Copyright
by
Zhigang Wei
2025

The Dissertation Committee for Zhigang Wei
certifies that this is the approved version of the following dissertation:

Power Estimation for FPGAs with Machine Learning Techniques

Committee:

Lizy K John, Supervisor

Earl E Swartzlander

Andreas Gerstlauer

Zhangyang Wang

Shuaiwen Song

Power Estimation for FPGAs with Machine Learning Techniques

by

Zhigang Wei

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2025

Dedication

Dedicated to:

my father Kangwen Wei,

my mother Laying Yu,

and my sister Yuting Wei

Acknowledgments

I would like to express my gratitude to my supervisor, Professor Lizy K John. Her invaluable advice guide me through my entire exploration of my research. Her intellectual rigor and insightful feedback have profoundly shaped my research and she taught me how to present my work to others more effectively. I appreciate her constant patience, support and encouragement when I was having great difficulties during the COVID time. I am immensely thankful for her mentorship and confident that the lessons learned under her guidance will continue to influence my career and personal endeavors.

I would like to thank Professor Earl E Swartzlander, Professor Andreas Gerstlauer, Professor Zhangyang Wang and Professor Shuaiwen Song to serve as my Ph.D. committee. Their feedback on my progress review helped me to improve the dissertation quality. Their constructive advice and comments have played a crucial role in the refinement of my work.

I feel so fortunate to collaborate with Aman Arora, who has led me a lot on my start of my research and gave me plent of thoughtful advice on work. I still remembered the first paper we work together till night. The dissertation cannot be complete with his collaboration. Furthermore, I want to express my deep gratitude to Emily Shriver for her suggestions and reviews on my research paper. I learned a lot about paper writing during the discussion among us.

I was blessed to meet all the lovely students at UT. The company of LCA members help me through my Ph.D. life. While there are many names to mention, I would like to express my sincere appreciation to Qinzhe Wu, Ruihao Li, Siyuan Ma, and Shuang Song for their invaluable discussions. Their perspectives and insights have not only enhanced my work but also made my life at UT more engaging and enjoyable. I am deeply thankful to Bagus Hanindhito for his help on GPU management and Allison Seigler for her availability helping on my reserarch work. I would also like

to sincerely appreciate Zachary Susskind, Mugdha Jadhao, Shashank Nag, Alan T L Bacellar, Shagnik Pal for providing feedback in my presentations at LCA weekly meetings. I want to extend my gratitude to Shaohui Liu, Pragnesh Patel for the discussion on my research work.

Lastly but most importantly, I am forever grateful to my dear family: my father Kangwen Wei, my mother Laying Yu and my younger sister Yuting Wei. I am fortunate to have them in my life. Their unwavering mental and financial support, continuous love help me to complete the lengthy Ph.D. program. Their consistent encouragement helped me out of the low ebb. I would never have made it without them.

Abstract

Power Estimation for FPGAs with Machine Learning Techniques

Zhigang Wei, PhD
The University of Texas at Austin, 2025

SUPERVISOR: Lizy K John

Field-programmable gate arrays (FPGAs) are becoming increasingly popular across various domains, including data centers and embedded systems, where they function as reconfigurable hardware accelerators to enhance the performance of computation-intensive tasks. With the growing complexity of architectures, increased integration density, and larger chip sizes in modern FPGAs, power efficiency has emerged as a critical design constraint, making power consumption a primary concern. To enable accurate and fast estimation of design-time power, prior work employed Machine Learning (ML) techniques to bypass the time-consuming phases including synthesis, implementation, and RTL simulation phases.

However, as FPGA development becomes faster and more types of FPGAs come in scenes, these ML-based power models, which can only be used to predict the power for one FPGA, no longer meet the requirement of power evaluation on varieties of FPGAs. These ML-based power models methodology suffers some inefficiency for a new model construction or retraining for a new FPGA due to the need on large volume of data. While there are no existing sufficient dataset for the relevant topic, new dataset preparation is time-consuming because collecting the ground-truth metrics and features for a single design needs the whole phases including High-Level-Synthesis

(HLS), synthesis and implementation, and simulation. To collect the metrics for sufficient number of designs on one FPGA, therefore, takes a great amount of time from days to weeks. This dissertation explores novel techniques or methods to improve efficiency in generating an accurate ML-based model for the estimation of design power.

The first contribution of this dissertation is an open-source dataset called *HLSDataset*. It is a well-curated open-source dataset for ML-assisted FPGA design using HLS. Currently, it contains more than 40,000 HLS designs. The dataset can be used in not only design-time power prediction but also performance prediction and resource estimation etc. The data set also contains the methodology and source code to efficiently automate the data set generation.

The second contribution of this dissertation is a cross-FPGA power predictor called *XPNet*. Prior works have attempted predicting one FPGA's power from training a model on itself, while no prior work has presented a model to perform cross-FPGA power prediction. *XPNet* is ML-based power model constructed with Transfer-Learning technique. *XPNet* constructs a GNN-based model that can be efficiently transferred between FPGAs. Within AMD/Xilinx device family, *XPNet* is capable of generating a model that produces 8.4% average error with only 20 designs on a new FPGA. *XPNet* can also adapt a model trained on Xilinx device to Intel device with only 20 designs and the adapted model maintains 10.34% average error.

The last contribution of this dissertation is an architecture aware power predictor called *ATAPP*. *ATAPP* constructs GNN models based on designs on seen FPGAs. Unlike any other existing ML-based power model, *ATAPP* make prediction based on design representation and FPGA architecture representation. The design representation is a graph generated with intermediate representation (IR) codes, Finite State Machine Datapath (FSMD) model and toggling behavior on the operator. The architecture representation is generated using architecture model from RapidWright and positional encoding techniques. *ATAPP* shows the ability to produce accurate

prediction (13.09% average error) for unseen designs on unseen FPGAs.

Table of Contents

List of Tables 13
List of Figures 15
Chapter 1: Introduction 17
1.1 Field-Programmable Gate Array 17
1.2 High-Level-Synthesis (HLS) 18
1.3 FPGA Power 19
1.4 Problem Description and Motivation 22
1.5 Contributions of Dissertation 25
1.6 Thesis Statement 26
1.7 Dissertation Organization 26
Chapter 2: Background and Related Work 27
2.1 FPGA Power Consumption 27
2.2 Dataset for HLS 27
2.3 Machine Learning for HLS designs and power estimation 29
Chapter 3: Methodology 32
3.1 High Level Synthesis 32
3.2 Activity Model Generation 34
3.3 Tools 35
Chapter 4: HLSDataset: Open-Source Dataset for ML-Assisted FPGA Design using High Level Synthesis 36
4.1 HLSDataset Construction 39
4.1.1 C source code manipulation 40
4.1.2 Auto-generation of Tcl scripts 40
4.1.3 Data collection 41
4.2 Properties of HLSDataset 43
4.2.1 The contents of HLSDataset 43
4.2.2 Statistical overview of HLSDataset 45
4.3 HLSDataset Applications 48
4.4 Case Studies 49
4.4.1 Case Study 1: Power Estimation in FPGA HLS via GNNs . . . 49
4.4.2 Case Study 2: Estimation of Quality of Results in HLS with ML 51
4.5 Summary 53

Chapter 5: XPNet: Cross-FPGA Power Prediction from High Level Language Code . . . 54
5.1 Problem Formulation . . . 56
5.2 The XPNet Framework . . . 58
5.2.1 Train and Fine-Tune . . . 59
5.2.1.1 Label Generator: . . . 59
5.2.1.2 Feature Generator: . . . 59
5.2.1.3 Design Selector: . . . 61
5.2.2 Inference . . . 63
5.2.3 Model . . . 63
5.3 Experimental Setup . . . 65
5.3.1 Devices . . . 65
5.3.2 Dataset . . . 65
5.4 Results . . . 68
5.4.1 How effective is Transfer-Learning in XPNet? . . . 68
5.4.2 How effective is XPNet in predicting across devices? . . . 69
5.4.3 How much can the Design Selector help? . . . 71
5.4.4 Comparison on different ML Models on XPNet . . . 72
5.4.5 How does XPNet perform in comparison to other power models? 75
5.5 Summary . . . 77
Chapter 6: ATAPP: Architecture and Technology Aware Power Predictor for Unseen FPGAs . . . 78
6.1 Problem Formulation . . . 82
6.1.1 Problem 1: Generate the FPGA architecture representation . . . 82
6.1.2 Problem 2: Generate the design (circuit) representation with switching activity . . . 83
6.1.3 Problem 3: Build the prediction model with both the design (circuit) and the unseen FPGA architecture . . . 83
6.2 ATAPP details . . . 84
6.2.1 Architecture representation . . . 84
6.2.1.1 FSR extraction . . . 86
6.2.1.2 Extract tile map . . . 88
6.2.1.3 Encode each tile type . . . 89
6.2.1.4 Dimensionality reduction . . . 90
6.2.1.5 Positional Encoding . . . 90
6.2.2 Design representation . . . 91

6.2.3 Predictive Model 92
6.2.3.1 GNN encoder 93
6.2.3.2 MLP encoder and MLP decoder 94
6.3 Experiment Setup 95
6.4 Baseline Solutions 96
6.5 Results 98
6.5.1 Model Evaluation 98
6.5.2 Robustness study on clock period 101
6.5.3 Robustness study on generations of FPGAs 102
6.6 Summary 103
Chapter 7: Conclusion 105
7.1 Summary 105
7.2 Future Work 106
Works Cited 109

List of Tables

2.1 Comparing HLSDataset with prior open-source datasets for training ML models for chip design (general) 27

2.2 Comparing HLSDataset with prior open-source datasets for training ML models for chip design (Use case in ML) 28

4.1 General overview of HLSDataset 39

4.2 Descriptions of features included in the CSV files provided with HLS-Dataset 44

4.3 Prior ML-based prediction via HLS work, the used ML model, prediction tasks, the used dataset for training and the availability of the dataset. 47

4.4 **Dynamic power estimation errors** - Training dataset and testing dataset are from Polybench subset of HLSDataset. Results for ZU9EG and XC7V585T. 50

4.5 **Resource estimation errors** - Training dataset and testing dataset are from Machsuite and Polybench subsets of HLSDataset. Results for ZU9EG. 52

5.1 The features used by prior power models vs XPNet 54

5.2 Overview of characteristics of FPGAs used in experiments 65

5.3 Benchmark kernels used for the dataset 66

5.4 XPNet vs. train from scratch, model error(%) 68

5.5 Models on AMD/Xilinx devices as source model and XIlinx or Intel as target (GNN) 69

5.6 Models on Intel devices as source models (GNN) 70

5.7 Model Error(%) for XPNet with different Design Selectors 71

5.8 The table including features being used in the MLP model 73

5.9 Models on AMD/Xilinx devices as source model and Xilinx or Intel as target (MLP) 73

5.10 Models on Intel devices as source models (MLP) 74

5.11 Model Accuracy comparison (same FPGA): XPNet vs PowerGear . . 75

5.12 Model Accuracy comparison (cross-FPGA): XPNet vs PowerGear . . 75

5.13 XPNet Speed comparison to prior approaches - training 77

5.14 XPNet Speed comparison to prior approaches - inference 77

6.1 A comparison of ATAPP with the existing work on power prediction. : feature supported; : feature unsupported; : feature partially supported. 80

6.2 Features used in ATAPP 92
6.3 Benchmark kernels used for the design dataset per FPGA 95
6.4 Overview of characteristics of FPGAs used in experiments 95
6.5 XPE estimation error compared to Vivado in different design phases for UltraScale Virtex vu440. P_{syn}, P_{impl}: power estimated by Vivado after syn and impl. P'_{syn}, P'_{impl}: power estimated by XPE after syn and impl 97
6.6 Accuracy Comparison of different methods
ATAPP: Prediction with design and arch features (Pos Enc=Positional Encoding); `Leave-one-out` for training and testing;
PowerGear: Train with the designs on FPGA8 and test with the designs on FPGA1-FPGA7;train on FPGA1 and test on FPGA8;
XPE: Estimates with design characteristics on each FPGA sheet . . 100
6.7 Accuracy Comparison on **unseen devices** vs. **unseen generations**:
unseen devices: the same generation devices with test device exist in training;
unseen generations: the same generation devices with test device do not exist in training;
FPGA 1-3 from ultrascale+, FPGA 4-5 from ultrascale, FPGA 6-8 from 7series. 103

List of Figures

1.1 The functionality of HLS 19

1.2 Left: static power breakdown of a 90nm FPGA; Right: dynamic power breakdown of a Virtex-II FPGA. 20

1.3 The common flow to evaluate power for HLS designs 21

1.4 (Left) Power consumption of multiple designs on 2 different FPGAs. For each design, blue represents one FPGA and red represents another. (Right) Histogram showing the power difference of each design between 2 different FPGAs. 23

3.1 The detailed view of HLS flow 33

3.2 The activity tracing and graph generation 34

4.1 The flow of general ML-based methods in HLS 37

4.2 Example template Tcl file to generate the optimization strategy for the application `bfs` from Machsuite 42

4.3 Resource utilization of designs generated for ZU9EG, applications are from Rosetta, Polybench, Machsuite and CHStone 46

4.4 Usage of HLSDataset to construct machine learning based power model 50

4.5 Usage of HLSDataset to construct machine learning model for estimation of resource utilization 51

5.1 Graph generated after HLS (left) and graph generated after implementation (right) 56

5.2 Overview of XPNet. Train and Fine-Tune (top). Inference (bottom) . 58

5.3 Detailed view of the Label Generator 59

5.4 Detailed view of the Feature Generator 60

5.5 Detailed view of the Design Selector for fine-tuning dataset 62

5.6 Model Architecture of XPNet 64

5.7 Resource utilization and latency for the *Xilinx* designs 67

5.8 Resource utilization for the *Intel* designs 67

5.9 XPNet with the fine-tune dataset created by *random design selection* 72

5.10 XPNet implemented with the MLP 75

6.1 Changes in power from one FPGA to another. Power may increase or decrease as shown (left) for example designs; Distribution of the relative power difference on two different FPGAs based on over 4000 designs (right) 79

6.2 Power estimation by state-of-the-art ML-based power model (left) vs. ATAPP (right) 81
6.3 FPGA Architecture Terminology 85
6.4 FPGA Architecture Encoding 87
6.5 Design representation generation flow 91
6.6 Detail structure of ATAPP composed of *design embeddings generation* using GNN encoder and *architecture embeddings generation* using MLP; the power regression is conducted with a MLP 93
6.7 Architecture of GNN encoder 94
6.8 Prediction vs ground-truth for dynamic power consumption of all designs. The black line in the middle of each figure indicates **zero error** (i.e. predicted power equals to the ground-truth power). **ATAPP**: **leave-one-out** strategy is used, the model is tested with the designs on one FPGA and trained with the designs on all other FPGAs. 8 models are trained and tested independently. **PowerGear**: train with the designs on FPGA8 and test with the designs on FPGA1-FPGA7; train with the designs on FPGA1 and test with the designs on FPGA8. Blue - ATAPP; Red - PowerGear; Green - Vivado XPE. 99
6.9 **ATAPP** is tested on designs with different target clock periods on different FPGAs. The training settings, **leave-one-out** strategy, are the same as shown in Table 6.6. **PowerGear** is tested on designs with different clock periods on different FPGAs. The model is trained with the designs on FPGA8. 101

Chapter 1: Introduction

1.1 Field-Programmable Gate Array

The rapid advancement of Internet technologies has led to the continuous expansion and diversification of digital data. In the era of big data, heterogeneous systems serve as a promising foundation for handling large-scale applications while meeting strict power and performance demands. Using heterogeneous computing techniques, various computational resources can specialize in their respective domains and collaborate to execute complex tasks efficiently. Both academia and industry are increasingly adopting field-programmable gate arrays (FPGAs) as a key component in the implementation of heterogeneous computing systems.

Field-programmable gate arrays (FPGAs) [1, 2] are a type of programmable device that can be configured into customized hardware structures that are tailored to specific applications. Typically, an FPGA consists of a two-dimensional array of configurable logic blocks (CLBs), interconnected through a routing network. Each CLB can be divided into multiple basic logic blocks containing a K-input look-up table (K-LUT) and a flip-flop (FF). Commercial FPGAs commonly feature four or six inputs per LUT. The K-LUT is a small static random access memory (SRAM) capable of implementing any logic function with up to K inputs. A flip-flop (FF) is positioned at the output of each LUT and can be bypassed or utilized, allowing the logic block to generate combinational or sequential logic as needed. These CLBs are interconnected and programmed with routing network to form the logic and mathematical functions. Within the routing network, the switching blocks (S blocks) facilitate connections between horizontal and vertical wiring, while the connection blocks (C blocks) link the input and output of the CLBs to the routing tracks. Beyond CLBs, modern FPGAs integrate additional hardware elements, such as block random access memories (BRAMs) and digital signal processing units (DSPs), to efficiently manage data buffering and arithmetic computations. These components are typically distributed

in a structured manner, similar to CLBs. Furthermore, FPGA chips incorporate programmable input and output (I/O) pins and clock trees. Advanced FPGA architectures also feature artificial intelligence (AI) engines and network-on-chip (NoC) structures, further enhancing their capabilities [3, 4].

There are a variety of ways to implement a design on an FPGA. One common way is to write Register-Transfer-Level (RTL) designs manually and synthesize and implement them on the FPGA board with commercial tools. However, it requires proficiency in RTL development and may not be friendly to application developers. Addtionally, any algorithm or design changes at application level may lead to rewriting of the whole RTL codes. Therefore, High-Level-Synthesis, an automated design process converting high-level applications into RTL code, is developed to improve productivity.

1.2 High-Level-Synthesis (HLS)

High Level Synthesis (HLS) [5] is an automated flow to transform an application written in a high-level language, such as C/C++ or SystemC, to a register transfer level (RTL) description as illustrated in the Figure 1.1. It enables rapid development hardware accelerators for various applications such as image processing, machine learning, and signal processing [6, 7, 8, 9]. HLS tools take directives/constraints, which is entered by the user, as input to manipulate the designs according to the user preference and meanwhile the generated designs should be easily mapped to RTL ips according to the technology libraray. To achieve optimal performance, HLS tools rely on pragmas, which are directives inserted into the source code to guide the synthesis process, and these pragmas, which includes array partition, loop pipeline and unrolling, can have various settings and values that significantly impact the resulting hardware design. Constraints, on the other hand, are the limit entered by the user manually which restricts the design synthesis and implementation under a certain circumstance. For example, with 1 adder constraints, the HLS tool will try

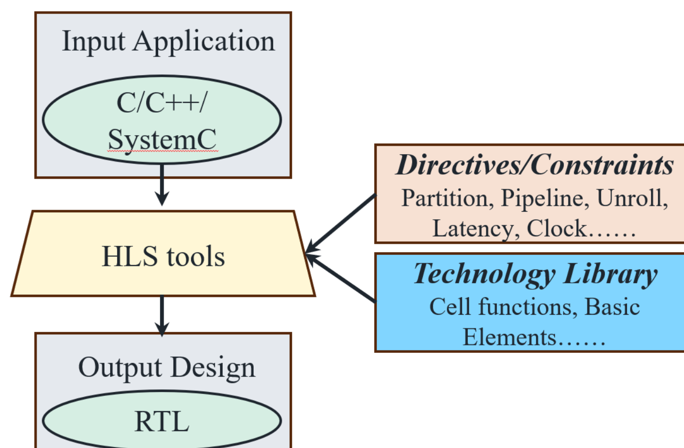


Figure 1.1: The functionality of HLS

to generate the RTL design with only one hardware adder.

Compared to traditional RTL-based hardware design flow, HLS offers many advantages. The HLS tool generates optimized RTL and reduces human error and design effort. Additionally, designers can focus on high-level algorithmic optimization without worrying about too much detail at the RTL level. It will greatly reduce the iteration time and thus increase productivity. HLS also improved the efficiency on design optimization due to the pragmas/directives control. HLS tools can generate different RTL designs with different pragma settings and it efficiently helps to find optimzal designs. Moreover, users can control the design generation with different constraints to meet their design budget or target.

1.3 FPGA Power

Power efficiency has emerged as one of the first-order constraints for hardware systems such as field-programmable gate arrays (FPGAs), and both the FPGA architecture and design optimization with regard to power efficiency usually necessitate knowledge of power consumption. The power consumption of an FPGA can be decomposed into two main parts: static power and dynamic power. Static power refers to the leakage power consumption when an FPGA is powered on with no circuit be-

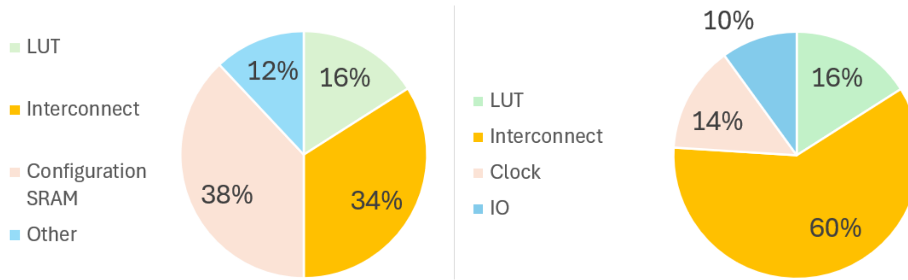


Figure 1.2: Left: static power breakdown of a 90nm FPGA; Right: dynamic power breakdown of a Virtex-II FPGA.

havior involved. It is dominated by routing interconnects, configuration SRAM cells, and LUTs [10] as shown on the left of Figure 1.2, and it is also highly dependent on process, voltage, and temperature. Dynamic power, on the other hand, is introduced by signal transitions which dissipate power by repeatedly charging and discharging the load capacitors. As illustrated in the right of Figure 1.2, the dynamic power consumption is dominated by the interconnect power and LUT power on Virtex-II FPGA [11]. There are many techniques to save static and dynamic power. Static power saving techniques includes power gating [12, 13], clock gating [14], dual voltage supplies [15, 16], power-aware placement and routing algorithms [17, 18]. Moreover, run-time power management strategies have gained a surge of attention. Adaptive voltage scaling [19], dynamic voltage frequency scaling (DVFS) [20] and task scheduling in FPGA-CPU systems [21, 22], have been reported as successful ways to reduce the run-time power consumptions of FPGAs.

The common practice to evaluate power is to use run-time power monitor or design-time power model[2]. While run-time power monitoring gives more accurate power profile of designs, it is more costly. In order to monitor run-time power consumption, dedicated power measurement devices or circuits are usually utilized. Although these devices can be used in modern FPGA systems, the scope of applicability is limited. They normally require extra space on the device and the granularity is in the order of milliseconds. The coarse-grained detection is not sufficient for a more fine-grained power monitoring, which is within hundreds of clock cycles. Emerging

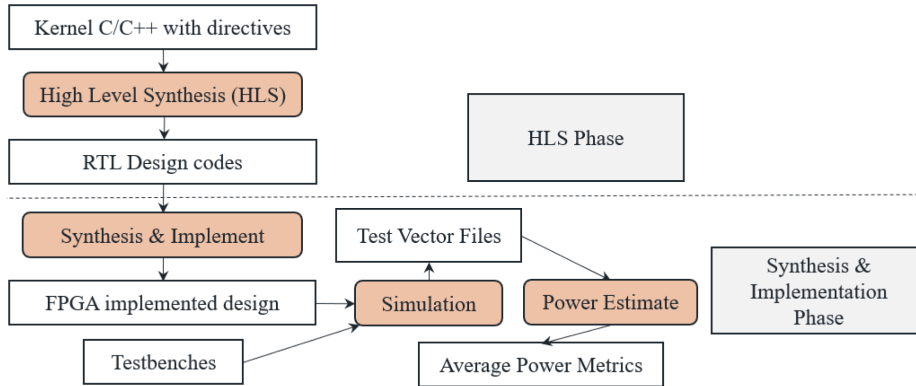


Figure 1.3: The common flow to evaluate power for HLS designs

technologies in integrated circuit design have motivated the realization of on-chip regulators with a short voltage scaling time, in the order of sub-nanoseconds [23]. With the use of on-chip regulators, many novel power management strategies have been implemented at fine temporal granularity, such as fine-grained DVFS [23, 24, 25], adaptive voltage scaling and dynamic phase scaling for on-chip regulators [26] for contemporary computing systems. The other faster way is to evaluate power of designs at design-time. The general flow to evaluate the average power of the FPGA design is illustrated in Figure 1.3. Firstly, HLS translate the application described by high-level-languages (C/C++) into RTL designs. Then the post-implement design is generated with synthesis and implementation. The post-implementation design is simulated with testbench to generate test vector files (e.g., saif file). The test vector files are then used to estimate the average power. With the aid of HLS tools, designers targeting hardware implementation for FPGAs are no longer required to know much details about the low-level hardware, such as micro-architectures of individual components and interconnection between them. HLS tools can provide rough estimation of performance and resource utilization of the generated hardware design with different pragma/directive values. Therefore, HLS greatly increase the productivity and enables efficient DSE [27, 28, 29, 30, 31, 32]. The common power evaluation flow illustrated in Figure 1.3, however, induces large overheads of design turnaround

time. More than 90% of time is consumed by sythesis, implementation and simulation phase, and these off-the-shelf HLS tools [33, 34] are still lacking a mature and efficient power analysis techniques.

1.4 Problem Description and Motivation

To enable accurate and fast estimation of design-time power, prior work employed Machine Learning (ML) techniques to bypass the time-consuming phases including synthesis, implementation and RTL simulation phases [35, 36, 37]. ML models are trained with features extracted from the HLS phase and labels from simulation-based power consumption. Using these ML models, post-implementation power can be inferred right after HLS stage. However, some inefficiency of these ML-based power models is observed. The source of this inefficiency will be discussed with three challenges in the following paragraphs.

First, the dataset preparation is time consuming. Obtaining accurate ML-based power models requires a large number of designs (e.g. thousands). Collecting data from these designs can take weeks to months [38, 39, 40, 41], depending on the computation resources available, because of the time-consuming steps from RTL-description to FPGA implementation and finally simulation-based power evaluation. This long turn-around time becomes a significant obstacle to building an ML model for an FPGA.

Second, the prior methods used to build ML models are vendor-specific. The ML-based power model is constructed with the features extracted after HLS backend. The features, as well as the way to extract those features, however, are quite specific to AMD/Xilinx vendors. The process cannot be applied to other tools such as Intel HLS, since the required scheduling/binding info cannot be accessed or they are in different format. Therefore, researchers have to spend a significant amount of time to do feature engineering for a new vendor. Moreover, the HLS backend features require extra time to extract which leads to extra time in inference.

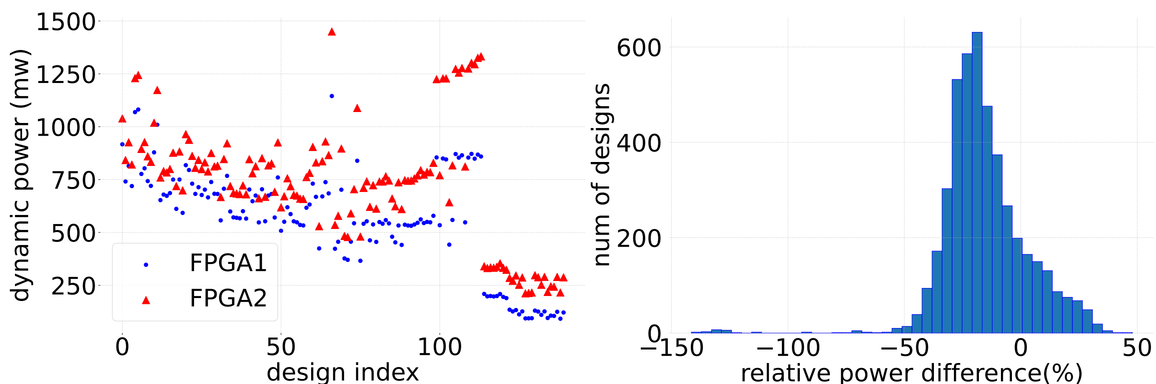


Figure 1.4: (Left) Power consumption of multiple designs on 2 different FPGAs. For each design, blue represents one FPGA and red represents another. (Right) Histogram showing the power difference of each design between 2 different FPGAs.

Third, current ML-based power models need to be trained from scratch for a new FPGA target. A pretrained FPGA model cannot be used on the other FPGA directly because these models are trained on a dataset of HLS features (from a specific set of workloads) and the power consumption (for a specific FPGA device). Therefore, even if the set of workloads remains the same, the power model cannot be directly reused for the power estimation of a different FPGA hardware because the trained model does not comprehend the new FPGA's architecture, thus, limiting their reuse. The power value will then certainly be different between the two FPGAs as shown in the left of Figure 1.4. The relative power difference distribution in the right Figure 1.4 shows that simple scaling or calibration is not sufficient for the power model on one FPGA to function correctly on the other.

The dissertation tries to tackle the above mentioned problems, and it is believed that the use case of the dissertation is broad. It would be very beneficial to estimate the power consumption of an HLS design on an FPGA that differs from the one available to you.In other words, when using ML, it would be very useful if a pretrained FPGA-specific power model can be reused to predict the power consumption on a different FPGA using a very limited dataset size. It would be even more efficient if the model can predict the power for an unseen FPGA. Such methodologies could

have many use cases:

- Consider that a company is launching a new FPGA. They have a model for power prediction for their older architecture. They can incrementally train this model using a few engineering designs of the new FPGA, and share this model with customers. An even better model can be directly used to predict power for this new FPGA without any new designs on it. The customers can use this to plan their systems and software with the new FPGA, and reduce time-to-market for their products.

- Having limited access to physical chips is a common occurrence, whether in academia or in industry. This can be because of chip shortages from supply-chain issues, or just because the chip is in heavy demand and each person/team gets to use it only for a limited amount of time. In this case, a model that can be trained quickly for a new device is very useful for system design and hardware software codesign.

- Consider an academic researcher proposing a new accelerator but they have power results for an FPGA they own. They want to compare their results to another paper, but the paper used a different FPGA to which they do not have access. They have the power prediction model for their own FPGA. They can incrementally train it for the FPGA used by the other paper using limited designs and predict the power consumption of their accelerator on the other FPGA, and then compare results.

- Consider academic researchers proposing a new/customized FPGA architecture to reduce the power consumption on a specific system. They will need to explore and sweep the FPGA architecture designs in order to find the optimal one. A model to predict the power on a new/unseen FPGA can greatly help to reduce the iteration time and increase the efficiency for future FPGA development.

In short, the ultimate goal of the dissertation tackles the problem of efficiently building ML-based power models for various types of FPGAs. In order to achieve the goal, the dissertation includes three important topics: (i) well-curated datasets and an efficient flow to build such a dataset, (ii) efficient construction of ML-based power model for a different FPGA with existing models, and (iii) a FPGA architecture-aware, technology-aware ML-based power model.

1.5 Contributions of Dissertation

This dissertation proposes novel techniques to improve the efficiency and accuracy for design-time dynamic power prediction with HLS designs on different FPGAs. The primary contributions can be summarized into the following three topics:

- **HLSDataset: Open-Source Dataset for ML-Assisted FPGA Design using High Level Synthesis**: HLSDataset is a well-curated open-source dataset for ML-assisted FPGA design using HLS. Currently it contains more than 4,000 HLS designs per FPGA and 10 FPGA designs (40,000 designs in total) are contained. It also contains the methodology to automate the dataset generation efficiently and the dataset can be easily extended with the automation.

- **XPNet: Cross-FPGA Power Prediction from High Level Language Code**: the first methodology which is capable of constructing an ML model that accurately estimates dynamic power **across** FPGAs. By employing only the features from HLS frontend, XPNet constructs a GNN-based model that can be transferred between FPGAs, even those from different vendor. XPNet produces 8.40% average error with pretrained model on one device and 20 designs on the other target device within AMD/Xilinx vendor family (Xilinx-to-Xilinx). It further shows a 10.34% average error predicting using a pretrained model on one AMD/Xilinx device and 20 designs on a target Intel device (Xilinx-to-Intel).

- **ATAPP: Architecture and Technology Aware Power Predictor for Unseen FPGAs**: ATAPP is a GNN-based model and the first ML model making power evaluation based on the FPGA architecture and designs. It is trained on seen FPGA and designs but can make power prediction for both unseen FPGA architecture and designs. Positional encoding is used to efficiently encode the FPGA architecture features and GNN encoder is developed to generate graph embeddings with switching activities-aware graph, which are extracted from IR level simulation. It shows the ability to predict the dynamic power with a 13.09% error on average for unseen FPGAs.

1.6 Thesis Statement

Machine learning techniques can be used to efficiently perform power estimation in the early stage of FPGA High-Level-Synthesis designs. The machine learning-based power model for different FPGAs can be efficiently constructed with well-curated datasets, transfer-learning techniques, and FPGA architecture and design representations.

1.7 Dissertation Organization

The organization of the dissertation is as follows. Chapter 2 summarizes the background of ML techniques used in Electronics Design Automation (EDA) domain and power estimation as well as prior datasets. Chapter 3 describes the general assumption, methodology and settings used throughout the dissertation. Chapter 4 presents a well-curated dataset (HLSDataset) and the automation flow. Chapter 5 explores a Cross-FPGA power predictor with transfer-learning techniques (XPNet), which efficiently adapt the power models across FPGAs. Chapter 6 discusses an FPGA architecture and technology-aware ML-based power model (ATAPP), which provide accurate power estimation for unseen FPGAs. Finally Chapter 7 concludes the dissertation and presents possible avenues for future work.

Chapter 2: Background and Related Work

2.1 FPGA Power Consumption

The FPGA power consumption can be decomposed into static power and dynamic power, and static power refers to the leakage power consumption when an FPGA is powered on while no circuit activities are involved. It mainly depends on the specification of a FPGA and therefore static power is easier to estimate. Dynamic power, on the other hand, is intrduced by signal transitions which dissipate power by repeatedly charging and discharging the load capacitors. Equation 2.1 formulates the dynamic power consumption as

$$P_{dyn} = \sum_{i \in I} \alpha_i C_i V^2 f \tag{2.1}$$

In the equation, α_i is the signal switching activity, C_i is the interconnect capacitance, V is the supply voltage, f is the operating frequency and i is an interconnect of the whole set I. For a specific FPGA, the supply voltage V is fixed, and C for each i is decided by the FPGA resources type and placement and routing algorithms. This dissertation mainly focus on the design-time dynamic power estimation.

2.2 Dataset for HLS

Work	# Samples	# Sources	Platform & Abstraction level	Tools
OpenABC-D [42]	870,000	29	ASIC RTL	OpenROAD
CircuitNet [43]	12,960	6	ASIC Physical Design	Synopsys DC
Dai [44]	1,300	65	FPGA HLS	Xilinx Vivado
MLSBench [40]	6,000	30	FPGA HLS	Xilinx Vivado
Spector [45]	8,300	9	FPGA HLS	Altera OpenCL SDK
Ours	18,876	34	FPGA HLS	Xilinx Vivado

Table 2.1: Comparing HLSDataset with prior open-source datasets for training ML models for chip design (general)

Work	Use Case in ML
OpenABC-D [42]	Estimation of quality of a synthesis recipe
CircuitNet [43]	Congestion prediction, DRC violation Prediction, IR drop prediction
Dai [44]	Quality of Results Estimation on one FPGA
MLSBench [40]	NA
Spector [45]	NA
Ours	Power Estimates, resource and timing estimation, operation delay estimate, cross-FPGAs studies, and more

Table 2.2: Comparing HLSDataset with prior open-source datasets for training ML models for chip design (Use case in ML)

The success of ML-based models depends on well-curated datasets. There are a few datasets for training ML models to assist in chip design problems in the ASIC domain. OpenABC-D [42] from NYU is a large-scale, labeled dataset produced by synthesizing open source designs with an open-source ASIC logic synthesis tool. This dataset can be used in developing, evaluating and benchmarking ML-guided logic synthesis but is applicable to a very small subset of problems i.e. prediction of ASIC synthesis results. CircuitNet [43] is another open-source dataset targeted for three prediction tasks in backend ASIC flows - congestion prediction, DRC (Design Rule Check) violation prediction, and IR drop prediction. It contains more than 10000 samples (in form of 2D image-like data) obtained by running open-source RISC-V designs through commercial EDA tools. This dataset is applicable to only a few physical design problems.

For FPGA HLS design flow, which is the focus of this paper, there are a few open-source datasets as well. Dai et al. [44] have open-sourced a dataset that is applicable to prediction of resource usage and delay (or frequency) for FPGAs from high-level applications written in C. The dataset is generated by using applications from suites such as CHStone, Machsuite and Rosetta, and the Vivado tool chain from Xilinx/AMD. This dataset is restricted to use only in estimation of resource usage and timing for FPGA, and contains only limited data. The data provided is only for 1 FPGA device, implying that this dataset can not be used for cross-FPGA predictions.

MLSBench [40] is an open-source dataset generated from 17 C/C++ and 13

SystemC benchmarks using Xilinx Vivado HLS tool flow. The C sources to generate the designs are from S2CBench [46], CHStone [47] and MachSuite [48]. The dataset contains only log files and reports generated from Xilinx Vivado HLS tool flow, but without directly consumable features, labels and RTL codes. Also, this dataset is limited to only one FPGA. Therefore, MLSBench is hard to extend and quite limited in ML usage.

Spector [45] is a benchmark suite that contains applications written in OpenCL. The authors run the benchmarks through Intel OpenCL SDK to generate 8300 hardware designs targeted for Intel FPGAs. In addition to just the benchmarks, several metrics for each design sample (based on compilation using Intel OpenCL SDK) are also provided. The focus is on HLS tool flows and design space exploration.

Table 2.1 and Table 2.2 compare the various properties of these datasets. It shows the number of samples contained in the dataset and number of sources used for generating the dataset. These datasets generally cater to limited usecases (eg: physical design prediction in [43], or RTL synthesis quality prediction in OpenABC-D [42] or resource usage prediction in Dai et al.[44]). Some need further expansion and curation to be readily usable by others. Retargeting the few available datasets for a new ML model requires significant manipulation and augmentation. So, researchers often generate their own dataset every time they want to solve a new problem. In this process, they have to rerun tool flows to generate reports and then write scripts to parse those reports repeatedly.

2.3 Machine Learning for HLS designs and power estimation

Machine Learning (ML) algorithms have gained popularity in the Electronic Design Automation (EDA) domain due to their extremely high efficiency, high quality [49], and owing to their great potential to solve NP-complete problems which are common in EDA domain. While traditional analytical solutions, on the other hand, lead to huge time and resource consumption. ML models have shown remark-

able success in various design phases of the EDA flow, such as High-Level Synthesis (HLS),[27, 35, 36, 50, 51, 52, 53, 54, 55, 56, 57, 58], logic synthesis [59, 60], and placement and routing in physical design [61, 62, 63, 64, 65]. As [49] points out there are four major tasks specifically for HLS: (1) Result prediction including timing, resource usage, power, maximum frequency, throughput, area, latency and operation delay [44, 52, 56]; (2) Cross-platform performance prediction such as performance prediction for new FPGA platforms and performance prediction for new applications through the execution on CPUs [54, 66]; (3)Active Learning where DSE (Design Space Exploration) for HLS is performed and ML models are used as surrogates for actual synthesis when evaluating a design [27, 30]; (4) Improving optimization algorithms where ML models are used to substitute traditional algorithms for hyper-parameter or configuration selection [67, 68].

Traditional accurate power analysis is usually inefficient due to long-running synthesis and simulation. Power is computed from the switching activities of individual signal nets and the capacitive load they drive. The approach is very accurate and serves as the sign-off standard; however, it comes with a very long turn-around time in simulation and computational cost. To address the problem, ML techniques have been widely employed in every design cycle to perform power estimation including architecture-level power prediction [69, 70, 71, 72, 73], RTL stage power modeling [74, 75, 76, 77, 78, 79, 80, 81, 82] and HLS stage power estimation [35, 36, 37, 54, 55, 56, 83]. Compared to architecture-level power estimation, HLS designs provide a closer look of the hardware designs, and thus more accurate power estimation can be produced. Althgouh RTL-based power evaluation is more accurate, HLS-level power estimation can save significant time without losing much fidelity of results. HL-Pow [36] adopts Convolutional-Neural-Networks (CNNs) to infer the measured power onboard. They generate the switching activity on the C-level operators and further link them to RTL operators with HLS report mapping information. The switching histogram is built for each operator and fed into their CNNs to infer the power. PowerGear [35], on the other hand, uses graph-neural networks (GNNs) to

perform the estimation. They extract the switching activities in a similar way as HL-Pow and recover a graph with operators and the switching characteristics. The graph samples are then used in a GNN model to infer the dynamic power. Unlike the previous two works, HLSPredict [54] uses Random Forest (RF) and Artificial Neural Network (ANN) to predict the power of HLS design with performance counters on a desktop CPU as features. All of the above-mentioned works perform well for the same FPGA they are trained with, but their models are not aware of FPGA architectures restricting their model usage on single FPGA.

Chapter 3: Methodology

3.1 High Level Synthesis

The High-Level-Synthesis (HLS) terminology used in this dissertation is from AMD/Xilinx [33] by default. In order to generate sufficient different designs using High-Level-Synthesis, the following types of pragmas/directives are used:

- *array partitioning.* Arrays can be partitioned into blocks or into their individual elements. The partition can be controlled by the directives using *set_directive_array_partition* command. When an array is partitioned into multiple blocks, the single array is implemented as multiple RTL RAM locks. When partitioned into elements, each element is implemented as a register in the RTL. In both cases, partioning allows more elements to be accessed in parallel and can help with performance: the design trade-off is between performance and the number of RAMs or registers required to achieved it.
- *loop pipelining.* Pipelining the loop allows the subsequent iterations of the loop to overlap and run concurrently. Pipelining a loop can be enabled by adding the *PIPELINE* pragma inside the body of the loop or use *set_directive_pipeline*.
- *loop unrolling.* A loop is executed for the number of iterations specified by the loop induction variable. Loops can be unrolled to create multiple copies of the loop body in the RTL design and it allows some or all loop iterations to occur in parallel. A loop can be unrolled by adding the *UNROLL* pragma inside the body of the loop or use *set_directive_unroll*.

The detailed view of HLS flow is concluded in Figure. 3.1. HLS is composed of frontend compilation and backend synthesis. The LLVM intermdiate representation (IR) code is generated after frontend compillation. IR code is an abstract,

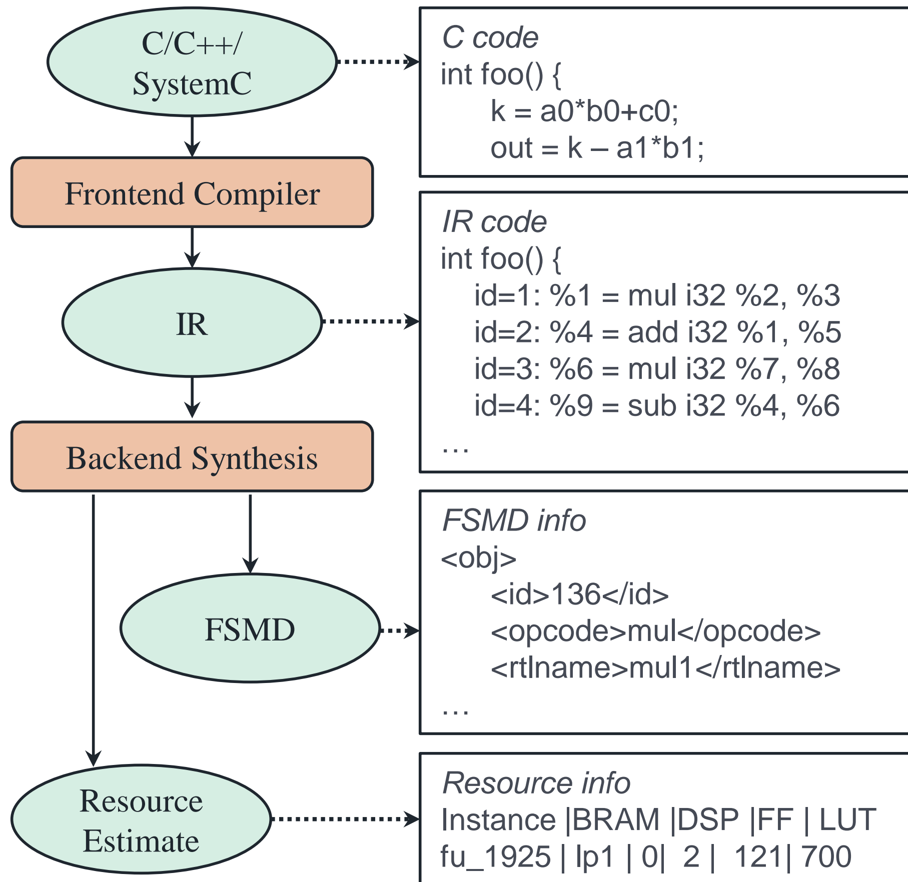


Figure 3.1: The detailed view of HLS flow

machine-independent code used as an intermediate step during the compilation of high-level programming languages into machine code or executable code. It serves as a bridge between the source code and the final machine code, allowing the compiler to perform various optimizations and transformations efficiently. Backend synthesis process will then generate final RTL after performing resource allocation, resource binding and scheduling. Resource allocation phase allocate hardware resources to the IR operators according to the design constraints and optimization goals. Meanwhile, each IR operation is binded to specific hardware units, ensuring efficient utilization of resources. Scheduling determines the order and timing of operations to optimize latency, throughput and area.

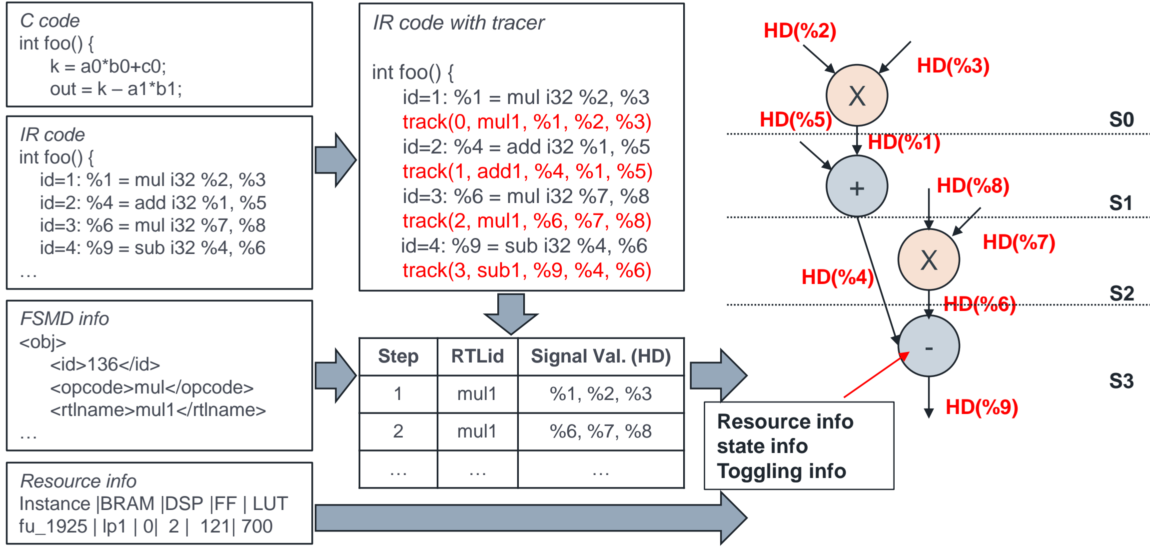


Figure 3.2: The activity tracing and graph generation

Besides the final RTL codes, two crucial files are also generated: Finite State Machine Datapath (FSMD) model and resource estimation reports. The FSMD model describes the FSM stages, dataflow, and RTL operator information including each RTL operator's ID, operand bitwidths, and related IR instructions. Resource estimation reports, on the other hand, provides the resource utilization for each operator. The three files: IR codes, FSMD model and resource estimation reports are essential to construct the power model.

3.2 Activity Model Generation

The toggling behavior is one of the main components in calculating dynamic power as shown in Equation 2.1. Although it is unrealistic to trace the toggling behavior of each net in the post-HLS stage, it is possible to trace the toggling behavior of each RTL operator or IR operator with C-level simulation [2, 37, 57, 58, 84]. The way to extract togging behaviors at IR-level is summarized in the left part of Figure 3.2. Multiple tracing function is inserted at each lines of the IR codes to record the value changes on each IR operator. The behavior of each IR operator is then

recorded as hamming distance (HD). A more accurate model with RTL-to-IR back tracing can be generated based on the observation that multiple IR operators can be mapped to the same RTL operator due to resource sharing in the HLS backend synthesis. Therefore, multiple IR operators may contribute to the activities of the same RTL operator in different steps. With FSMD model, it is possible to match each IR operator to the RTL operator. IR tracing is developed with LLVM compilation [85]. With the IR codes, a dataflow graph (DFG) can be generated. Each node represents an IR operator, while the edge attribute represents the toggling behavior of the associated IR operator. With the introduction of FSMD model and resource info, the graph can be refined to represent dataflow for RTL operators. The node attributes contains more info including resource usage to implement the operator, state number and activate ratio etc.

3.3 Tools

Experiments employed for this dissertation use the following tools:

- AMD/Xilinx Vivado for implementation for Xilinx FPGAs
- AMD/Xilnx Vivado for simulation of post-implementation design and power analysis
- Intel Quartus for implementation for Intel FPGAs
- AMD/Xilinx Vivado HLS and Intel HLS compiler for high level synthesis
- Python for scripting and ML model developement
- LLVM for IR annotation
- AMD/Xilinx Power Estimator (XPE) for baseline power implementation

Chapter 4: HLSDataset: Open-Source Dataset for ML-Assisted FPGA Design using High Level Synthesis

High-level synthesis (HLS) is able to convert software applications into FPGA hardware designs with different optimization strategies. It can greatly improve the productivity since hardware designers do not need to write low-level hardware description language (HDL) from scratch given an application written in a high-level language (HLL) like C, C++ or SystemC.

While HLS greatly helps to reduce the effort for the software to FPGA implementation conversion, it is quite time-consuming, especially when large design spaces need to be explored using various pragma settings. This is a common usecase when designing application-specific optimized designs targeting FPGAs, for example, when designing FPGA based accelerators for ML applications. Metrics such as resource usage and achieved clock frequency reported after HLS are estimates. To find the final metrics, the even slower implementation process (synthesis, place and route) is required. Even more efforts are needed to estimate power consumption accurately, since low-level simulation is required. For these reasons, efficient design space exploration targeting optimization of such metrics is hard. To address this challenge, machine learning (ML) based techniques are widely adopted to provide accurate resource usage and power estimation at early stage in HLS. S. Dai et al. [44] uses Lasso linear model, XGB and artificial neural network (ANN) to calibrate the resource usage and timing results from HLS reports. Graph neural networks (GNNs) and HLS reports are used to predict performance in the work by N. Wu et al. [55]. HL-Pow [36] and PowerGear [35] give solutions to predict power consumption using convolutional neural networks (CNNs) and GNNs respectively. E. Ustun et al. [52] builds graph samples using the IR (intermediate representation) generated during HLS and use them as input to GNNs to predict operation delay.

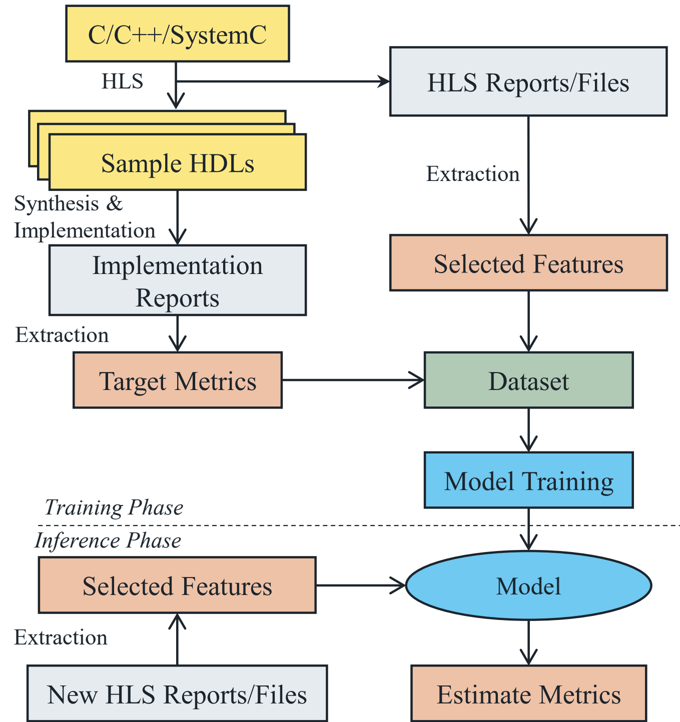


Figure 4.1: The flow of general ML-based methods in HLS

The flow of general ML-based methods in HLS domain is shown in Figure 4.1. ML based methods can provide fast and accurate metrics estimation with HLS reports, however, extensive dataset is needed to train the models to produce acceptable results. To generate task-specific dataset in HLS domain requires lots of effort:

1. Software source code should cover enough domains
2. Source code should be well manipulated with HLS directives so that HLS optimization can be applied
3. Varieties of optimization strategies need to be applied to the source code so that wide range of hardware designs can be generated
4. Implementation is needed if the post-implementation metrics are the prediction goal

5. Extensive scripting is required to extract the data from reports and preprocess before it can be consumed directly in ML models

6. Significant computing resources may be needed for large number of tool runs to collect enough data

Researchers have to generate their own dataset, which can be extremely time-consuming because of the aforementioned reasons. Due to the different prediction goal and ML models, existing datasets are proprietary and not shareable or reusable. However, there is an opportunity to reduce, and even eliminate, the redundant work for various researchers by creating a dataset that contains common usable information, allowing them to focus on training the ML models instead of generating the dataset. It is observed that resource usage reports, Intermediate Representation (IR) code, IR operator information, Finite State Machine Data path (FSMD) model from HLS are commonly used as the source of features. The resource utilization, timing information and power consumption values from post-implementation phase are the common metrics that researchers are interested to predict.

This chapter describes HLSDataset: a well-curated open-source dataset for ML-assisted FPGA design using HLS. The dataset can be used by a large subset of problems in this domain. The dataset currently contains nearly 9,000 Verilog designs per FPGA type, and two FPGA types are covered. To ensure diversity of designs, HLSDataset are generated from multiple applications across various benchmarks: Polybench [86], Machsuite [48], CHStone [47] and Rosetta [87], and each application is tuned to generate a variety of hardware design samples. The dataset contains all necessary files and reports for every design so that features and target metrics can be easily extracted. In this chapter, the dataset is described how it can be used, and its utility will be demonstrated by conducting two case studies. The dataset is expected to be widely usable and get even more useful with time through contributions by the FPGA research community.

The contribution of HLSDataset is listed as follows:

1. HLSDataset is a well-curated open-source dataset for ML-assisted FPGA design using HLS

2. More than 4,000 HLS designs per FPGA and 10 FPGA designs (40,000 designs in total) are contained

3. The methodology to automate the dataset generation is described in detail and can be easily replicated to extend the dataset.

4. Two in-depth case studies are conducted to demonstrate the effectiveness of HLSDataset.

4.1 HLSDataset Construction

Table 4.1 gives a general overview of HLSDataset. High level language (HLL) sources are used and they belong to various application domains such as multimedia, arithmetic, signal processing and machine learning, from multiple popular benchmark suites such as Polybench [86], Machsuite [48], CHStone [47] and Rosetta [87]. Xilinx Vivado/Vitis tool chains are used for HLS and implementation. Two FPGAs are used: ZU9EG and XC7V585T. The dataset will be extended to include more FPGAs, including Intel FPGAs. One target frequency of 100 MHz is used.

Category	Details
Num designs	$\tilde{5}$0,000
Num applications	34
Application sources	Polybench, Machsuite, CHStone, Rosetta
FPGAs	ZU9EG, XC7V585T
Clock frequency	100MHz
Domains	Multimedia, Arithmetic, Signal processing, ML
Machines	9 16-core Intel Xeon 5218 2.3GHz 384 GB RAM
Time	More than 4,500 hours
Tools	Xilinx Vivado/Vitis

Table 4.1: General overview of HLSDataset

4.1.1 C source code manipulation

Verilog designs generated from C benchmarks are highly dependent on HLS directives, pragmas and the target clock frequency. For generating the dataset, the design space of *loop unroll*, *loop pipeline* and *array partition* are the main focus. Loops in C code need to be labelled so that loop unroll and loop pipeline can be applied to generate efficient designs. Machsuite and Rosetta are already well-written with HLS directives, and their code can be directly used for the dataset generation. Polybench and CHStone source code are manipulated with HLS directives.

4.1.2 Auto-generation of Tcl scripts

The scope of generated Verilog designs can be huge, since the factors for *array partition* and *loop unroll* can vary greatly. The number of generated designs is determined by the number or the dimension of the factors. However, manually writing every Tcl script (Xilinx Vivado/Vitis tools use a Tcl script based interface), which is used to tune HLS solution for the generation of Verilog designs in the dataset, is time-consuming and unrealistic. In order to generate designs more efficiently, a template Tcl script for every C source code and a script to parse it are developed. The script will auto-generate Tcl files which can be directly used by the HLS tool. An example *template.tcl* is shown in Figure 4.2. It contains 4 blocks of lines which are classified into three types: static lines, array partition lines and loop optimization lines.

1. Static lines: The directive lines under static lines are not subject to change, they should be the same and written into every generated Tcl file.

2. Array partition lines: The first line indicates the sets of parameters applied for HLS. It contains a number denoting the number of directive lines, a list of numbers denoting the factor sets for array partition and a set of types for array partition. The rest of lines are the directive lines with placeholder that should

be replaced with the parameters defined by the first line. The placeholders inside the directive lines are replaced with the combination of factor sets and type sets, and every Tcl file will contain one combination of directive parameter. The array partition lines 2 in the Figure 4.2 generates 8 combination directive parameters in this case due to 4 factors and 2 types. Note that factor equalling to 1 means no array partition is applied.

3. Loop optimization lines: The first line denotes the number of nested loops and the number of directive lines for loop optimization. It is followed by loop optimization parameter lines, each of which indicates the depth of a loop, the name of a loop, whether to apply pipeline to the loop, whether to apply unroll to the loop and unroll factor sets. The rest of the lines are directive lines with placeholders that should be filled with settings from the loop optimization parameter lines. Unroll and pipeline are applied to at most one layer of nested loops, therefore, the number of generated directives is equal to the sum of the number of unroll factors among all the loops and the one without any loop optimization. The loop optimization lines in Figure 4.2 generates 8 combination directives for the loop optimization.

The blocks of directive lines are independent of each other, therefore the number of Tcl files is equal to the products of the number of directive parameter combination among all the blocks. In this example template, 384 Tcl files are generated, and different optimization strategies are expected. The method to generate multiple versions of Tcl files is summarized in **Algorithm 1**, each block of lines will be parsed into an object.

4.1.3 Data collection

IR code, IR operator information, FSMD model files from HLS, and resource utilization reports from both HLS and implementation are included in the dataset. In order to get the high-confidence power estimation, the testbench is written and

```
#static lines
set_directive_resource -core RAM_1P_BRAM "bfs" nodes
set_directive_resource -core RAM_1P_BRAM "bfs" edges

#array partition lines 1, factor dimension = 6
array_partition,2,[1 2 4 8 16 32],[cyclic]
set_directive_array_partition -factor [factor] -type [type] "bfs" nodes
set_directive_array_partition -factor [factor] -type [type] "bfs" edges

#array partition lines 2, factor dimension = 4*2 = 8
array_partition,2,[4 8 16 32],[cyclic block]
set_directive_array_partition -factor [factor] -type [type] "bfs" levels
set_directive_array_partition -factor [factor] -type [type] "bfs" level_counts

#loop optimization lines 1, factor dimension = 8
loop_opt,2,2
0,loop_horizons,,unroll,[2 5 10]
1,loop_nodes,pipeline,unroll,[2 4 8 16]
set_directive_pipeline bfs/[name]
set_directive_unroll -factor [factor] bfs/[name]
```

Figure 4.2: Example template Tcl file to generate the optimization strategy for the application `bfs` from Machsuite

exscuted wtih post-implementation functional simulation for vector-based power estimation.

There is a chance that the HLS tool generates the same design even though different optimization strategies are provided in the Tcl script. This can be caused by aggressive optimization parameters, which are identified as unachievable by the HLS tool. The tool then automatically downgrades the optimization parameters, which can match optimization parameters during another run. Therefore, redundant designs can be generated. Redundant designs are identified by checking the hierarchical resource utilization from HLS reports. If two or more designs have exactly the same utilization, only one will be kept in the dataset.

Algorithm 1: Method to generate multiple Tcl files

```
Input: template.tcl
Output: N different versions of Tcl files
s_lines, array_objs, loop_objs from template.tcl;
Generate N empty Tcl files
/* static lines for each Tcl file                                   */
for i ← 1 to N do
    Write s_lines to Tcl file
end
/* array partition directives                                       */
foreach o ∈ array_objs, f ∈ o.factors, t ∈ o.types do
    array_partition with factor f and type t
    Write array_partition to Tcl file
end
/* loop unroll and pipeline                                         */
foreach o ∈ loop_objs, f ∈ o.factors do
    Get the loop from loop_list in o
    Apply pipeline to loop if pipeline applies
    Apply unroll to loop with factor f if unroll applies
    Write pipeline and unroll to Tcl file
end
```

4.2 Properties of HLSDataset

4.2.1 The contents of HLSDataset

The features listed below are considered to sufficiently characterize each design sample:

1. Resource usage (the number of BRAM, DSP, FF and LUT)
2. Application domain (e.g., video/graph processing, linear algebra etc)
3. The number of arithmetic operators (e.g., add, mul), the number of logic operators (e.g., or, shift)
4. The number/size of primary inputs and outputs
5. The number of registers, memory and multiplexers

6. Clock period

Power consumption is also included, since it is crucial when low-power hardware designs are the final target. The raw reports and files from both HLS and implementation phases are preprocessed and generated into two CSV files for each benchmark. Each CSV file contains multiple entries depending on the number of generated hardware designs for the benchmark. The user can directly use the data in the CSV files to train their ML models, thereby avoiding any effort in changing source code, setting up and running tools, and parsing reports. The detailed contents of the CSV files are listed in Table 4.2.

Category	***post_hls_info.csv* description**
Resource #	Estimated usage and available number of BRAM, DSP, FF and LUT
Clock	Target, estimation and uncertainty of the clock period
Logic ops	The number of C and RTL logic operators and associated resource usage
Arith ops	The number of C and RTL arithmetic operators and associated resource usage
Data ports	Width and the number of data input and output ports
Category	***post_implementation_info.csv* description**
Power	Simulation-based dynamic power consumption
Resource #	Actual usage of BRAM, DSP, FF and LUT
Clock	Achieved clock frequency

Table 4.2: Descriptions of features included in the CSV files provided with HLS-Dataset

It is possible that the features that other researchers are interested in, are not present in the CSV files. Therefore, tar balls are also created and they contain all the necessary files for feature extraction to do ML training. These files are selected according to how prior works generate their own dataset. Each tar ball contains:

1. Generated Verilog code (**.v*)

2. IR code (**.bc*)

3. IR operator information (**.adb*)

4. FSMD model (**.adb.xml*)

5. Resource usage estimation from HLS (**.verbose.rpt* and **.verbose.rpt.xml*)

6. Resource utilization reports (*utilization.xls*) and timing reports (*timing.xls*) generated after implementation

Considering the reusability and ease-of-use of the dataset, Tcl scripts and source code files are included in the dataset so that researchers can easily extend the dataset with other benchmarks. Verilog testbenches are also included so that the generated designs can be easily evaluated with simulation-based power estimation.

Overall, the contents of HLSDataset are summarized as:

1. The CSV files containing features for each hardware design listed in Table 4.2

2. Tcl templates and actual scripts to generate the dataset

3. C source code files manipulated with HLS pragmas

4. Testbenches in Verilog to test generated Verilog designs

5. Tar balls containing raw files and reports from HLS and reports from implementation stage

Compared to the datasets used by prior works, HLSDataset gives a wider coverage of information for each design, and it gives higher chance for researchers to use or extract useful features directly; meanwhile, no efforts are needed to run the time-consuming HLS and implementation tool flows.

4.2.2 Statistical overview of HLSDataset

Figure 4.3 provides a view of the diversity of the HLSDataset, through the resource usage metrics of the designs (or samples) contained in the dataset. Box and whisker plots are used to show the distribution of LUTs (Look Up Tables), FFs

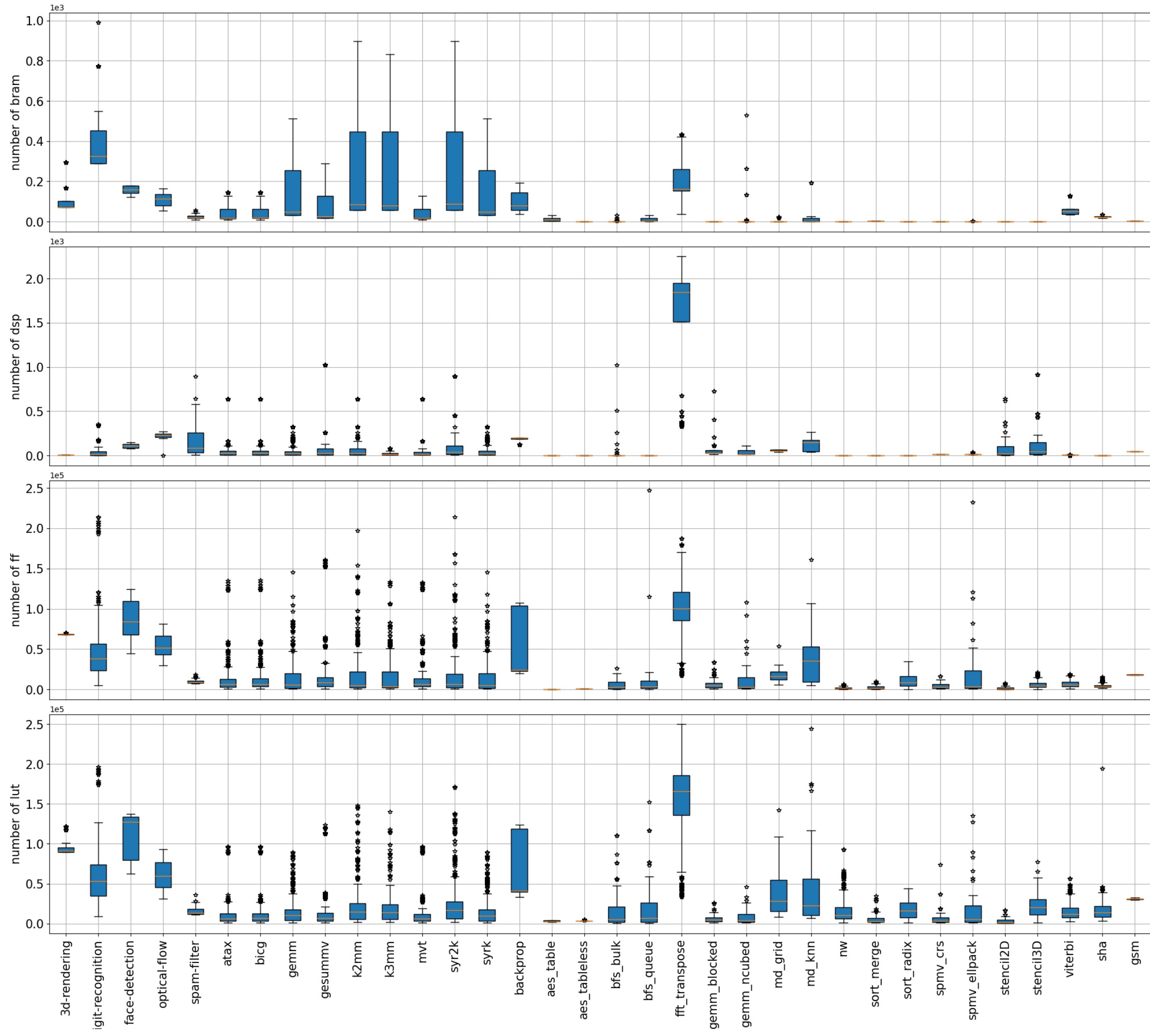


Figure 4.3: Resource utilization of designs generated for ZU9EG, applications are from Rosetta, Polybench, Machsuite and CHStone

Work	ML model	Task	C source	Feature and source
[44]	Lasso, XGB, ANN	Resource usage and timing	CHStone, Machsuite, S2CBench, Rosetta	Resource usage estimation for logic ops, arithmetic ops, memory and multiplexer; achieved clock period and uncertainty from **HLS reports**
[55]	GNN	Resource usage and timing	CHStone, Machsuite, Polybench	Graph samples based on **IR code**; operator type, used resource type and timing information from **HLS reports**
[36]	CNN	Power estimates	Polybench	Resource utilization and clock estimation by **HLS reports**; signal activities track and IR operator information from **IR code**; RTL operator information from **FSMD model**
[35]	GNN	Power estimates	Polybench	Signal activities track and IR operator information from **IR code**, Graph samples built with **IR code** and **FSMD model**, RTL operators information in **FSMD model**
[52]	GNN	Operation delay	Machsuite	Graph structures, operation type and bitwidth from **IR code**

Table 4.3: Prior ML-based prediction via HLS work, the used ML model, prediction tasks, the used dataset for training and the availability of the dataset.

(Flip Flops), DSPs (Digital Signal Processing Blocks) and BRAMs (Block RAMs) consumed by the designs generated from each application. As mentioned earlier, 4 widely used benchmark sets - Polybench, Machsuite, CHStone and Rosetta - are used to generate the dataset. Machsuite and Polybench are mainly composed of short programs and kernels, however, tuning the directives aggressively can still lead to large resource usage on FPGA. Rosetta, on the other hand, is composed of applications from ML and image or video processing domains. Each application of Rosetta contains multiple kernels, and it leads to larger resource usage on FPGA. The secure hash algorithm SHA and linear predictive coding analysis GSM are picked from CHStone due to their representative in the domain. CHStone arithmetic operation programs are not included due to the limited HLS design space in those applications.

4.3 HLSDataset Applications

HLSDataset can be applied to a multitude of prediction applications. Table 4.3 summarizes the prior works in the area of prediction at the HLS level. The data required for training ML models for each of these prior works is included in HLSDataset. Hence, HLSDataset can be effectively used for these and similar works.

Resource utilization estimates: HLSDataset can be directly used for post-implementation resource utilization estimates. Dai et al. [44] use Lasso linear model, XGB and artificial neural network (ANN) to improve the quality of HLS-generated resource utilization values with features extracted from HLS reports. Wu et al. [55] predict post-implementation resource usage by using the graph structure obtained from the IR codes generated by front-end of HLS tools. Fast estimation of resource usage find application in design space exploration while generating overlay architectures for FPGAs [88]. The features and feature source used to conduct the studies can be easily found and extracted from HLSDataset.

Timing and operation delay prediction: Wu et al. [55] demonstrates prediction of post-implementation critical path timing using IR codes and features from HLS reports. D-SAGE [52] builds graph samples using the IR generated during HLS and use them as input to GNNs to predict operation delay. HLSDataset contains the IR code files as well as HLS reports generated by Vivado HLS, and can be used to train such models to predict timing related information.

Power estimates: HL-Pow [36] and PowerGear [35] train ML models to predict power consumption using convolutional neural networks (CNNs) and GNNs respectively. Predicting power consumption needs data such as signal activities and operators obtained from the IR. While those signal activities are not directly included in HLSDataset, testbenches and stimulus are provided so that both RTL-level simulation and C-level simulation can be conducted. Necessary codes to run the simulation: IR codes and RTL designs are included in HLSDataset.

Beyond the above tasks, HLSDataset can be applied to many more potential

use cases. While the mentioned works target single-FPGA prediction, HLSDataset includes samples from multiple FPGAs. HLSDataset has the potential to be used for cross-FPGAs metric prediction, although no existing work shows this usecase. In addition to prediction of results, HLSDataset can be used to train models to optimize EDA tools and help on faster design space exploration. Moreover, HLSDataset can also be used to evaluate the ML model efficiency in HLS domain with the advancing of ML techniques.

The features and labels used by each ML models vary widely depending on the task and ML algorithm used from table 4.3. By including information from different levels in the CSV files and TAR balls in HLSDataset, all such ML models can be trained. Researchers can extract information from TAR balls and apply HLSDataset to many other applications.

4.4 Case Studies

HLSDataset covers large amount of features and metrics from post-HLS and post-implementation reports which can be used in machine learning models directly. Therefore, users can simply extract the necessary information from the dataset to train and test their models. In this section, two case studies are performed by training and testing ML models with HLSDataset to demonstrate the usage of it.

4.4.1 Case Study 1: Power Estimation in FPGA HLS via GNNs

In the first case study, the graph neural networks (GNNs) in PowerGear [35] are reproduced to predict the post-implementation power using both post-HLS features and signal information extracted from C-level simulation. Simulation power is ued as the ground truth power. The GNN models are trained and tested with the subset of HLSDataset on the same FPGA. IR code can be directly used to build graph samples which serve as the inputs to the GNNs. The usage of HLSDataset in this case study is shown in Figure 4.4.

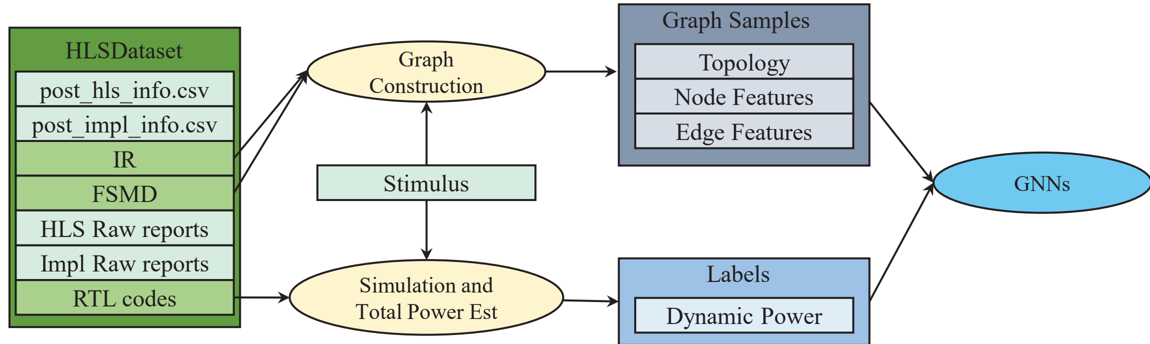


Figure 4.4: Usage of HLSDataset to construct machine learning based power model

Application	**Error of Dynamic Power (%)**	
	ZU9EG	**XC7V585T**
atax	3.89	5.25
bicg	3.90	5.60
gemm	5.24	6.50
gesummv	7.93	9.43
k2mm	4.25	6.00
k3mm	4.15	6.47
mvt	4.64	5.62
syrk	5.31	6.22
syr2k	6.41	6.46
average	**5.08**	**6.40**

Table 4.4: **Dynamic power estimation errors** - Training dataset and testing dataset are from Polybench subset of HLSDataset. Results for ZU9EG and XC7V585T.

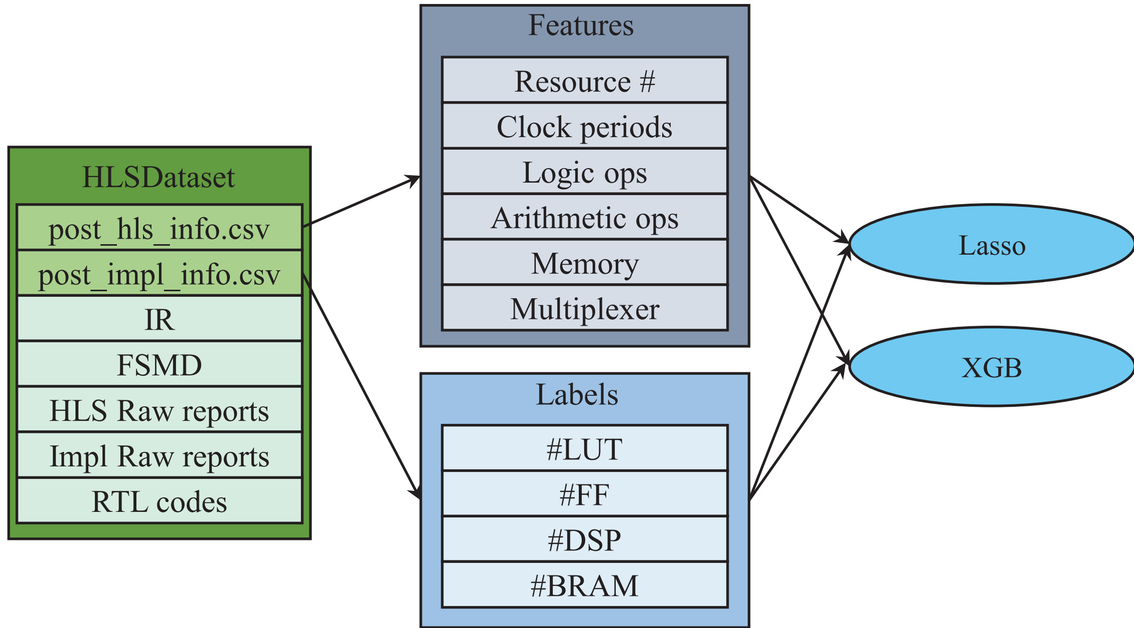


Figure 4.5: Usage of HLSDataset to construct machine learning model for estimation of resource utilization

The model is trained using the dataset from Polybench. One target application is left out of the nine applications as the test dataset and all the rest is used for training. With the iteration of the approach, one model is generated for every application. 10-fold cross-validation is used for model generation. All the above steps are repeated for the dataset from the other FPGA. All the training and testing run on Nvidia Ampere A100 GPU. The results for two FPGA devices, ZU9EG and XC7V585T, can be found in Table 4.4. The test errors for dynamic power range from 3.89% to 7.93% on ZU9EG and from 5.25% to 9.43% on XC7V585T, and the average errors are 5.08% and 6.40% respectively. The results show that HLSDataset can be used to perform ML-based power estimation tasks for FPGA.

4.4.2 Case Study 2: Estimation of Quality of Results in HLS with ML

The resource usage estimation (LUTs, FFs, DSPs, BRAMs) generated by HLS tools are fast but inaccurate compared to the post-implementation reports because

Resource	LUT	FF	DSP	BRAM
HLS Estimate	63.2%	34.1%	0.0%	1.8%
XGB	3.2%	2.3%	NA	0.1%
Lasso	13.2%	15.4%	NA	NA

Table 4.5: **Resource estimation errors** - Training dataset and testing dataset are from Machsuite and Polybench subsets of HLSDataset. Results for ZU9EG.

HLS tools simply sum up the contributions of instantiated functional units during the synthesis. This approach fails to capture the optimization effects and limitations imposed by resources on-chip. However, as [44] indicates, ML can help to predict more-accurate resource usage from estimates in the HLS reports.

The ML model is replicated but HLSDataset is used as training and test set to evaluate the ML model on estimation of post-implementation resource usage. The way to use HLSDataset is illustrated in Figure 4.5. Machsuite, Polybench subsets from HLSDataset are used to train the XGB and Lasso linear model. The features are extracted from FSMD file (.adb.xml) and resource estimates reports (.verbose.rpt.xml). The ground-truth resource utilization is extracted from post-implementation reports. All the files and reports are included in the dataset, only a parser is needed to extract necessary data to be used in the ML model. In this case, the single-task XGB and Lasso models are used. Random 20% of 8735 samples from the subsets are selected as the testing set and the rest are selected as the training/validation test set. 10-fold cross-validation is performed during training, and 75% of the training/validation set is selected for training and 25% for validation. The results are shown in Table 4.5. The HLS tool fails to provide good estimates for LUT and FF usage, while DSP and BRAM estimates are accurate. XGB and Lasso demonstrate a significant accuracy improvement in the estimation of LUT and FF usage. The results shown in this table differ from those in the original paper because there are differences in target FPGA, the dataset, features used to train the model and the version of HLS tools used for the dataset generation. Therefore, a comparison with the original work is not shown here.

4.5 Summary

This work presents HLSDataset, a dataset for ML-assisted FPGA design using HLS. HLSDataset covers a wider range of data than other datasets in this domain, and is the first open-source dataset of its kind that can be used for multiple studies. HLSDataset can be used in training ML models targeting different applications such as resource usage prediction, power prediction, etc. The methodology to generate the dataset is presented so that HLSDataset can be futher extended.

HLSDataset can be expanded by including data for more target frequencies (e.g. clock period = 5ns, 2.5ns, etc.). For future work, HLSDataset will be extended to include more benchmarks (e.g., S2CBench) and more FPGAs (including Intel FPGAs). While the design samples in HLSDataset are generated from C benchmark so that ML-assisted HLS based studies can be conducted, the dataset will be extended to include data from native Verilog benchmarks so that ML-assisted Verilog based studies are possible with the dataset.

Chapter 5: XPNet: Cross-FPGA Power Prediction from High Level Language Code

High Level Synthesis (HLS) [5] is an automated flow to transform an application written in a high-level language, such as C/C++ or Python, to a register-transfer level (RTL) description. Then, another automated process including logic-synthesis, placement and routing converts the RTL into an implementation suitable for field-programmable gate arrays (FPGAs). HLS is composed of two phases: front-end and back-end. The front-end compiles the C/C++ and HLS pragmas into HLS LLVM Intermediate Representation (IR) codes. The back-end will then make FPGA-specific optimizations on the IR code. Resource allocation, operator scheduling and binding happen in this stage. Although HLS is faster, the logic synthesis, placement and routing phases are much slower, which hinders quick and broad design space exploration. In addition, evaluating power involves writing stimulus and performing simulations, which requires even more time and effort.

Table 5.1: The features used by prior power models vs XPNet

	HLS frontend		HLS backend	
	IR	toggling	binding info	scheduling info
Prior work [35]	✓	✓	✓	✓
XPNet	✓	✓	✗	✗

To enable accurate and fast estimation of power, prior work employed Machine Learning (ML) techniques to bypass these time-consuming phases [35, 36]. ML models are trained with features extracted from the HLS phase and labels from simulation-based power consumption. Using these ML models, post-implementation power can be inferred right after HLS stage.

With the objective to enable fast adaption from a base learner (power model on one FPGA) to a target learner (power model on the other FPGA, even from

another vendor) with very limited dataset size, a novel cross-FPGA power modeling methodology XPNet is proposed. By utilizing only HLS frontend features, a GNN-based power model is built and it can be easily transferred between FPGAs, including FPGAs from different vendors. XPNet also introduces an innovative power data selection technique that enables efficient and fast ML-based power model adaption. The contributions in this work can be summarized as follows:

- XPNet is the first methodology which is capable of constructing an ML model that accurately estimates dynamic power **across** FPGAs. By employing only the features from HLS frontend as indicated in Table 5.1, XPNet constructs models that can be transferred between FPGAs, even those from different vendors.

- A GNN-based power model is proposed and it utilizes only HLS front-end features to predict power and the model can be adapted to FPGAs from different vendors.

- A design selection method is poposed to select a small set of designs for fine-tuning with the help of a bridge FPGA. With these selected power-informative designs, the model after transfer-learning is able to provide a better result than randomly picking designs.

- With XPNet, ML-based models can be efficiently constructed and they are able to produce 8.40% average error with pretrained model on one device and 20 designs on the other target device within AMD/Xilinx vendor family (Xilinx-to-Xilinx). XPNet further shows a 10.34% average error predicting using a pretrained model on one AMD/Xilinx device and 20 designs on a target Intel device (Xilinx-to-Intel).

- The time-to-result, that is, the total time involved in generating the dataset, training the model, and performing predictions is evaluated. XPNet achieves a

speedup of 232x compared to a non-fine-tuning based approach and a speedup of 5x in power inference compared to prior work.

5.1 Problem Formulation

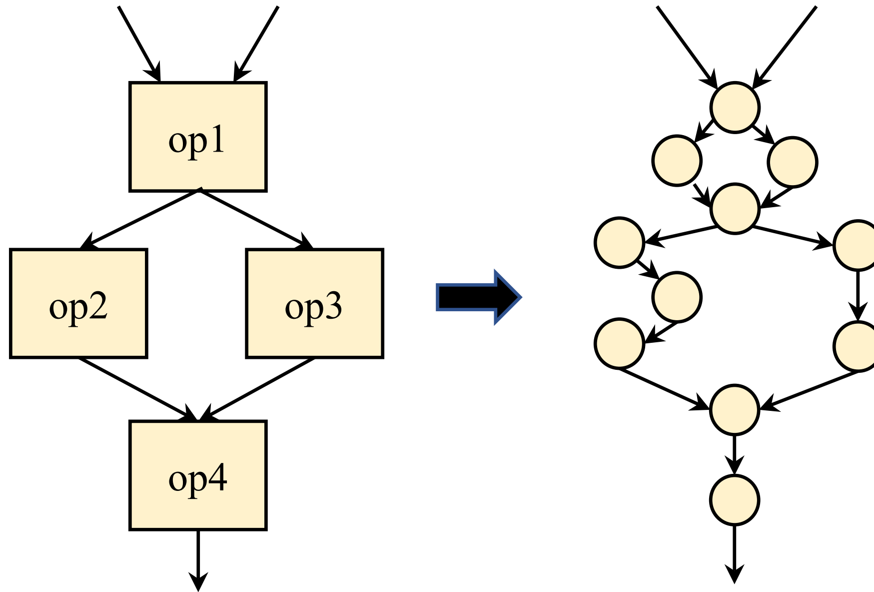


Figure 5.1: Graph generated after HLS (left) and graph generated after implementation (right)

In the flow of mapping C applications to an FPGA, two main graphs are generated by the High Level Synthesis (HLS) and implementation (i.e. placement and routing) tools. As illustrated in Figure 5.1, the coarse-grained high-level operator graph is generated after HLS, and each node represents one C-level operator and each edge represents the data flow between operators. The fine-grained or circuit-level graph is generated after implementation, and each node represents one logic gate and each edge represents the wire between two logic gates. The ground truth power is calculated based on the fine-grained graph with the Equation 5.1:

$$P_{dyn} = \sum_{i \in I} \alpha_i C_i V^2 f \tag{5.1}$$

In the equation, α_i is the signal switching activity, C_i is the interconnect capacitance, V is the supply voltage, f is the operating frequency and i is an interconnect of the whole set I. For a specific FPGA, the supply voltage V is fixed, and C for each i is decided by the FPGA resources type and placement and routing algorithms.

While different mapping algorithms and optimization strategies can lead to different circuit-level graphs generated from the same high-level graph, it is assumed that the optimization and mapping algorithm are fixed in the problem and dataset generation, therefore, each HLS-level graph can only generate one unique circuit-level graph.

Formally, let m be the total number of HLS-level graphs in the training set. Define $X = (x_1, \cdots, x_m)$ and $Y = (y_1, \cdots, y_m)$, where $x_j \in (\mathbb{R}^d, (V_j, E_j, R_j))$ denotes the global metadata and graph data obtained and preprocessed from HLS-level graph, V_j denotes set of nodes, E_j denotes set of edges and R_j denotes set of relation types on all edges, $\{\forall e \in E_j, \forall v \in V_j, \forall r \in R_j, e, v \in \mathbb{R}^d, r \in \mathbb{R}\}$, and $y_j \in \mathbb{R}$ denotes the power obtained from circuit-level simulation. The goal is to find a hypothesis $g : \mathbb{R}^d, (V, E, R) \rightarrow \mathbb{R}$ so that the mean absolute percentage error $err = \frac{1}{m}\sum_{i=1}^{m} |\frac{g(x_i)-y_i}{y_i}|$ is minimized. It can be intuitive to see from Equation 5.1 that the power is a linear combination of signal activities α_i and interconnect capacitance C_i and the task can be defined as a regression problem. However, features extracted from HLS-level graph are not necessarily linear mapping to α_i and C_i. Therefore, the solver should not be restricted to be linear.

It is worth mentioning that the above (X, Y) and the outcome hypothesis g are specific to one FPGA. Even though the HLS-graph for the new FPGA is not subject to change, the ground-truth power can still be vastly different due to the great difference on voltage supply V, capacitance C_i and achieved frequency f on a different FPGA. Therefore, given (X', Y'), the hypothesis g cannot achieve the same accuracy as it does on (X, Y). Intuitively, the task to solve on (X, Y) should be quite similar to the task on (X', Y'). The objective of this work is to find a hypothesis g'

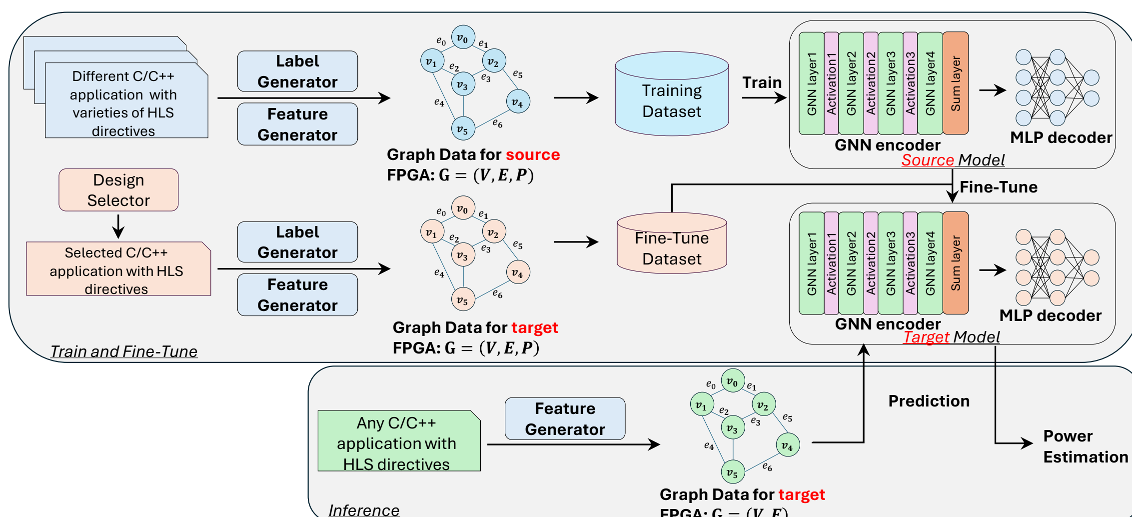


Figure 5.2: Overview of XPNet. Train and Fine-Tune (top). Inference (bottom)

with as few as possible data samples drawn from (X', Y') given pretrained model g on (X, Y) without much accuracy loss.

5.2 The XPNet Framework

Figure 5.2 depicts a high-level overview of XPNet which is a machine learning framework for cross-FPGA power prediction. In *Train and Fine-tune*, shown in the top part of the figure, the dataset is formed with the help of **Label Generator** and **Feature Generator** so that each sample can be directly digested by a regression ML model. The source FPGA model is trained with this dataset. With the **Design Selector**, designs that are the most power-informative (described in section 5.2.1.3) form the fine-tune dataset. The source model is fune-tuned to generate the target model. In *Inference*, shown in the bottom part of the figure, Feature Generator is used to generate features and these features are fed into the target FPGA model produced during *Train and Fine-tune* to predict the power for target FPGA.

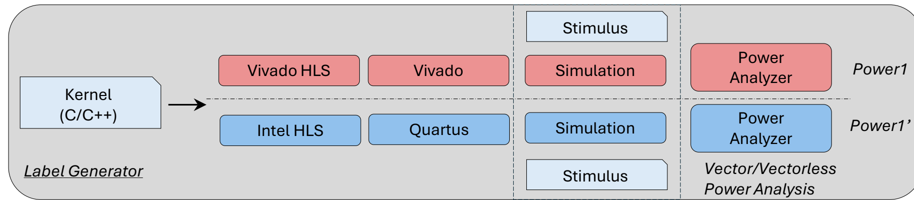


Figure 5.3: Detailed view of the Label Generator

5.2.1 Train and Fine-Tune

Train and Fine-Tune contains three important components: Label Generator, Feature Generator and Sample Selector.

5.2.1.1 Label Generator:

As shown in Figure 5.3, the Label Generator is essentially composed of HLS, place and route, simulation and power analyzer from different vendors. While the ground-truth average dynamic power can be collected from either vector-based power analysis or vector-less power analysis, correct toggling features which will be discussed in **Feature Generator** are included. In order to generate sufficient designs, Polybench and Machsuite subsets from HLSDataset[38] are used and the provided scripts are executed to extend the dataset to cover more FPGAs. Specifically, pragmas are applied to each kernel in the benchmark suites, - *array_partition loop_unrolling* and *loop_pipeline* - from the AMD/Xilinx tool suite to generate varieties of HLS designs. Equivalent or similar pragmas from Intel HLS compiler - *hls_numbanks*, *unroll* and *disable_pipeline* - are used for generating designs for an Intel FPGA. With this method, a variety of designs are generated for each FPGA.

5.2.1.2 Feature Generator:

The detailed overview of Feature Generator is illustrated in Figure 5.4. The open-source Vitis HLS frontend is used to compile the kernel code with pragmas into LLVM IR. The dataflow graph (DFG) is recovered with the LLVM IR. If a stimulus

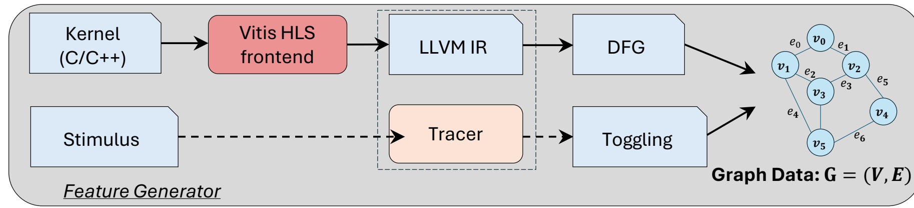


Figure 5.4: Detailed view of the Feature Generator

is provided, the IR code is annoated with a trace function to print out the toggling of each IR operator during the execution. In this way, the toggling behavior on each port of the IR operator is recorded. Since the different operators can consume different power even with the same toggling behavior, the collected toggling behavior is categorized accordingly. Up to 21 types of operators including logic operators, arithmetic operators, arbitration operators as well as memory operators from HLS are selected. They are summarized as follows:

- Arith: add, sub, mul, div, sqrt, fadd, fsub, fmul, fdiv, fsqrt
- Logic: and, or, xor, icmp, fcmp
- Mem: store, load, read, write
- Arbit: mux, select

The specific method to record and trace IR operator toggling activities has been sufficiently discussed by Chapter 3. the graph data is created using the DFG and toggling activities. The power-aware design representation is composed of two parts. The first is the design itself including the number and types of operators and the path to link these operators. The second component is the behavior of these operators which is normally represented as switching activities. The design generated by HLS tools, however, is determined by two input components. The first one is a high-level program description, expressed in C/C++, which define the semantics and functionality. The second is a set of pragmas that include parallelizing, pipelining

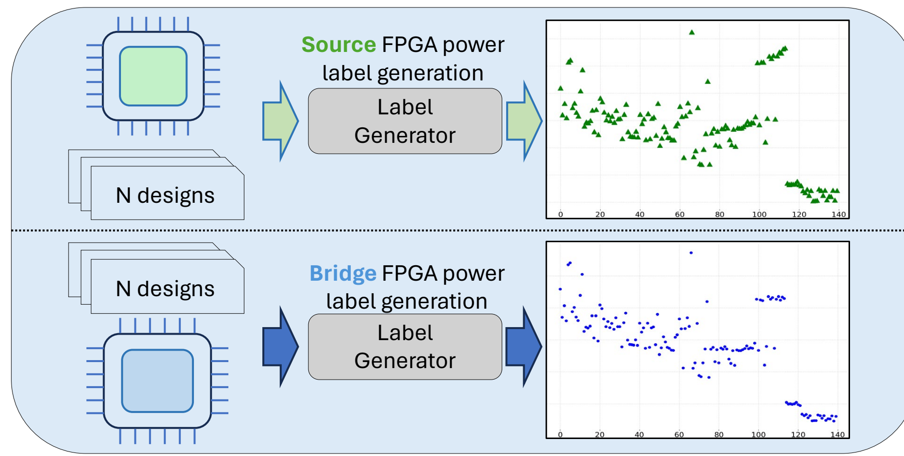
and array partitioning. Therefore, in order to correctly represent the two factors, LLVM IR codes generated by HLS tools are chosen. Similarly in [35, 37], the data flow graphs (DFGs) with the LLVM IR codes are generated from HLS tools and switching activities are obtained from IR-level simulation. Specifically, given a graph $G = (V, E)$, where V and E represent the node and edge set, respectively. Every node in the graph $\forall v \in V$ represents an IR operator with attributes that include opcode type, bit width, input, and output switching activities. Every edge in the graph $\forall e_{i,j} \in E$ where $e_{i,j}$ is the edge with i as the source and j as the sink. The edge $e_{i,j}$ contains switching activities $SA_{i,j}$ and the activation ratio $AR_{i,j}$:

$$SA_{i,j} = \frac{\sum HD(v_i(k), v_i(k-1))}{L}, \quad AR_{i,j} = \frac{N}{L} \tag{5.2}$$

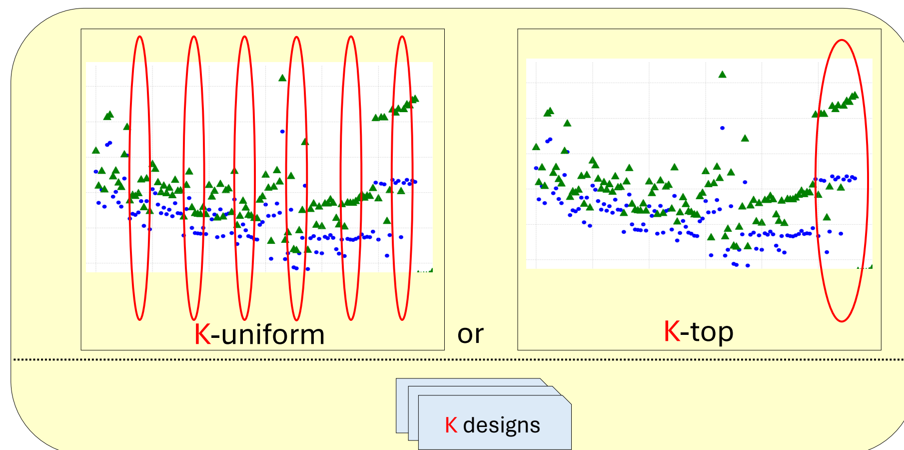
where HD refers to hamming distance, L refers to the latency of the design and N refers to the number of execution cycles that cause the change of the vertices v_i. The HD inside SA accumulates in every cycle when the vertices change. In addition to these two switching features, the edge type is also encoded and added to the edge attributes.

5.2.1.3 Design Selector:

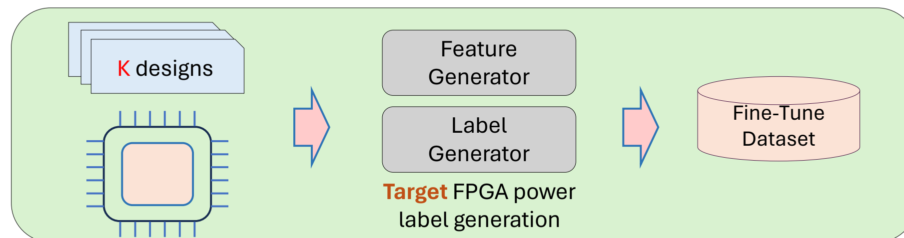
The detailed view of the Design Selector is shown in Figure 5.5. Different from common data pruning methods which pick samples from the whole dataset,the aim is to find the K designs that are power-informative for the target FPGA without generating the whole dataset for the target FPGA. Achieving this goal requires several steps in the Design Selector: *Step 1:* Generate abundant data samples for source FPGA and an extra FPGA (bridge FPGA) with the same designs (same C/C++ codes and HLS pragmas). *Step 2:* Compare the data samples from source FPGA and bridge FPGA to identify the K representative designs. *Step 3:* Use those designs to generate fine-tune data samples for target FPGA. Within the *Step 2*, the relative power difference of each design is caculated on the source FPGA and the bridge



Step 1: generate power labels for the same set of designs on two FPGAs



Step 2: compare data samples and pick K designs



Step 3: generate fine-tune dataset with K designs for **target** FPGA

Figure 5.5: Detailed view of the Design Selector for fine-tuning dataset

FPGA as:

$$d = \frac{|P_{source} - P_{bridge}|}{P_{source}} \tag{5.3}$$

With the power difference metrics, two different methods are used to identify K designs. The ***K-top*** method sorts the designs according to absolute relative power difference $|d|$ and picks K designs with largest power difference value. The K designs are then used to generate data samples from target FPGA. On the other hand, the ***K-uniform*** method sorts the designs according to relative power difference d and picks K designs evenly distributed within the range of the power difference, e.g., if the power difference of the designs range from -10% to 10% and 4 designs are desired, one design is randomly picked from each chunk of $[-10\%, -5\%]$, $[-5\%, 0\%]$, $[0\%, 5\%]$, $[5\%, 10\%]$. The final model generated by XPNet can then be used directly to predict power for a vast array of designs on the target FPGA.

5.2.2 Inference

The process in *Inference* is straightforward; the Feature Generator is used to generate features for the design. The stimulus can be included for the kernel so that the Feature Generator can take the toggling info from running the kernel. Alternatively, users can enter the preferred constant toggling rate into the Feature Generator. These features are fed into the target FPGA model which is produced in the *Train and Fine-tune* step to generate the average power prediction for the design. XPNet only requires kernel program with pragmas and stimulus running on that kernel to predict the average power.

5.2.3 Model

The model is composed of a GNN encoder and an MLP decoder for regression task. The GNN encoder can help to better extract the inherent information of the design since edge attributes, including switching features, are essential to perform power prediction. However, both GCN [89] and GAT [90] overlook the edge

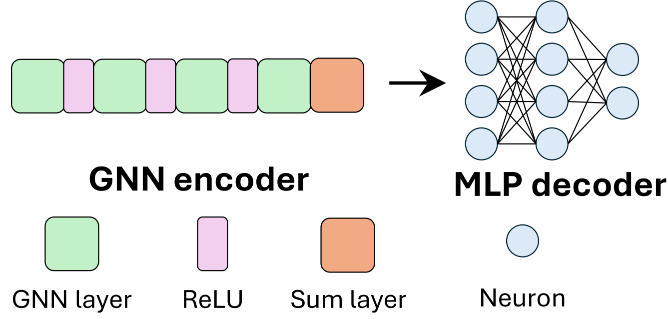


Figure 5.6: Model Architecture of XPNet

embeddings. Therefore, a more edge-expressive model [35] is adapted for the power estimation task. The architecture of the model is depicted in Figure 5.6. The GNN encoder contains 4 GNN layers, 4 activations and 1 sum layer. At GNN layer, each node collects information from its neighbors and node embeddings are updates with the equation defined as:

$$h_i^{(l)} = ReLU(W_1^{(l)} h_i^{(l-1)} + AGG^{(l)}) \tag{5.4}$$

where $h_v^{(l)}$ is the embedding vector of node i at layer $W_1^{(l)}$ is the learnable weight matrix for updating node embeddings at layer l. $AGG^{(l)}$ represents the aggregation function defined as:

$$AGG^{(l)} = \sum_{r \in R} \sum_{j \in N(i)} W_2^{(l)} W_r^{(l)} e_{i,j} \tag{5.5}$$

where $W_2^{(l)}$ is a learnable weight matrix at each layer, $e_{i,j}$ represents the edge pointing from vertices v_i to v_j, $W_r^{(l)}$ is relation-specific weight according to the type of v_i and v_j. Each vertices belong to arithmetic operator (A) or non-arithmetic operator (N). Therefore, there are 4 relations in R (A$\rightarrow$N, N$\rightarrow$A, N$\rightarrow$N, A$\rightarrow$A) and hence 4 weight matrix of $W_r^{(l)}$ at each layer. The graph level embedding is generated by the sum layer:

$$h_G = \sum_{l \in L} \sum_{v \in V} h_v^{(l)} \tag{5.6}$$

where L is the set of indexes of GNN layers and V is the set of vertices in the graph. The aggregation of all nodes across layers can enhance the generalization

ability of the model. The graph level embeddings are then fed to the MLP decoder to perform power regression:

$$P_{est} = MLP(h_G) \tag{5.7}$$

During fine-tuning, the GNN encoder is frozen and only the MLP decoder is updated to improve the efficiency.

5.3 Experimental Setup

5.3.1 Devices

As Table 5.2 shows, five devices are picked from three families of Xilinx: 7 series, Ultrascale and Ultrascale+. Two devices from Intel are selected: Stratix 10 TX and Arria 10 GX. As can be seen in Table 5.2, these devices are quite different from each other and they should be sufficient to demonstrate the effectiveness of XPNet.

Table 5.2: Overview of characteristics of FPGAs used in experiments

AMD/Xilinx					
Name	**Device**	**Tech node**	**#LUTs**	**#BRAMs**	**#DSPs**
FPGA1	xczu9eg	16nm	274,080	912	2,520
FPGA2	xc7v585t	28nm	364,200	795	1,260
FPGA3	xcvu440	22nm	2,532,960	2,520	2,880
FPGA4	xc7k480T	28nm	298,600	955	1,920
FPGA5	ku115	20nm	663,360	2,160	5,520
Intel					
Name	**Device**	**Tech node**	**#ALMs**	**#M20K**	**#DSPs**
FPGA6	S10TX	14nm	449,280	5,461	2,592
FPGA7	A10GX	20nm	339,620	2,423	1,518

5.3.2 Dataset

The Feature Generator and Label Generator are used to form the training dataset. AMD/Xilinx Vivado HLS 2018.3, Vivado 2018.3, quartus 2019 are used to synthesize and simulate the design to collect the design features and ground-truth

power. Similar to the method in PowerGear [35], more than 5000 designs are generated for each FPGA using multiple kernels from different benchmark suites including Polybench[86], Machsuite[48] and CHStone[47]. Details are summarized in Table 5.3.

Table 5.3: Benchmark kernels used for the dataset

Benchmark suite	**Kernel application**	**# Designs**
Polybench	`atax`, bicg, gemm, gesummv, k2mm, k3mm, mvt, syrk, syr2k	4779
Machsuite	`spmv_crs`, `stencil3d`, `stencil2d`	376
CHStone	`gsm`, `sha`	236

The method is further extended to include designs for Intel FPGAs. `atax`, `spmv_crs`, `stencil3d`, `stencil2d`, `gsm` and `sha` are split out from the dataset to form the testing set. Models are tested with the whole testing set unless specified in the section. The rest of the dataset is used for training. The post-implementation resource utilization and latency for the Xilinx designs is illustrated in Figure 5.7 and Intel's is shown in Figure 5.8. These designs are used to generate the complete dataset for the experiments.

While the testing set is kept alone, 10-fold cross-validation setting is used across the training dataset during training. All the results are obtained with the evaluation using the test dataset from the target FPGA which is split out in advance. the model performance is evaluated in terms of the mean absolute percentage error (MAPE) unless indicated otherwise in the subsections. The default Design Selector is K-top unless otherwise specified. Mean Absolute Error Percentage (MAPE) is used to evaluate the prediction accuracy:

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100\%, \tag{5.8}$$

where n refers to the number of samples, y_i is the actual power and $\hat{y}_i$ is the predicted power of i_{th} design, $\bar{y}$ is the mean of the actual power.

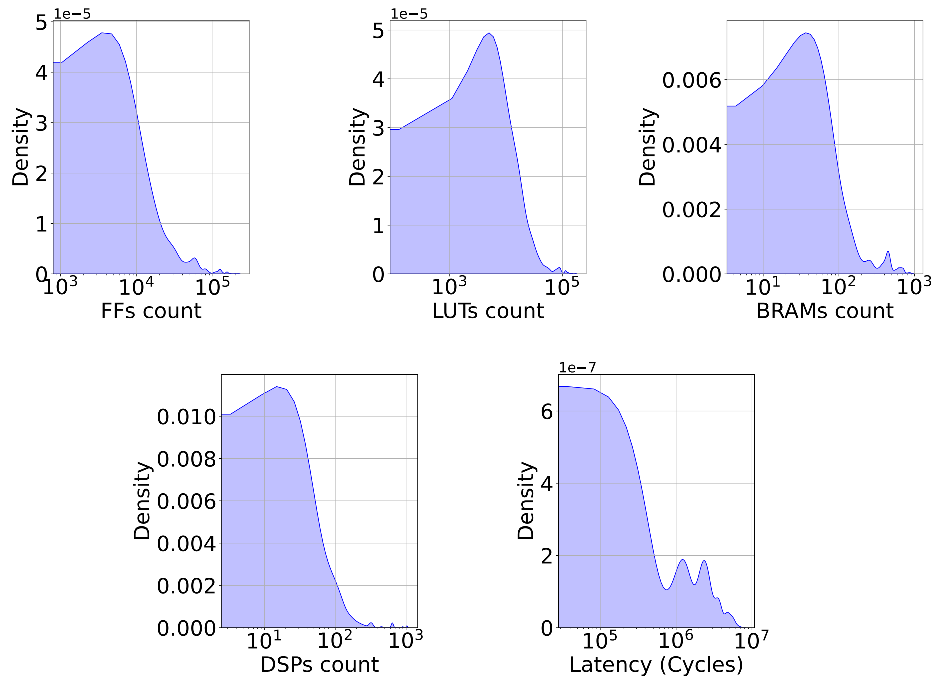


Figure 5.7: Resource utilization and latency for the *Xilinx* designs

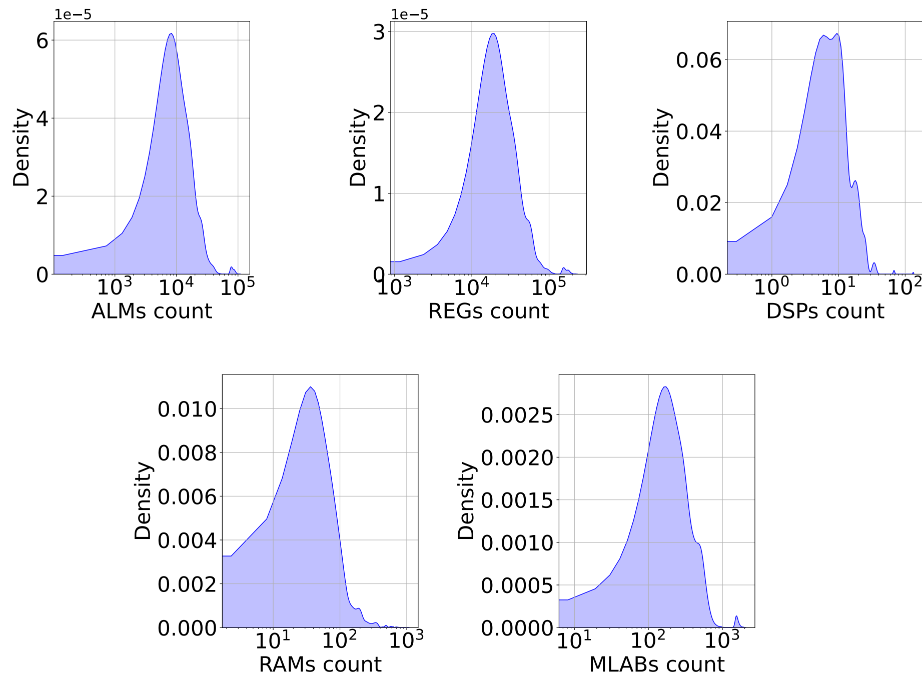


Figure 5.8: Resource utilization for the *Intel* designs

5.4 Results

5.4.1 How effective is Transfer-Learning in XPNet?

In this section, the effectiveness of XPNet is shown by comparing it to ordinary (from scratch) training. As shown in Table 5.4, two comparison experiments are conducted. All models in this section are tested with the designs from `atax`. In the first row, Scratch(3) refers to training the model from scratch with the FPGA3 dataset and testing on FPGA3 dataset, and XPNet(1-3) means training the source model with FPGA1 dataset and fine-tune the source model for FPGA3. FPGA2 is used as the bridge FPGA in these experiments. For various numbers of designs, scratch training uses them for training while XPNet uses them to fine-tune the pre-trained model. Each row contains the model error for the two different methods.

The error remains similar for XPNet over scratch training when the dataset size is large, because a sufficient quantity of designs from the target FPGA can provide good generalization. However, improvements become more obvious when the dataset size decreases since the limited number of designs from the target FPGA fail to fit the power model, when training from scratch. With XPNet, prior knowledge from the source FPGA in the model helps the model to stay accurate even when using only a small number of designs from the target FPGA.

Table 5.4: XPNet vs. train from scratch, model error(%)

# Designs	Scratch(3)	XPNet(1-3)	Scratch(4)	XPNet(1-4)
20	101.23	8.75	110.63	8.54
50	98.13	8.31	101.32	8.42
100	79.31	8.01	78.01	7.98
200	58.39	7.85	63.32	7.56
300	38.76	7.67	39.19	7.01
400	25.09	7.34	24.43	7.21
500	16.32	7.01	15.13	6.87
1000	10.14	6.76	10.34	6.21
2000	7.86	6.34	8.19	6.13
4000	6.13	6.02	6.45	6.10

Table 5.5: Models on AMD/Xilinx devices as source model and XIlinx or Intel as target (GNN)

Xilinx-to-Xilinx Transfer model error (%) (s=source, b=bridge, t=target)								
Target	**s1b2**	**s1b3**	**s1b4**	**s1b5**	**s2b1**	**s2b3**	**s2b4**	**s2b5**
t1	6.49	6.49	6.49	6.49	NA	8.23	8.54	8.13
t2	NA	8.54	8.87	8.91	6.40	6.40	6.40	6.40
t3	8.75	NA	8.53	8.56	8.35	NA	8.13	8.43
t4	8.54	8.67	NA	8.54	8.24	8.66	NA	8.74
t5	8.23	8.23	8.32	NA	8.02	8.12	8.07	NA
Target	**s3b1**	**s3b2**	**s3b4**	**s3b5**	**s4b1**	**s4b2**	**s4b3**	**s4b5**
t1	NA	8.71	8.14	8.32	NA	8.22	8.33	8.32
t2	8.72	NA	8.13	8.23	8.25	NA	8.65	8.54
t3	6.13	6.13	6.13	6.13	8.34	8.24	NA	8.43
t4	8.32	8.34	NA	8.84	6.45	6.45	6.45	6.45
t5	8.45	8.13	8.31	NA	8.05	8.22	8.23	NA

Xilinx-to-Intel Transfer model error (%) (s=source, b=bridge, t=target)								
Target	**s1b2**	**s1b3**	**s1b4**	**s1b5**	**s2b1**	**s2b3**	**s2b4**	**s2b5**
t6	9.32	10.12	9.98	10.01	10.01	10.45	10.15	10.23
t7	10.12	10.22	10.17	10.29	10.13	10.23	10.33	10.39
Target	**s3b1**	**s3b2**	**s3b4**	**s3b5**	**s4b1**	**s4b2**	**s4b3**	**s4b5**
t6	10.23	10.13	10.99	10.88	10.33	10.41	10.74	10.41
t7	10.89	10.43	10.87	10.65	10.57	10.65	10.13	10.31

5.4.2 How effective is XPNet in predicting across devices?

There are two specific questions regarding the generalization of XPNet: *How is the performance of the transferred model impacted when using different source and target FPGAs? How does XPNet perform when source FPGA and target FPGA are from different vendors?* In order to answer these questions, a set of experiments are conducted with different combinations of FPGAs as source, bridge and target. The results of these experiments are shown in Table 5.5 and Table 5.6 where s1b2 means FPGA1 is used as the source FPGA and FPGA2 is used as the bridge FPGA and t3 means that FPGA3 is taken as the target. The model error number when the target and source are the same is underlined; these models do not need a fine-tuning phase. All the other cases require 20-sample fine-tune datasets.

Same-board performance: the model is first tested on the same FPGA and the results are shown with the underlined number in the table. In this case, model is

Table 5.6: Models on Intel devices as source models (GNN)

Intel-to-Intel Transfer model error (%) (s=source, b=bridge, t=target)										
Target	**s6b1**	**s6b2**	**s6b3**	**s6b4**	**s6b7**	**s7b1**	**s7b2**	**s7b3**	**s7b4**	**s7b6**
t6	7.31	7.31	7.31	7.31	7.31	8.23	8.12	7.98	8.01	NA
t7	7.79	8.13	8.10	8.21	NA	8.18	8.18	8.18	8.18	8.18

Intel-to-Xilinx Transfer model error (%) (s=source, b=bridge, t=target)										
Target	**s6b1**	**s6b2**	**s6b3**	**s6b4**	**s6b7**	**s7b1**	**s7b2**	**s7b3**	**s7b4**	**s7b6**
t4	10.45	10.78	10.98	NA	10.87	10.31	10.14	10.69	NA	10.13
t5	10.14	10.65	10.42	10.88	11.12	10.09	10.63	11.03	10.86	10.79

trained and tested on the same FPGA. It shows averagely 6.37% error on 4 Xilinx devices and 7.75% error on 2 Intel devices.

Xilinx-to-Xilinx: when both the source and target FPGA are from Xilinx series, the transferred model error can achieve 8.40% in average. With the same settings of source and bridge FPGA, the model performance is within 1% difference with different target FPGAs. It is also worth mentioning that every FPGA is in a different technology node from each other as indicated in Table 5.2. There are several "NA"s in the **t5** row. **b5t5** is not considered as a valid setting since it requires the whole dataset of the target FPGA which is not the purpose of these experiments. *Xilinx-to-Intel:* When the source FPGA is from Xilinx and target FPGA is from Intel, the transferred model error can achieve 10.34%. A performance drop is observed when the target FPGA is from a difference vendor because when the pre-trained model is created with Xilinx FPGAs and the circuit features are captured from the front-end IR codes which are generated by xilinx-hls-llvm. Therefore, these circuit features can correctly reflect the circuit generated by Xilinx tool chains. Intel tool chains, however, may have different strategies/algorithms in high-level-synthesis so that the circuits generated by Intel cannot be the same as Xilinx even though the same HLS options are used in both cases. Hence some drop in model performance is understandable. *Intel-to-Intel:* When both the source FPGA and target FPGA are from Intel, the transferred model error can achieve 8.07% in average, which is quite close to the results gained from Xilinx-to-xilinx experiments. *Intel-to-Xilinx:* When the source FPGA is from Intel and target FPGA is from Xilinx, the transferred model error

can achieve 10.59% where a slight performance degradation happens due to the large difference between two vendors.

Bridge FPGA effect: The bridge FPGA is used when it is necessary to select designs that are used to fine tune the model for the target FPGA, the use of bridge FPGA is to identify the designs that make a difference in the power on two different FPGAs so that those power-informative designs can fine-tune the model better. Therefore, which bridge FPGA to pick is also important. When all the three FPGAs are from the same vendor, the performance of XPNet is quite stable. However, when the source and target FPGAs are from different vendors, the results with the bridge and target FPGA from the same vendor are always better than those with bridge and target FPGAs from different vendors (e.g., **s6b1t4** vs. **s6b7t4**). In order to achieve the best performance, it is the best to pick a bridge FPGA which is similar to the target FPGA.

In summary, with different combinations of source and bridge FPGAs, XPNet performance is not largely affected. XPNet can perform better when the source and target FPGA are from the same vendor while it maintains good performance even when they are from different vendors.

5.4.3 How much can the Design Selector help?

Table 5.7: Model Error(%) for XPNet with different Design Selectors

	XPNet (1-3)		**XPNet (1-4)**	
# Designs	**K-top**	**K-uniform**	**K-top**	**K-uniform**
20	8.75	10.13	8.54	10.88
50	8.31	9.98	8.42	10.23
100	8.01	9.32	7.98	9.36
500	7.01	8.35	6.87	7.56
1000	6.76	7.13	6.21	6.93
2000	6.34	6.89	6.13	6.23
4000	6.02	6.02	6.10	6.10

In this section, the performance of the two design selection methods is eval-

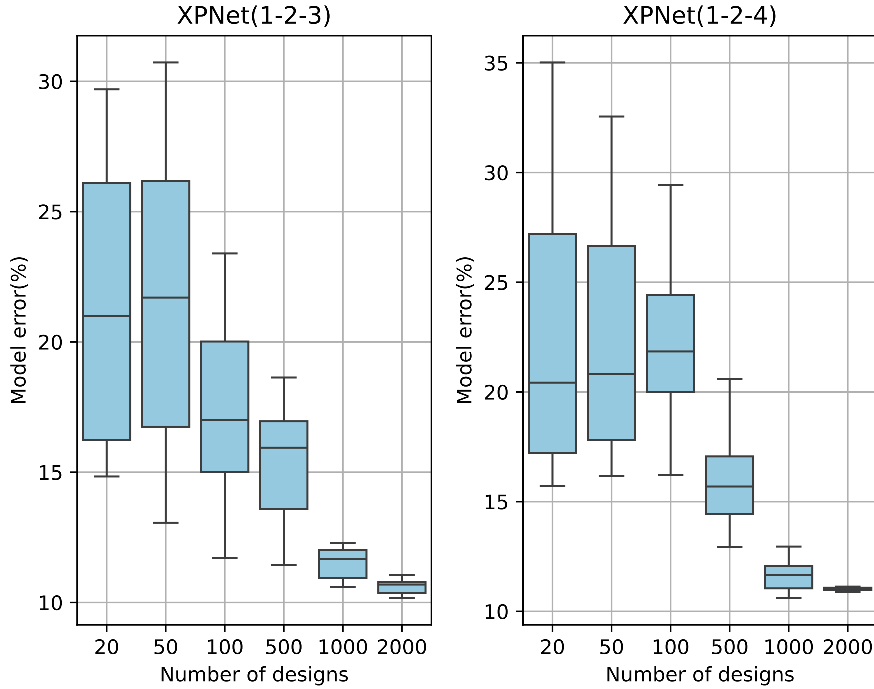


Figure 5.9: XPNet with the fine-tune dataset created by *random design selection*

uated in XPNet. A fine-tune dataset is formed with two different selectors: K-top selector, and K-uniform selector. They are evaluated with the `atax` test set from the target FPGA and their results are shown in Table 5.7. A random selector is implemented as the baseline to imitate the case where transfer-learning is applied without any design selection. The random selection is repeated 20 times and generate 20 different fine-tune datasets to fine-tune the models. They are evaluated with the same test set from the target FPGA and results are illustrated in Figure 5.9.

5.4.4 Comparison on different ML Models on XPNet

While K-top and K-uniform generate consistent fine-tune dataset every time and the model performance is quite stable, random selector cannot guarantee the same dataset and it is observed that the model performance can vary with different fine-tune dataset. The error increases for all three methods when the number of designs decreases. While the difference between the K-uniform and K-top is slight,

Table 5.8: The table including features being used in the MLP model

Type	S-type	Toggling	Clock(ns)
fmul	Arith	[1.505, 0.761, 1.569]	10
fmul	Logic	[3.305, 1.732, 0.259]	10
fmul	Mem	[2.345, 0.921, 0.337]	10
fmul	Arbit	[1.305, 0.513, 3.421]	10
fadd	Arbit	[1.533, 1.569, 1.539]	10
...	...	...	..

Table 5.9: Models on AMD/Xilinx devices as source model and Xilinx or Intel as target (MLP)

Xilinx-to-Xilinx Transfer model error (%) (s=source, b=bridge, t=target)								
Target	**s1b2**	**s1b3**	**s1b4**	**s1b5**	**s2b1**	**s2b3**	**s2b4**	**s2b5**
t1	8.58	8.58	8.58	8.58	NA	10.32	10.38	10.21
t2	NA	10.54	10.17	10.65	8.71	8.71	8.71	8.71
t3	10.05	NA	10.78	10.74	10.25	NA	10.14	10.33
t4	10.35	10.43	NA	10.23	10.14	10.22	NA	10.21
t5	10.11	10.13	10.21	NA	10.12	10.23	10.07	NA
Target	**s3b1**	**s3b2**	**s3b4**	**s3b5**	**s4b1**	**s4b2**	**s4b3**	**s4b5**
t1	NA	10.23	10.54	10.76	NA	10.02	10.11	10.12
t2	10.37	NA	10.41	10.57	10.15	NA	10.28	10.13
t3	8.33	8.33	8.33	8.33	10.22	10.18	NA	10.32
t4	10.23	10.45	NA	10.58	8.78	8.78	8.78	8.78
t5	10.65	10.19	10.59	NA	10.08	10.02	10.34	NA
Xilinx-to-Intel Transfer model error (%) (s=source, b=bridge, t=target)								
Target	**s1b2**	**s1b3**	**s1b4**	**s1b5**	**s2b1**	**s2b3**	**s2b4**	**s2b5**
t6	11.58	12.23	12.01	12.13	12.15	12.69	12.32	12.43
t7	12.41	12.51	12.87	12.69	12.01	12.32	12.65	12.34
Target	**s3b1**	**s3b2**	**s3b4**	**s3b5**	**s4b1**	**s4b2**	**s4b3**	**s4b5**
t6	12.11	12.21	13.01	12.91	12.38	12.47	12.45	12.54
t7	13.43	12.01	12.78	12.65	12.45	12.54	12.63	12.65

the models fine-tuned by random selector cannot guarantee the target model performance since the quality of fine-tune dataset cannot be guaranteed. However, as the fine-tune dataset size increases, random selector tends to provide models with stable performance. The model performance difference by three selectors become more obvious when the number of designs decreases, and the design selector by XPNet always performs better than random selector.

An MLP is also implemented to test the usability of the methodology and

Table 5.10: Models on Intel devices as source models (MLP)

Intel-to-Intel Transfer model error (%) (s=source, b=bridge, t=target)										
Target	**s6b1**	**s6b2**	**s6b3**	**s6b4**	**s6b7**	**s7b1**	**s7b2**	**s7b3**	**s7b4**	**s7b6**
t6	9.76	9.76	9.76	9.76	9.76	10.21	10.34	10.15	10.17	NA
t7	9.98	10.11	9.97	10.21	NA	10.01	10.01	10.01	10.01	10.01

Intel-to-Xilinx Transfer model error (%) (s=source, b=bridge, t=target)										
Target	**s6b1**	**s6b2**	**s6b3**	**s6b4**	**s6b7**	**s7b1**	**s7b2**	**s7b3**	**s7b4**	**s7b6**
t4	12.43	12.69	13.02	NA	13.31	12.13	13.14	13.01	NA	13.32
t5	12.41	12.78	12.98	12.77	13.28	12.14	12.87	13.22	12.88	13.11

the results are shown in Table 5.9 and Table 5.10. The flow to generate the model for target FPGA is illustrated in Figure 5.10. In order to make the data compatible with the MLP model, the activity/toggling info is tabulariezed according to each operator's type to create a feature table. **Toggling** is recorded using the same method as discussed in [35, 37]. A snippet of such a feature table can be seen in Table 5.8. **Type** column refers to the operator type. **S-type** column refers to the operator type that this operator is succeeded by. For example, in the first row, fmul is succeeded by an Arithmetic operator. each operator is categorized according to its **S-type**, and the toggling values (two input and one output) are aggregated for the operator with the same **Type** and **S-type**. The aggregated values are put in the **Toggling** column, each operator is assumed to contain two input toggling and one output toggling values. The **Clock** column is filled with the target clock period.

Compared to the GNN implementation, XPNet with MLP has on average 2% degradation. The feature table is generated manually to represent the HLS design. Compared to the GNN encoder, the feature table lacks of important data flow features and it leads to a worse representation of designs. It is also worth mentioning that XPNet with MLP gives the error for Xilinx-to-Xilinx transfer with 10.32% error on average while GNN gives 8.40%. For Xilinx-to-Intel transfer model, MLP can achieve 12.46% error on average vs. GNN 10.34% on average. With the Intel FPGA as source, MLP can achieve 10.14% error and 12.86% error with Intel-to-Intel and Intel-to-Xilinx transfer model respectively, however, GNN can achieve 8.07% and 10.59% accordingly.

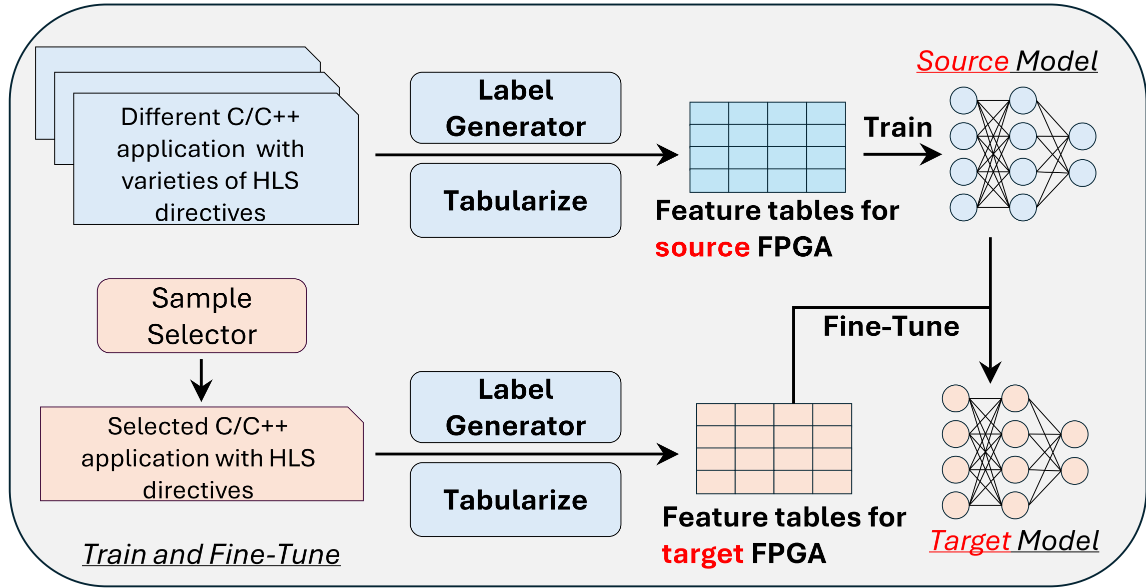


Figure 5.10: XPNet implemented with the MLP

Table 5.11: Model Accuracy comparison (same FPGA): XPNet vs PowerGear

Same FPGA model error (%)		
	Xilinx	**Intel**
PowerGear	5.74	NA
PowerGear with 20-top Design Selector	5.74	NA
PowerGear with 50-top Design Selector	5.74	NA
XPNet with 20-top Design Selector	6.37	7.75
XPNet with 50-top Design Selector	6.37	7.75

Table 5.12: Model Accuracy comparison (cross-FPGA): XPNet vs PowerGear

Cross-FPGA model error (%)				
	Xilinx-to-Xilinx	**Xilinx-to-Intel**	**Intel-to-Intel**	**Intel-to-Xilinx**
PowerGear	NA	NA	NA	NA
PowerGear with 20-top Design Selector	7.21	NA	NA	NA
PowerGear with 50-top Design Selector	6.01	NA	NA	NA
XPNet with 20-top Design Selector	8.40	10.34	8.07	10.60
XPNet with 50-top Design Selector	7.65	9.78	7.42	9.86

5.4.5 How does XPNet perform in comparison to other power models?

XPNet has two major advantages: First, it supports cross-FPGA power estimates and the device type is not restricted within one FPGA vendor; Second, it saves a great amount of time for building the model for a new FPGA. These two ad-

vantages are explicitly demonstrated in Table 5.11, Table 5.12 and Table 5.13, Table 5.14 respectively.

XPNet is compared vs. PowerGear [35], a state-of-art power model, in terms of model performance. However, PowerGear predicts on-board power consumption. The infrastructure is adjusted slightly so that it can take simulation power as the task to ensure a fair comparison. With the dataset containing data from Xilinx FPGAs, this PowerGear model error stays at 5.74%. PowerGear is also integrated with the design selector so that it can be adapted to a new FPGA within Xilinx vendor. It is much harder to adjust PowerGear to achieve cross-vendor prediction since the fundamental logic underneath is specific to Xilinx tool chains. As Table 5.11 and Table 5.11 shows, PowerGear performs better than XPNet for Xilinx devices because it uses post-HLS features, whereas XPNet uses fewer features and higher-level representation of designs. The use of only higher-level features is also a reason why XPNet can cover many more device types than PowerGear (including those from other vendors).

The efficiency of XPNet to build a model for a new FPGA is analyzed and summarized in Table 5.13 and Table 5.14. For model training, the time spent on dataset generation and power model training is evaluated in both XPNet and PowerGear. The overall time is composed of two parts: dataset preparation time and training time: $T = T_{dataset} + T_{training}$ where $T_{dataset}$ denotes the time to generate data samples for the designs on target FPGA. The time to generate each data sample is composed of HLS time, logic synthesis and implementation time, and simulation time. $T_{dataset}$ is estimated according to the dataset generation running on one Intel Xeon 5218 2.3GHz with 384GB RAM. $T_{training}$ includes training time or fine-tuning time. It is worth mentioning that $T_{dataset}$ can be reduced with more parallelism and using more powerful machines. Also, the time for training the pretrained model is not included in XPNet since it is considered as an one-time effort. Due to the existence of prior knowledge in the pretrained model, XPNet can save a significant amount of time than would be required in preparing a large dataset. It turns out XPNet is 232x faster than PowerGear to build a new model. For inference of power estimation on

Table 5.13: XPNet Speed comparison to prior approaches - training

Model for a new FPGA	**Model training**	
	Flow	**Runtime(m) / Speedup**
Traditional analysis model	NA	NA
PowerGear	∼5000 designs + Training	82,846 / 1x
XPNet	20 designs + Fine-tuning	356 / 232x

Table 5.14: XPNet Speed comparison to prior approaches - inference

Model for a new FPGA	**Power estimates/inference**	
	Flow	**Speedup**
Traditional analysis model	HLS + Syn, P&R + *Simulation* + Power analyzer	1x
PowerGear	HLS + Feature generation	4.8x
XPNet	HLS (front-end only) + Feature generation	24x

a new design, XPNet, with the invocation of only front-end HLS, is 24x faster than the traditional power analysis method and 5x faster than the state-of-the-art power model.

5.5 Summary

In this work, XPNet is proposed and it is a methodology to build a power model with high level language code that combines transfer learning techniques with novel data sample generation methods on a target FPGA. XPNet can build sufficient cross-FPGA power models that can be used to predict power even on FPGAs from different vendors. With the help of transfer learning, K-top sample selection and a bridge FPGA, the GNN based XPNet can achieve 8.40%, 10.32%, 8.07%, 10.59% average error on Xilinx-to-Xilinx, Xilinx-to-Intel, Intel-to-Intel, Intel-to-Xilinx transfer model, respectively. With fewer data samples needed to build a model for a new FPGA, a speedup of 232x is achieved in comparison to the state-of-art power model. Since only front-end HLS features are necessary to build the feature map, a speedup of 5x over the state-of-the-art power model can be observed during inference.

Chapter 6: ATAPP: Architecture and Technology Aware Power Predictor for Unseen FPGAs

Power efficiency has emerged as one of the first-order constraints for hardware systems such as field-programmable gate arrays (FPGAs), and both the FPGA architecture and design optimization with regard to power efficiency usually necessitate knowledge of power consumption. However, the power evaluation flow for FPGA designs induces large overheads of design turnaround time, and the problem becomes more serious when researchers want to explore the archiecture of an FPGA considering power metrics in addition to performance metrics. In general, accurate FPGA power estimation requires the signal activities of critical components and I/O ports to be obtained via vector-based gate-level simulation and a set of physical component measurements to be obtained through the Register-Transfer-Level(RTL)-based FPGA implementation flow, including synthesis and implementation. After these steps, analytical models can be used to infer power consumption. However, all the above steps need to be refined and repeated once the FPGA architecture changes, since the resource types or counts and their layout on the FPGA fabric can greatly affect the implementation phase even with the same RTL design. The technology node can further influence all the mentioned phases and the analytical model for power evaluation. The cycle-accurate gate-level simulation and design implementation flow with different FPGA architecture result in large runtimes. Overall, in a power-oriented hardware optimization loop, designers repeatedly perform the above power evaluation steps until power closure is achieved, which incurs long development time and high labor cost.

To overcome the challenge of post-implementation power evaluation in FPGA, previous works have used machine learning (ML) techniques [35, 36] to bypass bottleneck phases including synthesis and implementation. ML models are trained with features extracted from the High Level Synthesis (HLS) [5] phase and labels from

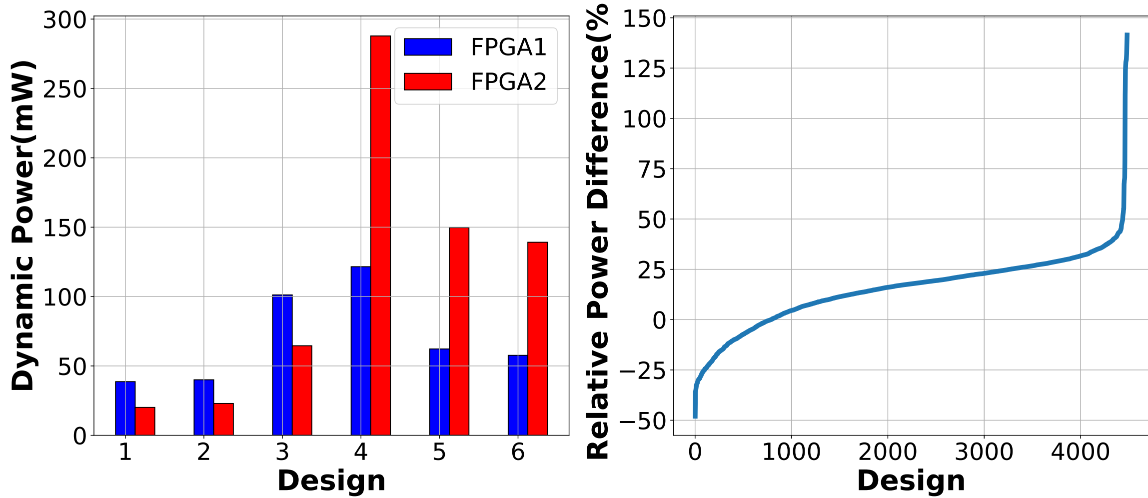


Figure 6.1: Changes in power from one FPGA to another. Power may increase or decrease as shown (left) for example designs; Distribution of the relative power difference on two different FPGAs based on over 4000 designs (right)

simulation-based power consumption. The models mentioned above are designed to predict power with HLS features on one specific FPGA, however, these models cannot be used in power prediction on unseen FPGAs. The power values over 4,000 designs are compared on 2 different FPGAs and the relative power differences (calculated as $\frac{P_2-P_1}{P_1}$) are shown in the right of Figure 6.1. For a specific view of the power comparison as illustrated in the left of Figure 6.1, 6 designs are picked and the power consumption of these 6 designs are compared on 2 different FPGAs (zu9eg as FPGA1 and 7v585t as FPGA2). There is a significant power difference between FPGA1 and FPGA2. Therefore, power models, which can make an accurate prediction on FPGA1, cannot make correct prediction on FPGA2 (a new unseen FPGA) due to the features dependent solely on the designs and the lack of consideration of the features of the FPGA architecture.

The prior analytical power model and existing ML-based power models are compared in Table 6.1. There are integrated power estimation tools inside AMD/Xilinx Vivado and it supports the power estimation for all the Xilinx FPGAs, however, it lacks the flexibility to estimate the power in different design cycles. It takes a long

Table 6.1: A comparison of ATAPP with the existing work on power prediction. ●: feature supported; ○: feature unsupported; ◐: feature partially supported.

Features	**Vivado**	**XPE**	**VTR**	**PowerGear**	**ATAPP**
High accuracy	●	○	◐	◐	●
Time efficient	○	●	○	◐	●
No logic synthesis	○	◐	○	●	●
No implementation	○	◐	○	●	●
Unseen FPGA Prediction	○	○	●	○	●

time to synthesize and simulate designs before accurate power estimation. The AMD Power Estimator(XPE) is an analytical power model that supports flexible estimation in different design phases. It supports all AMD/Xilinx devices, but accuracy is too low with early-stage HLS results [91]. Both Vivado and XPE only support power estimation on existing devices, and researchers are not allowed to manipulate the FPGA architectures. Verilog-to-Routing (VTR) [92] is an open source tool used for the exploration of FPGA architecture. Although manipulation of the FPGA architecture is allowed, it takes even longer than Vivado to synthesize the design and estimate the power. PowerGear [35] is state-of-the-art ML-based power model, it can make prediction accurately on the FPGA seen in the training, it perform poorly for the unseen FPGAs. In order to predict power for the unseen FPGA with PowerGear, it requires a complete dataset generation and training which restrict the usage of PowerGear on varities of FPGAs. Therefore, fast and accurate power evaluation on unseen FPGAs remains unsolved.

ATAPP is an FPGA architecture and technology-aware power predictor that combines both the features of the FPGA architecture and the switching activities of the designs to predict the average dynamic power consumption. Unlike the existing ML-based power model, ATAPP makes predictions based on two parts: FPGA architecture representation and design representation as shown in Figure 6.2. The architecture representation includes information such as resource type and counts, technology parameters, resource position in the FPGA fabric, and voltage levels. They are encoded into a finite length of vectors so that they can be digested by a

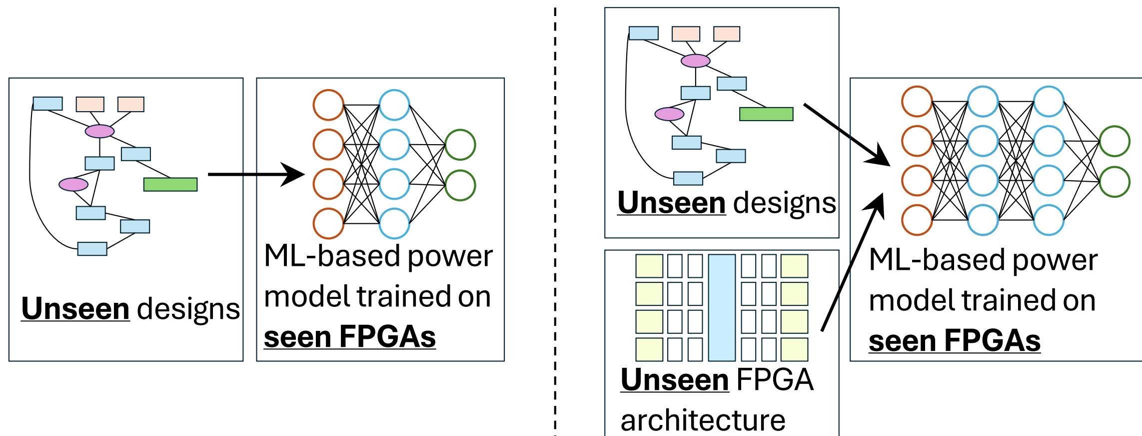


Figure 6.2: Power estimation by state-of-the-art ML-based power model (left) vs. ATAPP (right)

multi-layer perceptron (MLP) model to generate architecture embeddings. For the design representation,

The HLS design is represented as a graph which is generated with Intermediate Representation (IR) codes and Finite State Machine Datapath (FSMD) model produced by HLS compilation. Switching activities are extracted by simulating the IR codes and put them on the associated edges of the graph. A GNN encoder, which is adapted from UniMP [93], is applied to the HLS designs to generate design embeddings. Both architecture and design embeddings are concatenated and fed into another MLP for a regression task. ATAPP is the first work to predict power across FPGAs based on both FPGA architecture and HLS designs. In this chapter, Xilinx FPGAs are used as example devices, but the approach is extendable to other FPGA vendors and academic FPGA architecture research. The contributions are summarized as follows:

- A GNN-based model XPNet is proposed and it is an architecture-aware and technology-aware power prediction model. It is trained on seen FPGAs and seen designs (circuits) but can make power prediction for both unseen FPGA architectures and designs.

- Positional encoding is applied to efficiently encode the FPGA architecture features at the tile level and generate FPGA architecture embeddings with a MLP model.
- A GNN encoder based on UniMP layer is developed to generate graph embeddings with switching activities-aware graph, generated with IR codes and FSMD model.
- The experimental results show that XPNet can predict dynamic power with a 13.09% error on average.

6.1 Problem Formulation

In this work, the aim is to predict the power for a design implemented on a FPGA using an ML technique. For this matter, it is necessary to define and solve the following problems step by step:

6.1.1 Problem 1: Generate the FPGA architecture representation

In order to build an ML-based power model for different FPGAs, FPGA architecture representation is essential and it involves selecting and encoding a comprehensive set of features that accurately capture the architectural impact on power consumption. These features should be both representative of the FPGA characteristics and should be compatible with machine learning algorithms. The key challenge is to find a good architecture representation include ensuring sufficient granularity to reflect relevant details while avoiding unnecessary complexity that could hinder model generalization. Therefore, the FPGA architecture representation is split into several categories that can highly impact power consumption for the device denoted as d: resource counts and types β such as LUTs, flip-flops, DSP slices and BRAM, technology parameter and voltage levels γ and the layout map δ of resources and switch boxes. A device is determined by these three factors $d(\beta, \gamma, \delta)$.

6.1.2 Problem 2: Generate the design (circuit) representation with switching activity

In the flow of mapping C applications to an FPGA, two main graphs are generated by HLS and implementation (i.e. placement and routing) tools. The coarse-grained high-level operator graph is generated after HLS, and each node represents one IR-level operator, and each edge represents the data flow between operators. The fine-grained or circuit-level graph is generated after implementation, and each node represents one logic gate, and each edge represents the wire between two logic gates. The ground truth power is calculated based on the fine-grained graph with the equation $P = \sum_{i \in I} \alpha_i C_i V^2 f$ where α_i is the signal toggling activity, C_i is the interconnect capacitance, V is the supply voltage, f is the operating frequency, and i is an interconnect of the whole set I. To predict power as close to the ground-truth power using the coarse-grained graph, a concise graph are required. The graph $G = (V, E)$ should at least correctly represent the data flow and switching behavior in its coarse-graine level.

6.1.3 Problem 3: Build the prediction model with both the design (circuit) and the unseen FPGA architecture

Let g be the HLS generated codes with switching activities α on a FPGA device d with resource and interconnect representation β, technology parameter γ and layout map δ. Let P be the the ground-truth power generated by the vendor FPGA implementation and simulation tool:

$$P = F(g(\alpha), d(\beta, \gamma, \delta)) \tag{6.1}$$

The goal is to find a hypothesis H that approximates the results of function F for any given HLS generated codes g with any switching activities α on any FPGA device d that can be defined by the resource and interconnect features r, technology parameter t and layout map δ:

$$\min_H Loss(F(g(\alpha), d(\beta, \gamma, \delta)), H(g(\alpha), d(\beta, \gamma, \delta))) \tag{6.2}$$

6.2 ATAPP details

The objective of the study is to generate a general ML-based model which can make power predictions based on the FPGA architecture and design representation. As discussed in Section 6.1, the solution mainly includes an effective FPGA architecture and representation of the HLS design, as well as a model construction. ATAPP is trained with a set of FPGA architecture, design overlay on that FPGA and the associated ground-truth power. It can then make prediction with unseen FPGA architecture and design representation. The architecture representation is encoded in a particular way discussed in Section 6.2.1 and how the design representation is generated is discussed in Section 6.2.2. Section 6.2.3 discusses the detailed view of the model construction and how the architecture and design embeddings are produced. Based on the definition of architecture and design features, a dataset covering varities of FPGAs is generated for the studies.

6.2.1 Architecture representation

It is not straightforward to feed power-related FPGA architecture features directly to an ML-based model since not all of features are numeric. Representing an FPGA architecture, however, in a numerical way poses several challenges due to the inherent complexity, flexibility and highly configurable nature of FPGAs. These challenges arise from the spatial structure of the architecture, for example, how to arrange the basic unit on the fabric matters a lot to the implementation and hence the power consumption of the same HLS design but to make these arrangements consumable by an ML model requires extra effort.

As discussed in Section 6.1, the power prediction problem is defined by resource and interconnect representation, technology parameters and layout map. The resource representation and technology parameters can be expressed as numeric values. Therefore, in order to explore the effect of these features on power prediction, the FPGA type is scoped within the AMD/Xilinx ultrascale+, ultrascale and 7series

families. The resource count is one of the important features for the capability of an FPGA device. Therefore, the number of these resources, E_{res} (DSP, LUT, FF and BRAM), are included as key features. Even though the ultrascale and ultrascale+ FPGA devices use different DSP structure from 7series, no obvious number of utilized DSP difference is observed between the devices when implementing the same HLS design. All the FPGA devices mentioned use the same structure of other units, therefore, it is not necessary to go deeper to reach the gate-level implementation the basic components. Besides the available resource counts, technology parameters E_t, including technology node and operational voltage, are also critical to the power dissipation.

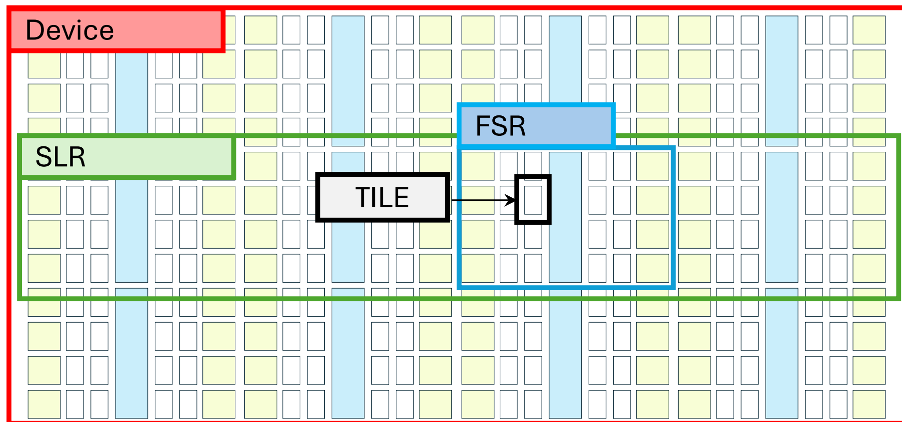


Figure 6.3: FPGA Architecture Terminology

When it comes to spatial features of the FPGA architecture, it is important to define which abstract level to reach. In Xilinx FPGAs, there are six major levels of hierarchy - the entire device all the way down to building blocks. They are **Device, Super Logic Region (SLR), Fabric Sub Region (FSR), Tile**, Site and Basic Element of Logic (BEL) [94]. The first four hierarchies can be seen in Figure 6.3. The device is at the highest level of Xilinx architecture and it is composed of replicated FSRs. SLRs are present in certain devices and each SLR contains a 2D array of FSRs. FSR is a 2D array of tiles in the fabric. Each tile is an instance composed of multiple

sites and each tile has a unique name with a coordinate suffix. Not all tiles contain sites (there exist NULL and empty tiles in Xilinx terminology), but those that do can have more than one. A site is referred to as a group of related elements and their connectivity. Similar to a tile, each site is associated with its own coordinate grid and there is a chance that two sites share the same grid space. The site type includes SLICEL and SLICEM which are the most common site types and are the basic configurable logic building blocks (CLBs) that contain LUTs and FFs. At the lowest level, the atomic unit is a BEL. BELs are the smallest, indivisible, and representable component in the fabric of an FPGA. In order to collect the aforementioned information, **RapidWright**[94] is used and it is an open source Java framework that provides accurate device model views of all Vivado-supported Xilinx devices including 7series, ultrascale and ultrascale+. The encoding flow of the FPGA architecture includes several steps and is summarized in Figure 6.4. The process is introduced in detail below.

6.2.1.1 FSR extraction

Since each device is composed of several similar FSRs, modeling every FSR separately is not required to create a representation of the entire FPGA architecture. Therefore, tile information is extracted from one of the FSRs. Specifically, `Device.getDevice()` is firstly used to choose the FPGA to extract, `Device.getClockRegion()` is used to extract the clock region. The FPGA tile collection is extracted with `Device.getTiles()`. all the tiles are checked and the tile collection is reduced down to the selected FSR with `ClockRegion.containsTile()`. Since every device may have different number of FSRs, the number of rows and columns of FSRs should be also included. They are included in the metadata and appended to the final embeddings.

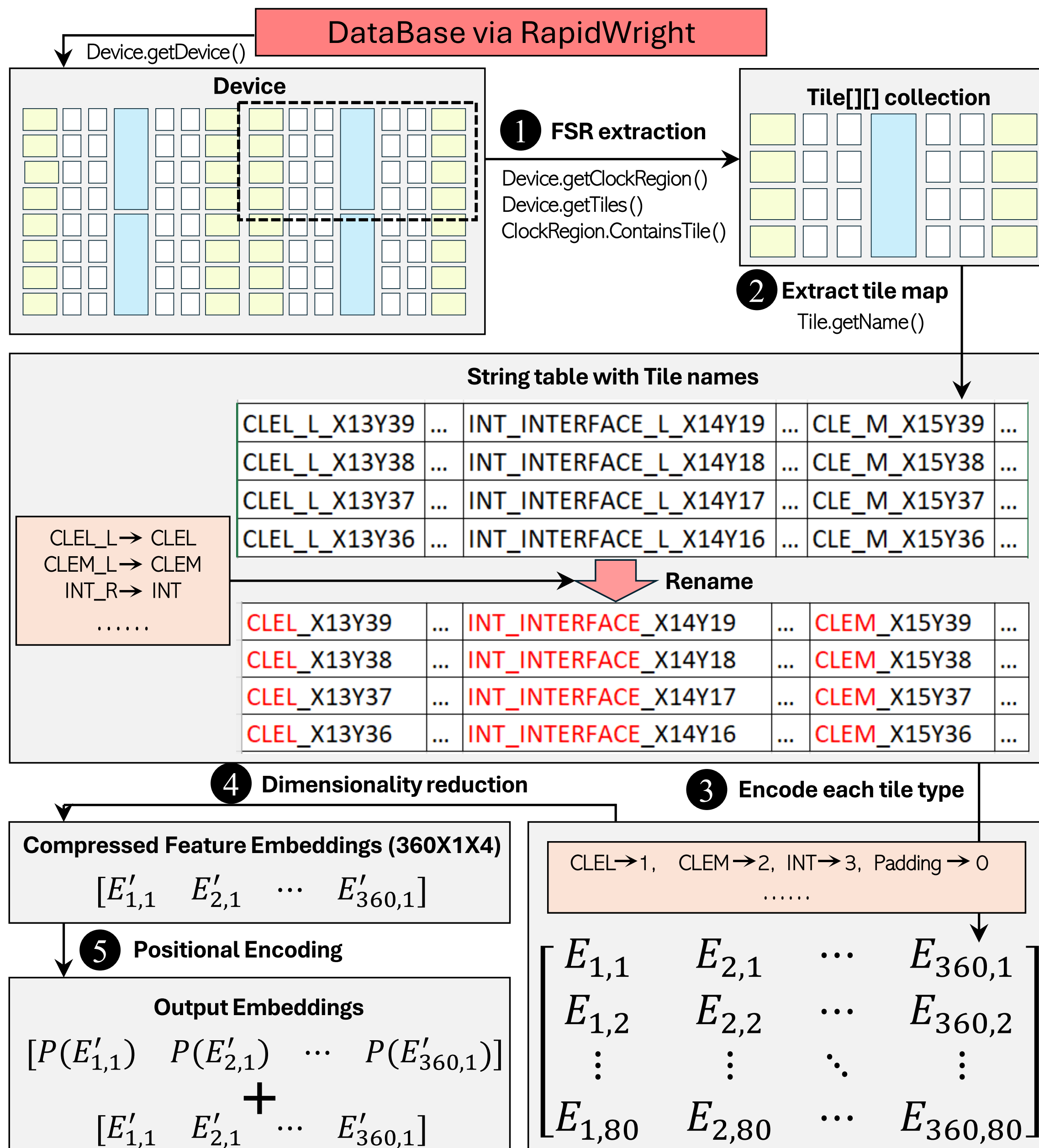


Figure 6.4: FPGA Architecture Encoding

6.2.1.2 Extract tile map

Once the tile collection of the selected FSR is obtained, the next step is to transfer these tile objects into a string table that is easy to understand and parse. `Tile.getName()` is used to find the name of each tile. Every tile name is composed of the resource type as its prefix and the grid location as its suffix. For example, `INT_X14Y519` refers to a switch box tile located at grid (14, 519). Before advance to the next step to encode the string table, it is necessary to to reduce the resource space first, since the total number of unique tile types can be too large and not all of them are frequently used in the device. The tile type counts range from 180 up to around 360, and to encode all of them incurs too much sparsity when generating the embedding and training may become difficult as a consequence. Therefore, the tile is selected to include a specific resource type and others are removed from the string table. Seven tile types are selected and they include `CLEL, CLEM, INT, INT_INTERFACE, DSP, BRAM, BRK` where `CLEL, CLEM` refer to the tile containing CLB, `INT, INT_INTERFACE` refer to the tile containing switch box and `BRK` refers to a break tile that disallows any crossing. However, each tile name cannot exactly fall into the seven categories due to the inconsistent naming nature in the FPGA series, for example, `CLBLL_L` is used in 7series FPGAs but `CLEL_L` is used in others. Therefore, the name of each tile is checked to determine where it belongs:

1. All tiles with name containing `CLEL` belong to `CLEL`. Therefore, tiles such as `CLEL_L` and `CLEL_R` should be encoded as `CLEL`. It is also noticed that `_L` and `_R` are used to distinguish the relative position of `CLEL`. Since only the resource type is important at this stage, it is sufficient to encode them in the same way.

2. All tiles with name containing `CLEL` belong to `CLEL`. `CLBLL` is encoded as `CLEL`. `CLBLM` and `CLE_M` are encoded as `CLEM.CLBLL` in 7series is encoded as `CLEL` and `CLBLM` in 7series and `CLE_M` are encoded as `CLEM`. Although two independent slices of the 7 series CLB are combined into one cohesive UltraScale architecture slice, top and bottom halves of UltraScale architecture slices are each similar

to a 7series slice and the total resources per CLB are similar therefore the above mentioned tiles can be encoded into the same category. CLEL and CLEM should not be considered as the same tile since CLEL contains CLB supporting logic implementation only while CLEM contains CLB that can be configured as distributed RAM or a shift register.

3. NULL tile is encoded as one of DSP, BRAM and BRK based on its column. NULL tiles normally appear in the same column as DSP, BRAM and BRK, therefore these NULL tiles are encoded in the same way as its associated resource. Meanwhile, the rule ① applies to the 3 tiles.

4. INT need to be distinguished from INT_INTERFACE due to the position and functionality difference. The rule ① applies to these two tiles as well, but INT_INTERFACE should not be counted as INT.

5. Since each grid usually contains more than one tile, all the tiles that do not belong to the seven tiles are removed. In very rare case if the removed tile is the only tile in that grid, replace it with a dummy 0.

With the above rules, more than 97% tiles from FSR can be encoded into the seven selected tile types and almost all the grids are occupied by them.

6.2.1.3 Encode each tile type

Once the renaming of the string table is completed. The table is encoded into a matrix of embeddings. Each grid location, which is identified by the suffix _X#Y# of each string, contains no more than 4 tiles. Therefore, each embedding should be a size 4 vector. For the grid which has less than 4 tiles, pad the embedding with a dummy 0. An example tile [CLEM_X31Y815, INT_X31Y815, CLEL_X31Y815] can be encoded as a single embedding [2, 3, 1, 0]. Considering that the grid size for FSR is not identical across different devices, The matrix size is defined to be 360 columns × 80 rows which is larger than the largest grid size of the selected FPGAs (350×77).

Zero paddings are added to the matrices for the empty place. A 3d matrix with size of 360×80×4 (#columns×#rows×#channels) is generated.

6.2.1.4 Dimensionality reduction

Xilinx leverages a columnar-based architectural approach to tile layout. That is, with a few exceptions, all tiles within a column are of the same type but tiles occupying the same row are typically different types. Therefore, the matrix size can be compressed from 360×80×4 into 360×1×4 since each row is simply a replicate of the first row. Extra metadata is needed to indicate the the number of rows and columns with valid tiles (not padding zero). The embeddings eventually becomes a 360×1×4 with zero paddings at the end.

6.2.1.5 Positional Encoding

Since the sequence to place the tiles on fabric is critical to solving the power prediction, *Positional Encoding (PE)* is used to encode the resource layout of FPGA devices. *PE* is a key concept in transformer models[95], designed to provide information about the order of sequence or spatial elements. Since transformer architectures are inherently permutation-invariant, they lack an innate sense of sequence order, which is critical for tasks involving sequential/spatial data. Hence, PE is added to the input embeddings to include order information. This encoding can be learned or predefined; a popular approach uses sinusoidal functions with different frequencies to generate unique values for each position in the sequence. This method ensures that the positional encoding generalizes to sequences of varying lengths and maintains properties conducive to understanding relative and absolute positions. A typical *PE* equation is defined as:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d})$$

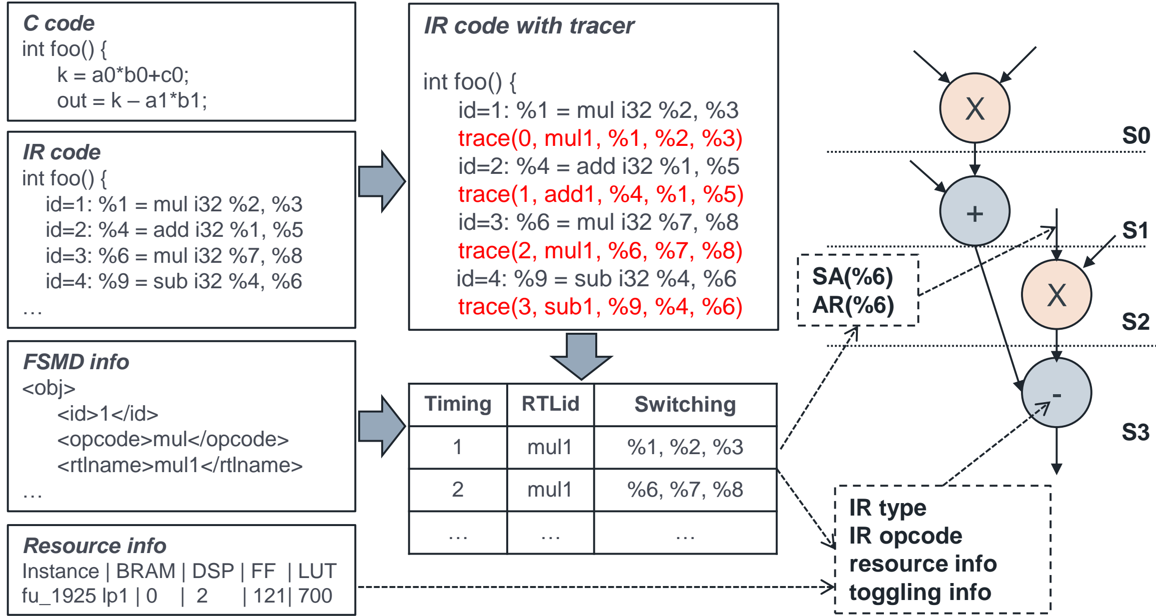


Figure 6.5: Design representation generation flow

where *pos* is the position of the input embedding in the sequence. i is the index of the dimension in the embedding vector and d is the dimensionality of the embedding which is 4 in the case. The positional encoding is calculated and directly added to the input embeddings to generate output architecture embeddings AE.

6.2.2 Design representation

The power-aware design representation is mainly composed of two parts. The first is the design itself including the number and types of operators and the path to link these operators. The second component is the behavior of these operators which is normally represented as switching activities. Therefore, in order to correctly represent the two factors, IR codes with the FSMD model generated by HLS tools are chosen. Similarly in [35, 37, 57, 58], the design generation flow used in ATAPP is summarized in Figure 6.5. Three main files generated from HLS tools are used: IR codes, FSMD model and resource utilization for each operator. In order to trace the toggling behavior of each operator, a trace function is inserted after every important

IR instruction. With the IR-level simulation, the behavior of each IR operator can be recorded. With the FSMD model, each IR operator can be mapped to a specific RTL hardware operator, the IR operators sharing the same hardware resources can be merged and a graph close to the hardware can be generated. For the generated graph $G = (V, E)$, where V and E represent the node and edge set, respectively, every node in the graph $\forall v \in V$ represents a IR operator with attributes that include opcode, opcode type, input, and output switching activities. Every edge in the graph $\forall e_{i,j} \in E$ where $e_{i,j}$ is the edge with i as the source and j as the sink. The edge $e_{i,j}$ contains switching activities $SA_{i,j}$ and the activation ratio $AR_{i,j}$:

$$SA_{i,j} = \frac{\sum HD(v_i(k), v_i(k-1))}{L}, \quad AR_{i,j} = \frac{N}{L} \tag{6.3}$$

where HD refers to hamming distance, L refers to the latency of the design and N refers to the number of execution cycles that cause the change of the vertices v_i. The HD inside SA accumulates in every cycle when the vertices change. In addition to these two switching features, the edge type is also encoded and added to the edge attributes.

6.2.3 Predictive Model

Table 6.2: Features used in ATAPP

Feature category	Format	Details
Architecture	360×1×4 matrix with PE: AE	Resource types and the arrangement of resources on fabric
Design	Graph data with node and edge attributes; Metadata	DFG recovered from IR codes, FSMD model and post-HLS reports including achieved clock period, overall latency and resource utilization
Metadata	A vector of size 21: $E_{res} \| E_t$	Available resource counts on the device, # of rows and columns of FSR, # of valid rows and columns of tiles, technology node (in nanometers), voltage levels on the device ($V_{ccint}, V_{ccbram}, etc.$), the size of FPGA

The features and format of the features used in the model are listed in Table 6.2. The detailed structure of ATAPP is shown in Figure 6.6. It is made up of three

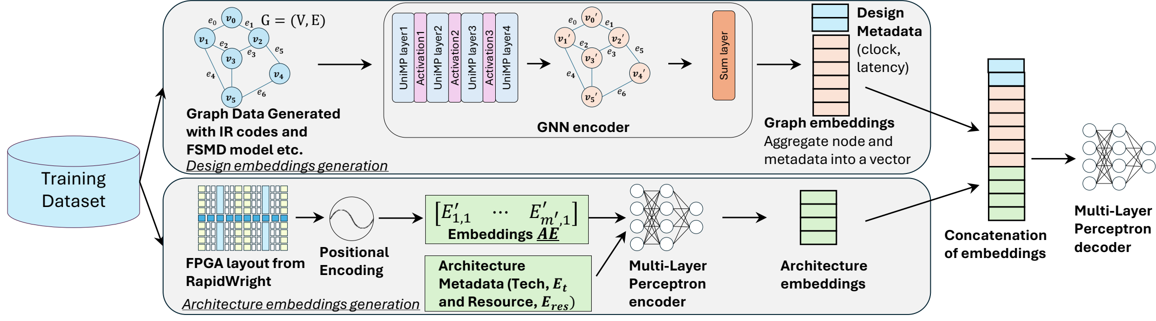


Figure 6.6: Detail structure of ATAPP composed of *design embeddings generation* using GNN encoder and *architecture embeddings generation* using MLP; the power regression is conducted with a MLP

major neural networks: a GNN encoder to generate graph embeddings for design representation, a MLP model used to generate architecture embeddings, and another MLP model used to perform the power regression task.

6.2.3.1 GNN encoder

Edge attributes, including switching features, are essential to perform power prediction. However, both GCN [89] and GAT [90] overlook the edge embeddings. Although PowerGear [35] propose an edge-expressive GNN, the convolution is performed on each node with neighbor edges where the neighbor nodes and further edges are not fully utilized. Moreover, the number of learnable edge weights is restricted by the number of edge relation types that cannot represent varieties of capacitance in circuits. UniMP [93], inspired by Transformer [95] used a different aggregation mechanism on each edge. It builds attention coefficients $\alpha_{i,j}$ with both edge and node attributes in every UniMP layer:

$$\alpha_{i,j}^{(l)} = softmax\left(\frac{(W_1^{(l)}h_i^{(l)})^T(W_2^{(l)}h_j^{(l)} + W_3 e_{i,j})}{\sqrt{D}}\right) \tag{6.4}$$

where l refers to the layer, $e_{i,j}$ represents the edge pointing from vertices v_i to v_j, h_i refers to the node embedding at vertices v_i, D is the hidden size of each head. In the end of the layer, Each node embedding is updated with message aggregation from the

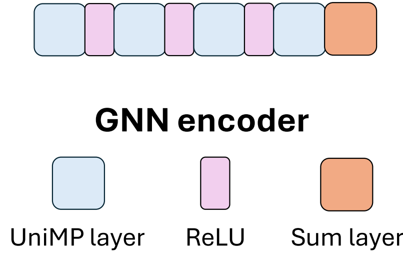


Figure 6.7: Architecture of GNN encoder

distant j to the source i:

$$h_i^{(l+1)} = \sum_{j \in N(i)} \alpha_{i,j}^{(l)} \left(W_4 v_j^{(l)} + e_{i,j} \right) \tag{6.5}$$

To generate one vector representation h'_G for the entire graph, all th node embeddings from all UniMP layers are aggreagated in the last sum layer. The graph embedding is concatenated with the design metadata h_M extracted from post-HLS reports and it includes achieved clock period and latency. The aggregation and concatenation are formularized as follows:

$$h'_G = \sum_{l \in L} \sum_{v \in V} h_v^{(l)}, \quad h_G = h'_G || h_M \tag{6.6}$$

where L is the set of indexes of GNN layers and V is the set of vertices in the graph. The aggregation of all nodes across layers can enhance the generalization ability of the model. The GNN encoder is made up of 4 UniMP layers, 3 ReLU activation layers, and 1 sum layer.

6.2.3.2 MLP encoder and MLP decoder

The MLP is used to encode the architecture because the feature space is relatively simple. The MLP encoder is made up of 2 hidden layers. The MLP decoder is used to perform power regression task and is composed of 2 hidden layers. Every MLP hidden layer is followed by ReLU activation. The power estimation P_est is

calculated as follows:

$$P_{est} = MLP_1 \left(h_G \| h_A \right), \quad h_A = MLP_2 \left(h_{AE} || E_t || E_{res} \right) \tag{6.7}$$

6.3 Experiment Setup

Table 6.3: Benchmark kernels used for the design dataset per FPGA

Benchmark suite	Kernel application	# Designs
Polybench	atax, bicg, gemm, gesummv, k2mm, k3mm, mvt, syrk, syr2k	4779
Machsuite	spmv_crs, stencil3d, stencil2d	376
CHStone	aes, gsm, sha	281

Table 6.4: Overview of characteristics of FPGAs used in experiments

#	Series	Device	Tech	#LUTs	#BRAMs	#DSPs
1	ultrascale+	zu9eg	16nm	274,080	912	2,520
2	ultrascale+	vu3p	16nm	394,080	720	2,280
3	ultrascale+	au25p	16nm	141,000	300	1,200
4	ultrascale	vu440	20nm	2,532,000	2,520	2,880
5	ultrascale	ku115	20nm	663,360	2,160	5,520
6	7series	7a200t	28nm	134,600	365	740
7	7series	7k480t	28nm	298,600	955	1,920
8	7series	7v585t	28nm	364,200	795	1,260

The dataset includes 5,391 designs for each FPGA (over 43,000 in total). These designs are generated from benchmark kernels of intermediate complexity that can be used as building blocks of larger applications. Specifically, kernels are selected from the widely used MachSuite[48], CHStone[47], and Polybench[86] benchmark. They include kernels with different computation intensities including linear algebra operations on matrices and vectors, data mining (correlation and covariance), stencil operations, encryption, and a dynamic programming application as shown in Table 6.3. The HLS design generation is automated using HLSFactory [96], which is a framework designed specifically for the HLS dataset generation. Since it does not

support the simulated power evaluation on the implemented designs, the simulation is run on the generated designs to collect ground-truth power for each FPGA.

AMD/Xilinx Vivado HLS 2018.3 and Vivado 2018.3 are used to synthesize and simulate the design to collect the design features and ground-truth power. 8 different FPGA devices from AMD/Xilinx are selected for the experiments as shown in Table 6.4. The FPGA architecture representation is processed with the raw data from RapidWright as discussed in section 6.2.1. The model is implemented and trained using PyTorch. The dataset is split into 85% for training, 15% for testing. 10-fold cross-validation during training with Adam optimizer and a learning rate of 0.001 are applied.

Three metrics are used to evaluate the prediction accuracy: correlation coefficient (R), Mean Absolute Error Percentage (MAPE) and Root Relative Square Error (RRSE), which are defined as following:

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100\%, \quad \text{RRSE} = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{6.8}$$

where n refers to the number of samples, y_i is the actual power and $\hat{y}_i$ is the predicted power of i_{th} design, $\bar{y}$ is the mean of the actual power. These metrics bring a comprehensive and fair evaluation of ML models from three aspects, where higher correlation R, lower MAPE and RRSE indicate better model performance and accuracy.

6.4 Baseline Solutions

While there exists no prior architecture-based power model with ML technique, AMD/Xilinx Power Estimator(XPE)[91] and PowerGear [35] are selected to compare against ATAPP. XPE is a spreadsheet based tool that estimates the power consumption of the design at any stage during the design cycle. It is typically used in

the pre-design and pre-implementation stages. It accepts design information through simple design wizards, analyzes them and provides a detailed power and thermal information. However, XPE is designed to estimate worst-case power and therefore the estimated power tends to be larger than the ground-truth power, which is generated with Vivado synthesis and simulation tools as described in Section 6.3. PowerGear, on the other hand, is a state-of-the-art GNN-based power model for HLS designs. It constructs graph samples for designs with HLS-generated reports and switching activities from simulation. Their objective is to generate the model for single FPGA, therefore, the model performs poorly when directly used in other FPGAs. Before examing the major experimental results, some preliminary experiments are conducted with these baseline solutions.

Table 6.5: XPE estimation error compared to Vivado in different design phases for UltraScale Virtex vu440. P_{syn}, P_{impl}: power estimated by Vivado after syn and impl. P'_{syn}, P'_{impl}: power estimated by XPE after syn and impl

{Act. Est.}	$\{P_{syn}, P'_{syn}\}$	$\{P_{impl}, P'_{impl}\}$
MAPE	26.26%	5.03%
{Act. Est.}	$\{P_{impl}, P'_{syn}\}$	$\{P_{impl}, P'_{hls}\}$
MAPE	37.78%	390.07%

XPE can estimate the power from any design cycle, but the accuracy of XPE is largely dependent on how much information can be entered to it. XPE considers the design resource usage, toggle rates and many factors which it combines with the device models to calculate the estimated power distribution. It accepts two primary sets of inputs: ① device usage, component configuration, clock, enable, and toggle rates, and ② device data models which are already integrated into the tool. With the specification of designs extracted from Vivado, XPE results still deviate from the Vivado results. A preliminary experiment is conducted: the implemented design is simulated and the switching activity is generated into a vector file (`.saif`). The vectors are then used in Vivado power estimator on both synthesized and implemented design to generate dynamic power P_{syn}, P_{impl} respectively, the settings are also ex-

tracted as XPE compatible files `syn.xpe, impl.xpe`. These files are then used in XPE to estimate the core dynamic power P'_{syn}, P'_{impl}. Vivado does not support power report at HLS stage and there is no way to extract `.xpe` file right after HLS. The HLS estimated resource and default toggling rate 12.5% are used to estimate power with XPE, the core dynamic power is denoted as P'_{hls}. The XPE error is concluded in Table 6.5. The first row $\{P_{syn}, P'_{syn}\}$ and $\{P_{impl}, P'_{impl}\}$ indicate that even with the same configuration, XPE still deviates from the actual value. Compared to the predicted power of the target P_{impl}, as the amount of information accepted by XPE decreases, the error increases significantly if no appropriate switching activity is entered.

PowerGear trains GNN models to predict the post-implementation dynamic power. the simulated power is used as ground truth power rather than the measured power in the original paper. The GNN model takes graph representation of HLS designs as input to make prediction. Although it performs quite well on the same FPGA it is trained on (MAPE=5.08%), the error increases greatly when the FPGA changes (MAPE=44.63%). The error comes from the difference in FPGA architectures between unseen FPGAs and trained FPGAs, and the model is not able to adapt to unseen FPGAs without consideration of FPGA architectures since PowerGear is not an architecture-aware power model, therefore, ATAPP is needed.

6.5 Results

6.5.1 Model Evaluation

Figure 6.8 and Table 6.6 show the comparison of ATAPP over PowerGear and XPE. ① For ATAPP, 10% designs are split out for testing purpose, these designs across all the FPGAs are not used during the training. When it comes to the actual training, all the data samples from one FPGA are split out, the model is trained with the rest of the data samples. The model is tested with the separate designs on the separate FPGA to ensure that both designs and FPGA are unseen to the model. This level-one-out strategy is iterated and applied to all 8 FPGAs. ② For PowerGear, same

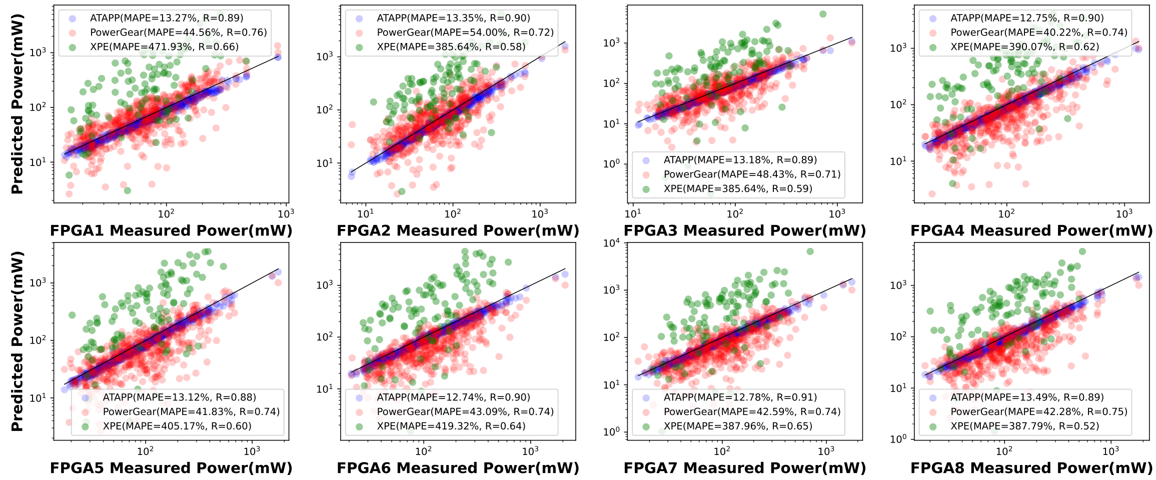


Figure 6.8: Prediction vs ground-truth for dynamic power consumption of all designs. The black line in the middle of each figure indicates **zero error** (i.e. predicted power equals to the ground-truth power). **ATAPP**: **leave-one-out** strategy is used, the model is tested with the designs on one FPGA and trained with the designs on all other FPGAs. 8 models are trained and tested independently. **PowerGear**: train with the designs on FPGA8 and test with the designs on FPGA1-FPGA7; train with the designs on FPGA1 and test with the designs on FPGA8. Blue - ATAPP; Red - PowerGear; Green - Vivado XPE.

10% designs are split out for testing. The model is trained with all the rest designs on FPGA8 and tested on all other FPGAs with the separate design. For the result on FPGA8, the model is trained with the designs on FPGA1. ③ For XPE results, the HLS estimated resource and default toggling rate 12.5% are used to conduct the power estimation across all the FPGAs. No post-HLS reports are used to make sure the power estimated by three methods are from before the HLS stage. In this way a fair comparison is guaranteed.

There are several key observations from the table and figure. First of all, ATAPP significantly outperforms original prior works for all the power estimations on 8 FPGAs with much lower RRSE and MAPE and higher correlation R. Secondly, the correlation of all three methods are larger than 0.5. It can be seen from Figure 6.8 that all three methods follow the same trend as the ground-truth power and it implies

Table 6.6: Accuracy Comparison of different methods
ATAPP: Prediction with design and arch features (Pos Enc=Positional Encoding); `Leave-one-out` for training and testing;
PowerGear: Train with the designs on FPGA8 and test with the designs on FPGA1-FPGA7;train on FPGA1 and test on FPGA8;
XPE: Estimates with design characteristics on each FPGA sheet

Method	ATAPP w/ PE			ATAPP w/o PE			PowerGear			XPE		
Device	MAPE	RRSE	R	MAPE	RRSE	R	MAPE	RRSE	R	MAPE	RRSE	R
FPGA1	13.27%	0.22	0.89	27.39%	0.48	0.81	44.56%	0.74	0.76	471.93%	9.73	0.66
FPGA2	13.35%	0.21	0.90	28.48%	0.47	0.80	54.00%	0.72	0.61	385.64%	5.34	0.58
FPGA3	13.18%	0.23	0.89	26.86%	0.46	0.79	48.43%	0.68	0.71	395.37%	6.74	0.59
FPGA4	12.75%	0.24	0.90	29.39%	0.48	0.82	40.22%	0.62	0.74	390.07%	7.58	0.62
FPGA5	13.12%	0.20	0.88	26.42%	0.45	0.81	41.83%	0.61	0.74	405.17%	6.86	0.60
FPGA6	12.74%	0.23	0.90	26.54%	0.47	0.81	43.09%	0.67	0.74	419.32%	6.96	0.64
FPGA7	12.78%	0.20	0.91	28.76%	0.48	0.79	42.59%	0.60	0.74	387.96%	6.99	0.65
FPGA8	13.49%	0.23	0.89	27.38%	0.46	0.79	42.28%	0.64	0.75	387.79%	5.85	0.52
Average	13.09%	0.22	0.89	27.34%	0.49	0.80	44.63%	0.66	0.72	405.41%	7.01	0.61

that all the three methods can correctly capture the design complexity. Meanwhile, ATAPP is more aligned with the ground-truth line with higher R value. It is expected that most of the green dots (XPE estimation) are beyond the ground-truth line since XPE tends to estimate worst-case power. Thirdly, XPE performs the worst among the three methods due to the poor estimation of switching activity. PowerGear can infer the switching activity from high-level code simulation which leads to better estimation than XPE, however, ATAPP performs better with the encoding of the FPGA architecture. Fourthly, ATAPP performance is quite stable across the experimental FPGAs: MAPEs are varying within [12.74%, 13.49%] (smaller is better) and RRSEs are within [0.20, 0.24] (smaller is better). The correlation metrics of ATAPP are around 0.90 (larger is better). Therefore, ATAPP has good generalizability.

Further experiments are conducted to study the effects of the positional encoding on the model performance. The ❺ **Positional Encoding** step in subsection 6.2.1 is removed and the compressed embeddings are used directly to train and test the model. Results are shown in the columns ATAPP w/o PE of Table 6.6. Compared to PowerGear and XPE, it shows that ATAPP is able to provide better performance with lower MAPE ranging within [26.42%, 29.39%], lower RRSE [0.45, 0.49] and higher R [0.79, 0.81] due to the introduction of FPGA architecture representation. However,

ATAPP w/ PE performs even better due to the feature embeddings with additional resource positions over the FPGA fabric.

6.5.2 Robustness study on clock period

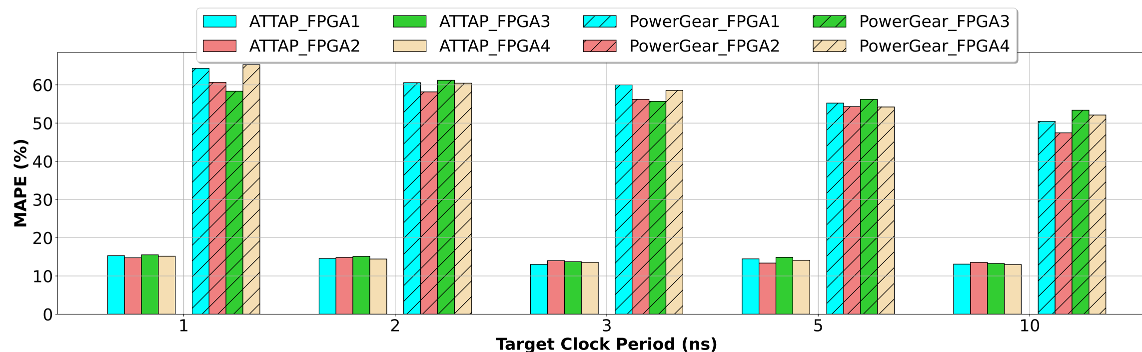


Figure 6.9: **ATAPP** is tested on designs with different target clock periods on different FPGAs. The training settings, **leave-one-out** strategy, are the same as shown in Table 6.6. **PowerGear** is tested on designs with different clock periods on different FPGAs. The model is trained with the designs on FPGA8.

Designs generated by HLS are greatly affected by the user-defined target clock period. Even with the same pragma and directive settings, the HLS tool can still generate different designs depending on the constraints, and one of the important constraints is the target clock period. In order to achieve the target clock period, HLS tools perform different operator scheduling and therefore generated designs are different. Further experiments are conducted to test the model sensitivity to the designs synthesized under different target clock period settings and results are shown in Figure 6.9.

ATAPP is trained with **leave-one-out** strategy to guarantee one FPGA is unseen to the model (all the designs on one FPGA are reserved and not used for training, 85% designs on the rest FPGA are used for training), 4 models (FPGA1-4) are generated independently. Every model is tested with 100 designs selected from the rest 15% designs in its reserved FPGA. Since most of the designs in the training are synthesized with the same target clock period (10ns), the 100 designs

are tuned with different target clock periods, the same pragmas/settings are applied but different target clock periods (1ns, 2ns, 3ns, 5ns, 10ns) are used to generate 500 designs (100 designs for each clock period) per FPGA. The 100 designs on each FPGA are used for testing the model sensitivity and results are shown in the up part of Figure 6.9. It is observed that ATAPP performs the best (∼13%) when making prediction for the designs with 10ns target clock period due to the training designs are under the same configuration, but when testing for the designs on the other target clock period, ATAPP can still maintain similar performance (13% - 15%) since the design representation used in ATAPP is generated at post-HLS stage and it covers sufficient information of the generated designs with different clock period.

In comparison, PowerGear is also tested with these designs and results are shown in the bottom part of Figure 6.9, the model is trained with 85% designs on FPGA8. PowerGear is not designed for predict power for unseen FPGAs, therefore, an obvious accuracy degradation compared to ATAPP is observed. There is a similar observation that the model performs the best on the designs synthesized with 10ns target clock period but maintain close MAPE for other designs because PowerGear takes achieved clock period from HLS reports as one of the features in the prediction.

6.5.3 Robustness study on generations of FPGAs

To demonstrate the help of ATAPP on development of FPGA with different generations, the model is trained without one particular generation of devices but tested with a device from that generation, for example, since FPGA 1-3 are from the same generation, they are split out from the training when ATAPP is test on those FPGAs. Results are shown in Table 6.7 and each row shows the specific devices used for training and testing ATAPP. There are several key observation from the results. First of all, ATAPP is able to achieve around 20% when predicting power on unseen generation of FPGAs and the performance is consistent across the tested FPGAs. Secondly, there is a performance degradation compared to training ATAPP with leave-one-out strategy although it is still better than the state-of-art ML-based

Table 6.7: Accuracy Comparison on **unseen devices** vs. **unseen generations**: **unseen devices**: the same generation devices with test device exist in training; **unseen generations**: the same generation devices with test device do not exist in training;
FPGA 1-3 from ultrascale+, FPGA 4-5 from ultrascale, FPGA 6-8 from 7series.

ATAPP Settings 1		Error for unseen devices			ATAPP Settings 2		Error for unseen generations		
Test	Training Set	MAPE	RRSE	R	Test	Training Set	MAPE	RRSE	R
FPGA1	FPGA 2-8	13.27%	0.22	0.89	FPGA1	FPGA 4-8	23.14%	0.38	0.84
FPGA2	FPGA 1, 3-8	13.35%	0.21	0.90	FPGA2	FPGA 4-8	22.70%	0.37	0.85
FPGA3	FPGA 1-2, 4-8	13.18%	0.23	0.89	FPGA3	FPGA 4-8	22.97%	0.38	0.84
FPGA4	FPGA 1-3, 5-8	12.75%	0.24	0.90	FPGA4	FPGA 1-3, 6-8	20.14%	0.34	0.85
FPGA5	FPGA 1-4, 6-8	13.12%	0.20	0.88	FPGA5	FPGA 1-3, 6-8	20.88%	0.35	0.86
FPGA6	FPGA 1-5, 7-8	12.74%	0.23	0.90	FPGA6	FPGA 1-5	20.99%	0.35	0.84
FPGA7	FPGA 1-6, 7	12.78%	0.20	0.91	FPGA7	FPGA 1-5	21.23%	0.37	0.84
FPGA8	FPGA 1-7	13.49%	0.23	0.89	FPGA8	FPGA 1-5	21.38%	0.36	0.84
Average		13.09%	0.22	0.89	Average		21.68%	0.36	0.85

power models. The degradation can be caused by less data being used for training (7 subsets vs. 4 - 5 subsets), therefore less knowledge on resource units or technology parameters are acquired by the model. Thirdly, the test results are FPGA 4-5 are slightly better than the rest due to the larger amount of designs used in the training. The experiment results show that ATAPP can be used for FPGA power evaluation even for different generations of FPGA devices and the performance of ATAPP on predicting for unseen generations can be improved with larger amount of data from other genrations of FPGAs.

6.6 Summary

ATAPP is an architecture and technology aware power predictor for unseen FPGA architecture and designs. The proposed method combines both FPGA architectural representation and design representation to predict average dynamic power. ATAPP extracts FPGA architectural features at the tile level to form embeddings and further augments these embeddings with a positional encoding mechanism to generate a better representation. The design representation is a graph generated from the IR data flow with FSMD model and operator switching activities at IR level. The experiments show that ATAPP provides an accurate estimate with 13.9% error

in unseen designs with unseen FPGA architectures. Robust studies on clock period are performed to ATAPP and results show that the model can maintain similar error with designs synthesized using different target clock periods.

Chapter 7: Conclusion

Power efficiency is becoming a more and more important factor for the system with Field-programmable gate arrays (FPGAs) as the compute density increases dramatically in the large language model era. Therefore, an efficient power evaluation method becomes essential during the exploration of power-efficient designs or FPGA architecture. To enable accurate and fast estimation of design-time power, researchers employed Machine Learning (ML) techniques to bypass the time-consuming phases including synthesis, implementation, and RTL simulation phases. However, these ML-based models no longer meet the requirement of power evaluation on varieties of FPGAs due to the inefficient dataset generation and inherent model drawbacks. To enhance the efficiency of constructing ML-based power models, the dissertation starts by efficient dataset generation, then move on to rapid model adaption across different FPGAs and ultimately develops a general FPGA architecture aware power model.

7.1 Summary

The dissertation makes three sequential contributions toward achieving an efficient power model for FPGAs. First, HLSDataset is developed. HLSDataset is a well-curated open-source dataset for ML-assisted FPGA design using HLS. The dataset generation is efficient with full automation and it can be easily extended with the provided scripts. The dataset can be used in power model development, resource estimation, performance prediction, etc. It has been verified with several case studies as well as this dissertation work. Currently it contains more than 40,000 designs covering 10 FPGAs and the number is still increasing.

Another contribution is XPNet, a cross FPGA power predictor from high level language code. XPNet creates a GNN-based power model that can predict

design-time dynamic power across FPGAs. Importantly, XPNet employs transfer-learning techniques to efficiently adapt existing models on source FPGAs to power prediction on target FPGAs. With the design selection method, XPNet only needs 20 designs to fine-tune the model and keep the sufficient accuracy in the meantime. XPNet demonstrates the ability to transfer power models across FPGAs from the same vendor as well as different vendors.

Finally, the dissertation contributes ATAPP architecture and technology aware power predictor for unseen FPGAs. Setting it apart from other existing ML-based power models, ATAPP bases its predictions on the representations of both the design and the FPGA architecture. The design is represented by a graph created from intermediate representation (IR) codes, the Finite State Machine Datapath (FSMD) model, and the toggling behavior of the operators. The architecture representation is formed using the architecture model from RapidWright, combined with positional encoding techniques. Notably, ATAPP demonstrates its capability to deliver accurate predictions (with an average error of 13.09%) for unseen designs on unseen FPGAs.

7.2 Future Work

There are many direction to enhance the dissertation work. Although HLS-Dataset is open-sourced and full-automated, it is not an end-to-end dataset generation framework. A more recent work HLSFactory [96] proposes a framework to generate dataset. It provides the facilities to collect and build custom HLS datasets using various frontends, supported HLS tools and data aggregation as well as provides built-in design dataset sources for users who want to run their own experiments out of the box. However, neither HLSDataset nor HLSFacotry provide a good support beyond AMD/Xilinx FPGA devices. It is necessary to extend the dataset to include FPGAs from varieties of vendors so that a more general study across FPGAs can be conducted. Moreover, while the dataset generation mainly relies on the commercial tools such as Vivado HLS or Intel HLS, it is even more beneficial to construct datasets for

academic tools such as VTR [92] since the tools are more transparent to the user and the version changes are less frequent.

Another direction to improve the abundance of datasets is to automate the generation of stimulus. The stimulus is used to cover wide range of power profile of a specific HLS designs. However, there is no efficient way to generate such stimulus for the generated RTL codes through the HLS even though C-RTL co-simulation helps to verify the functionality of the design. It is painful to write stimulus for every kernel codes and as the size of datasets grows, varieties of stimulus are necessary for every hardware design.

The current methodology to build ML model at HLS stage neglects the synthesis and implementation effects on power. The place and routing algorithms at implementation stage potentially affects the hardware power. While the current power model, with the assumption that place and routing is deterministic every time, provides good estimation on dynamic power, they can be further improved with the consideration of effects on place and routing. It is promising to extend the method to build model on FPGA HLS designs to ASIC deisgns. Different from FPGA developments, ASIC designers use basic components from a standard cell library to construct hardware circuits, and ASIC synthesis might be more complicated than FPGA synthesis due to the great flexibility in place and routing. It tends to be a challenging task to perform power prediction as early as FPGA HLS.

A more general FPGA power model can be developed with the even lower-level abstraction on FPGA architecture. ATAPP utilizes the tile-level architecture due to the similar circuit designs on basic elements among the AMD/Xilinx devices. Therefore, a great change on circuit level may lead to misleading prediction. It can be improved with extra features on circuit levels. Additionally, it will greatly improve the generalizability of the model if the FPGA architecture terminology can be unified across FPGAs from different vendors. One possible way is to implement the architecture using VTR format and with the datasets generated from VTR, researchers can

better explore the power effects of FPGAs across vendors.

Works Cited

[1] Habib Mehrez Umer Farooq, Zied Marrakchi. *FPGA architectures: An overview.* Springer, 2012.

[2] Zhe Lin. *Learning-based power modeling for FPGA : from design time to run time.* PhD thesis, Hong Kong University of Science and Technology, 2019.

[3] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Brian Gaide, and Ygal Arbel. Network-on-Chip Programmable Platform in VersalTM ACAP Architecture. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[4] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[5] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An Introduction to High Level Synthesis. In *IEEE Design Test of Computers*, 2009.

[6] Yunsheng Bai, Atefeh Sohrabizadeh, Zijian Ding, Rongjian Liang, Weikai Li, Ding Wang, Haoxing Ren, Yizhou Sun, and Jason Cong. Learning to Compare Hardware Designs for High-Level Synthesis. In *ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*, 2024.

[7] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2011.

[8] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology Systems(TRETS)*, 2022.

[9] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2011.

[10] T. Tuan and B. Lai. Leakage power analysis of a 90nm FPGA. In *IEEE Custom Integrated Circuits Conference (CICC)*, 2003.

[11] Deming Chen, Jason Cong, Yiping Fan, and Zhiru Zhang. High-Level Power Estimation and Low-Power Design Space Exploration for FPGAs. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2007.

[12] Tim Tuan, Arif Rahman, Satyaki Das, Steve Trimberger, and Sean Kao. A 90-nm Low-Power FPGA for Battery-Powered Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2007.

[13] Assem A. M. Bsoul, Steven J. E. Wilton, Kuen Hung Tsoi, and Wayne Luk. An FPGA Architecture and CAD Flow Supporting Dynamically Controlled Power Gating. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2016.

[14] Safeen Huda, Muntasir Mallick, and Jason H. Anderson. Clock gating architectures for FPGA power reduction. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2009.

[15] Y. Lin and L. He. Dual-vdd interconnect with chip-level time slack allocation for fpga power reduction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2006.

[16] G. Steiner and B. Philofsky. Managing power and performance with the Zynq UltraScale+ MPSoC. Technical report, AMD/Xilinx, 2016.

[17] Chin Hau Hoo, Yajun Ha, and Akash Kumar. A directional coarse-grained power gated FPGA switch box and power gating aware routing algorithm. In *2013 23rd International Conference on Field programmable Logic and Applications*, 2013.

[18] S. Kolluri. Ultrascale architecture low power technology overview. Technical report, AMD/Xilinx, 2015.

[19] Jose Luis Nunez-Yanez. Adaptive Voltage Scaling with In-Situ Detectors in Commercial FPGAs. *IEEE Transactions on Computers (TC)*, 2015.

[20] Jose Luis Nunez-Yanez, Mohammad Hosseinabady, and Arash Beldachi. Energy Optimization in Commercial FPGAs with Voltage, Frequency and Logic Scaling. *IEEE Transactions on Computers (TC)*, 2016.

[21] Achim Lösch, Tobias Beisel, Tobias Kenter, Christian Plessl, and Marco Platzner. Performance-centric scheduling with task migration for a heterogeneous compute node in the data center. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[22] Yi Lu, Thomas Marconi, Koen Bertels, and Georgi Gaydadjiev. A Communication Aware Online Task Scheduling Algorithm for FPGA-Based Partially Reconfigurable Systems. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2010.

[23] Wonyoung Kim, David Brooks, and Gu-Yeon Wei. A fully-integrated 3-level dc-dc converter for nanosecond-scale dvfs. *IEEE Journal of Solid-State Circuits (JSSC)*, 2012.

[24] Stijn Eyerman and Lieven Eeckhout. Fine-grained dvfs using on-chip regulators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2011.

[25] Paolo Mantovani, Emilio G. Cota, Kevin Tien, Christian Pilato, Giuseppe Di Guglielmo, Ken Shepard, and Luca P. Carlon. An FPGA-based infrastructure for fine-grained DVFS analysis in high-performance embedded systems. In *ACM/IEEE Design Automation Conference (DAC)*, 2016.

[26] Hadi Asghari-Moghaddam, Hamid Reza Ghasemi, Abhishek Arvind Sinkar, Indrani Paul, and Nam Sung Kim. VR-Scale: Runtime dynamic phase scaling of processor voltage regulators for improving power efficiency. In *ACM/IEEE Design Automation Conference (DAC)*, 2016.

[27] Hung-Yi Liu and Luca P. Carloni. On learning-based methods for design-space exploration with High-Level Synthesis. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013.

[28] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. Lattice-Traversing Design Space Exploration for High Level Synthesis. In *International Conference on Computer Design (ICCD)*, 2018.

[29] Guanwen Zhong, Alok Prakash, Siqi Wang, Yun Liang, Tulika Mitra, and Smail Niar. Design Space exploration of FPGA-based accelerators with multi-level parallelism. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[30] Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner. Adaptive Threshold Non-Pareto Elimination: Re-thinking machine learning for system level design space exploration on FPGAs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[31] Dong Liu and Benjamin Carrion Schafer. Efficient and reliable High-Level Synthesis Design Space Explorer for FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[32] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. Cluster-Based Heuristic for High Level Synthesis Design Space Exploration. *IEEE Transactions on Emerging Topics in Computing*, 2021.

[33] *Vivado design suite user guide: High level synthesis*. AMD/Xilinx, 2017.

[34] *Intel high level synthesis compiler: User guide*. Intel, 2019.

[35] Zhe Lin, Zike Yuan, Jieru Zhao, Wei Zhang, Hui Wang, and Yonghong Tian. PowerGear: Early-Stage Power Estimation in FPGA HLS via Heterogeneous Edge-Centric GNNs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022.

[36] Zhe Lin, Jieru Zhao, Sharad Sinha, and Wei Zhang. HL-Pow: A Learning-Based Power Modeling Framework for High-Level Synthesis. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2020.

[37] Dongwook Lee, Lizy K. John, and Andreas Gerstlauer. Dynamic Power and Performance Back-Annotation for Fast and Accurate Functional Hardware Simulation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.

[38] Zhigang Wei, Aman Arora, Ruihao Li, and Lizy John. HLSDataset: Open-Source Dataset for ML-Assisted FPGA Design using High Level Synthesis. In *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2023.

[39] Yunsheng Bai, Atefeh Sohrabizadeh, Zongyue Qin, Ziniu Hu, Yizhou Sun, and Jason Cong. Towards a comprehensive benchmark for high-level synthesis targeted to FPGAs. In *Thirty-Seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

[40] Pingakshya Goswami, Masoud Shahshahani, and Dinesh Bhatia. MLSBench: A Synthesizable Dataset of HLS Designs to Support ML Based Design Flows. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020.

[41] Lorenzo Ferretti, Jihye Kwon, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca Carloni, and Laura Pozzi. DB4HLS: A Database of High-Level Synthesis Design Space Explorations, 2021.

[42] Animesh Basak Chowdhury, Benjamin Tan, Ramesh Karri, and Siddharth Garg. OpenABC-D: A Large-Scale Dataset For Machine Learning Guided Integrated Circuit Synthesis, 2021.

[43] Zhuomin Chai, Yuxiang Zhao, Yibo Lin, Wei Liu, Runsheng Wang, and Ru Huang. Circuitnet: an open-source dataset for machine learning applications in electronic design automation (eda). *Science China Information Sciences*, 2022.

[44] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline FY Young, and Zhiru Zhang. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[45] Quentin Gautier, Alric Althoff, Pingfan Meng, and Ryan Kastner. Spector: An opencl fpga benchmark suite. In *International Conference on Field-Programmable Technology*, 2016.

[46] Benjamin Carrion Schafer and Anushree Mahapatra. S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis. *IEEE Embedded Systems Letters*, 2014.

[47] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. CHStone: A benchmark program suite for practical C-based high-level

synthesis. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2008.

[48] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[49] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, Xuefei Ning, Yuzhe Ma, Haoyu Yang, Bei Yu, Huazhong Yang, and Yu Wang. Machine learning for electronic design automation: A survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2021.

[50] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. Automated accelerator optimization aided by graph neural networks. In *ACM/IEEE Design Automation Conference (DAC)*, 2022.

[51] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. Robust GNN-based Representation Learning for HLS. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023.

[52] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. Accurate operation delay prediction for FPGA HLS using graph neural networks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2020.

[53] Nan Wu, Yuan Xie, and Cong Hao. IronMan-Pro: Multiobjective Design Space Exploration in HLS via Reinforcement Learning and Graph Neural Network-Based Modeling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2023.

[54] Kenneth O'Neal, Mitch Liu, Hans Tang, Amin Kalantar, Kennen DeRenard, and Philip Brisk. HLSPredict: cross platform performance prediction for FPGA high-level synthesis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

[55] Nan Wu, Hang Yang, Yuan Xie, Pan Li, and Cong Hao. High-Level Synthesis Performance Prediction Using GNNs: Benchmarking, Modeling, and Advancing. In *ACM/IEEE Design Automation Conference (DAC)*, 2022.

[56] 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 *International Conference on Field Programmable Logic and Applications (FPL)*, 2019.

[57] Dongwook Lee, Taemin Kim, Kyungtae Han, Yatin Hoskote, Lizy K. John, and Andreas Gerstlauer. Learning-based power modeling of system-level black-box IPs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015.

[58] Dongwook Lee and Andreas Gerstlauer. Learning-Based, Fine-Grain Power Modeling of System-Level Hardware IPs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2018.

[59] Cunxi Yu, Houping Xiao, and Giovanni De Micheli. Developing synthesis flows without human knowledge. In *ACM/IEEE Design Automation Conference (DAC)*, 2018.

[60] Walter Lau Neto, Max Austin, Scott Temple, Luca Amaru, Xifan Tang, and Pierre-Emmanuel Gaillardon. LSOracle: a Logic Synthesis Framework Driven by Artificial Intelligence: Invited Paper. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019.

[61] Mohamed Baker Alawieh, Wuxi Li, Yibo Lin, Love Singhal, Mahesh A. Iyer, and David Z. Pan. High-Definition Routing Congestion Prediction for Large-Scale FPGAs. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2020.

[62] Mingyang Kou, Jun Zeng, Boxiao Han, Fei Xu, Jiangyuan Gu, and Hailong Yao. GEML: GNN-based efficient mapping method for large loop applications on CGRA. In *ACM/IEEE Design Automation Conference (DAC)*, 2022.

[63] Zhiyao Xie, Yu-Hung Huang, Guan-Qi Fang, Haoxing Ren, Shao-Yun Fang, Yiran Chen, and Jiang Hu. RouteNet: Routability prediction for Mixed-Size Designs Using Convolutional Neural Network. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

[64] Chen-Chia Chang, Jingyu Pan, Tunhou Zhang, Zhiyao Xie, Jiang Hu, Weiyi Qi, Chun-Wei Lin, Rongjian Liang, Joydeep Mitra, Elias Fallon, and Yiran Chen. Automatic Routability Predictor Development Using Neural Architecture Search. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021.

[65] Yu-Hung Huang, Zhiyao Xie, Guan-Qi Fang, Tao-Chun Yu, Haoxing Ren, Shao-Yun Fang, Yiran Chen, and Jiang Hu. Routability-Driven Macro Placement with Embedded CNN-Based Prediction Model. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.

[66] Hosein Mohammadi Makrani, Hossein Sayadi, Tinoosh Mohsenin, Setareh rafatirad, Avesta Sasan, and Houman Homayoun. XPPE: cross-platform performance estimation of hardware accelerators using machine learning. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2019.

[67] Ryan Gary Kim, Janardhan Rao Doppa, and Partha Pratim Pande. Machine Learning for Design Space Exploration and Optimization of Manycore Systems.

In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

[68] Zi Wang and Benjamin Carrion Schafer. Machine Leaming to Set Meta-Heuristic Specific Parameters for High-Level Synthesis Design Space Exploration. In *ACM/IEEE Design Automation Conference (DAC)*, 2020.

[69] Jianwang Zhai, Chen Bai, Binwu Zhu, Yici Cai, Qiang Zhou, and Bei Yu. McPAT-Calib: A Microarchitecture Power Modeling Framework for Modern CPUs. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021.

[70] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2014.

[71] Qijun Zhang, Shiyu Li, Guanglei Zhou, Jingyu Pan, Chen-Chia Chang, Yiran Chen, and Zhiyao Xie. PANDA: Architecture-Level Power Evaluation by Unifying Analytical and Machine Learning Solutions. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023.

[72] Ajay Krishna Ananda Kumar, Sami Al-Salamin, Hussam Amrouch, and Andreas Gerstlauer. Machine learning-based microarchitecture- level power modeling of cpus. *IEEE Transactions on Computers*, 2023.

[73] Ajay Krishna Ananda Kumar and Andreas Gerstlauer. Learning-Based CPU Power Modeling. In *ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*, 2019.

[74] Zhiyao Xie, Xiaoqing Xu, Matt Walker, Joshua Knebel, Kumaraguru Palaniswamy, Nicolas Hebert, Jiang Hu, Huanrui Yang, Yiran Chen, and Shidhartha Das.

APOLLO: An Automated Power Modeling Framework for Runtime Power Introspection in High-Volume Commercial Microprocessors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[75] Yuan Zhou, Haoxing Ren, Yanqing Zhang, Ben Keller, Brucek Khailany, and Zhiru Zhang. PRIMAL: Power Inference using Machine Learning. In *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[76] Zhiyao Xie, Shiyu Li, Mingyuan Ma, Chen-Chia Chang, Jingyu Pan, Yiran Chen, and Jiang Hu. DEEP: Developing Extremely Efficient Runtime On-Chip Power Meters. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2022.

[77] Jianlei Yang, Liwei Ma, Kang Zhao, Yici Cai, and Tin-Fook Ngai. Early stage real-time SoC power estimation using RTL instrumentation. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2015.

[78] Wenji Fang, Yao Lu, Shang Liu, Qijun Zhang, Ceyu Xu, Lisa Wu Wills, Hongce Zhang, and Zhiyao Xie. MasterRTL: A Pre-Synthesis PPA Estimation Framework for Any RTL Design. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023.

[79] Prianka Sengupta, Aakash Tyagi, Yiran Chen, and Jiang Hu. How Good Is Your Verilog RTL Code? A Quick Answer from Machine Learning. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2022.

[80] Ceyu Xu, Chris Kjellqvist, and Lisa Wu Wills. SNS's not a synthesizer: a deep-learning-based synthesis predictor. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2022.

[81] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. GRANNITE: Graph Neural Network Inference for Transferable Power Estimation. In *ACM/IEEE Design Automation Conference (DAC)*, 2020.

[82] W. Rhett Davis, Paul Franzon, Luis Francisco, Billy Huggins, and Rajeev Jain. Fast and Accurate PPA Modeling with Transfer Learning. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021.

[83] Jihye Kwon and Luca P. Carloni. Transfer Learning for Design-Space Exploration with High-Level Synthesis. In *ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, 2020.

[84] Dongwook Lee. *Learning-Based System-Level Power Modeling of Hardware IPs*. PhD thesis, The University of Texas at Austin, 2017.

[85] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[86] L.-N. Pouchet. Polybench: The polyhedral benchmark suite, 2012. URL `http://web.cs.ucla.edu/~pouchet/software/polybench/`.

[87] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018.

[88] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. OverGen: Improving FPGA Usability through Domain-specific Overlay Generation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[89] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[90] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[91] AMD Power Estimator (XPE), 2024. URL `https://www.amd.com/en/products/adaptive-socs-and-fpgas/technologies/power-efficiency/power-estimator.html`.

[92] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jai Min Wang, Mohamed El-Dafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. VTR 8: High Performance CAD and Customizable FPGA Architecture Modelling. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2020.

[93] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjing Wang, and Yu Sun. Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2021.

[94] Chris Lavin and Alireza Kaviani. RapidWright: Enabling Custom Crafted Implementations for FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[95] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *International Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[96] Stefan Abi-Karam, Rishov Sarkar, Allison Seigler, Sean Lowe, Zhigang Wei, Hanqiu Chen, Nanditha Rao, Lizy John, Aman Arora, and Cong Hao. HLSFactory: A Framework Empowering High-Level Synthesis Datasets for Machine Learning

and Beyond. In *ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*, 2024.