POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Politecnico di Torino

Master's Degree Thesis

A Graph Neural Network-based High-Level Synthesis Performance Prediction

Supervisors

Prof. Luciano LAVAGNO

M. Usman JAMAL

Candidate

Zhuowei LI

April 2023

Summary

With the rapid expansion of hardware development, there is a growing demand for more efficient and effective design techniques. Starting from a C/C++ behavioral-level programming language, High-Level Synthesis (HLS) is a solution for rapid prototyping of application-specific hardware design, where designers can apply HLS directives to optimize hardware implementations by trading-off between cost and performance. However, current commercial HLS tools do not provide reliable Quality of Results (QoR) estimations, which prevents designers from making aforementioned trade-offs and ensuring that the design complies with the constraints. Under the background of the widespread use of Machine Learning (ML) to improve the predictability of Electronic Design Automation (EDA) tools, we proposed Graph Neural Network (GNN) based predictive models that inductively learn the principle behind the HLS and downstream implementation by exploiting the graph representation power of the HLS design point.

We first created a dataset by choosing 30 designs from the widely-known real-case HLS benchmark suites, covering a broad range of application domains. This approach ensures that the trained ML model is robust enough to be generalized across various applications. We propose a method that provides graph-based representations of HLS based designs containing both program semantics and HLS synthesis directives information. We develop a multi-objective GNN-based learning model targeting a post-implementation estimation of resource usage and timing for a design point in milliseconds without invoking HLS tool. The experimental results demonstrate that our proposed model not only provides an improved prediction up to 74 % compared to *Vitis HLS*, but can also extend the knowledge learned for generalizing on *unseen* designs

Acknowledgements

I wish to convey my heartfelt appreciation and deepest gratitude to all those who have contributed significantly to the successful completion of my thesis. Their guidance, support, and encouragement have been vital in my academic journey.

I would like to express my sincere gratitude to Professor Luciano Lavagno, my supervisor, for providing me with the opportunity to conduct this research under his guidance and within his research group.

I am deeply grateful to M. Usman Jamal, my co-supervisor, for his unwavering guidance, boundless patience, and willingness to share his vast knowledge throughout the research process. His input has been essential in bringing this work to fruition.

Their insightful feedback and expertise that were valuable in shaping my research journey and professional growth. I am also grateful for their time and effort in reviewing my work and providing constructive feedback.

Lastly, I would like to acknowledge my family and friends for their unwavering love, moral support, and understanding. Their faith in my abilities and their constant encouragement have been the driving force behind my determination.

Table of Contents

List of Figures

List of Tables

Acronyms

ASIC Application-Specific Integrated Circuit

CDFG Control Data Flow Graph

CP Critical Path

DAGNN Deep Adaptive Graph Neural Network

DFG Data Flow Graph

DNN Deep Neural Network

DSE design space exploration

DSP Digital Signal Processing unit

EDA Electronic Design Automation

FF Flip-Flop

FPGA Field-Programmable Gate Array

GAT Dynamic Graph Attention Network

GCN Graph Convolutional Network

GIN Graph Isomorphism Network

GNN Graph Neural Network

HLS High-Level Synthesis

IR Intermediate Representation

ISA Instruction Set Architecture

LLVM Low Level Virtual Machine

LUT Lookup Table

MAE Mean Absolute Error

ML Machine Learning

MLP Multi-Layer Perceptron

MSE Mean Squared Error

NN Neural Network

QoR Quality of Results

RMSE Root Mean Squared Error

RTL Register-Transfer Level

Chapter 1

Introduction

1.1 Background

With the rapid expansion of hardware development, there is a growing demand for more efficient and effective design techniques. In recent years, High-Level Synthesis (HLS) has emerged as a significant method for hardware design, allowing high-level programming languages such as C/C++ to be automatically transformed into hardware designs[1]. Traditionally, hardware design has been an expensive and time-consuming process that required a high level of expertise and specialized knowledge.

Yet, the introduction of HLS has made it possible for software developers to create hardware using familiar high-level programming languages. By utilizing HLS, designers may benefit from the abstraction and modularity offered by high-level programming languages, which enables them to produce more sophisticated and powerful hardware designs with more portability in less time. In addition, designers can apply HLS directives to optimize hardware implementations by trading-off between cost and performance. This easy-to-use setup allows designers to rapidly and effectively prototype hardware designs, enabling them to experiment with different design configurations before involving in the final implementation.

1.2 Motivation

HLS has developed to meet the aforementioned demands, but the current commercial HLS tools do not provide reliable Quality of Results (QoR) estimations. As a result, designers are unable to trade-off between cost and performance and guarantee that the design complies with the requirements because its estimation results in terms of timing and resource usage frequently deviate from the actual QoR attained from post-implementation.

The use of Machine Learning (ML) techniques to boost Electronic Design Automation (EDA) has attracted widespread attention In recent years. ML-assisted EDA offers several benefits to designers, including enhanced design quality, decreased design time and effort, and increased predictability. Predicting the behavior of electronic systems allows designers to know design results before they are implemented, which is one of the main advantages of ML-assisted EDA. Moreover, designers may employ ML algorithms to automate the design process, allowing them to swiftly explore a larger design space and identify the optimal solution.

In this work, we concentrate on estimating the post-implementation QoR using a Graph Neural Network (GNN) based ML model.

1.3 State Of The Art

Machine Learning techniques have been applied successfully to address different challenges during the chip design flow [2]. These techniques have also been applied to solve the difference between the QoR estimations in HLS and post-implementation results. Table 1.1 summarizes the relevant state-of-the-art ML-based approaches for HLS prediction tasks and compares them with our contributions.

Dai et al. [3] and Makrani et al. [4] proposes *non-graph-based* ML models to estimate post-implementation resource usage and timing of a design by extracting global features from HLS synthesis reports mainly. In [3], they use the Linear model (Lasso), Artificial Neural Network (ANN), and XGBoost to calibrate the results produced from HLS reports. [4] uses Linear Regression, ANN, Support Vector Machine (SVM), Random Forest (RF), and Ensemble of the four models. Even though these studies are encouraging, however, they require considerable feature extraction engineering after High-Level Synthesis stage. Another concern is the restricted generalizable capability as the inputs to the model can only be extracted after HLS step. This means that for every new and unseen design, one must need to run time-consuming HLS, possibly taking hours for larger designs, to collect the features first before estimating the Quality of Results. On the other hand, our model predicts QoR without going through the HLS step for the unseen designs.

Wu et al. [5] and Ustun et al. [6] uses a graph-based ML models to perform HLS prediction tasks. [5] proposes an end-to-end reinforcement learning-based framework for design space exploration. GNN-based performance predictor (GPP) is integrated inside the framework to predict the post-route LUT and DSP utilization, and timing based on the data-flow graph (DFG) representation. [6] build a customized GNN based model to automatically learn operation mapping patterns to reduce the delay prediction for HLS based designs. Their approach results in a 72% reduction in

RMSE compared to Vivado HLS. These works evidently show the effectiveness of using Graph Neural Networks but they don't include pragmas in their input representation.

De et al. [7] compares both graph-based and non-graph-based Machine Learning models to improve delay prediction accuracy for ASIC HLS and proposes a hybrid model comprised of both local (structural) and global (Domain Knowledge) features. The global features are extracted from HLS reports, and therefore, one must run HLS during the inference phase which can require a long time based on the design.

Wu et al. [8] propose a graph-based ML approaches to estimate resource usage and timing based on different HLS stages. The input graphs are constructed from the IR operator information (*.adb file) and the features are extracted from both *.adb file, and HLS intermediate results. **.adb file* contains information not only related to the operation type of a node but also information like whether a node is the starting node of a path and its cluster number. Instead, in our case, we take the information from LLVM Bitcode (*.bc) which is generated right after the HLS front-end compilation process. Moreover, the work in [8] doesn't consider the HLS synthesis directives unlike ours.

Table 1.1: *Comparison of ML-based Approaches for HLS Prediction Tasks*

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

1.4 Thesis Overview

This thesis aims to explore the use of GNNs for predicting post-implementation QoR metrics of HLS-generated designs. Fig. 1.1 depicts the overall workflow of our proposed framework. On the left-hand side, we present the general C-to-circuit workflow, which includes the input, HLS, and downstream implementation stages. On the right-hand side, we illustrate the input data and corresponding labels that we can obtain from the left-hand side of the figure. Specifically, we extract an HLS Intermediate Representation (IR) graph that represents the functionality of the design, which serves as the input data for our proposed GNN-based predictive model. The ground truth labels used to train the model are extracted from post-implementation reports that are generated after the place and route phase.

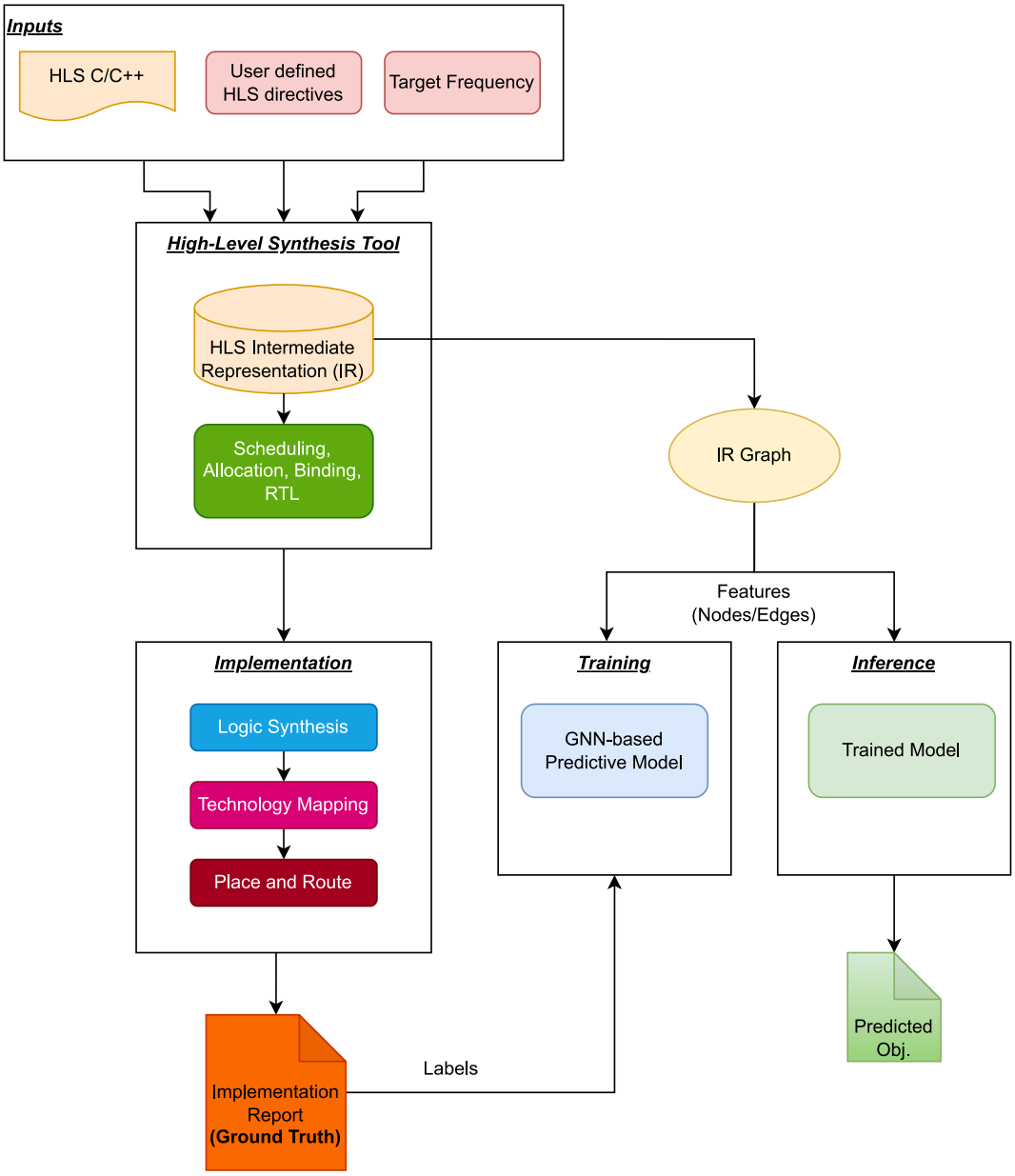

Figure 1.1: Overall framework flow

The thesis is structured into several chapters as follows:

- Chapter 1 introduces the thesis and outlines the motivation behind the research.
- Chapter 2 delves into the general concepts of HLS and *Vitis HLS*.

- Chapter 3 discusses Low Level Virtual Machine (LLVM) and its utilization to convert the program into a graph representation.
- Chapter 4 covers general concepts of ML, including categorical data encoding techniques.
- Chapter 5 explores the fundamental concepts of GNNs and the various variants
- Chapter 6 presents our dataset generation framework and feature selection.
- Chapter 7 details the architecture of our GNN-based predictive models, including their training and inference procedures.
- Chapter 8 presents the experimental results and evaluates the efficacy of our proposed approach.
- Chapter 9 concludes the thesis work and provides an outlook on future research directions.

Chapter 2

High Level Synthesis

2.1 Introduction

High-Level Synthesis (HLS) is a hardware design methodology for developing hardware circuits from high-level programming languages like C/C++. It has increasingly become a hot spot for hardware development in recent years, as it allows for more efficient and effective design techniques to cope with the rapid growth of hardware development.

HLS technology enables engineers to design digital systems at a higher level of abstraction than traditional hardware design methods. Instead of using a hardware description language (HDL) such as Verilog or VHDL to design digital systems, engineers can use a high-level programming language such as C, C++, or SystemC to describe the functionality. These high-level languages provide engineers with more abstraction and modularity, enabling them to create more complex and powerful hardware designs with greater portability in less time[9].

2.2 General Workflow

The workflow of HLS typically involves several phases and is shown in Fig. 2.1. Initially, the HLS tool translates the high-level functional description of the design into a Register-Transfer Level (RTL) representation. A hardware description language (HDL), such as VHDL or Verilog, is often used to express the resulting RTL representation. This HDL code can then be used with tools like synthesis and place-and-route for the physical implementation of the design. These tools optimize and map the RTL representation to a target architecture, such as Field-Programmable Gate Array (FPGA) or Application-Specific Integrated Circuit (ASIC), and eventually generate a configuration file or bitstream that can be loaded onto the target device such as for testing or deployment.

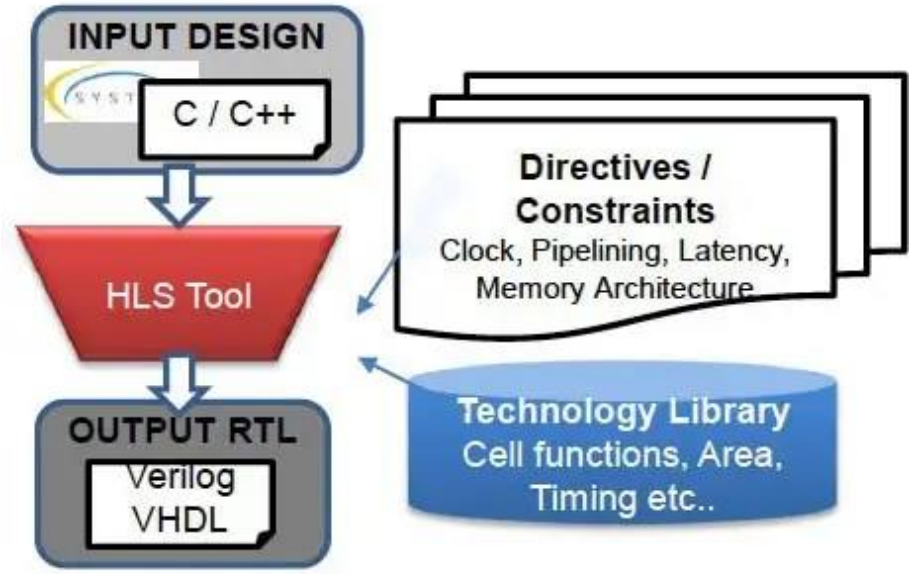

Figure 2.1: General HLS workflow[10]

2.3 Quality of Results

In HLS, Quality of Results (QoR) is a comprehensive and multidimensional metric employed in the field of digital circuit design to evaluate and optimize the overall performance, effectiveness, and functionality of a synthesized circuit. Numerous factors are taken into consideration by QoR, including resource usage, timing, power consumption, area, design productivity, reliability, and scalability. These factors collectively contribute to the success of the design project and its ability to meet specific requirements and goals.

In this section, we concentrate on two key factors of QoR in HLS: resource usage and timing, which are explained in depth as follows:

- **Flip-Flops (FFs)**, which serve as data storage components and can store a single binary digital bit of data, are the basic building blocks of digital circuits. These storage components are essential to the sequential logic used in electronics, where the current input and state as well as the output decide the next state.

- **Lookup Tables (LUTs)** are a type of digital circuit used in FPGAs to execute any Boolean function based on specified input-output relationships.

These programmable logic units serve as unique truth tables that are filled with values in accordance with the needs and instructions of the designer.

- **Digital Signal Processing units (DSPs)** are specialized processing components in FPGAs that are designed with the aim of efficiently performing arithmetic operations like multiplication and accumulation. These dedicated hardware units enable FPGA-based designs to handle computationally intensive tasks at high speeds.

- **Critical Path (CP)** is a crucial metric in evaluating the timing performance of digital circuits, which measures the longest propagation delay between input and output signals and directly affects the maximum operating frequency of the design.

2.4 Vitis High-Level Synthesis

Xilinx High-Level Synthesis tool *Vitis HLS* allows designers to write the design algorithm in high-level languages such as C, C++, and SystemC. This design algorithm is then translated into hardware specifications that can be used in programmable devices like FPGA. *Vitis HLS* significantly simplifies the tedious, time-consuming, and error-prone process of creating RTL code, which formerly required designers to grapple with low-level hardware implementation[11].

In essence, high-level synthesis serves as a link between hardware and software, offering a variety of advantages. The ability to work at a higher level of abstraction when developing high-performance hardware is provided by HLS, which boosts the productivity of hardware designers. Moreover, HLS offers the ability to create various solutions on several architectural platforms without altering the C/C++ source code. This makes it possible to explore the design space and aids in determining the best implementation.

2.4.1 Development Flow

Fig. 2.2 illustrates the *Vitis HLS* Development Flow which is explained as follows:

- Architect the algorithm in accordance with the necessary Design Principles

- **C-Simulation**: Compile C/C++ code with Clang. Execute the C/C++ code to simulate its behavior and ensure it works as expected by comparing it to the C/C++ testbench. The C/C++ top function serves as the main input to *Vitis HLS*.

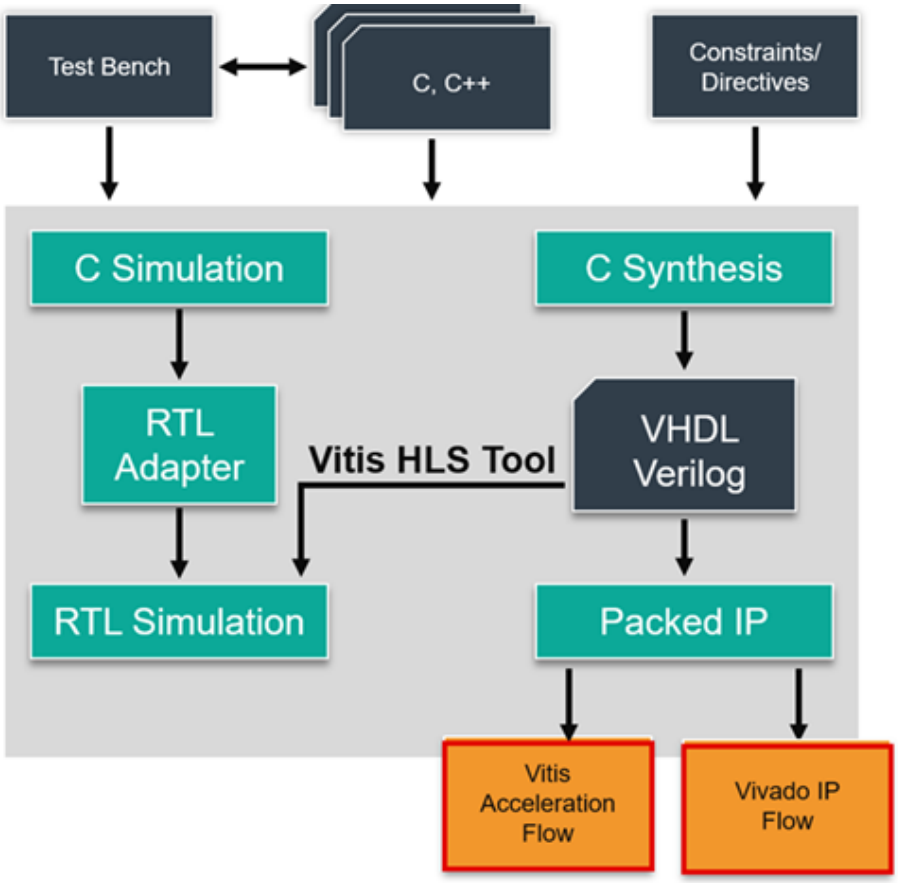

Figure 2.2: Vitis HLS workflow[11]

- **C-Synthesis**: Generate the RTL by synthesizing the C/C++ top function by using HLS. To instruct the synthesis process to carry out a certain optimization, HLS synthesis directives i.e. *HLS pragmas* and constraints can be imposed directly. once C-Synthesis is done, a report including estimated timing hardware resource utilization and timing information is generated. When C-Synthesis is finished, a comprehensive report with time and hardware resource usage estimation is produced, offering the designer crucial references for subsequent refinement and optimization.

- **Co-Simulation**: C/RTL Co-Simulation in *Vitis HLS* refers to the process of verifying and validating a hardware design written in RTL (register-transfer level) using a C/C++ simulation. The designer can verify the correctness of the RTL implementation of the design by comparing the results from the RTL and C/C++ testbench. Any discrepancies between the two outputs can be analyzed and corrected, and this simulation can be performed again until the

outputs match.

- **Analyze**: Review and investigate the HLS synthesis reports and co-simulation reports

- Repeat prior steps until the desired performance is achieved

In this work, we employ Vitis HLS *to create our dataset by synthesizing the well-known HLS benchmark.*

2.4.2 Optimization Directives

HLS synthesis directives i.e. *HLS pragmas* are essential for optimizing the HLS process in *Vitis HLS*, a prominent HLS tool developed by Xilinx. Pragmas provide directives to the compiler, guiding the generation of hardware implementations from high-level C, C++, or OpenCL code. They enable users to control the fine-grained aspects of the design, allowing for the optimization of specific objectives such as resource usage, performance, or power consumption.

In *Vitis HLS*, there are several types of synthesis directives, including loop-level, variable-level, and function-level. These optimizations can significantly improve the performance of the synthesized hardware, albeit often at the expense of increased resource utilization. The details are explained as follows:

- **Loop-level pragmas** focus on optimizing the execution of loops in the design, which includes loop pipelining, loop unrolling, loop flattening, etc.

  Loop pipelining reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations. Fig. 2.3 shows an example of the same function with and without loop pipelining. example Comparison of function with and without loop pipelining

  Loop unrolling is for creating multiple independent operations of the loop body rather than a single collection of operations, which enables some or all loop iterations to occur in parallel.

  Loop flattening allows nested loops to be flattened into a single loop hierarchy with improved latency.

- **Variable-level pragmas** are applied to specific variables in the design and can be used to control their storage or access characteristics. Examples include array partitioning, which can partition an array into smaller sub-arrays, improving parallel access, and array reshaping, which changes the array's storage organization to optimize specific access patterns.

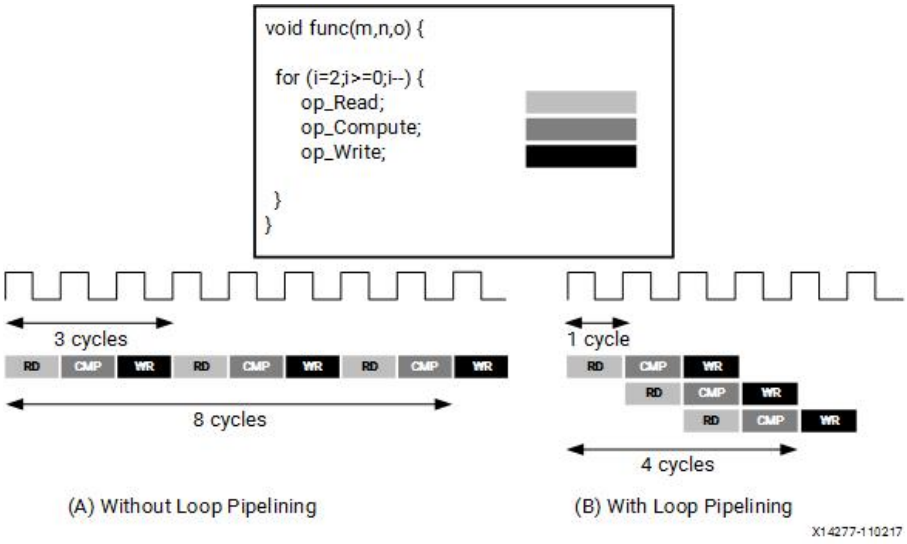

Figure 2.3: Comparison of function with and without loop pipelining in Vitis HLS[11]

- **Function-level pragmas** influence the overall behavior of a function. One Example of function-level pragmas is function inlining, which removes a function as a separate entity in the hierarchy. After inlining, the function is dissolved into the calling function and no longer appears as a separate level of hierarchy in the RTL.

In this work, we utilize aforementioned HLS synthesis directives i.e. HLS pragmas *create our dataset to cover as much the design space as possible*

Chapter 3

Low Level Virtual Machine

3.1 Introduction

Low Level Virtual Machine (LLVM) is a group of compiler and toolchain technologies that enables the development of both the front-end and back-end for programming languages on any Instruction Set Architecture (ISA)[12]. A language-independent IR, which serves as a portable high-level assembly language and may be improved to produce a new IR, is its main component. The new IR can then be translated and linked into platform-specific machine-dependent assembly language code.

By receiving IR code from a compiler and generating an optimized IR, LLVM can supply the middle layers of a comprehensive compiler system. Every instruction is in the static single assignment(SSA) form and provides a language-independent instruction set and type system. This makes it easier to analyze the dependencies between variables since each variable, also known as a typed register, is only allocated once and then frozen.

Moreover, LLVM is able to accept the IR from the GNU Compiler Collection (GCC) toolchain, enabling it to work with a variety of compiler front-ends that have already been developed for different projects. Without having to build and implement a unique compiler from scratch, its adaptability and portability make it a great tool for developers to construct new programming languages on any ISA.

3.2 LLVM Structure

The LLVM infrastructure provides a flexible and modular approach to compiler design, with separate front-end, mid-end, and back-end (see Fig. 3.1) components that can be combined to support a wide range of programming languages and hardware platforms.

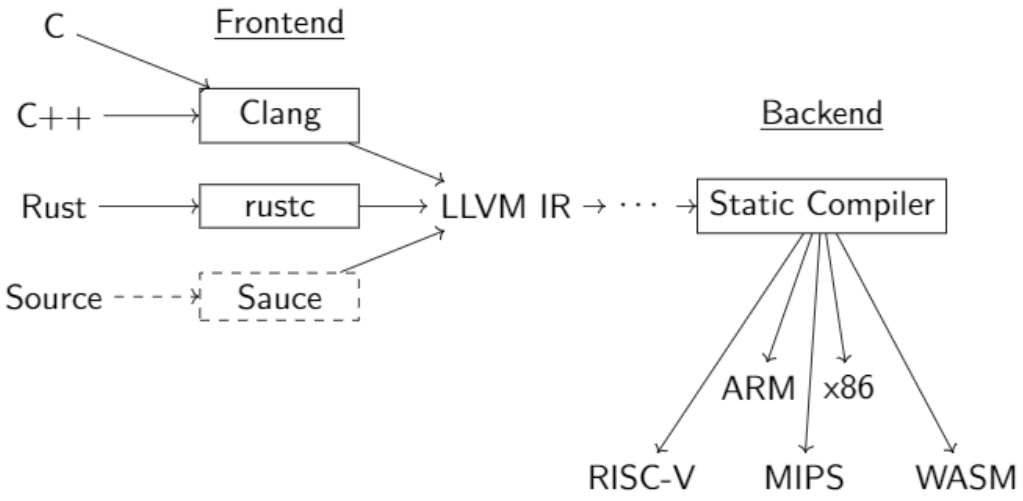

Figure 3.1: LLVM structure: Front-end, LLVM IR, Backend[13]

- The **front-end** of LLVM is responsible for converting source code written in a specific programming language into LLVM IR. This component is language-specific and is designed to handle the idiosyncrasies of the programming language. LLVM currently supports front-ends for a variety of programming languages, including C, C++, Objective-C, Fortran, Swift, and Rust. Each front-end takes the source code as input and produces LLVM IR as output.

- The **mid-end** of LLVM performs optimizations on the LLVM IR to improve program performance and efficiency. This component is responsible for performing a wide range of optimizations, such as dead code elimination, loop unrolling, and common subexpression elimination. These optimizations can significantly improve program performance by reducing the amount of redundant work that the program needs to do. The mid-end is also responsible for generating code that is suitable for the back-end to use.

- The **back-end** of LLVM generates machine code for a specific ISA based on the optimized LLVM IR. This component is responsible for translating the LLVM IR into instructions that can be executed by the target hardware. LLVM currently supports back-ends for a wide range of ISAs, including x86, ARM, MIPS, PowerPC, and RISC-V. Each back-end generates machine code that is tailored to the specific characteristics of the target hardware.

3.3 LLVM Intermediate Representation

A language-independent Intermediate Representation (IR) is a key component of the LLVM infrastructure. The LLVM IR is a low-level code representation designed to be used by the LLVM compiler framework. It is designed to be language-independent, meaning that it can be used to represent code written in any programming language.

The fundamental feature of the LLVM IR is that it can be extracted as a graph that can be represented in several forms, such as a Data Flow Graph (DFG) or Control Data Flow Graph (CDFG). While a CDFG provides both data and control dependencies, DFG only illustrates the data dependencies between program instructions. These graphs are helpful for program analysis and optimization because they provide insightful information about how programs behave.

Using LLVM IR as a graph representation has several benefits as follows:

- Providing a more compact representation of the program compared to a large sequence of instructions.
- Mapping naturally to program structure, where nodes represent instructions and edges represent dependencies, makes it easier to understand the behavior of the program, which is critical for program optimization.
- Developers can visualize program structure and performance characteristics more intuitively. This helps identify areas that can be optimized for better performance or detect bugs that may be challenging to spot through traditional debugging techniques.
- Representing a program as a graph allows developers to apply graph analysis techniques to the program. They can use graph algorithms for optimization tasks like dead code elimination, loop unrolling, and common subexpression elimination. They can also perform program analysis tasks such as data flow analysis and control flow analysis using graph-based techniques.

The LLVM IR is typically represented using one of two file formats: *.ll* or *.bc*. The *.ll* format is a human-readable text format that is easily understood and modified by developers. On the other hand, the *.bc* format is a binary format that is more efficient for storing and processing large amounts of code.

The LLVM Disassembler is part of the LLVM compiler infrastructure and is used to disassemble LLVM bitcode files (*.bc* files) into human-readable LLVM assembly language files (*.ll* files). The tool is commonly used for debugging and analyzing LLVM bitcode, as it allows users to examine the LLVM IR of a program in a human-readable format. Once the bitcode has been disassembled into LLVM assembly language, other tools can be used to further transform the code for various purposes.

3.4 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* (IR) graph. Because the essence of a program is a pre-designed process flow, an *IR* graph is a data structure that represents the information of a program and it can be swiftly extracted after the front-end compiler.

Modern HLS tools(see Fig. 3.2) are developed and based on state-of-the-art compilers like LLVM[14] and GCC[15]. The input is a *high-level code* representing the functionality of the design and can be written in programming languages like C/C++. The input is passed to the front-end of the HLS tool before different IR transformations are applied and on completion of the compilation outputs an *IR* geared towards hardware circuit generation. An *HLS IR* is made up of *basic blocks* where each block contains assembly-like instructions with neither branches in nor branches out except at the start and the exit of the block respectively. Each HLS IR can be transformed into a different kind of graph like *control-flow*, *data-flow*, and *call-flow* for specific information extraction.

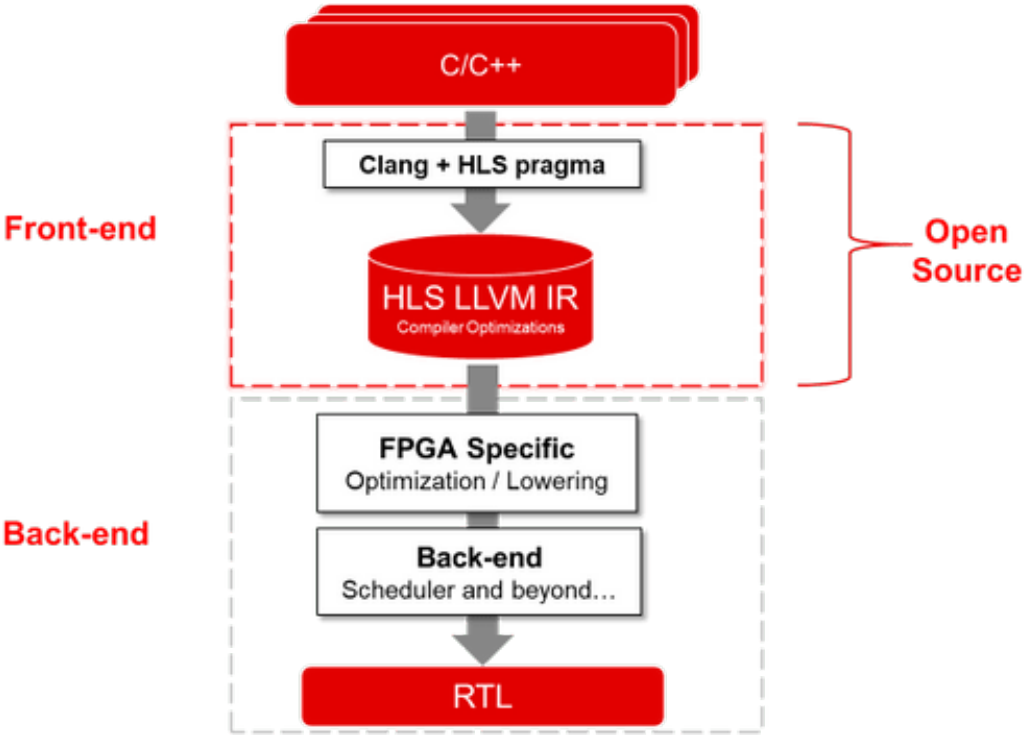

Figure 3.2: Vitis HLS software architecture[16]

In the control-flow, the graph nodes are inserted in terms of the LLVM instructions of the program, and the edges illustrate how instructions propagate; Data-flow reveals the formation flow from the raw data served as operands of instructions and from these operations to processed data in a form closely resembling Static Single Assignment(SSA). Operands are represented as graph nodes. Data dependencies between the nodes are determined by each node's potential existence of one or more in- and out-edges; And the call-flow expresses the calling relationship between sub-functions inside the program. *We have considered a combination of these three graphs for our experiments.*

In this work, we compiled our design program using the HLS front-end to obtain the HLS LLVM IR. We then converted the generated IR into a graph representation, which made it adaptable for use with GNNs. This allowed us to leverage the powerful representation capabilities of GNNs to gain insight into the program more effectively.

Chapter 4

Machine Learning

4.1 Introduction

Machine Learning (ML) is dedicated to developing algorithms that are able to automatically discover patterns in incoming data without requiring explicit programming[17]. This makes it possible for ML algorithms to handle a wide range of challenging problems, such as speech recognition, natural language processing, image recognition, and others.

Input data, a learning algorithm, and eventually a well-trained output model are the three key components that are often included in ML methods. A learning algorithm trains an output model from the input data so that it can make predictions or decisions about fresh data.

4.2 Supervised and Unsupervised Learning

This model can be trained using either supervised or unsupervised learning techniques.

- **Supervised learning** involves using labeled datasets for training models. These datasets are composed of input-output pairs, where each input is associated with a corresponding output, also known as a "label" or "target." The goal of supervised learning is to use these labeled examples to teach the model how to accurately predict the correct output given a new input. During the training process, the model is presented with many examples from the labeled dataset and learns to make predictions by adjusting its internal parameters based on the differences between its predicted outputs and the ground truth labels. By repeatedly presenting the model with labeled examples and adjusting its parameters, the model gradually improves its ability to accurately predict outputs for new inputs.

- **Unsupervised learning** can identify underlying patterns or structures in data without the need for labeled examples, which makes it valuable in circumstances when labeled data is not readily available or prohibitively expensive to obtain. Unsupervised learning methods work with unlabeled datasets, where the input data is not connected to any matching output. Instead of trying to predict a specific output, unsupervised learning algorithms focus on identifying relationships or similarities between the input data points. One common technique used in unsupervised learning is clustering, where the algorithm groups together similar data points based on their features. Another type of unsupervised learning is dimensionality reduction, which seeks to identify the most important or relevant features of the data by reducing its overall complexity. This can be useful for visualizing complex data in lower dimensions, or for identifying hidden variables or factors that are influencing the observed data.

In this work, we frame our prediction problem within the context of supervised learning.

4.3 Model Evaluation

Model evaluation is a crucial step in ML, as it allows us to assess the performance of a well-trained model and determine its ability to generalize unseen data. In this section, we discuss two commonly used approaches to model evaluation: single test and cross-validation.

- A single test is a simple approach to model evaluation that involves randomly splitting the dataset into three parts: training, validation, and test sets, typically in a 7:1:2 ratio. The model is trained on the training set, and the validation set is used to optimize hyperparameters and prevent overfitting. Finally, the model is tested on the test set to evaluate its performance. This approach is basic and computationally efficient, but it may not always provide an accurate estimate of the model's performance, especially if the dataset is small or unbalanced.

- Cross-validation, a more robust approach to model evaluation that addresses the limitations of the single test approach. Typically, there are two distinct implementations of this approach: conducting k-fold cross-validation on the whole dataset or only on the training set.

  The first implementation involves dividing the whole dataset into k equally-sized subsets, where the model is trained and tested k times. In each iteration, a different fold serves as the test set, and the remaining $k - 1$ folds combine

to form the training set. This ensures that each data point has a chance to be included in both the training and test sets, providing a comprehensive evaluation of the model's performance. The performance metrics of each fold are averaged to provide a reliable measure of the model's overall performance.

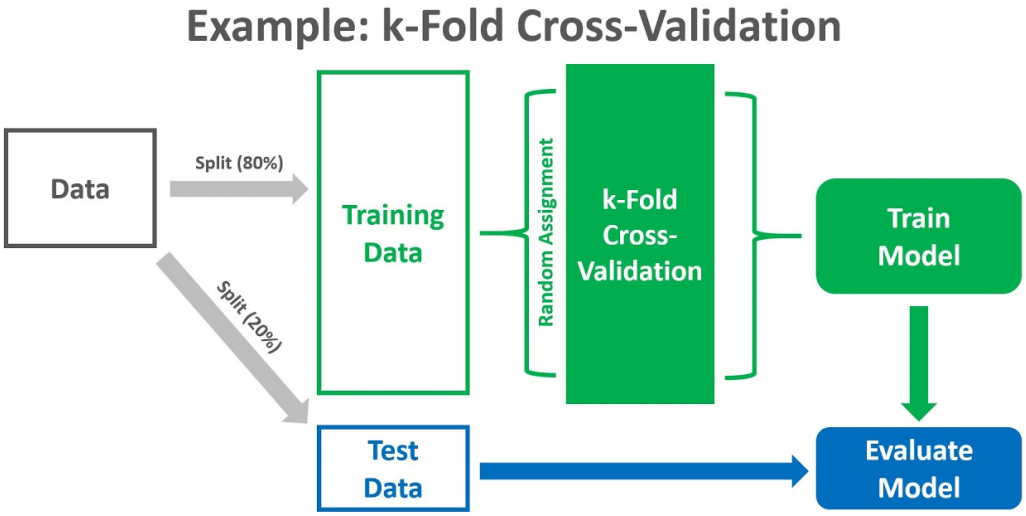

Figure 4.1: k-Fold Cross-validation on the training set[18]

The second implementation conducts k-fold cross-validation exclusively on the training set and reserves a separate holdout set for final model evaluation. As shown in Fig. 4.1, the training set is divided into k equally-sized folds, and the model is trained and tested k times in a similar manner to the first implementation. The cross-validation is only applied to the training set, while the holdout set is used to evaluate the model's performance on unseen data. Although less common, this approach provides additional validation for certain applications, ensuring that the model's true performance is evaluated on entirely new data.

The performance of the model can be evaluated using various metrics, depending on the problem domain and the type of model. For classification tasks, metrics such as accuracy, precision, recall, and F1 score are commonly used. For regression tasks, metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE) are often used.

4.4 PyTorch for Implementation

For developing and deploying ML models, an open-source framework called PyTorch is frequently utilized. It offers several optimization methods, such as automated differentiation, which enables the accurate determination of gradients during training[19]. In addition, PyTorch provides a rich set of libraries for building ML models, including TorchVision for computer vision, TorchText for natural language processing, and PyTorch Geometric[20] for graph-based ML.

4.5 Encoding Categorical Data

In ML, categorical data is a sort of data that represents qualitative variables with a limited number of discrete categories. Examples of categorical data include gender (male, female, other), education levels (high school, bachelor's degree, master's degree, Ph.D.), and work title (manager, engineer, analyst). Categorical data must be encoded in a numerical representation before it can be used as input for a ML model. One-hot encoding and embedding are two methods that are frequently used to encode categorical data.

- **One-hot encoding** is a well-liked method in ML for encoding categorical data. Each category is represented in this method as a binary vector, where each element in the vector corresponds to a category and is either 0 or 1, with 1 indicating the category's presence. As it enables the Neural Network (NN) to quickly distinguish between categories without assuming any underlying structure or relationships between them, one-hot encoding is frequently employed in NNs. Although its ease to use, one-hot encoding has some drawbacks. The fact that it produces high-dimensional sparse vectors, which can be difficult to process computationally, especially when working with huge datasets, is one of its main drawbacks. One-hot encoding also fails to capture any underlying structure or relationships between categories, which may restrict the model's capacity to generalize new categories that have not yet been discovered.[21].
- **Embedding** is another technique used for encoding categorical data in Machine Learning. Category representation in embedding is in the form of dense vectors of continuous values, as opposed to one-hot encoding, which uses high-dimensional, sparse vectors to represent categories. The idea behind embedding is to convert categorical data into a continuous space and learn a mapping lookup table where similarity is captured by representing similar categories by similar vectors in that space. Embedding is superior to one-hot encoding in several ways. First, it leads to low-dimensional dense vectors, which are more computationally efficient to handle. Second, embedding captures the underlying structure and relationships between categories, which can

improve the ability of the model to generalize to previously unseen categories. Embedding is commonly used in natural language processing tasks, where words are represented as embeddings to capture their semantic and syntactic relationships[22].

Chapter 5

Graph Neural Network

Recently, there has been a significant surge of interest in using Deep Neural Network (DNN) for the analysis of graphs, also classified as Graph Neural Networks (GNNs)[23]. GNNs are a subclass of NNs that operate on graph-structured data.

5.1 Graph Theory

5.1.1 Graphical Data

Graphs are a type of data structure that helps us to represent complex information explicitly by building relationships between objects. A graph is represented as $\mathcal{G}(V, E, A)$ where V and E are node and edge set respectively and $A \in \mathbb{R}^{n \times n}$ is an adjacency matrix that represents the structure of the graph with n nodes (see Fig. 5.1, Fig. 5.2 and Fig. 5.3). Additionally, a graph may have node features X where $X \in \mathbb{R}^{n \times d}$ is a node attribute matrix where the attribute of i-th node is represented by a d-dimension vector in an i-th row.

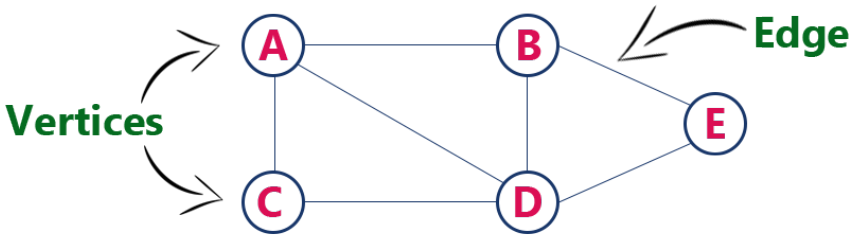

Figure 5.1: A simple graph example[24]

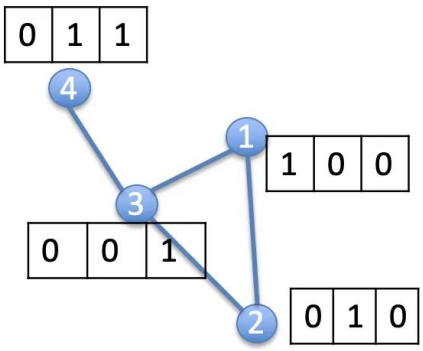

Figure 5.2: A simple example of a graph with a feature assigned to each node[24]

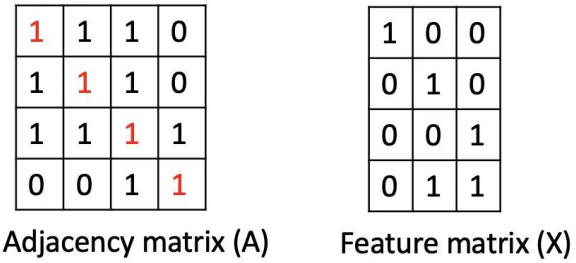

Figure 5.3: A simple example of the adjacency matrix and feature matrix[24]

5.1.2 Challenges in Analyzing Graph Data

The non-Euclidean nature of graphs, the absence of a fixed form, and the difficulty in visualizing large graphs present significant challenges when analyzing graph data, and a detailed explanation of these challenges is provided as follows:

- The inability to represent graphs in a Euclidean space makes it more difficult to analyze graph data since it is not easily mappable to a coordinate system. This makes it challenging to use traditional data analysis methods since they require a Euclidean space for data representation.
- The absence of a fixed form in graphs also makes it difficult to be analyzed. It can be challenging to recognize patterns or connections of a network due to the fact that graphs might share the same adjacency matrix yet have entirely different structures and are visually distinct.
- Due to the high dimensions and dense node grouping, displaying big networks is also a complex problem. Human interpretation and analysis of the graph data may be difficult because of its magnitude and complexity.

Despite these challenges, GNN offers a promising solution for analyzing graph-based data efficiently. By leveraging the structural information of graphs, GNN can extract meaningful features that capture the crucial structural patterns in the graph, and therefore gain insights from the complex graph data[25].

5.2 Overview and Architecture

GNN has emerged as a powerful tool for processing and learning from graph-structured data. The fundamental principle of GNNs is to learn features that capture significant structural patterns in the network by propagating information via edges and nodes. The direct operation on the node feature vectors (node embeddings) in the graph enhances its capacity to capture both structural and contextual information. A GNN is a model which learns a trainable non-linear mapping function F such that $H = F(A, X)$ where $H \in \mathbb{R}^{n \times k}$ is an embedding matrix which keeps the local structural information of the graph.

GNN iterates over each node's features through a series of cascading layers to extract the graph information by learning the features of each node. An aggregate (AGG) and update (UPDATE) function are applied to each node in every layer. An *AGG* function receives information from the neighboring nodes and sends the aggregated information to the $UPDATE$ function to update the current node embeddings based on the aggregated message and the node's previous layer embedding. Fig. 5.4 depicts an aggregation example in which node A is the target node. A *READOUT* function is applied after the last layer to summarize the embeddings from all the nodes to assemble a graph-level embedding vector. Eq. (5.1), Eq. (5.2) and Eq. (5.3) summarize the message passing-based architecture[26] of GNN model.

$$m_i^{t+1} = \text{AGG}\left(h_u^t \mid u \in \mathcal{N}(i) \cup \{i\}\right) \tag{5.1}$$

$$h_i^{t+1} = \text{UPDATE}\left(m_i^{t+1}\right) \tag{5.2}$$

$$h_{\mathcal{G}} = \text{READOUT}\left(h_i^T, i \in V\right) \tag{5.3}$$

where the embedding for node i at iteration t is denoted by h_i^t, $N(i)$ represents the neighborhood of node i, $\mathbf{m}_i^{t+1}$ represents the aggregated message from the target node and its neighbors, V is the node set of the graph $\mathcal{G}$, and T is the number of message-passing iterations, $h_{\mathcal{G}}$ is the final graph-level embedding.

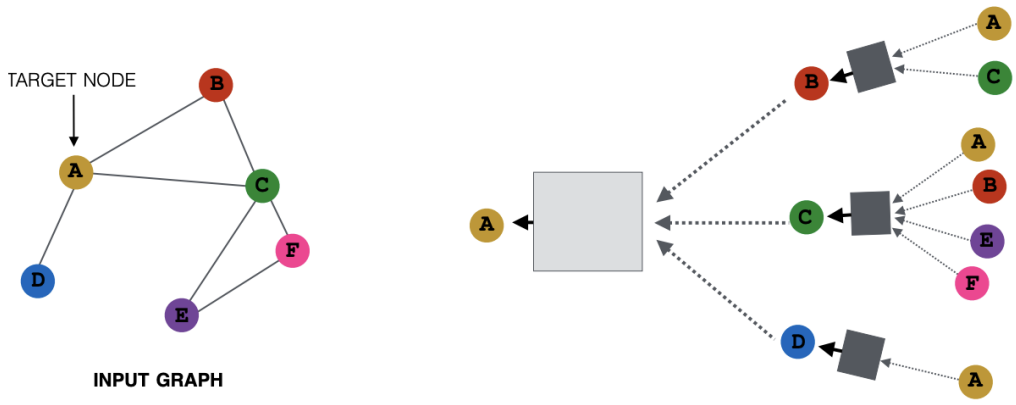

Figure 5.4: Aggregation of information from the neighbors of the target node[27]

5.3 Graph Neural Network Models Variants

Generally, different GNN models differ from each other based on different aggregate and update functions[25]. Any permutation-invariant operation can serve as an *AGG* function, and any differentiable function can be used as an $UPDATE$ function. Fig. 5.5 shows a general GNN model. *In our experimentation, we have used 4 different GNNs, specifically Graph Convolutional Network (GCN)[28], Dynamic Graph Attention Network (GAT)[29], Graph Isomorphism Network (GIN)[30] and Deep Adaptive Graph Neural Network (DAGNN)[31].*

5.3.1 Graph Convolutional Network

Graph Convolutional Network (GCN)[28] represents an efficient variant of Convolutional Neural Network(CNN) specifically designed for processing graph-structured data. The core idea behind GCN is to extend the concept of convolution, commonly used for grid-structured data (e.g., images). To achieve this, GCN employs a localized graph convolution operation that aggregates information from neighboring nodes and transforms the aggregated information using a shared weight matrix.

GCN leverages an aggregation function to implement a weighted summation of the embeddings of neighboring nodes, taking into account node degrees and edge weights. A typical GCN architecture consists of several layers, with each layer performing a graph convolution operation followed by a non-linear activation function. This architecture allows GCN to effectively capture both local and global information in graph-structured data.

The node-wise formulation that describes how GCN iterates the node embeddings

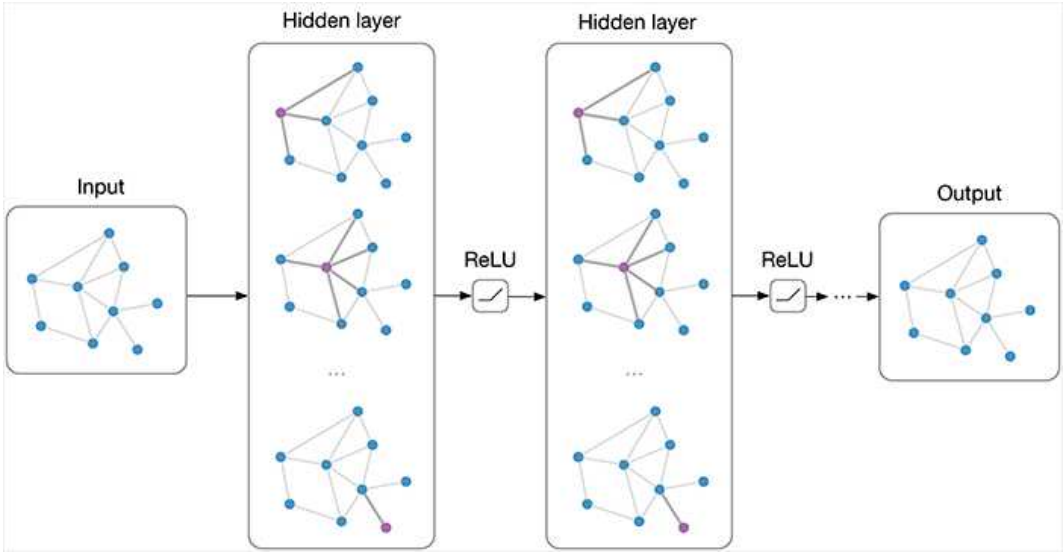

Figure 5.5: A general GNN model[32]

combining aggregation and update operation is presented as Eq. (5.4).

$$\mathbf{x}'_i = \mathbf{\Theta}^\top \sum_{j \in \mathcal{N}(v) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j \tag{5.4}$$

with $\hat{d}_i = 1 + \sum_{j \in \mathcal{N}(i)} e_{j,i}$, where $\mathcal{N}(i)$ is the set of neighbors of node i, $e_{j,i}$ denotes the edge weight from source node j to target node i which is set to 1.0 by default.

5.3.2 Dynamic Graph Attention Network

Dynamic Graph Attention Network (GAT) [29] aims to overcome the limitations of GCN, which assigns equal weights to each node neighbor embeddings. GAT employs masked self-attention layers to learn the importance of node neighbors, which enables it to assign various weights to the contributions of its neighbors accordingly in the update phase.

To achieve this, GAT introduces an innovative attention mechanism that allows the model to focus on relevant information. This mechanism involves learning attention coefficients that weigh the significance of each neighboring node during the aggregation process. GAT implements this attention mechanism using a shared self-attention function to compute the attention coefficients, which is normalized across all neighbors $j \in \mathcal{N}_i$ using softmax formulated as Eq. (5.5):

$$\alpha_{i,j} = \text{softmax}_j \left(e \left(\boldsymbol{h}_i, \boldsymbol{h}_j \right) \right) \tag{5.5}$$

where a scoring function $e \left(\boldsymbol{h}_i, \boldsymbol{h}_j \right) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ computes a score for every edge (j, i).

The scoring function indicating the importance of the features of the neighbor j to the node i is formulated as Eq. (5.6):

$$e\left(\boldsymbol{h}_i, \boldsymbol{h}_j\right) = \boldsymbol{a}^{\top} \operatorname{LeakyReLU}\left(\boldsymbol{W} \cdot [\boldsymbol{h}_i \| \boldsymbol{h}_j]\right) \tag{5.6}$$

where $\boldsymbol{h}_i$ and $\boldsymbol{h}_j$ are the feature vectors of nodes i and j, respectively, $\boldsymbol{W}$ is a trainable shared linear transformation, $\boldsymbol{a}$ is a trainable shared attention vector, $\|$ denotes concatenation, LeakyReLU represents Leaky Rectified Linear Unit, serving as an activation function for introducing non-linearity.

The computed attention coefficients are then used to perform a weighted summation aggregation of the target node's embeddings about those of its neighbor nodes. The node-wise formulation that describes how GAT iterates the node embeddings combining aggregation and update operation is presented as Eq. (5.7).

$$\mathbf{x}_i' = \alpha_{i,i}\boldsymbol{\Theta}\mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j}\boldsymbol{\Theta}\mathbf{x}_j \tag{5.7}$$

where $\mathcal{N}(i)$ is the set of neighbors of node i, the attention coefficients $\alpha_{i,j}$ are computed as Eq. (5.5).

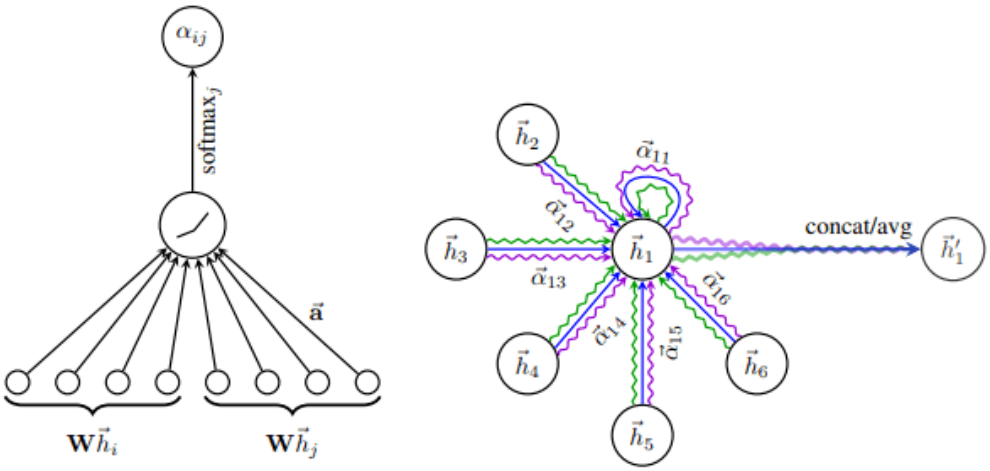

Figure 5.6: Attention in GAT. Left: The attention mechanism. Right: An illustration of multi-head attention on its neighborhood.

To improve the expressive power of the model, GAT typically employs multiple attention heads in parallel, allowing the model to capture different aspects of the graph structure simultaneously (see Fig. 5.6).

5.3.3 Graph Isomorphism Network

Graph Isomorphism Network (GIN)[30] is a powerful GNN variant with significant expressive capacity. The Weisfeiler-Lehman (WL) test indicates that the GIN

theoretical expressive is the upper limit of all GNN variations, which makes it the ideal model for dealing with graphs that exhibit structural similarities but are represented differently where it is challenging to determine their isomorphism.

The universal approximation theorem, which allows for arbitrary precision approximations of any permutation-invariant function when given sufficient depth, is the foundation for the expressive capability of GIN. In order to provide GIN the capacity to distinguish between various graphs, it uses MLPs as its aggregation functions. This guarantees that these functions are injective.

Fig. 5.7 depicts the architecture of GIN where the aggregation function employed by GIN is a MLP that processes the sum of embeddings of the target node and its neighborhood, followed by a linear transformation and a non-linear activation function. This process is repeated multiple times to generate a fixed-length feature vector representing the entire graph.

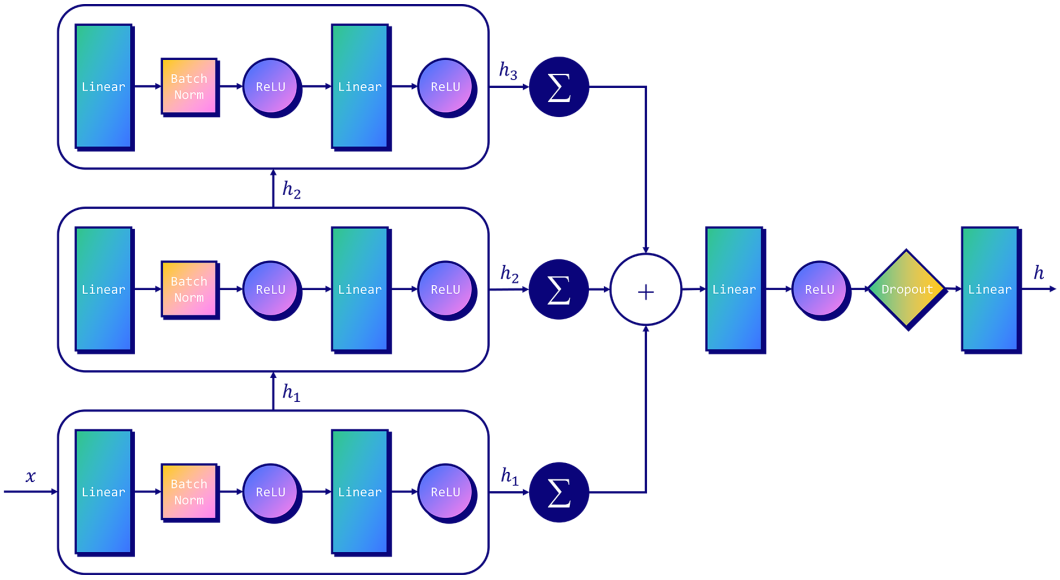

Figure 5.7: Architecture of GIN[33]

The aggregation, update, and readout functions of GIN are detail formulated as Eq. (5.8), Eq. (5.9), Eq. (5.10) respectively.

$$\mathbf{m}_i' = (1+\epsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \mathbf{x}_j \tag{5.8}$$

where $\mathcal{N}(i)$ is the set of neighbors of node i, ϵ is a trainable parameter or a fixed scalar, $\mathbf{m}_i'$ represents the aggregated message from the target node and its neighbors.

$$\mathbf{x}_i' = \mathrm{MLP}\left(\mathbf{m}_i'\right) \tag{5.9}$$

where MLP is serving as an injective update function for the aggregated message.

$$\mathbf{h}_{\mathcal{G}} = \text{CONCAT}\left(\text{READOUT}\left(\left\{\mathbf{h}_i^t \mid i \in V\right\}\right) \mid t = 0,1,\ldots,T\right) \tag{5.10}$$

where CONCAT denotes concatenation and READOUT can be a simple sum or mean of the node embeddings, h_i^t denotes the embedding for node i at iteration t, V is the node set of the graph $\mathcal{G}$, T is the number of GIN layers, i.e. the model depth.

5.3.4 Deep Adaptive Graph Neural Network

Deep Adaptive Graph Neural Network (DAGNN)[31] addresses the over-smoothing problems brought on by the deeper GNN model by introducing an adaptive adjustment mechanism.

Although neighborhood aggregation is performed via graph convolutions, one of the most significant graph operations, one layer of these neighborhood aggregation methods only takes into account immediate neighbors, which results in limited receptive fields. To obtain larger receptive fields, the depth of the graph convolution layer needs to be increased. However, performance suffers as going deeper. The cause of this performance degradation is the over-smoothing problem, in which individual nodes' embedding throughout propagation and update tends to be identical, resulting in the degradation of discriminative power.

To address this issue, DAGNN implements an adaptive adjustment mechanism, which adaptively adjusts the balance between local and global neighborhood information from each node. This mechanism is realized by introducing an adaptive weighting matrix that modulates the contribution of different graph layers to the final node embeddings. By learning this weighting matrix, DAGNN is able to adaptively incorporate information from large receptive fields, mitigating the over-smoothing problem and thus preserving the discriminative power of node embeddings. In addition, DAGNN decouples the representation transformation and propagation in the current graph convolution operation to boost the performance of deeper GNNs that can be used to achieve a larger receptive field.

Fig. 5.8 depicts the overall architecture where the DAGNN model consists of a series of graph convolutional layers, followed by an adaptive weighting layer that combines the embeddings from intermediate layers.

The formulation that describes how GIN works by combining aggregation, update

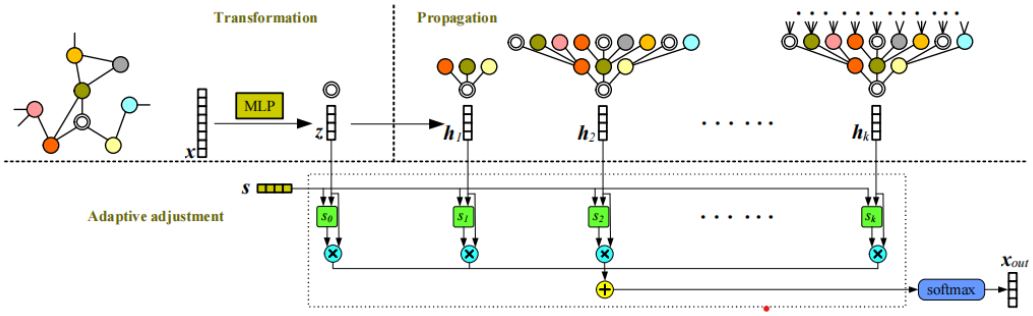

Figure 5.8: Architecture of DAGNN[31]

and readout operation is presented as Eq. (5.11):

$$\begin{aligned}
&\mathbf{Z} = \mathrm{MLP}(\mathbf{X}) && \in \mathbb{R}^{n\times c}\\
&\mathbf{H}_\ell = \widehat{A}^\ell \mathbf{Z}, \ell = 1, 2, \cdots, k && \in \mathbb{R}^{n\times c}\\
&\mathbf{H} = \mathrm{stack}\left(\mathbf{Z}, \mathbf{H_1}, \cdots, \mathbf{H_k}\right) && \in \mathbb{R}^{n\times(k+1)\times c}\\
&\mathbf{S} = \sigma(\mathbf{Hs}) && \in \mathbb{R}^{n\times(k+1)\times 1}\\
&\widetilde{\mathbf{S}} = \mathrm{reshape}(\mathbf{S}) && \in \mathbb{R}^{n\times 1\times(k+1)}\\
&\mathbf{X_{out}} = \mathrm{squeeze}(\widetilde{\mathbf{S}}\mathbf{H}) && \in \mathbb{R}^{n\times c}
\end{aligned} \tag{5.11}$$

Where n is the number of nodes, c is the expected number of node features, k is a hyperparameter that determines the depth of the model. The transformed feature matrix $\mathbf{Z} \in \mathbb{R}^{n\times c}$ is produced by applying an MLP. The symmetrical normalization propagation mechanism $\widehat{\mathbf{A}} = \widetilde{\mathbf{D}}^{-\frac{1}{2}}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-\frac{1}{2}}$ is utilized, where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ its diagonal degree matrix. $s \in \mathbb{R}^{c\times 1}$ is a trainable projection vector serving as the adaptive adjustment mechanism.

5.4 Transductive and Inductive Learning

GNNs can be categorized into two groups based on the learning method: *Transductive* and *Inductive* learning[34]. Transductive-based GNNs need to see the whole graph structure to learn each node embedding vector during the training. If there is a change in the structure of the graph, a model needs to be trained. Therefore, they are unable to generalize to unseen graphs. On the other hand, the Inductive-based GNNs learn a trainable function that aggregates the features from a node's neighborhood to generate embeddings of the nodes in the graph. Because this trainable function is shared throughout the graph, hence, the learned model can be applied to unseen graphs without going through the training again, which makes it generalizable.

In this work, we have conducted the training via Inductive learning for the GNN models.

5.5 Tasks Addressed by Graph Neural Networks

GNNs have demonstrated their effectiveness in a wide range of tasks. In this section, we discuss three renowned problems that GNNs can handle: node classification, link prediction, and graph classification.

- **Node classification** is the task of assigning labels to nodes in a graph based on their attributes and structural information. GNNs are ideally suited for this task because of their ability to learn the non-linear mapping of the node attributes based on connectivity. Fig. 5.9 shows an example of node classification task.

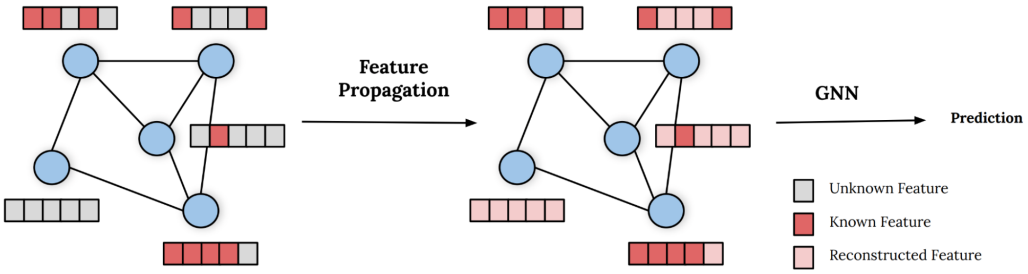

Figure 5.9: An simple example of node classification task in GNN[35]

- **Link prediction** is the task of predicting whether or not there will be linkages between nodes in a graph. It is a fundamental problem in graph analysis, with applications in a variety of fields, including recommendation systems, knowledge graph completion, and drug discovery. Since they can capture complicated interactions between nodes and their neighbors, GNNs are particularly effective for link prediction.[36].
- **Graph classification** is the task of assigning labels to entire graphs. By producing graph-level embeddings that summarize the overall structure and both node and edge features of the entire graph, GNNs can successfully handle graph classification tasks.[26].

In this work, we formulate our prediction problem as graph regression, which means that we are trying to predict continuous objective values related to the whole graph.

Chapter 6

Dataset Generation

The fundamental step in any ML problem is to have the data on which the ML model can be trained and tested.

6.1 Design Selection from HLS Benchmarks

The dataset should also include a diverse range of designs from various applications so that the trained ML model is robust enough to be generalized. For this purpose, we choose 30 designs from the widely-known real-case HLS benchmark suites, MachSuite[37], Polyhedral[38], and Rosetta[39]. The application domains of these designs encompass a wide range of areas, including Linear Algebra computations, Scientific simulations, Digital Signal Processing, Image Processing, Computer Graphics, Machine Learning algorithms, Data-mining, Stencils, and Sorting. The detailed application domain for each design is shown in Table 6.1.

Table 6.1: HLS Designs and Their Application Domains

Design	**Application Domain**
2mm	Linear algebra computations, 2 Matrix Multiplications ($D = A \cdot B$; $E = C \cdot D$)
3mm	Linear algebra computations, 3 Matrix Multiplications ($E = A \cdot B$; $F = C \cdot D$; $G = E \cdot F$)
3d-rendering	Computer graphics, rendering 2D images from 3D models (3D triangle mesh)
atax	Linear algebra computations, Matrix Transpose and Vector Multiplication
bicg	Linear algebra computations, BiCG Sub Kernel of BiCGStab Linear Solver

Continued on next page

Table 6.1 – *Continued from previous page*

Design	**Application Domain**
correlation	Digital signal processing, correlation of two signals
covariance	Digital signal processing, covariance calculation of two signals
doitgen	Linear algebra computations, Multiresolution analysis kernel (MADNESS)
fdtd_2d	Finite-difference time-domain simulation, 2-D Finite Different Time Domain Kernel
fft_strided	Digital signal processing, Image processing, Recursive formulation of the Fast Fourier Transform
gemm	General matrix multiplication, Matrix-multiply $C = \alpha \cdot A \cdot B + \beta \cdot C$
gemm_blocked	General matrix multiplication with a blocked algorithm, with better locality
gemm_ncubed	General matrix multiplication with different algorithm and computational complexity, $\mathcal{O}(n^3)$ algorithm for dense matrix multiplication
gemver	Linear algebra computations, Vector Multiplication and Matrix Addition
gesummv	Linear algebra computations, Scalar, Vector and Matrix Multiplication
jacobi_1d	Linear system solver, 1-D Jacobi stencil computation
jacobi_2d	Linear system solver, 2-D Jacobi stencil computation
lu	Linear algebra computations, LU decomposition of a matrix
md	Molecular dynamics simulation, chemistry, materials science research
merge	The mergesort algorithm, on an integer array
mvt	Linear algebra computations, Matrix-Vector Product and Transpose
optical flow	Image processing, computation of the movement of each pixel in five continuous image frames
seidel_2d	2-D Seidel stencil computation
spam_filter	Spam filtering using Naive Bayes classifier
spmv_crs	Sparse matrix-vector multiplication, using variable-length neighbor lists
spmv_ellpack	Sparse matrix-vector multiplication, using fixed-size neighbor lists

Continued on next page

Table 6.1 – *Continued from previous page*

Design	Application Domain
stencil2d	Scientific simulations, A two-dimensional stencil computation, using a 9-point square stencil
symm	Linear algebra computations, Symmetric matrix-multiply
trisolv	Linear system solver, solving triangular system of linear equations
trmm	Linear algebra computations, Triangular matrix-multiply

6.2 Data Generation using Vitis HLS and Vivado

To create multiple hardware implementations for each design, we leveraged different *HLS pragmas* i.e. synthesis directives (see Table 6.2) and various clock periods (2.5, 5, 7.5, 10 ns). This allows our predictive model to learn designs with different area-delay trade-offs. Thus, we build a dataset containing 2465 data points in total. We synthesized each design point in our dataset with *Vitis HLS* 2021.2 [11] and implemented with *Vivado* 2021.2 [40] (using *Vivado* defaults for synthesis and implementation) targeting different FPGA devices, *Avnet Ultra96v1* and *Xilinx ZCU104* boards. The dataset is generated on an Intel Xeon processor-based server with 128GB RAM.

Table 6.2: *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

The *ground truth* (actual resource usage and critical path timing) of each of these design points is extracted from the implementation and timing reports which are generated after the place and route phase. Table 6.3 shows the range of values of target objectives in our dataset.

6.3 Graph Generation for HLS Design

The input to our proposed GNN predictive model is an HLS IR graph representing the functionality of the design and is extracted after the front-end compilation. As

Table 6.3: Overall *Summary* of designs in our DATASET

	# of **LUTs**	# of **DSPs**	# of **FFs**	**C.P.** (ns)
Min.	8	0	24	1.5
Max.	53239	360	31004	8.562
Mean	2456	28	2619	3.72

mentioned in Section 3.4, we have considered a combination of control-flow, data-flow and call-flow graphs for our experiments. A program semantic information representation tool, ProGraML[41] is used to extract a graph from a given IR which combines the aforementioned graphs into a unique graph. It assigns separate nodes to operands explicitly and also keeps the function hierarchies by incorporating the call flow.

During the processing of ProGraML, LLVM instructions are converted into nodes of the generated graph with specific attributes like opcode. Referencing LLVM Language Reference Manual[42], we extended ProGraML capabilities to retrieve more critical information from the given LLVM instructions and append it as attributes of the corresponding nodes in the graph. Specifically, we added two key pieces of information for each LLVM instruction as follows:

- **Bitwidth** describes the number of bits in the operands that the instruction operates on, which affects the resources required to store and manipulate the operands and is thus highly relevant to the consumption of the hardware resources we are trying to predict. For example, an instruction that operates on 8-bit operands will require fewer hardware resources than an instruction that operates on 32-bit operands.

- **Opcode category** identifies the functionality of LLVM instruction. In general, the resource requirements and behaviors of various LLVM instruction categories vary, which makes it affects the overall performance and resource utilization. For example, the "Arithmetic" category contains instructions that perform arithmetic operations, such as adding or multiplying. These instructions typically consume more hardware resources than others.

Algorithm 1 illustrates the process of generating a graph representation of an HLS design. The input to the algorithm is the HLS design, and the output is the graph representation of the design.

Fig. 6.1 shows a toy example demonstrating how the input graph to our model is generated. For ease of demonstration purposes, only the apt nodes in the graph are presented.

Fig. 6.1 (a) shows the HLS code for implementing the dot product with two

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 ← Extended ProGraML(LLVM IR)

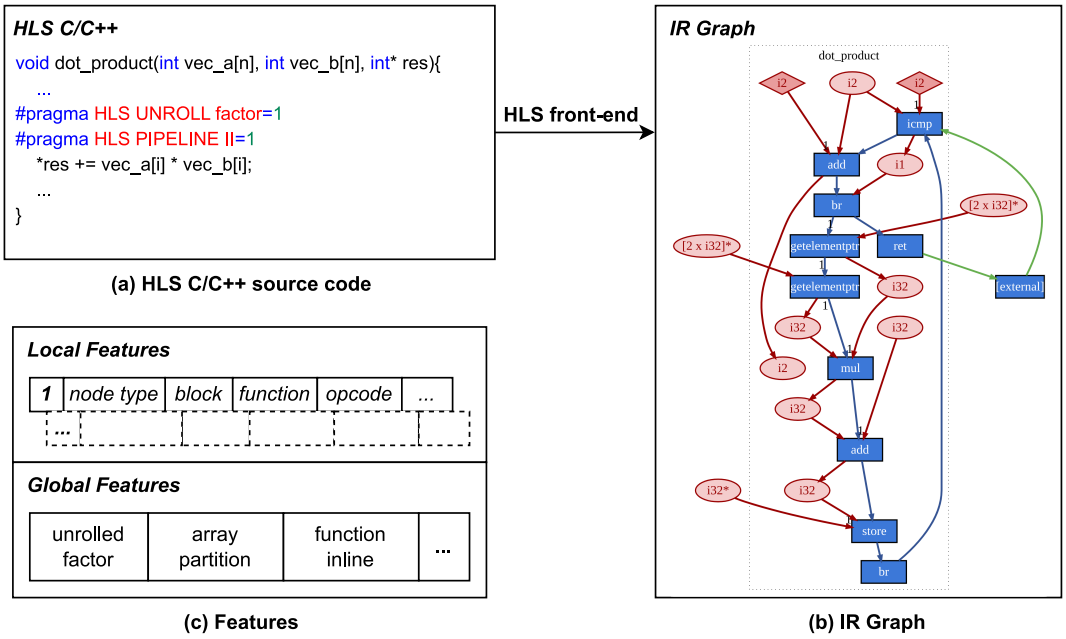

Figure 6.1: An HLS design example with its graph representation

HLS pragmas inside the loop. Fig. 6.1 (b) illustrate its graph representation which is extracted after the HLS front-end compilation. The graph contains two different kinds of nodes. The LLVM instructions are represented by the nodes in blue, which are connected to one another according to the control flow. The variables and constant values that show the operands of the instructions 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. 6.1 (c) shows the local and global features that can be directly extracted from the IR graph and user-defined optimization directives (See Section 6.4 for detail).

6.4 Features

Our proposed approach uses two different sets of features which are useful for predicting post-implementation QoR. These sets of features are extracted from two

distinct sources which are HLS IR code and user-defined HLS synthesis directives. We call these sets local and global features respectively.

Local features include structural and contextual information. Structural information tells about the connectivity of the nodes in the graph and is encoded as an adjacency matrix. Contextual information relates to the node and edge attributes and this information is explicitly encoded for each node and edge. For each node, its *type*, *category*, *opcode*, *block ID*, *function ID* and *bitwidth* are considered. For example, a *type* shows whether the node is an instruction, variable, or constant. As for an edge, we only considered its *type* which basically tells whether the edge belongs to control, data, or call flow. Details of the local features are listed in Table 6.4.

Table 6.4: *Local Features*: Nodes and Edges

	Feature	**Description**	**Examples**
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

In order to represent categorical features numerically (see Section 4.5), we employ both one-hot encoding and embedding techniques. To be more precise, we represent the node feature vector using a combination of one-hot encoding for the node type and embedding for the remaining features. The resulting node feature vector is constructed by concatenating the one-hot encoding of the node type with the embedding of the remaining features. As the node type is the most influential feature and we want to explicitly encode and convey this information to the model, we utilize one-hot encoding for this particular feature.

6.5 Normalization

Before the training, we pre-process the data and apply normalization so that each objective can contribute equally to the training loss. For example, in the case of resource usage, we normalize the resource utilization by dividing them by the total number of resources available on the FPGA.

Chapter 7

Predictive Model

We formulate the QoR predictive problem as a multi-objective regression task to estimate post-implementation *timing* and *resource usages* for LUT, FF, and DSP of a given design based on its HLS IR without invoking HLS tool.

7.1 Model structure

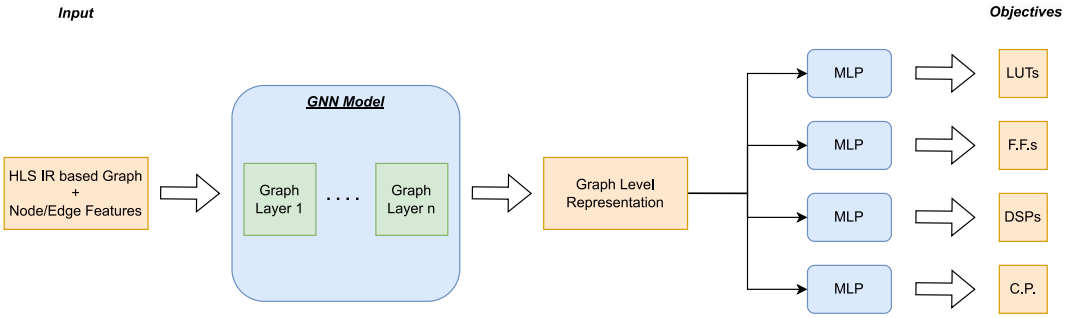

Figure 7.1: General Structure of the framework to evaluate different GNN models. The only difference is the type of GNN models.

We employ multi-task learning where a single GNN model is trained and the generated graph embeddings are shared with a set of MLPs to estimate the different objectives. The general structure of our model architecture is shown in Fig. 7.1. It takes the graph representation of the design as an input and creates the initial node/edge embeddings by encoding the attributes (Section Table 6.4). This information is then passed to the GNN model which updates the embeddings. These updated embeddings are used to generate a graph-level representation vector via pooling which is passed to a set of MLPs to predict multiple objectives.

We evaluated different GNN models (Section 5.3) using the same flow for a fair comparison and the only difference lies in the type of a GNN layer.

7.2 Training

Fig. 7.2 shows the *training flow* of the proposed framework. Once the dataset and the associated features are available, we can then perform the training to obtain a predictive model. The ground truth labels are extracted from the post-implementation reports. We train the GNN models by supervised learning to learn the underlying heuristics and optimization techniques to predict the desired objectives swiftly and accurately. During the training phase, an HLS design with its configurations is first fed into the HLS tool front-end where the IR is generated. The graph generator (see Section 6.3 for detail) will then convert the IR into a graph. The generated graph is passed to the GNN model and the final graph-level embedding is obtained, which is subsequently concatenated with the global features and passed to the non-graphical regression model for predicting the post-implementation QoR.

7.3 Inference

The *inference flow* of the proposed framework is shown in Fig. 7.3. The main purpose of the inference phase is to achieve fast and accurate QoR prediction of the design compared to HLS baseline without going through implementation processes which are time-consuming and resource intensive. In the inference phase, we apply the same pre-processing workflow for any new design to generate a graph and extract features respectively. Afterward, these are fed to the well-trained model to perform the prediction of the objectives. We perform two types of cross-validations to get a better assessment of the effectiveness of the different models. For the model selection, we first set aside a 20 % of the dataset, also called the hold-out set, randomly. This hold-out remains isolated and is only used at the end to assess the final performance and to select the best-performing models. Then, we executed the k-fold cross-validation on the remaining 80 % dataset where the hyper-parameters of the considered models are optimized and tuned. Furthermore, for the evaluation and effectiveness of the selected models, we also performed the generic k-fold cross-validation on the whole dataset as well. It is worth mentioning that our predictive models are able to finish the inference for objective estimation within milliseconds as opposed to the implementation phases which generally require minutes to hours.

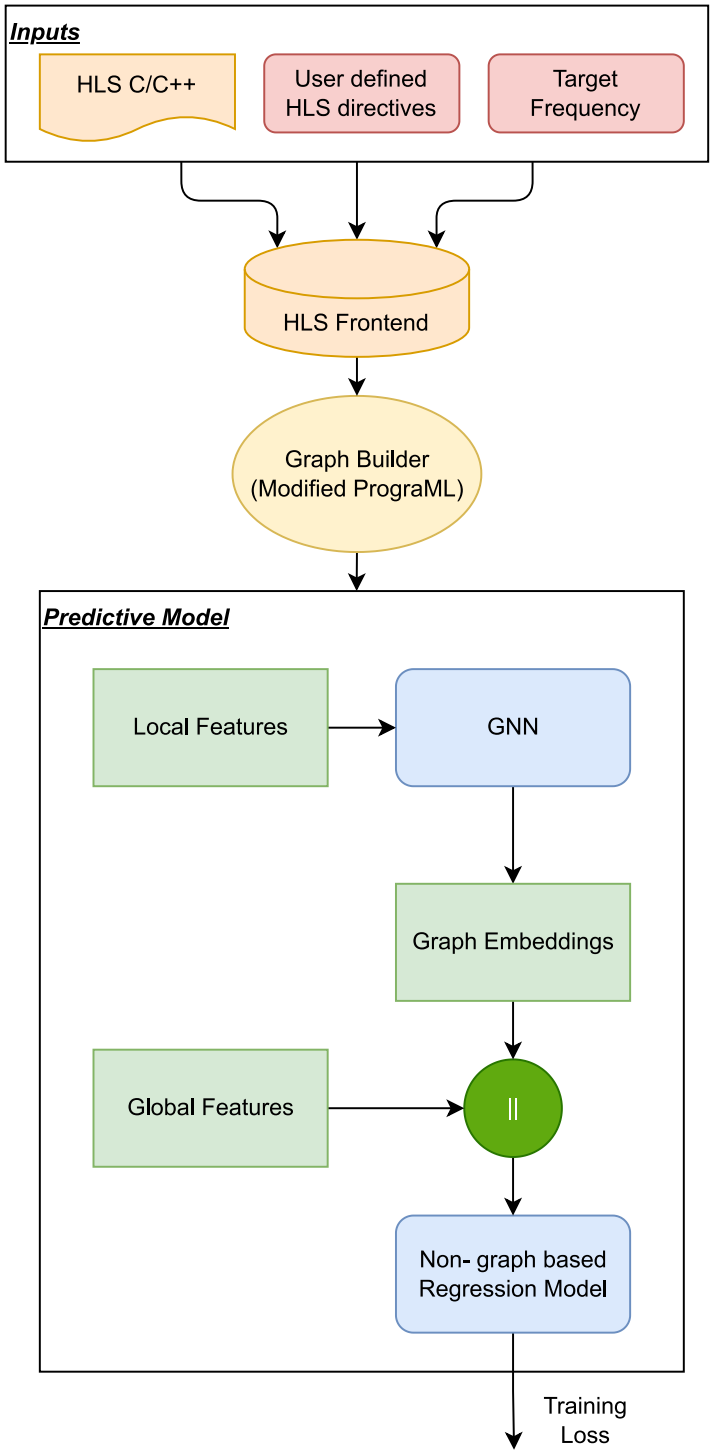

Figure 7.2: Training phase of the proposed framework

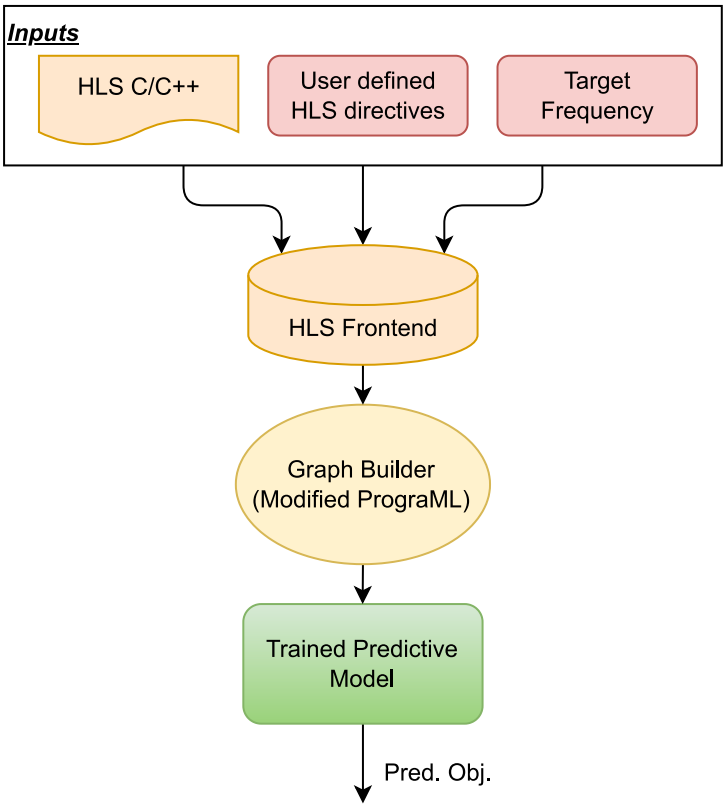

Figure 7.3: Inference phase of the proposed framework

Chapter 8

Experimental Results

We investigated numerous configurations using trial and error to identify the optimal setup, and then conducted an extensive experiment to thoroughly evaluate the effectiveness of our predictive models.

8.1 Setup

All the mentioned GNN models in Section 5.3 are implemented and trained using PyTorch Geometric[20] on a Linux host machine equipped with Nvidia GeForce RTX 3060 laptop GPU. For our experiments, each model is set to have 2 layers with an initial embedding size of 300 followed by 3 MLP layers for each objective. The graph-level representation is derived by using mean pooling. We trained all the models with Adam optimizer[43] for 100 epochs using a learning rate of 0.001, a weight decay of 0.0005, and an Exponential Linear Unit (ELU)[44] as an activation function. All these hyper-parameters are tuned on the validation set. As the prediction problem is formulated as a regression task, we use Root Mean Squared Error (RMSE) as a metric for the evaluation of the models. We perform 5-fold cross-validation to check the efficacy and robustness of the models and to select the best-performing model.

The designs in the dataset (Section 6.1) are synthesized and implemented using *Vitis HLS* 2021.2 [11] and *Vivado* 2021.2 [40] respectively. The ground truth labels (LUT, FF, DSP, CP) are extracted from the post-implementation reports. The dataset is randomly split into 70 % for training, 10 % for validation and 20 % for testing. We also compare our models with the commercial HLS tool (*Vitis HLS*), used as a *Baseline Model* and a graph learning-based performance prediction model [8].

As illustrated in Algorithm 2, we utilize the HLS Estimation Report as the baseline and the Post-Implementation Report as the ground-truth label. The HLS

Algorithm 2 *Baseline* and *Groundtruth* Extractor

Require: HLS design
RTL design, HLS report ← Vitis HLS(HLS design)
Bitstream, Post-implementation report ← Vivado(RTL design)
Baseline ← HLS baseline Extractor(HLS report)
Groundtruth ← Groundtruth Extractor(Post-implementation report)

Estimation Report is extracted to establish a baseline for our proposed approach. The Post-Implementation Report provides performance metrics after the RTL design has been implemented on a target FPGA device. These metrics are used to generate the ground-truth labels for training and evaluating our proposed GNN-based predictive models.

8.2 Model Evaluation

Based on the discussion in Section 4.3, we utilize both the single test and cross-validation approaches to evaluate the effectiveness of our proposed predictive model.

8.2.1 Single test

GNN models encode the local features into graph-level embeddings which are then passed to the non-graphical regression model. We first test the performance of the GNN models with local features. Fig. 8.1 shows the performance evaluation of GNN models with respect to the *Baseline* regarding LUT, FF, DSP and CP.

Please note that all the models performed better than the baseline for all the prediction objectives. Fig. 8.2 shows the prediction improvements of the models over the baseline. For LUT utilization prediction, GAT provide the best improvement of more than 55 % over the baseline. In the case of FF, GCN, GAT, and DAGNN give prediction improvements of more than 40 %. GAT is the best among all the models in reducing the prediction error with respect to the HLS baseline model for DSP utilization. For critical path timing (CP), all the models improve the prediction by more than 45 %. On average, GAT is the best-performing model in reducing the resource usage prediction, and DAGNN in terms of timing.

It is worth mentioning that all these GNN based predictive models perform better prediction than the HLS baseline model for all the targeted objectives. GIN provides the least advantage in terms of reducing the prediction error among all the tested models, especially in the case of DSP utilization prediction. On the basis of this empirical evidence, we drop GIN for the model selection phase.

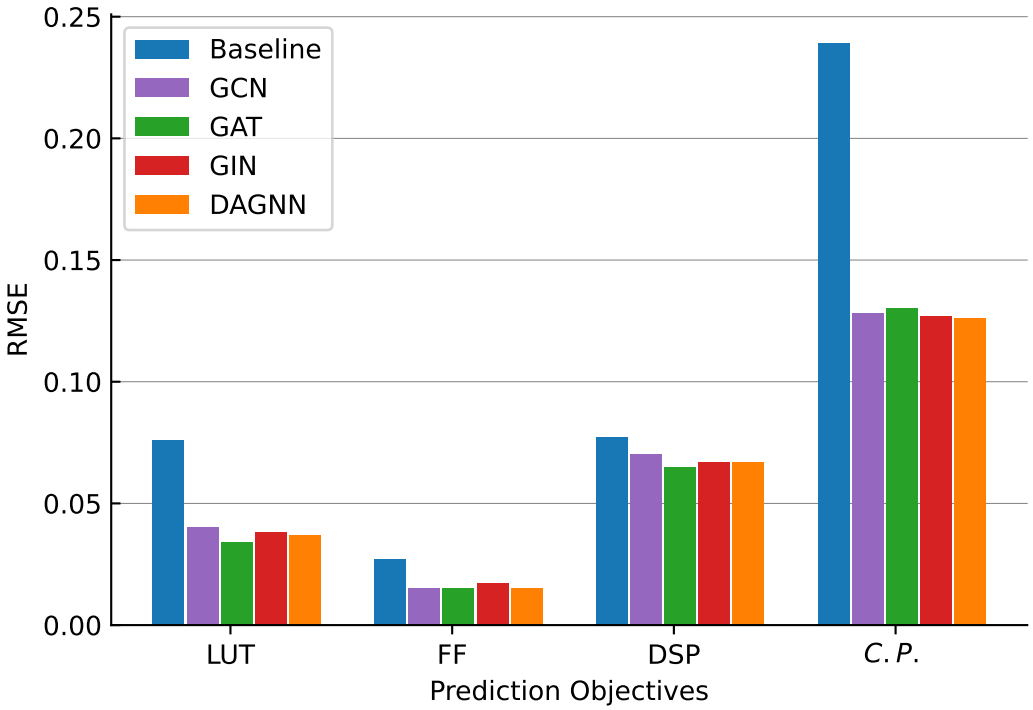

Figure 8.1: RMSE with Local Features only

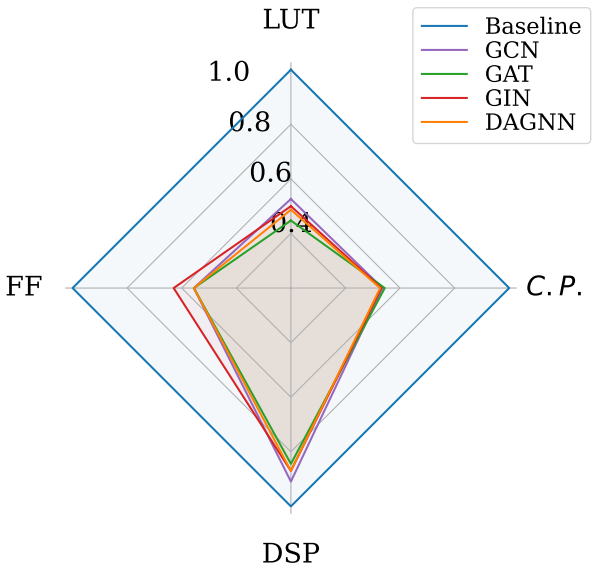

Figure 8.2: QoR Improvements with Local Features only

8.2.2 Cross-validation

We further evaluated the predictive models using a 5-fold cross-validation to obtain a more comprehensive evaluation of their performance after performing the single test. This approach allows us to train and test the models on different subsets of the dataset, reducing the risk of overfitting and providing a more accurate estimate of their predictive power. As previously discussed in Section 4.3, there are two distinct methods for conducting cross-validation: one involves applying the technique to the whole dataset, while the other involves applying it only to the training set. Both methods were employed, and the results are presented below.

- Cross-validation is exclusively applied to the training set.

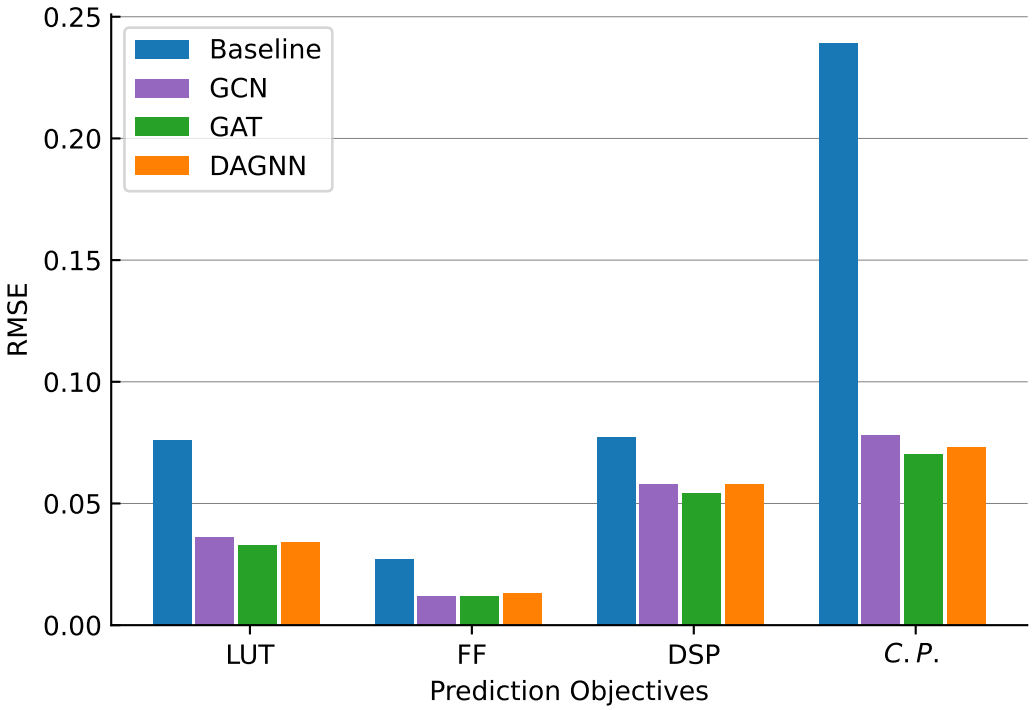

Figure 8.3: RMSE of 5-fold Cross Validation on the training set

For the *model selection* purpose, we apply 5-fold cross-validation with a holdout test set. Fig. 8.3 compares the performance evaluation of GCN, GAT and DAGNN models with respect to the HLS *Baseline*. Fig. 8.4 shows the Quality of Results (QoR) improvements of the selected model over the baseline for the target objectives. For the LUT and FF utilization predictions, all the selected graph-based models give improvements of more than 50 %; GAT based model provides the best 57 % for LUT and GCN based model provides the best 55 %

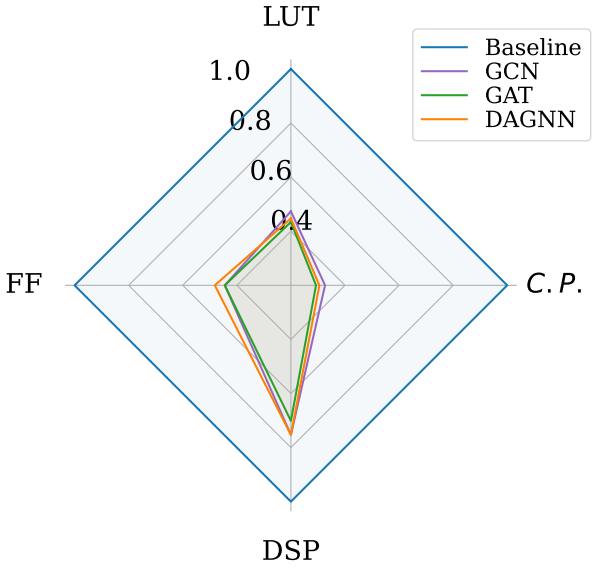

Figure 8.4: QoR Improvements of 5-fold Cross Validation on the training set

for FF with respect to the baseline, respectively. In the case of DSP prediction, GAT is clearly the winner and reduces the prediction error by more than 30 %. For critical path timing (CP), all the graph-based models provide the prediction improvement by more than 67 %, with GAT based predictive model being the best (71 %). GAT outperforms other graph-based models in three out of four target objectives and is not far behind in the prediction of FF utilization in which GCN based model performs the best. *On the basis of these results, we select GAT based model.*

- Cross-validation is applied to the whole dataset.

  To evaluate the selected model over the whole dataset, we perform a generic 5-fold cross-validation. Fig. 8.5 shows the Quality of Results prediction improvements over the whole data. It can be seen that the GAT based predictive model clearly provides better prediction with respect to the HLS baseline model, and gives up to 74 % performance prediction improvements.

8.3 Generality and Comparison with other work

To evaluate the generality of our chosen model, we conduct an evaluation on unseen data points from new applications. We randomly selected data points from our dataset and ensured that the training and testing sets are from distinct applications. By doing so, we can assess the capability of our model to generalize effectively to

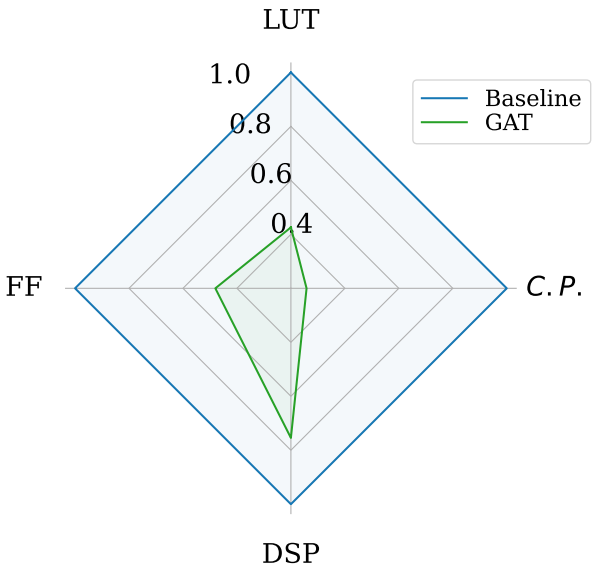

Figure 8.5: QoR Improvements of 5-fold Cross Validation on the whole dataset

unseen applications. Moreover, we conduct a quantitative comparative analysis with the state-of-the-art using the same dataset split method. Note that all these kernels are from diverse application domains and have different coding structures. We compare our model quantitatively with the graph-based Machine Learning approach in [8], as they open-source their work. For the comparison with other ML based approaches, Table 1.1 provides a qualitative analysis.

We take the best-performing regression model in [8] for resource usage and timing prediction, implement and train to the best of our knowledge, and refer to it as *PNA-HLS*. Fig. 8.6 shows that our GAT based predictive model reduces the prediction error for resource usage and timing prediction by 68 % and 34 % with respect to the HLS, respectively. Our proposed model also outperforms the state-of-the-art model *PNA-HLS* by 28 % and 22 % for resource usage and timing prediction respectively. Consequently, these findings substantiate the assertion that the proposed model possesses the capacity to generalize to unexplored domain applications, thereby achieving a performance that is superior to the current state-of-the-art. This shows the importance of encoding the design configurations (HLS synthesis directives) as features in the models.

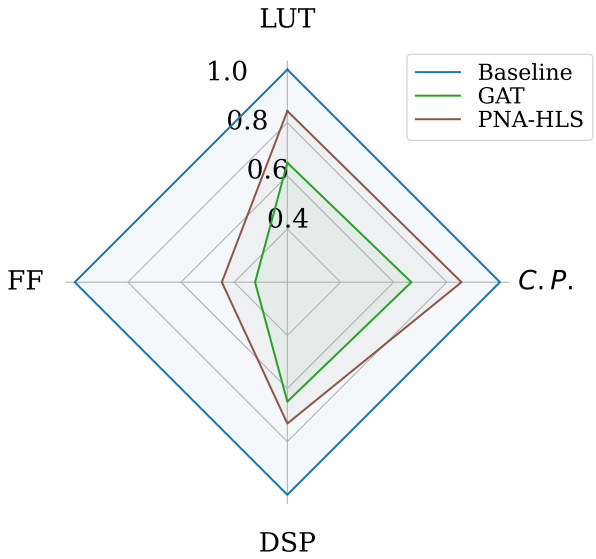

Figure 8.6: QoR Improvements on Unseen application domain designs

Chapter 9

Conclusion

9.1 Achievement

An High-Level Synthesis (HLS) offers great flexibility to optimize designs for area and performance but HLS estimated QoR often differ from actual post-implementation results achieved. In this thesis work, we proposed an HLS tool-agnostic GNN-based framework to estimate Quality of Results of HLS designs. We first developed a method to extract a graph-based representation of design straight from the HLS front-end, which encodes both program semantics and HLS synthesis directives information. Then, we proposed a multi-objective GNN-based learning model to predict resource usage and timing of HLS designs in milliseconds without invoking HLS tool synthesis process. The experimental results demonstrate that our proposed predictive model outperforms the commercial HLS tool for real-case applications from various domains. It also shows that our model is capable of extending the learned knowledge and generalizing it to unseen design cases.

9.2 Feature Work

We plan to extend our framework to cover larger designs and cover more application domains in future work.

Bibliography

[1] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D. Hämäläinen. «Are We There Yet? A Study on the State of High-Level Synthesis». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.5 (2019), pp. 898–911. DOI: 10.1109/TCAD.2018.2834439 (cit. on p. 1).

[2] Guyue Huang et al. «Machine learning for electronic design automation: A survey». In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 26.5 (2021), pp. 1–46 (cit. on p. 2).

[3] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline F.Y. Young, and Zhiru Zhang. «Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning». In: *Int'l Symp. on Field-Programmable Custom Computing Machines (FCCM)* (May 2018) (cit. on pp. 2, 3).

[4] Hosein Mohammadi Makrani, Farnoud Farahmand, Hossein Sayadi, Sara Bondi, Sai Manoj Pudukotai Dinakarrao, Houman Homayoun, and Setareh Rafatirad. «Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design». In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2019, pp. 397–403 (cit. on pp. 2, 3).

[5] Nan Wu, Yuan Xie, and Cong Hao. «Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning». In: *Proceedings of the 2021 on Great Lakes Symposium on VLSI*. 2021, pp. 39–44 (cit. on pp. 2, 3).

[6] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. «Accurate operation delay prediction for FPGA HLS using graph neural networks». In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020, pp. 1–9 (cit. on pp. 2, 3).

[7] Sayandip De, Muhammad Shafique, and Henk Corporaal. «Delay Prediction for ASIC HLS: Comparing Graph-based and Non-Graph-based Learning Models». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022) (cit. on p. 3).

[8] Nan Wu, Hang Yang, Yuan Xie, Pan Li, and Cong Hao. «High-level synthesis performance prediction using gnns: Benchmarking, modeling, and advancing». In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 2022, pp. 49–54 (cit. on pp. 3, 42, 47).

[9] Philippe Coussy. *High-level synthesis: from algorithm to digital circuit*. Springer International Publishing, 2018 (cit. on p. 6).

[10] Daniel Payne. *High-level synthesis and open source software algorithms*. July 2020. URL: https://semiwiki.com/semiconductor-services/284484-high-level-synthesis-and-open-source-software-algorithms/ (cit. on p. 7).

[11] Xilinx Inc. *Vitis High-Level Synthesis User Guide*. 2022. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_2/ug1399-vitis-hls.pdf (cit. on pp. 8, 9, 11, 34, 42).

[12] *The LLVM Compiler Infrastructure Project*. https://llvm.org/. Retrieved March 11, 2016 (cit. on p. 12).

[13] Erick Lumunge. *LLVM - an overview*. Apr. 2022. URL: https://iq.opengenus.org/llvm-overview/ (cit. on p. 13).

[14] Chris Lattner and Vikram Adve. «LLVM: A compilation framework for lifelong program analysis & transformation». In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86 (cit. on p. 15).

[15] *GCC, the GNU Compiler Collection*. 2022. URL: https://gcc.gnu.org/ (cit. on p. 15).

[16] Xilinx Inc. *High-Level Synthesis (HLS) Compiler for Xilinx FPGAs*. https://github.com/Xilinx/HLS. Accessed on March 22, 2023 (cit. on p. 15).

[17] Ethem Alpaydin. *Introduction to Machine Learning*. MIT Press, 2010 (cit. on p. 17).

[18] David E. Caughlin. *Chapter 48 applying K-fold cross-validation to logistic regression: R for HR: An introduction to human resource analytics using R*. URL: https://rforhr.com/kfold.html (cit. on p. 19).

[19] Adam Paszke et al. «PyTorch: An Imperative Style, High-Performance Deep Learning Library». In: *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*. 2019 (cit. on p. 20).

[20] Matthias Fey and Jan Eric Lenssen. «Fast Graph Representation Learning with PyTorch Geometric». In: *CoRR* abs/1903.02428 (2019). arXiv: 1903.02428. URL: http://arxiv.org/abs/1903.02428 (cit. on pp. 20, 42).

[21] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. «A Neural Probabilistic Language Model». In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435 (cit. on p. 20).

[22] Tomás Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. «Distributed Representations of Words and Phrases and their Compositionality». In: *CoRR* abs/1310.4546 (2013). arXiv: 1310.4546. URL: http://arxiv.org/abs/1310.4546 (cit. on p. 21).

[23] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. «A comprehensive survey on graph neural networks». In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24 (cit. on p. 22).

[24] Rajinikanth. *Data Structures*. URL: http://www.btechsmartclass.com/data_structures/introduction-to-graphs.html (cit. on pp. 22, 23).

[25] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. «Graph neural networks: A review of methods and applications». In: *AI open* 1 (2020), pp. 57–81 (cit. on pp. 24, 25).

[26] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. «Neural message passing for quantum chemistry». In: *International conference on machine learning*. 2017, pp. 1263–1272 (cit. on pp. 24, 31).

[27] *Graph neural networks for fraud detection*. URL: https://www.dataminingapps.com/2023/02/graph-neural-networks-for-fraud-detection/ (cit. on p. 25).

[28] Thomas N. Kipf and Max Welling. «Semi-Supervised Classification with Graph Convolutional Networks». In: *International Conference on Learning Representations (ICLR)*. 2017 (cit. on p. 25).

[29] Shaked Brody, Uri Alon, and Eran Yahav. «How Attentive are Graph Attention Networks?» In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=F72ximsx7C1 (cit. on pp. 25, 26).

[30] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. «How Powerful are Graph Neural Networks?» In: *International Conference on Learning Representations*. 2019 (cit. on pp. 25, 27).

[31] Meng Liu, Hongyang Gao, and Shuiwang Ji. «Towards deeper graph neural networks». In: *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2020, pp. 338–348 (cit. on pp. 25, 29, 30).

[32] *How powerful are graph convolutional networks?* URL: https://tkipf.github.io/graph-convolutional-networks/ (cit. on p. 26).

[33] *Gin: How to design the most powerful graph neural network*. URL: https://mlabonne.github.io/blog/gin/ (cit. on p. 28).

[34] Will Hamilton, Zhitao Ying, and Jure Leskovec. «Inductive representation learning on large graphs». In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 30).

[35] *Graph machine learning with missing node features*. URL: https://blog.twitter.com/engineering/en_us/topics/insights/2022/graph-machine-learning-with-missing-node-features (cit. on p. 31).

[36] Muhan Zhang and Yixin Chen. «Link prediction based on graph neural networks». In: *Advances in Neural Information Processing Systems* (2018), pp. 5165–5175 (cit. on p. 31).

[37] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. «Machsuite: Benchmarks for accelerator design and customized architectures». In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2014, pp. 110–119 (cit. on p. 32).

[38] Tomofumi Yuki Louis-Noël Pouchet Uday Bondugula. *PolyBench/C*. URL: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ (cit. on p. 32).

[39] Yuan Zhou et al. «Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs». In: *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)* (Feb. 2018) (cit. on p. 32).

[40] Xilinx Inc. *Vivado Design Suite User Guide*. 2022. URL: https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_2/ug973-vivado-release-notes-install-license.pdf (cit. on pp. 34, 42).

[41] Chris Cummins, Zacharias Fisches, Tal Ben-Nun, Torsten Hoefler, Michael O'Boyle, and Hugh Leather. «ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations». In: *Thirty-eighth International Conference on Machine Learning (ICML)*. 2021 (cit. on p. 35).

[42] *LLVM Language Reference Manual*. 2022. URL: https://llvm.org/docs/LangRef.html (cit. on p. 35).

[43] Diederik P Kingma and Jimmy Ba. «Adam: A method for stochastic optimization». In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 42).

[44] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. «Fast and accurate deep network learning by exponential linear units (elus)». In: *arXiv preprint arXiv:1511.07289* (2015) (cit. on p. 42).