end of the HLS tool. The general structure of our prediction model is shown in Fig. 2. The front-end consists of a GNN model that generates the graph-level representation for a given input design. The back-end consists of regression-based multi-layer perceptrons (MLPs) that are fed with the generated graph-level features to predict the estimation targets. Fig. 1 shows our overall framework flow and also shows the relationship between the general HLS-based hardware design workflow and our proposed predictive framework for estimating the QoR of an HLS-based design. Our proposed predictive framework is shown on the right side of the flowchart, showing both training and inference flows. In both phases, the IR-based graph is used as input data. During the training process, in addition to the input data, the corresponding ground truth (i.e., the actual number of resources and clock period) is also required, which is extracted from the post-implementation report. On the other hand, the inference process uses the trained model to predict the QoR of a new HLS design based on its IR.

The experimental results show that our graph-based prediction model not only outperforms the commercial HLS tool for realistic applications from various domains, but also has the ability to generalize to previously unseen design cases by

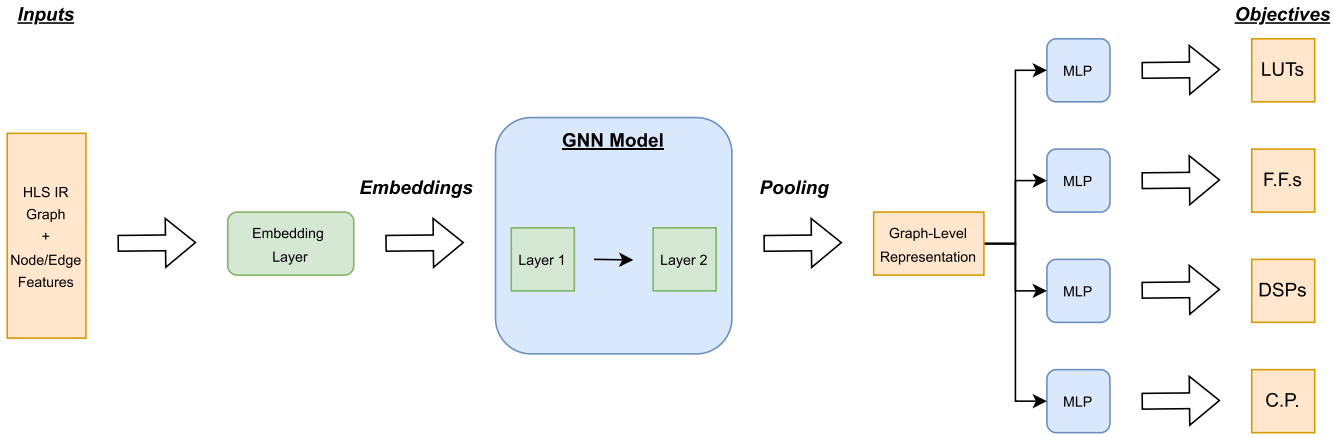


FIGURE 2. General structure of the framework to evaluate different graph neural network (GNN) models. Features are passed to the trainable embedding layer to create their dense vector representations. This vector representation with the corresponding HLS IR based graph is fed as an input to GNN model. A pooling operation is applied across all the nodes to create a single graph-level feature vector which is fed to four separate MLPs for each prediction objective (lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.)).

extending the learned knowledge. We target Xilinx FPGAs in this work, but our framework is based on LLVM IR, which makes it tool independent. Thus, it can be extended to other LLVM-based HLS tools.

Our main contributions are:

- We propose a method that exploits a graph-based representation of an HLS design that includes both program semantics and pragma information (Section IV-B), but not scheduling and binding information. This graph can be easily extracted from the LLVM model generated by the open source front end of *Vitis HLS* and other LLVM-based HLS tools.
- We develop a multi-objective GNN-based learning model aimed at estimating resource usage and timing for a design point using post-implementation training data, which can be evaluated very quickly (in milliseconds, see Section IV-D), even for medium-sized designs.
- The experimental results show that our proposed model not only provides an improved prediction accuracy up to 74 % compared to *Vitis HLS*, but it can also generalize the learned knowledge to estimate completely *unseen* designs (Section V), not just unseen variants of the same design where change only the synthesis directives, which is much easier.

The rest of this article is organized as follows: Section II summarizes existing ML-based HLS prediction techniques. Section III provides background on how an HLS design is represented as a graph and on GNNs. Section IV presents our overall approach to estimate QoR of HLS designs using GNNs. In Section V, we evaluate the proposed approach and provide experimental results, followed by conclusions and future work in Section VI.

II. RELATED WORK

ML techniques have been successfully applied to address various challenges during the chip design flow [17]. These techniques have also been used to resolve the difference between QoR estimates in HLS and post-implementation results. Some of this work is shown in Table 1.

Dai et al. [18] and Makrani et al. [19] propose *non-graph-based* ML models to estimate a design post-implementation resource usage and timing by extracting global features from HLS synthesis reports, thus requiring the most time-consuming HLS steps, namely scheduling and binding. Dai et al. [18] use a linear model (Lasso), an artificial neural network (ANN), and XGBoost to recalibrate the results generated from HLS reports. Makrani et al. [19] use Linear Regression, ANN, Support Vector Machine (SVM), Random Forest (RF), and an ensemble of the four models. However, their methods require the HLS reports as input, and their ability to correctly estimate *unseen* designs is questionable.

On the other hand, our solution input is the HLS LLVM IR after the front-end execution, so we generate the QoR estimates earlier, before the back-end execution. Therefore, the HLS back-end itself could benefit from our estimates (e.g., HLS scheduling could use our critical timing path predictions when balancing pipeline stages).

Another problem with previous work is the limited generalization capabilities, since the inputs (features) to the model can only be extracted after scheduling and binding. This means that for each new and unseen design, one must run time-consuming phases of HLS, which can take hours for larger designs, to collect the features needed to estimate QoR. Our model instead predicts QoR without going through the scheduling and binding steps, while, as we show in Section V, it can estimate new and unseen designs better than the HLS tool itself.

Wu et al. [14] and Ustun et al. [6] use graph-based ML models to perform HLS prediction tasks. Wu et al. [14] proposes an end-to-end reinforcement learning-based framework for design space exploration. A GNN-based performance predictor (GPP) is integrated into the framework to predict post-implementation resource utilization and timing based on the data flow graph (DFG) representation. They use a separate

TABLE 1. Comparison of ML-based approaches for high-level synthesis (HLS) prediction tasks.

Work	ML model		Target		Task	Feature Source	Tool
	Graph	Non-Graph	FPGA	ASIC			
[18]		✓	✓		Resource Usage and Timing	HLS reports	Vivado HLS
[19]		✓	✓		Resource Usage and Timing	HLS reports	Vivado HLS
[14]	✓		✓		DSE	DFG	Vivado HLS
[6]	✓		✓		Operation Delay	Operation Type and Bitwidths from HLS IR code	Vivado HLS
[16]	✓		✓		Resource Usage and Timing	IR operator information (*.adb) and HLS report	Vitis HLS
[15]	✓	✓		✓	Timing	HLS reports	Stratus HLS
This Work	✓		✓		Resource Usage and Timing	HLS LLVM IR	Vitis HLS

GNN-based model for each objective. Ustun et al. [6] builds a customized GNN-based model to automatically learn operation mapping patterns to improve operation delay prediction for HLS-based designs. Their approach improves the estimation accuracy by 72% with respect to *Vitis HLS*. Both works only consider data flow graphs (DFGs). They show the effectiveness of using graph neural networks even though they do not include pragmas in their input representation and focus on predicting only DSP clustering and clock period.

De et al. [15] compare both graph-based and non-graph-based machine learning models to improve delay prediction accuracy for ASIC HLS and propose a hybrid model that considers both local (structural) and global (domain knowledge) features. The global features are extracted from HLS reports, so one has to run HLS during the inference phase, which can take a long time for large designs. We run only the HLS front-end, which is much faster, and requires only global features that are known to the designer before HLS starts, namely design constraints.

Wu et al. [16] propose a graph-based ML approach to estimate resource usage and timing based on the results of different HLS stages. The input graphs are constructed from the IR operator information (`*.adb` file) and the features are extracted from both `*.adb` files and other HLS intermediate results. They formulate the prediction problem as a single-objective task; that is, a separate GNN-based model for each type of resource and for timing. Furthermore, the format of the `*.adb` file is not documented and can be changed at any time. Instead, we take the information from the LLVM bitcode (`*.bc`), which is a standard, well-documented format that is generated right after the HLS front-end compilation process, and we build a *single predictive* model for all the targeted objectives. Also, [16] does not consider the HLS synthesis directives, whereas we do.

Table 1 summarizes the relevant state-of-the-art ML-based approaches for HLS prediction tasks and compares them with our contributions.

III. PRELIMINARIES

A. DESIGN AS GRAPH

Most major compiler tools use graphs for optimization and transformation, and the first step in their compilation process is always to transform the input program into an *IR* graph.

Modern HLS tools are designed and based on state-of-the-art compilers such as LLVM [20] and GCC [21]. The input is a *high-level code* that represents the functionality of the design and can be written in programming languages such as C/C++. The input to the front end of the HLS tool undergoes several IR transformations, and the final output *IR* is geared towards hardware circuit generation, for example by using bit-width-accurate operations such as 7-bit additions. The *HLS IR* consists of *basic blocks*, where each block contains assembler-like instructions with no incoming or outgoing branches except at the beginning and end of the block. The HLS IR can be represented as different types of graphs such as *control flow*, *data flow*, and *call-flow* for specific information extraction.

In the control flow graph, the nodes correspond to the LLVM instructions of the program, and the edges represent how the instructions are sequenced, including conditional branches. The data flow graph contains the same nodes, while the edges represent values flowing between instructions, from results to input operands, using the static single assignment form. Finally, the call graph represents the calling relationship between sub-functions, starting from the top function being synthesized. *We considered a combination of these three graphs for our experiments.*

B. GRAPH NEURAL NETWORKS

Graphs are a type of data structure that helps represent complex information explicitly by establishing relationships between objects. Recently, there has been a significant surge of interest in using deep neural networks, also graph neural networks [22], to analyze graph-structured data. Associating feature vectors (also known as node embeddings) with graph nodes gives GNNs the ability to capture both structural and contextual information. An edge-weighted graph can be represented as $\mathcal{G}(V, E, A)$, where V and E are the set of vertices and edges, respectively, and $A \in \mathbb{R}^{n \times n}$ is an adjacency matrix where A_{ij} is the weight of the directed edge from vertex i to vertex j, or 0 if no such edge exists. An undirected graph can be represented by a symmetric adjacency matrix, and an unweighted graph can be represented by a matrix where all entries are either 0 or 1. Additionally, a graph can be associated with a matrix of *node features* $X \in \mathbb{R}^{n \times d}$, where the feature vector (with d elements) of the i-th node is the

i-th row. Thus, a GNN is a model that learns a trainable nonlinear mapping function F such that $H = F(A, X)$, where $H \in \mathbb{R}^{n \times k}$ is the output feature matrix.

A GNN model iteratively computes a *sequence* of feature matrices from the input node feature matrix through a series of cascading layers. All layers have exactly the same graph structure, but their feature vectors can have different sizes. An aggregate (*AGG*) and an update (*UPDATE*) functions are applied to each node in each layer. The *AGG* function receives the feature vector information from the neighboring nodes of the ith node in the tth layer and sends the aggregated information to the *UPDATE* function to update the ith node features in the $(t + 1)$th layer based on the aggregated value and possibly the ith node feature value in the tth layer. Note that this may change the size of the feature vector, while the graph remains the same across all layers. In addition, some classes of GNNs, such as dynamic graph attention network (GAT), also use edge features in the aggregation process. A *READOUT* function, such as mean or max pooling, is applied after the last layer to summarize the features from all nodes and produce a *single graph-level feature vector*, which is typically used as input to the MLP that produces the final GNN output(s).

For example, a convolutional neural network where all layers have exactly the same "image" size (along the x and y directions) can be seen as a special case of GNN, where (1) each node in the graph (except the "boundary" nodes, which have smaller neighborhoods) has a set of neighbors (i.e., nodes connected to it by edges) of the same size and shape as the convolution filter, (2) the number of channels in each layer is the number of elements in the node feature vector, (3) the *AGG* function computes a convolution (and pooling) operation over the neighbor's feature vectors, (4) the *UPDATE* function is a ReLU, and (5) the *READOUT* function concatenates the features of the last layer.

The message passing-based architecture [23] of a generic GNN model can be summarized as follows

$$m_i^{t+1} = \text{AGG}\left(h_u^t \mid u \in \mathcal{N}(i) \cup \{i\}\right) \tag{1}$$

$$h_i^{t+1} = \text{UPDATE}\left(m_i^{t+1}\right) \tag{2}$$

$$h_{\mathcal{G}} = \text{READOUT}\left(h_i^T, i \in V\right) \tag{3}$$

where the feature vector for node i at layer (also called iteration) t is denoted by h_i^t, $N(i)$ represents the neighborhood of node i, i.e. the set of nodes with an edge to i, where m_i^{t+1} represents the aggregated message from neighbors, V is the node set of the graph $\mathcal{G}$, T is the number of layers, i.e. message-passing iterations, of the GNN, and $h_{\mathcal{G}}$ is the final graph-level feature vector sent to the output MLP.

GNNs can be divided into two groups based on the learning method: *transductive* and *inductive*. Transductive-based GNNs need to see the whole graph structure to learn each node feature vector during training. If there is a change in the structure of the graph, a model has to be retrained. Therefore, they are not able to generalize to unseen graphs. On the other hand, the inductive-based GNN learns a trainable function that aggregates the feature vectors from a node neighborhood to generate feature vectors for the nodes in the graph. Because this trainable function is shared across the graph, like the filter of a CNN layer, the learned model can be applied to unseen graphs without re-training, making it generalizable. *In this work, we have performed the training via inductive learning for the GNN models*.

In general, different GNN models differ from each other based on different aggregate and update functions [24]. Any permutation-invariant operation can serve as an *AGG* function, and any differentiable function can be used as an *UPDATE* function. *In our experimentation, we have used 4 different GNNs, namely: graph convolutional network [25], dynamic graph attention network [26], graph isomorphism network [27], and deep adaptive graph neural network [28]. In this paper, we refer to these models as GCN, GAT, GIN and DAGNN respectively.*

IV. METHODOLOGY

In this paper, we present our *Graph-based Machine Learning* approach to estimating the quality of results (QoR) of *HLS*-based designs. The general framework is shown in Fig. 1, which first generates a graph based on HLS IR and then predicts the post-implementation QoR. The main steps are as follows:

- Section IV-A describes how the dataset is built by using different designs.
- Section IV-B presents how a graph is constructed and generated for an HLS design.
- Section IV-C defines the *local* and *global* feature vectors used by the predictive model.
- Section IV-D shows the proposed model and its *training* and *inference* phases.

A. DATASET GENERATION

The fundamental step in any ML problem is to obtain the data on which the ML model can be trained, validated, and tested. The dataset should include a variety of designs from different applications so that the trained ML model is robust enough to generalize. For this purpose, we choose 30 designs from the well-known HLS benchmark suites, namely MachSuite [29], Polyhedral [30], and Rosetta [31]. The application domains of these designs cover a wide range of areas such as linear algebra, image and signal processing, computer graphics, data mining, stencils, sorting, and ML. To create multiple hardware implementations for each design, we used different *HLS pragmas*, i.e., synthesis directives (see Table 2) and various clock periods (2.5 ns, 5 ns, 7.5 ns, and 10 ns). This allows our predictive model to learn designs with different area-delay tradeoffs. Thus, we build a data set with a total of 2465 data points. We synthesize each design point in our dataset with *Vitis HLS* 2021.2 [32] and implement it with *Vivado* 2021.2 [33] (using *Vivado* defaults for synthesis and implementation) targeting *Zynq UltraScale+* field-programmable gate array

TABLE 2. *Synthesis* pragma configurations.

Pragma	Configuration
Loop Pipelining	Enabled/Disabled
Loop Unrolling	Unrolling Factor
Loop Flattening	Yes/No
Array Partitioning	Block/Reshape/Cyclic/Complete
Function Inline	Yes/No

TABLE 3. Overall *Summary* of designs in our DATASET.

	# of **LUTs**	# of **DSPs**	# of **FFs**	**CP** (ns)
Minimum	8	0	24	1.5
Maximum	53 239	360	31 004	8.562
Average	2456	28	2619	3.72

devices. The *ground truth* (actual resource usage and critical path timing) of each of these design points is extracted from the implementation and timing reports generated after the place and route phase for maximum prediction accuracy. Table 3 shows the range of values of the target objectives in our dataset. The usage percentage for LUTs, DSPs and FFs with respect to available resources on field-programmable gate array chip used in our dataset is 75 %, 100 % and 22 % respectively.

B. INPUT

The input to our proposed GNN-based predictive model is an HLS IR graph representing the functionality of the design, extracted after the front-end compilation step. As mentioned in Section III-A, we used a combination of control flow, data flow, and call flow graphs for our experiments. A program semantic information representation tool, ProGraML [34] is used to extract the graphs from a given IR and combine them into a single graph. It merges information from control and data flow graphs and also preserves the function hierarchy by incorporating the call flow.

The out-of-the-box configuration of ProGraML converts the LLVM statements into nodes of the generated graph with some special features like the opcode. By using the LLVM language reference manual [35], we have extended ProGraML capabilities to retrieve more critical information from the LLVM statements, namely operand bit width, and opcode category, and append it to the feature vector of each corresponding node in the graph, as discussed below.

Bit-width describes the number of bits in the instruction operands, which is highly relevant to the hardware resource prediction. For example, an instruction that operates on 7-bit operands requires fewer hardware resources than an instruction that operates on 32-bit operands.

The Opcode category identifies the functionality of an LLVM instruction. In general, the resource requirements and behavior of different LLVM instruction categories vary, affecting overall performance and resource utilization. For example, the Arithmetic category contains statements that perform arithmetic operations, such as adding or multiplying. In principle, this information could be learned by a GNN from the opcode, but this would require more layers and more training data. We decided to provide it directly because it is design-independent and easy to derive automatically.

Algorithm 1 shows the steps to construct the graph for an HLS design.

Algorithm 1 Graph Generation for HLS Design

Require: HLS design
LLVM IR Bitcode ← Vitis HLS Front-end (HLS design)
LLVM IR ← LLVM Disassembler (LLVM IR Bitcode)
Graph Representation ← Modified ProGraML (LLVM IR)

Fig. 3 shows a toy example of how the input graph to our model is generated. For illustration purposes, only the most relevant nodes in the graph are shown. For example, we have omitted zero extension and alloca nodes in the figure, while our model considers them as well. Fig. 3 (a) shows the HLS code for implementing the dot product of two input vectors, including two sample HLS pragmas inside the loop. Fig. 3 (b) shows its graph representation, extracted after the HLS frontend compilation. The graph has two types of nodes. The LLVM statements are represented by the blue nodes, which are connected according to the control flow. The variable values and constant values that represent the operands and results of the statements in the data flow are represented by the nodes in red. Three different colors are used to symbolize different types of edges: blue, red, and green for control, data, and call, respectively. Fig. 3 (c) shows the local and global features that can be extracted directly from the IR graph and user-defined optimization directives (see Section IV-C for details).

C. FEATURES

Our proposed approach uses two different sets of features that are useful for predicting post-implementation QoR. These feature sets are extracted from two different sources, the HLS IR code and user-defined HLS synthesis directives. We call these sets local and global features, respectively. Local features contain structural and contextual information. Structural information describes the connectivity of nodes in the graph and is encoded as an adjacency matrix. Contextual information refers to node and edge properties, and this information is explicitly encoded as a feature vector for each node and edge.

For each node, its *type*, *category*, *opcode*, *block ID*, *function ID*, and *bitwidth* are taken into account. For example, *type* indicates whether the node is a statement, a variable,

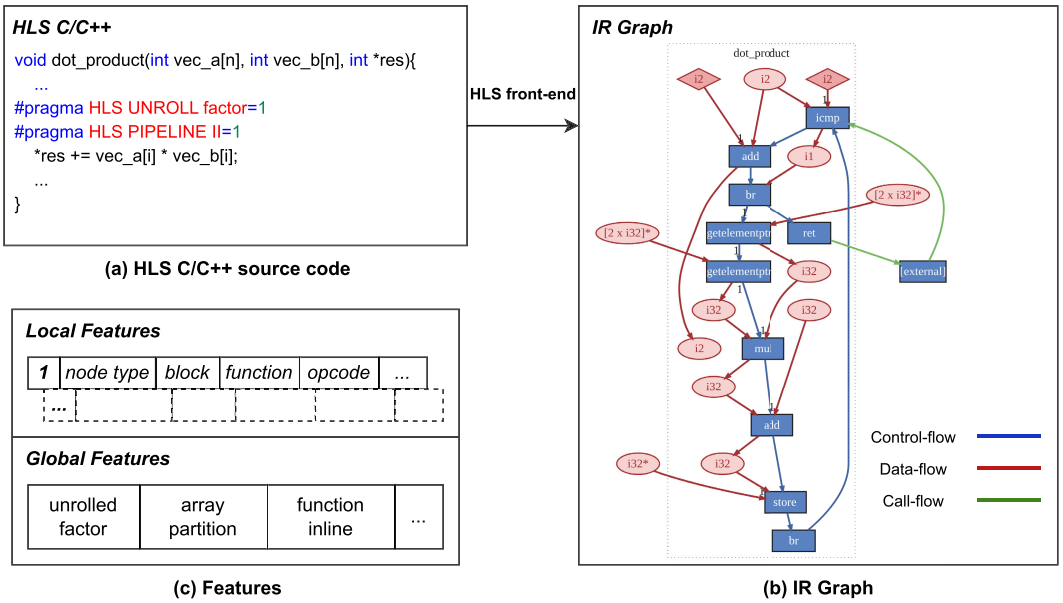


FIGURE 3. An High-level synthesis (HLS) design example with its graph representation (a) shows the HLS code for an implementation of dot product with two sample synthesis directives provided as an input to the HLS tool (b) shows its graph representation based on an intermediate-representation (IR) which is extracted after the HLS front-end compilation (c) shows the local and global features used by the model. Local features are extracted directly from the IR graph. Global features are user-defined synthesis directives.

TABLE 4. *Local features*: Nodes and edges.

	Feature	Description	Example Values
Node	Type	Node Type	Instruction, Variable, Constant
	Block	LLVM Block ID	0, 1, 2 etc.
	Function	Function ID	0, 1, 2 etc.
	Opcode Category	Opcode type based on LLVM	Unary, Binary, Terminator, etc.
	Opcode	Opcode of the node	add, icmp, shl etc.
	Bitwidth	Bitwidth of the operand	8, 16, 32, etc.
Edge	Type	Flow Type	Control, Data, Call

or a constant. For an edge, we only considered its *type*, which basically tells whether the edge belongs to control, data, or call flow. Details of local features are listed in Table 4. Edge features are only considered by one of the models we used, namely GAT.

Table 2 displays the list of global features. We feed the global feature vector into our prediction model by concatenating it with the final graph-level feature vector generated by *READOUT*. These features can be useful for integrating domain-specific knowledge into the model and improving its predictive capabilities. They have a direct impact on the timing and resource requirements of a design. For this reason, we explicitly provide this information to our model. These features are exactly the same as the main user-defined HLS synthesis directives, so they are well-known to the designer.

These global features could also be automatically extracted from the HLS IR code via *ssdm intrinsic functions* (function calls that encode both user-written and tool-generated synthesis directives) and encoded as local features for the nodes. However, as we show in Section V-B, the estimation accuracy is significantly improved by using global features as well. We leave to future work the exploration of different sets of local features and/or GNN architectures to overcome the need for global features and handle more complex kernels, including, for example, multiple pipelined loops.

D. MODEL

We formulate the QoR prediction problem as a multi-objective regression task to estimate the post-implementation *timing* and *resource usage* for LUT, FF, and DSP of a given design based on its HLS IR without scheduling and binding. We do not address the block RAM estimation problem because the HLS estimates are already quite reliable in this

case, although we could. We use multi-task learning, where a single GNN model is trained and the generated graph feature vector is fed to a set of MLPs to estimate the different objectives.

The general structure of our model architecture is shown in Fig. 2. It takes the graph representation of the design as input and creates the initial feature vector by converting the features extracted from LLVM (Table 4) via a trainable embedding layer [36]. This information is then passed to the GNN model, which iteratively updates the feature vectors layer by layer. These updated feature vectors are used to generate a graph-level representation vector via *mean pooling*, i.e. by computing an average vector of all final feature vectors of all nodes, which is then concatenated with the global feature vector and passed to a set of MLPs to predict multiple targets. We evaluated different GNN models (Section III-B), using the same flow for a fair comparison (so the only difference is the type of GNN layers).

Fig. 4 shows the *training flow* of the proposed framework. Before training, we preprocess the data and apply normalization so that each objective (timing, LUTs, DSPs, and FFs) can contribute equally to the training loss. For example, in the case of resource utilization, we normalize the resource utilization by dividing it by the total number of resources available on the field-programmable gate arrays, converting it to a percentage utilization. Once the dataset and associated features are available, we perform the training to obtain a predictive model. The ground truth labels are extracted from the post-implementation reports.

We train the GNNs model via supervised learning to learn the behavior of the underlying HLS heuristics and optimization techniques, such as scheduling, sharing, register allocation, etc., in order to quickly and accurately predict the desired objectives. During the training phase, an HLS design and its configurations are first fed into the HLS tool front-end, where the HLS IR is generated. The graph generator (see Section IV-B for details) then converts the HLS IR into the graph used by the GNN model as discussed above.

To select the best GNN model, we first randomly set aside a 20 % of the data set, also called the hold-out or test set. This hold-out is not used during training and validation, but only at the end to evaluate the final performance and report the results. Then, we perform training by 5-fold cross-validation on the remaining 80 % dataset, where the hyperparameters of the considered models are optimized and tuned.

The *inference flow* of the proposed framework is shown in Fig. 5. The main purpose of the inference phase is to achieve fast and accurate QoR prediction of the design compared to the HLS baseline without going through the implementation process, which is time consuming. In the inference phase, we apply the same preprocessing workflow to unseen designs (test set data points) to generate a graph and extract feature vectors. These are then fed into the already trained model to perform target prediction.

It is worth noting that our predictive model is able to complete the inference to estimate resources and timing in milliseconds given a graph, as opposed to the implementation phase, which typically takes minutes to hours. Table 5 shows the inference time of the predictive model and the time HLS tool takes for a single data point. Our model provides better estimates of resource usage and timing in less than half of a time with respect to the HLS tool. If we exclude the common time needed to execute the Front-End, our model is more than 60x faster than the Back-End.

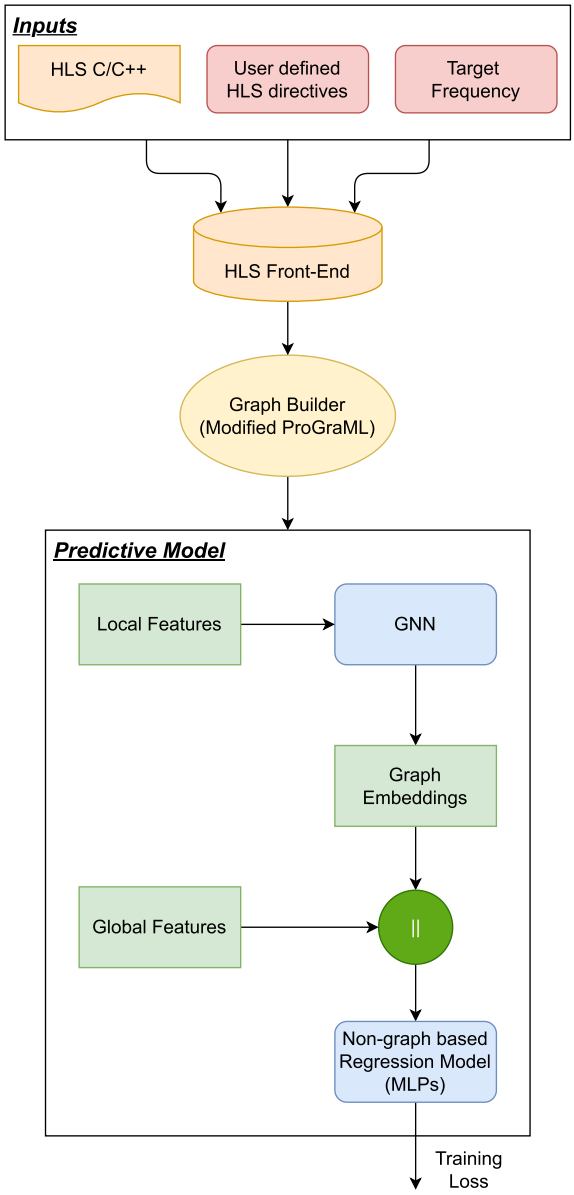


FIGURE 4. Training phase of the proposed framework.

V. EXPERIMENTAL RESULTS

A. SETUP

Our framework is deployed using PyTorch and all GNN models mentioned in Section III-B are implemented and trained using PyTorch Geometric [37] library on a Linux machine

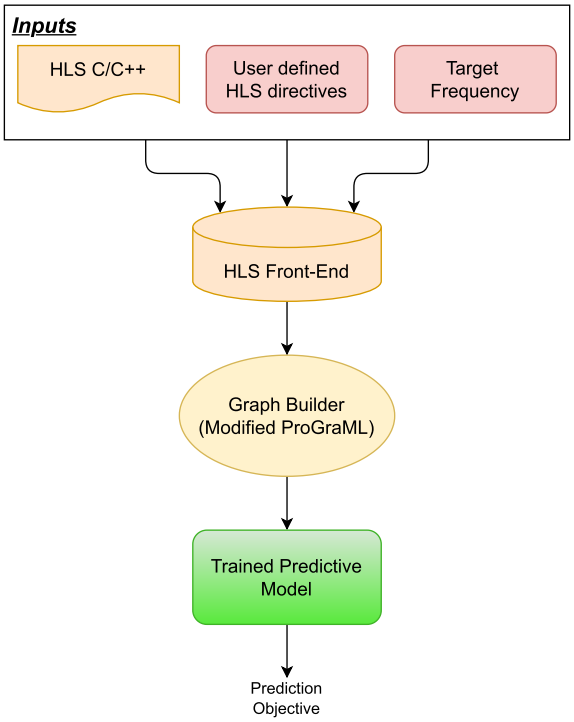


FIGURE 5. Inference phase of the proposed framework.

TABLE 5. Inference time for the proposed model vs HLS time per design point.

	HLS F.E. (s)	HLS B.E. (s)	Graph Generation (s)	Inference (s)	Total (s)
Model [1]	11.65	—	0.24	0.0054	11.90
HLS [2]	11.65	15.07	—	—	26.72

[1] Model Time = Front-End (F.E.) + Graph Generation + Inference
[2] HLS = Front-End (F.E.) + Back-End (B.E.)

equipped with an Nvidia GeForce RTX 3060 graphical processing unit. The designs in the dataset (Section IV-A) are synthesized and implemented using *Vitis HLS* 2021.2 [32] and *Vivado* 2021.2 [33], respectively. The ground truth labels of the four objectives (LUT, FF, DSP, C.P.) are extracted from the post-implementation reports. The dataset is randomly divided into 70 % for training, 10 % for validation, and 20 % for testing.

For our experiments, each model has the structure shown in Fig. 2, with the following characteristics:

1) A trainable embedding layer, that converts the features (see Table 4) from the HLS IR into a feature vector of size 300 that is fed as an input to the GNN layers.
2) Two GNN layers, i.e., the *AGG* and *UPDATE* functions are run twice, with input feature vector size 300, internal feature vector size of 128 and output feature vector size of 64.
3) A mean pooling layer, taking the mean of each one of the 64 output features across all nodes, and generating a single graph-level vector of 64 features that is fed to the MLPs.
4) Four separate MLPs, one for each prediction objective (LUT, FF, DSP, C.P.), each with three layers with 32, 16 and 1 output features respectively.

We trained the GNN-based models in an *inductive setting* [38] with the Adam optimizer [39] for 100 epochs, using a learning rate of 0.001, a weight decay of 0.0005, and an exponential linear unit [40] as the activation function. All of these hyperparameters, including the number of layers and feature vector sizes, are tuned using the validation set during the training phase. Since the prediction problem is formulated as a regression task, we use *root mean square error* as a metric for evaluating the models. We perform 5-fold cross-validation to check the effectiveness and robustness of the models and to select the best-performing model. We also compare our models with the commercial HLS tool (*Vitis HLS*) used as a *baseline model* and with a graph learning-based performance prediction model [16]. Algorithm 2 shows the process of extracting the baseline and ground truth values for a given HLS design.

Algorithm 2 Baseline and Ground Truth (G.T.) Extractor

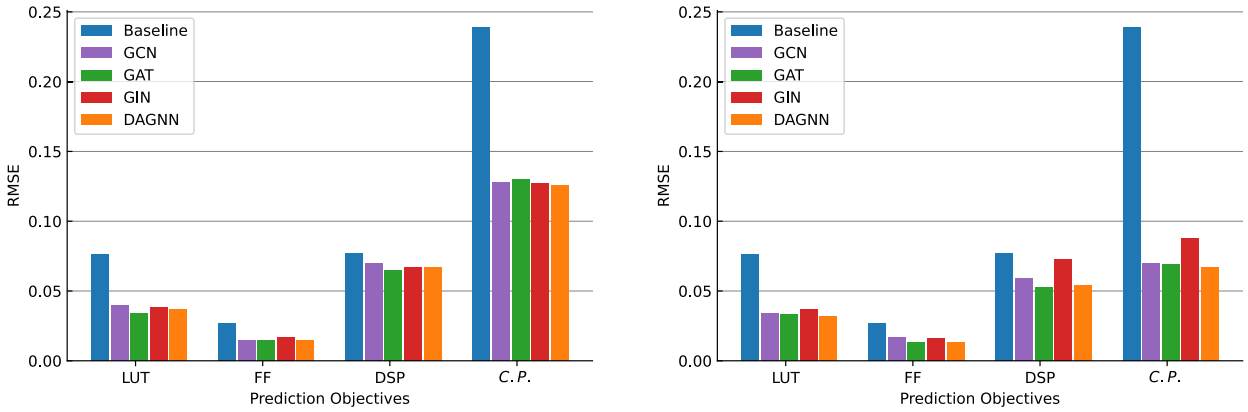
Require: An HLS design
RTL design, HLS report ← Vitis HLS (HLS design)
Post-implementation report ← Vivado (RTL design)
Baseline ← HLS baseline Extractor(HLS report)
G.T. ← G. T. Extractor(Post-implementation report)

B. MODEL EVALUATION AND MODEL SELECTION

A GNN model transforms the local feature vectors into a single graph-level feature vector, which is then passed to the non-graphic regression model (in this case using a 3-layer MLPs). We first test the performance of the GNN models with *only local features* (i.e., no manual information from the designer).

Fig. 6a shows the performance evaluation of GNN models with respect to *baseline* regarding LUT, FF, DSP and C.P. in terms of root mean square error, while Fig. 7a shows the prediction improvements of the models over the baseline. For LUT utilization prediction, GAT provides the best improvement, with more than 55 % over baseline. In the case of FF, GCN, GAT, and DAGNN give prediction improvements of more than 40 %. GAT is the best of all models at reducing prediction error relative to the HLS baseline model for DSP utilization. For C.P. timing, all models improve the prediction by more than 45 %. On average, GAT is the best model for improving resource utilization prediction, and DAGNN is the best for timing. Note that in practice, different models may be used for different objectives.

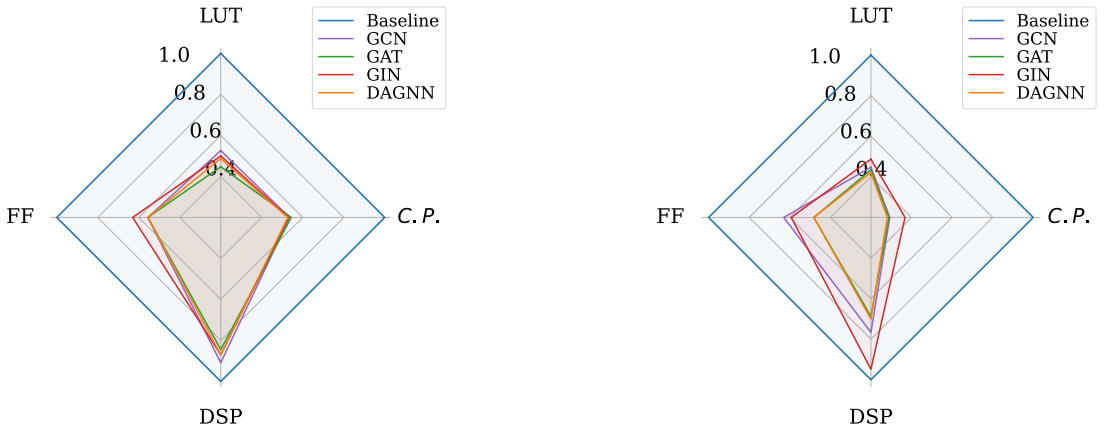
Fig. 6b shows the performance evaluation of the graph neural network (GNN) model with respect to the HLS baseline for the target objectives by *also using global features*.



(a) Only Local Features are considered.

(b) Both Local and Global Features are considered.

FIGURE 6. Performance comparison of different GNN-based predictive models for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with HLS *Baseline* on the *test set* (the lower the better).



(a) Only Local Features are considered.

(b) Both Local and Global Features are considered.

FIGURE 7. Quality of results (QoR) prediction improvements of different GNN-based predictive models for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with respect to HLS *Baseline* on the *test set*.

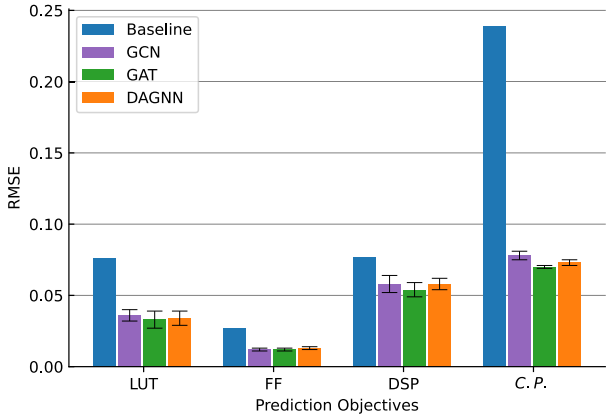
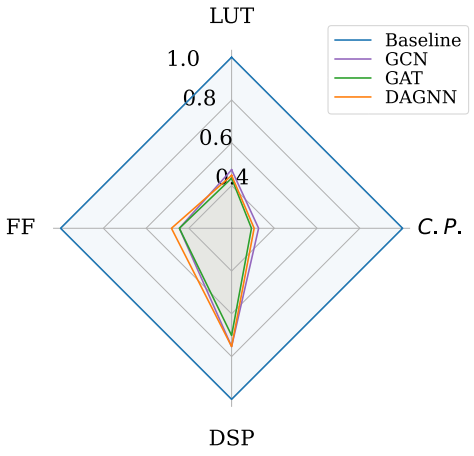


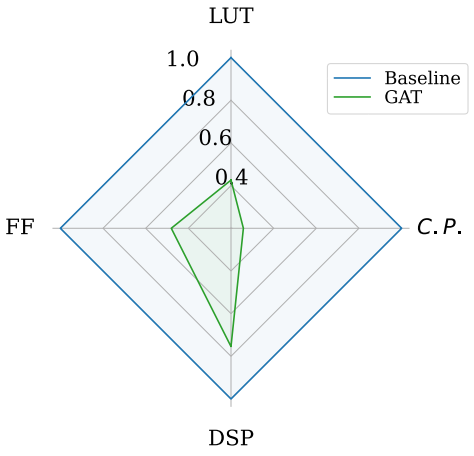
FIGURE 8. Performance comparison of best performing GNN-based predictive models for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with HLS *Baseline* (the lower the better). A *5-fold cross-validation* is performed. Both Local and Global features are considered.

Global features are concatenated with the graph features generated by the GNN models and fed into the non-graph regression model, MLP. Fig. 7b shows the quality of results (QoR) improvements of the GNN-based predictive models over the baseline. All models provide performance prediction improvements of more than 50 % for LUT utilization. For FF utilization, GAT and DAGNN are the best models at reducing the prediction error over the baseline, improving the QoR prediction by up to 52 %. In the case of DSP utilization, GAT provides the best prediction improvement among all GNN models with respect to the HLS baseline. The GAT model with both local and global features gives a relative improvement of almost 19 % over the GAT model with local features only.

For C.P. timing, GCN, GAT, and DAGNN based models improve the prediction by more than 70 % with respect to the baseline, with DAGNN providing the best improvement at 72 %.



(a) A *5-fold cross-validation* with the *holdout set* is performed for the best performing GNN-based predictive models.



(b) A *5-fold cross-validation* over the *whole dataset* is performed for the selected GAT-based predictive model.

FIGURE 9. Quality of results (QoR) prediction improvements of different GNN-based predictive models for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with respect to HLS *Baseline*. Both Local and Global features are used.

These GNN models provide a relative performance improvement of up to 47 % over the same models using only local features.

It is worth noting that all of these GNN-based prediction models perform better than the HLS baseline model for all target objectives. The comparable accuracies of the models provide a testament to the robustness of our proposed methodology, where the training method of the model and its architecture matter relatively little with respect to the improvements over the state of the art. Graph isomorphism network (GIN) provides the least benefit in terms of prediction error reduction among all the models tested, especially in the case of DSP usage prediction (see Fig. 7b). Based on this empirical evidence, we do not use the graph isomorphism network for the model selection phase.

For *model selection*, we use 5-fold cross-validation with a holdout test set (see Section IV-D for details). Fig. 8 compares the performance of the GCN, GAT and DAGNN models with respect to the HLS *baseline*. Error bars in the figure show the standard deviation of the models.

Fig. 9a shows the QoR improvements of the selected models over the baseline for the target objectives. For the LUT and FF utilization predictions, all selected graph-based models give improvements of more than 50 %. The GAT based model gives the best result of 57 % for LUT, while the GCN based model gives the best result of 55 % for FF. In the case of DSP prediction, GAT is the clear winner, reducing the prediction error by more than 30 %. For C.P. timing, all graph-based models provide a prediction improvement of more than 67 %, with the GAT-based prediction model being the best (71 %). GAT outperforms other graph-based models in three out of four target objectives and is not far behind in predicting FF usage (where the GCN-based model performs best). *Based on these results, we choose the GAT-based model*, but we could use different models for different objectives, as mentioned above.

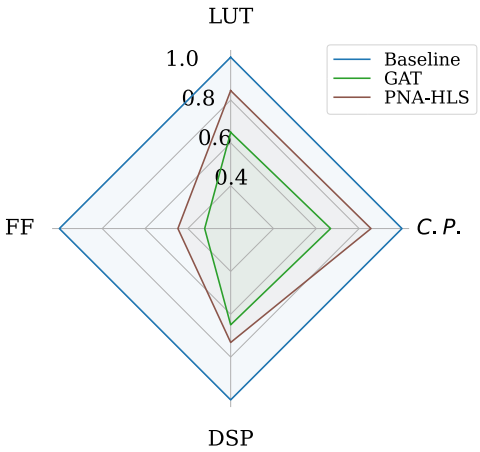


FIGURE 10. Quality of results (QoR) prediction improvements of our best performing GNN-based model for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with respect to HLS *Baseline* and the state-of-the-art on *Unseen Kernels*.

To evaluate the selected model over the entire dataset, we perform a generic 5-fold cross-validation. Fig. 9b shows the quality of results prediction improvements over the entire data. It is observed that the GAT based prediction model provides significantly better prediction with respect to the HLS baseline model, giving up to 74 % performance prediction improvements.

C. UNSEEN KERNELS AND COMPARISON WITH STATE-OF-THE-ART

To check the performance of our chosen model on unseen kernels and for a quantitative comparison with the state-of-the-art, we choose four kernels (`gemm_ncubed`, `optical_flow`, `jacobi2d`, and `stencil2d`),

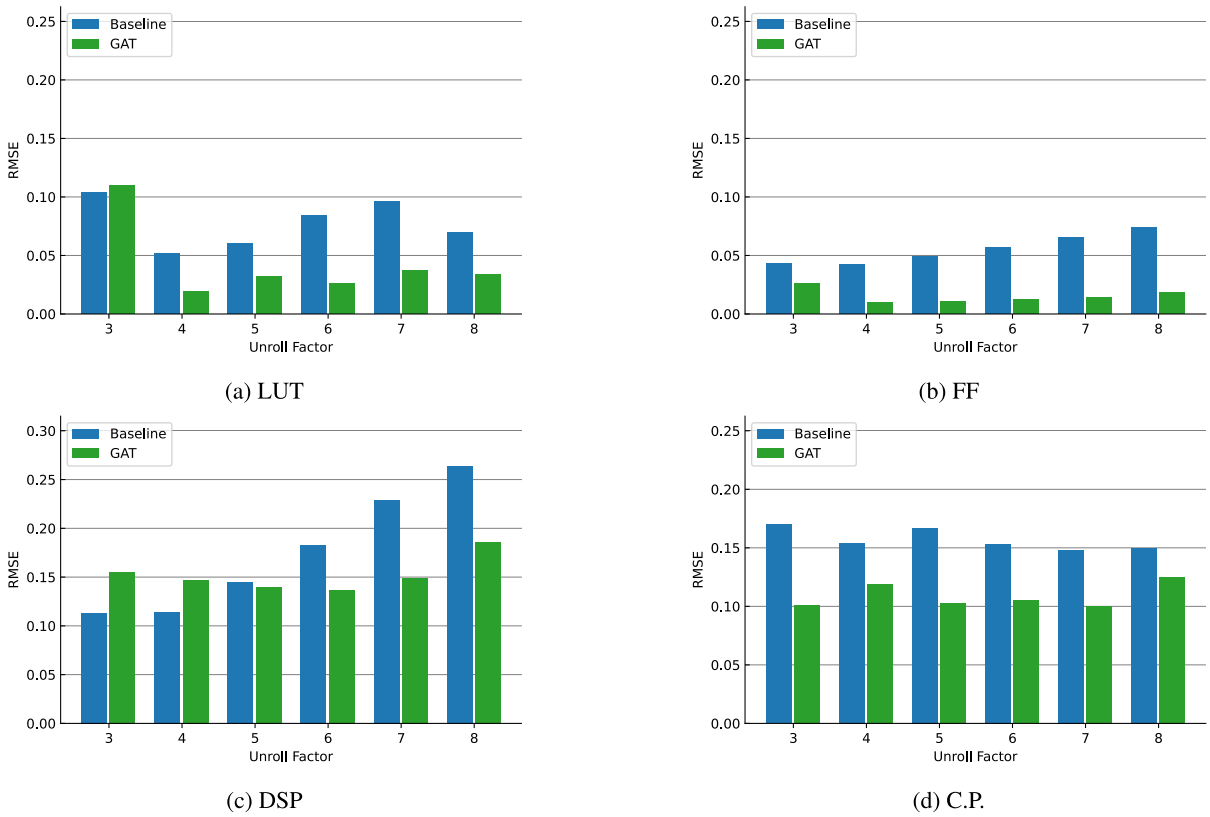


FIGURE 11. Performance comparison of the GAT-based predictive model for lookup table (LUT), flip-flop (FF), digital signal processing unit (DSP) and critical path (C.P.) with respect to HLS *Baseline* for different *loop unrolling factors* (the lower the better).

including all their design variants created using different HLS synthesis directives and clock frequencies, and use them as a test set for evaluation (in 5-fold cross-validation, the test set was chosen at random, so training saw many variants of each design). `gemm_ncubed` is a dense matrix multiplication algorithm, `optical_flow` computes the motion of each pixel in a sequence of image frames, `jacobi2d` is an iterative method that computes and updates each grid point by averaging its neighbors, and `stencil2d` performs stencil computation using a 9-point square stencil. Note that all of these kernels have different code structures.

We compare our model quantitatively with the graph-based machine learning approach [16], which provides their tool in open source. For comparison with other ML-based approaches, Table 1 provides a qualitative analysis.

We take the best-performing regression model in [16] for resource usage and timing prediction, train it to the best of our ability, and call it *PNA-HLS*. A separate PNA-HLS model is trained for each objective. Fig. 10 shows that our GAT-based prediction model reduces the prediction error for resource utilization and timing prediction by 68 % and 34 %, respectively, compared to HLS. Our proposed model also outperforms the state-of-the-art PNA-HLS model by 29 % and 22 % for resource utilization and timing prediction, respectively.

Fig. 11a, Fig. 11b, Fig. 11c and Fig. 11d show the performance of GAT-based prediction model for different loop unrolling factors and compares with the *Baseline* for LUT, FF, DSP and C.P. respectively. These figures further demonstrate that overall our proposed model provides better quality of results prediction compared to HLS.

VI. CONCLUSION

Although high-level synthesis (HLS) provides great flexibility for optimizing designs for area and performance, HLS-estimated quality of results (QoR) often differ from actual post-implementation results. We propose an HLS tool-agnostic graph neural network (GNN)-based framework for estimating quality of results of HLS designs, as long as the tool provides a publicly readable low level virtual machine-based intermediate-representation (IR). Of course, the GNN must be trained differently for each new tool, but once trained, it can be reused for different results.

First, a method is developed to extract a graph-based representation of a design directly from the HLS front-end output, encoding both program semantics and HLS synthesis directive information. Then, a multi-objective GNN-based learning model is proposed to predict resource usage and timing of HLS designs within milliseconds without invoking the HLS back-end to perform scheduling and binding. To address the issue of the limited ability of a pure GNN-based model

to be aware of global information such as design guidelines, this information is explicitly passed to the learning model in addition to the local features automatically extracted from the IR.

The experimental results show that our proposed prediction model outperforms both a commercial HLS tool and a state-of-the-art GNN-based tool [16] for realistic benchmark applications from different domains. It also shows that our model is capable of extending the learned knowledge and generalizing it to unseen design cases. We plan to extend our framework to larger designs and more application domains in future work.

REFERENCES

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011, doi: 10.1109/TCAD.2011.2110592.

[2] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS today: Successes, challenges, and opportunities," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, pp. 1–42, Dec. 2022, doi: 10.1145/3530775.

[3] Y. Ma, Z. He, W. Li, L. Zhang, and B. Yu, "Understanding graphs in EDA: From shallow to deep learning," in *Proc. Int. Symp. Phys. Design*, Mar. 2020, pp. 119–126, doi: 10.1145/3372780.3378173.

[4] D. S. Lopera, L. Servadei, G. N. Kiprit, S. Hazra, R. Wille, and W. Ecker, "A survey of graph neural networks for electronic design automation," in *Proc. ACM/IEEE 3rd Workshop Mach. Learn. CAD (MLCAD)*, Aug. 2021, pp. 1–6, doi: 10.1109/MLCAD52597.2021.9531070.

[5] W. Haaswijk, E. Collins, B. Seguin, M. Soeken, F. Kaplan, S. Süsstrunk, and G. De Micheli, "Deep learning for logic optimization algorithms," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–4, doi: 10.1109/ISCAS.2018.8351885.

[6] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang, "Accurate operation delay prediction for FPGA HLS using graph neural networks," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2020, pp. 1–9, doi: 10.1145/3400302.3415657.

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

[8] Y.-C. Lu, S. Pentapati, and S. K. Lim, "The law of attraction: Affinity-aware placement optimization using graph neural networks," in *Proc. Int. Symp. Phys. Design*, Mar. 2021, pp. 7–14, doi: 10.1145/3439706.3447045.

[9] Z. Guo, M. Liu, J. Gu, S. Zhang, D. Z. Pan, and Y. Lin, "A timing engine inspired graph neural network model for pre-routing slack prediction," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, Jul. 2022, pp. 1207–1212, doi: 10.1145/3489517.3530597.

[10] R. Kirby, S. Godil, R. Roy, and B. Catanzaro, "CongestionNet: Routing congestion prediction using deep graph neural networks," in *Proc. IFIP/IEEE 27th Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Oct. 2019, pp. 217–222, doi: 10.1109/VLSI-SoC.2019.8920342.

[11] Z. Lin, Z. Yuan, J. Zhao, W. Zhang, H. Wang, and Y. Tian, "PowerGear: Early-stage power estimation in FPGA HLS via heterogeneous edge-centric GNNs," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2022, pp. 1341–1346, doi: 10.23919/DATE54114.2022.9774682.

[12] Y. Zhang, H. Ren, and B. Khailany, "GRANNITE: Graph neural network inference for transferable power estimation," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6, doi: 10.1109/DAC18072.2020.9218643.

[13] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu, "High performance graph convolutional networks with applications in testability analysis," in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 1–6, doi: 10.1145/3316781.3317838.

[14] N. Wu, Y. Xie, and C. Hao, "Ironman: GNN-assisted design space exploration in high-level synthesis via reinforcement learning," in *Proc. Great Lakes Symp. VLSI (GVLSI)*, 2021, pp. 39–44, doi: 10.1145/3453688.3461495.

[15] S. De, M. Shafique, and H. Corporaal, "Delay prediction for ASIC HLS: Comparing graph-based and nongraph-based learning models," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 4, pp. 1133–1146, Apr. 2023, doi: 10.1109/TCAD.2022.3197977.

[16] N. Wu, H. Yang, Y. Xie, P. Li, and C. Hao, "High-level synthesis performance prediction using GNNs: Benchmarking, modeling, and advancing," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, Jul. 2022, pp. 49–54, doi: 10.1145/3489517.3530408.

[17] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen, J. Wu, Y. Xu, H. Zhang, K. Zhong, X. Ning, Y. Ma, H. Yang, B. Yu, H. Yang, and Y. Wang, "Machine learning for electronic design automation: A survey," *ACM Trans. Design Autom. Electron. Syst.*, vol. 26, no. 5, pp. 1–46, Sep. 2021, doi: 10.1145/3451179.

[18] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Y. Young, and Z. Zhang, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2018, pp. 129–132, doi: 10.1109/FCCM.2018.00029.

[19] H. M. Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. P. Dinakarrao, H. Homayoun, and S. Rafatirad, "Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 397–403, doi: 10.1109/FPL.2019.00069.

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2004, pp. 75–86, doi: 10.1109/CGO.2004.1281665.

[21] (2022). *GCC, the GNU Compiler Collection*. [Online]. Available: https://gcc.gnu.org/

[22] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021, doi: 10.1109/TNNLS.2020.2978386.

[23] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proc. 34th Int. Conf. Mach. Learn. (PMLR)*, 2017, pp. 1263–1272.

[24] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, Jan. 2020, doi: 10.1016/j.aiopen.2021.01.001.

[25] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. 5th Int. Conf. Learn. Represent. (ICLR)*, 2017, pp. 1–14. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[26] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" in *Proc. 10th Int. Conf. Learn. Represent. (ICLR)*, 2022, pp. 1–26. [Online]. Available: https://openreview.net/forum?id=F72ximsx7C1

[27] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *Proc. 7th Int. Conf. Learn. Represent. (ICLR)*, 2019, pp. 1–17. [Online]. Available: https://openreview.net/forum?id=ryGs6iA5Km

[28] M. Liu, H. Gao, and S. Ji, "Towards deeper graph neural networks," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2020, pp. 338–348, doi: 10.1145/3394486.3403076.

[29] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2014, pp. 110–119, doi: 10.1109/IISWC.2014.6983050.

[30] L.-N. Pouchet and U. Bondugula. (2022). *PolyBench/C*. [Online]. Available: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[31] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 269–278, doi: 10.1145/3174243.3174255.

[32] Xilinx. (2022). *Vitis High-Level Synthesis User Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_2/ug1399-vitis-hls.pdf

[33] (2022). *Vivado Design Suite User Guide*. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_2/ug973-vivado-release-notes-install-license.pdf

[34] C. Cummins, Z. Fisches, T. Ben-Nun, T. Hoefler, M. O'Boyle, and H. Leather, "ProGraML: A graph-based program representation for data flow analysis and compiler optimizations," in *Proc. 38th Int. Conf. Mach. Learn. (ICML)*, Jul. 2021, pp. 2244–2253.

[35] (2022). *LLVM Language Reference Manual*. [Online]. Available: https://llvm.org/docs/LangRef.html
[36] (2022). *Embedding—PyTorch 2.0 Documentation*. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html
[37] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," in *Proc. ICLR Workshop Represent. Learn. Graphs Manifolds*, 2019, pp. 1–9.
[38] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2017, pp. 1025–1035.
[39] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–15.
[40] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (ELUs)," in *Proc. 4th Int. Conf. Learn. Represent. (ICLR)*, 2016, pp. 1–14.



M. USMAN JAMAL (Graduate Student Member, IEEE) received the M.S. degree from Politecnico di Torino, Italy, in 2018, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunications, under the supervision of Prof. Luciano Lavagno. His research interests include high-level synthesis, low-power high-performance computing, and machine learning for electronic design automation.



ZHUOWEI LI (Graduate Student Member, IEEE) received the B.E. degree from the Guangdong University of Technology, China, in 2019. He is currently pursuing the M.S. degree with the Department of Electronics and Telecommunications, Politecnico di Torino. His research interests include machine learning and robotics.



MIHAI T. LAZARESCU (Senior Member, IEEE) received the Ph.D. degree in electronics and communications from Politecnico di Torino, Italy, in 1998. He was a Senior Engineer with Cadence Design Systems and founded several startups. He is currently an Assistant Professor with Politecnico di Torino. He coauthored over 60 scientific publications, four books, and international patents. His research interests include design tools for WSN/IoT platforms, ubiquitous environmental sensing, efficient neural networks, indoor human localization, edge and leaf IoT data processing, and high-level HW/SW co-design and synthesis.



LUCIANO LAVAGNO (Senior Member, IEEE) received the Ph.D. degree in electrical engineering and computer science from UC Berkeley, in 1992. He was an Architect with the POLIS HW/SW co-design tool. Since 1993, he has been a Professor with Politecnico di Torino, Italy. From 2003 to 2014, he was an Architect with Cadence CtoSilicon high-level synthesis tool. He coauthored four books and over 200 scientific articles. His research interests include the synthesis of asynchronous circuits, HW/SW co-design, high-level synthesis, and design tools for wireless sensor networks.

Open Access funding provided by 'Politecnico di Torino' within the CRUI CARE Agreement